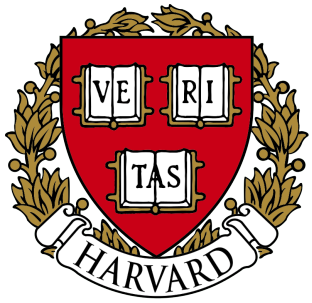
Data-driven Model Discovery
Final Report

AM205

Yongchao Chen, Jean-Guillaume Brasier, Nicolas Dhers



Institute for Applied Computational Science
Harvard University
December 18, 2021

Contents

1 Introduction

1.1 Context

Governing equations are defined as mathematical models that describe any biological or physical system. They define the relationship between known variables and unknown dependent variables. They help us better understand the intricacies of the world around us and allow us to solve for specific states. They are usually derived from first principles or universal laws. In solid mechanics, governing equations include the equations of motion relating body forces to stress, displacement, strain and stress relations. Fluid mechanics study conservation of mass, momentum and energy and the Navier-Stokes equations. In economics, governing equations refer to many differential equations that model concepts such as bifurcations and chaos. We note that most of these equations are differential equations and depend on changes in the temporal and/or spatial dimensions.

However, in some cases, we have not been able to model accurately a specific phenomenon. For many systems, the corresponding governing equations remain unknown. This is particularly the case in fields such as climate change science, finance, and epidemiology. With the improvement of data collection methods and tools, these systems can still be represented by large amounts of data. In these times of pandemic, we are constantly overflowed by data on the number of new COVID-19 cases, the number of deaths, the outbreak of a particular variant somewhere around the planet. In this project, we would like to study the extraction of governing equations coefficients through machine learning techniques. More specifically, we will want to look at systems where large amounts of data are available, therefore enabling us to learn unknown parameters from said data. Finding reliable ways to discover governing equations coefficients from data would dramatically change our approach to model systems. This would help scientific innovation in many different fields where data can be easily accessible and queried.

Machine learning techniques have made tremendous progress in the past decades. Our ability to understand and model the data has greatly improved and is a topic of active research. Furthermore, computational technologies like supercomputer clusters as well as computational tools like Spark have allowed us to extract, process and analyze larger amounts of data. Nonetheless, little progress has been made in the field of data-driven discovery of governing equations. This is due to the complexity of the task at hand and our limited ability to extrapolate results to the actual underlying dynamics of the system. The complexity also largely stems from the fact that we often face noisy and corrupt data as well as latent variables and multivariate and multiscale physics.

Historically, data measurements have often been at the basis of the discovery of governing equations. Using extensive planetary data, Kepler published his laws of planetary motion and the renown elliptic orbits theory in a heliocentric coordinate system. He was able to describe planetary orbits with an accuracy never reached before. While his model did not fully explain and describe the dynamics at the origin of planetary orbits, it paved the way

for the development of Newton's famous $F = ma$, describing the relation between the sum of forces and the acceleration of an object, thereby generalizing Kepler's results to cases where no data was available. Newton created a universal and interpretable model for physical dynamics (Silva et al. 2020). Similarly, we will try to infer physical relations from data available to us. In order to prove the accuracy of the methods used, we will generate synthetic data from known governing equations and we will try to obtain results that replicate the original equations.

1.2 Approach

One technique has been to use Sparse Regression of Nonlinear Dynamical Systems (SINDy) in order to identify the main features of a governing equations as well as the corresponding coefficients. The method was introduced in Brunton, Proctor, and Kutz 2016; SINDy is based on defining the relationship between the time derivative of a state variable $x(t)$ taken from any physical system of interest and a learnable function $f(x(t))$ such as:

$$x'(t) = f(x(t)) \tag{1}$$

SINDy involves learning the underlying parameters of the function f, which takes us from the state to its derivative with respect to time. Hence, each component of $f(x(t))$ can be represented as a sparse linear combination of basis functions $\theta_j(x)$:

$$f_i(x) = \theta_1(x) * xi_{1,i} + \theta_2(x) * xi_{2,i} + ... + \theta_k * xi_{k,i} \tag{2}$$

This boils down to solving the following regression problem:

$$X' = Theta(X) * Xi. \tag{3}$$

Where the capital letters designate matrices. Here, each row corresponds to one coordinate function of $f(x)$. In sparse regression, we start off with many different possibilities for our $f(x(t))$ that include many different function types and use a regularization technique that shrinks parameters to predictors that are less important for the regression problem.

However, SINDy relies on computing higher order derivatives in multiple dimensions (including space and time), limiting model discovery to densely sampled data-sets with low noise. Data for unknown systems generally suffer from both noise and sparsity, which therefore increase the error propagated through the higher-dimensional terms, especially when using numerical methods such as spline interpolation or simple finite-difference schemes (Both, Vermarien, and Kusters 2021). An alternative, especially for noisy data is to use Deep Learning for Model Discovery (DeepMoD) (Both, Choudhury, et al. 2021). It consists in building a surrogate for the data through a Neural Network, which is able to construct the function library and the time derivative of our state variable through automatic differentiation. It then uses sparse regression to select the most relevant predictors. This method significantly improves the accuracy of both the time derivative of the state variable and the library in

cases of noisy and sparse data. However DeepMod is purposely build for 1D or 2D systems, and the way the proposal library is constructed, allows little flexibility for higher dimensions.

In this project, we solved multiple systems of differential equations using both a SINDy approach (i.e. solely using sparse regression) and a DeepMod approach (i.e. Neural Network data surrogate and sparse regression). We started off by solving the solutions through Python ODE and IVP built-in routines and given chosen parameters and initial conditions. We 'noisified' our synthetically generated data and fed the data to the Sparse regressor algorithm, which produced the best parameter estimates. We will go further in details in the intricacies of the two approaches in the Methods section of this report. We finally compared our results and the two different methods' performances.

To do so, we looked at coupled ODEs of increasing difficulty. We first started off with a 2-Dimensional system of ordinary differential equations consisting of the vibration of a linear, damped harmonic oscillator with no external loading. The damping coefficients act on the velocity components (i.e. the temporal derivatives) while the spring constants act on the displacement values. The system is given by:

$$\begin{aligned} \frac{dx}{dt} &= -0.1x + 2y \\ \frac{dy}{dt} &= -2x - 0.1y \end{aligned} \tag{4}$$

We used an initial condition of $[x_0, y_0] = [1, 0]$.

We then looked at the predator-prey model, also known as the Lotka-Volterra equations. They are a pair of first-order nonlinear differential equations, often used to describe the dynamics in biological systems where a predator species interacts with a prey species. The equations are given as:

$$\begin{aligned} \frac{dx}{dt} &= \alpha x - \beta xy \\ \frac{dy}{dt} &= \delta xy - \gamma y \end{aligned} \tag{5}$$

In our example, we picked the following values for the above parameters: $(\alpha, \beta, \delta, \gamma) = (1, 0.5, 0.5, 2)$.

Finally we evaluated the 3D Lorenz System. It is a system of non-linear ordinary differential equations, known for having chaotic solutions for certain parameter values and initial conditions. In our case, we looked at the following example:

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x) \\ \frac{dy}{dt} &= x(\rho - z) - y \\ \frac{dz}{dt} &= xy - \beta z \end{aligned} \tag{6}$$

With $(\sigma, \rho, \beta) = (10, 28, \frac{8}{3})$ and initial condition $[x_0, y_0, z_0] = [-8, 8, 27]$.

1.3 Related Work

While we have oriented our research towards understanding the available tools that allow us to find underlying dynamical equations from data points, it is not one but many of the machine learning initiatives in finding expressions of unknown functions from data. Another related major problem in physical sciences has been symbolic regression: finding the symbolic expression of a unknown function from data. More specifically, for a given set of datapoints $\{x_1, x_2, ..., x_n, y\}$ such that $y = f(x_1, x_2, ..., x_n)$, the task is to find the symbolic expression of f. This is the motive behind AI Feynman (Udrescu and Tegmark 2020), a physics inspired symbolic regression framework.

2 Methods

2.1 SINDy

2.1.1 Theory

This section gives some insight on the paper by Brunton, Proctor, and Kutz 2016.

