APPROXIMATING SOLUTIONS OF GOVERNING EQUATIONS WITH FOURIER TRANSFORM AND SUPPORT VECTOR MACHINE

A THESIS

Submitted as partial fulfillment of the requirements for
sarjana komputer degree

By
Ahmad Izzuddin
NIM 1908919

PROGRAM STUDI ILMU KOMPUTER
FAKULTAS PENDIDIKAN MATEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS PENDIDIKAN INDONESIA
2025

ABSTRAK

Aproksimasi Solusi *Govering Equations* dengan *Fourier Transform* dan *Support Vector Machine*

Oleh
Ahmad Izzuddin
NIM:1908919

Model numerik sebuah sistem merupakan bagian penting dari ilmu pengetahuan dan *engineering*. Penggunaan *Machine Learning* (ML) dalam ruang ini untuk pembelajaran operator menyediakan alternatif sebagai *data-driven surrogates*. *Fourier Transform* menyediakan komponen kunci untuk mempelajari hubungan antara suatu fungsi dan turunannya. Berdasarkan *Spectral Neural Operators (SNO)*, kami mengusulkan kerangka kerja berbasis *Support Vector Machine (SVM)* untuk mempelajari persamaan dasar yang mengatur suatu sistem berdasarkan data. Kami mempelajari kelayakan dan interpretabilitas kerangka kerja yang diusulkan pada persamaan turunan dan persamaan Burgers. Model ini mampu belajar dari data acak yang secara matematis benar dan sebagian mampu melakukan generalisasi ke solusi eksak dari persamaan Burgers. Model yang dipelajari diinterpretasikan dan diverifikasi untuk mempelajari kontribusi yang benar dari koefisien fungsi masukan terhadap koefisien fungsi keluaran.

Kata kunci: *Operator Learning*, PDE, LSSVM.

ABSTRACT

Approximating Solutions of Governing Equations With Fourier Transform and Support Vector Machine

by
Ahmad Izzuddin
NIM:1908919

Numerical models of systems are a crucial part of science and engineering. The use of machine learning in this space for operator learning provides an alternative as data-driven surrogates. The Fourier Transform provides a key component for learning the relationship between a function and its derivatives. Building on Spectral Neural Operators (SNO), we propose a Support Vector Machine (SVM) based framework to learn the underlying governing equations of a system based on data. We study the viability and interpretability of the proposed framework on the derivative equation and the Burgers' equation. The model is able to learn from mathematically correct random data and is able to partially generalize to an exact solution of the Burgers' equation. The learned model is interpreted and verified to have learned the correct contributions of the input function coefficients to the output function coefficients.

Keywords: Operator Learning, PDE, LSSVM.

AHMAD IZZUDDIN
1908919

APPROXIMATING SOLUTIONS OF GOVERNING EQUATIONS WITH FOURIER TRANSFORM AND SUPPORT VECTOR MACHINE

Approved by,
Supervisor 1,

Prof. Dr. Lala Septem Riza, M.T.
NIP. 197809262008121001

Supervisor 2,

Dr. Muhammad Nursalman, M.T.
NIP. 197909292006041002

Acknowledged by,
Head of Program Studi Ilmu Komputer,

Dr. Muhammad Nursalman, M.T.
NIP. 197909292006041002

STATEMENT OF ORIGINALITY

Hereby, I declare that the thesis titled “Approximating Solutions of Governing Equations With Fourier Transform and Support Vector Machine” and all its contents are entirely my own work. I have not engaged in plagiarism or citation practices that contravene the ethical standards of academic conduct upheld within the scholarly community. By making this declaration, I accept full responsibility and am prepared to face any consequences or sanctions should any violation of academic ethics or claims regarding the originality of this work arise in the future.

Bandung, January 16, 2025

Ahmad Izzuddin
1908919

ACKNOWLEDGMENTS

I would like to take the time to acknowledge my friends, family, and everyone who have taken the time to support and encourage me through the completion process of my degree.

Bandung, January 16, 2025

Ahmad Izzuddin
1908919

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

Notation	Meaning
MSE	Mean Squared Error
RMSE	Root Mean Squared Error
sMAPE	Symmetric Mean Absolute Percentage Error
R^2	Coefficient of Determination
SoTA	State of The Art
MMS	Method of Manufactured Solution
PDE	Partial Differential Equation
FDM	Finite Difference Method
FEM	Finite Element Method
SVM	Support Vector Machine
SVR	Support Vector Regression
LSSVR	Least-Squares Support Vector Regression
ML	Machine Learning
PINN	Physics-Informed Neural Network

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

1.1 Background

Partial differential equations (PDE) are a common tool widely used in the modern scientific understanding and many engineering processes. This is because many systems can be described by the way they change and often this can be more intuitive. As an example, think of someone heating up a large frying pan on a gas stove. To describe how the pan heats up when the center is right above the burner, one can say that the center would heat up first followed by its surroundings. Eventually the pan would not heat up any further. Also notice that the edges of the pan would always be cooler than the center. However, once a more complex setup is introduced such as an uneven heat source or more complex materials with different heat rates of transferring heat it becomes much harder to describe how the pan heats up. A more useful way to describe the system is using the heat equation. For a temperature function $u(\mathbf{x}, t)$ of spatial coordinates vector $\mathbf{x}$ and time t, the generalized heat equation is defined in equation (1.1).

$$\frac{\partial u}{\partial t} = \boldsymbol{\nabla} \cdot (\alpha \boldsymbol{\nabla} u) \tag{1.1}$$

The divergence operator $\boldsymbol{\nabla}\cdot$ denotes sum of all first spatial derivatives. In three dimensions this is $\boldsymbol{\nabla}\cdot = \frac{\partial \cdot}{\partial x_1} + \frac{\partial \cdot}{\partial x_2} + \frac{\partial \cdot}{\partial x_3}$. And the gradient operator $\boldsymbol{\nabla}$ is the vector of all first spatial derivatives which in three dimensions is $\boldsymbol{\nabla} f = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \frac{\partial f}{\partial x_3} \end{bmatrix}^\intercal$. This equation says that the rate at which the temperature changes in time $\frac{\partial u}{\partial t}$ is proportional to how different the differences in temperature of a spot in the pan with its surroundings $\nabla^2 u$ multiplied with thermal diffusivity parameter α. One example of an insight this gives is that given a homogeneous material (α is constant) and the stove outputs heat at a constant rate the temperature will no longer change for a particular point once the temperature difference is constant. In other words once the function $u(\mathbf{x}, t)$ at some time $t > 0$ approaches a linear function in space, the temperature at all spots will no longer change in time. Mathematically this is because the time derivative is zero if the second spatial derivatives are also zero. Intuitively this is because once the temperature distribution approaches linear, the current spot on the pan is outputting as much heat it is getting. This insight is an example of why PDEs are useful.

Many other fields in physics such as waves, quantum dynamics, fluid dynamics, elastics, and many more also model systems using PDEs. PDEs are also used outside the physical sciences. In Finance, the Black-Scholes-Merton or Black-Scholes equation models the dynamics of the financial market. In ecology, PDEs are used to model population growth which is useful for modelling location dependent carrying capacity. This is used to model species distribution which can help develop better conservation policies. In the social sciences, PDEs such as the cross-diffusion model is used to explore the evolution of the urban environment (Jin et al., 2023). Crowd dynamics is another field where researchers have found it useful to model using PDEs (Hughes, 2000; Mukherjee et al., 2015). PDEs have a variety of applications, ranging from bird flocks, pedestrian crowds, to robotic swarms (Gong et al., 2023).

An increasingly critical use of PDEs is in Numerical weather prediction (NWP). The information weather forecasts provide is an integral part of modern life. Many sectors rely on timely and accurate weather forecasts. Individuals, day to day rely on forecasts for a range of different decisions ranging from activities they can do to personal safety from extreme weather. The weather also affects retail as consumer behavior change with whether it is rain or shine (Govind et al., 2020; Moon et al., 2018; J. Tian et al., 2018; X. Tian et al., 2021). Similarly, the types of transportation people use and operational decisions changes with the weather (Lepage & Morency, 2021; Nurmi et al., 2013). Another critical sector is agriculture, where extreme weather events and changes in climate threatens the food supply (Anwar et al., 2013; Cogato et al., 2019; Malhi et al., 2021). As the climate continues to warm and extreme weather events increase in frequency, the mitigation for these events become all the more important (*Climate Change 2022*, 2023). Because of this, forecasting is critical in making accurate decisions for policymakers and warnings of extreme weather events for civilians (Astitha & Nikolopoulos, 2023; Stott et al., 2016). A major challenge is the fact that observations of weather can be sparse in the sense that observations are localized and do not cover entire areas (Galkin et al., 2020; Monmonier, 1999; Wilby & Yu, 2013). This lack of spatially distributed meteorological information hampers decision-making on areas to be prioritized and what to prioritize based on location. For example, weather stations can only observe their immediate surroundings. And they are for the most part stationary. Even more mobile observation platforms like satellites only show the portion of the surface the satellite can see at any one time such as in figure 1.1. Notice that parts of the globe, especially near the equator have not been observed. While accumulating

observations through more satellite passes is possible, this, however, means short-lived features may not be observed. As impoverished regions often are the most data sparse, the effect compounds on the fact that for impoverished regions, scarce resources need to be effectively and efficiently applied. To this end, numerical models are employed to model a more representative view by using known physics such as the compressible Euler equations and statistical models (Kwasniok, 2012; Mengaldo et al., 2019). The next step is in forecasting future weather which in itself is a challenge. The information this could provide is invaluable because it means planning for future weather is possible.

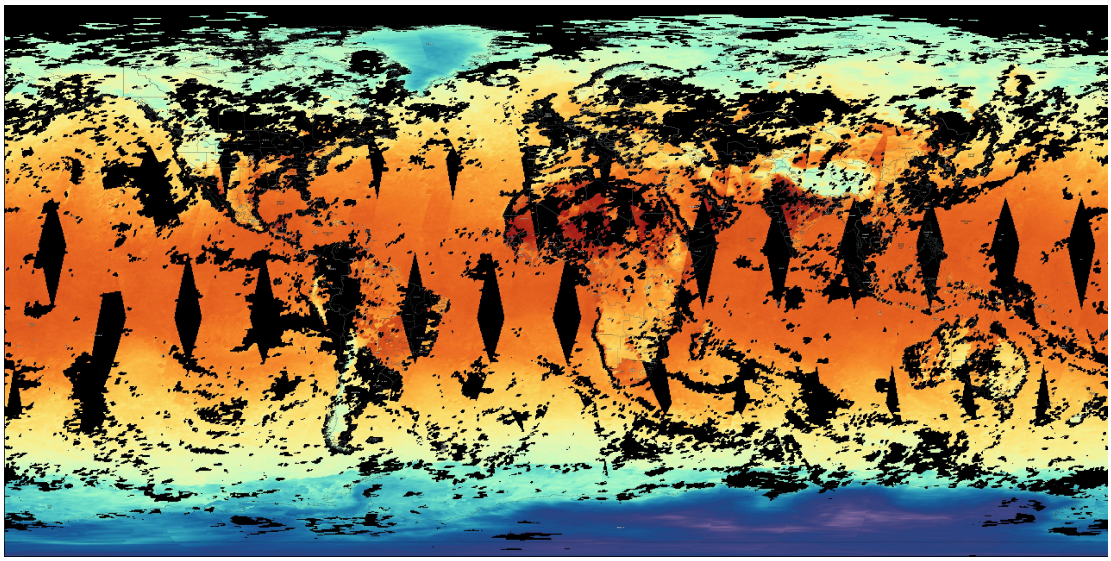


Figure 1.1: Daytime and nighttime surface air temperature from observations by the AIRS instrument onboard the NASA AQUA satellite. This shows observations recorded through a whole day on the 21st of May 2024. This image was produced using NASA Worldview (https://go.nasa.gov/46SaYyJ).

Forecasting the weather or evolution of any other system is formulated as an initial value problem where past observations are used to predict an unknown future state of the system. A different formulation can be how heat spreads over time in different types of frying pans from known information of the pan's materials, their heat conduction properties, and that the stove gives off constant heat. In general the scenario is predicting the effect (future atmospheric temperature distribution, etc.) of some given causes (current and past weather, etc.). This scenario is termed the forward problem. The inverse problem on the other hand solves for an unknown parameter using other known parameters and observations of the partial solution such as temperature field at the

object's surface. Traditional methods used to solve these problems utilized knowledge of the exact equations to numerically solve PDEs. As an example, the Finite Difference Method (FDM) approximates the solution by substituting the partial derivatives with finite differences and manipulating the equation such that the solution can be computed. The inverse problem on the other hand is determination of initial conditions or other parameters from observed state of the system. For example determining the heat source distribution given the temperature distribution at some point in time. One traditional approach to this would have meant solving the forward problem from an initial guess of the parameter and then evaluating some cost function such as how close the predicted solution is to the observed one. As expected, this would require many evaluations of the forward solver. This means that the cost of the solver would be greatly multiplied, emphasizing the importance of an efficient solver.

Traditional numerical solvers have enjoyed many decades of development due to their long history. There are many general approaches to solve PDEs and many more very specific approaches. Other than FDM, some widely used approaches include Finite Element Methods (FEM), Finite Volume Methods (FVM), Collocation Methods, and Spectral Methods. FEM and FDM are both mesh based approaches, meaning they rely on discretization of the computational domain. There are several challenges associated with this. First, irregular domains such as bio-inspired materials like bone, spider silk, or aggregate materials like gravel pose a challenge due to the complexity of the domain geometry (Buoni & Petzold, 2007; Gaul et al., 1991; Jia et al., 2024). Also, irregular domains which create large deformations or mesh entanglement cause these methods to become ineffective (J.-S. Chen et al., 2017). While there are strategies to mitigate this, by definition they are an additional layer of difficulty to the process. Second, multiscale applications where the micro and macro scales are both important require fine meshes such that the small scale structures are adequately simulated. This creates meshes with very large number of points that are very resource intensive (Buoni & Petzold, 2007). Third, a single evaluation of traditional mesh-based solvers may not be very costly, however multiple evaluations can add up. This is very apparent in inverse problems where the solver is queried multiple times to solve the forward problem in order to obtain parameter functions. Therefore, in problems where these issues important or resources are limited, mesh-free methods may be preferred. As an example, the spectral method solves PDEs by formulating the solution as a linear combination of global basis functions like the Fourier series. This can be likened to how music combines different

sound waves to produce the overall sound. For some function s, the approximation s^* gets closer to the original with more linear combinations of basis functions. The complex formulation of the Fourier series is presented in equation (1.2). With this approach, the approximated function is characterized completely by its coefficients c_k. This fact is exploited by the spectral method to solve differential equations. The solution function can be approximated by finding the mapping between input function coefficients and solution function coefficients. The mapping between a function s and its derivative s' can be found using the derivative equation from their coefficients leading to $\hat{s}_k = \hat{s}'_k / (2\pi i k)$. Practically this would be done with a finite number of coefficients ($k < \infty$) and the coefficients $\hat{s}'_k$ are obtained using a Discrete Fourier Transform (DFT) algorithm like the Fast Fourier Transform (FFT).

$$s \approx s^* = \sum_k \hat{s}_k e^{2\pi i k x} \tag{1.2}$$

This leads us to the fourth challenge with traditional methods which is that they require prior knowledge of analytic forms of PDEs. This is because traditional solvers use the equations to formulate solutions. This makes traditional numerical approaches unsuited to problems where the governing equations are unknown or partially known. These challenges altogether are some of what weather forecasting faces; NWP has the immense task of modeling multiple scales of the Earth's atmosphere while accounting for features of physical systems that are still not fully understood. Many other fields such as epidemiology (Brauer et al., 2019) and ecology (Holmes et al., 1994; Turchin, 2001) also face these challenges. As with many other fields, at some point in time the governing equations of many systems were not known. This has motivated research into alternative methods that do not completely rely on prior knowledge, are mesh free, and fast enough when solving forward problems.

Machine Learning for PDEs

With the increasing prevalence of machine learning methods and their use in more and more fields, research into their use for scientific computing has taken off in recent years. While statistical modeling has already been widely used in areas such as physical constants, stellar population studies, risk assessments of events such as earthquakes and coronal mass ejections (Anselmo & Pardini, 2005; Berliner, 2003; Bernardi et al., 2022; Reinhardt et al., 2016; Spanos, 2006; Uzan, 2003), the dominant approach for forward

modeling or inverse modeling has remained physics based numerical models. Machine learning has provided an alternative approach to model the solutions of PDEs. In their work, Aarts and Van Der Veer (2001) utilized neural networks to approximate each term of PDEs describing damped and undamped free vibrations and substituting them into the PDEs and associated initial conditions. The network parameters were then optimized to reduce the PDE residual and boundary condition loss using evolutionary algorithms. This approach was taken in order to make machine learning models more transparent which at the time was being pursued because of the high cost of optimizing uncertainties in water management numerical simulators. However, since this method approximates the mapping between coordinates and the values of a function and their derivatives, retraining would be necessary for changes to the function itself. This could become very costly as retraining costs accumulate. A more recent approach that also utilizes soft constrains from PDE residuals is termed physics informed neural network (PINN) (Raissi et al., 2019). The authors propose a framework that leverages advancements in computing, namely automatic differentiation (AD) techniques made readily available by modern machine learning libraries. In general, PINNs use AD to compute each term of the PDE from the output of the model and this is then substituted into the PDE to in order to compute the residuals and loss from boundary conditions. The network residual and boundary loss are then weighted and summed with the data loss. For a neural network $\hat{u}(\mathbf{x}, t)$ approximating the real solution $u(\mathbf{x}, t)$, the residual loss for the heat equation in equation (1.1) is equation (1.3). There are several advantages of incorporating physics knowledge into the model including regularization of the model outputs to be more consistent with physics, faster convergence, and less to no data required depending on whether the network is trained in a manner that is supervised, self-supervised, or a combination of both. One issue with using AD to compute the residual is that the network input needs to be the independent variable (i.e. coordinates, time, etc.). This once again means that if one wants to compute a different solution, the network needs to be retrained.

$$\mathcal{L}_{PDE} = \left(\frac{\partial \hat{u}(\mathbf{x}, t)}{\partial t} - \boldsymbol{\nabla} \cdot (\alpha \boldsymbol{\nabla} \hat{u}(\mathbf{x}, t)) \right)^2 \tag{1.3}$$

Learning PDEs with CNNs

Other works utilize convolutional neural networks (CNN) to compute the solution from input functions such as forcing terms or initial conditions. This approach generally means discretizing the functions on a grid and using these as training data. One study by R. Wang et al. (2020) predicts turbulent flow using spatial and temporal decomposition and a specialized U-Net, an architecture based on CNNs, to predict the velocity field from the decomposition of the previous velocity field. Part of the loss function is a regularization term for zero divergence in the velocity field to enforce incompressible fluid flow. This term was calculated using finite differences since auto differentiation is not applicable in this situation. Finite differences was also utilized in another CNN based fluid flow upscaling model by Gao et al. (2021b) to compute the residual terms of the steady incompressible Navier-Stokes equation. This model also inferred unknown physical parameters such as boundary conditions. However, this approach would mean the model would need to scale as a quadratic in 2D, cubic in 3D, and much steeper in higher dimensions. Outside fluid dynamics, the combination of specialized CNNs and finite differences or another numerical differentiation method have been used for many other PDEs including Poisson's equation for temperature fields (Gao et al., 2021a; Zhao et al., 2023), velocity models from seismic data (Muller et al., 2023), and seismic response of structures (Ni et al., 2022; Zhang et al., 2020). While the use of CNNs mean that discretization is implied, solutions of different initial conditions or parameter functions can be computed by inference and no retraining is required. This property is especially useful for many-query problems such as computing gradients for inverse problems.

Operator Learning

The mapping between discretized functions done by CNNs are related to an alternative approach that starts by viewing PDEs as operators, which are generalized mappings between spaces. One group of familiar operators are functions which maps between spaces of scalar values or vector values. PDEs on the other hand are operators that map between function spaces. A simple example is the derivative. The derivative takes in a function and returns the derivative of said function. In other words it is an operator that maps between the space of all functions to the space of derivatives of those functions. Another way to view operators starts by viewing functions as

infinite dimensional vectors. Where elements in the vector are the function's value evaluated at every point in space. The operator can be seen as a vector function mapping between these infinite dimensional vector spaces. Operators are important because a field of research has sprung up around this mathematical concept. Operator learning is the use of machine learning to learn operators using data driven approaches. As an analogy, function regression traditionally has been used to approximate the mapping between input values such as coordinates and output values of functions evaluated at said coordinates. In the case of operator learning, the mapping between function spaces are approximated. The aforementioned approaches using CNNs does this directly using the values of functions at discrete points. There are other approaches like DeepONet that does not require the uniform grid like CNNs (Lu et al., 2021). This architecture instead uses both input functions and coordinates as inputs. The output is the output function evaluated at the coordinates provided. This architecture is based on an extension for deep learning of the universal operator approximation theory for neural networks first proposed almost three decades ago at the time of writing by T. Chen and Chen (1995). The proposed architecture is composed of two subnetworks, where one termed the trunk $\hat{\mathbf{T}}(\mathbf{x}, t)$learns the latent mapping for coordinates and the other network termed the branch $\hat{\mathbf{B}}(\mathbf{f})$ learns the latent mapping for the input function f. The two latent mappings are combined through a dot product to obtain the approximated output function value $u(\mathbf{x}, t)$. This is formulated in equation (1.4).

$$u(\mathbf{x}, t) \approx \hat{G}(\mathbf{f})(\mathbf{x}, t) = \hat{\mathbf{B}}(\mathbf{f}) \cdot \hat{\mathbf{T}}(\mathbf{x}, t) \tag{1.4}$$

Fourier Neural Operators (FNO) is an alternative avenue for learning operators by utilizing the fact that functions can be decomposed into linear combinations of basis functions, namely trigonometric basis in this particular case (Li et al., 2021). With this method the input function value is mapped to its corresponding output function value. This is done by first lifting the input function value to a higher dimension using a neural network and this is then passed through blocks composed of a Fourier transform, then a linear transform and filtering of higher modes, and finally the inverse Fourier transform. These blocks are stacked to a desired depth and finally another network projects the outputs to the target dimension. The reason a linear can be used is that differentiation is multiplication in the Fourier domain. One drawback with FNO is the requirement that output functions are not parameterized by coordinates and therefore is implicitly relative to the input function coordinates. To avoid this issue, Fanaskov and Oseledets (2023)

reframes the problem by directly utilizing the coefficients of Fourier or Chebyshev basis. The model, termed Spectral Neural Operator (SNO) is trained on features of input function coefficients which are computed using Fourier or Chebyshev transforms and labels of output function coefficients using the same transforms. The authors point out one motivation for this approach which is that training neural networks on discretized data may not be ideal because unexpected outputs such as non-smooth interpolation may happen when the network is trained on one grid size and evaluated other grid sizes. With SNO, the interpolation of the function is smooth due to interpolation being done by Fourier basis functions which are sines and cosines for example. In a similar study, Du et al. (2024) extends the concept of mapping coefficients by proposing residuals in the spectral domain and leveraging Parseval's Identity to compute the spectral analog to the loss term in PINNs. This allows for self supervised learning in the spectral domain. The same benefits incorporating physics into PINNs also apply here without the pain points introduced by discretized model inputs and outputs. As a whole, operator learning creates an alternative approach that addresses the issue of retraining or recomputing the solution model. In addition, due to its data based approach, even systems with partially or fully unknown governing equations may be simulated.

A persistent challenge with all these approaches is the issue of optimization. While neural networks are modular and expressive which is proven by the universal approximation theorem (Cybenko, 1989; Hornik et al., 1989), their loss function present many local minima meaning it is non-convex. This can be mitigated by using advanced optimization techniques that can find a local minima close enough to the global minima such as Adam (Shrestha & Mahmood, 2019; Soydaner, 2020). However, the addition of PDE residuals into the loss function have worsened the highly non-convex loss landscape issue (Basir & Senocak, 2022; Krishnapriyan et al., 2021; Rathore et al., 2024). These problems range from the disparity in size of boundary and residual loss gradients to the fact that incorporation of residuals and boundary conditions themselves create a much more complex loss landscape. As a result, it is desirable to utilize a different machine learning algorithm that possesses a convex loss landscape. One family of such algorithms are Support Vector Machines (SVM) (Vapnik, 2000). The appeal of SVMs are the fact that the model is formulated as a quadratic programming problem. This means there are strong guarantees for convergence, generalization, and complexity. Another formulation called Least-Squares Support Vector Machines (LSSVM) reformulates the problem as a linear system (Suykens, 2005). This leads to an

easier problem that can be computed faster by well established algorithms like the many implementations of least squares solvers. Another advantage of the linear formulation is that this can be easily parallelized to exploit hardware like graphics processing units more widely known as GPUs in contrast to the commonly used Sequential Minimal Optimization (SMO) used for SVMs with quadratic objective functions.

