DATA-DRIVEN BLACK-BOX MODELING OF HIDDEN SYSTEMS OF ORDINARY AND PARTIAL DIFFERENTIAL EQUATIONS

A THESIS

Presented to the Department of Mathematics & Statistics

California State University, Long Beach

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Applied Statistics

Committee Members:

Seungjoon Lee, Ph.D. (Chair)
Tianni Zhou, Ph.D.
Rebecca Le, Ph.D.

College Designee:

Will Murray, Ph.D.

By Cory Shigeto Suzuki

B.S., 2023, California State University, Long Beach

December 2025

ABSTRACT

Ordinary and partial differential equations are widely used in many applications ranging from elementary Newtonian mechanics to economic options pricing. These models excellently capture the constantly changing nature of variables and provide feasible solutions to problems that require a quantitative perspective. While these models have been proven to be useful, there are existing limitations of ordinary differential equations. These models are dictated by strong assumptions such as initial conditions, boundary conditions, and strong functional forms of both closed-form and numerically approximated solutions. The purpose of this paper is to introduce a physical system from a dynamic, data-driven perspective and formulate a framework in proposing a black-box model of a physical object in motion using partial differential equations and functional approximations to systems of ordinary equations. We aim to utilize statistical methodologies such as dimensionality reduction techniques, proper orthogonal decomposition, regularized L1 regression, and Gaussian filters to extract hidden latent information from a video-based dataset. We propose that this data-driven approach best captures the dynamic nature of the data by providing a detailed analysis of model accuracy and performance and we provide numerical approximations as reasonable solutions to the proposed black-box models.

ACKNOWLEDGEMENTS

I would like to personally express my gratitude to all participating parties who have made the completion of this thesis possible. Firstly, to Dr. Seungjoon Lee, my thesis advisor and Unsupervised Machine Learning professor. I have learned a great deal about what it truly means to think like a data scientist and how to apply various techniques that can be applied to scientific machine learning. Many of the lessons learned have opened my eyes to the difficulties of analyzing and interpreting the results of complex, real-life datasets and Dr. Lee's guidance during the creation of this thesis was crucial in accurately explaining the underlying theoretical frameworks and putting the theory into practice.

Secondly, thank you to my parents, grandmother and countless friends for being an inspiration to go beyond the limits of my own potential. Without their continual words of encouragement and belief in my abilities, I would not have been able to finish or even continue the thesis during the many times of struggling with getting things right. Thank you for giving me all of the necessary reassurance during all the times I got overwhelmed with all the long nights of research and lines of code.

Thirdly, I would like to thank Dr. Tianni Zhou and Dr. Rebecca Le for joining my thesis committee. Not only have I have learned about different hypothesis tests and statistical analyses from Dr. Zhou, but I have also received encouragement and support in improving my determination while writing this thesis. Dr. Le has also provided support in enriching my mathematical statistics knowledge so that I am confidently able to read into the theory of mathematical statistics and the importance of understanding the stories told from working with big data.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The machine learning and big data landscape has significantly impacted the way humanity automates everyday tasks. Although the advent of deep learning has provided cutting-edge technology for various classification tasks, these algorithms also have the advantage of producing predictions and forecasts that facilitate data storytelling. Many industries in engineering have been gradually incorporating data-driven methods and machine learning models in their workloads to solve complex problems. Generally speaking, this movement has enabled research and advancements in scientific machine learning methodologies. This field applies both mathematical and statistical learning theory to solve difficult problems such as generating accurate dynamic models for physical systems, such as Ordinary and Partial Differential Equations, enabling humanity to model the constant change of systems in time and space.

The primary motivation behind this paper is to incorporate statistical supervised learning models to construct partial differential equation models from extracted hidden latent information. This information is wrangled from a video dataset, which is split into a deterministic number of frames. Our main motivation for this paper is to apply statistical methods such as regularized regression, dimensionality reduction, and other machine learning methods to extract latent information from each of the frames' data and create a mathematical model to describe an object's movement in relation to the physical system. This data-driven black-box model was also used to generate predictive snapshot frames of the object for any non-integer time step. This

allows the model the ability to provide a continuous prediction of the object's motion in which this paper presents our findings.

Literature Review and Summary of Established Work

Several researchers have undergone many studies and created publications to provide a general understanding of the current methodologies studied in data-driven partial differential equation models. Many papers utilize statistical algorithms such as regularized regression to extract the coefficients for black-box partial differential equations to model linear and nonlinear dynamics. While this thesis will place an emphasis on a linear system as our main concentration, it is necessary to lay out the research foundations for what is currently being studied.

Data Driven ODE Modeling of the High-Frequency Complex Dynamics via a Low-Frequency Dynamics Model

From the nonlinear perspective, it is often desirable to produce an interpretable system of differential equations that models the nonlinear dynamical motion of objects. Tsutsumi, Nakai, and Saiki (2024) investigate the usage of Radial Function-based Regression techniques with Neural Ordinary Differential Equations to model such phenomena. They used these physics-informed techniques to model the chaotic behavior of the Rössler equation in high dimensions from an observable time series. Tsutsumi, Nakai, and Saiki also demonstrate in this paper that the Radial Function-based Regression model does not perform well by itself on the target variable in the chaotic system. Therefore, they further show that the best model is an autonomously joint model that first captures an autonomous system of a base variable and secondly captures the nature of the target variable that is in terms of the base variable. In this model, the time derivatives are expressed as model features in the implementation of the Radial

Function-based Regression technique and describe the time series dynamics in the spatial domain without any prior background knowledge about the system in the study.

One of the limitations of the Radial Function-based Regression approach is that in high-dimensional data, the introduction of polynomial basis functions increases the computational cost of running the regression algorithm. A benefit that comes with this approach is that with both linear and nonlinear physical systems, the Radial Function-based Regression model can produce an infinite amount of forecasting predictions on the movement of the object within the system once the Regression model has been trained on the initial data (Tsutsumi, Nakai, and Saiki 2024). In general, the regression model is as follows, given the observable D-dimensional time series states $X = (X_1, X_2, \ldots, X_D)^T$:

$$\mathbf{F}(\mathbf{X}) = \beta_0 + \sum_{d=1,\ldots,D} \beta_d X_d + \sum_{j=1,\ldots,J} \beta_{D+j}\, \phi_j(\mathbf{X})$$

In equation above, the $\phi_j(\mathbf{X}) = exp(\frac{-||\mathbf{X}-c_j||^2}{\sigma^2})$ where the c_j's are distributed as lattice points, σ is the standard deviation of a Gaussian distribution, and the regression coefficients β_j are found via Taylor Series Expansion approximations of the time derivatives in the Rössler System. The paper further argues that for their problem, this baseline model is the first part of the autonomous base model given by the governing differential equation system $\frac{d\mathbf{X}}{dt} = \mathbf{F}(\mathbf{X}(t))$ for all positive integers t. Tsutsumi, Nakai, and Saiki then introduce a Fiber Model that acts as the second component of the autonomous system, which incorporates another Radial Function-based Regression model with an attached time step of the target variable Y. The joint Fiber model is given as follows:

$$\frac{d\mathbf{Y}}{dt} = aY_1 + \mathbf{G}(\mathbf{X})$$

$$\mathbf{G}(\mathbf{X}) = \gamma_0 + \sum_{d=1,\dots,D} \gamma_d X_d + \sum_{j=1,\dots,J} \gamma_{D+j}\, \phi_j(\mathbf{X})$$

Coarse-Scale PDEs from Fine-Scale Observations via Machine Learning

The next publication by Seungjoon Lee et al. on *Course-Scale PDE's from Fine-Scale Observations via Machine Learning* (2021) provides the statistical details on modeling complex spatiotemporal dynamics of physicochemical processes at the microscopic level. The paper discusses how Gaussian Process Regression, Diffusion Maps, Artificial Neural Networks (ANN), and Automatic Relevance Determination (ARD) can be used to extract latent information from the microscopic data to produce a black-box macroscale Partial Differential Equation. Furthermore, by using finite difference approximation methods, Lee et al. show that this PDE model can be utilized to extrapolate predictions for the future time stamps that are non-existent in the original microscopic data. By using feature selection and manifold learning approaches for a data-driven framework, long-term macroscopic predictions may be simulated with a reduction in spatiotemporal computational cost once the course evolution law and model are defined.

Their paper explains that after cleaning microscopic data to be finely coarse-scaled, feature selection methods via Gaussian Process Regression, Artificial Neural Networks, and Automatic Relevance Determination via manifold learning are used. Firstly, the temporal and spatial derivatives of the data are approximated via finite difference schemes and stencils. Then these approximations are compiled as the data and are fed through the Gaussian Process Regression methodology. It is stated that to extract the latent features that will be represented in the right-hand side of the Partial Differential equation in this process, it is assumed that each observation is a set of random variables that have a multivariate Gaussian distribution with

unknown means and covariance matrices. However, for the sake of this research, the authors assume zero mean and a covariance matrix K where

$$K_{i,j} = \kappa(\mathbf{x_i}, \mathbf{x_j}; \theta) = \theta_0 \exp(-\frac{1}{2}\sum_{l=1}^{k} \frac{(x_{i,l} - x_{j,l})^2}{\theta_l})$$

where $\theta = (\theta_0, \ldots, \theta_k)^T$ is a $k+1$ dimensional vector of hyperparameters and k is the number of inputs in the data. The optimal value for θ denoted as $\theta^* = \text{argmin}\{-\ln(p(\mathbf{y}|\mathbf{x}; \theta)\}$ which is the negative log marginal likelihood minimization problem with the training dataset $(\mathbf{x}, \mathbf{y})$. In the research publication, it is known that the time derivative in this regression model will follow a multivariate Gaussian distribution, and this can be represented by the following equation:

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{y}* \end{pmatrix} = N\left(\mathbf{0}, \begin{pmatrix} K + \sigma^2 I & K_* \\ K_*^T & K_{**} \end{pmatrix}\right)$$

The predictive distribution $\mathbf{y}*$ for the testing data also has a predictive mean and variance, given as $\bar{\mathbf{y}}* = K_*(K + \sigma^2 I)^{-1}\mathbf{y}$ and $K(\mathbf{y}*) = K_{**} - K_*^T(K + \sigma^2 I)^{-1}K_*$ where K_* is the covariance matrix between training and testing data and K_{**} is the covariance matrix between the testing data.

Lee et al. (2021) also provide background information on the application of deep Artificial Neural Networks that are composed of a connected graph of neurons connected by weights, biases, and an activation function. Namely, the macroscopic variables and their spatial derivatives are assigned at the input layer and are trained to obtain the time derivative in the output layer. Diffusion maps are also applied in this paper to conduct dimensionality reduction and machine learning on high-dimensional manifold data. This algorithm suggests that the eigenvectors of the normalized kernel matrices constructed from the data converge to the Laplace-Beltrami operator's eigenfunctions. These eigenfunctions are then used to nonlinearly

parametrize manifolds. The other methodology outlined in this paper is Gaussian Process ARD via the least absolute shrinkage and selection operator (LASSO) regression. The overarching goal of this algorithm is to search for the subsets of the input domain variables that can minimally parametrize the manifold with the mean squared error loss function.

Data-Driven Science & Engineering: Machine Learning, Dynamical Systems, and Control

This is a textbook written by Steven Brunton and Nathan Kutz (2017) on the methodologies of scientific machine learning and how they can be applied to various problems in the fields of engineering and physics. More specifically, the interest in this publication lies in the chapter of "Learning Coarse-Graining for PDEs." Brunton and Kutz outline the methodology of equation-free heterogeneous multi-scale modeling of multi-scale nonlinear dynamical systems, in which a similar framework will be utilized in this thesis.

Since numerical discretization transforms the Partial Differential Equation to be analyzed into an n-dimensional system of ordinary differential equations, it is easier and computationally efficient to utilize finite-difference methods and a machine learning model to generate approximations to the time and spatial derivatives of the system. This discretization results in a global coupling between all discretization points. A one-dimensional convolutional neural network is used to extract the Korteveg de-Vries (KdV) partial differential equation coefficients and identify the significant features in their textbook example, however the authors state that after the discretization using finite-difference schemes, other machine learning models such as sparse regression can be used as an alternative for linear problems, while non-linear problems may require deep neural networks since training on the high-dimensional data will require the parametrization of the data manifold. The advantage of the flexibility of this framework is that

by using any other models after discretization of the derivatives, the researcher may impose physical constraints to better optimize and provide a robust analysis of the physical system or may gain interpretability of the discovered governing equation through the deterministic choice of the model architecture. Brunton and Kutz (2017) also introduce a modern deep learning approach that utilizes a multi-resolution convolutional autoencoder, which integrates multi-grid methods, convolutional autoencoders, and transfer learning architectures which provides an adaptive methodology that takes advantage of a progressive training approach for multi-scale spatio-temporal data from physical systems. The transfer learning component ensures that learning taking place in the low-dimensional space is able to reconstruct new data in the high-dimensional space as the network deepens and enlarges to capture the governing equation.

Uncovering Closed-Form Governing Equations of Nonlinear Dynamics from Videos

A pattern that can be observed from the current literature is that many of the problems being addressed are concerning non-linear dynamics and chaos, which is a crucial point of research in the field of scientific machine learning. In this publication, Luan, Liu, and Sun (2021) introduce an unsupervised learning framework via data-driven modeling techniques in order to extract physical states and discover closed-form governing equations directly from raw video data. This is beneficial research as they handle high-dimensional unstructured datasets when explicit measurements of such data are not easily obtainable. The paper proposes an autoencoder implementation to localize moving objects and reconstruct frames from the videos. In addition, a transformer that deals with spatial to physical coordinate transformations is used to map pixel trajectories to corresponding latent states. The final component of this framework is a sparse regression model that extracts the best coefficients for a nonlinear system of Ordinary Differential Equations that govern such states of interest. To increase the accuracy of the state

extraction models, Luan, Liu, and Sun (2021) impose physical constraints from the system as regularizers during model training. Some examples covered in this paper include discovering the governing equations for Duffing, Van der Pol, and magnetic oscillators without prior knowledge of the governing equations. However, the actual governing equations for these systems are used as ground truths to compare with the predictions made by the framework. One of the advantages of this process is that the deep learning capabilities can handle nonlinear and unstructured data.

Proper Orthogonal Decomposition

Historically speaking, Proper Orthogonal Decomposition has been a technique that has been around since the 1950s and has been frequently used in recent times as a method of analyzing fluid dynamics and turbulence flows (Berkooz, Holmes, and Lumley 1993). The Proper Orthogonal Decomposition was made possible through the Karhunen-Loeve Theorem which states that a stochastic process can be represented as an infinite linear combination of orthogonal functions (Stark and Woods 1986). This idea is very similar to the fact that a function on a bounded interval can have a Fourier Series representation.

Anindya Chatterjee (2000) provides a brief introduction to the Proper Orthogonal Decomposition (POD) method, which is analogous to the Singular Value Decomposition (SVD) method for matrix decompositions. This is ideal since many large-scale data computations require an associated computational complexity cost, and so it is necessary to achieve computational efficiency when decomposing matrices for data analysis and processing. The POD method aims to decompose high-dimensional data to obtain a lower-dimensional representation of such data while preserving its latent structure. More specifically, the POD method is a continuous version of the SVD method, which is the discrete analog to POD. Chatterjee also

provides some examples of the implementation of the algorithm and compares the computational runtime to the SVD algorithm and the traditional Eigenvalue Decomposition algorithm, in which Chatterjee argues that POD produces promising matrix decompositions for experimental data used in image processing, structural vibration analysis, and fluid dynamics.

Proper Orthogonal Decomposition is effective at matrix decomposition in systems where data measurements are collected at multiple locations in the space domain over the time domain. By applying the method to an instantiated data matrix, it is easy to obtain the best set of orthonormal eigenvectors that provide the best representation of the data concerning the least squares minimization. Suppose that A is an $n \times m$ matrix. Then A can be decomposed as:

$$A = U\Sigma V^T$$

where U is of size $n \times n$, Σ is of size $n \times m$, and V is of size $m \times m$. It can be noted that in Σ, all elements are zero with the exception of the main diagonal. POD can also be used to find the kth lower order approximation of a given matrix A, so equation 1.6 can be rewritten as:

$$A = U\Sigma_k V^T$$

This POD approximation is the most optimal according to Chatterjee's paper (2000) since no other k matrix exists such that it is close to A in the Frobenius norm or discretized L^2 norm. In this approximation, the first k columns in the matrix V provide an orthonormal basis for approximating the data, and obviously imply that the columns of V are orthogonal eigenvectors. The POD may be computationally efficient; however, there are some drawbacks to this methodology. For example, Chatterjee (2000) states that POD is sensitive to the change of data points and coordinate transformations. In addition, POD and its continuous version, SVD, are linear decomposition techniques used for dimensionality reduction and feature extraction; hence, they are not suitable for datasets that have a nonlinear nature. Specifically, the linear nature of

these algorithms would not be particularly useful for analyzing data that lies in multidimensional manifolds.

CHAPTER 2

PRELIMINARIES AND BACKGROUND

Mathematical Preliminaries and Theory

Before presenting the results of this paper, it is worth noting that readers should be familiar with the formal mathematical theory of the methodologies used. Many ideas behind black-box partial differential equations arise from physics, real analysis, numerical analysis, and statistics. Namely, this section primarily reviews the theory of Ordinary Differential Equations, Partial Differential Equations, Numerical Methods for ODEs and PDEs, the LASSO regularized regression algorithm, and various dimensionality reduction techniques such as Singular Value Decomposition and Proper Orthogonal Decomposition. The underlying theory of these topics shape the underlying framework of this thesis.

Ordinary and Partial Differential Equations

Ordinary and partial differential equations are mathematical models that are frequently used to represent the systematic relationship between an independent variable and its associated rates of change over time and space. More generally, let $n \in \mathbb{N}$ and $y: Y \to \mathbb{R}$, where Y is an open subset of $\mathbb{R}^n$. Then an nth-order Ordinary Differential Equation (ODE) follows the functional form:

$$F\left(x, y, \frac{dy}{dx}, \ldots, \frac{d^{k-1}y}{dx^{k-1}}, \frac{d^k y}{dx^k}\right) = 0$$

where the function $F: \mathbb{R}^{n^k} \times \mathbb{R}^{n^{k-1}} \times \mathbb{R}^n \times \mathbb{R} \times Y \to \mathbb{R}$ is dependent on x, y, and the derivatives of the independent variable y (Boyce and DiPrima 2012). We note in particular that Ordinary Differential Equations only consist of the derivatives of a single dependent variable, usually

denoted traditionally as x or t in most physical problems. Linear differential equations take on the form of

$$a_0(x)y + a_1(x)y' + a_2(x)y'' + \ldots + a_n y^{(n)} = 0$$

where $a_0(x), a_1(x), a_2(x), \ldots, a_n(x)$ are differentiable functions of the independent variable and $y', y'', \ldots, y^{(n)}$ are the first, second, up to the nth derivatives of the dependent variable. An important note is that linear differential equations can be written as linear combinations of these differentiable functions with the derivatives of the dependent variable. Homogeneous differential equations are equal to zero while nonhomogeneous differential equations are equated to a single-variable function. Nonlinear ODEs generally require numerical methods for truncated approximations of their solutions which involve the applications of Taylor Series expansions. A few nonlinear equations such as the Riccati Equation can be transformed into an equivalent linear ODE which is easier to solve to obtain analytical, closed-form solutions. Linear ODEs can be solved with elementary functions and their integrals, provided the integrals exist.

Partial differential equations (PDE) are extensions of Ordinary Differential Equations to relate independent variables with the derivatives of $n \geq 2$ variables within the system. Let $u: U \to \mathbb{R}$ be an arbitrary, unknown function that is a solution to a partial differential equation. Let $x = (x_1, x_2, \ldots, x_n)$ be the variables belonging to the open subset U of the Euclidian space $\mathbb{R}^n$. Then the kth-order partial differential equation follows the general form:

$$F(D^k u, D^{k-1} u, \ldots, Du, u, x) = 0$$

In this general form, D is the partial differential operator and F is the mapping $F: \mathbb{R}^{n^k} \times \mathbb{R}^{n^{k-1}} \times \mathbb{R}^n \times \mathbb{R} \times U \to \mathbb{R}$. This class of equations also has linear and nonlinear variants in the same fashion as ODEs, where nonlinear problems are much harder to solve and may require

numerical methods to compute (Kværnø 2020). A notable example of a linear PDE is the diffusion equation $u_t = ku_{xx}$ where the diffusive constant $k \in \mathbb{R}$, and a nonlinear PDE example is Burgers' Equation $u_t + uu_x = \nu u_{xx}$, where the diffusion coefficient $\nu \in \mathbb{R}$. Differential equation problems can also have initial and boundary conditions imposed on them. When an initial value is imposed on a differential equation, it becomes and initial value problem (IVP). Initial conditions are of the form $f(t_0) = f_0$, $f'(t_0) = f_1$, $f''(t_0) = f_2$, and so forth. Suppose we have a first-order differential equation $y'(x) = f(x, y)$ with the initial condition $y(c) = y_0$ for some real-valued constants c and y_0. Then if we solve the differential equation and apply the initial condition, then our solution is of the form:

$$y(x) = y_0 + \int f(t, y)dt$$

where y is a function of x and t is the dummy variable of integration. Initial conditions can also be applied to the $n-1$th order for an nth order differential equation. For more information on various types of ordinary differential equations, one may refer to Boyce and DiPrima's *Elementary Differential Equations and Boundary Value Problems* (2012). For ordinary differential equations and partial differential equations, boundary conditions and initial conditions can be applied to the differential equation which formulates a boundary value problem (BVP). For example, the second-order Sturm-Liouville boundary value problem can be posed as:

$$X'' + \lambda^2 X = 0$$
$$X(0) = X(L) = 0$$

where X is a function of x and L is some real-valued constant. The λ^2 here are the associated eigenvalues for the Boundary Value Problem. The eigenvalues and solution for this problem are given by $\lambda_n = \frac{n\pi}{L}$ and $X_n(x) = \sin(\frac{n\pi x}{L})$ for $n \in \mathbb{N}$. These types of problems involve Dirichlet

boundary conditions where both conditions are using the original function. There are other sets of boundary conditions such as Neumann, Periodic, and Mixed boundary conditions. In this thesis, our models will assume no boundary conditions, an advantage that we will see that will play a role at a later time.

Fundamental Types of Partial Differential Equations & their Physics Intuition

In this section, two fundamental types of partial differential equations that play a major role in this thesis will be presented here for the sake of background knowledge. The diffusion equation, sometimes alternatively referred to as the heat equation, takes on the form:

$$u_t = ku_{xx}$$

for $0 < x < L$ and $t \geq 0$, where $k \in \mathbb{R}$ is the heat-diffusive constant. For the sake of simplicity, assume that we have boundary conditions $u(0,t) = 0$ and $u(L,t) = 0$ where L is a constant and initial condition $u(x,0) = f(x)$ where $f(x)$ is a real-valued function. The classical approach taken by many mathematicians such as Daniel Bernoulli and Leonhard Euler is known as the method of separation of variables. The rationale behind this method is to decompose the partial differential equation into a system of ordinary differential equations. Assume that the solution is of the form $u(x,t) = X(x)T(t)$. By taking the first derivative with respect to t and the second derivative with respect to x, we have the following:

$$u_t = X(x)T'(t)$$
$$u_{xx} = X''(x)T(t)$$

The above equations can be used to separate the variables which yields:

$$\frac{T'}{T} = \frac{X''}{X}$$

For convenience, we may equate the above equation to an arbitrary separation constant, call it $-\lambda$ and using algebraic manipulation gives a system of two ordinary differential equations. As seen below, the variables have been separated where the first equation below is a second-order ordinary differential equation and the following equation is a first-order ordinary differential equation.

$$X'' + \lambda X = 0$$

$$T' + k\lambda T = 0$$

Now to obtain the solution of the equation in terms of X, we apply the boundary conditions that require $X(0) = X(L) = 0$. By using Sturm-Liouville theory, the nontrivial solutions other than 0 can be found through the eigenfunctions $X_n(x) = sin(\frac{n\pi x}{L})$ with associated eigenvalues $\lambda_n = \frac{n^2\pi^2}{L^2}$ for $n \in \mathbb{N}$ Note that after finding these eigenvalues, they must be back-substituted into equation 2.10 and then it can be solved via the integration factor method. Hence our fundamental solutions for the heat equation lead to:

$$u_n(x,t) = \exp((-n^2\pi^2 k)/L^2)\sin(\frac{n\pi x}{L})$$

for $n \in \mathbb{N}$. Hence, we can represent these fundamental solutions as a linear combination via an infinite series. The general solution given by:

$$u(x,t) = \sum_{n=1}^{\infty} c_n\, u_n(x,t)$$

Applying the initial condition and recognizing the Fourier sine series representation allows us to solve for c_n. This leads to the particular solution where

$$c_n = \frac{2}{L}\int_0^L f\,(x)\sin(\frac{n\pi x}{L})dx$$

Now, the wave equation first studied by the mathematician d'Alembert is another type of model that arises in many natural physical systems that model the movement of waves. It is given in generality by:

$$u_{tt} = c^2 u_{xx}$$

for $0 < x < L$ and $t \geq 0$ where $c \in \mathbb{R}$ is the wave propagation constant. Here, it is traditional to impose the boundary conditions $u(0,t) = u(L,t) = 0$ and the initial conditions $u(x,0) = f(x)$ and $u_t(x,0) = g(x)$ where $f(x)$ and $g(x)$ are real-valued functions. Once again, the most common analytical approach is to use the method of separation of variables in an analogous fashion taken to address the particular solution to the diffusion equation. Instead, we have two second-order ordinary differential equations:

$$X''(x) + \lambda X = 0$$
$$T'' + c^2 \lambda T = 0$$

It can be shown with the separation of variables method that the particular solution for the wave equation in the situation of an elastic string having nonzero initial displacement with $u(x,0) = f(x)$ and $u_t(x,0) = 0$ is:

$$u(x,t) = \sum_{n=1}^{\infty} c_n \sin(\frac{n\pi x}{L})\cos(\frac{n\pi ct}{L})$$

where

$$c_n = \frac{2}{L}\int_0^L f\,(x)\sin(\frac{n\pi x}{L})dx$$

for $n \in \mathbb{N}$. As with many types of PDEs, one can augment the initial conditions and boundary conditions to model different type of wave propagation phenomena (Strauss 2008). This concludes the theory behind the classical diffusion equation and heat equation.

System of ODEs

Ordinary differential equations are useful on their own in many applications, however many dynamical physical systems require more complex models such as multiple ordinary differential equations that share a common solution. Hence in this section, this paper will introduce the theory and an example of systems of ordinary differential equations that will be relevant at a later time.