If we consider a non-linear or linear dynamical system for a state vector $x(t)$ as defined by (1) as well as a set of measurements of $x(t)$, we can reconstruct $f(x(t))$ through sparse regression using SINDy. For most of the dynamical systems out there, we can narrow down the vector function $f(x(t))$ to a small number of terms. Hence, we will try to provide a rich enough collection of candidate functions for simulating $f(x(t))$ and later identify the correct terms of the original governing equation through sparse regression.

To do so, we first need to find a set of datapoints from the dynamical system of study. These points will form our $x(t)$ vector. We also need the derivative of these points with respect to time at predefined time intervals. These will form our $\dot{x}(t)$ vector. The derivatives are either computed directly or through numerical approximations like finite difference approaches. These data are then aggregated into two matices X and $\dot{X}$ such that:

$$X = \begin{bmatrix} x_1(t_1) & x_2(t_1) & \cdots & x_n(t_1) \\ x_1(t_2) & x_2(t_2) & \cdots & x_n(t_2) \\ \vdots & \vdots & & \vdots \\ x_1(t_m) & x_2(t_m) & \cdots & x_n(t_m) \end{bmatrix}$$
$$\dot{X} = \begin{bmatrix} \dot{x_1}(t_1) & \dot{x_2}(t_1) & \cdots & \dot{x_n}(t_1) \\ \dot{x_1}(t_2) & \dot{x_2}(t_2) & \cdots & \dot{x_n}(t_2) \\ \vdots & \vdots & & \vdots \\ \dot{x_1}(t_m) & \dot{x_2}(t_m) & \cdots & \dot{x_n}(t_m) \end{bmatrix} \tag{7}$$

Next, we specify a set of candidate functions, such that: $\phi_i(x) : i = 1, 2, 3, ..., p$, which will be used in order to simulate $f(x(t))$. These candidate functions can be trigonometric functions, monomial functions up to a certain degree or rational functions. This is where it is useful to have some prior knowledge on the dynamical system, since we can limit the number of candidate functions we train over. These functions form the candidate matrix $\Theta(X)$ whose columns are the set of basis functions chosen:

$$\Theta(X) = \begin{bmatrix} \theta_1(X) & \theta_2(X) & \cdots \theta_p(X) \end{bmatrix} \tag{8}$$

We then assume that only a small portion of the above functions will be needed to represent $f(x(t))$, i.e. only a sparse linear combination of such functions. We therefore seek a set of sparse coefficient vectors Ξ, such that:

$$\Xi = \begin{bmatrix} \xi_1 & \xi_2 & \cdots & \xi_n \end{bmatrix} \tag{9}$$

To find the above, we use Sparse Regression and Sequentially Thresholded Least-Squares in order to balance accurate representation of the data and zeroing out of less important predictor values.

From (9), the vectors ξ_i give the coefficients for a linear combination of the basis functions $\theta_i(X)$ representing the ith component of the function $f(x(t))$, i.e. $f_i(x(t))$. Hence, we can write: $f_i(x) = \Theta(x)\xi_i$, where x is different from the numerical datapoints X and represents symbolic variables used in our candidate functions. Consequently, we can rewrite (1) with our identified candidate functions as such:

$$\dot{X} \approx \Theta(X)\Xi \tag{10}$$

The whole process is clearly illustrated in the original SINDy paper (see Figure 1) with the example of the Lorenz system, which we will later look at in our Results section**??**.

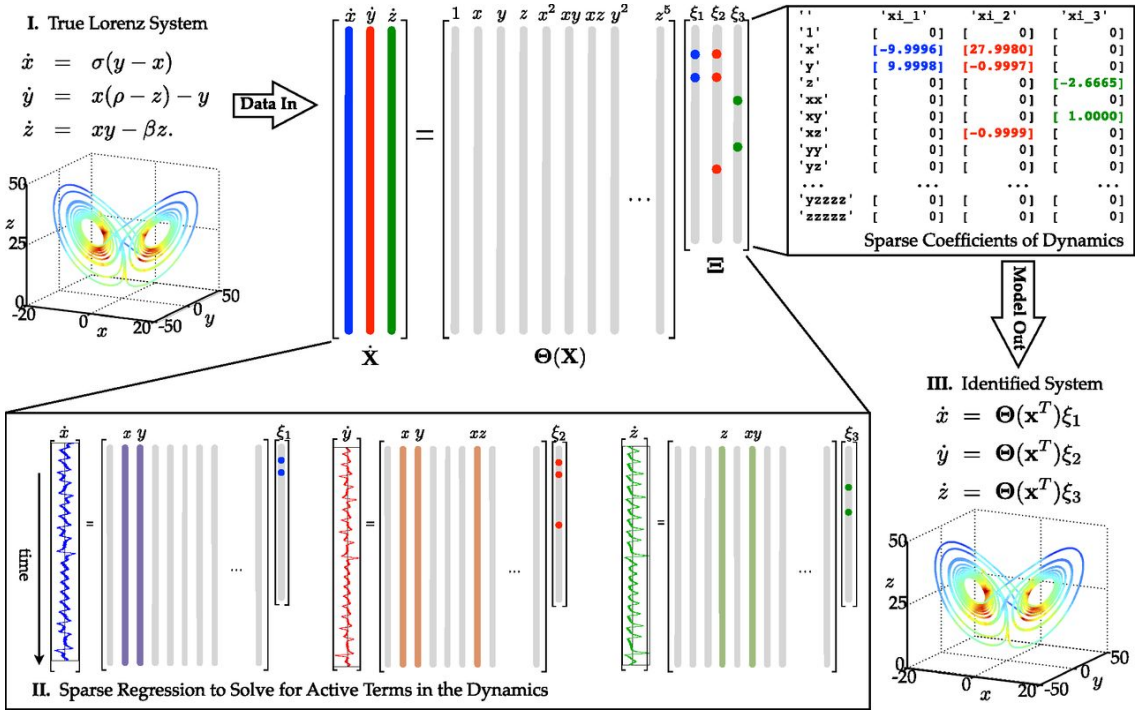


Figure 1: SINDy with Lorenz System

2.1.2 Implementation

Please refer to *simpleode.py, lorenz.py, lotka.py* in the SINDy folder for corresponding codes.

To implement SINDy, we made use of the Python API package PySINDy, which is freely available online. Its dependencies can easily be installed through pip on a local machine. A nice and clear html interface for their documentation is also available online. Despite

tuning much of their example code for our own purpose, some parts of the code have been reused from the documentation in order to implement the package's methods as efficiently as possible[1]. Ideas for looking at the simple vibration case as well as the Lorenz system of equation have been taken from the original SINDy paper published in 2016.**??**

We followed the same steps for all three of our example cases: vibration, Lorenz and Lotka-Volterra.

We first define a range of time as well as a time interval for our dynamical problem. We then use scipy's solve_ivp function to solve our system of differential equations given initial conditions as well as time range. We then select the candidate library from the feature_library submodule of the SINDy object. In our case, we simply look at monomial functions up to a certain degree.; this decision was made based on prior knowledge about the system. For model selection, we then build a SINDy object based on the feature library chosen as well as on a threshold. The threshold is used in the Sequentially Thresholded Least-Squares (STLSQ) algorithm that seeks to find the model that represents the data the best while zeroing out small entries ξ that are below the threshold. Before fitting our SINDy object to our data, we need to compute the $\dot{X}$ matrix. To do so, we simple calculate it from our functions. However, note that for non-synthetic data, this step would require the use of a finite difference scheme. We then loop over different noise levels in order to mimic the process of data-driven discovery with limited prior knowledge of the system. This also helps us determine how robust our identified model is with increasing noisiness. This is done by setting the threshold to be constant and equal to a constant small value. Finally, we loop over a set of threshold values in order to determine what the optimal value would be if we were to limit the error between our simulated data and our true training data. To do so, we keep our noise level to a constant small value.

2.2 DeepMod

2.2.1 Theory

This section gives some insight on the paper by Both, Vermarien, and Kusters 2021.