The advantageous properties of SVM based methods have attracted research into their use for solving PDEs. An early work using SVMs to solve PDEs by Youxi Wu et al. (2005) introduced a method for solving the forward problem of Electro-Impedance Tomography. This work solved for the mathematical model of EIT which is given by Maxwell's equations by modeling the trial function as using an ϵ-SVR model. Another approach much more similar to PINNs was presented by Mehrkanoon and Suykens (2015). The residual and initial/boundary conditions are imposed as equality constraints on an LS-SVM objective function. A different study by Leake et al. (2019), the incorporation of physics into the model is done slightly differently by utilizing the theory of functional connections to directly embed constraints into the solution. This means that the proposed method would satisfy the boundary condition exactly. However, the authors point out that for PDEs in higher dimensions deriving and implementing this method can become cumbersome. These approaches, however, do not learn the PDE operator itself. Meaning they are also not practical for many-query problems.

Operator Learning for Weather Forecasting

In terms of weather forecasting, operator learning has been applied in terms of initial value problems. This problem formulation is reminiscent of time series prediction problems widely found in machine learning research. Researchers Kurth et al. (2023) developed FourCastNet which utilized Adaptive Fourier Neural Operator (AFNO), a transformer based model containing the previously mentioned FNO computational blocks by Li et al. (2021). This model was then able to be trained in a massively parallel manner. In a comparison with a traditional model called the Integrated Forecasting System from the European Center for Medium Range Weather Forecasts (ECMWF), FourCastNet is faster and much more efficient in terms of inference time, resulting in about $80\,000$ times speed up for a 100-member ensemble forecast. This is while performing much better than a previous deep learning approach. In another study, Bonev et al. (2023) proposed a variation on neural operators called Spherical Fourier

Neural Operator (SFNO) which exploited the spherical nature of global forecasting by using Spherical Harmonic Transform (SHT) in place of Fourier Transform. This model when compared to AFNO and FNO, produced no visible artifacts in autoregressive rollouts for long range forecasting. In terms of forecasting, the model shows outcomes that matches the IFS which is a big leap forward in parity for traditional and machine learning based methods.

1.2 Problem Statement

The problems this work sets out to solve based on section 1.1 are:

1. What is the formulation a computational model for operator regression and therefore solving PDEs in the spectral domain using support vector machines?
2. Can the model learn the relationships represented by PDEs? And how does it perform on different problems?
3. How can one interpret the learned model?

1.3 Aims

Based on the stated problems in section 1.2, this study aims to accomplish the following:

1. The design and implementation of a computational model that maps coefficients in the spectral domain using Least-Squares Support Vector Regression (LSSVR).
2. Proof of the model's learning ability using three different problems.
3. Interpretation the model results and why some predictions turn out the way they do.

1.4 Contribution

In achieving the aims of this study the following contributions are made:

1. A novel use LSSVR which has a convex objective to learn solution operators of partial differential equations and operators in general.
2. Interpretation of machine learning model trained on operator data.

1.5 Limitations

This work is limited to the following:

1. Functions the model works with are only continuous functions.
2. The basis functions are limited to Fourier basis.
3. Modeled fields are compact or dense meaning not sparse. If this is not true, such as sparse observations, the data would need to be assimilated to create a dense field.

1.6 Outline

This thesis will be structured into five chapters with this chapter being the first. For further detail, chapter II will discuss and summarize the literature related to this study. The context of the study will be explained in terms of the state of the art (SoTA), the literature involved, and how this problem fits within the larger computing and physical sciences field. In chapter III, we discuss the methodology that is used in this study to achieve the aims that were set in the first chapter. And then, the results of applying the research methodology are presented in chapter IV. In this section, the results are also analyzed and discussed. Finally, in chapter V we draw conclusions from the results and future directions of research for the use of SVMs in operator learning.

CHAPTER II

LITERATURE REVIEW

2.1 Governing Equations

Mathematical expressions that describe the behavior of a system are called governing equations. Typically, governing equations are in the form of Differential Equations (DE). For some systems with several variables, the governing equations are Partial Differential Equations (PDE). These equations are the foundation of many engineering and scientific models (Anderson, 1992; Olver, 2014). In the physical sciences, the use of PDEs as governing equations is widespread. Many fields, including those outside the physical sciences, use these equations to model systems and solve problems whose solutions we take for granted today, ranging from fluid mechanics to ecology.

In recorded history, the early work of differential equations can be traced to the independent works of Gottfried Wilhelm Leibniz and Isaac Newton. The formulation of calculus and its associated notation paved the groundwork for further study into the mathematical description of change. PDEs as a subject of study was started in the 18th century with the works of Euler, d'Alembert, Lagrange, and Laplace (Brezis & Browder, 1998). Near the end of the 19th century, Poincaré stated that a variety of physical system share a similarity. This is followed by the statement that they should be treated in a similar fashion. He expressed his motivation for mathematical proofs that approximate physical phenomena. Meanwhile, through the 18th and 19th century, many equations such as the Euler equation of incompressible flow, Maxwell's equation in electromagnetic theory, and the Navier-Stokes equations for fluid flow. These are just some examples of the equations that were developed during that period of time. In the 20th century, what is known today as the Burgers' equation was introduced with the interest of modeling turbulent flow. Today, there are many more equations that have been introduced to solve many more different problems. The Poisson equation, for example, is used to model many problems including electric potential, steady-state temperature distribution, and Newtonian gravity (Selvadurai, 2000).

Governing equations in the form of PDEs describe the behavior of systems by quantifying changes. These changes, naturally, are represented as partial derivatives. We will discuss three situations where systems can be described in this way. First, in its

simplest form, a derivative equation models the relationship between the rate of change and the total change. In more relatable terms, the example for this is the relationship between speed and acceleration. Acceleration is the rate of change for speed. The function that describes speed will complement the acceleration depending on the kind of function that describes acceleration and vice versa. The mathematical formulation for the derivative equation is shown in equation (2.1). Because of the ubiquity of the relationship defined by this equation, it is perhaps the one with the most uses. Some applications for differential equations include radioactive decay (Groch, 1998; Huestis, 2002), population dynamics (Gopalsamy, 1992), and other dynamical systems (Katok et al., 2009).

$$\frac{\mathrm{d}u\left(t\right)}{\mathrm{d}t} = a\left(t\right) \tag{2.1}$$

Solving the derivative equation can be analytically if the exact form is known. This involves common techniques in calculus such as separation of variables (Braun, 1993). In other cases, the solution can be computed using numerical methods such as Euler's method and the Runge-Kutta method among others. These numerical methods use the knowledge about the differential equations to compute the solutions. Both the Euler's method and Runge-Kutta method compute the solution by adding the current value of the solution with the estimated difference with the next value of the solution to produce the next value of the solution. The estimated difference is typically computed by with some combination the derivative and step size. This is where the knowledge of the derivative equation to be solved is used. By storing each predicted solution value and using it as the current solution value, the entire solution is computed.

The second situation we discuss is the Burgers' equation. This equation was initially used for describing the distribution of speed in a fluid. Although not entirely physically accurate, this equation describes how differences in speed observed within a fluid can emerge. These differences in speed can lead to shocks where the gradient approach infinity (Orlandi, 2000). The forced-viscous equation can be seen in equation (2.2). The equation is made up of four terms including the forcing term. The first is the time derivative. The second term is the convection term which makes this equation nonlinear because the solution is multiplied by another form of the solution, which in this case is the first derivative in space. The third term is the advection term. Finally, we have the

forcing term that encompasses all external influence on the solution.

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2} = f \tag{2.2}$$

An intuitive way to understand how the solution would evolve in time is to think of the time derivative in terms of all the other components. This way we can see how the solution would change from one time to the next at all points. The viscosity ν modulates the advection term. The larger the viscosity value, the more the solution resists forming shocks. This is because, the second derivative makes the time derivative have higher values for areas of rapid change. Specifically, it makes areas that change rapidly in space to lower solution values have lower time derivative values. For areas with rapid change in space to higher solution values, it makes those areas have higher time derivative values. The result is that it spreads out the rapid changes in space or "curves" over time.

The convolution term behaves in an amplifying manner. For areas that have the same signs for their solution value and the space derivative, it would attract it to lower values. For areas with differing signs for the solution value and space derivative, it would attract it to a higher value. As a whole this would make the solution tend toward large negative gradients and discourage large positive gradients. As the negative gradient grows, if unimpeded by the advection term, it would eventually lead to the formation of a shock wave. Finally, the forcing term affects the time derivative directly and attract the solution toward which ever direction defined by the forcing term. Solutions to the Burgers' equation can be obtained analytically. One of the most straightforward ways is the Cole-Hopf transform (Wazwaz, 2010). Using this method, the solution to the Burgers' equation u can be computed from any valid solutions of the heat equation ϕ. The solution of the Burgers' equation is obtained by using the Cole-Hopf transform written in equation (2.3). For example, one solution from this transform is shown in equation (2.4) was specifically mentioned by Wood (2006) to be complex enough to test how well numerical methods solving the Burgers' equation can handle the nonlinearity. Similar solutions and other solutions using also exists in other literature (Benton & Platzman, 1972; Wazwaz, 2010).

$$u = -2\nu\frac{\phi_x}{\phi} \tag{2.3}$$

$$u(x,t) = \frac{2\nu\pi e^{-\pi^2\nu t}\sin(\pi x)}{a + e^{-\pi^2\nu t}\cos(\pi x)} \tag{2.4}$$

The use of the Cole-Hopf transform, however, limits the range of possible solutions because of the requirement of a valid heat equation solution. For a wider range of solutions, numerical methods are predominantly used to solve the initial value problem for the Burgers' equation. This can be done by computing the time derivative term from the other terms. However, both the convection and advection terms create a challenge in solving the equation numerically. For small viscosity values, the equation is dominated by the convection term. And as previously mentioned this means the development of sharp gradients and possibly shock waves. As the gradient become sharper, the solution would require finer and finer time steps in order to stay at a consistent error for each step, otherwise the solution can become unstable. High viscosity values introduce a similar issue. The problem comes from rapidly varying large values introduced by the second derivative. This is a product of the "amplification" caused by derivatives which is compounded by higher orders. The rapid variation also mean that smaller time steps would need to be used by standard numerical solvers. For numerical solvers, such as the forward Euler method we discussed earlier, insufficiently fine time steps can lead to numerical instability. This imposes a constraint on how large the time steps and or on how the numerical solvers can treat these equations. The differential equations that cause this issue are called stiff PDEs. To overcome these challenges, specialized numerical solvers for stiff PDEs have been developed (Kassam & Trefethen, 2005).

The third situation we discuss here is the weather system. Today, we have much better understanding of the governing equations of the weather system. The governing equations of the weather include the momentum equations (for example wind), gravitational force, friction, centrifugal force, Coriolis force, the hydrostatic equation, and thermodynamic equations (Martin, 2014). These components each describe how different variables change and the interaction between those variables, such as wind and temperature.

The interplay between the governing equations of weather are very complex. In addition, some constituting equations like the Navier-Stokes, can be very nonlinear. These are some reasons why analytical solutions for the governing equation are

not feasible for realistic scenarios. Today, the weather is modeled by numerical methods. Initially, many numerical models that were in use were based on the Finite Differences Method (FDM). After 1972, other approaches regained interest, namely the spectral methods (Mengaldo et al., 2019; Pu & Kalnay, 2018). Much more recently, other advances include ensemble forecasting where multiple models represent the probabilistic distribution that needs to be accounted for. Another development is advanced data assimilation methods for various observational data sources (Park & Xu, 2013). Due to the size differences involved with simulating the weather, some parts such as turbulence or the presence of aerosols adds further challenges. Many operational models, use approximations such as turbulence closure because the unknowns outnumber the equations. Due to these challenges, there has been research into the use of machine learning and other statistical methods for this specific area (Bonev et al., 2023; Lang et al., 2024). Machine learning-based approaches provide the ability to predict the weather with increasing accuracy and speed using a black box model which complements the physics-based models (Bi et al., 2023). Ongoing research and technological developments continue to refine these models, aiming for more precise and reliable weather forecasts.

2.2 Machine Learning

Machine Learning (ML) has experienced a surge of interest in the past two decades. A major contributor to this is the increase in computing power. Specifically Neural Networks (NN), found renewed interest with parallel algorithms being able to run on hardware accelerators like Graphics Processing Units (GPU). Machine Learning can be categorized into two major paradigms which are supervised learning and unsupervised learning (Alpaydın, 2020). In unsupervised learning, the model is usually asked to find the structures within the data. For example, the different groups that share a similar structure or how often such structures appear. On the other hand, supervised learning involves training models with pairs of features and labels. The labeled dataset gives a certain direction the for the model to learn. For example, labels that indicate whether a cat is present or not in an image. Other labels might instead indicate the number of cats in the image. These labels essentially asks the model to look for certain structures in the features that are shared between common labels.

Supervised learning is further categorized into the type of prediction task. The

first task is classification. In this task, models are trained to group samples into different categories. Samples within the same category would share a common structure within its features. For example, a regulatory body of the government wants to detect fraudulent activities. A model could be trained to classify whether a bank account is involved in fraudulent activity or not. This is done by training on a collection of both fraudulent and normal transaction data. By processing the data into more meaningful features such as average transactions or the rate of transactions and incorporating other information about the account a pair of features and labels are created. This can then be used to train the model to distinguish between fraudulent and legitimate activities.

The second task we will discuss is regression, which we will focus on for the rest of this section. In regression the model is tasked to predict continuous values based on the features it receives (Alpaydın, 2020; Matloff, 2017). This is in contrast to the discrete values used as labels in the classification task. Applying machine learning to the problem of modeling systems, like traditional numerical methods, is a regression task. For example, if we want a model to learn the relationship between the coefficients of two different functions in place of the spectral method, this would be a case of regression. As more concrete example, the prediction of weather variables is a regression task. Typically, the features are based on the current and previous states of the atmosphere. The model is then trained to predict the future atmospheric state. The variables that represent atmospheric state such as temperature and wind speed are continuous variables. Specifically, if the task is to predict from features outside the range of the training data, such as predicting future weather, this is called extrapolation. On the other hand, if the features exists within the range of the training data, such as approximating the wind speed between weather station, the action is called interpolation. Regardless, for both, the model learns to approximate the continuous labels from the features.

On the subject of weather prediction, the goal is sometimes to predict several time steps ahead of the reference input features. The term lead time is used in this scenario to refer to the amount of time ahead of the reference time. Weather predictions for several time steps ahead is particularly useful in order to allow for longer term planning. For example, management of the power grid and renewable power sources rely on weather forecasts to manage supply. For these cases, the model is usually trained to predict the state at the next time step based on the current time step. To achieve the predictions for several time steps ahead, the model would need to predict using

its previous predictions. This is repeated until the target lead time is reached. This use of the model is called autoregression (Maccarrone et al., 2021). An important property of the model for this task is stability. This is because errors will compound very quickly if the model is unstable. For ML models that has flexible loss functions like NNs, the model can be made more accurate by including the autogression during training and incorporating the autoregression error into the loss function (Gregor et al., 2014). Together, autoregression and regression, form an essential tool for solving many problems involving the prediction of continuous values.

2.3 Operator Learning

Machine learning, as with many other fields, has found its foothold in scientific computing. The term that has become popular for this field is Scientific Machine Learning (SciML) (Baker et al., 2019). This field, just like scientific computing, is interdisciplinary with mathematics, physics, computer science, and other disciplines contributing to solve the specific challenges involved with datasets specific to these individual disciplines. The field seeks to extract insights from scientific datasets by building new methods that are scalable, interpretable, and reliable. This is partly driven by the advances in computing, specifically machine learning, and the potential for and accumulation of scientific data. The motivations for developing domain-specific machine learning tools include the difficulty in using and interpreting general purpose ML algorithms for the problems researchers seek to solve with scientific datasets.

A set of tools called Operator Learning has been attracting interest within SciML. These tools learn the mappings between functions defined by an operator (Boullé & Townsend, 2024; Kovachki et al., 2023). Mathematically, an operator is a mapping between function spaces. For example, the derivative operator $\frac{\mathrm{d}}{\mathrm{d}x}$ maps the space of all differentiable functions with the space of derivative functions. For this example, the model would learn the relationship between the derivative function and the antiderivative function. More generally, a model $\hat{\mathcal{A}}$ is used to approximate an unknown operator $\mathcal{A}$. Given data that represent two functions (f, u) with $f \in \mathcal{F}$ and $u \in \mathcal{U}$ on a domain Ω which is a subset of the n-dimensional real number line, and an operator $\mathcal{A} : \mathcal{F} \mapsto \mathcal{U}$ such that $\mathcal{A}(f) = u$, the objective is to find the approximation $\hat{\mathcal{A}}$. The search for the approximation is due such that for any new data $f' \in \mathcal{F}$ and $u' \in \mathcal{U}$, we get $\hat{\mathcal{A}}(f') \approx \mathcal{A}(f')$.

The model $\hat{\mathcal{A}}$ is optimized by minimizing the loss function shown in equation (2.5), with model parameters θ.

$$\min_{\theta} L\left(\hat{\mathcal{A}}\left(f;\theta\right),u\right) \tag{2.5}$$

There are currently different model architectures for the learning operators. Most of these are implemented as Neural Networks (NN). The Neural Operator proposes a method to generalize NNs to learning mappings between function spaces (Li et al., 2021). The Fourier Neural Operator (FNO) variant adds Fourier transforms to the composition of layers within the NN. Another NN layer is added to learn the mapping within the Fourier domain. The outputs of this layer is then Inverse transformed and passed on. This entire process is called the FNO block. Adding multiple of these blocks allow for learning more complex relationships. Another approach called Spectral Neural Operators, instead maps the coefficients of inputs to the coefficients of output functions (Fanaskov & Oseledets, 2023). For coefficients obtained from Fourier transforms, the model is approximating the standard spectral method by taking advantage of the properties of the Fourier coefficients we discuss in section 2.4. Essentially, derivatives are multiplication in the Fourier domain. A similar work extended the idea by leveraging the known PDE to be part of the loss function (Du et al., 2024). The model is penalized for the residual computed using the PDE. This allows the NN to be trained in a self-supervised setting. Another major model is the so called DeepONet architecture proposed by Lu et al. (2021). This model can be seen as foregoing the basis function the Fourier Transform enforces and instead learns the basis functions from the data (Meuris et al., 2023). Because of this, the model has two components. The first, named the trunk network acts as the learned basis functions. This network is given evaluation coordinates and computes the values of the "basis functions". The second network, called the branch network, learns the mapping between the input function representation and the coefficients for the output function. The dot product of the outputs of both trunk and branch networks are evaluated to obtain the value of the predicted function at the specified coordinate. The final proposed approach we will discuss here doesn't use Deep Learning. The approach proposed by Nelsen and Stuart (2024) uses Random Feature Models (RFM) to learn operators. This model enjoys the advantage of using one of the simplest forms of machine learning.

2.4 Fourier Transform and Series

The Fourier transform and its associated series is a well known topic in both computing and the physical sciences (Smith, 2007). The Fourier transform is used extensively both in theoretical and practical scenarios. In mathematics and physics, the Fourier transform is used widely for many subjects such as harmonic analysis, quantum mechanics, and of course in solving PDEs. More practical uses of the transform include signal processing, communication systems, data compression such as JPEG and MP3, and cryptography. For solving PDEs, the Fourier transform is used to simplify the equations involved. This is due to the unique properties associated with the Fourier transform.

The Fourier series can be represented in two ways, the first is the real representation by using sine and cosines as the basis functions in the series shown in equation (2.6), where k is the wave number, L is the period, and $\hat{a}_k$ and $\hat{b}_k$ are the coefficients for the k-th wave number. The second representation is the complex representation which uses Euler's formula as the basis functions shown in equation (2.7) with complex valued coefficients $\hat{c}_k$. Both of these are equivalent and are just the different representations. The relationship between the two representations is defined in equation (2.8).

$$f(x) = \hat{a}_0 + \sum_{k=0}^{\infty} \hat{a}_k \cos(2\pi kx/L) + \hat{b}_k \sin(2\pi kx/L) \tag{2.6}$$

$$f(x) = \sum_{k=-\infty}^{\infty} \hat{c}_k e^{2\pi ikx/L} \tag{2.7}$$

$$\hat{c}_k = \begin{cases} \hat{a}_0, & k = 0 \\ (\hat{a}_k - \hat{b}_k)/2, & k > 0 \\ (\hat{a}_k + \hat{b}_k)/2, & k < 0 \end{cases} \tag{2.8}$$

To understand the complex coefficients better, we will do a brief summary of complex numbers. Complex numbers are numbers consisting of real and imaginary components. Complex numbers are written as in equation (2.9) with a and b being the real and imaginary components respectively. The imaginary component is multiplied

by the imaginary unit $i = \sqrt{-1}$.

$$c = a + bi \tag{2.9}$$

$$\tag{2.10}$$

The Discrete Fourier Transform (DFT) is shown in equation (2.11). The DFT is used for discrete function values, which is the case for most practical applications (MathWorks, 2024; Smith, 2007). This transform takes the discretized values and performs point wise multiplication between the values of the function and the basis function for a specific wave number. This is equivalent to the dot product between vectors which measures how large the projection of one vector on the other. Intuitively, this process "measures" how much of the discretized function is aligned with the basis function of that specific wave number. This measure is the coefficient for that wave number. The computation is repeated for all wave numbers. The inverse DFT calculates the function values from the computed coefficients as shown in equation (2.12).

$$\hat{c}_k = \sum_{n=0}^{N-1} f_n e^{-2\pi i k n/N} \tag{2.11}$$

$$f_n = \sum_{k=-N/2}^{N/2-1} \hat{c}_k e^{2\pi i k n/N} \tag{2.12}$$

The implementation of the DFT and inverse DFT can affect the performance of the operation. The naive approach of implementing the equations exactly, while flexible in ways such as the wave numbers to compute coefficients for, is not the most efficient way. For faster computation, the aptly named Fast Fourier Transform (FFT) algorithm is used (Chu & George, 2000). This algorithm computes the DFT efficiently by exploiting symmetries that exist in and properties of the DFT. The Cooley-Tukey algorithm, arguably one of the most well known variation, uses a divide and conquer strategy by dividing the DFT into two at each step. This means that the discretized function would need to have discretization that is of the power of two. Methods have been developed to get around this, such as padding the data with zeroes to the next power of two and scaling the results accordingly. FFT algorithms can achieve a computational complexity of $O(n \log(n))$ in comparison to $O(n^2)$ complexity of the naive DFT algorithm.

The Fourier series has an important property that is very important to this study.

Taking the derivative of the Fourier series in equation (2.7) with respect to x results in equation (2.13). Assuming we represent the derivative function is represented as another Fourier series with coefficients $\hat{f'}_k$, the derivative coefficients can be computed from the coefficients of the original function f as shown in equation (2.14). This shows that the derivative of a Fourier series can be obtained trivially by simply multiplying the coefficients with the coefficients of the basis function exponents. The antiderivative can similarly be obtained by using the same relation and dividing the coefficients of the original function with the same scaling factor. Notice that the wave number is included in the scaling factor. This means that higher frequency coefficients get scaled more. For derivatives this amplifies the higher frequencies more and the antiderivative will smooth out the function by suppressing the higher frequencies.

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}x}f(x) &= \sum_{k=-N/2}^{N/2-1} \hat{c}_k(2\pi ik/L)e^{2\pi ikx/L} \qquad (2.13)\\ &= \sum_{k=-N/2}^{N/2-1} \hat{f'}_k e^{2\pi ikx/L}\end{aligned}$$

$$\hat{f'}_k = (2\pi ik/L)\hat{c}_k \qquad (2.14)$$

This property of Fourier series is used in the spectral method (Boyd, 2001; Shen, 2011). The ability to compute derivatives as a simple multiplication motivates its use to solve PDEs. By using Fourier series to represent the solution to PDEs, one can isolate the solution coefficients in terms of the other values. This is a powerful tool for linear PDEs. For nonlinear PDEs like the Burgers' equation, the solution coefficients are not able to be obtained as easily. This is due to the multiplication between the solution and its derivatives. The solution is to iteratively compute the value of the solution coefficient until it converges. For problems in domains with more than one dimension, such as space and time for the Burgers' equation, one of the dimensions can be integrated separately using a different method like Euler's method. This hybrid approach is called the Method of Lines. Using the Fourier series, the spectral method is able to solve PDEs to a high degree of accuracy and efficiency for smooth functions.