Systems of ODEs can be classified into linear and non-linear classes. Often times, it is ideal to reframe systems of ODEs into their matrix equivalents. For a system of n first-order linear equations,

$$\begin{aligned} x_{1'} &= p_{11}(t)x_1 + \ldots + p_{1n}(t)x_n + g_1(t), \\ &\vdots \\ x_{n'} &= p_{n1}(t)x_1 + \ldots + p_{nn}(t)x_n + g_n(t) \end{aligned}$$

We can express this system in matrix algebra notation where $\mathbf{x}(t)$ is the vector consisting of the elements $x_1(t), \ldots, x_n(t)$, $\mathbf{g}(t)$ is the vector consisting of the components $g_1(t), \ldots, g_n(t)$, and $p_{11}(t), \ldots, p_{nn}(t)$ are elements of an $n \times n$ matrix $\mathbf{P}(t)$. The resulting equation is $\mathbf{x}' = \mathbf{P}(t)\mathbf{x} + \mathbf{g}(t)$. One can solve this system by first finding the homogeneous solution by making $\mathbf{g}(t) = \mathbf{0}$. So we can denote the solutions as

$$\mathbf{x}^{(1)}(t) = \begin{pmatrix} x_{11}(t) \\ x_{21}(t) \\ \vdots \\ x_{n1}(t) \end{pmatrix}, \ldots, \mathbf{x}^{(k)}(t) = \begin{pmatrix} x_{1k}(t) \\ x_{2k}(t) \\ \vdots \\ x_{nk}(t) \end{pmatrix}, \ldots$$

As an example, we will take a look at the analytical solution for the following system of ODE's:

$$\mathbf{x}' = \begin{pmatrix} 2 & 0 \\ 0 & -3 \end{pmatrix} \mathbf{x}$$

This is equivalent to solving the separate ODE's $x_{1'} = 2x_1$ and $x_{2'} = -3x_2$. Solving these ODE's gives us:

$$\mathbf{x} = \begin{pmatrix} c_1 e^{2t} \\ c_2 e^{-3t} \end{pmatrix} = c_1 \begin{pmatrix} 1 \\ 0 \end{pmatrix} e^{2t} + c_2 \begin{pmatrix} 0 \\ 1 \end{pmatrix} e^{-3t}$$

A further analysis of this problem to determine if this is a fundamental set of solutions to the system in addition to being the general solutions will require one to check for linear independence at each point in the t interval by taking the determinant of the Wronskian matrix and ensuring it is nonzero. While this is outside of the scope of this thesis, the reader may consult Boyce and DiPrima's text or any introductory text on ODEs for more information on the theory behind systems of ODEs.

Numerical Methods for Ordinary and Partial Differential Equations

Many ODE's and PDE's often require the implementation of numerical algorithms via computer programming to approximate their solutions, provided a closed-form solution does not exist or is hard to obtain. The most common approach used is by iteratively computed finite differences, which approximate derivatives using stencils from the function's data. For example, to approximate u_t, we take the current value of the function and subtract it by the previous value in the grid stencil. We then divide this difference by the change in step size. So,

$$u_t = \frac{u^{(i+1)} - u^{(i)}}{h}$$

where i denotes the iteration number and h is the step size. This idea can be extended to all derivatives in the differential equation up to the nth order as desired. As an example, suppose that we are given the following boundary value problem:

$$\frac{d^2y}{dx^2} = -g$$
$$y(0) = 0$$
$$y(5) = 50$$

for some real-valued constant g. If we let $n = 10$ be the number of deterministic subintervals, then we can find the time step value h. In this example, $h = 0.5$ since we take the difference of the given time interval $[0,5]$ and divide by the number of subintervals which is 10. Since the finite difference approximation for the second derivative with respect to x is

$$\frac{d^2y}{dx^2} = \frac{y_{i-1} - 2y_i + y_{i+1}}{h^2}$$

for $i = 1,2,\ldots,n-1$, then equating this to the right-hand side $-g$, we will then have the following system of equations

$$\begin{pmatrix} 1 & 0 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ \cdots \\ y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} 0 \\ -gh^2 \\ \cdots \\ -gh^2 \\ 50 \end{pmatrix}$$

The system of $n + 1$ equations can now be solved for the solutions $y_0, y_1, \ldots, y_{n-1}, y_n$. By using finite difference methods, we can approximate the solution of the boundary value problem, in which we provide the code for this in the appendices. Solving this system of equations discretely results in the following approximated solution plot in Figure 2.1.

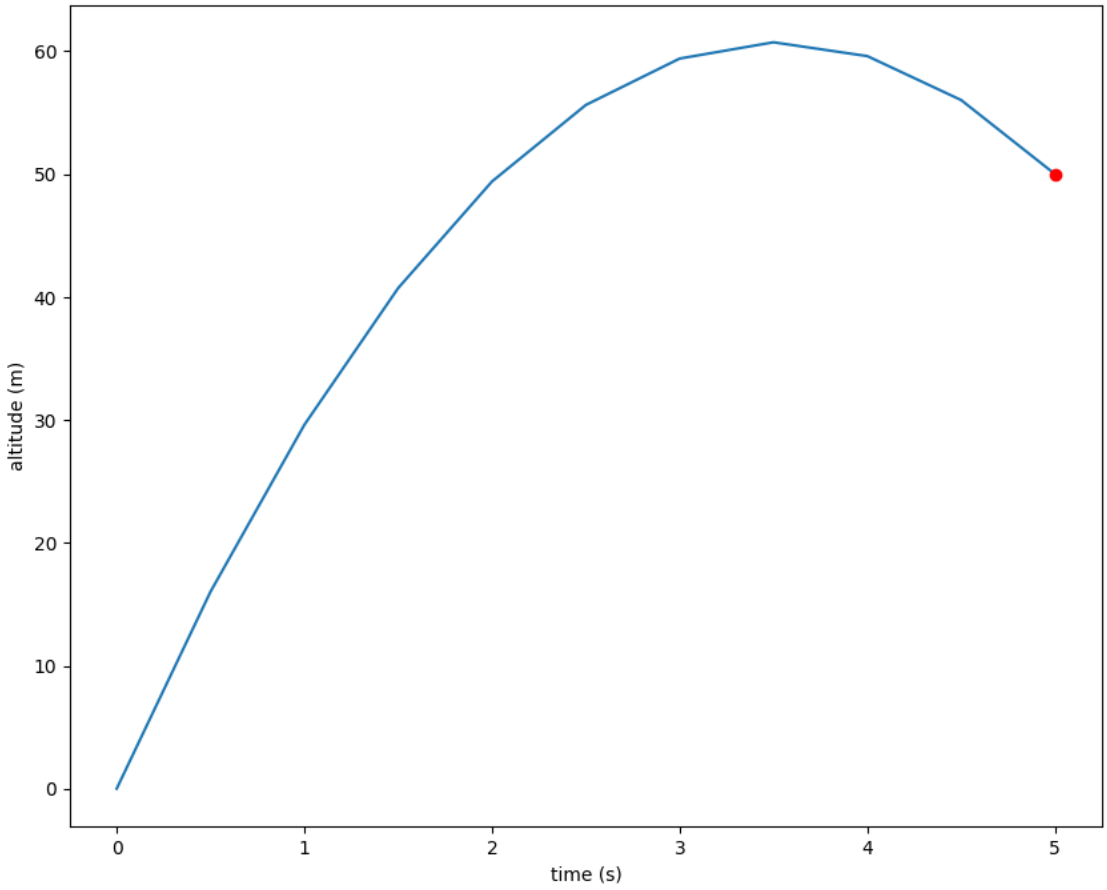

FIGURE 2.1. Boundary value problem solution plot

We will also demonstrate the finite difference method for the diffusion/heat equation. Below is the problem that we will solve with the associated boundary and initial conditions:

$$u_t = u_{xx}$$
$$u(0,t) = u(1,t) = 0 \qquad \text{Boundary Conditions}$$
$$u(x,0) = \sin(\pi x) \qquad \text{Initial Conditions}$$

To approximate the solution for this partial differential equation using the finite difference method, we will use the pseudocode outlined in Algorithm 1.

Algorithm 1 Diffusion Equation Finite Difference

$M \leftarrow 20$
$N \leftarrow 1000$
$dx \leftarrow 1.0/M$
$dt \leftarrow 1.0/N$
$\lambda \leftarrow dt/dx^2$
if $\lambda > 0.5$ **then**
    STOP
end if
$u(0) \leftarrow u(1) \leftarrow 0$
for i in $1, ..., N$ **do**
    $u(i+1) \leftarrow u(i)$
    for j in $1, ..., M$ **do**
        $u(i+1, j) \leftarrow u(j) + \lambda(u(j+1) - 2u(j) + u(j-1))$
    end for
    $u(i) \leftarrow u(i+1)$
end for

Implementing Algorithm 1 in Python will generate the following solution plot in Figure 2.2. We can see that since our initial condition is a sine function, our approximated solution in solid blue will closely follow the analytical solution $u(x,t) = \exp(-\pi^2 t)\sin(\pi x)$ which is indicated by the yellow dashed line in Figure 2.2. Doing a simple maximum error analysis throughout the whole domain between the true and approximated solutions yields an error of 1.0625×10^{-3}.

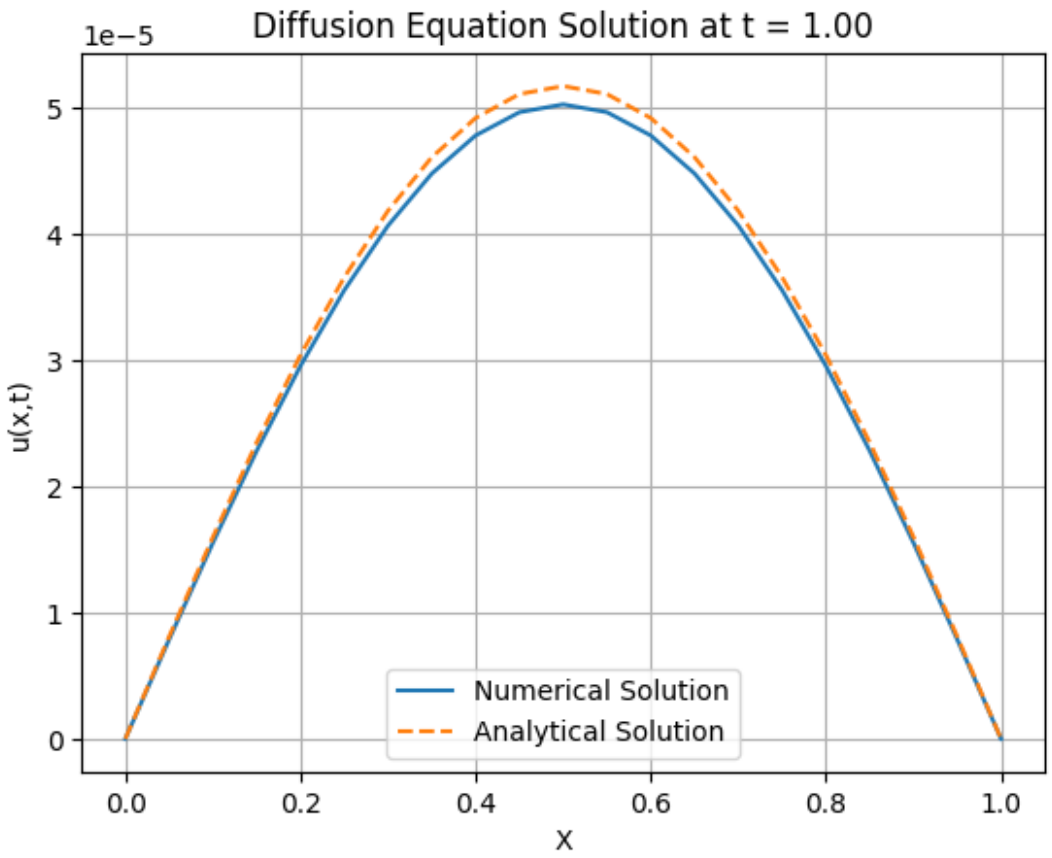

FIGURE 2.2. Solution Plot for Diffusion Equation for $0 < t < 1$

All relevant codes for this example will be given in the appendices. This concludes the preliminary theoretical foundation of the application of the finite difference method. More information on finite difference methods and numerical analysis for ODEs and PDEs can be found in Burden, Faires, and Burden's (2015) *Numerical Analysis* text and in Kværnø's lecture notes textbook.

Simple and Multiple Linear Regression

In this subsection, it is assumed to be necessary to provide a general summary of simple and multiple linear regression analysis. Linear regression analysis models formulate the relationship between one or more explanatory predictors and a single response variable. In the machine learning paradigm, linear regression is a supervised learning algorithm that learns from

labeled data and creates predictions for the response variable. The simple linear regression model is given by:

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i$$

where β_0 is the intercept, β_1 is the slope, and x_i are the rows of the uni-variate predictor variable. Here, the random errors $\epsilon_i \sim N(0, \sigma^2)$. Other assumptions for linear regression include the independence of the errors, linearity between the dependent and independent variables, and homoscedasticity of the residuals (having equal variance) (Kutner, Nachtsheim, and Neter 2004). To find the optimal regression coefficients β_0 and β_1, the following minimization problem is used to find the estimates $\hat{\beta}_0$ and $\hat{\beta}_1$:

$$\min_{\beta_0, \beta_1} \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)^2$$

This is the least squares method to minimize the sum of squared error, where the above problem is obtained since $\epsilon_i = y_i - \beta_0 - \beta_1 x_i$ which follows from the simple linear regression model. By taking the first derivatives of the objective function with respect to each of the coefficient parameters, this yields the following estimates:

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n} (x_i - \bar{x})^2}$$

where $\hat{y_i} = \hat{\beta}_0 + \hat{\beta}_1 x_i$. In this new estimated line of best fit, it is customary to omit the usage of the random errors and instead use $e_i = y_i - \hat{y_i}$. These are the residuals which measure the deviance between the ground truth data values and the predicted values from the linear regression model.

Now for multiple linear regression for p number of predictors, it is more convenient to express the multiple linear regression model in matrices and vectors. Suppose we have the model $y_i = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p + \epsilon_i$. Then we can rewrite the model in matrix form:

$$\mathbf{y} = X\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

where $\mathbf{y}$ is the response vector $(y_1, y_2, \ldots, y_n)^T$, $\boldsymbol{\beta}$ is the regression coefficient vector consisting of $(\beta_0, \beta_1, \ldots, \beta_p)^T$, $\boldsymbol{\epsilon}$ is the random error vector $(\epsilon_1, \epsilon_2, \ldots, \epsilon_n)^T$ and the X is the following design matrix:

$$X = \begin{pmatrix} 1 & x_{11} & x_{12} & \ldots & x_{1p} \\ 1 & x_{21} & x_{22} & \ldots & x_{2p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \ldots & x_{np} \end{pmatrix}$$

Observe that since the response vector is of size $n \times 1$, we must have a $n \times p$ design matrix being multiplied by a $p \times 1$ coefficient vector. This is intuitive as the model assumes that we have p many predictors. In an analogous fashion, it must hold by the laws of vector addition that both $X\boldsymbol{\beta}$ and $\boldsymbol{\epsilon}$ are of size $n \times 1$. Each random error in the random vector is normally distributed with zero mean and finite variance σ^2. It can be shown via matrix algebra that the estimated response vector is of the form:

$$\hat{\mathbf{y}} = X(X^T X)X^T \mathbf{y}$$

Regression analysis includes the construction of an Analysis of Variance (ANOVA) table which decomposes the sources of errors in the regression model. For example, the ANOVA table for k number of predictors is provided in Table 2.1.

TABLE 2.1. ANOVA Table for Multiple Linear Regression

Source	df	SS	MS	F
Regression	k	SSR	MSR = SSR/k	MSR/MSE

Error	n-(k+1)	SSE	MSE = SSE/(n-(k+1))
Total	n-1	SSTO	

In Table 2.1, SSR denotes the sum of squares regression (equation 2.28), SSE is the sum of squares error, and SSTO is the total sum of squares in the model (equation 2.30) which are given by the formulas below.

$$\sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2$$

$$\sum_{i=1}^{n}(y_i - \hat{y})^2$$

$$\sum_{i=1}^{n}(y_i - \bar{y})^2$$

Therefore in natural language, the Sum of Squares regression represents the sum of squares of the difference between each predicted value and the mean response. In a similar fashion, the Sum of Squares error is the sum of squares of the difference between each actual value and the predicted response and the Sum of Squares total is the sum of squares of the difference between each actual value and the mean response. To obtain the mean squared regression and mean squared error denoted as MSR and MSE respectively, divide the sum of squares by the degrees of freedom (df). The corresponding F statistic which follows an F-distribution with respective degrees of freedoms numerator and denominator k and $n-(k+1)$, is utilized in the hypothesis testing for the statistical significance of the k coefficients. The F-distribution is given as:

$$f(x; d_1, d_2) = \frac{1}{Beta(\frac{d_1}{2}, \frac{d_2}{2})}(\frac{d_1}{d_2})^{\frac{d_1}{2}} x^{\frac{d_1}{2}-1}(1 + \frac{d_1}{d_2}x)^{-\frac{d_1+d_2}{2}}$$

for $0 < x < \infty$. It is worthy to note that the F-distribution's probability density function also consists of the Beta distribution. Here, d_1 and d_2 denote the respective numerator and denominator degrees of freedoms. For more details on the theory behind the F-distribution, one can consult Wackerly, Mendenhall, and Schaefer's *Mathematical Statistics with Applications* (2008). At a $100\%(1-\alpha)$ level of significance, the following test is used:

$$H_0: \beta_i = 0$$
$$H_a: \beta_i \neq \beta_j$$

for $i \neq j$. If the p-value $P(F > F^*)$ is less than the given deterministic significance level, then we reject the null hypothesis H_0 and we have sufficient evidence to conclude that the regression coefficients are significant. Otherwise, we fail to reject the null hypothesis H_0 and we do not have sufficient evidence to suggest that the coefficients are significant. In conjunction with this type of hypothesis testing for feature selection, we also use regularized regression via the LASSO algorithm to provide a more robust analysis. More specifically, LASSO will be used to extract the coefficients of our physics-driven ODEs and PDEs in which the theory is presented in the following subsection.

Least Absolute Shrinkage and Selection Operator (LASSO) Algorithm

As independently described by Tibshirani (1996), the least absolute shrinkage and selection operator (LASSO) Algorithm is a statistical regression analysis method that performs feature selection and regularization in the L^1 norm. It is used to shrink insignificant regression coefficients down to zero under the assumption of a regression model $y = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p + \epsilon_i$, where the random errors $\epsilon_i \sim N(0, \sigma^2)$. Under this regression model with p predictors, the LASSO algorithm can be written as a constrained quadratic programming minimization problem:

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{N} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2$$

subject to: $\sum_{j=1}^{p} |\beta_j| \leq t$

where p represents the number of predictors and N is the number of samples. It is important to note that in contrast to Ridge regularized regression, the L^1 LASSO penalty has the absolute value of the coefficients while the ridge L^2 penalty is the sum of the squared coefficients. Here, t is specifically chosen such that the expected prediction error estimate is minimized, and one can notice that LASSO thereby is able to perform continuous subset selection through this algorithm. There are many interpretations as to how LASSO performs feature selection and shrinks the β coefficients down to zero. As provided in Figure 2.3, the constraint area has corners. When the sum of squares is exactly at one of these corner points, the corresponding coefficient on the axis where the corner lies is set to zero.

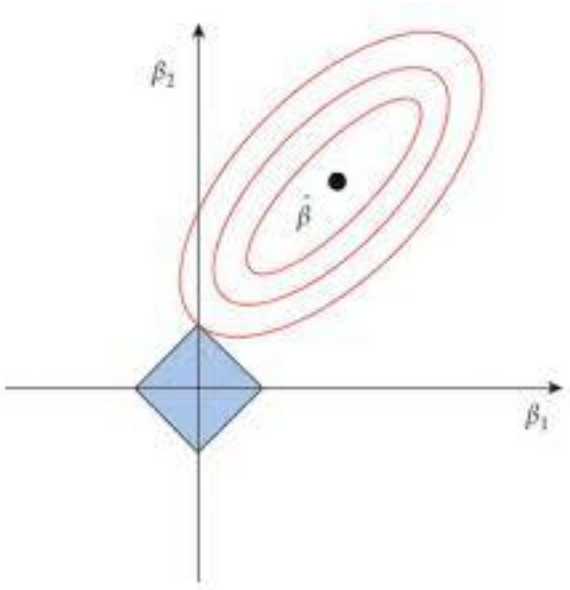

FIGURE 2.3. LASSO Regularized Regression

From the Bayesian perspective, Tibshirani states that LASSO regression is simply a case of linear regression where the coefficients each follow a Laplace distribution prior given by the probability density function (PDF)

$$f(x|\mu, b) = \frac{1}{2b}\exp(-\frac{|x-\mu|}{b})$$

for the support $x \in \mathbb{R}$. A graph of this function is provided in Figure 2.4.

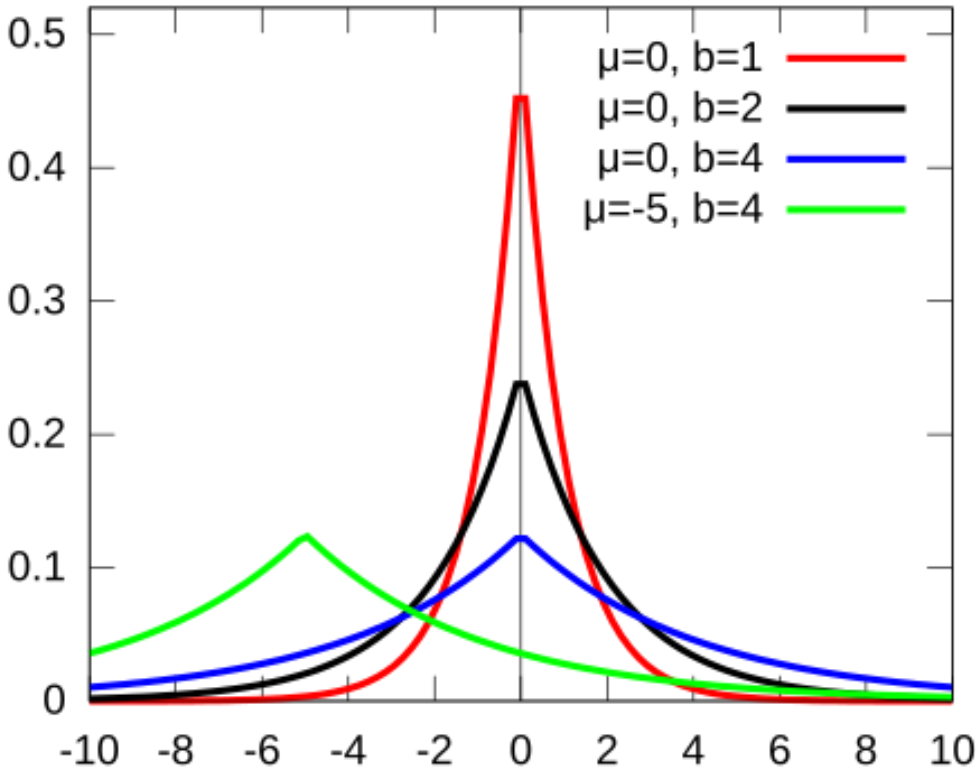

FIGURE 2.4. Laplace Distribution for specific values of μ and b

This formulation is intuitive as well since the Laplace distribution has a sharp peak at zero so all of the probability density is located closer to zero more than the Normal distribution (Tibshirani 1996). While the principal investigation in this paper utilizes LASSO in the regression context to perform feature selection, it is important to note that LASSO can also be extended to other applications such as Generalized Linear Models and Proportional Hazards Models.

Proper Orthogonal Decomposition

The Proper Orthogonal Decomposition (POD) algorithm is a dimensionality reduction method that numerically compresses data into smaller components while preserving the information of the original data. It is a continuous version of the discretized Singular Value Decomposition (SVD) method and achieves the same purpose in dimensionally reducing data matrices of size $n \times m$ where $n, m \in \mathbb{N}$. This method also applies to square matrices of size $n \times n$ as well. As mentioned previously in Chatterjee's paper (2000) on POD, a given matrix can be decomposed via orthogonal decomposition along the principal components of the matrix. Once the original data matrix is broken down, it can extrapolate approximations and "reconstruct" the original data matrix with r low-rank modes. In this section, we will provide an example of POD on a matrix of size 5×5 and analyze how POD can be utilized to decompose and reconstruct an approximation of the original matrix with 2 modes.

Assume that we have the following 5×5 matrix:

$$A_{5\times5} = \begin{pmatrix} 2 & 3 & 1 & 5 & 4 \\ 4 & 6 & 2 & 10 & 8 \\ 1 & 1 & 1 & 2 & 2 \\ 3 & 4 & 2 & 6 & 5 \\ 5 & 7 & 3 & 12 & 9 \end{pmatrix}$$

By POD, we can decompose the matrix A into the form $U\Sigma V^T$ where each matrix is of size 5×5 for this example. Note that the matrices U and V are orthogonal matrices, so their columns are orthonormal and have unit length. By using computer-assisted numerical methods, we can find the decomposition of the matrix A and attempt to reconstruct an approximation of it using 2 of the modes from U, the matrix consisting of the left singular vectors. For convenience, a plot of the singular values from Σ are provided in addition to the approximations in Figure 2.5 while the approximations are given in Figure 2.6. It can be observed that with 2 modes, our reconstruction

of matrix *A* is quite close to the original. One can increase the number of modes to use for the approximation for higher accuracy; however, this may affect the computational efficiency and cost of the algorithm, so caution should be exercised appropriately given the computer's limitations. The full complete source codes for this example is provided in the appendices of this paper.

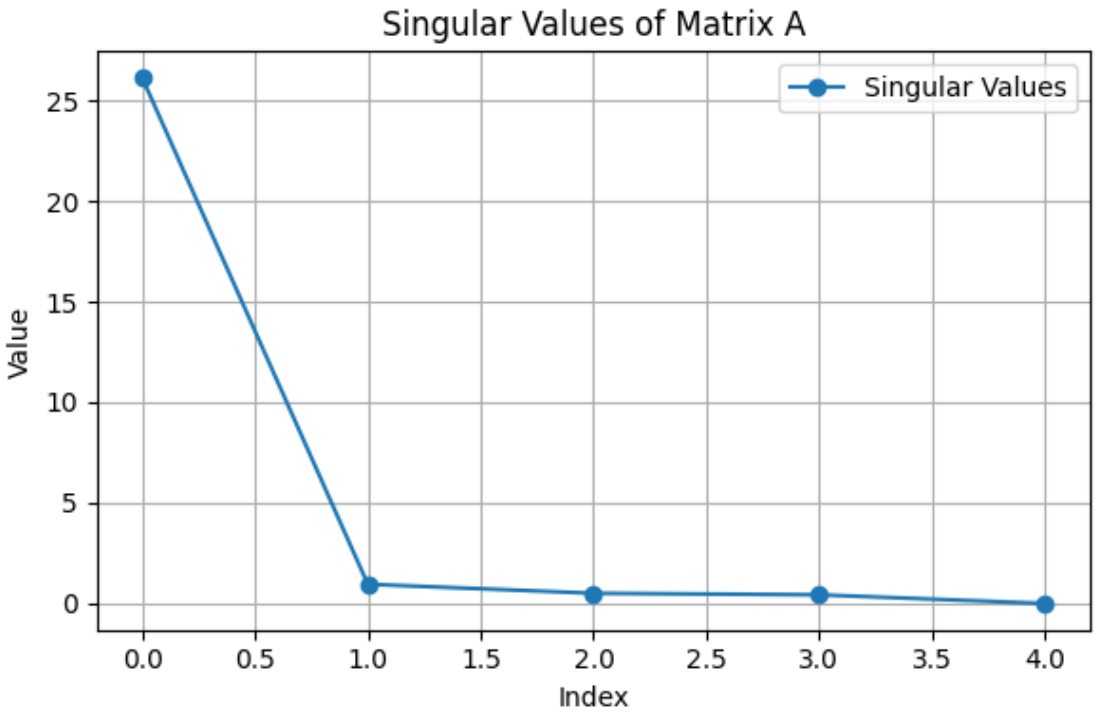

FIGURE 2.5. Singular Values Plot for Matrix A

FIGURE 2.6. 2 Mode Approximations of Matrix A

CHAPTER 3

ANALYSIS OF PDE MODEL A: AN OBJECT MOVING FROM LEFT TO RIGHT

Sparse Regression Analysis and LASSO Framework for PDE Model A

The aim of this chapter is to introduce the theoretical and practical framework of sparse regression and LASSO methods. More specifically, the results from the sparse regression model will extract the appropriate coefficients while the LASSO model will perform feature selection in order to provide the relevant intuition for the remaining partial derivatives. We will call this the physics-driven intuition model while the sparse regression via the p-value feature selection method will be known as the data-driven intuition model. The data we will be implementing our methodology on will be synthetic data with a targeted object moving from the left to the right. As mentioned in the current literature, the spatial derivatives will be estimated using iterative finite difference schemes and the time derivative will be estimated with both a finite difference scheme and a forward Euler's method to perform integration and develop a solution $u(x,t)$ from the partial differential equation u_t.