The goal of DeepMod is to discover the underlying partial differential equations (PDEs) of a set of datapoints $u(x,t)$. The problem can be written as:

$$\partial_t u(x,t) = \mathcal{F}\left(u, u_x, uu_x, u_{xx}, \ldots\right) \tag{11}$$

Where $\mathcal{F}$ is the unknown function to discover. To do so, a large set of models are generated by considering all the permutations of a library of candidate terms. This library, like SINDy, depends on the the nature of the problem. In most cases it involves a polynomial basis of functions and their spatial derivatives up to a certain order. Most of the the phenomenons encountered in nature can be modeled with a set of PDEs with terms of at most a second

[1] https://pysindy.readthedocs.io/en/latest/examples/2_introduction_to_sindy.html

order in time and second order in space. Computing all the possible models resulting from combinations of u, $\frac{\partial u}{\partial x}$, and $\frac{\partial^2 u}{\partial x^2}$ is unfeasible. Therefore a sparse regression like SINDy is prefered here, the problem thus becomes:

$$\partial_t u = \Theta \xi \tag{12}$$

where $\partial_t u$ is a column vector of size N containing the time derivative of each sample and Θ contains all M possible terms for each of the samples, so that we can write it as

$$\Theta = \begin{bmatrix} 1 & u\left(\{x,t\}_0\right) & u_x\left(\{x,t\}_0\right) & \dots & u^2 u_{xx}\left(\{x,t\}_0\right) \\ 1 & u\left(\{x,t\}_1\right) & u_x\left(\{x,t\}_1\right) & \dots & u^2 u_{xx}\left(\{x,t\}_1\right) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & u\left(\{x,t\}_N\right) & u_x\left(\{x,t\}_N\right) & \dots & u^2 u_{xx}\left(\{x,t\}_N\right) \end{bmatrix}$$

The problem becomes analogous to (10). Most of the terms in Θ will not contribute to $\partial_t u$, effectively rendering ξ sparse. Sparsity is achieved in this case by using LASSO (L1-penalization) to zero out terms, and select only the most relevant. The particularity of DeepMod is to combine the sparse regression with a feed-forward neural network which is a function approximate $\hat{u}(x,t)$ of $u(x,t)$. Neural networks are a powerful tool because:

1. They approximate nonlinear functions well, without needing to linearize.
2. They are completely differentiable.

To train the neural network, the loss $\mathcal{L}$ is minimized using standard optimization schemes readily implemented in PyTorch (Paszke et al. 2019):

$$\mathcal{L} = \mathcal{L}_{MSE} + \mathcal{L}_{Reg} + \mathcal{L}_{L_1} \tag{13}$$

with:

$$\mathcal{L}_{MSE} = \frac{1}{N}\sum_{i=1}^{N} \left|u\left(\{x,t\}_i\right) - \hat{u}_i\right|^2, \quad \mathcal{L}_{Reg} = \frac{1}{N}\sum_{i=1}^{N} \left|\Theta_{ij}\xi_j - \partial_t \hat{u}_i\right|^2, \quad \mathcal{L}_{L_1} = \lambda \sum_{i=2}^{M} \left|\xi^i\right| \tag{14}$$

The trick here is to update the selection vector ξ with the parameters of the neural network. This is done by adding the regularization loss $\mathcal{L}_{Reg}$. To minimize error, automatic differentiation is used to compute the coefficients of Θ, which come from differentiating the output $\hat{u}$ of the updated neural network. Even with L1 regularization, some terms will most likely not be zeroed out in ξ, a threshold is applied to keep only terms with highest weight.

DeepMod therefore builds on SINDy, and proposes a more blackbox approach to model discovery. Indeed, DeepMod does not require the user to compute time derivatives, as it is done with automatic differentiation within the framework.

A schematic of the DeepMod framework is given in Figure 2.

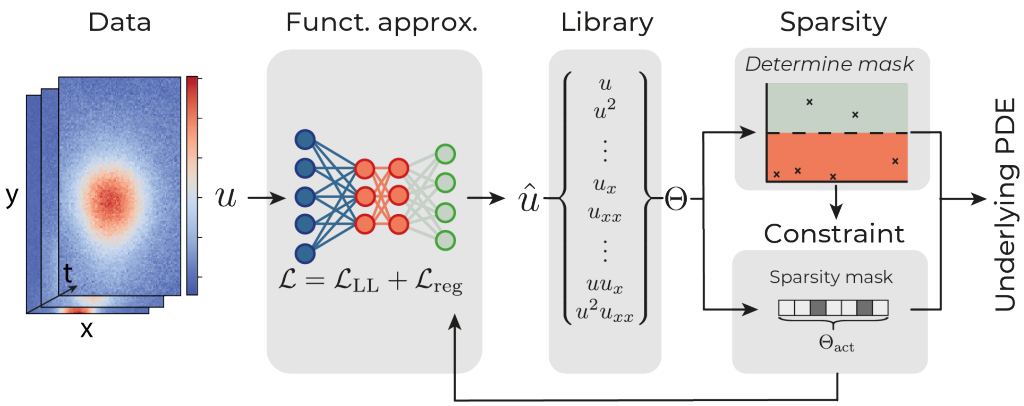


Figure 2: DeepMod framework with 2D advection diffusion data as input. The sparsity mask refers to the selection vectorξ. $\mathcal{L}_{LL} = \mathcal{L}_{MSE} + \mathcal{L}_{L1}$

2.2.2 Implementation

Please refer to *simpleode.py, lorenz.py, lotka_volterra.py* in the DeepMod folder for corresponding codes.

To implement DeepMod, we made use of the Python API package DeePyMod, freely available on Github and easily installable on a local machine through pip. Some documentation is provided through hosted html, although most of it is either out of date, or the exact solutions are not reproducible. Most of the the code was adapted directly from the package's source code. Custom libraries and training loops were rewritten for our own purpose. [2].

Similarly to the SINDy framework, to generate some synthetic data, we first define a range of time as well as a time interval for our dynamical problem. We then use scipy's `solve_ivp` function to solve our system of differential equations given initial conditions as well as time range. Noise of various levels is added to our synthetic data to mimic experimental measures and limited prior knowledge of the system. These points are then randomly sampled to form the input vector. All our dynamical problems are systems of coupled PDEs, where each time derivative requires a specific sparsity mask. Therefore we need to build custom libraries for each problem. These libraries inherit the properties of the standard DeepMod library, but give us the manual control over how many sparsity masks we want to output. In our case, the neural network surrogate takes a time vector in input, and outputs a N-dimensional vector, where N is the number of variables in each system (2 for the damped harmonic oscillator and lotka-volterra models and 3 for the lorenz system). Thus our custom library outputs N sparsity masks instead of the standard 1 in the original library. Our library considers all permutations of variables and their interactions. In 2D this would be C (coefficient), x, y

[2]https://phimal.github.io/DeePyMoD/

and xy.

To train our model, we use ADAM (Kingma and Ba 2017) with a learning rate of 10^{-4}, and a constant tuned threshold. We use the following noise levels: $0.001, 0.01, 0.1, 0.5, 1.0$. For each level, the Root Mean Squared Error (RMSE) between the model output and the scipy `solve_ivp` result, is computed. The framework uses a sparsity scheduler, which is a sort of timer that periodically applies the sparsity mask to the coefficients. The sparsity estimator is non-differentiable, and determining the sparsity mask before the neural network has reasonably approximated the data can negatively affect training. In the custom training loop, data is split into a train and test-set and the sparsity mask only updates when the MSE on the test-set starts to increase. Training is stopped when the maximum number of iterations has been reached or when the L1-regularization coefficient λ does not vary from one iteration to the other by 10^{-5}.

3 Results

3.1 SINDy

By running the corresponding code files, you will be able to reconstruct the governing equations (4), (5) and (6) introduced in the first section of this report. In this section, we look at reconstructed data, i.e. data that was generated based on the recreated governing equation. This allows for easy comparison with our original dataset.

3.1.1 2D Linear Damped Harmonic Oscillator

First, we note that we were able to reconstruct the original governing equation within some error for 2000 points.