2.5 Method of Manufactured Solutions

A method commonly used for testing PDE solvers is the Method of Manufactured Solutions (MMS). This is to verify that the program runs but also is accurate in its solutions. The concern of this method is not the physical realism of the solutions, rather it is a purely mathematical concern. This is because verification of a method means making sure the code accurately implements the mathematical equation. This very specific goal, places no requirement on the realism of any solutions used in the verification of the method (Roache, 2002; Salari & Knupp, 2000; Vedovoto et al., 2011).

The method solves PDEs by simply utilizing the equations involved. This method requires that the form of the PDE used must account for a forcing term. This is typically done by adding the forcing term as the residual. The residual is simply the right-hand side after moving all terms to the left-hand side. Without a forcing term, which translate to a forcing term of zero, the residual is the discrepancy between the right-hand side and left-hand side of the equation. By encapsulating the residual in the forcing term, any solution can be chosen because the discrepancy is now accounted as the outside forces the forcing term represent. This is the principle behind MMS. Combining this approach with the spectral method and Fourier series with arbitrary coefficients, the forcing term can be computed as a Fourier series given the solution as another Fourier series and other parameters. For verification purposes of numerical methods, using the solutions and other parameters generated with MMS require the implemented method to admit forcing terms. An example of MMS can be seen in section 4.1.2.

2.6 Least Squares Support Vector Machine

An arguably fundamental model widely used both in pedagogical settings or otherwise is the Support Vector Machine (SVM). It dates back to works by Vapnik and Lerner (1963) and Vapnik and Chervonenkis (1964). As the development on SVM continued, what originally was a model for classification of separable data generalized to regression tasks as well (Vapnik, 2000). The support vector machine is essentially a straight line fitting model. We will refer to the line as a hyperplane for a more general term in multiple dimensions. For classification, this hyperplane divides the samples into two sides. One side being of one class and the other for samples not in that class. The optimization is done such that the line is at the maximum distance from the nearest

points. This formulation still has a major weakness, which is that the model many not find a line that perfectly separates all the points. To allow for some error, slack terms are added to the constraints of the model. For regression, SVMs follow a similar approach with classification. However, rather than separating the inputs, the line is optimized to best fit the samples.

The quadratic nature of the SVM objective function means that it is guaranteed to have an optimum point. However, with the motivation to further simplify the SVM, the Least-Squares Support Vector Machine (LSSVM) was introduced by Suykens (2005). The aim of this proposed method was to have a simpler formulation of the SVM while retaining the majority of benefits it provided. Here, we will focus on the regression problem and as such the LSSVM for regression. Given a pair of features $\mathbf{x} \in \mathcal{X}$ and labels $y \in \mathcal{Y}$, the approximate linear model is given by equation (2.15) where W is the model weights and b is the bias which are both parameters of the model.

$$y = W^\intercal \mathbf{x} + b \tag{2.15}$$

Training the model means optimizing the model by adjusting its parameters to minimize the objective function in equation (2.16) where e_k is the difference between the prediction $W^\intercal \mathbf{x} + b$ and the label y.

$$\min_{W,b,e} J(W, e) = \frac{1}{2} W^\intercal W + C \frac{1}{2} \sum_{k=1}^{n} e_k^2 \tag{2.16}$$

Such that

$$y_k = W^\intercal \mathbf{x}_k + b + e_k \qquad k = 1, \ldots, n \tag{2.17}$$

To find the optimal point for the objective function given the constraint equation (2.17), we must construct the Lagrangian which combines the objective function with the constraint. The Lagrangian is shown in equation (2.18) where α_k is the Lagrange multiplier for the k^{th} sample constraint.

$$L(W, b, e; \alpha) = \frac{1}{2} W^\intercal W + C \frac{1}{2} \sum_{k=1}^{n} e_k^2 - \sum_{k=1}^{n} \alpha_k \left(W^\intercal \mathbf{x}_k + b + e_k - y_k \right) \tag{2.18}$$

To compute the optimal parameters, first the point at which the gradient is zero for

each parameter must be found. This is simply by equating to zero the derivative of the Lagrangian with respect to each parameter. This is called the Karush-Khun-Tucker optimality condition. Computing this for our Lagrangian results in equation (2.19).

$$\begin{aligned}
\frac{\partial L}{\partial W} = 0 \rightarrow W &= \sum_{k=1}^{n} \alpha_k \mathbf{x}_k \\
\frac{\partial L}{\partial b} = 0 \rightarrow 0 &= \sum_{k=1}^{n} \alpha_k \\
\frac{\partial L}{\partial e_k} = 0 \rightarrow \alpha_k &= Ce_k \qquad k = 1, \ldots, n \\
\frac{\partial L}{\partial \alpha_k} = 0 \rightarrow y_k &= W^\intercal \mathbf{x}_k + b + e_k \qquad k = 1, \ldots, n
\end{aligned} \tag{2.19}$$

Substituting the relationships for the optimal parameters, we can rewrite the Lagrangian in terms of the Lagrangian multipliers and the bias. The reasoning for this choice will be explained later. The substitution yields equation (2.21).

$$\begin{aligned}
L(W, b, e; \alpha) &= \frac{1}{2} \left(\sum_{k=1}^{n} \alpha_k \mathbf{x}_k^\intercal \right) \left(\sum_{l=1}^{n} \alpha_l \mathbf{x}_l \right) + \frac{1}{2C} \sum_{k=1}^{n} \alpha_k^2 \\
&\quad - \sum_{k=1}^{n} \alpha_k \left(\sum_{l=1}^{n} \alpha_l \mathbf{x}_l^\intercal \mathbf{x}_k + b + \frac{\alpha_k}{C} - y_k \right) \qquad (2.20) \\
&= \frac{1}{2} \sum_{k=1}^{n} \sum_{l=1}^{n} \alpha_k \alpha_l \mathbf{x}_k^\intercal \mathbf{x}_l + \frac{1}{2C} \alpha^\intercal \alpha \\
&\quad - \sum_{k=1}^{n} \sum_{l=1}^{n} \alpha_k \alpha_l \mathbf{x}_l^\intercal \mathbf{x}_k - b \times 0 - \frac{1}{C} \alpha^\intercal \alpha + \sum_{k=1}^{n} \alpha_k y_k \\
&= -\frac{1}{2} \sum_{k=1}^{n} \sum_{l=1}^{n} \alpha_k \alpha_l \mathbf{x}_k^\intercal \mathbf{x}_l - \frac{1}{2C} \alpha^\intercal \alpha + \alpha^\intercal \mathbf{y} \qquad (2.21)
\end{aligned}$$

subject to

$$\sum_{k=1}^{n} \alpha_k = 0 \tag{2.22}$$

To extend the model further, the kernel trick was introduced by Vapnik (2000). This allows SVMs to learn nonlinear problems. In order to achieve, the input data is mapped to a higher dimension. For example, a set of points along one dimension can be lifted to two dimension by adding another coordinate that is the square of the first.

This operation leads to the line being able to separate the samples in the middle from the samples towards both edges by intersecting the now parabolic line of samples in two points instead of the original one. Doing this with more dimensions mean that the model can add more nonlinearity. This is done by simply replacing the features $\mathbf{x}_k$ with the mapped features $\phi(\mathbf{x}_k)$. However, for high dimensional mappings this causes the size of the weights W to increase accordingly. This is especially concerning when we use mappings to infinite dimensions. Fortunately, the dual formulation only contains inner products (dot products for Euclidean spaces) for all operations with the features. This means that a kernel function K can be used to compute these inner products in the higher dimensions without having to deal with the actual high dimensional mappings directly. The first term of the dual can be reformulated as a matrix of the kernel values multiplied by the vectors of Lagrange multipliers α. The kernel matrix Ω has elements of $K(\mathbf{x}_k, \mathbf{x}_l)$. In addition, the minimization of the derived dual in equation (2.21) is equivalent to the maximization of the equation multiplied by -1. Putting all the above together we can construct the block matrix notation for the LSSVR optimal parameters shown in equation (2.23) with $\mathbf{1}_n = \langle 1, \ldots, 1 \rangle$, $\mathbf{y} = [y_1, \ldots, y_n]$, and $\alpha = [\alpha_1, \ldots, \alpha_n]$. Solving for the parameters which are the Lagrangian multipliers α and bias b can be done by doing least-squares.

$$\begin{bmatrix} 0 & \mathbf{1}_n^\intercal \\ \mathbf{1}_n & \Omega + \frac{I}{C} \end{bmatrix} \begin{bmatrix} b \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ \mathbf{y} \end{bmatrix} \tag{2.23}$$

The construction of matrices in equation (2.23) and the solving procedure is presented by the pseudocode in algorithm 1. Given a training set of length n, features $\mathbf{X}$, and labels $\mathbf{y}$. Training the LSSVM means computing the values of Lagrange multipliers α and bias b where K is the kernel function, $\mathbf{A}$ is the right-hand side matrix of size $n+1$ by $n+1$, $\mathbf{H}$ is sum of the kernel matrix and the diagonal matrix of reciprocals of the regularization parameter C of size n by n, $\mathbf{I}$ is the identity matrix, $\mathbf{B}$ is the right-hand side vector of size $n+1$, and $\mathbf{S}$ is the solution parameter vector of size $n+1$.

Algorithm 1 LSSVR Training

procedure SOLVELSSVR($\mathbf{X}, \mathbf{y}, C$)
  $\Omega \leftarrow [[]]$ ▷ Construct matrix of inner products in high dimensional space
  for $k = 0 \rightarrow n$ **do**
    for $l = 0 \rightarrow n$ **do**
      $\Omega_{k,l} \leftarrow K(\mathbf{X}_k, \mathbf{X}_l)$
    end for
  end for
  $\mathbf{H} \leftarrow \Omega + \frac{\mathbf{I}}{C}$
  $\mathbf{A} \leftarrow [[]]$ ▷ Construct left-hand side matrix
  $\mathbf{A}_{0,0} \leftarrow 0$
  for $k = 0 \rightarrow n$ **do**
    $\mathbf{A}_{k+1,0} \leftarrow 1$
    $\mathbf{A}_{0,k+1} \leftarrow 1$
  end for
  for $k = 0 \rightarrow n$ **do**
    for $l = 0 \rightarrow n$ **do**
      $\mathbf{A}_{k+1,l+1} \leftarrow \mathbf{H}_{k,l}$
    end for
  end for
  $\mathbf{B} \leftarrow []$ ▷ Construct left-hand side of the equation
  $\mathbf{B}_0 \leftarrow 0$
  for $k = 0 \rightarrow n$ **do**
    $\mathbf{B}_{k+1} \leftarrow \mathbf{y}_k$
  end for
  $\mathbf{A}^\dagger \leftarrow leastSquares(\mathbf{A})$ ▷ Compute solution using least-squares
  $\mathbf{S} \leftarrow \mathbf{A}^\dagger \mathbf{B}$
  $b \leftarrow \mathbf{S}_0$
  for $k = 0 \rightarrow n$ **do**
    $\alpha_k \leftarrow \mathbf{S}_{k+1}$
  end for
  return α, b
end procedure

After training the model can be used for prediction of unseen features. The pseudocode for prediction is shown in algorithm 2 for prediction features $\mathbf{U}$ with p samples. The trained model uses the training features themselves $\mathbf{X}$ with n samples, the learned multipliers of training points α, and the bias b.

Algorithm 2 LSSVR Prediction

procedure PREDICTIONLSSVR($\mathbf{U}, \alpha, \mathbf{X}, b$)
    $\Omega \leftarrow [[]]$ ▷ Construct matrix of inner products in high dimensional space
    for $k = 0 \rightarrow p$ **do**
        for $l = 0 \rightarrow n$ **do**
            $\Omega_{k,l} \leftarrow K(\mathbf{U}_k, \mathbf{X}_l)$
        end for
    end for
    $\mathbf{v} \leftarrow \Omega\alpha + \mathbf{1}_m b$
    return $\mathbf{v}$
end procedure

The choice of kernel used to construct the kernel matrix is problem specific. Three kernels are most commonly used in SVMs. These kernels are:

1. Linear Kernel
   This kernel is the simplest with it being the dot product. It is recommended for cases where the number of features overwhelm the number of samples. The kernel is shown in equation (2.24). Because this kernel operates in the original feature space, it is effective for linear problems.

$$K(\mathbf{x}_k, \mathbf{x}_l) = \mathbf{x}_k \cdot \mathbf{x}_l \tag{2.24}$$

2. Polynomial Kernel
   The polynomial kernel is shown in equation (2.25). This kernel allows the model to learn more complex nonlinear decision boundaries. The parameter c is a constant and d is the degree of the polynomial. The degree of the polynomial need to be monitored closely in order to avoid overfitting.

$$K\left(\mathbf{x}_k, \mathbf{x}_l\right) = \left(\mathbf{x}_k \cdot \mathbf{x}_l + c\right)^d \tag{2.25}$$

3. Radial Basis Function Kernel
   The RBF kernel is presented in equation (2.26). This kernel maps the inputs to infinite dimensional space. This enables the SVM to create decision boundaries which are not rigid. The RBF kernel is especially effective for nonlinear relationships between features and labels. The kernel has a parameter σ which

is the scale parameter. This parameter affects how the kernel treats structures of different scales in the features. This parameter can be approximated roughly from the average variance of across all features multiplied by the number of features.

$$K(\mathbf{x}_k, \mathbf{x}_l) = \exp\left(-\frac{\|\mathbf{x}_k, \mathbf{x}_l\|^2}{2\sigma^2}\right) \tag{2.26}$$

After an SVM model is trained, it can be dissected and analyzed to interpret the decisions the model came up with using the work of Üstün et al. (2007). There are two tools that (Üstün et al., 2007) introduced, which are the Correlation Image (CI) and the p-vector. First, the correlation image is aimed to present the contribution of each input feature to the kernel matrix. This is because of the loss of input feature information after computing the inner product using the kernel function. Given N training samples and m input features, the correlation image matrix can be computed as the matrix multiplication between the kernel matrix of the training input features with a shape of (N, N) and the matrix of all training input features with a shape of (N, m). The matrix multiplication results in a matrix with the same shape as the matrix of training samples. To show the correlation, the rows of the result matrix will need to be sorted according to the output values.

The second tool is the p-vector. This tool seeks to inform the learned contributions of each input feature towards the output. The Lagrangian multipliers contain information on the contribution of each training sample towards the output. To link this with the input features, the multipliers are multiplied against the matrix of training inputs. This is done by transposing the matrix of training inputs and performing a matrix multiplication against the Lagrange multipliers. The result shows how much contribution each input contributes to the output. If we generalize to multiple outputs, the result is a matrix with the rows representing input features and columns representing each output.

Both of these tools allow the user to analyze which inputs are important. In addition, the relationships learned can also be seen albeit with the caveat that the contributions are basically sums different contribution terms. And, there is method to separate each contribution term currently. Despite the shortcomings, these tools are invaluable in interpreting and understanding the learned model.

LSSVMs have many benefits but also its own trade-offs. The main disadvantage of LSSVMs is the fact that they do not have the sparse property of SVMs. This can result in more computation during training and prediction since every single sample would need to be included in the computations. A simple mitigation can be done by filtering training samples with small absolute values of Lagrangian multipliers (Haifeng Wang & Dejin Hu, 2005). However, the benefits of simple matrix operations outweigh the disadvantages. This is because of the parallelization benefits of algorithms for simple matrix operations. With access to parallel hardware accelerators, the speedup for both training and inference can be very large.

2.7 Regression Metrics

In order to evaluate the prediction of the model, four metrics are used. The metrics are divided into absolute metrics and relative metrics. Absolute metrics require context of scale to interpret their meaning (Alpaydın, 2020; Botchkarev, 2019). Meanwhile, relative metrics often have a set range of values that it can have. This allows for quick interpretation of the severity of prediction errors. The metrics we discuss here are:

1. MSE and RMSE
   The Mean Squared Error (MSE) and its root RMSE, are both very popular metrics. Given target values y_i and predictions y'_i such that the difference is $e_i = y'_i - y_i$, the MSE is shown in equation (2.27).

   $$MSE = \frac{1}{n}\sum_{i=1}^{n} e_i^2 \tag{2.27}$$

   For The RMSE is simply the square root of the MSE. The MSE has the property of inflating large errors. The squaring also makes it so that the values are positive. This means that the absolute lowest error value is zero. However, the squaring does complicate the interpretation because the units are no longer the same as the predictions. Here the RMSE comes into play, because of the root square, the units are back to the original target units. This makes the MSE and RMSE very useful.

2. MAE
   The second metric is the MAE. This metric is calculated as the mean of the absolute differences between the prediction and the target. The MAE is calculated as shown

in equation (2.28).

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|e_i| \tag{2.28}$$

This metric is good at representing the errors without any distortions. Because there is no nonlinear components, the metric is very straightforward to interpret. Due to the use of the absolute value, the minimum value of the metric is also zero.

3. R^2

   The third metric is the coefficient of determination. This metric can be seen as the ratio between the model's accuracy and a baseline model of mean values. Mathematically, this metric is calculated using equation (2.29) given target values y_i and predictions y_i'.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - y_i')^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{2.29}$$

   Because the R^2 is a relative metric, there is a range of possible values. First, the highest value possible is one. This happens when the predictions are exactly the same as the targets. For predictions that are no better than the average value, the score is zero. Scores below zero mean that the predictions are worse than the average value.

4. sMAPE

   The final metric is the symmetric Mean Absolute Percentage Error. This metric has a range of values between zero for no error and two for high error. The values can be scaled by multiplying by 100 to obtain the percentages. Given target values y_i and predictions y_i' such that the difference is $e_i = y_i' - y_i$, the sMAPE is calculated using equation (2.30) where ϵ is a small value to prevent divisions by zero.

$$sMAPE = \frac{2}{N}\sum_{i=1}^{N}\frac{|e_i|}{\max(|y_i| + |y_i'|, \epsilon)} \tag{2.30}$$

   Despite the name, sMAPE does have issues with asymmetry arising from the denominator. For cases where the prediction overpredicts and underpredicts by

the same amount, depending on the value of the target, the metric may penalize the prediction closer to zero more than the one that is further. This is despite both predictions being the same absolute difference away from the target value.

2.8 Python Language and PyTorch Library

The origins of the Python programming language began in the late 1980s as with its designer Guido van Rossum (G. V. Rossum, 2009). The language began with internal releases at the designer's workplace. Over thirty years later, today, the language has reached the stable version of 3.14. Many today use Python as a high level language with very convenient features. Python is a multi-paradigm language with support for many styles of programs from standalone scripts to entire applications that use paradigms like object-oriented programming. Python also acts as a bridge for using software components written in other languages (G. Rossum, 1998). The language has a syntax that is relatively easy to understand. This has brought many people to the ecosystem and has consistently ranked the language as one of the most used programming languages in the world (stackoverflow, 2024). Specifically for the field of machine learning, Python enjoys a wide following (Jetbrains, 2023). The machine learning ecosystem within python leverages Python's capability to interact with other languages to provide performant code that is still easy to use. This can be seen across different Machine Learning libraries in Python, such as Tensorflow, PyTorch, and Jax.

PyTorch specifically, was originally developed by Meta AI as an open source project. Today it is part of the Linux Foundation (Jim Zemlin, 2022). As one of the most successful and important machine learning software project, PyTorch has a big community which provides support. The PyTorch ecosystem of other projects built around PyTorch, such as torchdiffeq (R. T. Q. Chen, 2021), are part of what makes PyTorch an appealing choice when undertaking a Machine Learning project. At its core, PyTorch provides an implementation of tensors which are generalizations of matrices for arbitrary dimensions. The tensors are surrounded by operations that are related to its manipulation and use, such as matrix multiplications or even linear solvers. In addition, PyTorch also provides many of the tools for working with Neural Networks. Other parts of PyTorch deal with the data and preprocessing, such as a way to randomly sample data or the family of operations surrounding FFTs. The features and all other benefits of both Python and PyTorch, allow for rapid prototyping and is conducive to research projects.

The community means help is also not far away.

CHAPTER III

RESEARCH METHODOLOGY

3.1 Research Design

This study is carried out under the framework of a research design. In this section, this design will be laid out and explained. The design includes the process from the beginning to the end of the overall study. The research design is visually depicted in figure 3.1.

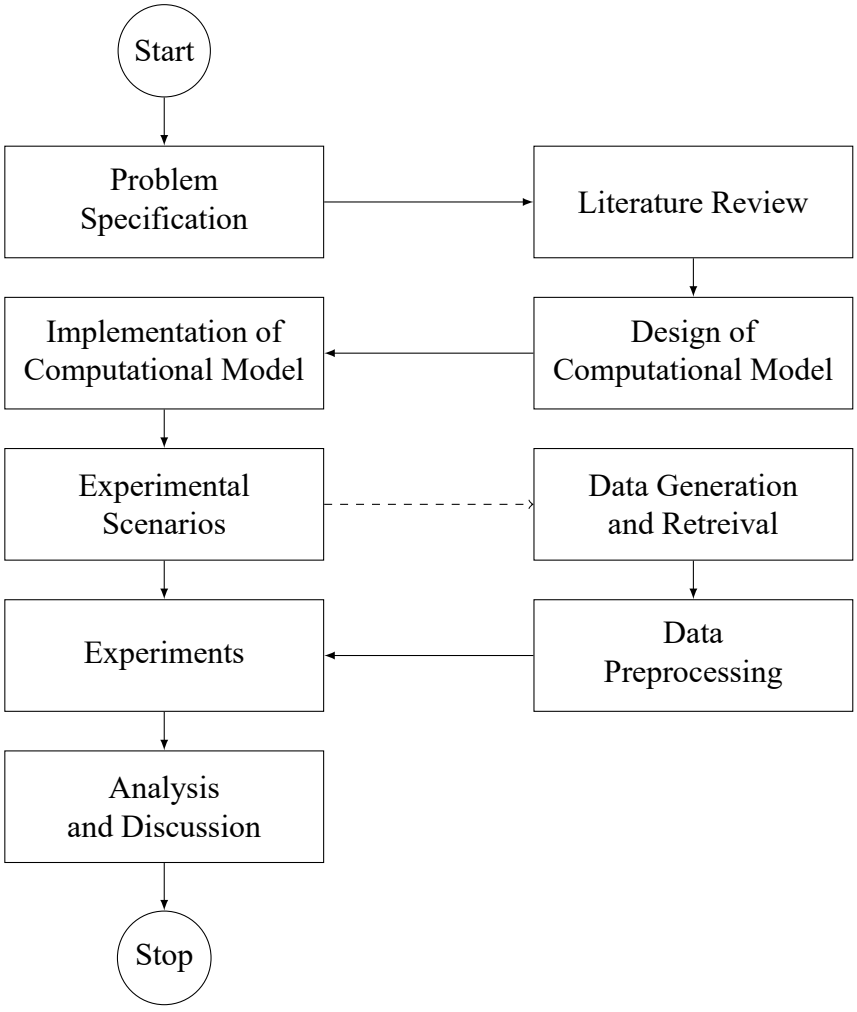


Figure 3.1: Research design diagram

The study is carried out in 9 main phases. Each phase has a specific aim to

accomplish such that the subsequent phases are able to proceed. Each phase is explained as follows:

1. Problem Specification
   In this phase, relevant literature of related works and other supporting materials will be reviewed to lay the groundwork of this study. Specifically, the literature review will encompass partial differential equations and their real world associations, traditional methods employed to solve them, systems with partially known or unknown governing equations, machine learning approaches to the problem, least-squares support vector machines, performance metrics and tools that will be used such as the PyTorch library.

2. Design of Computational Model
   This phase of the study is allocated to the design of the computational model based on the literature that has been reviewed. The design of the overall process includes data retrieval, preprocessing, and the core spectral regression model itself.

3. Implementation of Computational Model
   After the computational model design is completed, the model is implemented in the Python Language using the PyTorch library as a core component. Implementation of the computational model is done using an iterative software development model. This development model was chosen due to its adaptability to unexpected challenges which is necessary because of the challenges and unknowns with developing a novel method.

4. Planning of Experimental Scenarios
   To properly gauge the performance of the proposed method at approximating operators, specifically the solution operator of partial differential equations, experimental scenarios are developed in this phase. Specifically, the scenarios serves three goals which are proof of concept or validation that the method is able to learn operators, a case study of a more complex system with real world data, and a using the model as a surrogate. Together, these experimental scenarios determine the applicability of the model to the problem. In addition, the model will be compared to a baseline model in order to put into context the proposed model's performance. In addition, kernel-input correlation matrices and input-Lagrangian multiplier inner product matrix will also be computed.

5. Data Generation and Retrieval

   This phase of the study is tied to the experimental scenarios that have been developed. After determination of the scenario specifics, the data that will be used for training, validation, and testing of the model in each scenario will be either generated or retrieved from an external source. In addition, in each scenario the hyperparameters of the model will be determined using Bayesian optimization as an alternative to the traditional grid search method.