Data Preprocessing and Context

The original dataset was synthetically processed through the Microsoft Powerpoint application. To mimic frame dissection from a standard .mp4 video file, each Powerpoint slide represents a single frame. Out of simplicity, we simulated a circle of radius one inch moving from the left side of the slide page to the right side over the sequence of fifty frames. This circle will represent our targeted object in which we desire to find data-driven and physics-driven partial differential equation models within this system. Since our methodology will assume no strict boundary conditions, our circle starts one centimeter away from the left border of the first

slide and ends one centimeter away from the right border of the fiftieth slide. It was carefully ensured that in each intermediary slide that the circle moved from the left to the right uniformly across equidistant spacing between slides.

Once the slides have been prepared, each slide was saved and converted into a .jpg file. Afterwards, a folder in the personal computer directory contained all images denoted as *image_1.jpg* to *image_50.jpg*. Some samples of the raw frame data have been provided for the reader's convenience.

The frames were then imported into Python as a dataframe using the Pandas library's *read_csv* method and the SK Image library's IO parser. While our images are now in matrix form, one issue that was encountered was that the circles in each of the frames had rough edges. To remedy this, we implemented a Gaussian filter with dispersion parameter $\sigma = 55$ to smooth out the edges of the circle and consequently introduce naturally occurring noise in the data. It is important to note that the higher the dispersion parameter, the greater amount of noise will be introduced to the data.

The next step in the preprocessing stage was to encode the data so that 1 corresponds to pure black and 0 corresponds to pure white in the frames' data matrices. This implies that pure gray colors are indicated by 0.5 and we would have our matrices contain elements that lie in between 0 and 1 on a continuous scale. A few examples of this preprocessing step are given for demonstrative purposes along with their corresponding data matrices.

This concludes the data and preprocessing stage and we now continue to the implementation of the sparse regression and LASSO framework in the methodology section of this chapter.

Methodology

Once all images were imported into the Python notebook environment with their corresponding numerical labels, we computed the gradients using Numpy's gradient method which implements a finite difference scheme to approximate the spatial derivatives. Obviously, our circle or object is only moving from the left to the right over the duration of the fifty frames, so only the first-order derivatives and second-order derivatives with respect to x and y were computed using the np.gradient method. Our target function u will represent the original set of preprocessed image matrices. The time derivative u_t is approximated as the difference between each pair of images. Therefore, the following finite difference schemes with frame step size $h = 1$ were used:

$$u_x = \frac{u(x)^{(i+1)} - u(x)^{(i)}}{2h}$$

$$u_y = \frac{u(y)^{(i+1)} - u(y)^{(i)}}{2h}$$

$$u_{xx} = \frac{u(x)^{(i+1)} - 2u(x)^{(i)} + u(x)^{(i-1)}}{h^2}$$

$$u_{yy} = \frac{u(y)^{(i+1)} - 2u(y)^{(i)} + u(y)^{(i-1)}}{h^2}$$

$$u_t = u^{(i+1)} - u^{(i)}$$

where $i + 1$ denotes the next iteration, i denotes the current iteration, and $i - 1$ denotes the previous iteration. After computing the first and second order derivatives numerically with respect to x, y, and t, we then concatenate all of the data matrices into one single dataframe. hence our dataset prior to regression model fitting consists of the six features: u, dx, dy, dxx, dyy, and dfudt. u is the original images, dx is the first derivative of u with respect to x, dy is the first

derivative of u with respect to y, dxx is the second derivative of u with respect to x, dyy is the second derivative of u with respect to y, and dfudt is the first derivative of u with respect to t.

Due to the limited availability of this synthetic dataset, the first forty-five images have been set aside as part of the training set and images forty-six to fifty have been set aside for the testing set for model validation purposes after model fitting. In this dataframe, the data is separated between the response dfudt and the other features which will be part of the design matrix. A sparse regression model via the Ordinary Least Squares algorithm is fitted onto this data and the following results are outputted.

To extract the coefficients for our data-driven partial differential equation, we must take a look at the p-values and discard any features that have an associated p-value greater than the $\alpha = 0.05$ significance level. From the initial regression results in the above figure, we keep u, dx, and dxx while discarding dy and dyy. One additional observation is that dy and dyy are perfectly correlated and that their coefficients are 1.297×10^5 and -1.297×10^5 respectively, so there are some signs of multicollinearity between the two features. Since their p-values are way above the significance level, we have sufficient evidence and justification to drop these from the model. Running the OLS regression algorithm a second time with the irrelevant features dropped from the dataframe confirm the best coefficients for the model. The R^2 goodness of fit coefficient for the data-driven regression model is 0.914 which indicates a strong, positive correlation between the spatial and time derivatives. The corresponding OLS regression output in Figure 3.1 implies that we keep the features u, u_x, and u_{xx} and their coefficients since the p-values for these features are less than the 0.05 significance level.

	coef	std err	t	P>|t|	[0.025	0.975]
x1	-0.0014	1.51e-05	-94.544	0.000	-0.001	-0.001
x2	-13.3337	0.001	-1.18e+04	0.000	-13.336	-13.331
x3	1.297e+05	1.3e+09	9.97e-05	1.000	-2.55e+09	2.55e+09
x4	95.2591	0.074	1288.315	0.000	95.114	95.404
x5	-1.297e+05	1.3e+09	-9.97e-05	1.000	-2.55e+09	2.55e+09

Omnibus:	3642833.020	Durbin-Watson:	0.000
Prob(Omnibus):	0.000	Jarque-Bera (JB):	393422392.159
Skew:	-0.039	Prob(JB):	0.00
Kurtosis:	29.481	Cond. No.	1.49e+14

FIGURE 3.1. Data-Driven Regression for Model A

Hence, our data-driven partial differential equation model is:

$$u_t = -0.0014u - 13.3337u_x + 95.2591u_{xx}$$

Now for the physics-driven model, our intuition for the "ideal" model will be supported by running the processed data through the Least Absolute Shrinkage and Selection Operator (LASSO) and rerunning the OLS regression algorithm without the features dropped by LASSO. Running this algorithm initially with a deterministic penalty parameter, say 0.1, none of the irrelevant features have been dropped from the regression model. Therefore, we will utilize Sci-Kit Learn's *LassoCV* module which implements LASSO and k-fold cross validation to perform hyperparameter tuning for the best penalty. With the best penalty being 0.001 from performing 5-fold cross validation, rerunning LASSO on the data returns a coefficient for the dx feature, or u_x. We drop the other coefficients since LASSO drives all the other coefficients to zero.

```
LASSO Coefficients: [-0.00925312 -0.         -0.          0.         -0.        ]
```

FIGURE 3.2. LASSO Output for Model A

For precision, we rerun the OLS method with all other features dropped. The Python output in Figure 3.3 provides the updated coefficient for the surviving feature.

	coef	std err	t	P>|t|	[0.025	0.975]
dx	-13.3348	0.001	-1.08e+04	0.000	-13.337	-13.332

FIGURE 3.3. Physics-Driven Regression Output for Model A

According to the output, our physics-driven partial differential equation is:

$$u_t = -13.3348u_x$$

For the physics-driven model, the R^2 has decreased to 0.896. This intuition makes sense since our model is moving only from the left to the right, so the presence of the second derivative with respect to x may provide too much diffusion and hence introduce noisy predictions from the data. Furthermore, this supports the fact that a regression model with a higher R^2 does not necessarily guarantee a "better model" than another model with a lower R^2 coefficient. In the next section, we will compare and contrast the predicted reconstructions of the test data images and provide an error analysis from both a visual and a statistical perspective.

Discovered Governing Equations and Interpretation of Results

With equations 3.6 and 3.7 discovered, the primary goal of obtaining these "black-box" models are to reconstruct predictions in the form of images using these models. While modern autoencoders and generative adversarial networks are popular algorithms in the machine learning paradigm, one detriment to them is that they can only produce predictions and new data with little to no interpretability. More precisely, there are no explicit mathematical models that can explain object trajectories over time in the newly generated data. Our framework addresses this shortcoming by providing a data-driven model and a physics-driven model that can describe an object's trajectory in a physical system. Furthermore, we claim that we can also extrapolate new data and reconstruct images using these partial differential equations.

To achieve this, it is necessary to create a modified image-iterative version of the Forward Euler method algorithm in the code. This is to ensure that as we generate new extrapolations for any time t outside of the number of frames in our dataset that our cumulative error is minimized. Therefore, the following algorithm scheme was initialized in the pseudocode below. The reader may find the full code for this algorithm in the appendices.

One note to make is that during the initial reconstruction of the predicted images, the images were appearing to be lighter than the previous predicted images in the iterative Euler method. This implies that the current pixels in the newly generated data were accumulating the previous pixel's error. This is a consequence of utilizing a forward Euler algorithm since the error and image noise accumulate, and hence the future predictions exponentially deviate from the actual ground truth images. To address this issue, we utilize a Gaussian Filter on each predicted image between each iteration. By applying this denoising mechanism to Algorithm 2 below, we can ensure that the noise and error are minimally accumulated.

Algorithm 2 Modified Image-iterative Euler Method

$N \leftarrow$ steps
$u \leftarrow u_0$
images $\leftarrow u$
for i in $1, ..., N$ **do**
    $u_x \leftarrow \frac{u(x)^{(i+1)} - u(x)^{(i)}}{2h}$
    $u_{xx} \leftarrow \frac{u(x)^{(i+1)} - 2u(x)^{(i)} + u(x)^{(i-1)}}{h^2}$
    $u_{yy} \leftarrow \frac{u(y)^{(i+1)} - 2u(y)^{(i)} + u(y)^{(i-1)}}{h^2}$
    $u_t \leftarrow -0.0014u - 13.3337u_x + 95.2591u_{xx}$
    $u \leftarrow u + (dt * u_t)$
    $u \leftarrow gaussian\_filter(u, \sigma = 10)$
    images $\leftarrow u$
end for

Algorithm 2 can be used to discretely integrate each of the data matrices generated by the model u_t to obtain the predicted images u. Below, we provide a before-and-after analysis of our predictions in comparison to the images in the test dataset. The test set MSE which compares the extrapolations to the last five original ground-truth images for both the data-driven and physics-driven models are given in Table 3.1.

TABLE 3.1. Table of MSEs for Model A

Model Type	Image 1	Image 2	Image 3	Image 4	Image 5
Data-Driven	0.0	0.0024	0.0099	0.0221	0.0425
Physics-Driven	0.0	0.0024	0.0099	0.0221	0.0372

Interestingly, the test MSEs are very similar with the exception of the fifth test image, as the MSE for the data-driven model is much higher than the MSE for the physics-driven model. Therefore, we can conclude that the physics-driven PDE model generalizes the test image data better. The before images are the noisy predictions before the implementation of the Gaussian filter, and it can be seen that our predicted images are much better with this filtering mechanism in the after images as shown in Figures 3.6 and 3.7.

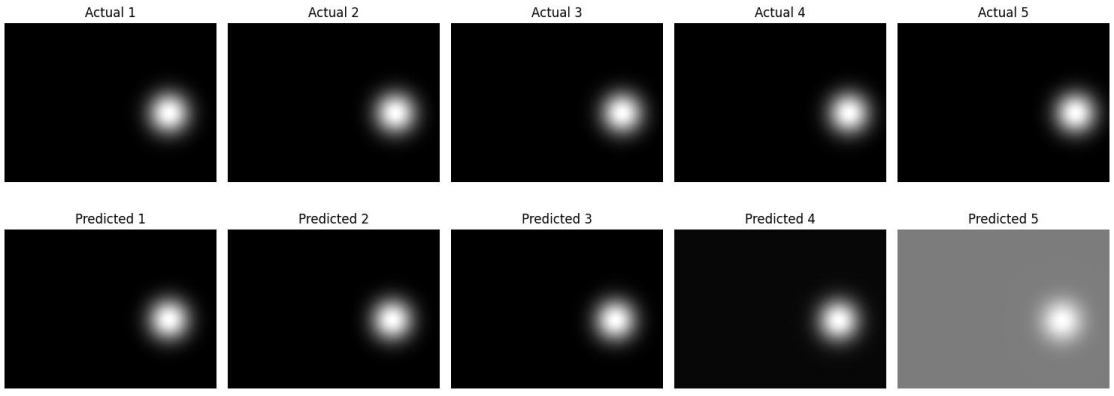

FIGURE 3.4. Data-Driven Prediction Extrapolations Before Denoising

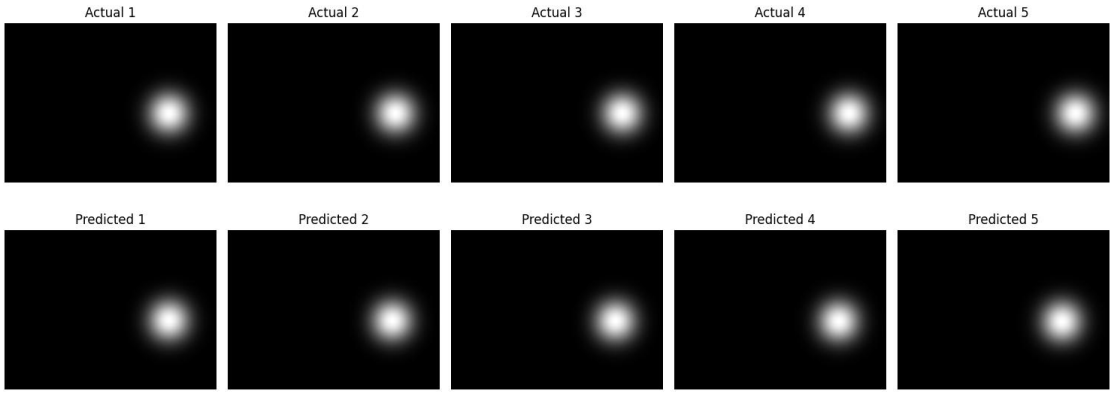

FIGURE 3.5. Data-Driven Prediction Extrapolations After Denoising

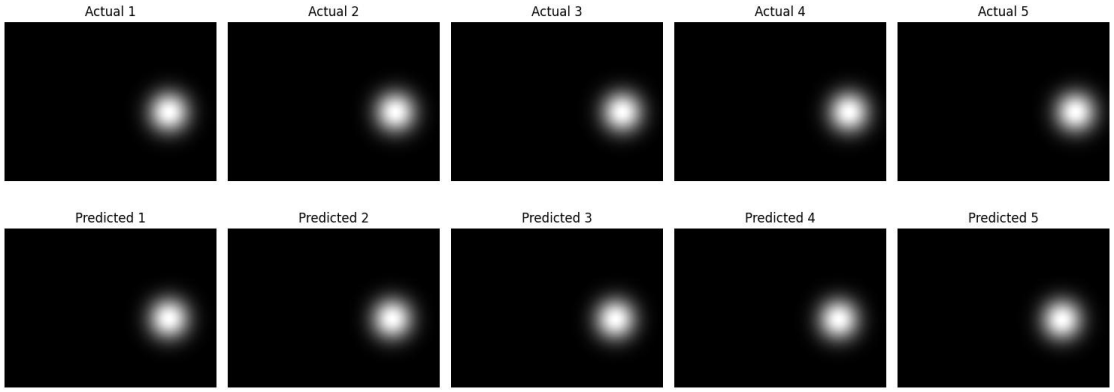

FIGURE 3.6. Physics-Driven Prediction Extrapolations Before Denoising

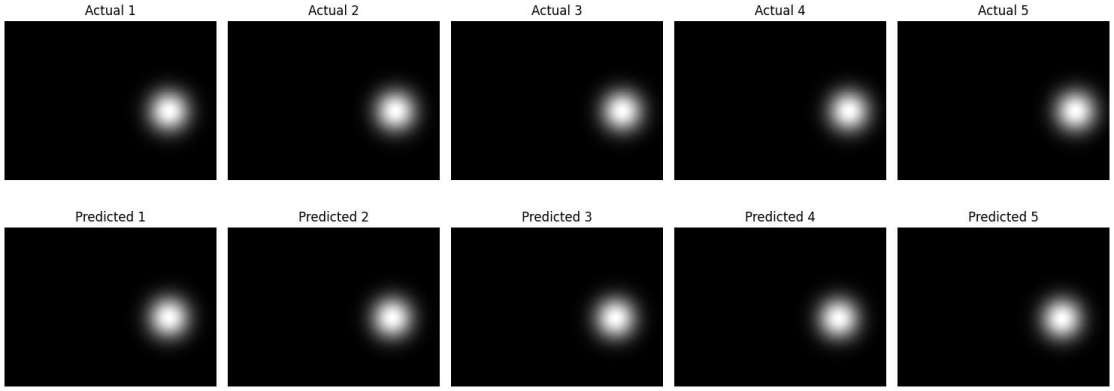

FIGURE 3.7. Physics-Driven Prediction Extrapolations After Denoising

CHAPTER 4

ANALYSIS OF PDE MODEL B: ANALYSIS OF OBJECT MOVING LEFT TO RIGHT AND EXPANDING

Sparse Regression Analysis and LASSO Framework for PDE Model B

The aim of this chapter is to once again introduce the theoretical and practical framework of sparse regression and LASSO methods. More specifically, the results from the sparse regression model will extract the appropriate coefficients while the LASSO model will perform feature selection in order to provide the relevant intuition for the remaining partial derivatives. We will call this the physics-driven intuition model while the sparse regression via the p-value feature selection method will be known as the data-driven intuition model. The data we will be implementing our methodology on will be synthetic data with a targeted object moving from the left to the right while expanding at an unknown constant rate. As mentioned in the current literature, the spatial derivatives will again be estimated using iterative finite difference schemes and the time derivative will be estimated with both a finite difference scheme and a forward Euler's method to perform integration and develop a solution $u(x,t)$ from the partial differential equation u_t.

Data Preprocessing and Context

The original dataset was synthetically processed through the Microsoft Powerpoint application. To mimic frame dissection from a standard .mp4 video file, each Powerpoint slide represents a single frame. Out of simplicity, we simulated a circle of radius one inch moving from the left side of the slide page to the right side over the sequence of fifty frames. In addition, the circle will be expanding at a constant rate. This circle will represent our targeted object in

which we desire to find data-driven and physics-driven partial differential equation models within this system. Since our methodology will assume no strict boundary conditions, our circle starts one centimeter away from the left border of the first slide and ends one centimeter away from the right border of the fiftieth slide. It was carefully ensured that in each intermediary slide that the circle moved from the left to the right uniformly across equidistant spacing between slides while making the expansion.

Once the slides have been prepared, each slide was saved and converted into a .jpg file. Afterwards, a folder in the personal computer directory contained all images denoted as *image_1.jpg* to *image_50.jpg*. Some samples of the raw frame data have been provided for the reader's convenience.

The frames were then imported into Python as a dataframe using the Pandas library's *read_csv* method and the SK Image library's IO parser. While our images are now in matrix form, one issue that was encountered was that the circles in each of the frames had rough edges. To remedy this, we implemented a Gaussian filter with dispersion parameter $\sigma = 55$ to smooth out the edges of the circle and consequently introduce naturally occurring noise in the data. It is important to note that the higher the dispersion parameter, the greater amount of noise will be introduced to the data.

The next step in the preprocessing stage was to encode the data so that 1 corresponds to pure black and 0 corresponds to pure white in the frames' data matrices. This implies that pure gray colors are indicated by 0.5 and we would have our matrices contain elements that lie in between 0 and 1 on a continuous scale. A few examples of this preprocessing step are given for demonstrative purposes along with their corresponding data matrices.

This concludes the data and preprocessing stage and we now continue to the implementation of the sparse regression and LASSO framework in the methodology section of this chapter.

Methodology

Once all images were imported into the Python notebook environment with their corresponding numerical labels, we computed the gradients using Numpy's gradient method which implements a finite difference scheme to approximate the spatial derivatives. Obviously, our circle or object is moving from the left to the right and expanding over the duration of the fifty frames, so we require the first-order derivatives and second-order derivatives with respect to x and y to be computed using the np.gradient method. In addition, u_{xy} will be approximated as well. By Clairaut's theorem, we will not need to calculate the converse u_{yx}. Clairaut's theorem is stated below for the reader's convenience. A more general version of Clairaut's theorem for partial derivatives for more than two variables can be found in Tao's real analysis text (2006).

Clairaut's Theorem: Let $f: X, Y \to Z$ be a function on the open region $\mathbb{R} \subset \mathbb{R}^2$. If f has continuous second-order partial derivatives that exist at every point in $\mathbb{R}$, then $f_{xy} = f_{yx}$.

Our target function u will represent the original set of preprocessed image matrices. The time derivative u_t is approximated as the difference between each pair of images. Therefore, the following finite difference schemes with frame step size $h = 1$ were used:

$$u_x = \frac{u(x)^{(i+1)} - u(x)^{(i)}}{2h}$$

$$u_y = \frac{u(y)^{(i+1)} - u(y)^{(i)}}{2h}$$

$$u_{xx} = \frac{u(x)^{(i+1)} - 2u(x)^{(i)} + u(x)^{(i-1)}}{h^2}$$

$$u_{yy} = \frac{u(y)^{(i+1)} - 2u(y)^{(i)} + u(y)^{(i-1)}}{h^2}$$

$$u_{xy} = \frac{u(x,y)^{(i+1,j+1)} - u(x,y)^{(i+1,j-1)} - u(x,y)^{(i-1,j+1)} + u(x,y)^{(i-1,j-1)}}{4h}$$

$$u_t = u^{(i+1)} - u^{(i)}$$

where $i + 1$ denotes the next iteration, i denotes the current iteration, and $i - 1$ denotes the previous iteration. After computing the first and second order derivatives numerically with respect to x, y, and t, we then concatenate all of the data matrices into one single dataframe. hence our dataset prior to regression model fitting consists of the six features: u, dx, dy, dxx, dxy, dyy, and dfudt. u is the original images, dx is the first derivative of u with respect to x, dy is the first derivative of u with respect to y, dxx is the second derivative of u with respect to x, dxy is the second derivative of u with respect to y of the first derivative of u with respect to x. dyy is the second derivative of u with respect to y, and dfudt is the first derivative of u with respect to t.

Due to the limited availability of this synthetic dataset, the first forty-five images have been set aside as part of the training set and images forty-six to fifty have been set aside for the testing set for model validation purposes after model fitting. In this dataframe, the data is separated between the response dfudt and the other features which will be part of the design matrix. A sparse regression model via the Ordinary Least Squares algorithm is fitted onto this data and the following results are outputted.

To extract the coefficients for our data-driven partial differential equation, we must take a look at the p-values and discard any features that have an associated p-value greater than the $\alpha =$

0.05 significance level. From the initial regression results in Figure 4.1, we keep u, dx, dxx, and dyy while discarding dy and dxy.

	coef	std err	t	P>|t|	[0.025	0.975]
x1	0.0321	8.47e-06	3787.074	0.000	0.032	0.032
x2	-10.9031	0.001	-1.12e+04	0.000	-10.905	-10.901
x3	4.728e+04	8.98e+08	5.26e-05	1.000	-1.76e+09	1.76e+09
x4	-3.3925	0.124	-27.409	0.000	-3.635	-3.150
x5	138.6948	0.085	1633.341	0.000	138.528	138.861
x6	-4.728e+04	8.98e+08	-5.26e-05	1.000	-1.76e+09	1.76e+09

FIGURE 4.1. Data-Driven Regression Output for Model B

Since their p-values are way above the significance level, we have sufficient evidence and justification to drop these from the model. Running the OLS regression algorithm a second time with the irrelevant features dropped from the dataframe confirm the best coefficients for the model. Hence, our data-driven partial differential equation model is:

$$u_t = 0.0321u - 10.9031u_x - 3.3925u_{xy} + 138.6948u_{xx} + 0.0031u_{yy}$$

Now for the physics-driven model, our intuition for the "ideal" model will be supported by running the processed data through the Least Absolute Shrinkage and Selection Operator (LASSO). Running this algorithm initially with a deterministic penalty parameter, say 0.1, none of the irrelevant features have been dropped from the regression model. Therefore, we will utilize Sci-Kit Learn's *LassoCV* module which implements LASSO and k-fold cross validation to perform hyperparameter tuning for the best penalty. With the best penalty being 0.001 from performing 5-fold cross validation, rerunning LASSO on the data returns a coefficient for the dx feature, or u_x.

```
LASSO Coefficients: [ 0.0173299 -0.        0.       -0.        0.        0.      ]
```

FIGURE 4.2. LASSO Output for Model B

	coef	std err	t	P>|t|	[0.025	0.975]
u	0.0321	8.47e-06	3787.074	0.000	0.032	0.032
dx	-10.9031	0.001	-1.12e+04	0.000	-10.905	-10.901
dxy	-3.3925	0.124	-27.409	0.000	-3.635	-3.150
dxx	138.6948	0.085	1633.341	0.000	138.528	138.861
dyy	0.0031	0.001	3.165	0.002	0.001	0.005

FIGURE 4.3. Physics-Driven Regression output for Model B

For precision, we rerun the OLS method with all other features dropped as done in Figure 4.3, and so our physics-driven partial differential equation is:

$$u_t = 0.0321u - 10.9031u_x + 138.6570u_{xx} + 0.0031u_{yy}$$

This intuition makes sense since our model is moving only from the left to the right, so the presence of the second derivative with respect to x may provide too much diffusion and hence introduce noisy predictions from the data. In the next section, we will compare and contrast the predicted reconstructions of the test data images and provide an error analysis from both a visual and a statistical perspective.

Discovered Governing Equations and Interpretation of Results

With the equations discovered, the primary goal of obtaining these "black-box" models are to reconstruct predictions in the form of images using these models as similarly done in chapter 3. We again claim that we can extrapolate new data and reconstruct images using these partial differential equations.

To achieve this, it is necessary to create a modified image-iterative version of the Forward Euler method algorithm in the code. This is to ensure that as we generate new extrapolations for any time t outside of the number of frames in our dataset that our cumulative

error is minimized. Therefore, the following algorithm scheme was initialized in the pseudocode below. The reader may find the full code for this algorithm in the appendices.

One note to make is that during the initial reconstruction of the predicted images, the images were appearing to be lighter than the previous predicted images in the iterative Euler method. This implies that the current pixels in the newly generated data were accumulating the previous pixel's error. This is a consequence of utilizing a forward Euler algorithm since the error and image noise accumulate, and hence the future predictions exponentially deviate from the actual ground truth images. To address this issue, we utilize a Gaussian Filter on each predicted image between each iteration. By applying this denoising mechanism to the algorithm, we can ensure that the noise and error are minimally accumulated.

Algorithm 3 Modified Image-iterative Euler Method

$N \leftarrow$ steps
$u \leftarrow u_0$
images $\leftarrow u$
for i in $1, ..., N$ **do**
  for j in $1, ..., N$ **do**
    $u_x \leftarrow \frac{u(x)^{(i+1)} - u(x)^{(i)}}{2h}$
    $u_{xx} \leftarrow \frac{u(x)^{(i+1)} - 2u(x)^{(i)} + u(x)^{(i-1)}}{h^2}$
    $u_{yy} \leftarrow \frac{u(y)^{(i+1)} - 2u(y)^{(i)} + u(y)^{(i-1)}}{h^2}$
    $u_{xy} = \frac{u(x,y)^{(i+1,j+1)} - u(x,y)^{(i+1,j-1)} - u(x,y)^{(i-1,j+1)} + u(x,y)^{(i-1,j-1)}}{4h}$
    $u_t \leftarrow 0.0321u - 10.9031u_x - 3.3925u_{xy} + 138.6948u_{xx} + 0.0031u_{yy}$
    $u \leftarrow u + (dt * u_t)$
    $u \leftarrow gaussian\_filter(u, \sigma = 55)$
    images $\leftarrow u$
  end for
end for

Algorithm 3 can be used to discretely integrate each of the data matrices generated by the model u_t to obtain the predicted images u. Below, we provide a before-and-after analysis of our predictions in comparison to the images in the test dataset. The R^2 goodness of fit coefficients for the data-driven and physics-driven models are both 0.878, indicating a positive, strong correlation between the spatial and time derivatives, a similar result to the analysis of model A in chapter 3. The MSEs for model B are presented in Table 4.1.