We first compare our reconstructed datapoints with our synthetic datapoints which we can consider as the ground truth. Figure 3 demonstrates the similarity between both sets of data. Note that we have chosen the smallest noise level (i.e. a value of 10^{-4}) as well as a larger value (i.e. 0.1) for the reconstructed datasets; these values corresponds to the standard deviations chosen for the normal distributions we are sampling from. Note that Figure 3 shows the trajectories as t goes from 0 to 20.

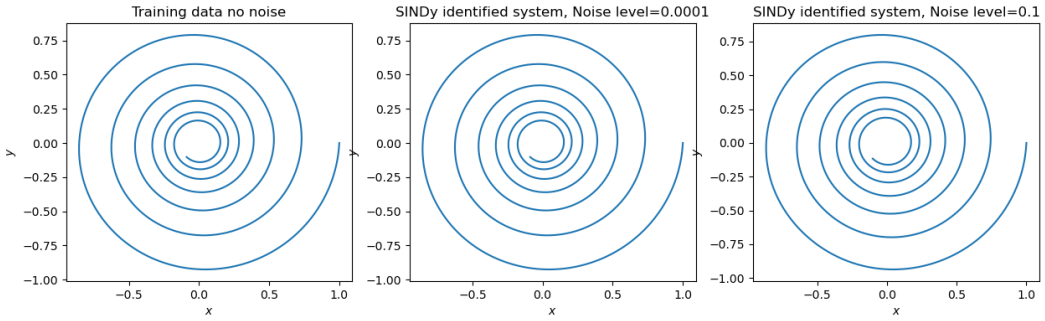


Figure 3: 2D Oscillator model: Trajectory comparison with two noise levels

Next, Figure 4 shows a clearer picture of the evolution of both y and x with respect to time. We can also clearly see that, as expected, a lower noise level has allowed us to more accurately simulate the data.

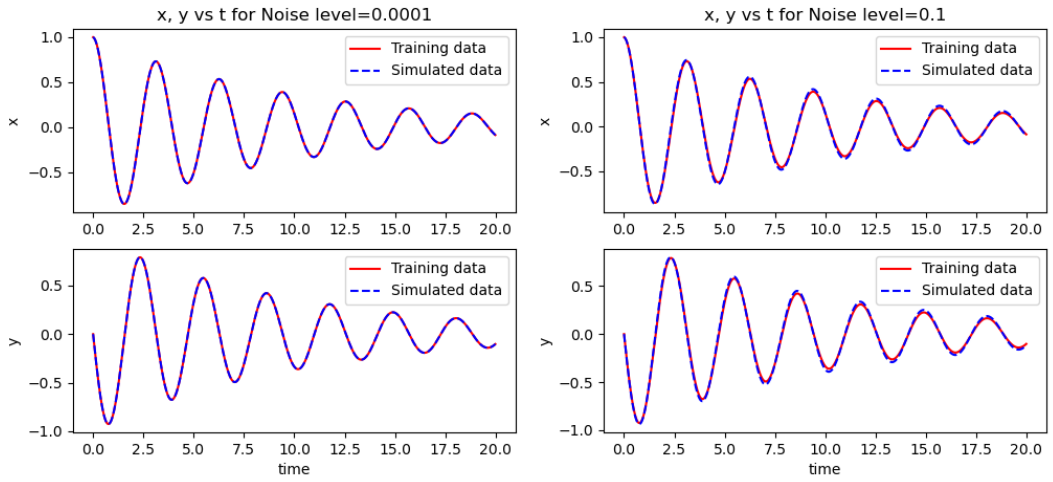
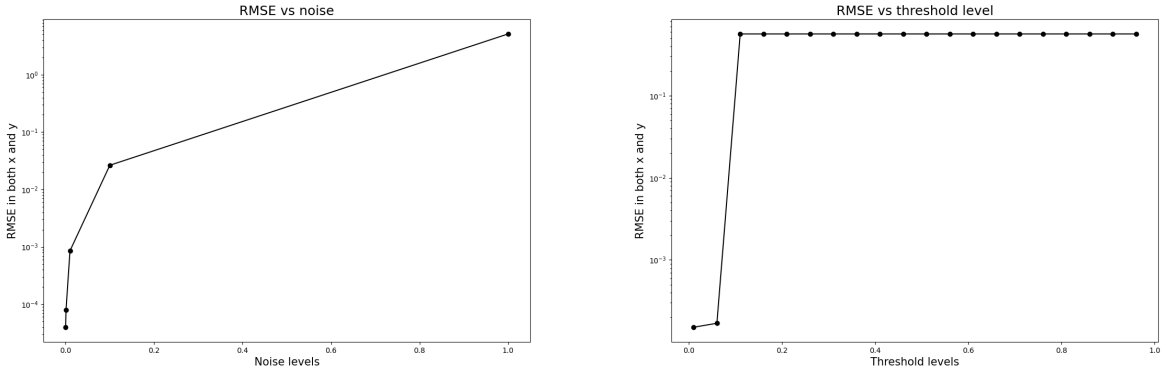


Figure 4: 2D Oscillator model: Comparison of x and y with respect to time between reconstructed (simulated) and original data (training data)

Finally, we also showed that increasing noise level as well as increasing threshold level both lead to higher errors. Note that we computed the RMSE by taking the Root Mean Squared Error of both X and Y components. Figures 5a and 5b tell us that an optimal choice of parameters would be a threshold of 0.01 and noise level of 10^{-4}. However, it is important not to overfit to the data, and we would need to compare performance on a hold-out test set in order to accurately find the optimal set of parameters that generalizes best. Note that we are able to reach quite low levels of error on this training set.



(a) RMSE vs noise (b) RMSE vs threshold value

Figure 5: 2D Oscillator model: RMSE vs hyperparameters

3.1.2 Predator-Prey Model

First, we note that we were able to reconstruct the original governing equation within some error for 100,000 points

Figure 6 demonstrates the similarity between our reconstructed datapoints and our synthetic datapoints. Note that we stuck with our previous values for the threshold and noise level values for visualization purpose.

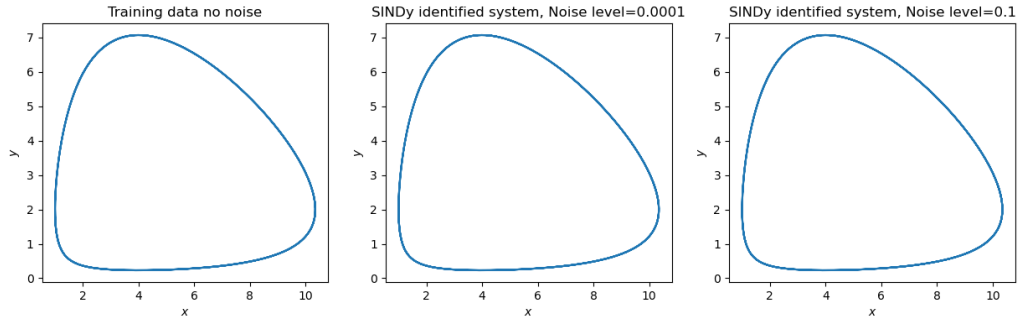


Figure 6: Lotka-Volterra model: Trajectory comparison with two noise levels

Next, Figure 7 shows a clearer picture of the evolution of both y and x with respect to time. We can also clearly see that, as expected, a lower noise level has allowed us to more accurately simulate the data.