6. Data Preprocessing

   The next phase after data generation or retrieval is preprocessing. This is a crucial step in allowing the core LSSVR to learn the mappings in spectral space. In order to do this, the data will need to be reshaped, and important features selected for. Finally, inputs to the LSSVR will need to be normalized or scaled such that the LSSVR model can much more easily learn the data.

7. Experiments

   This phase of the study executes the experimental scenarios that was determined previously. The preprocessed data will be used in accordance to the preplanned scenarios.

8. Analysis and Discussion

   The final phase of the study is the analysis and discussion of the results. Analysis of each experimental scenario will assess the extent of the model's capabilities. Another component of the analysis is interpretation the trained model and how it comes to the predictions that it makes. The discussion will also touch on the hyperparameter optimization and comparisons with the baseline model. This will show how the model performs differently compared to the baseline.

3.2 Literature Study Method

As a basis for this study, information surrounding the topic is collected from literature sources such as books, journal articles, and other academic works like dissertations. The tools that are used to find these sources include Google Scholar and Google Search. The search terms used start with two terms which are partial differential equations and machine learning. Based on reading the most relevant literature, further search terms are created from variants of previous search terms combined with terms from literature

that has been found.

3.3 Data Generation and Retrieval Method

Data to be used in this study are acquired in two different ways. The first method is data generation. This is motivated by the fact that some partial differential equations do not have much open and accessible real world data available. As such the systems are simulated using the equations themselves. In simple scenarios like computing the antiderivative u of some function f, this can be done by randomly generating u and then taking their derivatives to compute f. The random function generation itself is done by generating random coefficients for basis functions like the Fourier series. Once all functions are generated, random noise is added to ensure that the model is also robust towards inexact measurements. A diagram illustrating the process is shown in figure 3.2.

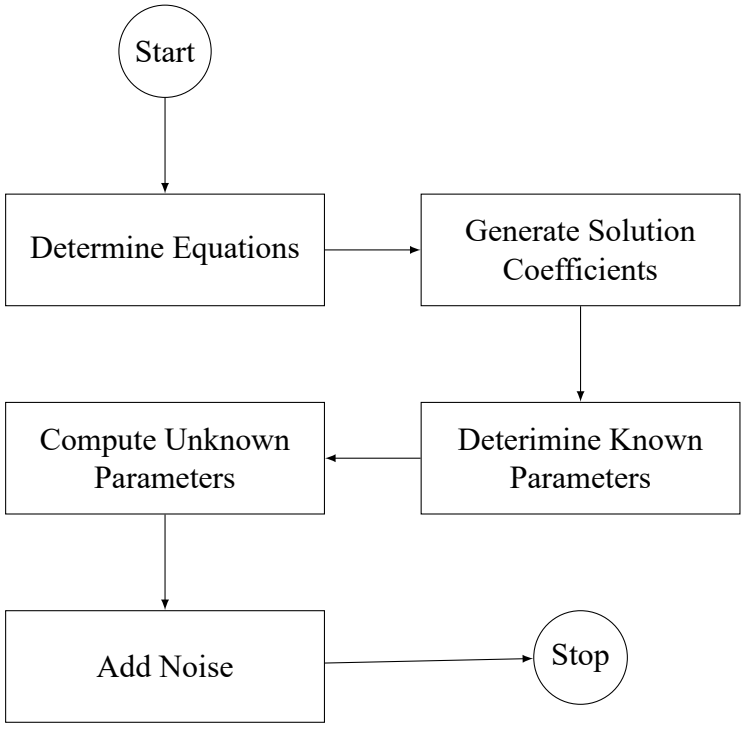


Figure 3.2: Data generation diagram

Data retrieval is the other option which in this case is used to retrieve data for weather prediction. Based on the scenario the data is to be used for, relevant subsets of the dataset is retrieved to a working machine. This study specifically uses a dataset

provided by the European Center for Medium Range Weather Forecast (ECMWF) named ERA5 hourly data on single levels from 1940 to present (C3S, 2018). The first step is selecting a geographical area to study, time period, NetCDF4 data format, and variables such as temperature at 2 meters above the surface. Then the second step is specifications are then used to request the data through the climate data store application programming interface. The downloaded data is then parsed using the Xarray library and preprocessed.

3.4 Computational Model Implementation Method

Implementation of the computational model follows an iterative development model. This model basically consists of repeated cycles or iterations of software development. Each cycle is loosely based on the waterfall development model, namely the processes of requirement gathering, analysis & design, implementation, and testing. This iterative property is crucial for this study because all the requirements cannot be known beforehand. This model minimizes the risk in developing complex software with partially known initial requirements. A diagram of the model can be seen in figure 3.3.

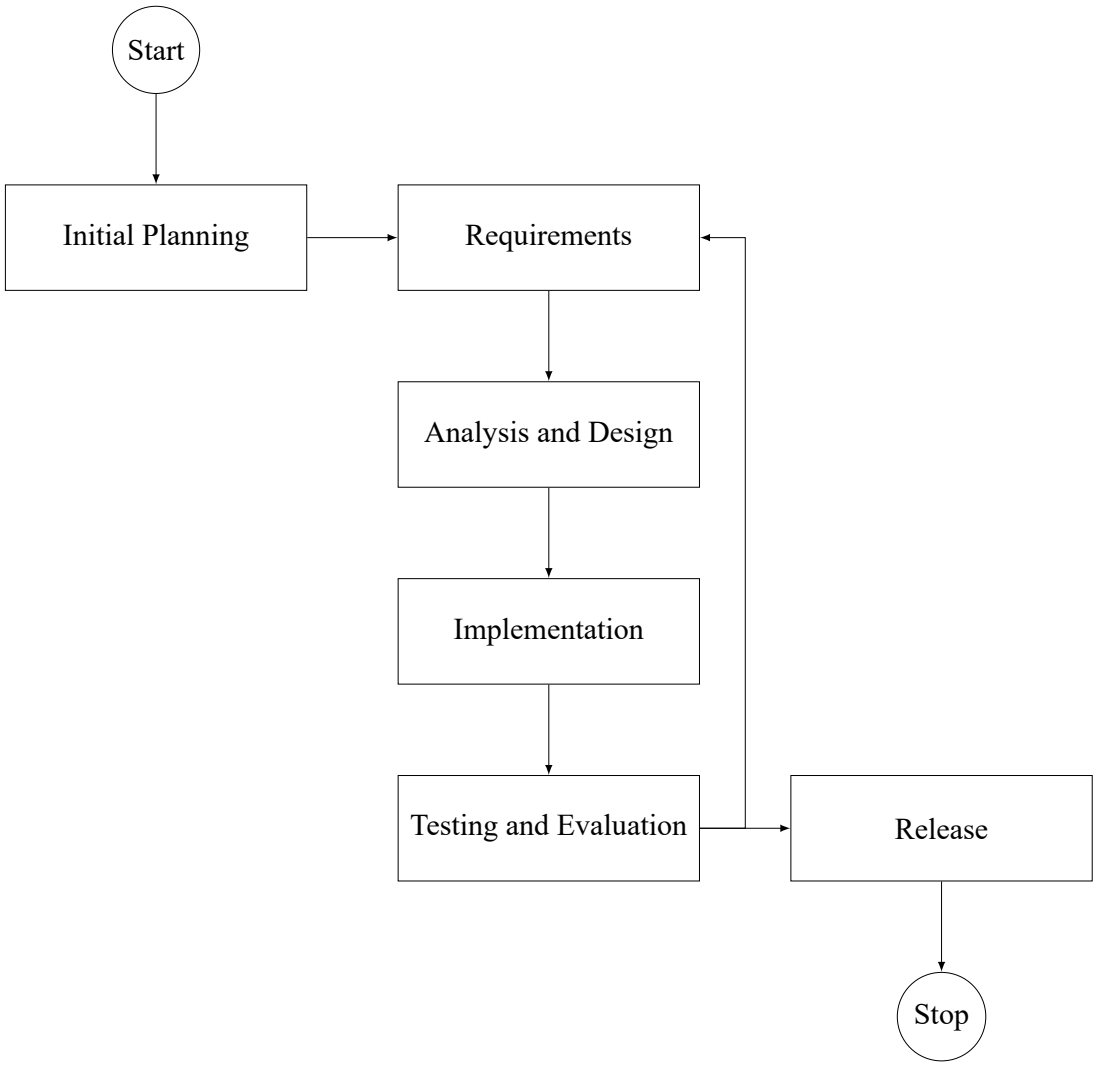


Figure 3.3: Iterative development model

This development model was adapted from Bittner and Spence (2006). Each process in the model is explained as follows:

1. Initial Planning

   The first process of the iterative development model sets up the parameters for the iterative development. These parameters are the broad scope of the project and initial requirements such as core components like LSSVR. The initial requirements are crucial in guiding the rest of the development as more requirements are gathered. The initial planning also determines technical choices like tooling and choice of language. The planning process also includes a rough architecture based

on the initial requirements.

2. Requirements

   This process separates out the backlog of requirements into those that will be worked on in the current iteration. These requirements ideally share some functionality in order to allow the same context to be retained within the iteration which reduces the burden of context switching.

3. Analysis and Design

   This process an analysis of the requirements of the current iteration is performed. This involves determining the constraints and goals for each requirement. Constraints also need to consider previous iterations such that the requirements can be fulfilled effectively and efficiently without regressing progress that has been made such as breaking existing functionality. From this analysis, the design is updated so that the requirement can be fulfilled. The design itself informs how the code should be structured. The interfaces used in the code are also specified.

4. Implementation

   The implementation process applies the design that has been produced. In addition to implementing the program, this process also implements tests which validate and verify the produced code. In the case of complex pieces of the design, an approach of creating tests and assertions first is taken. This way, the code can be run quickly and fail immediately to allow a quicker convergence on the intended functionality.

5. Testing and Evaluation

   Once the design is implemented, tests that have been created are run. Each feature is tested in an integrated manner to expedite the process while still providing enough confidence in the software. The results of the testing process are evaluated and any failed tests are resolved whether by reimplementation or reassessment of the correctness of the tests. Any missing functionality or feature discovered during evaluation are added to the backlog of requirements.

6. Release

   After an iteration, the current version of the software is released. The iterative nature of this development model necessitates the use of a versioning system. As such each release can be identified by a version number loosely based on semantic versioning (Preston-Werner, 2023). This version numbering system consists of

major number that indicates breaking changes, minor numbers which indicate features and non-breaking changes, and lastly patch numbers are used for fixes. A version number is read as `MAJOR.MINOR.PATCH`. During active development, only major version 0 will be used to prevent too many major versions. The released software is hosted in its own GitHub repository at https://github.com/nidduzzi/SpectralSVR.

3.5 Research Tools

The tools that are used in this study can be categorized into hardware and software. The following are hardware that are used in this study:

- Kaggle CPU Kernel ("Getting Started on Kaggle | Kaggle," n.d.)
  - 4 Cores Intel Xeon
  - 30 Gigabytes RAM
  - 20 Gigabytes Working storage
- Kaggle GPU Kernel ("Getting Started on Kaggle | Kaggle," n.d.)
  - 4 Cores Intel Xeon
  - 29 Gigabytes RAM
  - 1 Nvidia P100 GPU
  - 20 Gigabytes Working storage
- Personal Computer
  - Intel i5-8300H
  - 16 Gigabytes RAM
  - Nvidia GeForce GTX 1060 Mobile
  - 1.5 Terabytes Storage

The software tools used in this study are the following:

- Python

- Jupyter Notebook
- Git
- Poetry Python package manager
- Web Browser
- LaTeX

CHAPTER IV

RESULTS AND DISCUSSION

This chapter is organized into five sections which are the results of data generation, data retrieval, design of computational model, implementation of computational model, experimental scenarios, and the experiments themselves.

4.1 Data Generation

This study uses two data acquisition approaches. The first approach which is discussed in this section is data generation. This process involves manufacturing data randomly in such a way that they obey the PDEs to be modeled. The generated data consists of features and labels. These features and labels in the context of PDEs are parameters and the solution of the PDEs, respectively. The PDEs enforce a relationship between parameters and the solution. This relationship is implicitly encoded into the features and labels, which is what machine learning models can learn. Because the solution and parameters to PDEs are functions and there are many kinds with different properties, generating all the different kinds of functions is a difficult task. This is why this study focuses on Fourier functions. This means that out of the space of all functions F which include categories such as polynomials $f(x) = a_n x^n + \cdots + a_1 x + a_0$ where $f \in F$, and a_n are constants; we only consider the subset of functions $U \subset F$ which are of the form $u(x) = \sum_k^m c_k e^{2\pi i k \frac{x}{P}}$.

Using the subset of Fourier functions has several benefits which has motivated the choice. First these functions are characterized purely by their coefficients c_k, meaning there is no need to store discretized values which potentially saves space and computation. The second benefit is that other functions such as polynomials can be approximated by them using the Fourier transform. This means that despite limiting the set of functions to Fourier functions, the behavior of PDEs with other sets of functions can be approximated to a certain extent. The final benefit is the mature ecosystem around these functions which include fast algorithms for the Fourier transform and even numerical approaches for solving known PDEs like the previously mentioned spectral method which we discussed in section 2.4. In summary, the generated data consists of features and labels which are Fourier coefficients implicitly defining the relationship

enforced by a PDE and scenario. To proceed with the data generation, there are 4 steps involved:

1. Scenario and PDE Determination: The first step of data generation is determining the scenario related to the PDE or governing equations. First, the PDE to be modeled is determined based on the goals of the dataset. Then, one or more of the parameters is chosen to predict the solution. The chosen parameters will be called the input functions and the solution will be called output functions from here on. The second part of the scenario is the domain or more simply the physical space occupied by the system to be modeled. The domain will be used to compute the function values in relation to the physical space from coefficients of Fourier functions.

2. Parameter Determination: The second step is determining all parameters other than that input parameter based on the scenario and governing equation. These parameters may be coefficients such as material properties like density or viscosity, or forcing terms which model external influence on the system like a heat source in the case of the heat equation. Depending on the parameters, solutions of PDEs may behave very differently, such as the appearance of discontinuities in the solution to low viscosity Burgers' equation. Because of this, the choice of parameters is guided by what the dataset seeks to do.

3. Random Coefficient Generation: The third step of data generation is randomly generating the solutions for the chosen equation. Generating random functions in the space of Fourier functions exploits the fact that the coefficients characterize the function completely. By randomly assigning coefficients c_k, many functions can be generated randomly with very little cost. Since the coefficients need to be complex numbers as in equation (2.9), both real and imaginary components are generated independently by assigning a random value to each component for each wave number k. They are then put together again into complex numbers.

   Since only real functions are of interest in this study, the generation cost can be approximately halved. This is because for real functions the coefficients for negative wave numbers k are complex conjugate of the positive wave numbers. This means that once the positive coefficients are generated, one only needs to compute their complex conjugate and concatenate the result with the coefficients

of positive wave numbers. For dimensions higher than one, a simpler approach is used. The coefficients are generated for all wave numbers including the negative ones. The inverse Fourier transform is computed and this results in complex functions. The real components of these functions are kept and the Fourier transform is applied to get the coefficients of the real functions. Finally, these generated coefficients then be used with the basis functions as input functions.

4. Forcing Term Computation: Finally, in order to ensure that the generated solution and chosen parameters satisfy the equation, the forcing term is computed as the residual of the equation of interest with the parameters that was previously determined. This computation is done using the spectral method.

5. Function Value and Noise Computation: The generated solution and forcing functions are labeled as input and output functions. The values of both functions in the domain can then be computed using the scenario determined in the first step. Once all coefficients are generated, the function values are computed with an inverse Fourier transform. The real component of the function values are retained, and the imaginary component are zeroed out. Noise is added to the function values here as needed. The processed function values are then converted back into coefficients using a Fourier transform. This processing is necessary to ensure that the coefficients are only describing the real function.

These four steps are the general processes involved in generating datasets for this study. Further specifics of the generation process of each dataset is explained in their respective subsections.

4.1.1 Data Generation: Anti-derivative

The first dataset is a simple one dimensional derivative. This was chosen as a simple proof of concept of the ability to solve a differential equation. The equation is related to many real-world problems such as acceleration and speed. One can imagine a train in an ideal world where acceleration is directly translated into speed. When the train accelerates at time t by some amount a, we can expect the train to have some speed u. In this ideal world, the relationship between speed and acceleration can be modeled with a simple differential equation (2.1). This scenario is found in many real systems albeit often with many more details such as different components of acceleration from

friction, drag, gravity, and other factors. As previously mentioned in section 4.1, both velocity u and acceleration a are modeled with Fourier series in equations (4.1) and (4.2) respectively, where T is the period representing the length of the domain.

$$u\left(t\right) = \sum_{k} \hat{u}_k e^{2\pi i k t/T} \tag{4.1}$$

$$a\left(t\right) = \sum_{k} \hat{a}_k e^{2\pi i k t/T} \tag{4.2}$$

The domain of the scenario is a two-hour time window. This number was chosen because it is around the ideal length of travel time on high speed rail in comparison to air travel and car travel (Givoni, 2006; Nash, 1991; X. Wang & Zhang, 2019). This is the first step in generating this dataset.

In the second step, as the derivative equation (2.1) does not contain any parameters other than the acceleration which is the input parameter, there are no other parameters to determine. Therefore, the data generation process proceeds to generating coefficients for the speed functions $\hat{u}$. The coefficients are assigned randomly from a Gaussian distribution with a mean of zero and standard deviation of one. This choice was made such that most wave numbers will have a coefficient of close to zero leaving a sparse set of wave numbers to mostly affect the resulting function. In total, 5000 unique functions are generated with 50 complex coefficients each.

In the third step, the output function coefficients are computed. To find the relation between $\hat{u}_k$ and $\hat{a}_k$, we need to find substitutes for each term in equation (2.1). To do this, we take the derivative of equation (4.1) which result in equation (4.3). Using this we can substitute the terms in equation (2.1) with equations (4.2) and (4.3) giving equation (4.4). Finally, after some algebraic manipulation we obtain the relationship between the input a^* and output function u^* in terms of their coefficients in equation (4.5). One also needs to choose the integration constant $\hat{u}_0$ because at $k = 0$ equation (4.5) becomes a division by zero.

$$\frac{\mathrm{d}u\left(t\right)}{\mathrm{d}t} = a\left(t\right) \tag{4.3}$$

$$\sum_{k} \hat{u}_k \times \left(2\pi i k/T\right) e^{2\pi i k t/T} = \sum_{k} \hat{a}_k e^{2\pi i k t/T} \tag{4.4}$$

$$\hat{a}_k = \hat{u}_k \left(2\pi i k/T\right) \tag{4.5}$$

In the final step, using randomly generated values of $\hat{a}_k$, the corresponding values of $\hat{u}_k$ are computed with equation (4.5). These coefficients are then used to compute the function values inside the domain. The two-hour time window is represented by a grid of 500 discrete points. With the coefficients and evaluation points ready, the function values are computed using equations (4.1) and (4.2). Next, noise is added to the function values in order to motivate models learning on the dataset to generalize on noise. To also allow evaluation of how well the model performs with different levels of noise, the samples are duplicated into three copies for each high, medium and low noise levels. The function values of each copy is perturbed with noise from a Gaussian distribution with zero mean and standard deviation of some percentage of the average function value standard deviation. The percentages of each high, medium, and low noise levels are 5, 10, and 50 percent. The perturbed function values are then transformed back into their coefficients. An example generated function and its different perturbed versions is shown in figure 4.1a. The horizontal axis indicates time which is displayed in units of hours. The vertical axis indicates velocity in abstract units.

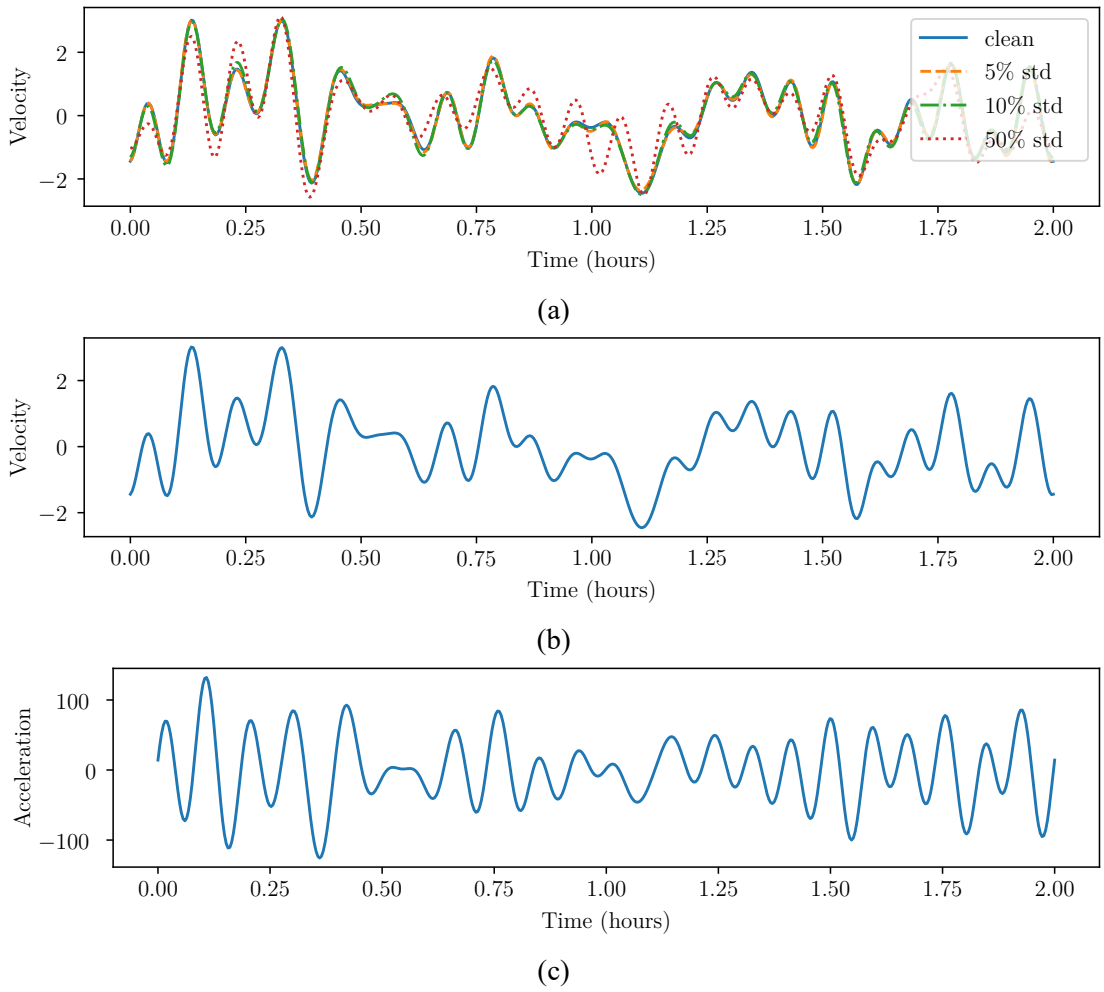


Figure 4.1: (a) A generated function with no noise (clean), low noise level (5%), medium noise level (10%), and high noise levels (50%). (b) Antiderivative function with no noise. (c) Derivative function with no noise.

4.1.2 Data Generation: Burgers' Equation

The second dataset generated in this study concerns the Burgers' equation. This equation has been used to model a variety of cases including fluid dynamics, traffic flow, and, shock waves (Bec & Khanin, 2007; Bonkile et al., 2018; Jameson, 2007; Orlandi, 2000). This equation is also used as a base problem for testing the effectivity of numerical methods in solving non-linear PDEs (Banks et al., 2012; Barter, 2008; Bonkile et al., 2018; Tabatabaei et al., 2007). The nonlinear term in the equation creates steep gradients and even shock waves which are discontinuous with low viscosity conditions. These challenges test the stability of numerical solvers. This is the scenario

and reason for the choice of this PDE. To control the viscosity and therefore the steepness of gradients, the formulation of the Burgers' equation considered in this study is the forced viscous Burgers' equation in one dimension. For a velocity of $u(x,t)$, viscosity of ν, and forcing term of $f(x,t)$, the formulation of the forced Burgers' equation is shown in equation (2.2).

The domain of we consider here is based on an exact solution of the equation. This is done so that the learned model can be tested on an exact solution and capture enough of the details within the solution within the domain. In this study, we use a specific solution by Wood (2006) which can be seen in equation (2.4). This solution is periodic in space but not in time. Therefore, the domain in space will be based on the spatial periodicity of the exact solution which is two. As such, the space domain spans from 0 to 2. The time domain, on the other hand, is chosen to span from 0 to 10 such that enough details of the solution is included.