TABLE 4.1. Table of MSEs for Model B

Model Type	Image 1	Image 2	Image 3	Image 4	Image 5
Data-Driven	0.0	0.0158	0.0501	0.1161	0.1974
Physics-Driven	0.0	0.0158	0.0501	0.1161	0.1974

The test MSEs for both the data-driven and physics-driven PDE models are the same for model B's data, however it is evident that the MSE cumulatively increases over time in both models. The before images are the noisy predictions before the implementation of the Gaussian filter, and it can be seen that our predicted images are much better with this filtering mechanism.

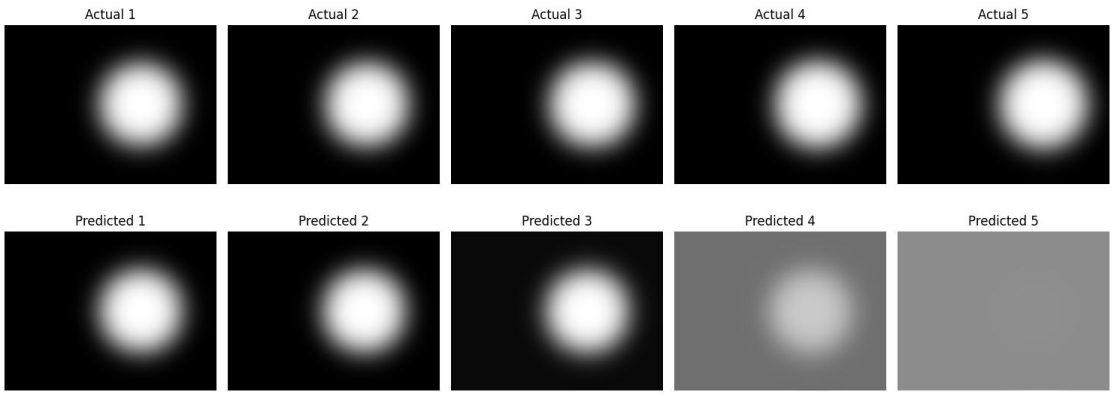

FIGURE 4.4. Prediction Extrapolations Before Gaussian Filter Denoising

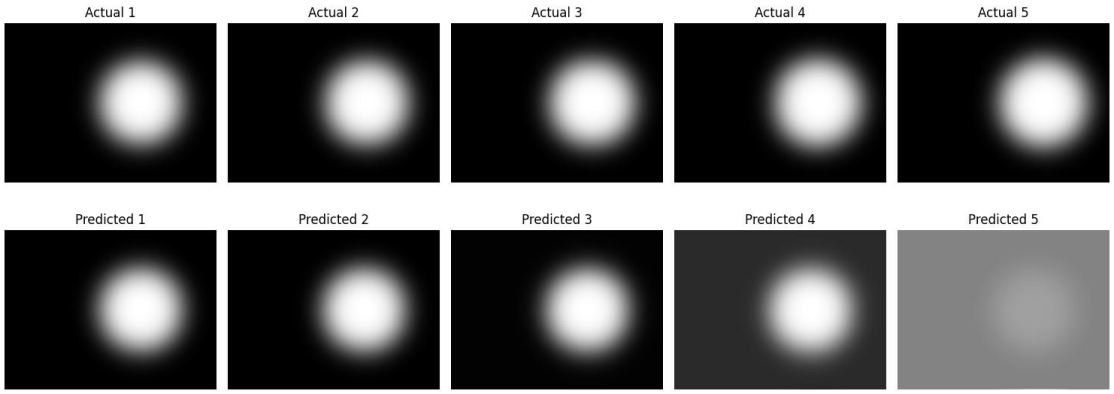

FIGURE 4.5. Prediction Extrapolations After Gaussian Filter Denoising

As shown in Figure 4.5, it can be observed that the predicted image extrapolations above that were produced from the data-driven partial differential equation model accumulated noise even after applying a Gaussian filter for denoising. However, this may be an indication that we should consider the physics-driven model which may provide more promising results due to its intuition. This turns out to be the case as after applying the same iterative denoising algorithm with the physics-driven model, the extrapolations accumulate less noise and the error between the actual and predicted images are minimized. The physics-driven extrapolations before and after the application of Gaussian filter denoising is given in Figures 4.6 and 4.7 respectively.

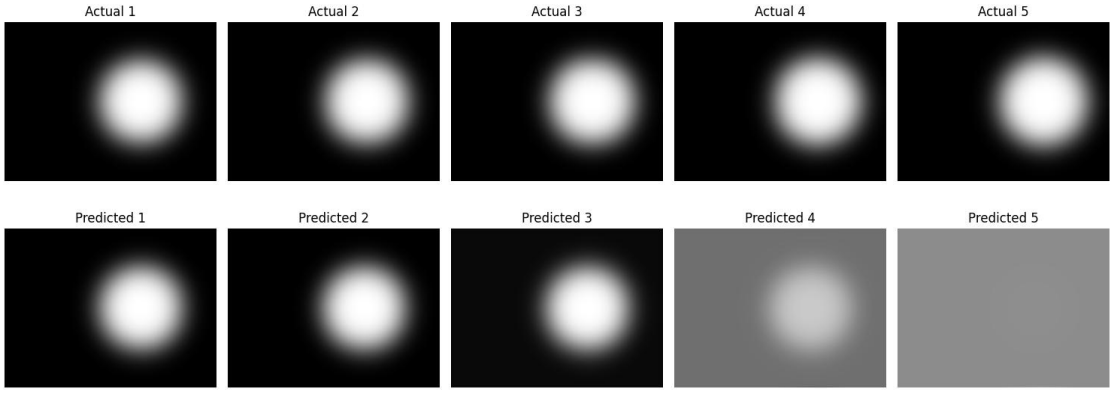

FIGURE 4.6. Prediction Extrapolations From Physics-Driven PDE Before Denoising

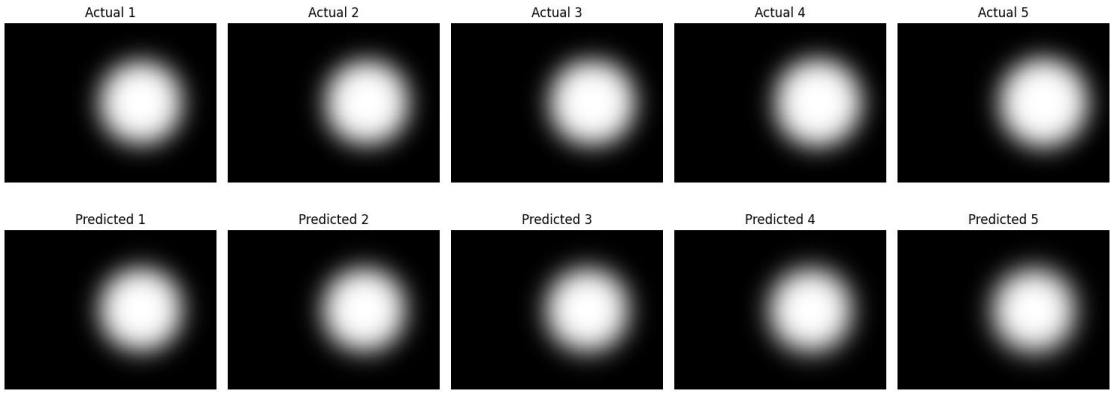

FIGURE 4.7. Prediction Extrapolations From Physics-Driven PDE After Denoising

CHAPTER 5

WATER DROPLET DATA ANALYSIS: APPLICATION OF DISCOVERING ODE SYSTEMS WITH PROPER ORTHOGONAL DECOMPOSITION

In this chapter, we consider a thirty-second clip of a water droplet falling into a body of water and split the dataset into image frames. Since these images are more complex due to the ripples caused by the surface tension between the droplet and the body of water, we will now consider utilizing the approach of applying the Proper Orthogonal Decomposition (POD) algorithm in order to extract the latent data-driven model. Instead of considering a partial differential equation model, our model will consist of a system of Ordinary Differential Equations to provide a data-driven model for the extrapolations generated by the POD algorithm.

Data Preprocessing of Water Droplet Video Clip

In the previous chapters of this paper, the datasets consisted of fifty images of a circle moving from the left to the right and expanding. Similarly, the thirty-second video clip had to be split into image frames that are equidistant with respect to seconds. In addition, each of the images needed to be transformed to grayscale from their original color. A few images before and after image preprocessing have been provided below for the reader's convenience in Figures 5.1 and 5.2.

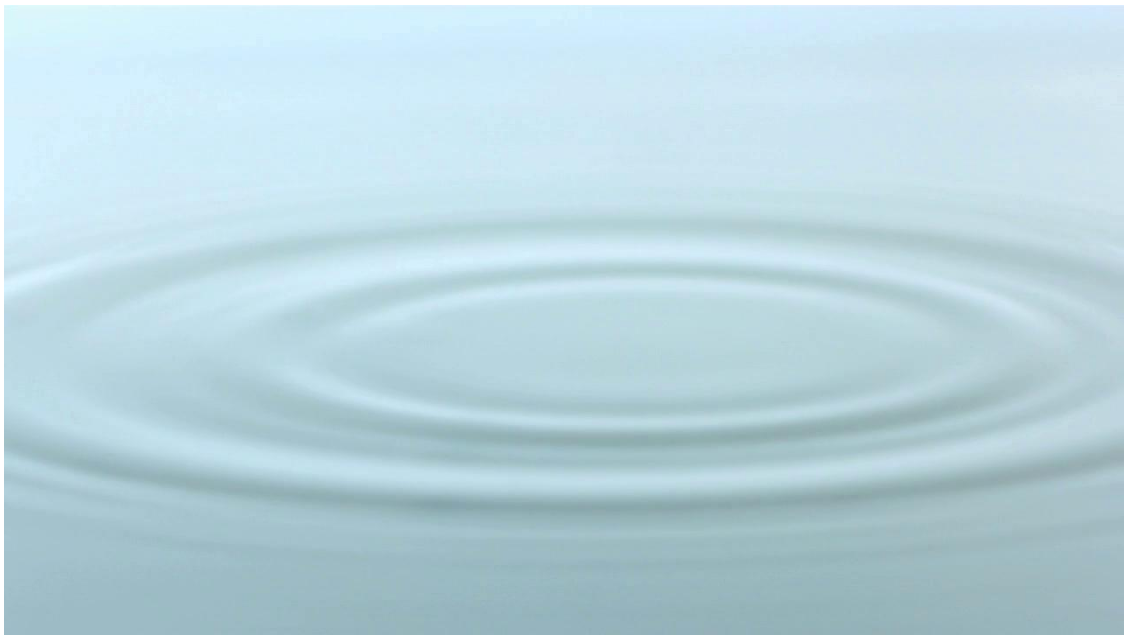

FIGURE 5.1. Original Frame Prior to Grayscale Transformation

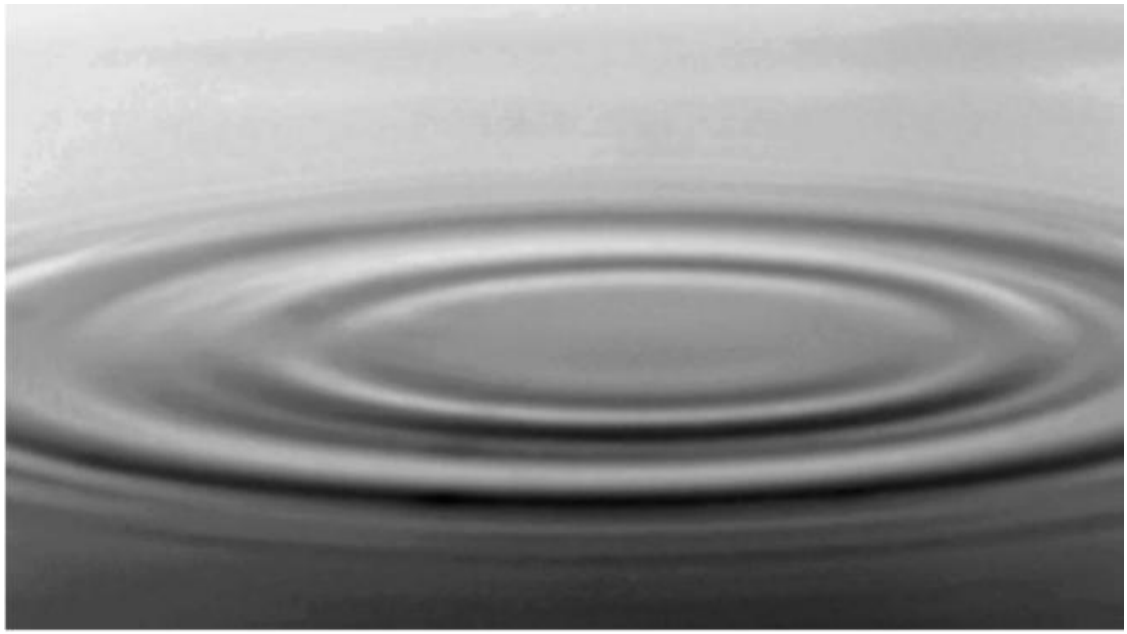

FIGURE 5.2. Frame After Grayscale Transformation

Proper Orthogonal Decomposition Framework

As outlined in the literature review, the Proper Orthogonal Decomposition algorithm is a continuous version of the Singular Value Decomposition algorithm. The POD framework that is utilized for the waterdrop frames will extract dominant spatial modes and their temporal

coefficients. By using this methodology, it is possible to describe a system of Ordinary Differential Equations along with their functional solutions that accurately models the spatio-temporal dynamics of the waterdrop. After utilizing the Proper Orthogonal Decomposition algorithm on the data, linear regression and LASSO will be used to extract the coefficients for functional components of the system as performed in the analysis presented in chapters 3 and 4.

Suppose we have the data matrix A. Recall from the literature review that POD can be used to decompose the matrix A in the following fashion.

$$A = U\Sigma V^T$$

In the above equation, the orthonormal spatial basis functions extracted from this algorithm represent the dominant spatial pattern modes, which are expressed in the columns of U. The time-dependent coefficients form the rows of ΣV^T, providing insight on how much each mode contributes at each snapshot in time. Therefore, the reconstructed data can be given by:

$$\tilde{u}(x, y, t) = \bar{u}(x, y, t) + \Sigma_{i=1}^{r} \phi_i a_i(t)$$

Here, the mean field $\bar{u}(x, y, t)$ is the mean of the original data matrix and $\tilde{u}(x, y, t)$ is the reconstructed data matrix. This equation also consists of a linear combination of i-th spatial modes and the i-th time coefficients and is the r-th low-rank approximation for image reconstruction. As performed before, our linear regression and LASSO step will extract the latent system of ODEs from the low-rank POD approximation of the original data matrix. The full code for the POD procedure is provided in the Appendix for the reader's convenience.

Discovered ODE System and Analysis of Results

Discovering the latent system of ODEs not only requires one to compute the reconstruction approximation via Proper Orthogonal Diagonalization, but also use LASSO

regularized regression. Due to computational constraints, this paper will only consider the first six modes produced by Proper Orthogonal Decomposition. A graphical visualization of the first six orthogonal modes are provided in Figure 5.3. A reconstructed image from using six POD modes for the twenty-fifth original snapshot is also provided in Figure 5.4. One observation that can be made is that POD seems to perform the reconstruction accurately with minimal visual differences from the source snapshot.

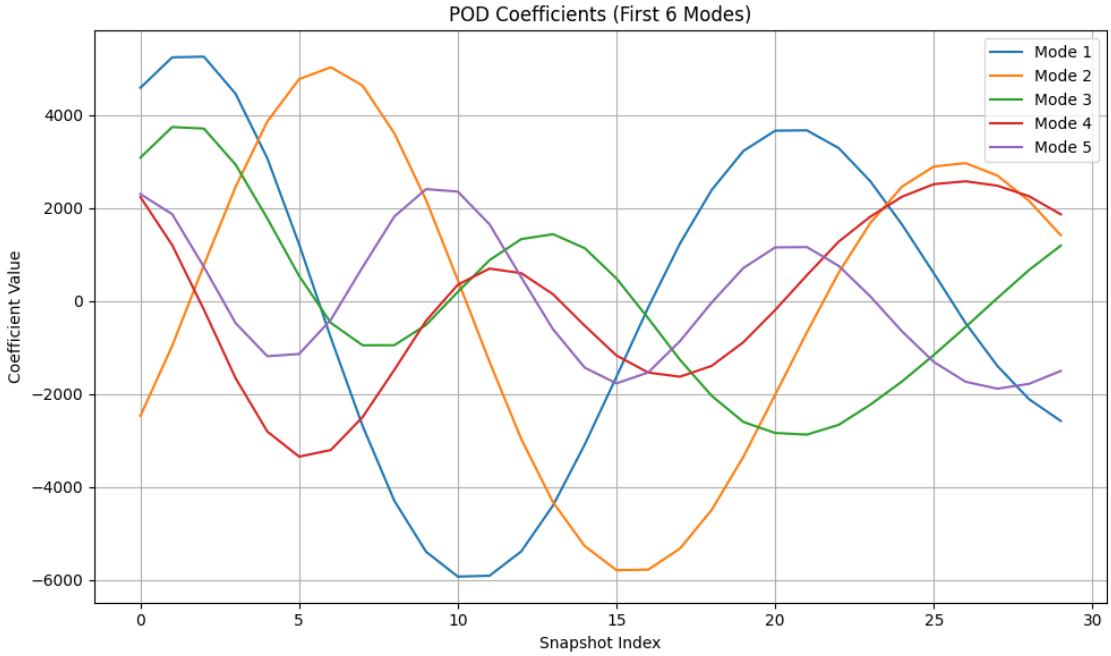

FIGURE 5.3. First Six Modes from POD

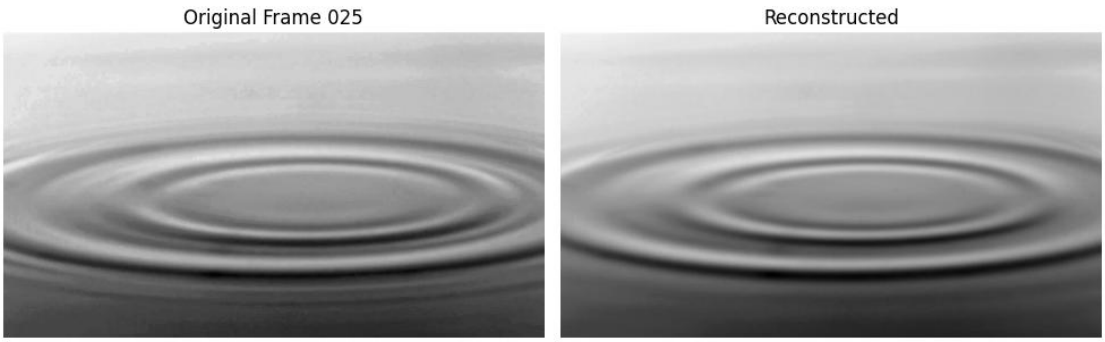

FIGURE 5.4. Original Frame and Reconstruction of Snapshot 25

The POD modes will now be transformed into a data matrix in order to perform linear regression via the OLS algorithm in Python, followed by running LASSO for coefficient importance. Here, the response will be set as the sixth mode and the rest of the modes will be treated as variables in the regression model. Therefore, we seek the system of the form:

$$\frac{d\boldsymbol{\alpha}}{dt} = A\boldsymbol{\alpha}(t)$$

where $\boldsymbol{\alpha}(t) = (\alpha_1(t), \ldots, \alpha_r(t))^T$. In our analysis, we have chosen to use 6 spatial modes from POD and thus we will have 6 linear ordinary differential equations in our system of ODEs. This can alternatively be written as a linear combination:

$$\frac{d\alpha_i}{dt} = \sum_{j=1}^{6} a_{ij}\, \alpha_j(t)$$

for $i = 1, \ldots, 6$. The a_{ij}s are the entries of the matrix A and the $a_j(t)$s are the elements of the time coefficient vector $\boldsymbol{\alpha}(t)$. It is rather difficult to find and solve this system, so we propose an alternative novel approach. Instead, harmonic regression and LASSO will be used to estimate the family of POD coefficient functions shown in Figure 5.3. These functions will be formed as a linear combination with the POD modes to provide an approximated solution to generate image reconstructions and future extrapolation forecasts. Since the functions will be of the parameter t, we can use this data-driven black-box solution to generate the extrapolations for any real-valued time step. This framework is very similar to the ones before for analyzing PDE models A and B.

To perform harmonic regression, we will assume that each of the POD mode functions can be expressed as a series of sines and cosines due to the harmonic nature of the POD mode coefficient graph. This means that

$$f_i(t) = a_0 + \sum_{j=1}^{k} a_j \left(\sin(j\pi t) + cos(j\pi t)\right)$$

provided that we have k terms left after regression. To best optimize the fit of the curve being trained on the POD mode data, LASSO regularized L_1 regression is utilized for feature selection. The algorithm will then perform optimization in L_1 to drop all insignificant coefficients that tend to zero and give precision on the surviving coefficients. Our final function approximations are given below.

$$f_1(t) = -0.0483\sin(2\pi t) + 0.1467\sin(3\pi t) + 0.0222\sin(4\pi t) + 0.0122\sin(5\pi t) \\ +0.0314\cos(\pi t) + 0.0930\cos(2\pi t) - 0.0104\cos(4\pi t)$$

$$f_2(t) = 0.0196\sin(\pi t) + 0.0877\sin(2\pi t) - 0.0356\sin(3\pi t) - 0.0265\sin(4\pi t) \\ -0.0114\sin(5\pi t) + 0.0341\cos(\pi t) + 0.0840\cos(2\pi t) - 0.1435\cos(3\pi t) - \\ 0.0164\cos(4\pi t) - 0.0102\cos(5\pi t)$$

$$f_3(t) = 0.0543\sin(\pi t) + 0.1506\sin(2\pi t) + 0.0414\sin(3\pi t) + 0.0308\sin(4\pi t) \\ +0.0148\sin(5\pi t) + 0.0119\sin(6\pi t) + 0.0519\cos(\pi t) + 0.0886\cos(3\pi t)$$

$$f_4(t) = -0.0231\sin(\pi t) - 0.0162\sin(2\pi t) - 0.0988\sin(3\pi t) + 0.0344\sin(4\pi t) \\ +0.0154\sin(5\pi t) + 0.1430\cos(2\pi t) + 0.0822\cos(3\pi t) + 0.0215\cos(4\pi t)$$

$$f_5(t) = 0.1140\sin(\pi t) - 0.0613\sin(2\pi t) - 0.0396\sin(3\pi t) + 0.0266\sin(4\pi t) \\ +0.0656\sin(5\pi t) + 0.0264\sin(6\pi t) + 0.0175\sin(7\pi t) + 0.0147\sin(8\pi t) \\ +0.0126\sin(9\pi t) + 0.0104\sin(10\pi t) + 0.0845\cos(\pi t) - 0.0220\cos(2\pi t) \\ -0.0302\cos(3\pi t) + 0.0679\cos(4\pi t) + 0.0111\cos(5\pi t)$$

$$f_6(t) = 0.0432\sin(\pi t) - 0.0179\sin(2\pi t) + 0.0846\sin(4\pi t) - 0.0920\sin(5\pi t) \\ -0.0328\sin(6\pi t) + 0.0130\cos(\pi t) - 0.0373\cos(4\pi t) - 0.0858\cos(5\pi t)$$

These functions have also been superimposed with the original POD mode coefficients for visual inspection. It can be observed that the harmonic regression and LASSO framework fitted very closely with the ground truth curve for POD modes 3, 4, and 5, while modes 1, 2, and 6 have the most error.