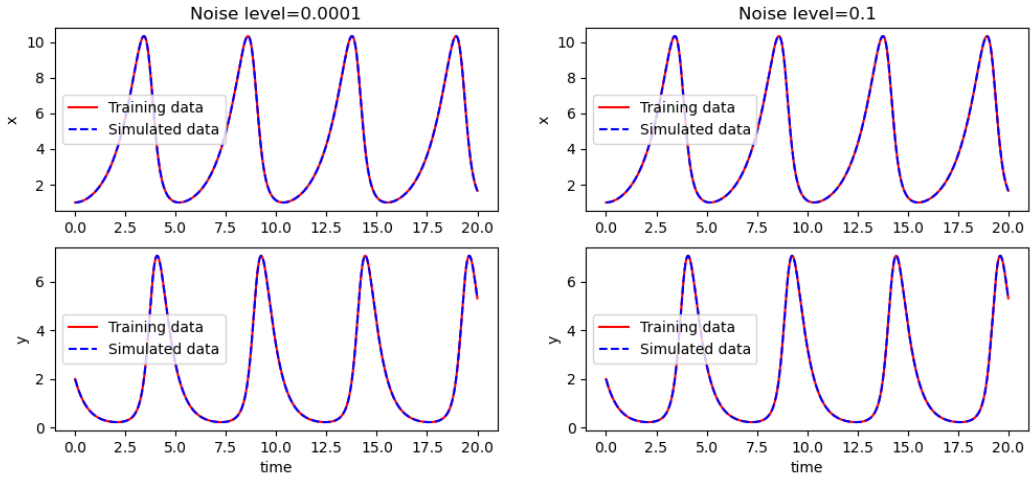
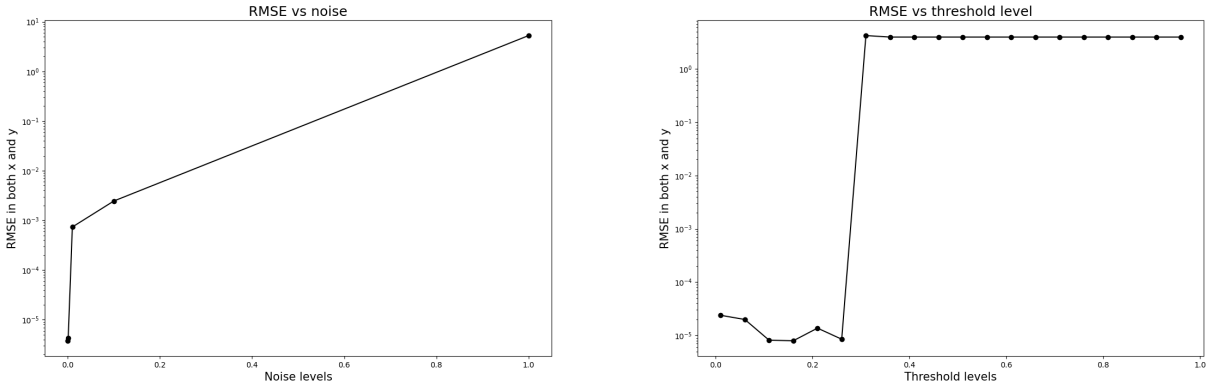


Figure 7: Lotka-Volterra model: Comparison of x and y with respect to time between reconstructed (simulated) and original data (training data)

Finally, we also showed that increasing noise level as well as increasing threshold level both lead to higher errors. Note that we computed the RMSE similarly as before, by taking the Root Mean Squared Error of both X and Y components. Figures 8a and 8a tell us that an optimal choice of parameters would be a threshold of 0.16 and noise level of 10^{-4}. However, it is important not to overfit to the data, and we would need to compare performance on a hold-out test set in order to accurately find the optimal set of parameters that generalizes best. Note that we are able to reach quite low levels of error on this training set.



(a) RMSE vs noise

(b) RMSE vs threshold value

Figure 8: Lotka-Volterra model: RMSE vs hyperparameters

3.1.3 Lorenz System

First, we note that we were able to reconstruct the original governing equation within some error for 100,000 points

Figure 9 demonstrates the similarity between our reconstructed datapoints and our synthetic datapoints. Note that we stuck with our previous values for the threshold and noise level values for visualization purpose.

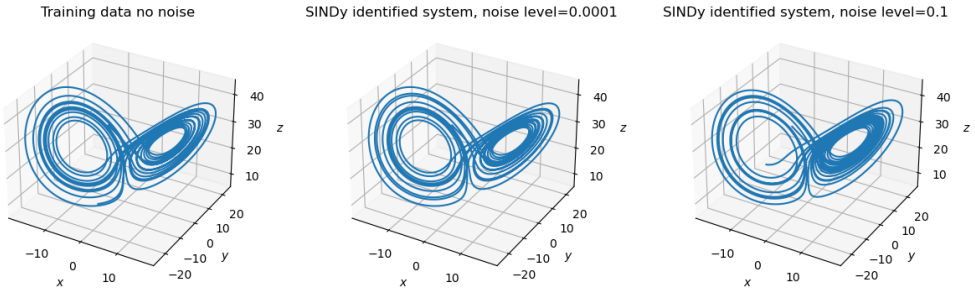


Figure 9: Lorenz System model: Trajectory comparison with two noise levels

Next, Figure 19 shows a clearer picture of the evolution of both y and x with respect to time. We can also clearly see that, as expected, a lower noise level has allowed us to more accurately simulate the data. For higher noise levels, we see that reconstructed data starts diverging from the ground truth after some time.

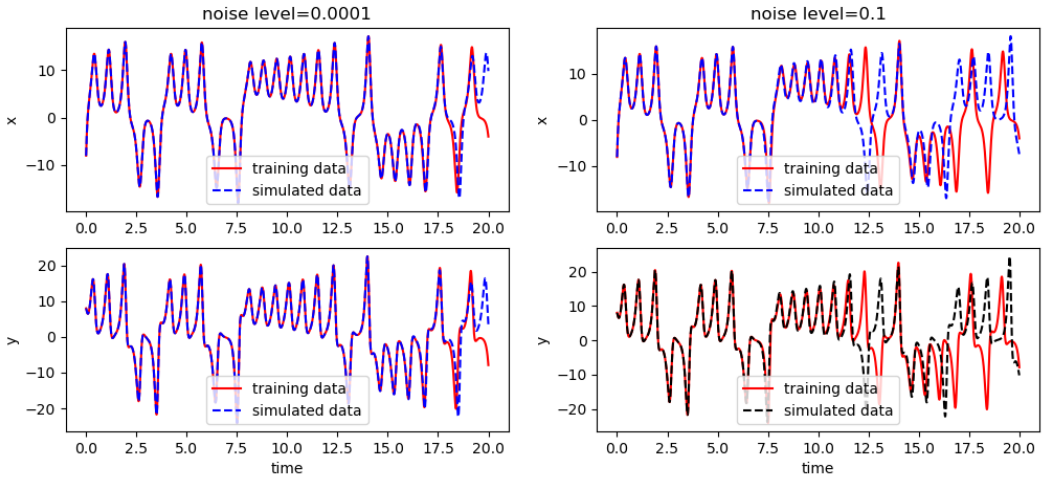
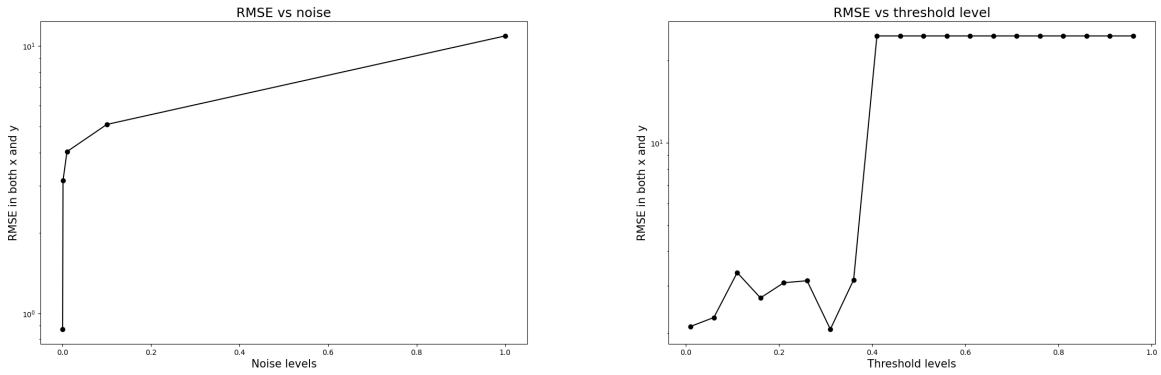


Figure 10: Lorenz System model: Comparison of x and y with respect to time between reconstructed (simulated) and original data (training data)

Finally, we also showed that increasing noise level as well as increasing threshold level both lead to higher errors. Note that we computed the RMSE similarly as before, by taking the Root Mean Squared Error of both X and Y components. Figures 11a and 11b tell us that an optimal choice of parameters would be a threshold of 0.01 and noise level of 10^{-4}. However, it is important not to overfit to the data, and we would need to compare performance on a hold-out test set in order to accurately find the optimal set of parameters that generalizes

best. Note that we are able to reach quite low levels of error on this training set.