Continuing to the second step, the parameters involved in equation (2.2) are the viscosity ν and the forcing term f. Since the forcing term is dependent on the solution in this step, the only parameter to determine is the viscosity. We choose three constant viscosity values which are 0.1, 0.01, and 0.0. These values are chosen so that a variety of behaviors are represented in the dataset from viscous to inviscid flow.

After this, in the third step, the solution functions u are generated with 8 modes in time and 8 in space and 500 unique function samples for each viscosity value. Since we don't factor the viscosity in at this point, for ease of comparison, we ensure the same solutions are generated every time by using a new instance of the generator with the same seed number each time. This results in the solutions being the same for all viscosity and other parameters of the equation are adapted to this.

Next we account for the different viscosity values in this stage. The forcing term is computed using equation (2.2). The solution field u and forcing term f are modeled with Fourier series in equations (4.6) and (4.7) where k is a vector of spatial and temporal wave number such that k_t is the temporal wave number and k_x is the spatial wave

number. The period in time is represented by T and in space it is L.

$$u\left(x,t\right)=\sum_{k_x}\sum_{k_t}\hat{u}_k e^{2\pi i(k_x x/L+k_t t/T)} \tag{4.6}$$

$$f\left(x,t\right)=\sum_{k_x}\sum_{k_t}\hat{f}_k e^{2\pi i(k_x x/L+k_t t/T)} \tag{4.7}$$

Substituting the terms in equation (2.2) with the respective Fourier series, we get the equation equation (4.9). A problem one notices is the nonlinear term $u\frac{\partial u}{\partial x}$. A naive approach would multiply all wave numbers terms with each other as shown in equation (4.8).

$$u\frac{\partial u}{\partial x}=\left(\sum_{k_x}\sum_{k_t}\hat{u}_k e^{2\pi i(k_x x/L+k_t t/T)}\right)\times\left(\sum_{k_x}\sum_{k_t}(2\pi i k_x/L)\hat{u}_k e^{2\pi i(k_x x/L+k_t t/T)}\right) \tag{4.8}$$

This is computationally expensive operation with N^2 multiplications (Larios, 2021; Orszag, 1972; Roberts & Bowman, 2011; Shen, 2011). To avoid this complexity, the term is first reformulated into $\frac{\partial}{\partial x}(u^2/2)$. Then to avoid aliasing, the coefficients are padded with 50% zeros such that the padded coefficients are 3/2 times the size of the original (Larios, 2021; Orszag, 1971). Then we use the padded coefficients to compute the function values in the physical domain using the inverse transform. And then, the point wise squaring operation is performed and transform the results back to spectral domain. The resulting coefficients are then trimmed back to their original size before padding. Finally, we multiply the resulting coefficients $\hat{uu}_k$ by the derivative constants. This operation is much more efficient with only $1.5N$ multiplications. This is still more efficient than the naive approach even after taking into account the transforms involved adds $O(2\times 1.5Nln(1.5N))$ operations if using a Fast Fourier Transform algorithm. Putting all the above together, results in equation (4.9).

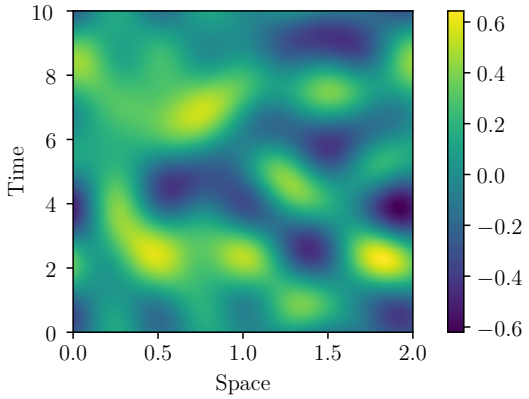
$$\begin{aligned}\sum_{k_x}\sum_{k_t}\hat{f}_k e^{2\pi i(k_x x/L+k_t t/T)} =&\sum_{k_x}\sum_{k_t}(2\pi i k_t/T)\hat{u}_k e^{2\pi i(k_x x/L+k_t t/T)}\\ &+\sum_{k_x}\sum_{k_t}(2\pi i k_x/L)\hat{uu}_k e^{2\pi i(k_x x/L+k_t t/T)}\\ &-\nu\sum_{k_x}\sum_{k_t}(2\pi i k_x/L)^2\hat{u}_k e^{2\pi i(k_x x/L+k_t t/T)}\end{aligned} \tag{4.9}$$

$$\sum_{k_x}\sum_{k_t}\hat{f}_k = \sum_{k_x}\sum_{k_t}(2\pi i k_t/T)\hat{u}_k + (2\pi i k_x/L)\hat{u}\hat{u}_k - \nu(2\pi i k_x/L)^2\hat{u}_k \tag{4.10}$$

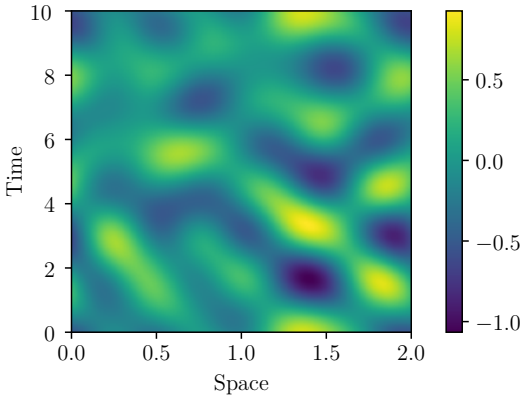
$$\hat{f}_k = (2\pi i k_t/T)\hat{u}_k + (2\pi i k_x/L)\hat{u}\hat{u}_k - \nu(2\pi i k_x/L)^2\hat{u}_k \tag{4.11}$$

After simplifying the equation, we get the coefficients as in equation (4.11). Using this equation, the exact forcing term corresponding to the randomly generated solution can be computed with relatively low cost. Finally, the function values of both the solutions and forcing terms are perturbed by adding Gaussian noise with a mean of zero and standard deviation of 10% of the function value standard deviation to the values of the inverse transform. The perturbed coefficients are then recomputed from the sum and the final perturbed functions are obtained.

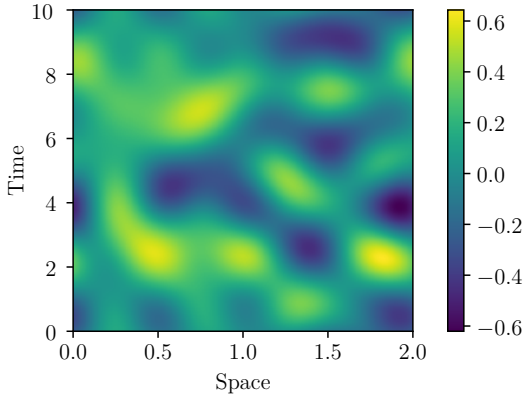
An example of the solution and forcing term pairs that has been generated is shown in figure 4.2. Using the same seed we can see that the solution functions of the same sample are the same. However, with the forcing terms, we see much larger amplitudes for higher frequencies for larger viscosity values. This is inline with equation (4.11). The squared multiplier gets large much more quickly for higher frequencies due to the squaring operation. The since this effect is controlled by the viscosity via multiplication, the higher the viscosity, the larger the high frequency amplitudes of the forcing term. This, in other words, means that more viscous fluids require a stronger forcing term to affect the solution in the same way with as fluids with a lower viscosity.



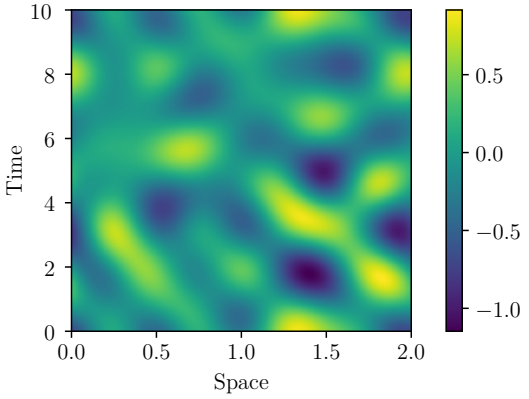
(a) Solution function for $\nu = 0$.



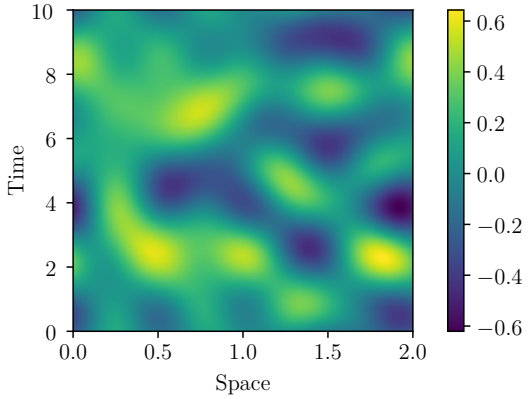
(b) Forcing function for $\nu = 0$.



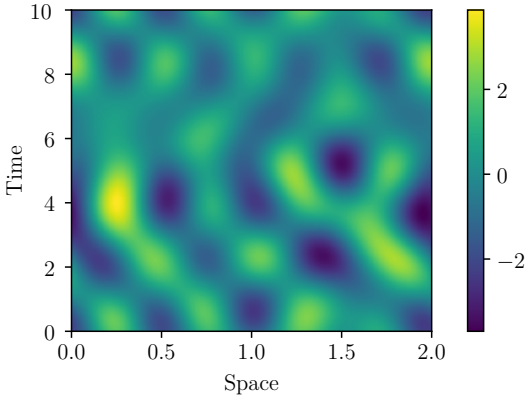
(c) Solution function for $\nu = 0.01$.



(d) Forcing function for $\nu = 0.01$.



(e) Solution function for $\nu = 0.1$.



(f) Forcing function for $\nu = 0.1$.

Figure 4.2: Example sample pairs of solution and forcing terms for the Burgers' equation dataset.

4.2 Data Retrieval

The second data acquisition approach retrieves data of a from an external source. This approach downloads the external data to a local machine. In this study, the specific dataset that will be used is the ERA5 dataset from the European Center for Medium-range Weather Forecast (ECMWF). Specifically, the ERA5 dataset is a reanalysis which means it combines observational data from all over the world in order to present a more complete picture of the weather system. This combination process, named 4D variational data assimilation, takes into account the physics known to be involved in the system. As an example weather station data which take measurements such as wind speed, humidity, and temperature of the immediate surroundings of the weather station is very limited in giving a broader picture of weather over a larger area. Even technologies like earth observations satellites are limited to what they can observe at any single point in time as most cannot see the entirety of the earth's surface at once. The ERA5 dataset specifically, assimilates previous forecasts with observational data every 12 hours. The ERA5 reanalysis is available at the following URL (https://cds.climate.copernicus.eu/datasets/reanalysis-era5-single-levels).

The original dataset made available by ECMWF has a grid size of $0.25°$ by $0.25°$ in latitude and longitude for atmospheric data. Data of oceanic waves are also available at a coarser grid size of $0.5°$ by $0.5°$. However, we will use a cloud optimized version called WeatherBench2 that is easier to retrieve (Rasp et al., 2023). This is because the data is available in several more grid sizes such as $1.5°$. In addition, the data can be readily downloaded to a local machine without waiting for further processing on the server. The guide for working with the dataset is available at (https://weatherbench2.readthedocs.io/en/latest/data-guide.html). In our specific case, we use the version labeled `1959-2023_01_10-6h-64x32_equiangular_conservative.zarr`. This dataset spans from year 1959 to 2023 with a grid size of $5.625°$ or resolution of 64 by 32 which spans from $-87.19°$ to $87.19°$ in latitude and $0.0°$ to $354.4°$ in longitude. The dataset is also resampled down to 6-hour intervals that starts at midnight UTC January 1959 1st and ends at 18:00 UTC January 10th 2023.

While the original data provided by ECMWF is available in NetCDF or GRIB, the WeatherBench2 version is provided using the Zarr format in a Google Cloud Storage Bucket. This setup allows the use of libraries that support the Zarr format to access the dataset remotely. The particular library we use is called *Xarray* version `2024.9.0`

(Hoyer & Hamman, 2017; Hoyer et al., 2024). Accessing the remote dataset is done as shown in figure 4.3. In order to ensure access is granted, the storage option token is set to anon so that the method uses the anonymous only public access mode of authentication ("GCSFS — GCSFs 2023.12.2post1+1.G8e500c6.Dirty Documentation," n.d.).

```
import xarray as xr
ds = xr.open_zarr(
    "DATASET_ADDRESS",
    storage_options = {"token": "anon"},
)
```

Figure 4.3: Example of using Xarray to access a publicly accessible Zarr dataset hosted on Google Cloud Storage Bucket.

The specific dataset resolution we use is 64 by 32 representing longitude and latitude respectively. The ERA5 reanalysis dataset is hosted at the following URL: gs://weatherbench2/datasets/era5/1959-2023_01_10-6h-64x32_equiangular_conservative.zarr. The corresponding climatology is hosted at gs://weatherbench2/datasets/era5-hourly-climatology/1990-2019_6h_64x32_equiangular_conservative.zarr. The climatology is the average for hours 0, 6, 12, and 18 for each day of the year from 1990 until 2019. This average helps in defining the expected state of the weather at specific points in time based on the historical averages for that time of day and year.

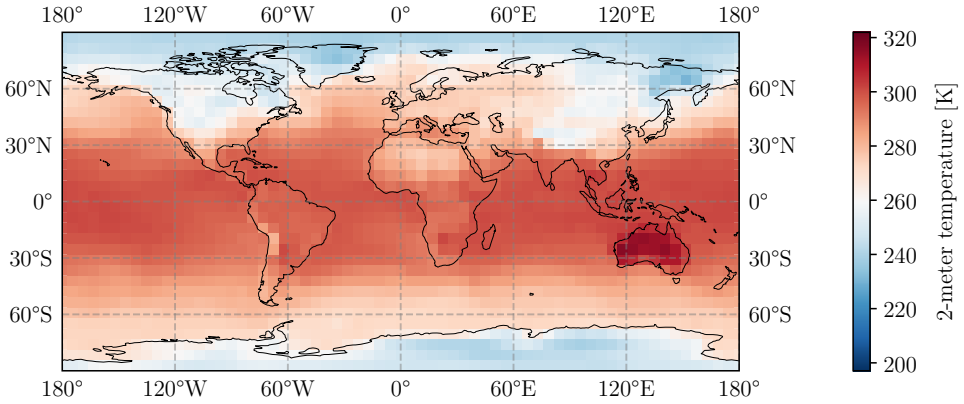
For the reanalysis dataset, we limit the time range to the years 1995-2013 for training and 2020-2022 for testing. The specific time ranges for training are chosen to balance the diversity within the data and capacity of the machine the processing will be done on. The testing set time range was chosen to be as close to those shown on the WeatherBench 2 website. The filter for time and variables is shown in figure 4.4. The reason for the different time ranges between training and testing is to reduce the risk of information leakage across the training and testing sets. This ensures that the testing scores we measure will be for the most part from the generalization that model learned during training. The filter is an array the length of the number of samples. The array values are all the datetime values that the year ranges are true for. This results in an array of boolean values. Using this boolean filter we can index the data so that we get the filtered data. Then, the filtered data is saved locally so that we can access it

in a more performant way. The encoding chunks may need to be removed in order to evade the Xarray library from mistakenly persisting the remote encoding after slicing data (Ian, n.d.).

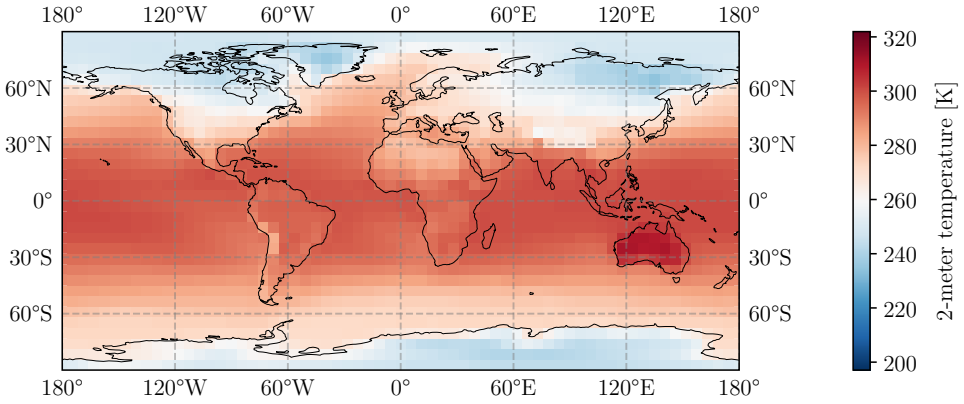
```
years_idx = ds['time.year']
data_idx = (years_idx > START_YEAR) & (years_idx < END_YEAR)
da_orig = ds['2m_temperature'][data_idx]
# save locally
da_orig.encoding.pop('chunks',None)
da_orig.to_zarr(DATA_FILENAME, mode='w')
# open local version
da = xr.open_zarr(DATA_FILENAME)
```

Figure 4.4: Example of filtering of Xarray dataset by year and variable. The data is saved and then loaded locally.

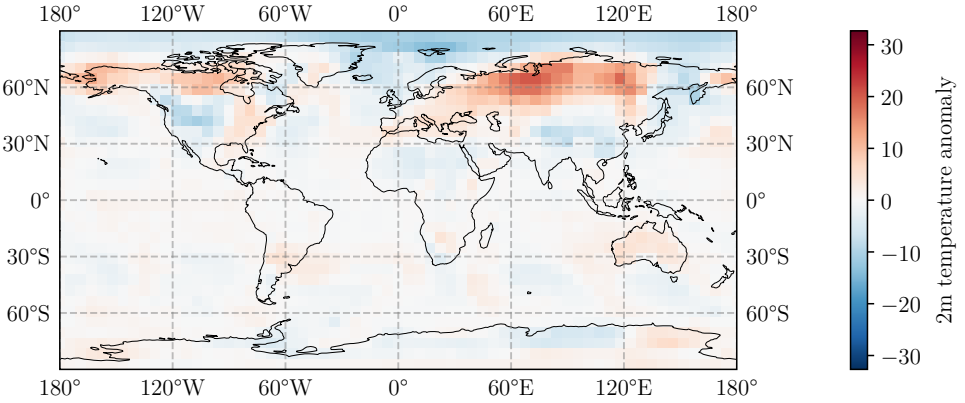
The dataset includes 62 variables with some variables also spanning 13 discrete vertical levels in the atmosphere. For our use, we will only be using the 2-meter temperature of the atmosphere. This is the air temperature 2 meters above the surface be it land, sea, or inland water. This variable is in units of kelvin (K). Vertically, the values of this variable is obtained by interpolating between the model values at the surface and the lowest model vertical level. An example of the data that was retrieved can be seen in figure 4.5. The data shown is in the equiangular projection. This means that towards the poles, the shapes become more distorted. While this is not the most ideal way to represent the data, using the data in this way adds to the challenge of how the model can adapt to the use of suboptimal configurations in the data. To compute the anomaly, we can simply subtract the current state in figure 4.5a by the climatology in figure 4.5b resulting in figure 4.5c.



(a) 2-meter temperature



(b) Climatology



(c) 2-meter temperature anomaly

Figure 4.5: Data retrieved: (a) 2-meter temperature at January 1, 1995 6:00 UTC. (b) 2-meter temperature climatology for the first day of the year at hour 6 UTC. (c) Anomaly for 2-meter temperature at January 1, 1995 6:00 UTC.

After filtering, our training set has the dimensions of 27 760 time steps, 64 longitude,

and 32 latitude. The testing set on the other hand has 4384 time steps and the same values for the other two dimensions. For uses, such as a validation set during hyperparameter optimization, one can use a subset of the training set. The dimensions of the climatology are 4 for hour (0, 6, 12,& 18), 366 for day of year, 64 for longitude, and 32 for latitude.

4.3 Design of Computational Model

In this section, we will discuss the design of the computational model that is proposed in this work. An illustration of the computational model is show in figure 4.6.

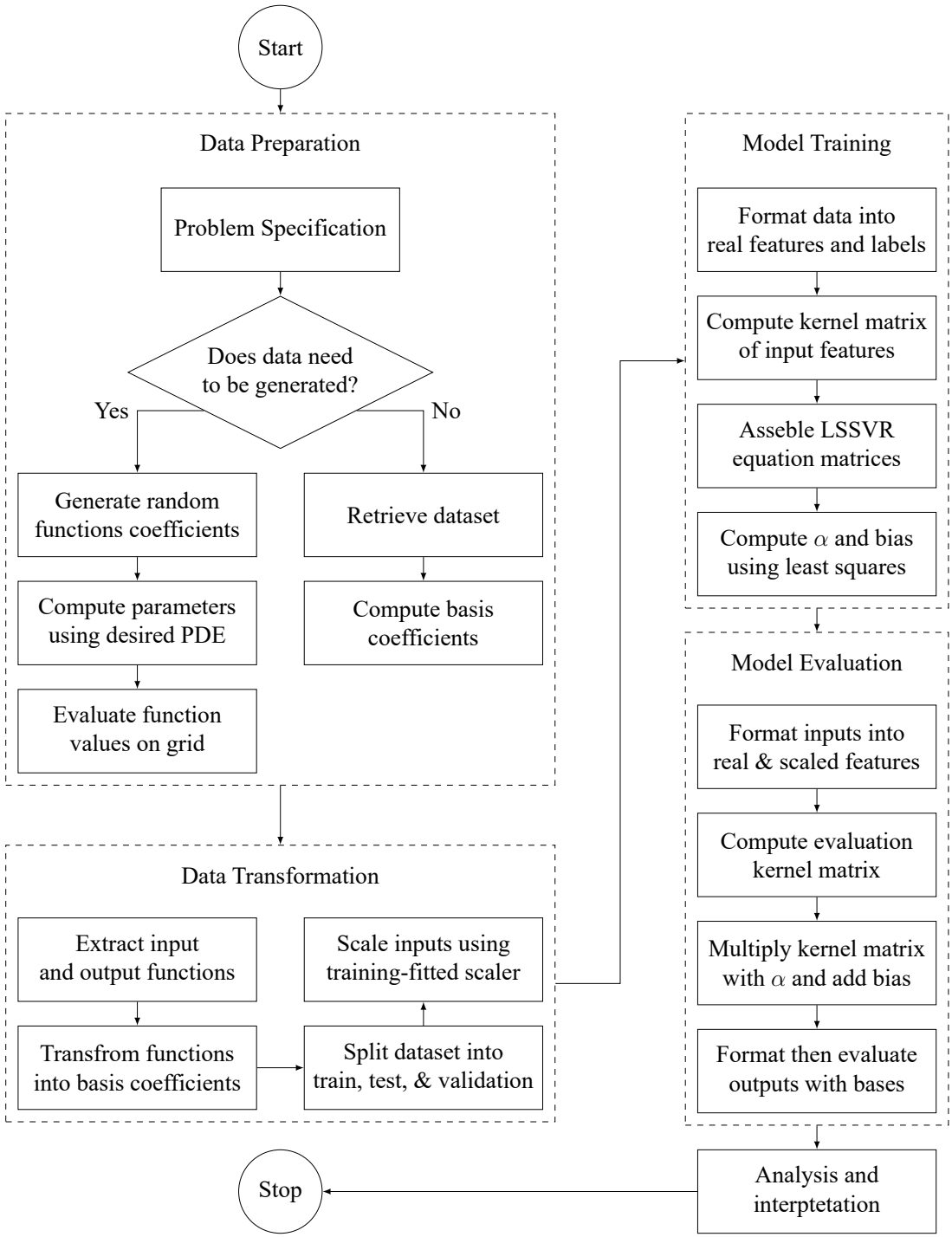


Figure 4.6: Computational Model of SpectralSVR.

The computational model is divided into four large phases. The processes within these phases are based on the broad outcomes of the phase. The phases themselves

are data preparation, data transformation, model training, and model evaluation. Each phase is composed of processes that together works toward the output of the larger process. At the end, a final process of analysis and interpretation is carried out.

4.3.1 Data Preparation

The data preparation phase starts off the computation by preparing the data that is to be used in this study. This phase starts with specifying the problem to be solved. This translates to the dataset that will be used to represent these problems. The first type of dataset is generated based on a known governing equation or PDE. Using said equation, the goal is to generate pairs of functions that are related to each other by the equation. In the next process, generally, the solution function is first generated randomly. This is done by generating random numbers to use as coefficients of a set of basis functions. Then parameters of the equation is determined, such as term coefficients. Using the generated solution and predetermined parameters, a forcing term can generally be computed to satisfy the equation. Augmentations such as noise can then be added in the final process which is evaluating the function values. These processes result in a generated dataset. To verify the generation method and any models trained on the generated data one can use exact analytical solutions of the governing equations.