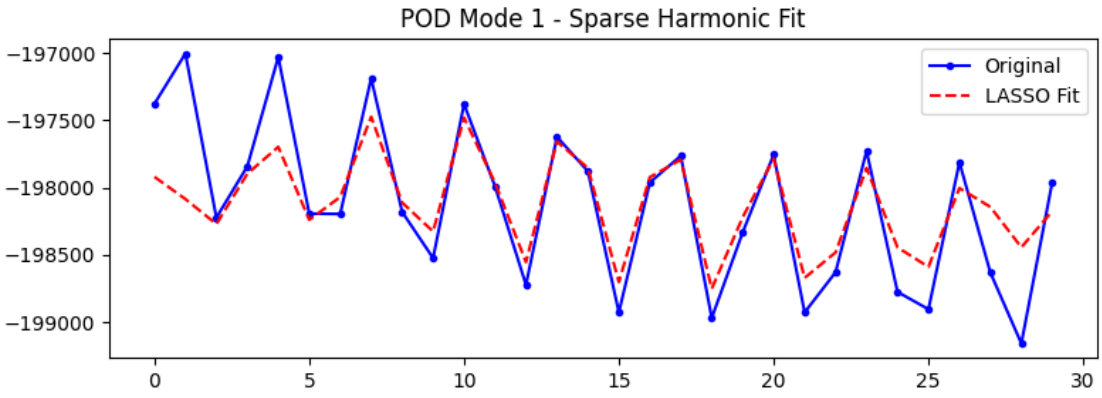

FIGURE 5.5. POD Mode 1 Harmonic Regression

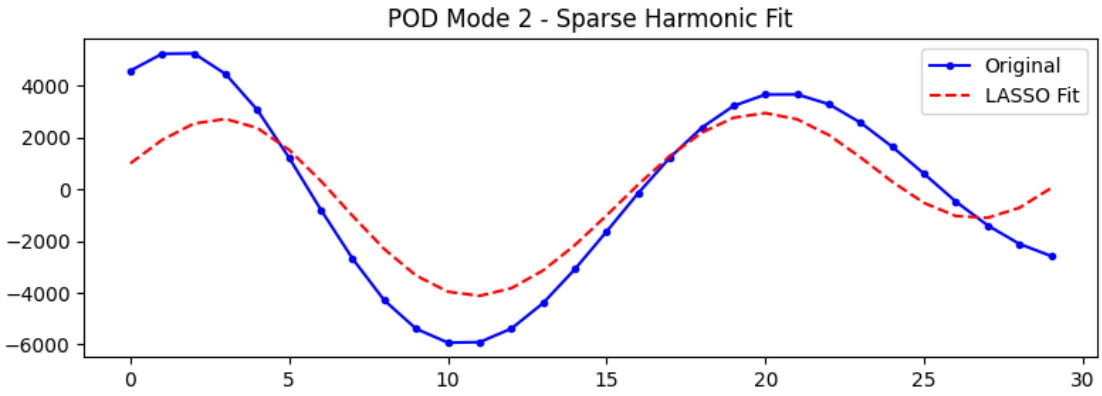

FIGURE 5.6. POD Mode 2 Harmonic Regression

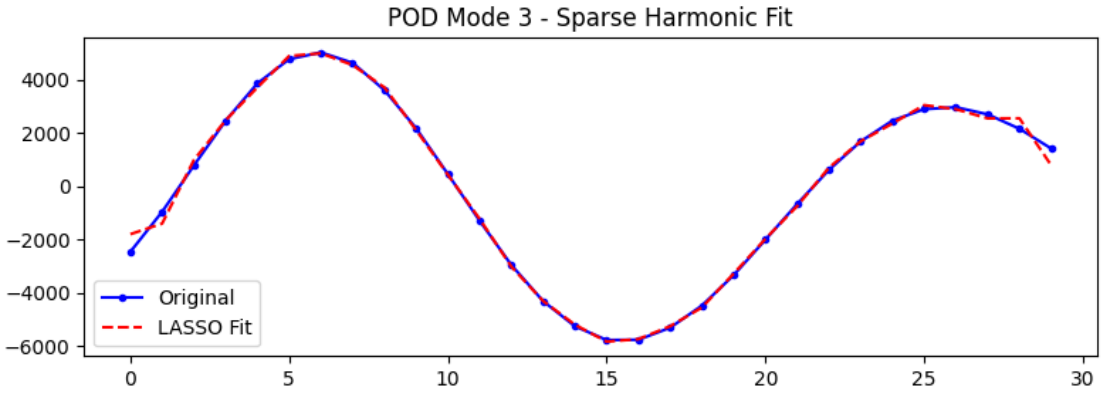

FIGURE 5.7. POD Mode 3 Harmonic Regression

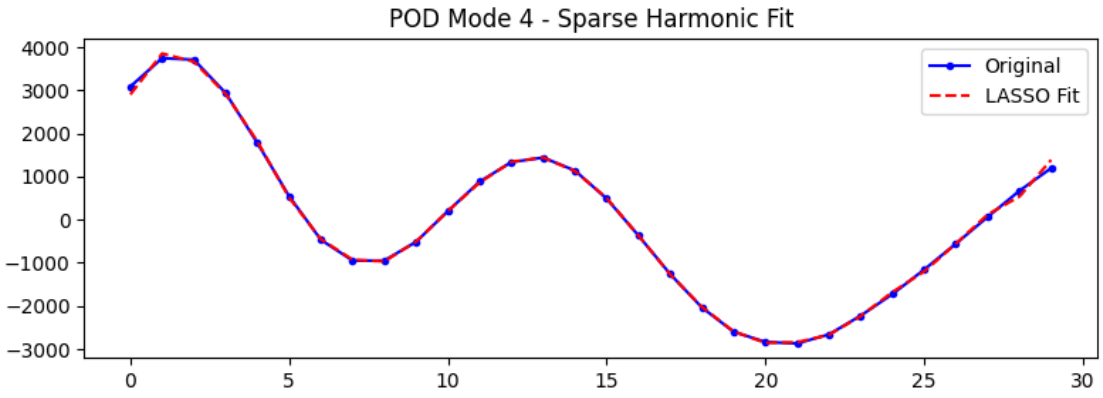

FIGURE 5.8. POD Mode 4 Harmonic Regression

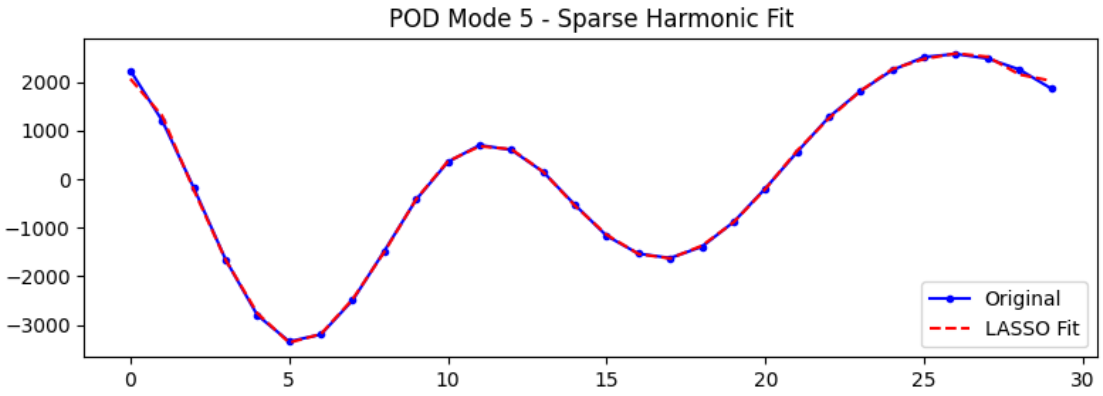

FIGURE 5.9. POD Mode 5 Harmonic Regression

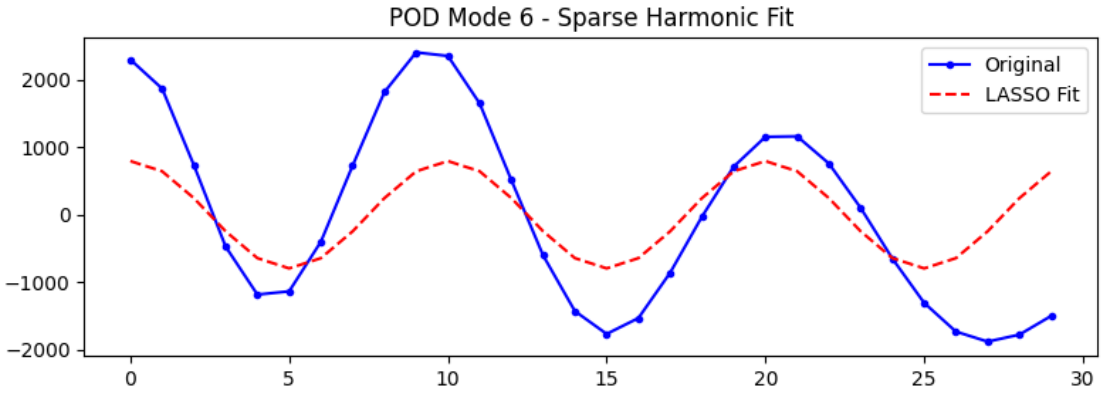

FIGURE 5.10. POD Mode 6 Harmonic Regression

After obtaining the function forms of the POD mode coefficients, we can now obtain the theoretical solution, denoted as $T(t)$, for the system of ODEs. This solution follows the form

$$T(t) = \sum_{i=1}^{6} f_i\,(t)\,\text{mode}_i$$

One can obtain the system of ODEs by differentiating the solution and repeating the regression analysis. Since there are six functions, the analysis must be conducted six times, with each i-th function as the response and the remaining functions as the feature variables. As a result, each derivative can be represented in terms of the other functions. Since this process can be rather difficult due to computational processing power, we will omit this as we are most interested in discovering the data-driven solution to this system. The solution $T(t)$ can now be utilized to reconstruct the original waterdrop snapshots and generate extrapolations for any real-valued time step in the future. The extrapolations are given in Figure 5.11. From visual inspection of the extrapolations, the concentric wave motion caused by the initial water droplet has been captured by the solution, and the test MSE is consistent with no major increases. To ensure image quality between the original snapshots and extrapolations, we computed the Structural Similarity scores using Sci-Kit Image's SSIM module. The SSIM ranges from 0 to 1, with 1 being perfectly structurally similar and 0 otherwise. The test MSEs and SSIM coefficients are given in Tables 5.1 and 5.2, and the SSIM scores hover around 0.98, indicating strong structural similarities between the original snapshots and extrapolations.

TABLE 5.1. Table of MSEs for Waterdrop Model

Image 1	Image 2	Image 3	Image 4	Image 5
29.549	27.217	28.064	29.878	27.261

TABLE 5.2. Table of SSIM Scores for Waterdrop Model

Image 1	Image 2	Image 3	Image 4	Image 5
0.9812	0.9813	0.9822	0.9815	0.9824

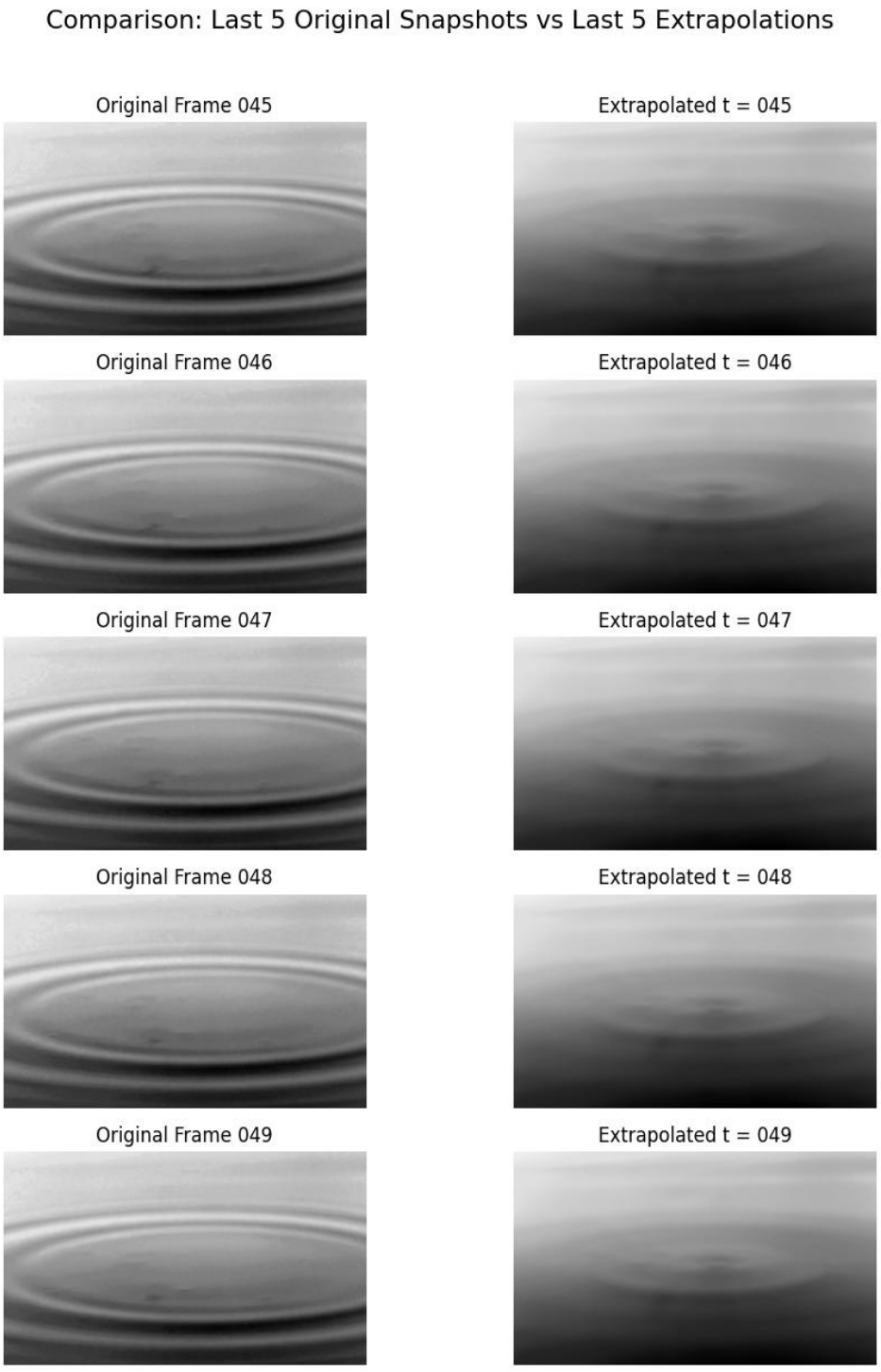

FIGURE 5.11. Model extrapolations

CHAPTER 6

CONCLUDING REMARKS AND FUTURE WORK

Summary

In this paper, we investigated the data-driven and physics-driven statistical framework of applying scientific machine learning methodologies to extract black-box models for physical systems. These models were used with numerical algorithms to reconstruct images and generate new extrapolated images from the learned data. The first partial differential equation model described the translation of a circular object moving from the left to the right and the second model described the translation and expansion of an object moving left to right while constantly diffusing. The synthetic image frame data was generated via Microsoft Powerpoint to mimic frame dissection from .mp4 videos. Our last model was a system of Ordinary Differential Equations learned from the orthogonal modes generated from Proper Orthogonal Decomposition and the Least Absolute Shrinkage and Selection Operator regularized regression algorithm. This was a culmination of the preceding chapters since the frames were spliced from a .mp4 video of a waterdrop reaching contact with a still aqueous surface.

While there were varying degrees of success, this paper introduced these methodologies as a comparable alternative to many popular deep learning algorithms such as Autoencoders and Generative Adversarial Networks. The framework introduced here and the aforementioned network algorithms can be utilized for image extrapolation and the generation of reconstructed images, however our framework's advantage is that the black-box PDE and ODE models provide a mathematical and physics-driven intuition on how future images should be generated. Autoencoders and Generative Adversarial Networks cannot provide this backbone and hence

many of the extrapolated images generated by these models are solely based on user-fed prompt queries and large amounts of training data. One of the main drawbacks of the framework introduced in this paper is the computational complexity as it grows larger due to the amount of algorithms used in the framework. To express the worst-case computational runtime of algorithms, we will be referring to big-O notation which is often used in the computer science disciplines. For more information about big-O notation and its relationship with the asymptotic growth of functions, please refer to Cormen et al.'s *Introduction to Algorithms* (2022). For example, POD has computational runtimes of $O(mn^2)$ or $O(m^2n)$ for any $n \times m$ data matrix (whichever m or n is largest gets the square). In addition, linear regression with n observations and p predictor variables has runtime $O(np^2 + p^3)$ to solve for the ordinary least squares coefficient estimates. Lastly, LASSO with n observations and p predictors has a runtime of $O(p^3n)$ if $p < n$ and $O(p^2)$ if $p \geq n$. While these algorithms run in polynomial time, their complexity will exponentially increase as more features and observations are added to the original data.

Future Work

Accomplishing the task of learning these black-box models from the image data, there are improvements and other considerations to think of for future research in this discipline. One improvement to this study could be to fully analyze the computational runtimes of the framework with more than fifty image frames in the data matrix. What happens to the efficiency of the computations as we increase the number of dissected frames? Is there a consequential introduction of unwanted noise in the reconstructed image data as this increase is incrementally made? Many of the PDEs and ODEs learned from the images are linear since the dynamical nature of our data were linear. We may be interested to adapt the POD and LASSO framework

towards nonlinear algorithmic implementation such as replacing POD with a nonlinear dimensionality reduction technique such as an Isomap or a t-distributed Stochastic Neighborhood Embedding. Instead of linear regression and LASSO, what if deep learning via artificial neural networks are used to learn the nonlinear dynamics of more complex physical systems expressed in the image data? In reference to the waterdrop ODE system, what if the symbolic ODEs can be extracted in addition to the discovered solutions proposed in this paper? These are unanswered questions that may hold merit and promising results for the next steps in further revolutionizing image and video-based scientific machine learning.

APPENDICES

APPENDIX A

CHAPTER 2 SOURCE CODES

Python Code for Finite Difference ODE Example (Figure 2.1)

```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

n = 10
h = (5-0) / n

# Get A
A = np.zeros((n+1, n+1))
A[0, 0] = 1
A[n, n] = 1
for i in range(1, n):
    A[i, i-1] = 1
    A[i, i] = -2
    A[i, i+1] = 1

print(A)

# Get b
b = np.zeros(n+1)
b[1:-1] = -9.8*h**2
b[-1] = 50
print(b)

# solve the linear equations
y = np.linalg.solve(A, b)

t = np.linspace(0, 5, 11)

plt.figure(figsize=(10,8))
plt.plot(t, y)
plt.plot(5, 50, 'ro')
plt.xlabel('time (s)')
plt.ylabel('altitude (m)')
plt.show()
```

Python Code for Finite Difference PDE Example (Figure 2.2)

```
import numpy as np
import matplotlib.pyplot as plt

M = 20
N = 1000
dx = 1.0/M
dt = 1.0/N
lambda_ = dt/dx**2
```

```
if lambda_ > 0.5:
    print("Warning: Scheme may be unstable.")

x = np.linspace(0, 1, M+1)
u = np.sin(np.pi * x)
u[0] = u[-1] = 0

for i in range(N):
    u_new = u.copy()
    for j in range(1, M):
        u_new[j] = u[j] + lambda_* (u[j+1] - 2*u[j] + u[j-1])
    u = u_new

t_final = N*dt
u_analytical = np.exp(-np.pi**2 * t_final) * np.sin(np.pi * x)
plt.plot(x, u, label="Numerical Solution")
plt.plot(x, u_analytical, label="Analytical Solution", linestyle="--")
plt.title(f"Diffusion Equation Solution at t = {t_final:.2f}")
plt.xlabel("X")
plt.ylabel("u(x,t)")
plt.legend()
plt.grid(True)
plt.show()

# Compute max error at final time
max_error = np.max(np.abs(u - u_analytical))

# Print max error
print(f"Maximum error at t = {t_final:.2f}: {max_error:.6e}")

u_all = np.zeros((N+1, M+1))  # store full solution: rows = time, cols = spac
e
u_all[0, :] = np.sin(np.pi * x)  # initial condition
u_all[0, 0] = u_all[0, -1] = 0   # enforce boundary

for n in range(N):
    u_new = u_all[n, :].copy()
    for j in range(1, M):
        u_new[j] = u_all[n, j] + lambda_ * (u_all[n, j+1] - 2*u_all[n, j] + u
_all[n, j-1])
    u_new[0] = u_new[-1] = 0  # enforce boundary
    u_all[n+1, :] = u_new

u_exact_all = np.zeros_like(u_all)
for n in range(N+1):
    t_n = n * dt
    u_exact_all[n, :] = np.exp(-np.pi**2 * t_n) * np.sin(np.pi * x)
```

```
error_array = np.abs(u_all - u_exact_all)
max_error_overall = np.max(error_array)
l2_error_overall = np.sqrt(np.sum(error_array**2) * dx * dt)

print(f"Max error over all (x,t): {max_error_overall:.6e}")
```

Python Code for Proper Orthogonal Decomposition of Matrix A (Figure 2.7)

```
    import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

# Original matrix A
A = np.array([
    [2, 3, 1, 5, 4],
    [4, 6, 2, 10, 8],
    [1, 1, 1, 2, 2],
    [3, 4, 2, 6, 5],
    [5, 7, 3, 12, 9],
])

# Perform SVD
U, S, VT = np.linalg.svd(A, full_matrices=True)

# Reconstruct the matrix from SVD
S_matrix = np.zeros((5, 5))
np.fill_diagonal(S_matrix, S)
A_reconstructed = U @ S_matrix @ VT

# Truncate to first 2 modes
U2 = U[:, :2]
S2 = np.diag(S[:2])
VT2 = VT[:2, :]
A_approx = U2 @ S2 @ VT2

# Prepare DataFrames for display
df_U = pd.DataFrame(U, columns=[f"Mode {i+1}" for i in range(U.shape[1])])
df_S = pd.DataFrame(np.diag(S_matrix), columns=["Singular Values"])
df_VT = pd.DataFrame(VT, columns=[f"t{i+1}" for i in range(VT.shape[1])])

# Plot singular values
plt.figure(figsize=(6, 4))
plt.plot(S, 'o-', label='Singular Values')
plt.title('Singular Values of Matrix A')
plt.xlabel('Index')
plt.ylabel('Value')
plt.grid(True)
```

```
plt.legend()
plt.tight_layout()
plt.show()

# Show heatmaps
fig, axs = plt.subplots(1, 3, figsize=(18, 5))
sns.heatmap(A, annot=True, fmt=".1f", ax=axs[0], cmap="Blues")
axs[0].set_title("Original Matrix A")

sns.heatmap(A_reconstructed, annot=True, fmt=".1f", ax=axs[1], cmap="Greens")
axs[1].set_title("Reconstructed A from Full SVD")

sns.heatmap(A_approx, annot=True, fmt=".1f", ax=axs[2], cmap="Oranges")
axs[2].set_title("Approximation using 2 Modes")

plt.tight_layout()
plt.show()
```