(a) RMSE vs noise (b) RMSE vs threshold value

Figure 11: Lorenz System model: RMSE vs hyperparameters

3.2 DeepMod

By running the corresponding code files, you will be able to reconstruct to some extent the governing equations introduced in the first section of this report. Each program outputs the coefficients to reconstruct the governing PDE. In this section, we look at reconstructed data, i.e. data that was simulated based on the recreated governing equation. This allows for easy comparison with our original dataset.

3.2.1 2D Linear Damped Harmonic Oscillator

First, we note that we were able to reconstruct the original governing equation within some error for 250 sampled points. Note that we have chosen the smallest noise level (i.e. 0.001) as well as a the largest for which the reconstruction is acceptable but not complete (i.e. 0.1); these values corresponds to the standard deviations chosen for the normal distributions we are sampling from.

We will first look at the input data in the trajectory phase space with various levels of noise. This is represented in Figure 12. As we can see, with chosen parameters for our model, a noise level of 0.1 renders the training data near random and it is impossible from the data-points alone to visually retrace the trajectory.

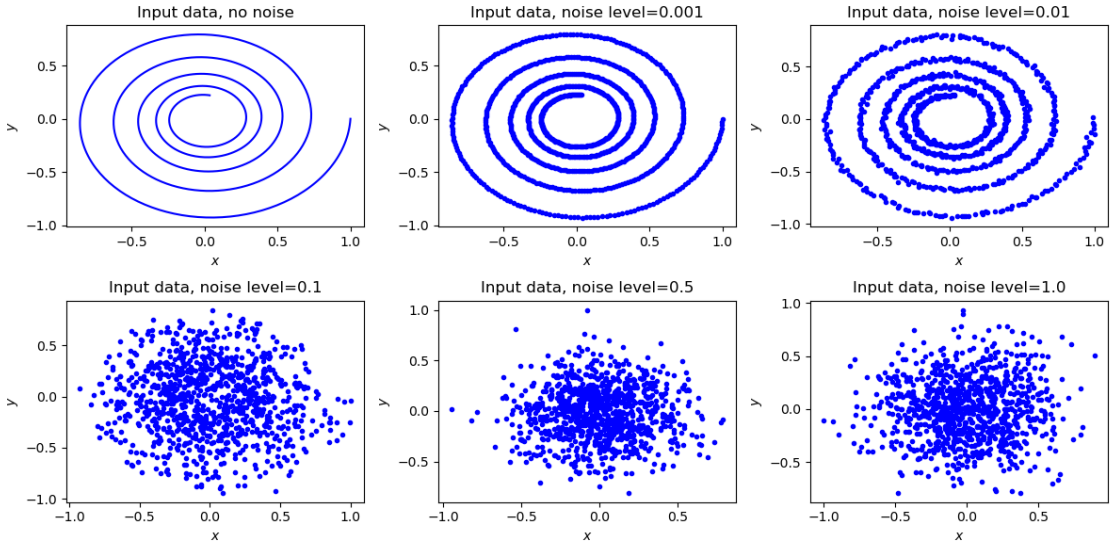


Figure 12: 2D Oscillator model: DeepMod input data represented in trajectory phase space

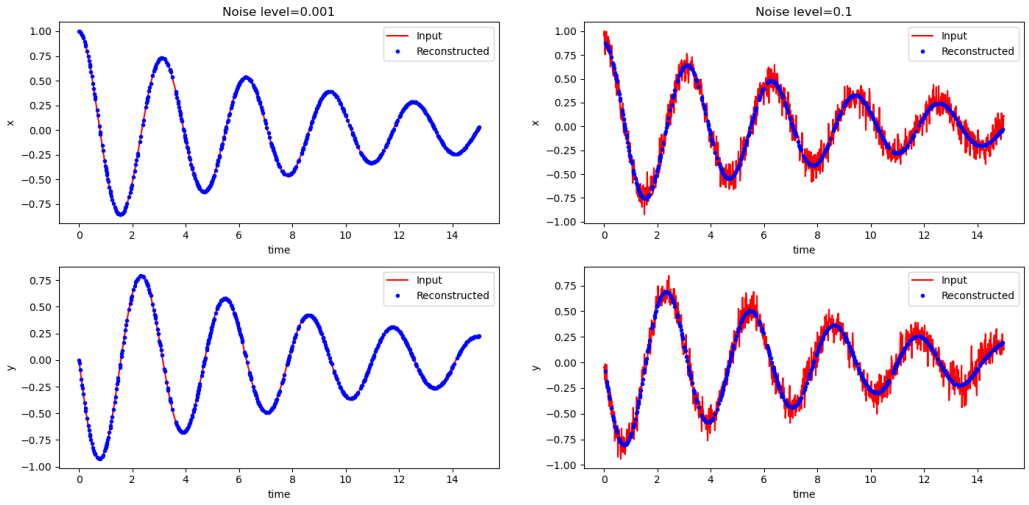
Once the model has been trained, and the sparse library has been determined, we can use our trained model to reconstruct our data, allowing us to compare to the input. These are represented in Figures 13a and 13b. As we can see for a low noise level of 0.001, the data is perfectly reconstructed. However as the noise level increases to 0.1, as time increases, the reconstruction deviates from the input. For noise levels superior to 0.5 included (not represented), the model failed to converge, and the reconstructions are completely off.

Table 1 shows the selected scaled coefficients. As we can see we obtain near exactly the true coefficients (scaled by 1/2)

Mask	**C**	**x**	**y**	**xy**
$\dot{x}(t)$	0	-0.0475	0.995	0
$\dot{y}(t)$	0	-0.999	-0.0502	0

Table 1: 2D Oscillator model: scaled coefficients after sparse selection for a noise level of 0.1

Figure 14 shows the evolution of the RMSE for the different noise levels. As expected, the RMSE increases as the noise increases.



(a) Noise level: 0.001 (b) Noise level: 0.1

Figure 13: 2D Oscillator model: Comparison of x (top) and y (bottom) with respect to time between reconstructed data and input data.

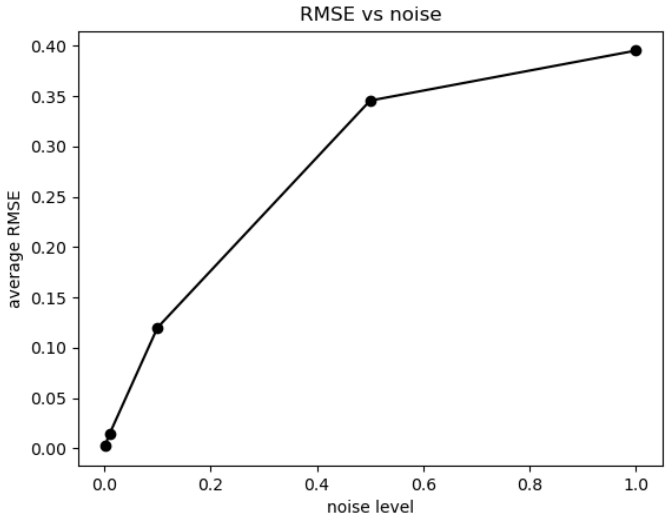


Figure 14: 2D Oscillator model: RMSE vs noise levels for fixed threshold of 0.04

3.2.2 Predator-Prey Model

First, we note that we were able to reconstruct the original governing equation within some error for 250 sampled points. Note that we have chosen the smallest noise level (i.e. 0.001) as well as a the largest for which the reconstruction is acceptable but not complete (i.e. 0.1);

these values corresponds to the standard deviations chosen for the normal distributions we are sampling from.