The second type of dataset is retrieved from an external source. The data source is first identified, in our case it is the WeatherBench 2 collection of datasets. The specific dataset from the collection of different dataset versions is first determined. The dataset is available in the *Zarr* format. And then, we determine the filters for the portions of the dataset we are interested in. Once the dataset and filters are determined, the retrieval process starts by accessing the dataset using the *Xarray* library (Hoyer & Hamman, 2017; Hoyer et al., 2024). The filters are applied as in figure 4.4. Finally, the filtered version is stored locally. The local dataset is also stored in the *Zarr* format.

4.3.2 Data Transformation

The second phase of the computational model transforms the data that has been prepared into a form that can be used with the proposed model. In this study, there are two ways a dataset can be used. The first and simplest way is to predict the solution function from the forcing term for a given set of PDE parameters. In this case, the input function is the forcing term. In other words, the input function is the predictor or independent variable

that will be used to predict the solution function. As such, the output function is the solution function. The antiderivative dataset is used in this way. The derivative, which represents the acceleration function, is used to predict the velocity function. Which means that the input function is the acceleration function and the output function is the velocity function.

The second use of a dataset depends on whether it can be interpreted as a mapping of a previous function to the next function. In other words, a series of functions related to the previous function as a sum of the previous function and a perturbation. A commonly used form of this are time series. Generally, a time series can be represented as equation (4.12). This function describes the evolution of function f as a sum of some initial state f^0 and perturbation which is its rate of change $\frac{\mathrm{d}f}{\mathrm{d}t}$ at a point in time close to the current time multiplied by the time step δt. This is the first order Taylor polynomial. Adding more terms with higher order derivatives can be done to increase the accuracy. This describes the same process that is used in the finite difference method.

$$f^{t+1} = f^t + \delta t \times \frac{\mathrm{d}f}{\mathrm{d}t} + \dots \tag{4.12}$$

In this case the previous function and any other independent variables like the forcing term involved in the right-hand side of the equation is used to predict the difference between the future state and the current state. It would mean that each sample function in the data would need to at least have two dimensions. This is because one dimension represents the evolution of the function over time, while the other dimensions define the spatial domain where the function's values are computed at each time step. This condition is true for both the Burgers' equation dataset and the WeatherBench 2 collection of ERA5 datasets. First, the Burgers' equation dataset, while is also able to be used in the first way as a mapping between the forcing term and the solution, one of the traditional ways of solving it is using the method of lines. This entails solving the dimensions other that time with methods such as the finite differences method or the spectral method. The time dimension is then separately solved using FDM such as the form shown in equation (4.12) (Sadiku & Obiozor, 2000; Schiesser, 2012). This is the form that the Burgers' equation dataset will be used in. In this case, algebraic manipulation of equation (2.2) results in the time derivative of the solution as

in equation (4.13).

$$\frac{\partial u}{\partial t} = f - u\frac{\partial u}{\partial x} + \nu\frac{\partial^2 u}{\partial x^2} \tag{4.13}$$

The most important component of the right-hand side of the equation, the time derivative, is expressed in terms of the solution function, the forcing term and any other partial derivatives. Because of this, it means we can train the model to learn the relationship between the solution function combined with the forcing term and the time derivative. Putting this together with equation (4.12), it is possible to predict the solution function in a future time based on the current solution function and forcing term. The input functions in this case are both the value of the solution function and the forcing term at the current time step. The output function is the time derivative which is computed by rearranging equation (4.12) resulting in equation (4.14).

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \frac{f^{t+1} - f^t}{\delta t} \tag{4.14}$$

Perhaps in a more obvious manner, the same is also applied to the WeatherBench 2 collection of ERA5 datasets. The 2-meter temperature is a series of snapshots in time of the distribution of temperature all over the world at 2 meters above the surface. The input function in this case is the temperature distribution at the current time step. The output function is the time derivative of the temperature distribution computed using equation (4.14).

The input and output functions are then transformed into basis coefficients using the Fourier transform we discussed in section 2.4. Once the coefficients are computed, the input and output sample pairs are split into train, test, and validation subsets. For the first type of input and output function pair, since the samples are independent of each other, a random sampling of the dataset without replacement is done to select the samples for each subset. For the second type of input and output function pairs, if there are multiple samples of functions as in the Burgers' equation dataset, the same process with random sampling is followed. On the other hand, if only one function sample is available, such as the WeatherBench 2 dataset, an issue that arises is the fact that the samples are dependent on one another. This is because a previous time step influences the next time step. This is a problem for generalizing the learned model, since random sampling would lead to the model learning information in the future of the testing subset

(Kapoor & Narayanan, 2023; Kaufman et al., 2012). As such, the portioning of the dataset is done by dividing the sample into different sections of the function in time.

Finally, the data is scaled. This is done because of how support vector machines require scaling for inputs in order to perform well with certain hyperparameters such as the RBF kernel (Ben-Hur & Weston, 2010). For our case we use standard scaling based on the inputs of the training subset. This approach transforms the dataset by manipulating the distribution characteristics of the dataset (Ahsan et al., 2021). Specifically, the standard deviation σ and mean x_0 of the dataset values x are processed such that the resulting values have a mean of 0 and standard deviation of 1. The function shown in equation (4.15) represents how each sample is processed with respect to the standard deviation σ and mean x_0 of each feature.

$$z(x; \sigma, x_0) = \frac{x - x_0}{\sigma} \tag{4.15}$$

The scaling is important for the RBF kernel specifically due to the fact that the kernel utilize a scale parameter. This parameter affects how scale of values influence the kernel. Another and perhaps more general benefit of scaling is the uniformity it enforces across features. This is useful because all feature weigh the outcome equally. In our use case, this equality is a reflection of what the data actually represents. Each feature which is a discretized value or its coefficient representation has equal value to any other feature. No point or coefficient is any more or less value compared to another point or coefficient.

The scaling is applied to input functions only because scaling of the outputs does not affect the performance of the model (Ben-Hur & Weston, 2010). To also prevent information leakage across training and testing sets, the mean and standard deviation of each feature is computed using the training subset only. Once the parameter values are obtained, all input function samples in the dataset are transformed into their scaled versions. This means that the distribution for the testing and validation sets may be outside the target standard deviation and mean.

4.3.3 Model Training

Once the data has been preprocessed, the model is trained using pairs of training features and labels. The training phase is separated into four processes. First, the preprocessed

input and output function representation need to be real numbers. This is because the LSSVR model was made to work with real numbers. Any complex values such as Fourier coefficients are represented as flattened pairs of the real and imaginary components of complex values.

The training can proceed to solve equation (2.23). As the original LSSVR equation was made for regression of a single value of $\mathbf{y}$, we constructed an extended the original equation in order to train multiple regression models to predict each coefficient in an output function sample. The assumption of this extension is that the inputs and model hyperparameters such as kernels and the regularization constant C are the same across individual regressors. Therefore, it follows that the solution vector and target vector can both be transformed into their matrix form with each column representing the individual regressor which results in equation (4.16). This vectorized version is equal to m separate instances of equation (2.23). In essence, instead of a vector of scalar sample output values and a vector of model parameters, the extended equation consists of a matrix of rows of vector output values and a matrix of model parameters with each column representing the separate model for each output and row representing the Lagrange multipliers for each sample. The procedure to solve equation (4.16) adjusts algorithm 1 for the extra dimension spanning the multiple outputs of size m.

$$\begin{bmatrix} 0 & \mathbf{1}_n^\intercal \\ \mathbf{1}_n & \Omega + \frac{I}{C} \end{bmatrix} \begin{bmatrix} b_0 & \cdots & b_m \\ \alpha_0 & \cdots & \alpha_m \end{bmatrix} = \begin{bmatrix} 0 & \cdots & 0 \\ \mathbf{y}_0 & \cdots & \mathbf{y}_m \end{bmatrix} \tag{4.16}$$

Second, as the training process of an LSSVR described in requires constructing the first matrix on the left-hand side and the matrix on the right-hand side, each component of the matrices themselves will also need to be constructed. The kernel matrix Ω is first constructed by computing the kernel values between each sample input with every other sample input. If the training set has n samples, then the resulting kernel matrix would be of size n by n. Third, using the inputs $\mathbf{X}$, outputs $\mathbf{y}$, predetermined regularization constant C, and the obtained kernel matrix Ω, equation (4.16) is assembled. Finally, the matrix of learned parameters α and $\mathbf{b}$, which are the Lagrange multipliers and bias respectively, are solved for using a linear equation solver such as a matrix Moore-Penrose inverse or linear least squares solver.

For our study, we choose to not optimize the hyperparameters to focus on the broad

performance with default hyperparameter values. The scaling parameter of the RBF kernel is computed from the training features. This is computed as the square root of the sum of feature variances. The regularization parameter used will be a default value of 1. Using the same values allow for easier comparisons for comparable results.

4.3.4 Model Evaluation

The fourth phase of the computational model is the evaluation of the learned model. This phase starts with formatting the inputs. If the inputs aren't scaled yet, then it is scaled using the scaling function fitted on the training set. This is the case for new data such as new measurements or an analytical solution of the learned PDE. Once the inputs that is to be evaluated is scaled in the same way as the training inputs, the model can predict the corresponding outputs.

The procedure to compute the predicted outputs follows algorithm 2. This leads to the second process which is computing the kernel matrix between the training input features $\mathbf{X}$ and the evaluation input features $\mathbf{U}$. The matrix is constructed as the kernel value between each evaluation sample feature and every training sample feature. For p evaluation samples and n training samples, this results in a kernel matrix Ω with a shape of p rows and n columns.

In the third process, the constructed kernel matrix is then multiplied with the matrix of learned Lagrange multipliers α. The multiplication is the matrix multiplication process. Then, the bias vector is added to each row to produce the final predicted outputs $\mathbf{v}$. This process is represented by equation (4.17).

$$\mathbf{v} = \Omega\alpha + \mathbf{1b} \tag{4.17}$$

The fourth process is concerned with the fact that the predicted output $\mathbf{v}$ is a matrix of real numbers. The LSSVR output needs to be formatted before it can be used in the proposed model. This is done by converting the outputs into the same format as the training labels. As we use the Fourier basis, the model output is converted back into a matrix of complex numbers using the inverse of the first process during training. Finally, these predicted Fourier basis coefficients can be used to evaluate the predicted functions themselves.

4.4 Implementation of Computational Model

This section discusses the implementation of modeling and approximation of PDEs using LSSVR. Using the computational model design in section 4.3, the computational model is implemented as a library using the Python programming language. The development process itself is based on the iterative development model as mentioned in section 3.4. In this section, the output iterative processes are consolidated and explained together.

4.4.1 Analysis

The aim of any program that is developed in this study is to model the relationship between functions defined by a partial differential equation or other operators and governing equations. Since the potential subject of modeling can be wide-ranging, the software product that will be developed is a library. This is done so that the tools and proposed model that is developed will be able to be used with a variety of PDEs. With this in mind, the software must also be performant and compatible with the larger machine learning and scientific computing community.

To demonstrate the use and capability of the software, three case studies are chosen to represent the variety of PDEs. The first case study is the antiderivative equation. This simple case is chosen as a proof of concept to show that the proposed model works in the most simple case. The second case study, which is the Burgers' equation, adds a level of complexity because of the nonlinearity involved with the solution function. The third case study, which is the ERA5 weather dataset, was chosen to show how the model performs on data with real world observations incorporated into it. This dataset also presents a challenge because of the uncertainties involved with weather simulation (Herrera et al., 2017). These datasets are discussed in sections 4.1 and 4.2.

In addition, the software product is developed within certain limitations. The limitations are:

1. The operating and implementation language of the library is Python.
2. The library is to be used in user code
3. The library depends on other libraries including the following core dependencies PyTorch, NumPy, Xarray, Zarr, Pandas, and tqdm.

4.4.2 Design

In this subsection, the architectural design of the software product is explained. The architecture consists of modules with similar concerns. This was chosen because of the intuition built by the use of other libraries with a similar architecture. The broad aim of modeling PDEs with LSSVR and basis functions coefficients can be separated into smaller concerns of data, model, and basis functions. Each of these concerns are addressed by a corresponding module. The explanation of each module are as follows:

1. Utilities

   This module provides basic tools that are used across the entire library. The utilities this module needs to provide include scaling, conversion between real and complex representations of matrices, resizing coefficient tensors, and adaptors for or the traditional ordinary differential equation solvers. Scaling functionality is used for inputs into the support vector machines. And as previously mentioned, since LSSVR only works with real numbers, the tools for converting between complex and their real number representations is also necessary. Another useful tool for working with basis coefficients is a way to resize the coefficients in terms of how many basis functions is desired. An example of the usage is downsizing Fourier coefficients from 12 to 8 modes. The last utility mentioned here provides the ability to solve ordinary differential equations. This utility is useful for generating data for certain types of PDEs such as the Burgers' equation.

2. Basis Functions

   As the proposed model will learn the relationships defined by PDEs within the spectral domain, a large proportion of the computation will also concern the use of the basis functions themselves. The module will define the functionality and data structure that is expected to be used when working with basis functions in this library. First and foremost, the central function of this module is to store and facilitate the connection between the coefficients in the spectral domain and their values in the physical domain. This is dependent on the basis functions used. For the Fourier basis that we use in this study, this means that the transformations are the discrete Fourier transform and the discrete inverse Fourier transform. Other than this core functionality, information on the characteristics of the coefficients such as the number of modes, in other words basis functions, should also be provided by this module. In addition, other functionality included in this module

are visualization of coefficients in physical space, generation of random functions, time dependent & complex/real coefficients, noise perturbation of functions, and differential operators.

3. Model

   The proposed SpectralSVR model is implemented in this module. As the proposed model essentially wraps a support vector regression model in such a way that it can process coefficients, this module will house the SpectralSVR wrapper itself and the LSSVR variant of support vector regression that is used in this study. The SpectralSVR wrapper must provide three core pieces of functionality. First, it must translate complex input and output samples into their real representations. This applies to both training and evaluation. Second, in addition to predicting the coefficients during evaluation, the wrapper should also be able to predict the function values themselves for any given set of point coordinates. This is important for quick evaluations of a predicted function at an arbitrary point. The third functionality is inverse input function search for a given output. This is useful for inverse problems where the parameters are unknown, but the solution is partially known. Aside from the core functionality, a complementary functionality is testing for the learned model.

   The model module also implements the LSSVR model. This implementation provides two main functionalities. The first is training the LSSVR model with some given model hyperparameters such as the regularization constant and kernel. The second functionality is the evaluation of the learned model. Both functionalities allow for multiple output regression. This is required to learn the mappings for many coefficients at once. Other functionality provided by the implementation include interpretability functions and model serialization for saving and loading the learned model to disk.

4. Problems

   This module provides the functionality concerned with generating data. Two out of the three datasets in this study are generated with this module. In general this module provides two pieces of functionality. The first is generation of the data itself. This functionality generates at least two sets of functions that are related to each other by the chosen PDE. The generation is based on at least two parameters which are the number of samples to generate and the number of modes or basis

functions to generate the coefficients for. The second functionality provided by this module is the residual based on the chosen PDE. This is useful to verify if any set of functions comply with the PDE. For example, the residual may be used to verify the generated data. Another use is to grade the predictions by the learned model.

The design presented here is the product of multiple iterations that started with building the core model and then the basis functions and finally the data. Throughout the iterations, the utilities were also added to alleviate the need for general tools.

4.4.3 Implementation

The design in section 4.4.2 is implemented within the same iterative loop. As such, the feedback from tests is immediate, and any fixes can be implemented quickly. The implementation is done using the Python language and a number of Python libraries. To facilitate the management of dependencies, the implementation makes use of the package management tool Poetry (Eustace, 2024). In addition, the Python version is managed using the tool PyEnv. The implementation process starts with installation of the required dependencies. The dependencies are specified in the `pyproject.toml` as shown in figure 4.7.

```
...
[tool.poetry.dependencies]
python = "^3.10.6"
numpy = "^1.24.0"
tqdm = "^4.65.0"
jaxtyping = "^0.2.20"
pandas = "^2.2.2"
torchmetrics = "^1.4.2"
scikit-learn = "^1.5.0"
matplotlib = "^3.9.0"
torch = {version = "^2.4.1+cu124", source = "pytorch-gpu"}
torchvision = {version = "^0.19.1+cu124", source = "pytorch-gpu"}
torchaudio = {version = "^2.4.1+cu124", source = "pytorch-gpu"}
ray = {extras = ["tune"], version = "^2.37.0"}
hyperopt = "^0.2.7"
ipywidgets = "^8.1.5"
torchdiffeq = "^0.2.4"
xarray = "^2024.10.0"
zarr = "^2.18.3"
gcsfs = "^2024.10.0"

[[tool.poetry.source]]
name = "pytorch-gpu"
url = "https://download.pytorch.org/whl/cu124"
priority = "explicit"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Figure 4.7: Contents of pyproject.toml configuration file that define the dependencies using Poetry.

After ensuring that the correct version of python is being used with pyenv, the dependencies specified in the `pyproject.toml` file can be installed using the shell command `poetry install`. At the end of each iteration, the dependencies are subject to change depending on whether any new ones are needed or old dependencies can be removed.

In the following, we will discuss the implementation of each module and how the phases of computational model relates to each implemented functionality. The base data structure that is used in the implementation is the PyTorch Tensor (Ansel et al., 2024). Tensors are a generalization of scalars and array-like structures such as vectors and matrices. This is essential since many computations that is done in the proposed

model are based on matrices and higher dimensional arrays. PyTorch Tensors allow the elements to be of different data types such as floating point numbers of different precision or even complex numbers. In addition to basic operations for working with the Tensors such as matrix multiplications, PyTorch also provides many algorithms for use when working with Tensors such as linear equation solvers or the Fast Fourier Transform. Another reason for choosing to use PyTorch is the ability to parallelize computation using specialized processors such as GPUs. Lastly, PyTorch is widely used in the machine learning community which reflects on its reliability, wealth of community knowledge and support, and active development.

Utilities

The utilities module provide common tools that are used when working with the developed software. First, because the basis function used in this study is the Fourier basis, a large proportion of concern is the storage of Fourier basis coefficients. The Fourier series representation that is used in this study is the complex representation as show in equation (1.2). However, as previously mentioned, least squares support vector machines is constructed to work with real numbers. In order for LSSVR to learn the relationship between complex numbers, a representation of complex numbers in terms of real numbers is necessary. Since sample features and labels for LSSVR need to be flattened into one dimension which result in a matrix with rows of samples, the representation of complex numbers using real numbers can be done after features and labels have been flattened. An example of such a matrix of two samples with four features each can be seen in equation (4.18). To represent the complex number features as real numbers, once can simply split a single complex number into two real numbers. When this is done to the entire matrix of complex numbers, the result is equation (4.19).

$$\begin{bmatrix} -1.03+0i & -0.52+0.39i & -0.54+0i & -0.52-0.39i \\ -0.85+0i & -0.39+0.19i & 0.87+0i & -0.39-0.19i \end{bmatrix} \tag{4.18}$$

$$\begin{bmatrix} -1.03 & 0 & -0.52 & 0.39 & -0.54 & 0 & -0.52 & -0.39 \\ -0.85 & 0 & -0.39 & 0.19 & 0.87 & 0 & -0.39 & -0.19 \end{bmatrix} \tag{4.19}$$

There are several ways to do this in PyTorch, such as extracting the real and imaginary element components into separate matrices and then interleaving the columns to create the real representation. However, a more performant version can be made by using

built-in functions of the library that directly allows views of complex valued tensors as real tensors and vice versa. The `view_as_real` and `view_as_complex` functions allow the conversion between real and complex representations by representing complex valued tensors by adding a dimension of size two for each component of complex numbers. The resulting implementation can be seen in figure 4.8.

```
def to_real_coeff(x: torch.Tensor) -> torch.Tensor:
    return torch.view_as_real(x).flatten(-2)

def to_complex_coeff(x: torch.Tensor) -> torch.Tensor:
  return torch.view_as_complex(
    x.reshape((*x.shape[:-1], -1, 2))
  )
```

Figure 4.8: Utility function to convert between complex matrices and the real representations.

Another important utility used on inputs for the proposed SpectralSVR is a scaling function. This utility has been implemented in a variety of ways and using different tools. However, as the standard scaling is not provided by PyTorch, we have also implemented our own version. The scaling function accepts PyTorch tensors or tuples of tensors of real or complex elements. First, the scaling function is implemented as a class that is instantiated. The scaling function is then fitted onto the reference data to obtain the standard deviation and mean of each feature. This information is stored as properties of the class. Any complex tensors are converted into their real representations using the functions in figure 4.8. Once the scaling function instance is fitted, any set of tensors with the same number of features (columns) and element type (complex or real) can be roughly transformed into a standard deviation of one and mean of zero for every feature. This is done by subtracting the fitted mean of each feature from the elements in the column and then dividing with the feature's standard deviation. The implementation also provides a method to retrieve a scaling function fitted on a subset of the original tuple of tensors. Serialization for saving to and loading from disk is also implemented. This is done simply by using the PyTorch save and load functions on the scaling function instance itself. For some use cases, an inverse scaling operation to the original standard deviation and mean is also essential. The implementation is simply the reverse of the transformation process where the elements are multiplied by the standard deviation and the mean is the added.

Another group of utility in the module is concerned with managing the number of modes a tensor of coefficients has. First, a function called `resize_modes` accepts a tensor of coefficients and the target number of modes in each dimension. The first dimension is always assumed to represent the different samples. Because of this the target modes will only affect the dimensions after the first. The wave numbers are also assumed to be symmetric about zero such that the highest absolute values of the wave number is in the middle of each dimension. The wave number decreases symmetrically as you move towards the edges of the dimension and reaching the lowest absolute wave number at the edges. This is done to ensure compatibility with the way existing libraries such as PyTorch works with tensors of Fourier coefficients. Because of this, shrinking a tensor of coefficients along a dimension means removing the center of the tensor along that dimension. And expanding to a target number of modes that is larger means that the tensor is padded with zeros in the center along the desired dimension. In addition, the remaining coefficients may be rescaled in order to balance the effect of adding or removing coefficients. The implementation is presented in figures 4.9 and 4.10.

```
for dim, (target_mode, current_mode) in enumerate(
    zip(target_modes, current_modes), 1
):
    device = x.device
    start_range = torch.tensor(
        range((target_mode - 1) // 2 + 1), dtype=torch.int
    ).to(device=device)
    end_range = torch.tensor(
        range(current_mode - target_mode // 2,
          current_mode
        ),
        dtype=torch.int,
    ).to(device=device)

    x_resized = torch.concat(
        (
            x_resized.index_select(dim, start_range),
            x_resized.index_select(dim, end_range),
        ),
        dim,
    )
```

Figure 4.9: Example of shrinking a tensor of samples' coefficients to a target number of modes.

```
for dim, (target_mode, current_mode) in enumerate(
    zip(target_modes, current_modes), 1
):
    device = x.device
    start_range = torch.tensor(
        range((current_mode - 1) // 2 + 1), dtype=torch.int
    ).to(device=device)
    # make sure that end range is empty if
    # the coefficient is only size 1
    end_range = torch.tensor(
        range(current_mode // 2, current_mode)
        if current_mode > 1
        else range(0),
        dtype=torch.int,
    ).to(device=device)
    padding_size = target_mode - current_mode
    modes = list(x_resized.shape)
    modes[dim] = padding_size
    padding = torch.zeros(modes).to(x_resized)

    x_resized = torch.concat(
        (
            x_resized.index_select(dim, start_range),
            padding,
            x_resized.index_select(dim, end_range),
        ),
        dim,
    )
```

Figure 4.10: Example of expanding a tensor of samples' coefficients to a target number of modes.

Next, the second way to manage the size of a tensor along a dimension is to interpolate the current values. The implementation uses simple linear interpolation. This is especially useful for situations such as coefficients that are themselves functions of time. This is a common application such as when solving the initial condition problem of a function in space and time. One might represent the function as a Fourier series with basis as functions of space and coefficients as functions of time. In this case, the coefficients for time values that are not available can be interpolated. The implementation of this is shown in figure 4.11.