APPENDIX B

CHAPTER 3 SOURCE CODES

Python Code for Object Translation (Circle Moving Left to Right)

```
import os
import re
import glob
from PIL import Image
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import skimage as ski
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Lasso, LassoCV, LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
import seaborn as sns
from sklearn.ensemble import GradientBoostingRegressor
import pysindy as ps
import statsmodels.api as sm

image_1 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Toy_Example_Data/moving_square_1.jpg", as_gray=True)
image_1 = np.where(image_1 == 1, 0, 1)
image_1 = ski.filters.gaussian(image_1, sigma=55)
image_1 = image_1 * 1e10
plt.imshow(image_1, cmap="gray")
#print(image_1)

h = 1

# Forward Difference for first derivatives and second derivatives.

dx1 = np.gradient(image_1, h, axis=1)
dy1 = np.gradient(image_1, h, axis=0)
dxx1 = np.gradient(dx1, h, axis=1)
dyy1 = np.gradient(dy1, h, axis=0)

print(dx1)
print(dy1)
print(dxx1)
print(dyy1)

image_2 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Toy_Example_Data/moving_square_2.jpg", as_gray=True)
image_2 = np.where(image_2 == 1, 0, 1)
image_2 = ski.filters.gaussian(image_2, sigma=55)
image_2 = image_2 *1e10
```

```
plt.imshow(image_2, cmap="gray")

dx2 = np.gradient(image_2, h, axis=1)
dy2 = np.gradient(image_2, h, axis=0)
dxx2 = np.gradient(dx2, h, axis=1)
dyy2 = np.gradient(dy2, h, axis=0)

image_3 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Toy_Example_Data/moving_square_3.jpg", as_gray=True)
image_3 = np.where(image_3 == 1, 0, 1)
image_3 = ski.filters.gaussian(image_3, sigma=55)
image_3 = image_3 * 1e10
plt.imshow(image_3, cmap="gray")

dx3 = np.gradient(image_3, h, axis=1)
dy3 = np.gradient(image_3, h, axis=0)
dxx3 = np.gradient(dx3, h, axis=1)
dyy3 = np.gradient(dy3, h, axis=0)

image_4 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Toy_Example_Data/moving_square_4.jpg", as_gray=True)
image_4 = np.where(image_4 == 1, 0, 1)
image_4 = ski.filters.gaussian(image_4, sigma=55)
image_4 = image_4 * 1e10
plt.imshow(image_4, cmap="gray")

dx4 = np.gradient(image_4, h, axis=1)
dy4 = np.gradient(image_4, h, axis=0)
dxx4 = np.gradient(dx4, h, axis=1)
dyy4 = np.gradient(dy4, h, axis=0)

image_5 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Toy_Example_Data/moving_square_5.jpg", as_gray=True)
image_5 = np.where(image_5 == 1, 0, 1)
image_5 = ski.filters.gaussian(image_5, sigma=55)
image_5 = image_5 * 1e10
plt.imshow(image_5, cmap="gray")

dx5 = np.gradient(image_5, h, axis=1)
dy5 = np.gradient(image_5, h, axis=0)
dxx5 = np.gradient(dx5, h, axis=1)
dyy5 = np.gradient(dy5, h, axis=0)

image_6 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Toy_Example_Data/moving_square_6.jpg", as_gray=True)
```

```
image_6 = np.where(image_6 == 1, 0, 1)
image_6 = ski.filters.gaussian(image_6, sigma=55)
image_6 = image_6 * 1e10
plt.imshow(image_6, cmap="gray")

dx6 = np.gradient(image_6, h, axis=1)
dy6 = np.gradient(image_6, h, axis=0)
dxx6 = np.gradient(dx6, h, axis=1)
dyy6 = np.gradient(dy6, h, axis=0)

image_7 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Toy_Example_Data/moving_square_7.jpg", as_gray=True)
image_7 = np.where(image_7 == 1, 0, 1)
image_7 = ski.filters.gaussian(image_7, sigma=55)
image_7 = image_7 * 1e10
plt.imshow(image_7, cmap="gray")

dx7 = np.gradient(image_7, h, axis=1)
dy7 = np.gradient(image_7, h, axis=0)
dxx7 = np.gradient(dx7, h, axis=1)
dyy7 = np.gradient(dy7, h, axis=0)

image_8 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Toy_Example_Data/moving_square_8.jpg", as_gray=True)
image_8 = np.where(image_8 == 1, 0, 1)
image_8 = ski.filters.gaussian(image_8, sigma=55)
image_8 = image_8 * 1e10
plt.imshow(image_8, cmap="gray")

dx8 = np.gradient(image_8, h, axis=1)
dy8 = np.gradient(image_8, h, axis=0)
dxx8 = np.gradient(dx8, h, axis=1)
dyy8 = np.gradient(dy8, h, axis=0)

image_9 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Toy_Example_Data/moving_square_9.jpg", as_gray=True)
image_9 = np.where(image_9 == 1, 0, 1)
image_9 = ski.filters.gaussian(image_9, sigma=55)
image_9 = image_9 * 1e10
plt.imshow(image_9, cmap="gray")

dx9 = np.gradient(image_9, h, axis=1)
dy9 = np.gradient(image_9, h, axis=0)
dxx9 = np.gradient(dx9, h, axis=1)
dyy9 = np.gradient(dy9, h, axis=0)
```

```
image_10 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_10.jpg", as_gray=True)
image_10 = np.where(image_10 == 1, 0, 1)
image_10 = ski.filters.gaussian(image_10, sigma=55)
image_10 = image_10 * 1e10
plt.imshow(image_10, cmap="gray")

dx10 = np.gradient(image_10, h, axis=1)
dy10 = np.gradient(image_10, h, axis=0)
dxx10 = np.gradient(dx10, h, axis=1)
dyy10 = np.gradient(dy10, h, axis=0)

image_11 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_11.jpg", as_gray=True)
image_11 = np.where(image_11 == 1, 0, 1)
image_11 = ski.filters.gaussian(image_11, sigma=55)
image_11 = image_11 * 1e10
plt.imshow(image_11, cmap="gray")

dx11 = np.gradient(image_11, h, axis=1)
dy11 = np.gradient(image_11, h, axis=0)
dxx11 = np.gradient(dx11, h, axis=1)
dyy11 = np.gradient(dy11, h, axis=0)

image_12 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_12.jpg", as_gray=True)
image_12 = np.where(image_12 == 1, 0, 1)
image_12 = ski.filters.gaussian(image_12, sigma=55)
image_12 = image_12 * 1e10
plt.imshow(image_12, cmap="gray")

dx12 = np.gradient(image_12, h, axis=1)
dy12 = np.gradient(image_12, h, axis=0)
dxx12 = np.gradient(dx12, h, axis=1)
dyy12 = np.gradient(dy12, h, axis=0)

image_13 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_13.jpg", as_gray=True)
image_13 = np.where(image_13 == 1, 0, 1)
image_13 = ski.filters.gaussian(image_13, sigma=55)
image_13 = image_13 * 1e10
plt.imshow(image_13, cmap="gray")

dx13 = np.gradient(image_13, h, axis=1)
dy13 = np.gradient(image_13, h, axis=0)
```

```
dxx13 = np.gradient(dx13, h, axis=1)
dyy13 = np.gradient(dy13, h, axis=0)

image_14 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_14.jpg", as_gray=True)
image_14 = np.where(image_14 == 1, 0, 1)
image_14 = ski.filters.gaussian(image_14, sigma=55)
image_14 = image_14 * 1e10
plt.imshow(image_14, cmap="gray")

dx14 = np.gradient(image_14, h, axis=1)
dy14 = np.gradient(image_14, h, axis=0)
dxx14 = np.gradient(dx14, h, axis=1)
dyy14 = np.gradient(dy14, h, axis=0)

image_15 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_15.jpg", as_gray=True)
image_15 = np.where(image_15 == 1, 0, 1)
image_15 = ski.filters.gaussian(image_15, sigma=55)
image_15 = image_15 * 1e10
plt.imshow(image_15, cmap="gray")

dx15 = np.gradient(image_15, h, axis=1)
dy15 = np.gradient(image_15, h, axis=0)
dxx15 = np.gradient(dx15, h, axis=1)
dyy15 = np.gradient(dy15, h, axis=0)

image_16 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_16.jpg", as_gray=True)
image_16 = np.where(image_16 == 1, 0, 1)
image_16 = ski.filters.gaussian(image_16, sigma=55)
image_16 = image_16 * 1e10
plt.imshow(image_16, cmap="gray")

dx16 = np.gradient(image_16, h, axis=1)
dy16 = np.gradient(image_16, h, axis=0)
dxx16 = np.gradient(dx16, h, axis=1)
dyy16 = np.gradient(dy16, h, axis=0)

image_17 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_17.jpg", as_gray=True)
image_17 = np.where(image_17 == 1, 0, 1)
image_17 = ski.filters.gaussian(image_17, sigma=55)
image_17 = image_17 * 1e10
plt.imshow(image_17, cmap="gray")
```

```
dx17 = np.gradient(image_17, h, axis=1)
dy17 = np.gradient(image_17, h, axis=0)
dxx17 = np.gradient(dx17, h, axis=1)
dyy17 = np.gradient(dy17, h, axis=0)

image_18 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_18.jpg", as_gray=True)
image_18 = np.where(image_18 == 1, 0, 1)
image_18 = ski.filters.gaussian(image_18, sigma=55)
image_18 = image_18 * 1e10
plt.imshow(image_18, cmap="gray")

dx18 = np.gradient(image_18, h, axis=1)
dy18 = np.gradient(image_18, h, axis=0)
dxx18 = np.gradient(dx18, h, axis=1)
dyy18 = np.gradient(dy18, h, axis=0)

image_19 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_19.jpg", as_gray=True)
image_19 = np.where(image_19 == 1, 0, 1)
image_19 = ski.filters.gaussian(image_19, sigma=55)
image_19 = image_19 * 1e10
plt.imshow(image_19, cmap="gray")

dx19 = np.gradient(image_19, h, axis=1)
dy19 = np.gradient(image_19, h, axis=0)
dxx19 = np.gradient(dx19, h, axis=1)
dyy19 = np.gradient(dy19, h, axis=0)

image_20 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_20.jpg", as_gray=True)
image_20 = np.where(image_20 == 1, 0, 1)
image_20 = ski.filters.gaussian(image_20, sigma=55)
image_20 = image_20 * 1e10
plt.imshow(image_20, cmap="gray")

dx20 = np.gradient(image_20, h, axis=1)
dy20 = np.gradient(image_20, h, axis=0)
dxx20 = np.gradient(dx20, h, axis=1)
dyy20 = np.gradient(dy20, h, axis=0)

image_21 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_21.jpg", as_gray=True)
image_21 = np.where(image_21 == 1, 0, 1)
```

```
image_21 = ski.filters.gaussian(image_21, sigma=55)
image_21 = image_21 * 1e10
plt.imshow(image_21, cmap="gray")

dx21 = np.gradient(image_21, h, axis=1)
dy21 = np.gradient(image_21, h, axis=0)
dxx21 = np.gradient(dy21, h, axis=1)
dyy21 = np.gradient(dy21, h, axis=0)

image_22 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_22.jpg", as_gray=True)
image_22 = np.where(image_22 == 1, 0, 1)
image_22 = ski.filters.gaussian(image_22, sigma=55)
image_22 = image_22 * 1e10
plt.imshow(image_22, cmap="gray")

dx22 = np.gradient(image_22, h, axis=1)
dy22 = np.gradient(image_22, h, axis=0)
dxx22 = np.gradient(dx22, h, axis=1)
dyy22 = np.gradient(dy22, h, axis=0)

image_23 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_23.jpg", as_gray=True)
image_23 = np.where(image_23 == 1, 0, 1)
image_23 = ski.filters.gaussian(image_23, sigma=55)
image_23 = image_23 * 1e10
plt.imshow(image_23, cmap="gray")

dx23 = np.gradient(image_23, h, axis=1)
dy23 = np.gradient(image_23, h, axis=0)
dxx23 = np.gradient(dx23, h, axis=1)
dyy23 = np.gradient(dy23, h, axis=0)

image_24 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_24.jpg", as_gray=True)
image_24 = np.where(image_24 == 1, 0, 1)
image_24 = ski.filters.gaussian(image_24, sigma=55)
image_24 = image_24 * 1e10
plt.imshow(image_24, cmap="gray")

dx24 = np.gradient(image_24, h, axis=1)
dy24 = np.gradient(image_24, h, axis=0)
dxx24 = np.gradient(dx24, h, axis=1)
dyy24 = np.gradient(dy24, h, axis=0)

image_25 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
```

```
n_Suzuki
/Toy_Example_Data/moving_square_25.jpg", as_gray=True)
image_25 = np.where(image_25 == 1, 0, 1)
image_25 = ski.filters.gaussian(image_25, sigma=55)
image_25 = image_25 * 1e10
plt.imshow(image_25, cmap="gray")

dx25 = np.gradient(image_25, h, axis=1)
dy25 = np.gradient(image_25, h, axis=0)
dxx25 = np.gradient(dx25, h, axis=1)
dyy25 = np.gradient(dy25, h, axis=0)

image_26 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_26.jpg", as_gray=True)
image_26 = np.where(image_26 == 1, 0, 1)
image_26 = ski.filters.gaussian(image_26, sigma=55)
image_26 = image_26 * 1e10
plt.imshow(image_26, cmap="gray")

dx26 = np.gradient(image_26, h, axis=1)
dy26 = np.gradient(image_26, h, axis=0)
dxx26 = np.gradient(dx26, h, axis=1)
dyy26 = np.gradient(dy26, h, axis=0)

image_27 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_27.jpg", as_gray=True)
image_27 = np.where(image_27 == 1, 0, 1)
image_27 = ski.filters.gaussian(image_27, sigma=55)
image_27 = image_27 * 1e10
plt.imshow(image_27, cmap="gray")

dx27 = np.gradient(image_27, h, axis=1)
dy27 = np.gradient(image_27, h, axis=0)
dxx27 = np.gradient(dx27, h, axis=1)
dyy27 = np.gradient(dy27, h, axis=0)

image_28 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_28.jpg", as_gray=True)
image_28 = np.where(image_28 == 1, 0, 1)
image_28 = ski.filters.gaussian(image_28, sigma=55)
image_28 = image_28 * 1e10
plt.imshow(image_28, cmap="gray")

dx28 = np.gradient(image_28, h, axis=1)
dy28 = np.gradient(image_28, h, axis=0)
dxx28 = np.gradient(dx28, h, axis=1)
```

```
dyy28 = np.gradient(dy28, h, axis=0)

image_29 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_29.jpg", as_gray=True)
image_29 = np.where(image_29 == 1, 0, 1)
image_29 = ski.filters.gaussian(image_29, sigma=55)
image_29 = image_29 * 1e10
plt.imshow(image_29, cmap="gray")

dx29 = np.gradient(image_29, h, axis=1)
dy29 = np.gradient(image_29, h, axis=0)
dxx29 = np.gradient(dx29, h, axis=1)
dyy29 = np.gradient(dy29, h, axis=0)

image_30 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_30.jpg", as_gray=True)
image_30 = np.where(image_30 == 1, 0, 1)
image_30 = ski.filters.gaussian(image_30, sigma=55)
image_30 = image_30 * 1e10
plt.imshow(image_30, cmap="gray")

dx30 = np.gradient(image_30, h, axis=1)
dy30 = np.gradient(image_30, h, axis=0)
dxx30 = np.gradient(dx30, h, axis=1)
dyy30 = np.gradient(dy30, h, axis=0)

image_31 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_31.jpg", as_gray=True)
image_31 = np.where(image_31 == 1, 0, 1)
image_31 = ski.filters.gaussian(image_31, sigma=55)
image_31 = image_31 * 1e10
plt.imshow(image_31, cmap="gray")

dx31 = np.gradient(image_31, h, axis=1)
dy31 = np.gradient(image_31, h, axis=0)
dxx31 = np.gradient(dx31, h, axis=1)
dyy31 = np.gradient(dy31, h, axis=0)

image_32 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_32.jpg", as_gray=True)
image_32 = np.where(image_32 == 1, 0, 1)
image_32 = ski.filters.gaussian(image_32, sigma=55)
image_32 = image_32 * 1e10
plt.imshow(image_32, cmap="gray")
```

```
dx32 = np.gradient(image_32, h, axis=1)
dy32 = np.gradient(image_32, h, axis=0)
dxx32 = np.gradient(dx32, h, axis=1)
dyy32 = np.gradient(dy32, h, axis=0)

image_33 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_33.jpg", as_gray=True)
image_33 = np.where(image_33 == 1, 0, 1)
image_33 = ski.filters.gaussian(image_33, sigma=55)
image_33 = image_33 * 1e10
plt.imshow(image_33, cmap="gray")

dx33 = np.gradient(image_33, h, axis=1)
dy33 = np.gradient(image_33, h, axis=0)
dxx33 = np.gradient(dx33, h, axis=1)
dyy33 = np.gradient(dy33, h, axis=0)

image_34 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_example_Data/moving_square_34.jpg", as_gray=True)
image_34 = np.where(image_34 == 1, 0, 1)
image_34 = ski.filters.gaussian(image_34, sigma=55)
image_34 = image_34 * 1e10
plt.imshow(image_34, cmap="gray")

dx34 = np.gradient(image_34, h, axis=1)
dy34 = np.gradient(image_34, h, axis=0)
dxx34 = np.gradient(dx34, h, axis=1)
dyy34 = np.gradient(dy34, h, axis=0)

image_35 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_35.jpg", as_gray=True)
image_35 = np.where(image_35 == 1, 0, 1)
image_35 = ski.filters.gaussian(image_35, sigma=55)
image_35 = image_35 * 1e10
plt.imshow(image_35, cmap="gray")

dx35 = np.gradient(image_35, h, axis=1)
dy35 = np.gradient(image_35, h, axis=0)
dxx35 = np.gradient(dx35, h, axis=1)
dyy35 = np.gradient(dy35, h, axis=0)

image_36 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_36.jpg", as_gray=True)
image_36 = np.where(image_36 == 1, 0, 1)
image_36 = ski.filters.gaussian(image_36, sigma=55)
```

```
image_36 = image_36 * 1e10
plt.imshow(image_36, cmap="gray")

dx36 = np.gradient(image_36, h, axis=1)
dy36 = np.gradient(image_36, h, axis=0)
dxx36 = np.gradient(dx36, h, axis=1)
dyy36 = np.gradient(dy36, h, axis=0)

image_37 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_37.jpg", as_gray=True)
image_37 = np.where(image_37 == 1, 0, 1)
image_37 = ski.filters.gaussian(image_37, sigma=55)
image_37 = image_37 * 1e10
plt.imshow(image_37, cmap="gray")

dx37 = np.gradient(image_37, h, axis=1)
dy37 = np.gradient(image_37, h, axis=0)
dxx37 = np.gradient(dx37, h, axis=1)
dyy37 = np.gradient(dy37, h, axis=0)

image_38 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_38.jpg", as_gray=True)
image_38 = np.where(image_38 == 1, 0, 1)
image_38 = ski.filters.gaussian(image_38, sigma=55)
image_38 = image_38 * 1e10
plt.imshow(image_38, cmap="gray")

dx38 = np.gradient(image_38, h, axis=1)
dy38 = np.gradient(image_38, h, axis=0)
dxx38 = np.gradient(dx38, h, axis=1)
dyy38 = np.gradient(dy38, h, axis=0)

image_39 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_39.jpg", as_gray=True)
image_39 = np.where(image_39 == 1, 0, 1)
image_39 = ski.filters.gaussian(image_39, sigma=55)
image_39 = image_39 * 1e10
plt.imshow(image_39, cmap="gray")

dx39 = np.gradient(image_39, h, axis=1)
dy39 = np.gradient(image_39, h, axis=0)
dxx39 = np.gradient(dx39, h, axis=1)
dyy39 = np.gradient(dy39, h, axis=0)

image_40 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
```

```
/Toy_Example_Data/moving_square_40.jpg", as_gray=True)
image_40 = np.where(image_40 == 1, 0, 1)
image_40 = ski.filters.gaussian(image_40, sigma=55)
image_40 = image_40 * 1e10
plt.imshow(image_40, cmap="gray")

dx40 = np.gradient(image_40, h, axis=1)
dy40 = np.gradient(image_40, h, axis=0)
dxx40 = np.gradient(dx40, h, axis=1)
dyy40 = np.gradient(dy40, h, axis=0)

image_41 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_41.jpg", as_gray=True)
image_41 = np.where(image_41 == 1, 0, 1)
image_41 = ski.filters.gaussian(image_41, sigma=55)
image_41 = image_41 * 1e10
plt.imshow(image_41, cmap="gray")

dx41 = np.gradient(image_41, h, axis=1)
dy41 = np.gradient(image_41, h, axis=0)
dxx41 = np.gradient(dx41, h, axis=1)
dyy41 = np.gradient(dy41, h, axis=0)

image_42 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_42.jpg", as_gray=True)
image_42 = np.where(image_42 == 1, 0, 1)
image_42 = ski.filters.gaussian(image_42, sigma=55)
image_42 = image_42 * 1e10
plt.imshow(image_42, cmap="gray")

dx42 = np.gradient(image_42, h, axis=1)
dy42 = np.gradient(image_42, h, axis=0)
dxx42 = np.gradient(dx42, h, axis=1)
dyy42 = np.gradient(dy42, h, axis=0)

image_43 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_43.jpg", as_gray=True)
image_43 = np.where(image_43 == 1, 0, 1)
image_43 = ski.filters.gaussian(image_43, sigma=55)
image_43 = image_43 * 1e10
plt.imshow(image_43, cmap="gray")

dx43 = np.gradient(image_43, h, axis=1)
dy43 = np.gradient(image_43, h, axis=0)
dxx43 = np.gradient(dx43, h, axis=1)
dyy43 = np.gradient(dy43, h, axis=0)
```

```
image_44 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_44.jpg", as_gray=True)
image_44 = np.where(image_44 == 1, 0, 1)
image_44 = ski.filters.gaussian(image_44, sigma=55)
image_44 = image_44 * 1e10
plt.imshow(image_44, cmap="gray")

dx44 = np.gradient(image_44, h, axis=1)
dy44 = np.gradient(image_44, h, axis=0)
dxx44 = np.gradient(dx44, h, axis=1)
dyy44 = np.gradient(dy44, h, axis=0)

image_45 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_45.jpg", as_gray=True)
image_45 = np.where(image_45 == 1, 0, 1)
image_45 = ski.filters.gaussian(image_45, sigma=55)
image_45 = image_45 * 1e10
plt.imshow(image_45, cmap="gray")

dx45 = np.gradient(image_45, h, axis=1)
dy45 = np.gradient(image_45, h, axis=0)
dxx45 = np.gradient(dx45, h, axis=1)
dyy45 = np.gradient(dy45, h, axis=0)

image_46 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_46.jpg", as_gray=True)
image_46 = np.where(image_46 == 1, 0, 1)
image_46 = ski.filters.gaussian(image_46, sigma=55)
image_46 = image_46 * 1e10
plt.imshow(image_46, cmap="gray")

dx46 = np.gradient(image_46, h, axis=1)
dy46 = np.gradient(image_46, h, axis=0)
dxx46 = np.gradient(dx46, h, axis=1)
dyy46 = np.gradient(dy46, h, axis=0)

image_47 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_47.jpg", as_gray=True)
image_47 = np.where(image_47 == 1, 0, 1)
image_47 = ski.filters.gaussian(image_47, sigma=55)
image_47 = image_47 * 1e10
plt.imshow(image_47, cmap="gray")

dx47 = np.gradient(image_47, h, axis=1)
```

```
dy47 = np.gradient(image_47, h, axis=0)
dxx47 = np.gradient(dx47, h, axis=1)
dyy47 = np.gradient(dy47, h, axis=0)

image_48 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_48.jpg", as_gray=True)
image_48 = np.where(image_48 == 1, 0, 1)
image_48 = ski.filters.gaussian(image_48, sigma=55)
image_48 = image_48 * 1e10
plt.imshow(image_48, cmap="gray")

dx48 = np.gradient(image_48, h, axis=1)
dy48 = np.gradient(image_48, h, axis=0)
dxx48 = np.gradient(dx48, h, axis=1)
dyy48 = np.gradient(dy48, h, axis=0)

image_49 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_49.jpg", as_gray=True)
image_49 = np.where(image_49 == 1, 0, 1)
image_49 = ski.filters.gaussian(image_49, sigma=55)
image_49 = image_49 * 1e10
plt.imshow(image_49, cmap="gray")

dx49 = np.gradient(image_49, h, axis=1)
dy49 = np.gradient(image_49, h, axis=0)
dxx49 = np.gradient(dx49, h, axis=1)
dyy49 = np.gradient(dy49, h, axis=0)

image_50 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_50.jpg", as_gray=True)
image_50 = np.where(image_50 == 1, 0, 1)
image_50 = ski.filters.gaussian(image_50, sigma=55)
image_50 = image_50 * 1e10
plt.imshow(image_50, cmap="gray")

dx50 = np.gradient(image_50, h, axis=1)
dy50 = np.gradient(image_50, h, axis=0)
dxx50 = np.gradient(dx50, h, axis=1)
dyy50 = np.gradient(dy50, h, axis=0)

# Concatenated data matrix of pixel values and gradients.

us2 = [image_1, image_2, image_3, image_4, image_5, image_6, image_7, image_8,
 image_9, image_10,
      image_11, image_12, image_13, image_14, image_15, image_16, image_17, i
mage_18,
```

```
      image_19, image_20, image_21, image_22, image_23, image_24, image_25,
      image_26, image_27, image_28, image_29, image_30, image_31, image_32, i
mage_33,
      image_34, image_35, image_36, image_37, image_38, image_39, image_40, i
mage_41,
      image_42, image_43, image_44, image_45]


uts2 = [us2[i+1] - us2[i] for i in range(len(us2) - 1)]


dxs2 = [dx1, dx2, dx3, dx4, dx5, dx6, dx7, dx8, dx9, dx10, dx11, dx12, dx13,
dx14,
          dx15, dx16, dx17, dx18, dx19, dx20, dx21, dx22, dx23, dx24, dx25, d
x26,
          dx27, dx28, dx29, dx30, dx31, dx32, dx33, dx34, dx35, dx36, dx37, d
x38, dx39,
          dx40, dx41, dx42, dx43, dx44, dx45]


dys2 = [dy1, dy2, dy3,dy4, dy5, dy6, dy7, dy8, dy9, dy10, dy11, dy12, dy13, d
y14,
       dy15, dy16, dy17, dy18, dy19, dy20, dy21, dy22, dy23, dy24, dy25, dy26,

          dy27, dy28, dy29, dy30, dy31, dy32, dy33, dy34, dy35, dy36, dy37, d
y38, dy39,
          dy40, dy41, dy42, dy43, dy44, dy45]


dxxs2 = [dxx1, dxx2, dxx3, dxx4, dxx5, dxx6, dxx7, dxx8, dxx9, dxx10, dxx11,
dxx12, dxx13, dxx14,
          dxx15, dxx16, dxx17, dxx18, dxx19, dxx20, dxx21, dxx22, dxx23, dxx2
4, dxx25, dxx26,
          dxx27, dxx28, dxx29, dxx30, dxx31, dxx32, dxx33, dxx34, dxx35, dxx3
6, dxx37, dxx38, dxx39,
          dxx40, dxx41, dxx42, dxx43, dxx44, dxx45]

dyys2 = [dyy1, dyy2, dyy3, dyy4, dyy5, dyy6, dyy7, dyy8, dyy9, dyy10, dyy11,
dyy12, dyy13, dyy14,
          dyy15, dyy16, dyy17, dyy18, dyy19, dyy20, dyy21, dyy22, dyy23, dyy2
4, dyy25, dyy26,
          dyy27, dyy28, dyy29, dyy30, dyy31, dyy32, dyy33, dyy34, dyy35, dyy3
6, dyy37, dyy38, dyy39,
          dyy40, dyy41, dyy42, dyy43, dyy44, dyy45]

flattened_arr_u2 = [u.flatten() for u in us2]
merged_u2 = np.concatenate(flattened_arr_u2)
dfu2 = pd.DataFrame({'u': merged_u2})
```

```
# Flatten each matrix into a column vector (1D)
flattened_arrays2 = [dx.flatten() for dx in dxs2]

# Concatenate all arrays into a single column
merged_column2 = np.concatenate(flattened_arrays2)

# Create DataFrame with one column
dfx2 = pd.DataFrame({'dx': merged_column2})

flattened_arr_dy2 = [dy.flatten() for dy in dys2]
merged_dy2 = np.concatenate(flattened_arr_dy2)
dfy2 = pd.DataFrame({'dy': merged_dy2})

flattened_arr_dxx2 = [dxx.flatten() for dxx in dxxs2]
merged_dxx2 = np.concatenate(flattened_arr_dxx2)
dfxx2 = pd.DataFrame({'dxx': merged_dxx2})

flattened_arr_dyy2 = [dyy.flatten() for dyy in dyys2]
merged_dyy2 = np.concatenate(flattened_arr_dy2)
dfyy2 = pd.DataFrame({'dyy': merged_dyy2})

flattened_arr_udt2 = [dut.flatten() for dut in uts2]
merged_dut2 = np.concatenate(flattened_arr_udt2)
dfudt2 = pd.DataFrame({'dfudt': merged_dut2})

combined_arr2 = pd.concat([dfu2, dfx2, dfy2, dfxx2, dfyy2, dfudt2], axis=1)
combined_arr2

# Drop rows with all zeros

combined_arr2 = combined_arr2[(combined_arr2 != 0).any(axis=1)]
print(combined_arr2.shape)

# Drop columns with all zeros

combined_arr2 = combined_arr2.loc[:, (combined_arr2 != 0).any(axis=0)]
print(combined_arr2.shape)

combined_arr2 = combined_arr2.dropna()
print(combined_arr2.shape)
combined_arr2

# Concatenated data matrix of pixel values and gradients.

us3 = [image_46, image_47, image_48, image_49, image_50]
```

```
uts3 = [us3[i+1] - us3[i] for i in range(len(us3) - 1)]


dxs3 = [dx46, dx46, dx48, dx49, dx50]


dys3 = [dy46, dy47, dy48, dy49, dy50]


dxxs3 = [dxx46, dxx47, dxx48, dxx49, dxx50]

dyys3 = [dyy46, dyy47, dyy48, dyy49, dyy50]

flattened_arr_u3 = [u.flatten() for u in us3]
merged_u3 = np.concatenate(flattened_arr_u3)
dfu3 = pd.DataFrame({'u': merged_u3})

# Flatten each matrix into a column vector (1D)
flattened_arrays3 = [dx.flatten() for dx in dxs3]

# Concatenate all arrays into a single column
merged_column3 = np.concatenate(flattened_arrays3)

# Create DataFrame with one column
dfx3 = pd.DataFrame({'dx': merged_column3})

flattened_arr_dy3 = [dy.flatten() for dy in dys3]
merged_dy3 = np.concatenate(flattened_arr_dy3)
dfy3 = pd.DataFrame({'dy': merged_dy3})

flattened_arr_dxx3 = [dxx.flatten() for dxx in dxxs3]
merged_dxx3 = np.concatenate(flattened_arr_dxx3)
dfxx3 = pd.DataFrame({'dxx': merged_dxx3})

flattened_arr_dyy3 = [dyy.flatten() for dyy in dyys3]
merged_dyy3 = np.concatenate(flattened_arr_dy3)
dfyy3 = pd.DataFrame({'dyy': merged_dyy3})

flattened_arr_udt3 = [dut.flatten() for dut in uts3]
merged_dut3 = np.concatenate(flattened_arr_udt3)
dfudt3 = pd.DataFrame({'dfudt': merged_dut3})

combined_arr3 = pd.concat([dfu3, dfx3, dfy3, dfxx3, dfyy3, dfudt3], axis=1)
combined_arr3

# Drop rows with all zeros
```

```
combined_arr3 = combined_arr3[(combined_arr3 != 0).any(axis=1)]
print(combined_arr3.shape)

# Drop columns with all zeros

combined_arr3 = combined_arr3.loc[:, (combined_arr3 != 0).any(axis=0)]
print(combined_arr3.shape)

combined_arr3 = combined_arr3.dropna()
print(combined_arr3.shape)
combined_arr3

combined_arr2.to_csv('C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon_Su
zuki
/toy_train_data.csv', index=False, encoding='utf-8')
combined_arr3.to_csv('C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon_Su
zuki
/toy_test_data.csv', index=False, encoding='utf-8')

train = pd.read_csv("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon_Suz
uki
/toy_train_data.csv")
test = pd.read_csv("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon_Suzu
ki
/toy_test_data.csv")

data_X = train.drop(columns=["dfudt"])
data_y = train["dfudt"]
data_mat_X = data_X.values
data_mat_y = data_y.values

reg_mod = sm.OLS(data_mat_y, data_mat_X)
results = reg_mod.fit()
summary = results.summary()
print(summary)

lasso_ode = Lasso(alpha=0.1)
lasso_ode.fit(data_mat_X, data_mat_y)
print("lasso coefficients: ", lasso_ode.coef_)

# Computing prediction accuracy from testing set.

y_pred_ode = lasso_ode.predict(data_mat_X)
mse_ode = mean_squared_error(data_mat_y, y_pred_ode)
print("Mean Squared Error:", mse_ode)

# Tuning alpha parameter via CV with cv=5 folds.
```

```
lasso_cv_ode = LassoCV(alphas=np.logspace(-3,1,100), cv=5)
lasso_cv_ode.fit(data_mat_X, data_mat_y)

print("Optimal alpha:", lasso_cv_ode.alpha_)

# Checking number of zero coefficients (how many features dropped)

check = sum(lasso_cv_ode.coef_ == 0)
check

# LASSO Regularized regression for feature selection.

lasso_ode = Lasso(alpha=lasso_cv_ode.alpha_)
lasso_ode.fit(data_mat_X, data_mat_y)

print("LASSO Coefficients:", lasso_ode.coef_)

data_mat_X = train.drop(columns=["u", "dy", "dxx", "dyy", "dfudt"])
data_mat_X.values

reg_mod2 = sm.OLS(data_mat_y, data_mat_X)
results2 = reg_mod2.fit()
summary2 = results2.summary()
print(summary2)

from scipy.ndimage import gaussian_filter

def forward_euler_pde_mixed(u0, dx, dy, dt, steps):
    """
    Forward Euler integration for the PDE:
    u_t = 0.0321u - 10.9031u_x - 3.3925u_xy + 138.6948u_xx

    Parameters:
        u0: np.ndarray, initial image
        dx, dy: spatial resolutions
        dt: time step
        steps: number of steps (e.g., 5 for Image_51 to Image_55)

    Returns:
        List of predicted images
    """
    u = u0.copy()
    images = [u.copy()]

    for _ in range(steps):
        # First derivative in x
        u_x = (np.roll(u, -1, axis=0) - np.roll(u, 1, axis=0)) / (2 * dx)
```

```
        # Second derivative in x
        u_xx = (np.roll(u, -1, axis=0) - 2*u + np.roll(u, 1, axis=0)) / (dx**
2)

        # Second derivative in y
        u_yy = (np.roll(u, -1, axis=1) - 2*u + np.roll(u, 1, axis=1)) / (dy**
2)



        # Time derivative

        # Data Driven
        u_t = (-0.0014 * u - 13.3337 * u_x + 95.2591 * u_xx - 0.0030 * u_yy)

        # Physics Driven
        # u_t = (-13.3348 * u_x)

        # Forward Euler update
        u = u + dt * u_t
        u = gaussian_filter(u, sigma=1)
        images.append(u.copy())

    return images

h = 1

image_45 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_45.jpg", as_gray=True)
image_45 = np.where(image_45 == 1, 0, 1)
image_45 = ski.filters.gaussian(image_45, sigma=55)
image_45 = image_45 * 1e10
plt.imshow(image_45, cmap="gray")

dx45 = np.gradient(image_45, h, axis=1)
dy45 = np.gradient(image_45, h, axis=0)
dxx45 = np.gradient(dx45, h, axis=1)
dyy45 = np.gradient(dy45, h, axis=0)

image_46 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_46.jpg", as_gray=True)
image_46 = np.where(image_46 == 1, 0, 1)
image_46 = ski.filters.gaussian(image_46, sigma=55)
image_46 = image_46 * 1e10
plt.imshow(image_46, cmap="gray")
```

```
dx46 = np.gradient(image_46, h, axis=1)
dy46 = np.gradient(image_46, h, axis=0)
dxx46 = np.gradient(dx46, h, axis=1)
dyy46 = np.gradient(dy46, h, axis=0)

image_47 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_47.jpg", as_gray=True)
image_47 = np.where(image_47 == 1, 0, 1)
image_47 = ski.filters.gaussian(image_47, sigma=55)
image_47 = image_47 * 1e10
plt.imshow(image_47, cmap="gray")

dx47 = np.gradient(image_47, h, axis=1)
dy47 = np.gradient(image_47, h, axis=0)
dxx47 = np.gradient(dx47, h, axis=1)
dyy47 = np.gradient(dy47, h, axis=0)

image_48 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_48.jpg", as_gray=True)
image_48 = np.where(image_48 == 1, 0, 1)
image_48 = ski.filters.gaussian(image_48, sigma=55)
image_48 = image_48 * 1e10
plt.imshow(image_48, cmap="gray")

dx48 = np.gradient(image_48, h, axis=1)
dy48 = np.gradient(image_48, h, axis=0)
dxx48 = np.gradient(dx48, h, axis=1)
dyy48 = np.gradient(dy48, h, axis=0)

image_49 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_49.jpg", as_gray=True)
image_49 = np.where(image_49 == 1, 0, 1)
image_49 = ski.filters.gaussian(image_49, sigma=55)
image_49 = image_49 * 1e10
plt.imshow(image_49, cmap="gray")

dx49 = np.gradient(image_49, h, axis=1)
dy49 = np.gradient(image_49, h, axis=0)
dxx49 = np.gradient(dx49, h, axis=1)
dyy49 = np.gradient(dy49, h, axis=0)

image_50 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Toy_Example_Data/moving_square_50.jpg", as_gray=True)
image_50 = np.where(image_50 == 1, 0, 1)
image_50 = ski.filters.gaussian(image_50, sigma=55)
```

```
image_50 = image_50 * 1e10
plt.imshow(image_50, cmap="gray")

dx50 = np.gradient(image_50, h, axis=1)
dy50 = np.gradient(image_50, h, axis=0)
dxx50 = np.gradient(dx50, h, axis=1)
dyy50 = np.gradient(dy50, h, axis=0)

dx = dxx = 1.0
dt = 0.1   # You can reduce this if the solution becomes unstable
steps = 5

predicted_images = forward_euler_pde_mixed(image_45, dx, dxx, dt, steps)

# View Image 53
import matplotlib.pyplot as plt
plt.imshow(predicted_images[3], cmap='gray')
plt.title("Predicted Image 47")
plt.show()

truth = [image_45, image_46, image_47, image_48, image_49, image_50]
predicted = [predicted_images[i] for i in range(1, 6)]

fig, axes = plt.subplots(2, 5, figsize=(15, 6))

for i in range(5):
        # Actual images on the top row
    axes[0, i].imshow(truth[i], cmap='gray')
    axes[0, i].set_title(f"Actual {i+1}")
    axes[0, i].axis('off')

        # Predicted images on the bottom row
    axes[1, i].imshow(predicted_images[i], cmap='gray')
    axes[1, i].set_title(f"Predicted {i+1}")
    axes[1, i].axis('off')

plt.tight_layout()
plt.show()
```