Similarly, we will first look at the input data in the trajectory phase space with various levels of noise. This is represented in Figure 15. As we can see, with chosen parameters for our model, a noise level of 0.5 renders the training data near random and it is impossible from the data-points alone to visually retrace the trajectory.

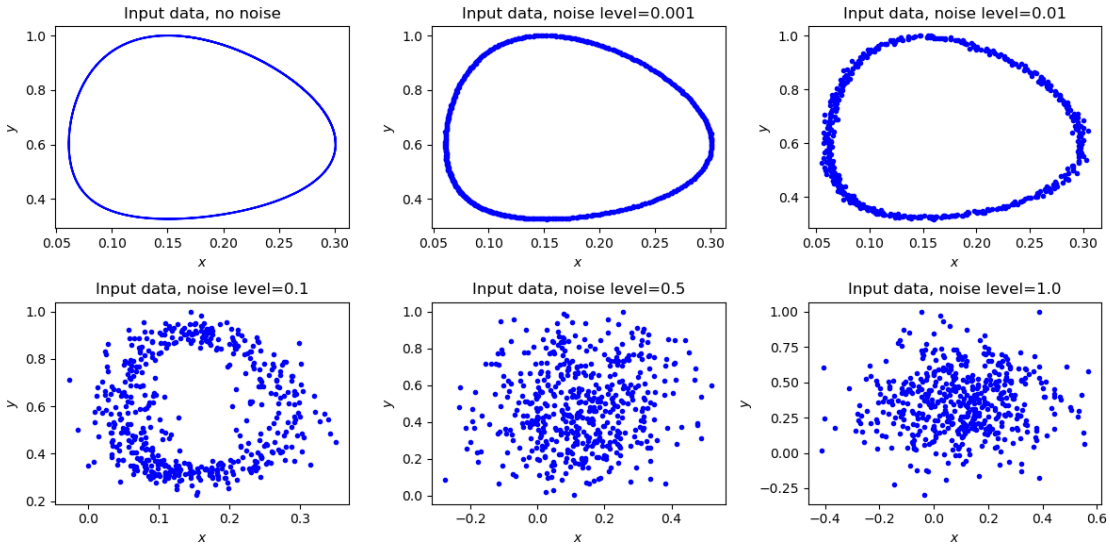
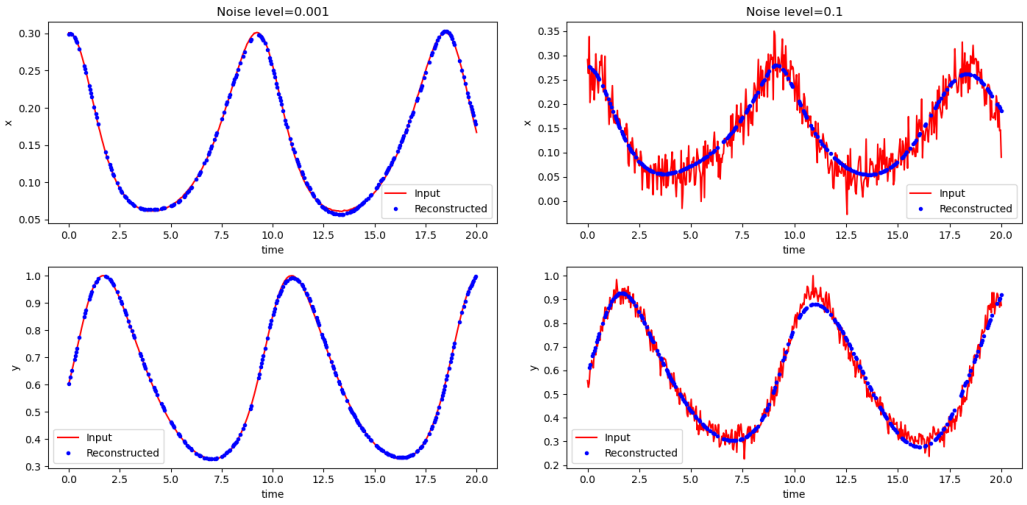


Figure 15: Lotka-Volterra model: DeepMod input data represented in trajectory phase space

Once the model has been trained, and the sparse library has been determined, we can use our trained model to reconstruct our data, allowing us to compare to the input. These are represented in Figures 16a and 16b. With the noise level set at 0.1, as time increases, the reconstruction deviates from the input. For noise levels superior to 0.5 included (not represented), the model failed to converge, and the reconstructions are completely off.

If we look at Table 2 we notice that the scaled coefficients for a noise level of 0.1 are not completely sparse, at that the unwanted terms have not been completely zero-d out. In this case, raising the threshold to 0.75 is sufficient to retrieve the correct model.



(a) Noise level: 0.001 (b) Noise level: 0.1

Figure 16: Lotka-Volterra model: Comparison of x (top) and y (bottom) with respect to time between reconstructed data and input data.

Mask	**C**	**x**	**y**	**xy**
$\dot{x}(t)$	0	2.193	-0.530	-2.475
$\dot{y}(t)$	-0.700	0.740	-1.196	1.492

Table 2: Lotka-Volterra model: scaled coefficients after sparse selection for a noise level of 0.1

Figure 17 shows the evolution of the RMSE for the different noise levels. As expected, the RMSE increases as the noise increases. We can clearly see that the noise level convergence threshold is set around 0.01, below which, the model converges.

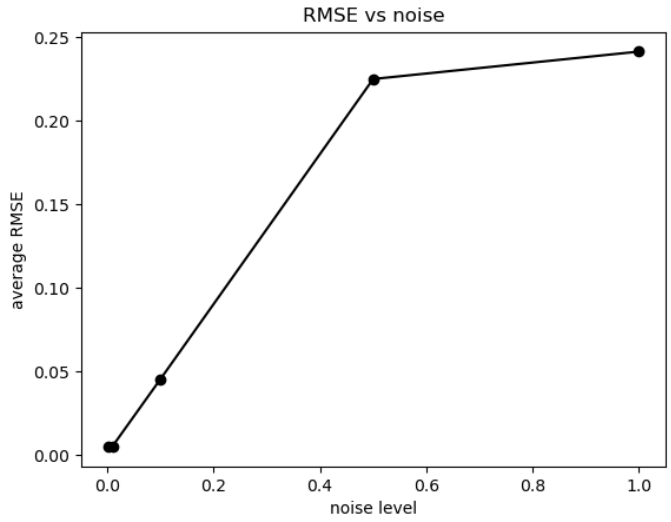


Figure 17: Lotka-Volterra model: RMSE vs noise levels for fixed threshold of 0.5

3.2.3 Lorenz System

First, we note that we were not able to reconstruct the original governing equation. Even for a very low noise level of 0.001, and after an exhaustive search for optimal hyperparameters, model convergence was never achieved.

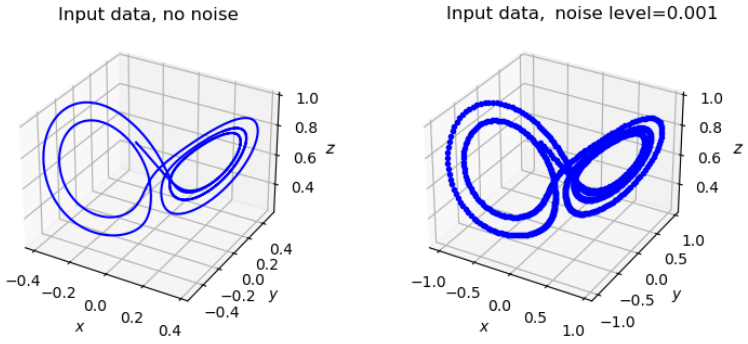


Figure 18: Lorenz System model: DeepMod input data represented in trajectory phase space

As we can see from Figure 18, the sampled input data to DeepMod for a noise level of 0.001 is very similar to that of the noiseless data. Figure 19 compares the reconstruction to the original input. As we can clearly see, convergence is not achieved. The model only captures vaguely the dynamics between $t = 2.5$ and $t = 3.5$. When we correlate this with the chosen coefficient after the sparsity masks were applied as shown in Table 3, we realise that the model is selecting the wrong coefficients. Taking into account the normalisation and

scaling, the model should be selecting $-x$ and y for $\dot{x}$, however, we notice that cross terms such as xz and yz are also being selected with a strong coefficient.