```
    index_floor = index_float.floor().to(torch.int)
    index_ceil = index_float.ceil().to(torch.int)
    x_ceil = x.index_select(dim, index_ceil)
    x_floor = x.index_select(dim, index_floor)
    # interpolate coefficients
    index_shape = [1 for _ in range(x_floor.ndim)]
    index_shape[1] = -1
    index_scaler = (
        ((index_float - index_floor) / (
          index_ceil - index_floor
        ))
        .reshape(index_shape)
        .nan_to_num()
    )
    # ynt + scaler * (ynt1 - ynt)
    # (1 - scaler) * ynt + scaler * ynt1
    x_interp = torch.lerp(x_floor, x_ceil, index_scaler.to(x))
```

Figure 4.11: Example of interpolating for indices (floating point indices) between the available ones (integer indices).

Finally, this module also provides the types and reexports implementations of ordinary differential equation solvers from the torchdiffeq library (R. T. Q. Chen, 2021). The implementation can be seen in figure 4.12.

```
RHSFuncType = Callable[[torch.Tensor, torch.Tensor],
  torch.Tensor
]
SolverSignatureType = Callable[[
        RHSFuncType, torch.Tensor, torch.Tensor,
    ],
    torch.Tensor
]
MixedRHSFuncType = Callable[[
        torch.Tensor, torch.Tensor, torch.Tensor,
    ],
    torch.Tensor,
]
MixedSolverSignatureType = Callable[
    [MixedRHSFuncType, torch.Tensor, torch.Tensor],
    torch.Tensor,
]

def euler_solver(
    rhs_func: RHSFuncType,
    y0: torch.Tensor,
    t: torch.Tensor,
):
    solution = torch.zeros((len(t), *y0.shape)).to(y0)

    j = 1
    solution[j - 1] = y0
    for t0, t1 in zip(t[:-1], t[1:]):
        dt = t1 - t0
        y = solution[j - 1]
        solution[j] = y + dt * rhs_func(t0, y)
        assert (
            solution.isnan().sum() == 0
        ), f"solver encountered nan at timestep {j} (t={t0})"
        j = j + 1
    return solution

implicit_adams_solver: SolverSignatureType = partial(
    odeint, method="implicit_adams", options={"max_iters": 4}
)  # type: ignore

lsoda_solver: SolverSignatureType = partial(
    odeint, method="scipy_solver", options={"solver": "LSODA"}
)  # type: ignore
```

Figure 4.12: Implementation of ODE solvers and the types.

Basis Functions

This module implements two classes. The first is the base Basis class. The second is a subclass implementing the Fourier basis specifically. These classes provide the functionality needed to have an easier time when working with basis functions and their coefficients. First, the core function of these classes is to provide a way to transition between the spectral domain and the physical domain. This is implemented in the base class by storing the coefficients of sample functions as a class property. The coefficients are assumed to be at least of one function.

```
...
    @staticmethod
    def transform(
        f: torch.Tensor,
        res: TransformResType | None = None,
        periodic: bool = True,
        periods: PeriodsInputType = None,
        allow_fft: bool = True,
    ) -> torch.Tensor:
        if not torch.is_complex(f):
            f = f * (1 + 0j)
        res = transformResType_to_tuple(res, tuple(f.shape[1:]))
        periods = periodsInputType_to_tuple(periods, f.shape[1:])
        # perform 1d transform over every dimension
        F = f
        for cdim in range(1, ndims):
            F = FourierBasis._ndim_transform(
                F,
                dim=cdim,
                func="forward",
                res=res[cdim - 1],
                periodic=periodic,
                period=periods[cdim - 1],
                allow_fft=allow_fft,
            )

        return F
...
```

Figure 4.13: Implementation of forward Fourier transform.

The transitions between the coefficients and functions values are implemented as a function to transform function values to coefficients and an inverse transform function to compute function values from coefficients. The base class enforces this by

defining abstract methods which any subclass will need to implement. The FourierBasis subclass then implements the transform and inverse transform functions specific for transitioning between function values and Fourier basis coefficients. As before, the implementation assumes that the first dimension indexes the samples and the rest represent the coefficient wave numbers. This means that the implementation must allow for multidimensional functions. For the Fourier transform and inverse transform, the multidimensional version is relatively simple to implement. The multidimensional Fourier transform essentially performs the one dimensional Fourier transform along one dimension. And then, using the result to perform the one dimensional Fourier transform along the next dimension. This process is repeated until the last dimension. The implementation of the forward transform is presented in figure 4.13. The inverse transform simply changes the `func` parameter to "inverse".

The Fourier transform along each dimension is done by flattening every other dimension into the sample dimension. This essentially means that the function values are "sliced" along the dimension currently being transformed, and that each "slice" is its own "sample". Mathematically, this is simply due to the matrix multiplication of basis functions along one dimension with every other dimension. The implementation of this is shown in figure 4.14.

```
...
    @staticmethod
    def _ndim_transform(
        f: torch.Tensor,
        dim: int,
        func: Literal["forward", "inverse"],
        res: slice,
        periodic: bool,
        period: float,
        allow_fft: bool,
    ) -> torch.Tensor:
        # flatten so that each extra dimension is
        # treated as a separate "sample"
        # move dimension to transform to the end
        # so that it can stay intact after f is flatened
        f_transposed = f.moveaxis(dim, -1)
        # flatten so that the last dimension is intact
        f_flatened = f_transposed.flatten(0, -2)

        F_flattened = FourierBasis._raw_transform(
            f_flatened,
            func=func,
            res=res,
            periodic=periodic,
            period=period,
            allow_fft=allow_fft,
        )
        # unflatten so that the correct shape is returned
        F_transposed = F_flattened.reshape(
          (*f_transposed.shape[:-1], res.step)
        )
        F = F_transposed.moveaxis(-1, dim)

        return F
...
```

Figure 4.14: Implementation of n-dimensional Fourier transform for a specific dimension.

```
...
    @staticmethod
    def _raw_transform(
        f: torch.Tensor,
        func: Literal["forward", "inverse"],
        res: slice,
        periodic: bool,
        period: float,
        allow_fft: bool,
    ) -> torch.Tensor:
        match func:
            case "forward":
                sign = -1
            case "inverse":
                sign = 1
        mode = f.shape[1]
        domain_starts_at_0 = res.start == 0
        domain_end_equal_to_period = res.stop == period
        can_use_fft = (
            domain_starts_at_0
            and domain_end_equal_to_period
            and periodic
            and allow_fft
        )
        if can_use_fft:
            if func == "forward":
                F = torch.fft.fft(
                  f, dim=1, n=res.step, norm="backward"
                )
            elif func == "inverse":
                F = torch.fft.ifft(
                  f, dim=1, n=res.step, norm="forward"
                )
        else:
            if periodic:
                n = res.start + torch.arange(res.step).to(f)
                n = n / res.step * period
            else:
                n = torch.linspace(
                    res.start, res.stop, res.step
                ).to(f)
            e = FourierBasis.fn(
                n.view(-1, 1),
                mode,
                periods=period,
                constant=sign * 2j * torch.pi,
            )

            F = torch.mm(f, e.T)
        return F
...
```

Figure 4.15: Implementation of the one dimensional Fourier Transform.

The one dimensional Fourier transform itself is implemented as a raw transform function. This function accepts input of a matrix of values to transform, a parameter to indicate the type of transformation (inverse transform or forward transform), the evaluation boundary and number of grid points, whether the evaluation should assume the function is periodic, the size of the function domain, and if the Fast Fourier Transform is allowed to be used. The evaluation boundaries, actual function domain size, whether the function can be assumed to be periodic, and if FFTs is allowed to be used together determine the Discrete Fourier Transform algorithm that is used. The FFT algorithm is used if the following are all true: the evaluation starts at zero and ends with the same value as the period, the function can be assumed to be periodic, and FFT is allowed. Otherwise, the transform is computed naively using equations (2.11) and (2.12). The naive approach allows for evaluating function values of coefficients at various grid sizes and evaluation boundaries. This is useful for being able to evaluate at resolutions other than that of the original discretized function or coefficients. One use for this is plotting the function at different parts or resolutions. The relationship with the physical domain means that the boundaries of the domain is important. This is because the function values that will be used may only be valid within certain boundaries. Because of this, the information of physical domain bounds must also be stored. This is done by storing the span of the function in each dimension. The implementation we have gone with doesn't store any other information than the span in each dimension. Because of this, the information would need to be stored outside the Basis class instance and any outputs or operations with the classes will need to take this into account. The implementation of the one dimensional Fourier transform is presented in figure 4.15.

The secondary set of functionality provided by the basis classes are mathematical operators on the functions the coefficients represent. There are four operators implemented, which are the addition, subtraction, derivative, and antiderivative operators. These four operations can be further categorized into arithmetic and calculus operations. The arithmetic operations, which are addition and subtraction, are implemented by following how the actual mathematical operations would be carried out on two functions which are Fourier series as shown in equation (4.20). For subtraction, the plus sign is simply replaced with the minus sign.

$$\sum_k \hat{u}_k e^{2\pi ikx} + \sum_k \hat{f}_k e^{2\pi ikx} = \sum_k \left(\hat{u}_k + \hat{f}_k\right) e^{2\pi ikx} \tag{4.20}$$

The implementation leverages the operator overloading in Python classes. In addition, the overload should be done to the `__add__` function. This function adds the current instance to another object which is the other basis instance for our use case. We also want the operation to return an independent instance without any references to the previous instances. To do this, all properties are copied into a new instance. This is then used to perform the arithmetic operation. The implementation in figure 4.16 demonstrates the addition operation. For subtraction, a similar implementation is done using the `__sub__` function and switching the plus sign with the minus sign.

```
...
    def __add__(self, other: Self):
        if isinstance(other, self.__class__):
            if other.coeff is None:
                return self.copy()
            elif self.coeff is None:
                return other.copy()
            else:
                result = self.resize_modes(other)
                result.coeff = result.coeff + other.coeff
                return result
        else:
            raise TypeError(
                f"unsupported operand type(s) for +: '{
                    self.__class__
                }' and '{type(other)}'"
            )
...
```

Figure 4.16: Implementation of basis addition function.

The calculus operations for Fourier basis take advantage of the properties of the Fourier series. As discussed in section 2.4, derivatives and integration become multiplication and division in the spectral space of Fourier series. Our implementation takes advantage of this as shown in figure 4.17. Similar to how subtraction is just a modification of the addition operation, the integral or antiderivative operator is also implemented simply by changing the operation with the term multiplier from multiplication to division.

```
...
    def grad(self, dim: int = 0, ord: int = 1) -> Self:
        copy = self.copy()
        if dim == 0 and self.time_dependent:
            # time dependent use finite differences
            dt = self.periods[0] / (self.time_size - 1)
            coeff = copy.coeff
            for o in range(ord):
                coeff = torch.gradient(coeff, spacing=dt, dim=1)[0]
            copy.coeff = coeff
        else:
            if self.time_dependent:
                # disregard time dimension
                dim = dim - 1
            k = copy.wave_number(copy.modes[dim])
            multiplier_dims = [1 for _ in range(copy.ndim)]
            multiplier_dims[dim] = copy.modes[dim]
            if self.time_dependent:
                multiplier_dims = (1, *multiplier_dims)
            multiplier = (
                2
                * torch.pi
                * 1j
                * k.reshape(multiplier_dims).to(copy.coeff)
                / self.periods[dim]
            )
            multiplier = multiplier.pow(ord)
            coeff = copy.coeff.mul(multiplier)
            coeff[:, ..., 0] = torch.tensor(0 + 0j)
            copy.coeff = coeff
        return copy
...
```

Figure 4.17: Implementation of basis grad function.

The last group of functionality implemented in the Basis and FourierBasis classes provides convenience and often used procedures when working with basis functions. First, random coefficient generation for the Fourier basis is implemented. This is done as discussed in step 3 of the data generation process discussed in section 4.1. The implemented function shown in figure 4.18 is then wrapped in another function that creates a new FourierBasis instance with the generated coefficients.

```
...
    @classmethod
    def generate_coeff(
        cls,
        n: int,
        modes: int | tuple[int, ...],
        generator: torch.Generator | None = None,
        random_func: Callable[..., torch.Tensor] = torch.randn,
        complex_funcs: bool = False,
        scale: bool = True,
    ) -> torch.Tensor:
        if isinstance(modes, int):
            modes = (modes,)
        n_modes = len(modes)
        assert n_modes > 0, "modes should have at least one element"
        random_func = partial(random_func, generator=generator)
        if complex_funcs:
            coeff = random_func((n, *modes), dtype=torch.complex64)
        else:
            coeff = random_func((n, *modes), dtype=torch.complex64)

            vals = cls.inv_transform(coeff)
            coeff = cls.transform(vals.real + 0j)
        if scale:
            scaler = torch.tensor(modes).sum() * 0.2
            coeff = coeff.mul(scaler)
        return coeff
...
```

Figure 4.18: Implementation of Fourier Basis coefficient generation function.

Other convenience functionality also provided include plotting, function value perturbation, and function value evaluation based on domain span information that has been stored in the class instance. The function value evaluation is especially useful for experimenting and for the plotting functionality itself. This is because, the span was given to the constructor and does not need to be composed again with the inverse transform function.

Model

The SpectralSVR model is implemented in this module. There are two components in this module. First, there is the implementation of least-squares support vector regression (LSSVR) using PyTorch. This implementation is a reimplementation and modification of an LSSVR implementation using scikit-learn by Florêncio et al.

(2020). The modifications are mainly concerned with using PyTorch in place of scikit-learn. Other modifications are concerned with the performance side of things, which are computing the kernel matrix in batches to avoid memory spikes and the ability to compute on hardware accelerators supported by PyTorch such as GPUs. The LSSVR implementation starts with the constructor which accepts model hyperparameters which include the regularization parameter, kernel function and any kernel parameters, verbosity level, kernel matrix computation batch size, and device to perform computations on. The received parameters are then stored as properties of the class instance. For the kernel specifically, it will be initialized with the correct default parameters when the fitting function is called. This is because if the kernel of choice is the radial basis function, the default scaling parameter needs to be computed from the training features as the square root of the sum of feature variance values.

The optimization function that fits the model to the training data solves equation (2.23). This function is shown in figure 4.19. The first half of the function sets up the left and right hand side matrices. The left hand side matrix is constructed in part with the kernel matrix. The kernel function used is batched to reduce the peak memory footprint. These matrices are then used in the second half to compute the solution using the least squares solver function from PyTorch called lstq. Finally, the result is split into the biases and Lagrange multipliers which are then returned as a tuple. The optimizing function is wrapped in a more user-friendly function which allows the use of NumPy arrays in addition to the PyTorch Tensor used in the optimizing function. The wrapper function is named fit. It conforms the inputs provided by the user into the expected types and matrix shapes of rows for samples and columns for input features or output targets. The processed inputs and outputs are also stored as support vectors for use during prediction. The fit function returns the class instance once training is done. This is for method chaining purposes for ease of use when using the fit function in a chain with other functions we will discuss next, such as the prediction function.

```
...
    def _optimize_parameters(
      self,
      X: torch.Tensor,
      y_values: torch.Tensor,
    ):

        A = torch.empty(
          (X.shape[0] + 1,) * 2,
          device=self.device,
          dtype=self.dtype,
        )
        A[1:, 1:] = self._batched_K(X, X)
        A[1:, 1:].diagonal().copy_(
            A[1:, 1:].diagonal()
            + torch.ones(
                (A[1:, 1:].shape[0],),
                device=self.device,
                dtype=self.dtype,
            ).to()
            / self.C
        )
        A[0, 0] = 0
        A[0, 1:] = 1
        A[1:, 0] = 1
        shape = np.array(y_values.shape)
        shape[0] += 1
        B = torch.empty(
          list(shape), device=self.device, dtype=self.dtype
        )
        B[0] = 0
        B[1:] = y_values

        solution: torch.Tensor = torch.linalg.lstsq(
            A.to(dtype=torch.float), B.to(dtype=torch.float)
        ).solution.to(dtype=self.dtype)

        b = solution[0, :]
        alpha = solution[1:, :]

        return (b, alpha)
...
```

Figure 4.19: Implementation of Least-squares Support Vector Regression fitting function.

The learned multipliers and bias parameters are then used for predicting outputs from unseen sample features. Equation (4.17) shows the computation that needs to

be done to compute the predicted output. The prediction process first constructs both matrices on the left-hand side of the equation. This is done by computing the kernel matrix between the prediction features and the training features, otherwise known as support vectors. The prediction function therefore is very straight forward with an input of unseen features and the learned parameters. The function output is simply the predicted outputs. The implementation of this function is shown in figure 4.20.

```
...
NumpyArrayorTensor = (
    np.ndarray[typing.Any, np.dtype[np.float_]] |
    torch.Tensor
)
...
    def predict(
        self,
        X: NumpyArrayorTensor,
    ) -> NumpyArrayorTensor:
        """Predicts the labels of data X given a trained model.
        - X: ndarray of shape (n_samples, n_attributes)
        """
        is_torch = isinstance(X, torch.Tensor)
        if is_torch:
            if X.ndim == 1:
                X_reshaped_torch = X.reshape(-1, 1)
            else:
                X_reshaped_torch = X
            X_= X_reshaped_torch.clone().to(
                self.device, dtype=self.dtype
            )
        else:
            if X.ndim == 1:
                X_reshaped_np = X.reshape(-1, 1)
            else:
                X_reshaped_np = X
            X_ = torch.from_numpy(X_reshaped_np).to(
                self.device, dtype=self.dtype
            )

        KxX = self._batched_K(X_, self.sv_x)

        y_pred = KxX @ self.alpha + self.b
        predictions: NumpyArrayorTensor
        if is_torch:
            predictions = y_pred.to(X)
        else:
            predictions = y_pred.cpu().numpy()

        if X.ndim == 1:
            return predictions.reshape(-1)
        else:
            return predictions
...
```

Figure 4.20: Implementation of Least-squares Support Vector Regression prediction function.

Aside from these core functions, interpretation of the model using the methods introduced by Üstün et al. (2007) are also implemented in the LSSVR class. First, using the support vectors and the kernel matrix computed between each support vectors, the correlation image is computed. This image, which is a correlation matrix, is intended to visualize the importance of features within kernel functions. This is simply computed with matrix multiplication between the kernel matrix and the support vectors. The implementation is shown in figure 4.21. The batched kernel function is used in this case to mitigate the memory footprint.

```
...
    def get_correlation_image(self):
        return self._batched_K(
            self.sv_x, self.sv_x
        ).mm(self.sv_x)
...
```

Figure 4.21: Implementation of Least-squares Support Vector Regression correlation image function.

The other interpretation method introduced by Üstün et al. (2007) visualizes the relationship learned between the input features and the outputs. This visualization is computed as the matrix multiplication between the support vectors and the learned Lagrange multipliers. The support vector matrix is transposed so that the multiplication is done on the dimension that indexes each sample. This computation is implemented as shown in figure 4.22. The resulting matrix is termed the p-matrix. The columns of this matrix represents the outputs and the rows represents the input features. The p-matrix values show how each feature contribute to the output learned by the model.

```
...
    def get_p_matrix(self):
        return self.sv_x.T.mm(self.alpha)
...
```

Figure 4.22: Implementation of Least-squares Support Vector Regression p-matrix function.

The second component of the model modules is the SpectralSVR implementation that uses a multi-output support vector regression (SVR) model to learn in the spectral

domain, such as the LSSVR implementation in the first component. The SpectralSVR is implemented as a class that extends the LSSVR to be able to learn from features and labels which are complex valued, which is the case for Fourier series coefficients. First, the constructor used to initialize each SpectralSVR instance accepts a Basis instance, an SVR instance which is the implemented LSSVR by default, and the verbosity of the model. These parameters are then used when the training function shown in figure 4.23 is called. Training the SpectralSVR model begins by calling the train function with parameters of the input function representation, output function representation, and an indicator of whether the output function is time dependent. Then, the function ensures that the features and labels are matrices. And then, if the input or output function representations are complex, they are transformed into the real representation. These formatted training data features and labels are then used to train the LSSVR model itself. Finally, to again allow for chaining methods, the function returns the current class instance.

```
...
    def train(
        self,
        f: torch.Tensor,
        u_coeff: torch.Tensor,
        u_time_dependent: bool = False,
    ):
        self.basis.time_dependent = u_time_dependent
        if self.basis.coeff_dtype.is_complex:
            u_coeff = to_complex_coeff(u_coeff)

        if f.ndim > 2:
            f = f.flatten(1)
        if u_coeff.ndim > 2:
            u_coeff = u_coeff.flatten(1)

        if torch.is_complex(u_coeff):
            u_coeff = to_real_coeff(u_coeff)
        if torch.is_complex(f):
            f = to_real_coeff(f)
        self.svr.fit(f, u_coeff)
        return self
...
```

Figure 4.23: Implementation of SpectralSVR training function.

Using the learned data to predict other output function representations from unseen

input functions can be done by directly calling the predict function of the LSSVR. This results in the real representation which needs to be converted into the complex representation for the FourierBasis. Once the conversion is done, the coefficients can be used to construct a new FourerBasis instance of the predictions. From there, the predicted functions can be evaluated and plotted. However, if one simply wants to evaluate the value of the predicted function at arbitrary points, another function is implemented to provide ease of use for this exact case. The implemented function is called forward as displayed in figure 4.24. The arguments to this function are the input functions, the evaluation points, and the domain span. The function first predicts the output function coefficients. Then, the predicted coefficients are multiplied with the basis function values at each point to compute the final predicted point values.

```
...
    def forward(
        self,
        f: torch.Tensor,
        x: torch.Tensor,
        periods: tuple[float, ...]
        | None = None,
    ) -> torch.Tensor:
        if len(x.shape) == 1:
            x = x.unsqueeze(-1)

        # compute coefficients
        if torch.is_complex(f):
            f = to_real_coeff(f)
        coeff = self.svr.predict(f)
        # convert to complex if basis needs complex values so that the reshaping is
        if self.basis.coeff_dtype.is_complex:
            coeff = to_complex_coeff(coeff)

        return self.basis.evaluate(
            coeff=coeff.reshape((f.shape[0], *self.modes)),
            x=x,
            periods=periods,
            time_dependent=self.basis.time_dependent,
        )
...
```

Figure 4.24: Implementation of SpectralSVR forward function (pointwise prediction).

The final core piece of functionality the SpectralSVR class provides is the inverse parameter estimation. This is implemented to solve inverse problems such as predicting

the initial conditions for the Burgers' equation or heat equation. This function essentially does the gradient descent on a loss function in such a way that the output function matches some expected output. The loss function is simply a measure of how far the outputs of the current predicted inputs are from the target outputs. The loss function is formulated as the mean squared error of the difference between the predicted output and expected output. Using some randomly initialized values as the inputs, the predicted output is then used to compute the loss. The gradient of the loss with respect to the inputs are then computed and then used to perform optimization of the inputs using the ADAM optimizer. This is implemented as the inverse function shown in figure 4.25. The function arguments are the expected output function coefficients, evaluation points, the loss function used, how many optimization loops are done, the random number generator, and the gain which is used in generating the random initial input functions. The function returns the estimated input function coefficients in the real representation. For Fourier coefficients this means that the predicted input coefficients need to first be converted to complex valued tensors and then used to construct a FourierBasis instance. For convenience, a wrapper like the forward function is also implemented for evaluating the function values of the predicted input coefficients.

```
...
    def inverse_coeff(
        self,
        u_coeff: torch.Tensor,
        loss_fn: Callable[
            [torch.Tensor, torch.Tensor], torch.Tensor
        ] = mean_squared_error,
        epochs=100,
        generator=torch.Generator().manual_seed(42),
        gain=0.05,
        **optimizer_params,
    ):
        f_shape = (u_coeff.shape[0], self.svr.sv_x.shape[1])
        complex_coeff = u_coeff.is_complex()
        original_device = u_coeff.device
        u_coeff = to_real_coeff(u_coeff.flatten(1)).to(self.svr.device)

        # inverse problem
        f_coeff_pred = (
            torch.randn(f_shape, generator=generator).to(self.svr.device) * gain
        )
        f_coeff_pred.requires_grad_()
        optim = torch.optim.Adam([f_coeff_pred], **optimizer_params)

        for epoch in range(epochs):
            optim.zero_grad()
            u_coeff_pred = self.svr.predict(f_coeff_pred)
            loss = loss_fn(u_coeff_pred, u_coeff)
            loss.backward()
            optim.step()
        optim.zero_grad()
        f_coeff_pred.requires_grad_(False)
        if complex_coeff:
            f_coeff_pred = to_complex_coeff(f_coeff_pred)
        return f_coeff_pred.to(original_device)
...
```

Figure 4.25: Implementation of SpectralSVR inverse coeff function (parameter estimation).

The SpectralSVR class also provides a convenience function for testing the performance of the learned model. This function is shown in figure 4.26. The function arguments are the testing inputs, expected testing outputs, and a resolution for evaluation of function values. The first step is to convert the testing input into real representations and then predicting the output function. The predicted output are then compared to the testing outputs using many metrics including RMSE, MSE, MAE, R^2,

sMAPE, RRSE, and the number of Nan values present in the predicted output. The same comparison process is carried out again for the function values which are evaluated at the specified resolution. Finally, the function returns the metric values as a nested tuple for both the coefficients predictions and the function value predictions.