APPENDIX C

CHAPTER 4 SOURCE CODES

Python Code for Translation and Expansion of Object (Circle Moving Left to Right and Expanding)

```
import os
import re
import glob
from PIL import Image
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import skimage as ski
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Lasso, LassoCV, LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
import seaborn as sns
from sklearn.ensemble import GradientBoostingRegressor
import pysindy as ps
import statsmodels.api as sm

image_1 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Expand_Data/Slide1.jpg", as_gray=True)
image_1 = np.where(image_1==1, 0, 1)
image_1 = ski.filters.gaussian(image_1, sigma=55)
image_1 = image_1 * 1e10
plt.imshow(image_1, cmap="gray")

h = 1

# Forward Difference for first derivatives and second derivatives.

dx1 = np.gradient(image_1, h, axis=1)
dy1 = np.gradient(image_1, h, axis=0)
dxy1 = np.gradient(dx1, h, axis=0)
dxx1 = np.gradient(dx1, h, axis=1)
dyy1 = np.gradient(dy1, h, axis=0)

image_2 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Expand_Data/Slide2.jpg", as_gray=True)
image_2 = np.where(image_2 == 1, 0, 1)
image_2 = ski.filters.gaussian(image_2, sigma=55)
image_2 = image_2 * 1e10
plt.imshow(image_2, cmap="gray")

dx2 = np.gradient(image_2, h, axis=1)
dy2 = np.gradient(image_2, h, axis=0)
```

```
dxy2 = np.gradient(dx2, h, axis=0)
dxx2 = np.gradient(dx2, h, axis=1)
dyy2 = np.gradient(dy2, h, axis=0)

image_3 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Expand_Data/Slide3.jpg", as_gray=True)
image_3 = np.where(image_3 == 1, 0, 1)
image_3 = ski.filters.gaussian(image_3, sigma=55)
image_3 = image_3 * 1e10
plt.imshow(image_3, cmap="gray")

dx3 = np.gradient(image_3, h, axis=1)
dy3 = np.gradient(image_3, h, axis=0)
dxy3 = np.gradient(dx3, h, axis=0)
dxx3 = np.gradient(dx3, h, axis=1)
dyy3 = np.gradient(dy3, h, axis=0)

image_4 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Expand_Data/Slide4.jpg", as_gray=True)
image_4 = np.where(image_4 == 1, 0, 1)
image_4 = ski.filters.gaussian(image_4, sigma=55)
image_4 = image_4 * 1e10
plt.imshow(image_4, cmap="gray")

dx4 = np.gradient(image_4, h, axis=1)
dy4 = np.gradient(image_4, h, axis=0)
dxy4 = np.gradient(dx4, h, axis=0)
dxx4 = np.gradient(dx4, h, axis=1)
dyy4 = np.gradient(dy4, h, axis=0)

image_5 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Expand_Data/Slide5.jpg", as_gray=True)
image_5 = np.where(image_5 == 1, 0, 1)
image_5 = ski.filters.gaussian(image_5, sigma=55)
image_5 = image_5 * 1e10
plt.imshow(image_5, cmap="gray")

dx5 = np.gradient(image_5, h, axis=1)
dy5 = np.gradient(image_5, h, axis=0)
dxy5 = np.gradient(dx5, h, axis=0)
dxx5 = np.gradient(dx5, h, axis=1)
dyy5 = np.gradient(dy5, h, axis=0)

image_6 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Expand_Data/Slide6.jpg", as_gray=True)
```

```
image_6 = np.where(image_6 == 1, 0, 1)
image_6 = ski.filters.gaussian(image_6, sigma=55)
image_6 = image_6 * 1e10
plt.imshow(image_6, cmap="gray")

dx6 = np.gradient(image_6, h, axis=1)
dy6 = np.gradient(image_6, h, axis=0)
dxy6 = np.gradient(dx6, h, axis=0)
dxx6 = np.gradient(dx6, h, axis=1)
dyy6 = np.gradient(dy6, h, axis=0)

image_7 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Expand_Data/Slide7.jpg", as_gray=True)
image_7 = np.where(image_7 == 1, 0, 1)
image_7 = ski.filters.gaussian(image_7, sigma=55)
image_7 = image_7 * 1e10
plt.imshow(image_7, cmap="gray")

dx7 = np.gradient(image_7, h, axis=1)
dy7 = np.gradient(image_7, h, axis=0)
dxy7 = np.gradient(dx7, h, axis=0)
dxx7 = np.gradient(dx7, h, axis=1)
dyy7 = np.gradient(dy7, h, axis=0)

image_8 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Expand_Data/Slide8.jpg", as_gray=True)
image_8 = np.where(image_8 == 1, 0, 1)
image_8 = ski.filters.gaussian(image_8, sigma=55)
image_8 = image_8 * 1e10
plt.imshow(image_8, cmap="gray")

dx8 = np.gradient(image_8, h, axis=1)
dy8 = np.gradient(image_8, h, axis=0)
dxy8 = np.gradient(dx8, h, axis=0)
dxx8 = np.gradient(dx8, h, axis=1)
dyy8 = np.gradient(dy8, h, axis=0)

image_9 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon
_Suzuki
/Expand_Data/Slide9.jpg", as_gray=True)
image_9 = np.where(image_9 == 1, 0, 1)
image_9 = ski.filters.gaussian(image_9, sigma=55)
image_9 = image_9 * 1e10
plt.imshow(image_9, cmap="gray")

dx9 = np.gradient(image_9, h, axis=1)
dy9 = np.gradient(image_9, h, axis=0)
```

```
dxy9 = np.gradient(dx9, h, axis=0)
dxx9 = np.gradient(dx9, h, axis=1)
dyy9 = np.gradient(dy9, h, axis=0)

image_10 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide10.jpg", as_gray=True)
image_10 = np.where(image_10 == 1, 0, 1)
image_10 = ski.filters.gaussian(image_10, sigma=55)
image_10 = image_10 * 1e10
plt.imshow(image_10, cmap="gray")

dx10 = np.gradient(image_10, h, axis=1)
dy10 = np.gradient(image_10, h, axis=0)
dxy10 = np.gradient(dx10, h, axis=0)
dxx10 = np.gradient(dx10, h, axis=1)
dyy10 = np.gradient(dy10, h, axis=0)

image_11 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide11.jpg", as_gray=True)
image_11 = np.where(image_11 == 1, 0, 1)
image_11 = ski.filters.gaussian(image_11, sigma=55)
image_11 = image_11 * 1e10
plt.imshow(image_11, cmap="gray")

dx11 = np.gradient(image_11, h, axis=1)
dy11 = np.gradient(image_11, h, axis=0)
dxy11 = np.gradient(dx11, h, axis=0)
dxx11 = np.gradient(dx11, h, axis=1)
dyy11 = np.gradient(dy11, h, axis=0)

image_12 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide12.jpg", as_gray=True)
image_12 = np.where(image_12==1, 0, 1)
image_12 = ski.filters.gaussian(image_12, sigma=55)
image_12 = image_12 * 1e10
plt.imshow(image_12, cmap="gray")

dx12 = np.gradient(image_12, h, axis=1)
dy12 = np.gradient(image_12, h, axis=0)
dxy12 = np.gradient(dx12, h, axis=0)
dxx12 = np.gradient(dx12, h, axis=1)
dyy12 = np.gradient(dy12, h, axis=0)

image_13 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide13.jpg", as_gray=True)
```

```
image_13 = np.where(image_13 == 1, 0, 1)
image_13 = ski.filters.gaussian(image_13, sigma=55)
image_13 = image_13 * 1e10
plt.imshow(image_13, cmap="gray")

dx13 = np.gradient(image_13, h, axis=1)
dy13 = np.gradient(image_13, h, axis=0)
dxy13 = np.gradient(dx13, h, axis=0)
dxx13 = np.gradient(dx13, h, axis=1)
dyy13 = np.gradient(dy13, h, axis=0)

image_14 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide14.jpg", as_gray=True)
image_14 = np.where(image_14 == 1, 0, 1)
image_14 = ski.filters.gaussian(image_14, sigma=55)
image_14 = image_14 * 1e10
plt.imshow(image_14, cmap="gray")

dx14 = np.gradient(image_14, h, axis=1)
dy14 = np.gradient(image_14, h, axis=0)
dxy14 = np.gradient(dx14, h, axis=0)
dxx14 = np.gradient(dx14, h, axis=1)
dyy14 = np.gradient(dy14, h, axis=0)

image_15 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide15.jpg", as_gray=True)
image_15 = np.where(image_15 == 1, 0, 1)
image_15 = ski.filters.gaussian(image_15, sigma=55)
image_15 = image_15 * 1e10
plt.imshow(image_15, cmap="gray")

dx15 = np.gradient(image_15, h, axis=1)
dy15 = np.gradient(image_15, h, axis=0)
dxy15 = np.gradient(dx15, h, axis=0)
dxx15 = np.gradient(dx15, h, axis=1)
dyy15 = np.gradient(dy15, h, axis=0)

image_16 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide15.jpg", as_gray=True)
image_16 = np.where(image_16 == 1, 0, 1)
image_16 = ski.filters.gaussian(image_16, sigma=55)
image_16 = image_16 * 1e10
plt.imshow(image_16, cmap="gray")

dx16 = np.gradient(image_16, h, axis=1)
dy16 = np.gradient(image_16, h, axis=0)
```

```
dxy16 = np.gradient(dx16, h, axis=0)
dxx16 = np.gradient(dx16, h, axis=1)
dyy16 = np.gradient(dy16, h, axis=0)

image_17 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide17.jpg", as_gray=True)
image_17 = np.where(image_17 == 1, 0, 1)
image_17 = ski.filters.gaussian(image_17, sigma=55)
image_17 = image_17 * 1e10
plt.imshow(image_17, cmap="gray")

dx17 = np.gradient(image_17, h, axis=1)
dy17 = np.gradient(image_17, h, axis=0)
dxy17 = np.gradient(dx17, h, axis=0)
dxx17 = np.gradient(dx17, h, axis=1)
dyy17 = np.gradient(dy17, h, axis=0)

image_18 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide18.jpg", as_gray=True)
image_18 = np.where(image_18 == 1, 0, 1)
image_18 = ski.filters.gaussian(image_18, sigma=55)
image_18 = image_18 * 1e10
plt.imshow(image_18, cmap="gray")

dx18 = np.gradient(image_18, h, axis=1)
dy18 = np.gradient(image_18, h, axis=0)
dxy18 = np.gradient(dx18, h, axis=0)
dxx18 = np.gradient(dx18, h, axis=1)
dyy18 = np.gradient(dy18, h, axis=0)

image_19 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide19.jpg", as_gray=True)
image_19 = np.where(image_19 == 1, 0, 1)
image_19 = ski.filters.gaussian(image_19, sigma=55)
image_19 = image_19 * 1e10
plt.imshow(image_19, cmap="gray")

dx19 = np.gradient(image_19, h, axis=1)
dy19 = np.gradient(image_19, h, axis=0)
dxy19 = np.gradient(dx19, h, axis=0)
dxx19 = np.gradient(dx19, h, axis=1)
dyy19 = np.gradient(dy19, h, axis=0)

image_20 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide20.jpg", as_gray=True)
```

```
image_20 = np.where(image_20 == 1, 0, 1)
image_20 = ski.filters.gaussian(image_20, sigma=55)
image_20 = image_20 * 1e10
plt.imshow(image_20, cmap="gray")

dx20 = np.gradient(image_20, h, axis=1)
dy20 = np.gradient(image_20, h, axis=0)
dxy20 = np.gradient(dx20, h, axis=0)
dxx20 = np.gradient(dx20, h, axis=1)
dyy20 = np.gradient(dy20, h, axis=0)

image_21 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide21.jpg", as_gray=True)
image_21 = np.where(image_21 == 1, 0, 1)
image_21 = ski.filters.gaussian(image_21, sigma=55)
image_21 = image_21 * 1e10
plt.imshow(image_21, cmap="gray")

dx21 = np.gradient(image_21, h, axis=1)
dy21 = np.gradient(image_21, h, axis=0)
dxy21 = np.gradient(dx21, h, axis=0)
dxx21 = np.gradient(dy21, h, axis=1)
dyy21 = np.gradient(dy21, h, axis=0)

image_22 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide22.jpg", as_gray=True)
image_22 = np.where(image_22 == 1, 0, 1)
image_22 = ski.filters.gaussian(image_22, sigma=55)
image_22 = image_22 * 1e10
plt.imshow(image_22, cmap="gray")

dx22 = np.gradient(image_22, h, axis=1)
dy22 = np.gradient(image_22, h, axis=0)
dxy22 = np.gradient(dx22, h, axis=0)
dxx22 = np.gradient(dx22, h, axis=1)
dyy22 = np.gradient(dy22, h, axis=0)

image_23 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide23.jpg", as_gray=True)
image_23 = np.where(image_23 == 1, 0, 1)
image_23 = ski.filters.gaussian(image_23, sigma=55)
image_23 = image_23 * 1e10
plt.imshow(image_23, cmap="gray")

dx23 = np.gradient(image_23, h, axis=1)
dy23 = np.gradient(image_23, h, axis=0)
```

```
dxy23 = np.gradient(dx23, h, axis=0)
dxx23 = np.gradient(dx23, h, axis=1)
dyy23 = np.gradient(dy23, h, axis=0)

image_24 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide24.jpg", as_gray=True)
image_24 = np.where(image_24 == 1, 0, 1)
image_24 = ski.filters.gaussian(image_24, sigma=55)
image_24 = image_24 * 1e10
plt.imshow(image_24, cmap="gray")

dx24 = np.gradient(image_24, h, axis=1)
dy24 = np.gradient(image_24, h, axis=0)
dxy24 = np.gradient(dx24, h, axis=0)
dxx24 = np.gradient(dx24, h, axis=1)
dyy24 = np.gradient(dy24, h, axis=0)

image_25 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide25.jpg", as_gray=True)
image_25 = np.where(image_25 == 1, 0, 1)
image_25 = ski.filters.gaussian(image_25, sigma=55)
image_25 = image_25 * 1e10
plt.imshow(image_25, cmap="gray")

dx25 = np.gradient(image_25, h, axis=1)
dy25 = np.gradient(image_25, h, axis=0)
dxy25 = np.gradient(dx25, h, axis=0)
dxx25 = np.gradient(dx25, h, axis=1)
dyy25 = np.gradient(dy25, h, axis=0)

image_26 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide26.jpg", as_gray=True)
image_26 = np.where(image_26 == 1, 0, 1)
image_26 = ski.filters.gaussian(image_26, sigma=55)
image_26 = image_26 * 1e10
plt.imshow(image_26, cmap="gray")

dx26 = np.gradient(image_26, h, axis=1)
dy26 = np.gradient(image_26, h, axis=0)
dxy26 = np.gradient(dx26, h, axis=0)
dxx26 = np.gradient(dx26, h, axis=1)
dyy26 = np.gradient(dy26, h, axis=0)

image_27 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide27.jpg", as_gray=True)
```

```
image_27 = np.where(image_27 == 1, 0, 1)
image_27 = ski.filters.gaussian(image_27, sigma=55)
image_27 = image_27 * 1e10
plt.imshow(image_27, cmap="gray")

dx27 = np.gradient(image_27, h, axis=1)
dy27 = np.gradient(image_27, h, axis=0)
dxy27 = np.gradient(dx27, h, axis=0)
dxx27 = np.gradient(dx27, h, axis=1)
dyy27 = np.gradient(dy27, h, axis=0)

image_28 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide28.jpg", as_gray=True)
image_28 = np.where(image_28 == 1, 0, 1)
image_28 = ski.filters.gaussian(image_28, sigma=55)
image_28 = image_28 * 1e10
plt.imshow(image_28, cmap="gray")

dx28 = np.gradient(image_28, h, axis=1)
dy28 = np.gradient(image_28, h, axis=0)
dxy28 = np.gradient(dx28, h, axis=0)
dxx28 = np.gradient(dx28, h, axis=1)
dyy28 = np.gradient(dy28, h, axis=0)

image_29 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide29.jpg", as_gray=True)
image_29 = np.where(image_29 == 1, 0, 1)
image_29 = ski.filters.gaussian(image_29, sigma=55)
image_29 = image_29 * 1e10
plt.imshow(image_29, cmap="gray")

dx29 = np.gradient(image_29, h, axis=1)
dy29 = np.gradient(image_29, h, axis=0)
dxy29 = np.gradient(dx29, h, axis=0)
dxx29 = np.gradient(dx29, h, axis=1)
dyy29 = np.gradient(dy29, h, axis=0)

image_30 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide30.jpg", as_gray=True)
image_30 = np.where(image_30 == 1, 0, 1)
image_30 = ski.filters.gaussian(image_30, sigma=55)
image_30 = image_30 * 1e10
plt.imshow(image_30, cmap="gray")

dx30 = np.gradient(image_30, h, axis=1)
dy30 = np.gradient(image_30, h, axis=0)
```

```
dxy30 = np.gradient(dx30, h, axis=0)
dxx30 = np.gradient(dx30, h, axis=1)
dyy30 = np.gradient(dy30, h, axis=0)

image_31 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide31.jpg", as_gray=True)
image_31 = np.where(image_31 == 1, 0, 1)
image_31 = ski.filters.gaussian(image_31, sigma=55)
image_31 = image_31 * 1e10
plt.imshow(image_31, cmap="gray")

dx31 = np.gradient(image_31, h, axis=1)
dy31 = np.gradient(image_31, h, axis=0)
dxy31 = np.gradient(dx31, h, axis=0)
dxx31 = np.gradient(dx31, h, axis=1)
dyy31 = np.gradient(dy31, h, axis=0)

image_32 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide32.jpg", as_gray=True)
image_32 = np.where(image_32 == 1, 0, 1)
image_32 = ski.filters.gaussian(image_32, sigma=55)
image_32 = image_32 * 1e10
plt.imshow(image_32, cmap="gray")

dx32 = np.gradient(image_32, h, axis=1)
dy32 = np.gradient(image_32, h, axis=0)
dxy32 = np.gradient(dx32, h, axis=0)
dxx32 = np.gradient(dx32, h, axis=1)
dyy32 = np.gradient(dy32, h, axis=0)

image_33 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide33.jpg", as_gray=True)
image_33 = np.where(image_33 == 1, 0, 1)
image_33 = ski.filters.gaussian(image_33, sigma=55)
image_33 = image_33 * 1e10
plt.imshow(image_33, cmap="gray")

dx33 = np.gradient(image_33, h, axis=1)
dy33 = np.gradient(image_33, h, axis=0)
dxy33 = np.gradient(dx33, h, axis=0)
dxx33 = np.gradient(dx33, h, axis=1)
dyy33 = np.gradient(dy33, h, axis=0)

image_34 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide34.jpg", as_gray=True)
```

```
image_34 = np.where(image_34 == 1, 0, 1)
image_34 = ski.filters.gaussian(image_34, sigma=55)
image_34 = image_34 * 1e10
plt.imshow(image_34, cmap="gray")

dx34 = np.gradient(image_34, h, axis=1)
dy34 = np.gradient(image_34, h, axis=0)
dxy34 = np.gradient(dx34, h, axis=0)
dxx34 = np.gradient(dx34, h, axis=1)
dyy34 = np.gradient(dy34, h, axis=0)

image_35 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide35.jpg", as_gray=True)
image_35 = np.where(image_35 == 1, 0, 1)
image_35 = ski.filters.gaussian(image_35, sigma=55)
image_35 = image_35 * 1e10
plt.imshow(image_35, cmap="gray")

dx35 = np.gradient(image_35, h, axis=1)
dy35 = np.gradient(image_35, h, axis=0)
dxy35 = np.gradient(dx35, h, axis=0)
dxx35 = np.gradient(dx35, h, axis=1)
dyy35 = np.gradient(dy35, h, axis=0)

image_36 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide36.jpg", as_gray=True)
image_36 = np.where(image_36 == 1, 0, 1)
image_36 = ski.filters.gaussian(image_36, sigma=55)
image_36 = image_36 * 1e10
plt.imshow(image_36, cmap="gray")

dx36 = np.gradient(image_36, h, axis=1)
dy36 = np.gradient(image_36, h, axis=0)
dxy36 = np.gradient(dx36, h, axis=0)
dxx36 = np.gradient(dx36, h, axis=1)
dyy36 = np.gradient(dy36, h, axis=0)

image_37 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide37.jpg", as_gray=True)
image_37 = np.where(image_37 == 1, 0, 1)
image_37 = ski.filters.gaussian(image_37, sigma=55)
image_37 = image_37 * 1e10
plt.imshow(image_37, cmap="gray")

dx37 = np.gradient(image_37, h, axis=1)
dy37 = np.gradient(image_37, h, axis=0)
```

```
dxy37 = np.gradient(dx37, h, axis=0)
dxx37 = np.gradient(dx37, h, axis=1)
dyy37 = np.gradient(dy37, h, axis=0)

image_38 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide38.jpg", as_gray=True)
image_38 = np.where(image_38 == 1, 0, 1)
image_38 = ski.filters.gaussian(image_38, sigma=55)
image_38 = image_38 * 1e10
plt.imshow(image_38, cmap="gray")

dx38 = np.gradient(image_38, h, axis=1)
dy38 = np.gradient(image_38, h, axis=0)
dxy38 = np.gradient(dx38, h, axis=0)
dxx38 = np.gradient(dx38, h, axis=1)
dyy38 = np.gradient(dy38, h, axis=0)

image_39 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide39.jpg", as_gray=True)
image_39 = np.where(image_39 == 1, 0, 1)
image_39 = ski.filters.gaussian(image_39, sigma=55)
image_39 = image_39 * 1e10
plt.imshow(image_39, cmap="gray")

dx39 = np.gradient(image_39, h, axis=1)
dy39 = np.gradient(image_39, h, axis=0)
dxy39 = np.gradient(dx39, h, axis=0)
dxx39 = np.gradient(dx39, h, axis=1)
dyy39 = np.gradient(dy39, h, axis=0)

image_40 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide40.jpg", as_gray=True)
image_40 = np.where(image_40 == 1, 0, 1)
image_40 = ski.filters.gaussian(image_40, sigma=55)
image_40 = image_40 * 1e10
plt.imshow(image_40, cmap="gray")

dx40 = np.gradient(image_40, h, axis=1)
dy40 = np.gradient(image_40, h, axis=0)
dxy40 = np.gradient(dx40, h, axis=0)
dxx40 = np.gradient(dx40, h, axis=1)
dyy40 = np.gradient(dy40, h, axis=0)

image_41 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide41.jpg", as_gray=True)
```

```
image_41 = np.where(image_41 == 1, 0, 1)
image_41 = ski.filters.gaussian(image_41, sigma=55)
image_41 = image_41 * 1e10
plt.imshow(image_41, cmap="gray")

dx41 = np.gradient(image_41, h, axis=1)
dy41 = np.gradient(image_41, h, axis=0)
dxy41 = np.gradient(dx41, h, axis=0)
dxx41 = np.gradient(dx41, h, axis=1)
dyy41 = np.gradient(dy41, h, axis=0)

image_42 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide42.jpg", as_gray=True)
image_42 = np.where(image_42 == 1, 0, 1)
image_42 = ski.filters.gaussian(image_42, sigma=55)
image_42 = image_42 * 1e10
plt.imshow(image_42, cmap="gray")

dx42 = np.gradient(image_42, h, axis=1)
dy42 = np.gradient(image_42, h, axis=0)
dxy42 = np.gradient(dx42, h, axis=0)
dxx42 = np.gradient(dx42, h, axis=1)
dyy42 = np.gradient(dy42, h, axis=0)

image_43 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide43.jpg", as_gray=True)
image_43 = np.where(image_43 == 1, 0, 1)
image_43 = ski.filters.gaussian(image_43, sigma=55)
image_43 = image_43 * 1e10
plt.imshow(image_43, cmap="gray")

dx43 = np.gradient(image_43, h, axis=1)
dy43 = np.gradient(image_43, h, axis=0)
dxy43 = np.gradient(dx43, h, axis=0)
dxx43 = np.gradient(dx43, h, axis=1)
dyy43 = np.gradient(dy43, h, axis=0)

image_44 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide44.jpg", as_gray=True)
image_44 = np.where(image_44 == 1, 0, 1)
image_44 = ski.filters.gaussian(image_44, sigma=55)
image_44 = image_44 * 1e10
plt.imshow(image_44, cmap="gray")

dx44 = np.gradient(image_44, h, axis=1)
dy44 = np.gradient(image_44, h, axis=0)
```

```
dxy44 = np.gradient(dx44, h, axis=0)
dxx44 = np.gradient(dx44, h, axis=1)
dyy44 = np.gradient(dy44, h, axis=0)

image_45 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide45.jpg", as_gray=True)
image_45 = np.where(image_45 == 1, 0, 1)
image_45 = ski.filters.gaussian(image_45, sigma=55)
image_45 = image_45 * 1e10
plt.imshow(image_45, cmap="gray")

dx45 = np.gradient(image_45, h, axis=1)
dy45 = np.gradient(image_45, h, axis=0)
dxy45 = np.gradient(dx45, h, axis=0)
dxx45 = np.gradient(dx45, h, axis=1)
dyy45 = np.gradient(dy45, h, axis=0)

image_46 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide46.jpg", as_gray=True)
image_46 = np.where(image_46 == 1, 0, 1)
image_46 = ski.filters.gaussian(image_46, sigma=55)
image_46 = image_46 * 1e10
plt.imshow(image_46, cmap="gray")

dx46 = np.gradient(image_46, h, axis=1)
dy46 = np.gradient(image_46, h, axis=0)
dxy46 = np.gradient(dx46, h, axis=0)
dxx46 = np.gradient(dx46, h, axis=1)
dyy46 = np.gradient(dy46, h, axis=0)

image_47 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide47.jpg", as_gray=True)
image_47 = np.where(image_47 == 1, 0, 1)
image_47 = ski.filters.gaussian(image_47, sigma=55)
image_47 = image_47 * 1e10
plt.imshow(image_47, cmap="gray")

dx47 = np.gradient(image_47, h, axis=1)
dy47 = np.gradient(image_47, h, axis=0)
dxy47 = np.gradient(dx47, h, axis=0)
dxx47 = np.gradient(dx47, h, axis=1)
dyy47 = np.gradient(dy47, h, axis=0)

image_48 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide48.jpg", as_gray=True)
```

```
image_48 = np.where(image_48 == 1, 0, 1)
image_48 = ski.filters.gaussian(image_48, sigma=55)
image_48 = image_48 * 1e10
plt.imshow(image_48, cmap="gray")

dx48 = np.gradient(image_48, h, axis=1)
dy48 = np.gradient(image_48, h, axis=0)
dxy48 = np.gradient(dx48, h, axis=0)
dxx48 = np.gradient(dx48, h, axis=1)
dyy48 = np.gradient(dy48, h, axis=0)

image_49 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide49.jpg", as_gray=True)
image_49 = np.where(image_49 == 1, 0, 1)
image_49 = ski.filters.gaussian(image_49, sigma=55)
image_49 = image_49 * 1e10
plt.imshow(image_49, cmap="gray")

dx49 = np.gradient(image_49, h, axis=1)
dy49 = np.gradient(image_49, h, axis=0)
dxy49 = np.gradient(dx49, h, axis=0)
dxx49 = np.gradient(dx49, h, axis=1)
dyy49 = np.gradient(dy49, h, axis=0)

image_50 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide50.jpg", as_gray=True)
image_50 = np.where(image_50 == 1, 0, 1)
image_50 = ski.filters.gaussian(image_50, sigma=55)
image_50 = image_50 * 1e10
plt.imshow(image_50, cmap="gray")

dx50 = np.gradient(image_50, h, axis=1)
dy50 = np.gradient(image_50, h, axis=0)
dxy50 = np.gradient(dx50, h, axis=0)
dxx50 = np.gradient(dx50, h, axis=1)
dyy50 = np.gradient(dy50, h, axis=0)

train = pd.read_csv("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon_Suz
uki
/moving_expand_train.csv")
test = pd.read_csv("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon_Suzu
ki
/moving_expanding_test.csv")
train

test
```

```
data_X = train.drop(columns=["dfudt"])
data_y = train["dfudt"]

data_mat_X = data_X.values
data_mat_y = data_y.values

reg_mod = sm.OLS(data_mat_y, data_mat_X)
results = reg_mod.fit()
summary = results.summary()
print(summary)

lasso_ode = Lasso(alpha=0.1)
lasso_ode.fit(data_mat_X, data_mat_y)

print("LASSO Coefficients:", lasso_ode.coef_)

# Computing prediction accuracy from testing set.

y_pred_ode = lasso_ode.predict(data_mat_X)
mse_ode = mean_squared_error(data_mat_y, y_pred_ode)
print("Mean Squared Error:", mse_ode)

# Tuning alpha parameter via CV with cv=5 folds.

lasso_cv_ode = LassoCV(alphas=np.logspace(-3,1,100), cv=5)
lasso_cv_ode.fit(data_mat_X, data_mat_y)

print("Optimal alpha:", lasso_cv_ode.alpha_)

# Checking number of zero coefficients (how many features dropped)

check = sum(lasso_cv_ode.coef_ == 0)
check

# LASSO Regularized regression for feature selection.

lasso_ode = Lasso(alpha=lasso_cv_ode.alpha_)
lasso_ode.fit(data_mat_X, data_mat_y)

print("LASSO Coefficients:", lasso_ode.coef_)

data_mat_X = train.drop(columns=["dy", "dfudt"])
data_mat_X.values

reg_mod2 = sm.OLS(data_mat_y, data_mat_X)
results2 = reg_mod2.fit()
summary2 = results2.summary()
print(summary2)
```

```
from scipy.ndimage import gaussian_filter

def forward_euler_pde_mixed(u0, dx, dy, dt, steps):
    """
    Forward Euler integration for the PDE:
    u_t = 0.0321u - 10.9031u_x - 3.3925u_xy + 138.6948u_xx

    Parameters:
        u0: np.ndarray, initial image
        dx, dy: spatial resolutions
        dt: time step
        steps: number of steps (e.g., 5 for Image_51 to Image_55)

    Returns:
        List of predicted images
    """
    u = u0.copy()
    images = [u.copy()]

    for _ in range(steps):
        # First derivative in x
        u_x = (np.roll(u, -1, axis=0) - np.roll(u, 1, axis=0)) / (2 * dx)

        # First derivative in y
        u_y = (np.roll(u, -1, axis=1) - np.roll(u, 1, axis=1)) / (2 *dy)

        # Second derivative in x
        u_xx = (np.roll(u, -1, axis=0) - 2*u + np.roll(u, 1, axis=0)) / (dx**
2)

        # Second derivative in y
        u_yy = (np.roll(u, -1, axis=1) - 2*u + np.roll(u, 1, axis=1)) / (dy**
2)

        # Mixed derivative u_xy
        u_ip1_jp1 = np.roll(np.roll(u, -1, axis=0), -1, axis=1)
        u_ip1_jm1 = np.roll(np.roll(u, -1, axis=0), 1, axis=1)
        u_im1_jp1 = np.roll(np.roll(u, 1, axis=0), -1, axis=1)
        u_im1_jm1 = np.roll(np.roll(u, 1, axis=0), 1, axis=1)
        u_xy = (u_ip1_jp1 - u_ip1_jm1 - u_im1_jp1 + u_im1_jm1) / (4 * dx * dy)

        # Time derivative

        # Data Driven

        u_t = (0.0321 * u - 10.9031 * u_x - 3.3925 * u_xy + 138.6948 * u_xx +
 0.0031 * u_yy)
```

```
        # Physics Driven

        #u_t = (0.0321 * u - 10.9031 * u_x + 138.6570 * u_xx + 0.0031 * u_yy)
        # - 4.728e+04 * u_yy
        # - 3.3925 * u_xy

        # Forward Euler update
        u = u + dt * u_t
        #u = gaussian_filter(u, sigma=10)
        images.append(u.copy())

    return images

h=1

image_45 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide45.jpg", as_gray=True)
image_45 = np.where(image_45 == 1, 0, 1)
image_45 = ski.filters.gaussian(image_45, sigma=55)
image_45 = image_45 * 1e10
plt.imshow(image_45, cmap="gray")

dx45 = np.gradient(image_45, h, axis=1)
dy45 = np.gradient(image_45, h, axis=0)
dxy45 = np.gradient(dx45, h, axis=0)
dxx45 = np.gradient(dx45, h, axis=1)
dyy45 = np.gradient(dy45, h, axis=0)

image_46 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide46.jpg", as_gray=True)
image_46 = np.where(image_46 == 1, 0, 1)
image_46 = ski.filters.gaussian(image_46, sigma=55)
image_46 = image_46 * 1e10
plt.imshow(image_46, cmap="gray")

dx46 = np.gradient(image_46, h, axis=1)
dy46 = np.gradient(image_46, h, axis=0)
dxy46 = np.gradient(dx46, h, axis=0)
dxx46 = np.gradient(dx46, h, axis=1)
dyy46 = np.gradient(dy46, h, axis=0)

image_47 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide47.jpg", as_gray=True)
image_47 = np.where(image_47 == 1, 0, 1)
```

```
image_47 = ski.filters.gaussian(image_47, sigma=55)
image_47 = image_47 * 1e10
plt.imshow(image_47, cmap="gray")

dx47 = np.gradient(image_47, h, axis=1)
dy47 = np.gradient(image_47, h, axis=0)
dxy47 = np.gradient(dx47, h, axis=0)
dxx47 = np.gradient(dx47, h, axis=1)
dyy47 = np.gradient(dy47, h, axis=0)

image_48 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide48.jpg", as_gray=True)
image_48 = np.where(image_48 == 1, 0, 1)
image_48 = ski.filters.gaussian(image_48, sigma=55)
image_48 = image_48 * 1e10
plt.imshow(image_48, cmap="gray")

dx48 = np.gradient(image_48, h, axis=1)
dy48 = np.gradient(image_48, h, axis=0)
dxy48 = np.gradient(dx48, h, axis=0)
dxx48 = np.gradient(dx48, h, axis=1)
dyy48 = np.gradient(dy48, h, axis=0)

image_49 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide49.jpg", as_gray=True)
image_49 = np.where(image_49 == 1, 0, 1)
image_49 = ski.filters.gaussian(image_49, sigma=55)
image_49 = image_49 * 1e10
plt.imshow(image_49, cmap="gray")

dx49 = np.gradient(image_49, h, axis=1)
dy49 = np.gradient(image_49, h, axis=0)
dxy49 = np.gradient(dx49, h, axis=0)
dxx49 = np.gradient(dx49, h, axis=1)
dyy49 = np.gradient(dy49, h, axis=0)

image_50 = ski.io.imread("C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joo
n_Suzuki
/Expand_Data/Slide50.jpg", as_gray=True)
image_50 = np.where(image_50 == 1, 0, 1)
image_50 = ski.filters.gaussian(image_50, sigma=55)
image_50 = image_50 * 1e10
plt.imshow(image_50, cmap="gray")

dx50 = np.gradient(image_50, h, axis=1)
dy50 = np.gradient(image_50, h, axis=0)
dxy50 = np.gradient(dx50, h, axis=0)
```

```
dxx50 = np.gradient(dx50, h, axis=1)
dyy50 = np.gradient(dy50, h, axis=0)

dx = dy = 1.0
dt = 0.1   # You can reduce this if the solution becomes unstable
steps = 5

predicted_images = forward_euler_pde_mixed(image_45, dx, dy, dt, steps)

# View Image 53
import matplotlib.pyplot as plt
plt.imshow(predicted_images[4], cmap='gray')
plt.title("Predicted Image 47")
plt.show()

truth = [image_45, image_46, image_47, image_48, image_49, image_50]
predicted = [predicted_images[i] for i in range(1, 6)]

fig, axes = plt.subplots(2, 5, figsize=(15, 6))

for i in range(5):
        # Actual images on the top row
    axes[0, i].imshow(truth[i], cmap='gray')
    axes[0, i].set_title(f"Actual {i+1}")
    axes[0, i].axis('off')

        # Predicted images on the bottom row
    axes[1, i].imshow(predicted_images[i], cmap='gray')
    axes[1, i].set_title(f"Predicted {i+1}")
    axes[1, i].axis('off')

plt.tight_layout()
plt.show()
```