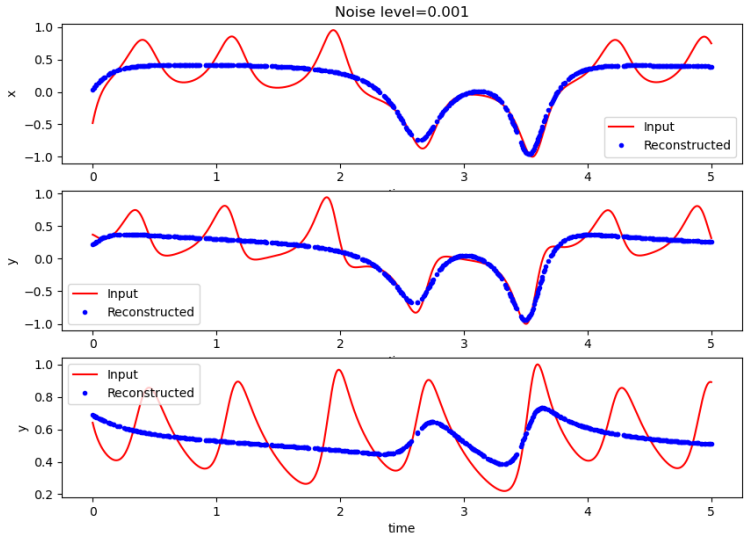


Figure 19: Lorenz System model: Comparison of x (top), y (middle) and z (bottom) with respect to time between reconstructed data and input data.

Mask	**C**	**x**	**y**	**z**	**xy**	**xz**	**yz**
$\dot{x}(t)$	0	3.309	0.836	0	0	-5.653	1.543
$\dot{y}(t)$	-1.827	4.118	0.714	1.768	0	-5.633	0.680
$\dot{z}(t)$	-2.462	1.603	-1.781	1.758	0.892	0	0

Table 3: Lorenz System model: scaled coefficients after sparse selection for a noise level of 0.001

3.3 Comparison of Methods

For the 2D harmonic oscillator and predator-prey models, for noise levels up to 0.1, both SINDy and DeepMod were are able to accurately find the governing equations are reconstruct the data, if tuned adequately. For a noise level range of 0.001 to 0.1 - for the 2D oscillator model:

- SINDy reports a mean RMSE of $\sim 10^{-4}$ to $\sim 10^{-3}$ for 2000 data points and outputs:

$$x' = -0.100 * x + 2.000 * y$$
$$y' = -2.000 * x - 0.100 * y$$

- DeepMod reports a mean RMSE of $\sim 10^{-2}$ to $\sim 10^{-1}$ for 250 data points and outputs (scaled):

$$x' = -0.0496 * x + 0.997 * y$$
$$y' = -0.998 * x - 0.0489 * y$$

For the predator-prey model:

- SINDy reports a mean RMSE of ~ 0.1 to ~ 1 for 100,000 data points and outputs:

$$x' = 1.000 * x - 0.500 * x * y$$
$$y' = -2.000 * y + 0.500 * x * y$$

- DeepMod reports a mean RMSE of $\sim 10^{-3}$ to $\sim 5 * 10^{-2}$ for 250 data points and outputs (scaled):

$$x' = 2.595 * x - 2.929 * x * y$$
$$y' = -1.788 * y + 2.178 * x * y$$

In terms of readability, DeepMod only outputs the magnitude of the coefficients. Because data was normalized, it is not always evident to scale back the coefficients.

Depending on the model, one method performs better than the other in terms of precision. It is important to note that SINDy requires 10 to 50 times more data points to achieve similar results than DeepMod. For example, with 2000 data points and a noise level of 0.1, SINDy was unable to converge for the predator-prey system. However, for the Lorenz system, SINDy performs better than DeepMod, which is unable to converge for this task, even with a dense sampling. Even though SINDy did not find the exact solution for any noise level, it still managed to converge to a similar model. This is mainly because DeepMod was not designed for 3D systems.

In terms of computation time, overall SINDy converges much faster than DeepMod, this is due to the fact that training a neural network it quite costly, both in terms of time and computation power.

Thus, for our specific task of finding the analytical expressions of coupled ODEs, SINDy performs better than DeepMod overall. Data used was synthetic and therefore, we could provide as much as needed for a model to converge. In a real-world scenario where access to quality scientific data is often limited, SINDy might not be a suitable solution, and DeepMod would be the method of choice.

4 Conclusion and Discussions

In summary, we have demonstrated different powerful techniques to discover the underlying physical equations that describe a dataset. This was achieved through sparse regression and a sparse regression approach that also involves neural networks. Although computations were made on synthetic data for demonstration purposes, these results can be extrapolated to non-synthetic 'real-world' data. We showed the benefits of using such models, and their potential use in situations where governing equations are unknown and large datasets can easily be queried. Our demos included simple systems of coupled differential equations from different fields, including mechanics, physics and biology. We discussed the importance of having prior knowledge on the system dynamics and the function types to expect from the equations to be discovered. This is of course a true challenge in real-world scenarios where we might need to consider a very large class of potential functions. This will likely weaken the performance of these algorithms. We also discussed the limits of these methods, which exhibited weaker performance when the noise level was increased and when the number of data points decreased. This tuning of noise levels allowed us to test the robustness of the techniques as well as mimic real-world data scenario. Noise can be a problem when fitting a continuous-time system, which requires taking time derivatives. The latter may be difficult to obtain from high noise data. Finally, care must be taken when drawing conclusions from learned models. While these are great tools to make informed decisions, we should always make sure the results make sense and we should acknowledge the limitations of the conclusions that may be drawn from said results.

Future work would include performing additional testing of these methods on datasets with unknown governing equations. In this project, we have solely demonstrated and compared methods on synthetically generated data in order to show the intricacies of the techniques. Additionally, it would be interesting to study different fields like financial markets dynamics or neuroscience equations to display the strength of these methods in a variety of settings. It is key to understand that these techniques are not one-size-fits-all approaches and should be tuned in accordance with the data. They both contain multiple hyperparameters that need to be carefully chosen depending on the example at hand to obtain coherent results. As data science develops and computational resources improve, these methods will improve and will be even more critical in physics and science.

References

Both, Gert-Jan, Subham Choudhury, et al. (Mar. 2021). "DeepMoD: Deep learning for model discovery in noisy data". In: *Journal of Computational Physics* 428, p. 109985. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2020.109985. URL: http://dx.doi.org/10.1016/j.jcp.2020.109985.

Both, Gert-Jan, Gijs Vermarien, and Remy Kusters (2021). *Sparsely constrained neural networks for model discovery of PDEs*. arXiv: 2011.04336 [cs.LG].

Brunton, Steven L., Joshua L. Proctor, and J. Nathan Kutz (2016). "Discovering governing equations from data by sparse identification of nonlinear dynamical systems". In: *Proceedings of the National Academy of Sciences* 113.15, pp. 3932–3937. ISSN: 0027-8424. DOI: 10.1073/pnas.1517384113. eprint: https://www.pnas.org/content/113/15/3932.full.pdf. URL: https://www.pnas.org/content/113/15/3932.

Kingma, Diederik P. and Jimmy Ba (2017). *Adam: A Method for Stochastic Optimization*. arXiv: 1412.6980 [cs.LG].

Paszke, Adam et al. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Silva, Brian M. de et al. (2020). "Discovery of Physics From Data: Universal Laws and Discrepancies". In: *Frontiers in Artificial Intelligence* 3, p. 25. ISSN: 2624-8212. DOI: 10.3389/frai.2020.00025. URL: https://www.frontiersin.org/article/10.3389/frai.2020.00025.

Udrescu, Silviu-Marian and Max Tegmark (2020). *AI Feynman: a Physics-Inspired Method for Symbolic Regression*. arXiv: 1905.11481 [physics.comp-ph].