APPENDIX D

CHAPTER 5 SOURCE CODES

Python Code for Proper Orthogonal Decomposition and ODE System Analysis

```
import numpy as np
import matplotlib.pyplot as plt
import os
from PIL import Image
import pandas as pd
from scipy.optimize import curve_fit

# Step 1: Load images from frame_start.jpg to frame_end.jpg
def load_images(folder='C:/Users/coryg/OneDrive/Desktop/
STAT_698_Thesis_Joon_Suzuki/Waterdrop_data', prefix='frame_', ext='.jpg', sta
rt=0, end=49):
    snapshots = []
    image_shape = None
    for i in range(start, end + 1):
        filename = os.path.join(folder, f"{prefix}{i:03d}{ext}")
        img = Image.open(filename).convert('L')
        if image_shape is None:
            image_shape = img.size[::-1]  # (height, width)
        snapshots.append(np.array(img).flatten())
    return np.array(snapshots).T, image_shape

# Step 2: Perform POD – with or without mean removal
def perform_pod(data_matrix, with_mean_removal=True):
    if with_mean_removal:
        mean_field = np.mean(data_matrix, axis=1, keepdims=True)
        fluctuations = data_matrix - mean_field
        U, S, Vt = np.linalg.svd(fluctuations, full_matrices=False)
        modes = U
        coeffs = np.diag(S) @ Vt
    else:
        mean_field = np.zeros((data_matrix.shape[0], 1))  # No subtraction
        U, S, Vt = np.linalg.svd(data_matrix, full_matrices=False)
        modes = U
        coeffs = np.diag(S) @ Vt
    return modes, coeffs, S, mean_field

# Step 3: Reconstruct using first r modes
def reconstruct(data_mean, modes, coeffs, r):
    approx = data_mean + modes[:, :r] @ coeffs[:r, :]
    return approx

# Step 4: Show POD coefficients as table
def show_coeff_table(coeffs, r):
    df = pd.DataFrame(coeffs[:r, :].T, columns=[f"Mode {i+1}" for i in range
(r)])
```

```
    print("\nPOD Coefficients (first r modes across snapshots):")
    print(df.round(4).to_string(index=False))

# Step 5: Plot original vs reconstructed frame
def plot_comparison(original, reconstructed, image_shape, frame_idx, offset):
    plt.figure(figsize=(10, 4))

    plt.subplot(1, 2, 1)
    plt.imshow(original.reshape(image_shape), cmap='gray')
    plt.title(f"Original Frame {frame_idx + offset:03d}")
    plt.axis('off')

    plt.subplot(1, 2, 2)
    plt.imshow(reconstructed.reshape(image_shape), cmap='gray')
    plt.title("Reconstructed")
    plt.axis('off')

    plt.tight_layout()
    plt.show()

# Main
if __name__ == "__main__":
    r = 6  # Number of modes
    target_frame = 29
    frame_start = 20
    frame_end = 49

    data_matrix, image_shape = load_images(start=frame_start, end=frame_end)

    # Toggle this to True/False for comparison
    use_mean_removal = False

    modes, coeffs, S, mean_field = perform_pod(data_matrix, with_mean_removal
=use_mean_removal)
    approx_matrix = reconstruct(mean_field, modes, coeffs, r)

    show_coeff_table(coeffs, r)
    plot_comparison(
        data_matrix[:, target_frame],
        approx_matrix[:, target_frame],
        image_shape,
        target_frame,
        offset=frame_start
    )

# Step 6: Plot time evolution of POD coefficients for first r modes
def plot_pod_coefficients(coeffs, r):
    plt.figure(figsize=(10, 6))
```

```
    time_steps = np.arange(coeffs.shape[1])
    for i in range(r-1):
        plt.plot(time_steps, coeffs[i+1, :], label=f"Mode {i+1}")
    plt.xlabel("Snapshot Index")
    plt.ylabel("Coefficient Value")
    plt.title(f"POD Coefficients (First {r} Modes)")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

# Call this function
plot_pod_coefficients(coeffs, r)

from sklearn.linear_model import LassoCV

time = np.arange(coeffs.shape[1])
max_freq = 10  # maximum harmonic frequency to test

# Create harmonic feature matrix for LASSO: sin(kx), cos(kx)
X = np.column_stack([
    np.sin(2 * np.pi * k * time / len(time)) for k in range(1, max_freq + 1)
] + [
    np.cos(2 * np.pi * k * time / len(time)) for k in range(1, max_freq + 1)
])
feature_names = [f"sin({k}f)" for k in range(1, max_freq + 1)] + [f"cos({k}f)
" for k in range(1, max_freq + 1)]

# Apply LASSO to the first 3 POD mode time series
lasso_results = {}
for mode_idx in range(6):
    y = coeffs[mode_idx, :]
    lasso = LassoCV(cv=5).fit(X, y)
    y_pred = lasso.predict(X)
    selected_features = [(name, coef) for name, coef in zip(feature_names, la
sso.coef_) if abs(coef) > 1e-2]

    lasso_results[f"Mode {mode_idx+1}"] = {
        "model": lasso,
        "prediction": y_pred,
        "selected_features": selected_features
    }

    # Plot
    plt.figure(figsize=(8, 3))
    plt.plot(time, y, 'b.-', label='Original')
    plt.plot(time, y_pred, 'r--', label='LASSO Fit')
    plt.title(f"POD Mode {mode_idx+1} - Sparse Harmonic Fit")
    plt.legend()
```

```
    plt.tight_layout()

# import pandas as pd

# Display selected harmonic features for each mode

results_summary = []
for mode, result in lasso_results.items():
    for term, coef in result["selected_features"]:
        results_summary.append({"Mode": mode, "Term": term, "Coefficient": co
ef})

#print(results_summary)
df_results = pd.DataFrame(results_summary)
df_results

def mode1(t):
    return (-0.0483*np.sin(2*np.pi*t) + 0.1467*np.sin(3*np.pi*t) + 0.0222*np.
sin(4*np.pi*t) +
            0.0122*np.sin(5*np.pi*t) + 0.0314*np.cos(1*np.pi*t) + 0.0930*np.c
os(2*np.pi*t) +
            -0.0104*np.cos(4*np.pi*t))

def mode2(t):
    return (0.0196*np.sin(1*np.pi*t) + 0.0877*np.sin(2*np.pi*t) - 0.0356*np.s
in(3*np.pi*t) -
            0.0265*np.sin(4*np.pi*t) - 0.0114*np.sin(5*np.pi*t) + 0.0341*np.c
os(1*np.pi*t) +
            0.0840*np.cos(2*np.pi*t) - 0.1435*np.cos(3*np.pi*t) - 0.0164*np.c
os(4*np.pi*t) -
            0.0102*np.cos(5*np.pi*t))

def mode3(t):
    return (0.0543*np.sin(1*np.pi*t) + 0.1506*np.sin(2*np.pi*t) + 0.0414*np.s
in(3*np.pi*t) +
            0.0308*np.sin(4*np.pi*t) + 0.0148*np.sin(5*np.pi*t) + 0.0119*np.s
in(6*np.pi*t) +
            0.0519*np.cos(1*np.pi*t) + 0.0886*np.cos(3*np.pi*t))

def mode4(t):
    return (-0.0231*np.sin(1*np.pi*t) - 0.0162*np.sin(2*np.pi*t) - 0.0988*np.
sin(3*np.pi*t) +
            0.0344*np.sin(4*np.pi*t) + 0.0154*np.sin(5*np.pi*t) + 0.1430*np.c
os(2*np.pi*t) +
            0.0822*np.cos(3*np.pi*t) + 0.0215*np.cos(4*np.pi*t))

def mode5(t):
    return (0.1140*np.sin(1*np.pi*t) - 0.0613*np.sin(2*np.pi*t) - 0.0396*np.s
in(3*np.pi*t) +
```

```
            0.0266*np.sin(4*np.pi*t) + 0.0656*np.sin(5*np.pi*t) + 0.0264*np.s
in(6*np.pi*t) +
            0.0175*np.sin(7*np.pi*t) + 0.0147*np.sin(8*np.pi*t) + 0.0126*np.s
in(9*np.pi*t) +
            0.0104*np.sin(10*np.pi*t) + 0.0845*np.cos(1*np.pi*t) - 0.0220*np.
cos(2*np.pi*t) -
            0.0302*np.cos(3*np.pi*t) + 0.0679*np.cos(4*np.pi*t) + 0.0111*np.c
os(5*np.pi*t))

def mode6(t):
    return (0.0432*np.sin(1*np.pi*t) - 0.0179*np.sin(2*np.pi*t) + 0.0846*np.s
in(4*np.pi*t) -
            0.0920*np.sin(5*np.pi*t) - 0.0328*np.sin(6*np.pi*t) + 0.0130*np.c
os(1*np.pi*t) -
            0.0373*np.cos(4*np.pi*t) - 0.0858*np.cos(5*np.pi*t))

mode_functions = [mode1, mode2, mode3, mode4, mode5, mode6]

# Load image data
def load_image_data(folder, prefix="frame_", ext=".jpg", start=0, end=49):
    image_paths = [os.path.join(folder, f"{prefix}{i:03d}{ext}") for i in ran
ge(start, end + 1)]
    image_shape = Image.open(image_paths[0]).convert('L').size[::-1]
    images = [np.array(Image.open(p).convert('L')).flatten() for p in image_p
aths]
    return np.array(images).T, image_shape

# POD
def perform_pod(data_matrix, with_mean_removal=True):
    if with_mean_removal:
        mean_field = np.mean(data_matrix, axis=1, keepdims=True)
        fluctuations = data_matrix - mean_field
        U, S, Vt = np.linalg.svd(fluctuations, full_matrices=False)
        modes = U
        coeffs = np.diag(S) @ Vt
    else:
        mean_field = np.zeros((data_matrix.shape[0], 1))
        U, S, Vt = np.linalg.svd(data_matrix, full_matrices=False)
        modes = U
        coeffs = np.diag(S) @ Vt
    return modes, coeffs, S, mean_field

# Main Execution
folder = "C:/Users/coryg/OneDrive/Desktop/STAT_698_Thesis_Joon_Suzuki/
Waterdrop_data"
# Adjust this to your local path
data_matrix, image_shape = load_image_data(folder, start=0, end=49)
modes, coeffs, S, mean_field = perform_pod(data_matrix)
```

```
# Extrapolate
r = 6
future_times = np.linspace(45, 49, 5)
future_coeffs = np.array([[mode_functions[i](t) for t in future_times] for i
in range(r)])
extrapolated_images = mean_field + modes[:, :r] @ future_coeffs[:r, :]

# Show extrapolated frame
plt.imshow(extrapolated_images[:, 0].reshape(image_shape), cmap='gray')
plt.title(f"Extrapolated Frame (t = {future_times[0]:.2f})")
plt.axis('off')
plt.show()

def compare_last_snapshots_with_last_extrapolations(data_matrix, extrapolated
_images, image_shape, future_times, start_index=45):
    fig, axes = plt.subplots(2, 5, figsize=(15, 6))  # Transposed: 2 rows, 5
columns
    fig.suptitle("Comparison: Last 5 Original Snapshots vs Last 5 Extrapolati
ons", fontsize=16)

    for i in range(5):
        # Original snapshot (top row)
        original_img = data_matrix[:, start_index + i].reshape(image_shape)
        axes[0, i].imshow(original_img, cmap='gray')
        axes[0, i].set_title(f"Original Frame {start_index + i:03d}")
        axes[0, i].axis('off')

        # Extrapolated image (bottom row)
        extrap_img = extrapolated_images[:, i].reshape(image_shape)
        axes[1, i].imshow(extrap_img, cmap='gray')
        axes[1, i].set_title(f"Extrapolated t = {start_index + i:03d}")
        axes[1, i].axis('off')

    plt.tight_layout(rect=[0, 0, 1, 0.94])
    plt.show()

compare_last_snapshots_with_last_extrapolations(
    data_matrix,
    extrapolated_images,
    image_shape,
    future_times
)

residual1 = np.abs(data_matrix[:, 45] - extrapolated_images[:, 0])
residual2 = np.abs(data_matrix[:, 46] - extrapolated_images[:, 1])
residual3 = np.abs(data_matrix[:, 47] - extrapolated_images[:, 2])
residual4 = np.abs(data_matrix[:, 48] - extrapolated_images[:, 3])
residual5 = np.abs(data_matrix[:, 49] - extrapolated_images[:, 4])
```

```
mse1 = np.mean(residual1**2)
mse2 = np.mean(residual2**2)
mse3 = np.mean(residual3**2)
mse4 = np.mean(residual4**2)
mse5 = np.mean(residual5**2)

mse1, mse2, mse3, mse4, mse5

from skimage.metrics import structural_similarity as ssim

ssim_score1, diff1 = ssim(data_matrix[:, 45].reshape(image_shape), extrapolat
ed_images[:, 0].reshape(image_shape), full=True, data_range=255)

ssim_score2, diff2 = ssim(data_matrix[:, 46].reshape(image_shape), extrapolat
ed_images[:, 1].reshape(image_shape), full=True, data_range=255)

ssim_score3, diff3 = ssim(data_matrix[:, 47].reshape(image_shape), extrapolat
ed_images[:, 2].reshape(image_shape), full=True, data_range=255)

ssim_score4, diff4 = ssim(data_matrix[:, 48].reshape(image_shape), extrapolat
ed_images[:, 3].reshape(image_shape), full=True, data_range=255)

ssim_score5, diff5 = ssim(data_matrix[:, 49].reshape(image_shape), extrapolat
ed_images[:, 4].reshape(image_shape), full=True, data_range=255)

ssim_score1, ssim_score2, ssim_score3, ssim_score4, ssim_score5
```

REFERENCES

Berkooz, Gal, Philip Holmes, and John L. Lumley. 1993. “The Proper Orthogonal Decomposition in the Analysis of Turbulent Flows.” *Annual Review of Fluid Mechanics*, 25: 539-575.

Boyce, William, and Richard DiPrima. 2012. *Elementary Differential Equations and Boundary Value Problems.* John Wiley & Sons.

Brunton, Steven L., and Nathan J. Kutz. 2017. *Data-Driven Science & Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press.

Burden, Richard L., J. Douglas Faires, and Annette M. Burden. 2016. *Numerical Analysis*. Cengage Learning.

Chatterjee, Anindya. 2000. “An Introduction to the Proper Orthogonal Decomposition.” *Current Science*, 78: 808-817.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms.* The MIT Press.

Kutner, Michael, Christopher Nachtsheim, and John Neter. 2004. *Applied Linear Regression Models.* McGraw-Hill.

Kværnø, Anne. “Partial Differential Equations and Finite Difference Methods.” TMA 4125, Norwegian University of Science and Technology.
Received 4 November 2020. Course handout.

Lee, Seungjoon, Mahdi Kooshkbaghi, Konstantinos Spiliotis, Constantinos I. Siettos, and Ioannis G. Kevrekidis. “Course-Scale PDEs from Fine-Scale Observations via Machine Learning.” *arXiv.* 1-13.

Luan, Lele, Yang Liu, and Hao Sun. “Uncovering Closed-form Governing Equations of Nonlinear Dynamics from Videos”. *arXiv.* 1-25.

Stark, Henry, and John W. Woods. 1986. *Probability, Random Processes, and Estimation Theory for Engineers.* Prentice-Hall Inc.

Strauss, Walter A. 2008. *Partial Differential Equations: An Introduction.* John Wiley & Sons.

Tao, Terence. 2006. *Analysis II*. Hindustan Book Agency.

Tibshirani, Robert. “Regression Shrinkage and Selection via the Lasso.” *Journal of the Royal Statistical Society.* 58: 267-288. https://www.jstor.org/stable/2346178

Tsutsumi, Natsuki, Kengo Nakai, and Yoshitaka Saiki. “Data-driven ODE Modeling of the High Frequency Complex Dynamics Via a Low-Frequency Dynamics Model.” *arXiv.*

Wackerly, Dennis D., William Mendenhall, and Richard L. Scheaffer. 2008. *Mathematical Statistics with Applications.* Thomson Higher Education.