```
...
    def test(
        self,
        f: torch.Tensor,
        u_coeff_targets: torch.Tensor,
        res: ResType = 200,
    ):
        if torch.is_complex(f):
            logger.debug("transform f to real")
            f = to_real_coeff(f)
        u_coeff_preds = self.svr.predict(f)
        if torch.is_complex(u_coeff_targets):
            logger.debug("transform u_coeff to real")
            u_coeff_targets = to_real_coeff(u_coeff_targets)

        grid = self.basis.grid(res).flatten(0, -2)
        u_preds = self.basis.evaluate(
            coeff=to_complex_coeff(u_coeff_preds),
            x=grid,
            time_dependent=self.basis.time_dependent,
        ).real
        u_targets = self.basis.evaluate(
            coeff=to_complex_coeff(u_coeff_targets),
            x=grid,
            time_dependent=self.basis.time_dependent,
        ).real
        return {
            "spectral": get_metrics(
                u_coeff_preds, u_coeff_targets
            ),
            "function value": get_metrics(u_preds, u_targets),
        }
...
```

Figure 4.26: Implementation of SpectralSVR test function.

Problems

The final module implements the PDEs that will serve as problems the proposed modules will solve. The implementation is mainly responsible for generating datasets from the PDEs and computing the PDE residuals. This implementation is focused

on the two synthetic datasets which are the antiderivative problem and the Burgers' equation problem. The problem base class is implemented as an abstract class that the specific problems will subclass. The base class requires any subclass to implement three methods. First, the data generation method which receives specifications such as what basis functions to use, how many function samples to create, how many modes are needed for each sample, and a PyTorch generator that will be used in generating the random basis coefficients. Other arguments more specific to each problem will be implemented by the specific subclass. This function is then expected to return a tuple of Basis instances. The second and third functions are the spectral and function value residuals for the PDE of each subclass, respectively. These three functions make up the functionality that is offered by all Problem subclasses. The implementation of the Problem subclass can be seen in figure 4.27.

```python
import abc
import torch
from ..basis import BasisSubType

class Problem(abc.ABC):
    def __init__(self) -> None:
        super().__init__()

    @abc.abstractmethod
    def generate(
        self,
        basis: type[BasisSubType],
        n: int,
        modes: int | tuple[int, ...],
        *args,
        generator: torch.Generator | None = None,
        **kwargs,
    ) -> tuple[BasisSubType, ...]:
        pass

    @abc.abstractmethod
    def spectral_residual(
        self, u: BasisSubType, *args, **kwargs
    ) -> BasisSubType:
        pass

    @abc.abstractmethod
    def residual(
        self, u: BasisSubType, *args, **kwargs
    ) -> BasisSubType:
        pass
```

Figure 4.27: Implementation of Problem base class.

The antiderivative problem is relatively simple to implement. The generation function takes advantage of the gradient operation and generate function provided by the Basis subclasses. The arguments to this function are the same as the base class with the addition of an integration constant parameter. By default, the value of the integration constant is zero. The function then returns the tuple of basis subclass instances representing the derivative and antiderivative functions. This function can be seen in figure 4.28.

```
...
    def generate(
        self,
        basis: Type[BasisSubType],
        n: int,
        modes: int | tuple[int, ...],
        u0: float | int | complex,
        *args,
        generator: torch.Generator | None = None,
        **kwargs,
    ) -> tuple[BasisSubType, BasisSubType]:
        if isinstance(modes, int):
            modes = (modes,)
        # generate solution functions
        u = basis.generate(
            n, modes, generator=generator, *args, **kwargs
        )
        # compute derivative functions
        ut = u.grad()
        # set the integration coefficient
        if isinstance(u0, complex):
            if u.coeff.is_complex():
                u.coeff[:, 0] = torch.tensor(u0)
            else:
                u.coeff[:, 0] = torch.tensor(u0).real
        elif isinstance(u0, float) or isinstance(u0, int):
            if u.coeff.is_complex():
                u.coeff[:, 0] = torch.tensor(u0 + 0j)
            else:
                u.coeff[:, 0] = torch.tensor(u0)
        else:
            u.coeff[:, 0] = u0

        return (u, ut)
...
```

Figure 4.28: Implementation of Antiderivative generate function.

The residual functions are similarly easy to implement as shown in figure 4.29.

```
...
    def spectral_residual(
        self, u: BasisSubType, ut: BasisSubType
    ) -> BasisSubType:
        residual = u.grad() - ut
        if residual.coeff is not None:
            residual.coeff[:, 0].mul_(0)

        return residual

    def residual(
        self, u: BasisSubType, ut: BasisSubType
    ) -> BasisSubType:
        u_val, grid = u.get_values_and_grid()
        ut_val = ut.get_values()
        dt = grid[1, 0] - grid[0, 0]
        u_grad = torch.gradient(
            u_val, spacing=dt.item(), dim=1
        )[0]
        residual_val = u_grad - ut_val
        residual = u.copy()
        residual.coeff = u.transform(residual_val)
        return residual
...
```

Figure 4.29: Implementation of Antiderivative residual functions.

For the Burgers' equation, the problem subclass for this PDE implemented a generate function with two generation paths. But before discussing the specifics of the two paths, the function has some extra arguments and preprocessing of those arguments. This time the function asks for the kind of functions the initial condition and the forcing term needs to be, whether they are random or constant. The combination of the kind of functions the initial conditions and forcing terms are will determine the path of generation that the function will take. The function also requires the viscosity parameter nu which is 0.01 by default. The domain also needs to be specified with a default value of 0 to 1 with 200 grid points for both space and time. The solver which is used for one of the generation procedures is also an argument with a default of the implicit Adams solver from torchdiffeq (R. T. Q. Chen, 2021). Finally, whether the output basis coefficients are time dependent or not is passed in as an argument alongside with the PyTorch generator. The implementation of this portion of the function can be seen in figure 4.30.

```
...
    def generate(
        self,
        basis: Type[BasisSubType],
        n: int,
        modes: int | tuple[int, ...],
        u0: ParamInput | BasisSubType = "random",
        f: ParamInput | BasisSubType = 0,
        nu: float = 0.01,
        space_domain=slice(0, 1, 200),
        time_domain=slice(0, 1, 200),
        solver: SolverSignatureType = implicit_adams_solver,
        time_dependent_coeff: bool = True,
        *args,
        generator: torch.Generator | None = None,
        **kwargs,
    ) -> tuple[BasisSubType, BasisSubType]:
        if isinstance(modes, int):
            modes = (modes,)

        device = "cuda:0" if torch.cuda.is_available() else "cpu"

        L = space_domain.stop - space_domain.start
        x = (
            basis.grid(slice(
                space_domain.start, space_domain.stop, modes[0]
            )).flatten(0, -2)
            .to(device=device)
        )
        T: float = time_domain.stop - time_domain.start
        nt = int(time_domain.step)
        # nt = int(T / (0.01 * nu) + 2)
        dt = T / (nt - 1)
        t = basis.grid(time_domain).flatten().to(device=device)
        periods = (T, L)
...
```

Figure 4.30: Implementation of Burgers generate function.

The first generation path and most similar to the antiderivative case is when both the initial condition and forcing terms are random. This branch of the generate function utilizes the residual function to compute the forcing term from the randomly generated solution function. This method of generating the solution and forcing term is stable and relatively straightforward. The implementation of this branch is shown in figure 4.31.

```
...
        if u0 == "random" and f == "random":
            # use method of manufactured solution
            time_mode = modes[0]
            if len(modes) > 1:
                modes = modes[1:]
            u = basis.generate(
                n,
                (time_mode, *modes),
                periods=periods,
                generator=generator,
            )
            res_modes = tuple(slice(0, L, mode) for mode in modes)
            fst = self.spectral_residual(
                u,
                basis(
                    basis.generate_empty(n, (time_mode, *modes))
                ),
                nu,
            )
            u_gen = u
            f_gen = fst
            # convert to timed dependent coeffs
            if time_dependent_coeff:
                u_val = u.get_values(res=(time_domain, *res_modes))
                u_coeff = basis.transform(
                    u_val.flatten(0, 1)
                ).reshape((n, nt, *modes))
                u_gen = basis(
                    coeff=u_coeff,
                    time_dependent=True,
                    periods=periods
                )

                f_val = fst.get_values(
                    res=(time_domain, *res_modes)
                )
                f_coeff = basis.transform(
                    f_val.flatten(0, 1)
                ).reshape((n, nt, *modes))
                f_gen = basis(
                    coeff=f_coeff,
                    time_dependent=True,
                    periods=periods
                )
...
```

Figure 4.31: Implementation of Burgers generate function manufactured method.

The second case is when at least one of the initial condition or forcing term is not random. This second case requires solving the Burgers' equation using traditional numerical methods. The generate function calls a function that has the responsibility of setting up the constant initial conditions or constant forcing terms and then calls the ODE solver specified in the generate function arguments. The right-hand side function computes the time derivative based on equation (4.14). The implementation of this is show in figure 4.32. Putting the above together gives us the implementation of the method of lines generation of Burgers' equation solution and forcing term as show in figure 4.33.

```
...
    @staticmethod
    def rhs(
        basis: type[Basis],
        nu: float,
        u_hat: torch.Tensor,
        f_hat: torch.Tensor,
    ) -> torch.Tensor:
        u = basis(u_hat)
        dealias_modes = tuple(
            int(mode * 1.5) for mode in u.modes
        )
        u_dealiased = u.resize_modes(
            dealias_modes, rescale=False
        )
        u_val = basis.inv_transform(u_dealiased.coeff)
        uu_x_hat_dealiased = 0.5 * basis.transform(u_val**2)
        uu_x = basis(
            uu_x_hat_dealiased
        ).resize_modes(u.modes, rescale=False).grad()

        u_u_x_hat = uu_x.coeff
        u_xx_hat = u.grad().grad().coeff
        u_t_hat = nu * u_xx_hat + f_hat - u_u_x_hat
        return u_t_hat
...
```

Figure 4.32: Implementation of Burgers generate function time derivative.

```
...
        u0.coeff = u0.coeff.to(device=device)
        fst.coeff = fst.coeff.to(device=device)
        print(f"generating with {len(t)} time steps")

        def f_func(t: torch.Tensor, x: torch.Tensor = x):
            x = x.tile((1, 2))
            x[:, 0] = t
            return basis.transform(
                fst(x).reshape((-1, modes[0]))
            )

        def rhs_func(t: torch.Tensor, y0: torch.Tensor):
            y0 = to_complex_coeff(y0)
            return to_real_coeff(
                cls.rhs(basis, nu, y0, f_func(t))
            )

        u_hat = solver(rhs_func, to_real_coeff(u0.coeff), t)
        if basis.coeff_dtype.is_complex:
            u_shape = u_hat.shape
            u_hat = to_complex_coeff(
                u_hat.flatten(0, 1)
            ).reshape(
                (*u_shape[:2], *u0.coeff.shape[1:])
            )

        u_hat = u_hat.movedim(0, 1)
        if timedependent_solution:
            u = basis(u_hat, **kwargs, time_dependent=True)
        else:
            u_hat = u_hat.reshape((n * nt, modes[0]))
            u = basis(
                basis.transform(
                    basis.inv_transform(u_hat).reshape(
                        (n, nt, modes[0])
                    )
                ),
                **kwargs,
            )
        u.coeff = u.coeff.cpu()
        fst.coeff = fst.coeff.cpu()
        return (u, fst)
...
```

Figure 4.33: Implementation of Burgers generate function method of lines.

The spectral residual of the Burgers' equation which is used in the method of

manufactured solution generation approach is implemented as shown in figure 4.34. As with the right-hand side function, we use the pseudospectral approach for computing the convolution term to avoid large computational expenses for higher number of modes. The function value residual is also implemented similarly to the antiderivative version. The derivatives are computed with the PyTorch gradient function. And then using equation (2.2), the residual is computed by subtracting the forcing term from both sides.

```
...
    def spectral_residual(
        self, u: BasisSubType, f: BasisSubType, nu: float
    ) -> BasisSubType:
        u_t = u.grad(dim=0, ord=1)

        dealias_modes = tuple(
            int(mode * 1.5) for mode in u.modes
        )
        u_dealiased = u.resize_modes(
            dealias_modes, rescale=False
        )
        u_val = u.inv_transform(u_dealiased.coeff)
        uu_dealiased = u.copy()
        uu_dealiased.coeff = u.transform(u_val.pow(2).mul(0.5))
        uu_x = uu_dealiased.resize_modes(
            u.modes, rescale=False
        ).grad(dim=1)

        u_xx = u.grad(dim=1, ord=2)
        nu_u_xx = u_xx
        nu_u_xx.coeff = nu_u_xx.coeff * nu

        residual = u_t + uu_x - nu_u_xx - f
        return residual
...
```

Figure 4.34: Implementation of Burgers spectral residual function.

4.4.4 Testing

The implementation is followed by tests of the functionality core to this study. The testing is done using the PyTest library. The testing method used is a black box approach. The tests are written in a separate `tests` directory. Each module is tested by a separate file. The tests are done to ensure compliance of the implementation to the requirements of the software. The final results are presented in table 4.1. Each version and iteration of

the development must ensure that the tests are passed before any changes are committed to the GitHub repository.

No	Process	Test Target	Test Result
1	Transform	Transform invertible with inverse transform	Achieved
2	Evaluation	Transform coefficient evaluation close to original values	Achieved
3	Plot	Plotting function is run successfully	Achieved
4	Perturb	Perturb function is run successfully	Achieved
5	Operations	Operation functions is run successfully	Achieved
6	Complex Coefficients Conversion	Conversion between real and complex coefficient representation is invertible for odd and even number of modes	Achieved
7	SpectralSVR Train	Training function is run successfully	Achieved
8	SpectralSVR Prediction	Coefficient prediction error within tolerable range	Achieved
9	SpectralSVR Prediction	Function value prediction error within tolerable range	Achieved

Table 4.1: Testing results of the SpectralSVR Library implementation using black box method.

4.5 Experimental Scenarios

To achieve the goals of this study, as outlined in section 1.3, we present three experimental scenarios. These scenarios represent different experimental configurations that will be applied to our implementation of the SpectralSVR library. A table of the configurations for each scenario can be seen in table 4.2.

Table 4.2: The configuration of experimental scenarios in this study.

Configuration	Scenario 1	Scenario 2	Scenario 3
Data Source	Synthetic (computed from equations)	Synthetic (computed from equations) with three viscosity values	ERA5 dataset
Data Augmentations	Perturbations of three different noise levels	Perturbation of 10% the mean standard deviation of function values	
Data Sample Size	5000 unique functions for each noise level	500 unique functions for each viscosity value	7000 time steps
Number of Modes			
Scaling	Standard scaling	Standard scaling	Standard scaling
Dataset Split	80% Training and 20% Testing for each noise level	80% Training and 20% Testing for each viscosity value	80% Training and 20% Testing
Regularization Parameter (C)	1.0	1.0	1.0
Kernel	RBF ($\sigma = features \times X.var()$)	RBF ($\sigma = features \times X.var()$)	RBF ($\sigma = features \times X.var()$)
Additional Tests	Exact solutions	Exact solutions	
Interpretation	p-matrix and correlation image	p-matrix and correlation image	p-matrix and correlation image

Each scenario serves a different purpose. The first scenario is constructed to establish whether the model can learn any relationships at all in data that implicitly defines differential equations. This scenario is a proof of concept of whether the model is capable of accurately learning the relationship between derivatives and their anti-derivatives. As the simplest case, it provides a foundation for understanding the

model and its behavior. The 5% noise model is then used to predict the antiderivative of a cosine function with a period of one as shown in equation (4.21). The exact solution is known as equation (4.22) (Abramowitz & Stegun, 1972). The second goal of this scenario is to determine how noise affects the model's predictions. This is especially important for the inverse problem because of how sensitive the derivative is to changes in the antiderivative. To further solidify how the model fares for inverse problems, this scenario will also assess the performance of a trained model for a very simple inverse problem setup to determine the derivatives of the testing subset from the antiderivatives. This setup will utilize the model as a surrogate forward solver and optimize for a loss function using the inverse function of the SpectralSVR class as described in section 4.4.3.

$$f(x) = 2\pi \cos(2\pi x) \tag{4.21}$$

$$f(x) = \sin(2\pi x) \tag{4.22}$$

The second scenario extends the goal of the first scenario by examining the model and its behavior when learning a nonlinear PDE. The data subsets with different viscosity values are intended to allow us to learn more about how this will affect the performance on the exact solutions for lower viscosity numbers. The model will learn to predict future states of systems governed by the Burgers' equation from past states. The model for each viscosity will be used to predict equation (2.4) from initial conditions at time equals zero and the corresponding forcing term of a constant value of zero. The exact equation prediction will be done using a rollout method. The initial condition along with the initial forcing term will be given to the model to predict the solution at the first time step. The model will then use the predicted solution with the forcing term at the first time step to predict the solution at the second time step. This is repeated until the stopping condition of time $t = 10$. The forcing term for the exact solution is a constant zero. Along with the rollout of the exact solution, we also perform roll out on one of the test solution.

The third scenario is designed to demonstrate the modeling capability of SpectralSVR on a more practical problem of weather prediction. The earth weather system is a complex collection of processes. Using the ERA5 dataset, the model will be used to predict future states of the atmosphere from previous states. Specifically,

the model will learn to predict the temperature 2 meters above the surface. That means that other than the time and spatial resolution not being the highest, the model will also learn from partial information of the relationships defined by the equation since no other variables will be included. This choice was made as an analog for situation with unknown PDEs and therefore unknown variables. With these situations, there is no guarantee that the variables measured have any relation with the system. These three scenarios together address the last two aims of this study.

4.6 Experimental Results

In this section, the results of each experimental scenario is presented and analyzed. The results come in the form of testing metrics mentioned in section 2.7. The model interpretations are displays of the correlation image and the p-matrix. The function values and coefficients themselves will also be plotted depending on the number of dimensions.

4.6.1 Scenario 1

The first scenario results are presented in four parts. The first is the coefficient prediction metrics which is shown in table 4.3. The table shows the noise level and the performance metrics.

Table 4.3: Performance metrics of coefficient prediction in scenario 1 by noise level.

Noise level	MSE	RMSE	MAE	R^2	sMAPE
5%	0.86	0.93	0.57	0.96	0.31
10%	2.06	1.43	0.93	0.91	0.46
50%	15.59	3.95	3.04	0.50	1.04

The scenario is implemented as a Jupyter notebook which imports relevant libraries and our SpectralSVR library. The notebook starts with creating the data with the Antiderivative problem subclass. All random processes use the same PyTorch Generator instance with a seed of 42. The data is then augmented by perturbing the function values to create three versions with the noise levels mentioned in the scenario. Looping through each version, the training and testing subsets are randomly sampled to result in 80% and 20% of the original dataset respectively. Then the scaling function is fitted on

the training subset inputs. And then, both the training and testing inputs are transformed using the fitted scaling function. Next, an instance of SpectralSVR is trained and tested. Finally, the metrics are stored and the next noise level is processed. Next the 5% noise model is used to predict the antiderivative of a sine function.

The kernel scaling hyperparameter sigma has the same value of 10 for all noise levels. This is because the inputs are all scaled to approximately have a standard deviation of one. The value itself is from the number of features which comes out to 100 total inputs to the LSSVR.

Table 4.3 shows that the model is able to generalize from the training data and learn the simple linear antiderivative operator. Focusing on the coefficient of determination (R^2) scores, we can see that the model is off by 0.04 to the perfect score of 1.0 for the lowest noise level. While increasing noise levels does degrade the performance, this is partly due to the target values also having been perturbed. This fact can be seen if we look at how predictions from the very same trained models on noisy inputs are compared to the noise-free versions of the antiderivative functions which is shown in table 4.4. The metrics show that the model predictions are slightly closer to the unperturbed function coefficients compared to the perturbed versions. This can be seen as the influence of the independent synthetic measurement noise in the perturbed targets being removed from the testing metrics.

The higher noise levels shows that the model is still relatively accurate. Looking at the R^2 score for the 50% noise level, the model is more accurate than a baseline mean value model. The R^2 score of 0.5 means that the model predictions has roughly half the error the baseline would produce.

Table 4.4: Performance metrics of coefficient prediction compared to unperturbed targets in scenario 1 by noise level.

Noise level	MSE	RMSE	MAE	R^2	sMAPE
5%	0.79	0.89	0.51	0.96	0.28
10%	1.80	1.34	0.80	0.92	0.40
50%	9.30	3.05	2.23	0.62	0.90

The function values are also evaluated and the values of the metrics between the noisy targets and predictions are shown in table 4.5. While the relative metrics R^2 and

sMAPE stay close to the values we obtained for the coefficients themselves, since the metrics are now being computed in the physical domain, the scales have changed and because of that the values of the absolute metrics now reflect the change in scale. For reference, the scale of the target coefficient values for the 50% noise level dataset in their real representation can range on average from -13.54 to 13.6 across the entire dataset. Therefore, an RMSE value of 3.94 for the predictions compared to the noisy targets mean that the error ratio is roughly 0.148 or just above 1 to 7. For the function values of the 50% noise dataset the values range on average from -2.5 to 2.49. This was more difficult to compute since there is no direct way to compute the amplitude of the function from just the coefficients. Here we used the inverse transform and extracted the maximum and minimum values of each function from the discrete values. Since the RMSE in table 4.5 for the 50% noise dataset is 0.63, the error ratio comes to 0.126. One side note about table 4.5 is that since the model instances are the same as the ones used in the coefficient evaluations, the hyperparameters are still the same including the kernel scaling factor sigma.

Table 4.5: Performance metrics of function value from evaluated coefficient prediction in scenario 1 by noise level.

Noise level	MSE	RMSE	MAE	R^2	sMAPE
5%	0.03	0.18	0.15	0.97	0.39
10%	0.08	0.29	0.23	0.92	0.55
50%	0.62	0.79	0.63	0.49	1.05

For the metrics between the predictions and the noise-free version of the target function values, the results parallel the outcomes of the coefficient predictions. The metrics show that the model performs slightly better with more pronounced differences for the higher noise levels. This echoes our previous statement on the effect of independent “measurement” noise in the target values on the evaluation metrics.

Table 4.6: Performance metrics of function value from evaluated coefficient prediction compared to unperturbed targets in scenario 1 by noise level.

Noise level	MSE	RMSE	MAE	R^2	sMAPE
5%	0.03	0.18	0.14	0.97	0.38
10%	0.07	0.27	0.21	0.93	0.53
50%	0.37	0.61	0.49	0.62	0.94

The next part of this scenario is the results from predicting the exact solution of equation (4.21) which is equation (4.22). The function values will be computed with the same discretization as the number of modes we used in the training set. Once the values are obtained for both functions, the coefficients are computed. Then instancing the FourierBasis class with the coefficients of the derivative function, we can create perturbed versions according to the noise levels of the training set. Finally, the derivative is used to predict the noise-free antiderivative. The results are presented in tables 4.7 and 4.8. For 5% and 10% noise levels, the model shows that it has learned the relation. However, the 50% noise level, the model has been unable to predict the results well enough. The marked difference in error across all metrics between the 50% and the lower noise levels is more apparent for this specific exact problem compared to the test sets. A clear picture of this can be seen when we compare the sMAPE metric. For the test set, the predictions on average results in a value of 1, however, for this exact problem the sMAPE value is 1.4 to 1.7.

Table 4.7: Performance metrics of coefficient prediction of exact antiderivative in scenario 1 by noise level.

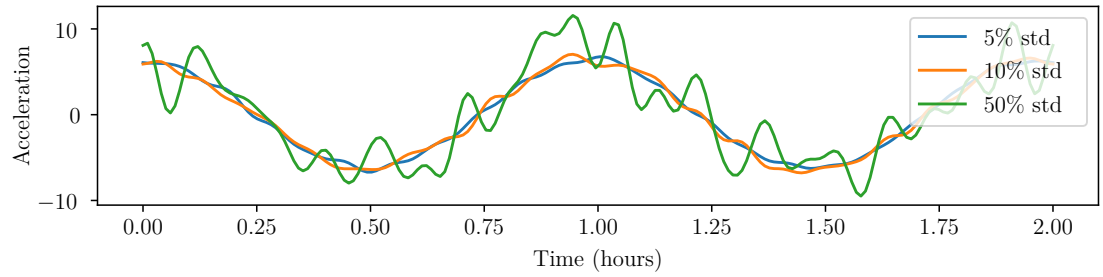
Noise level	MSE	RMSE	MAE	sMAPE
5%	0.09	0.29	0.16	1.18
10%	0.80	0.89	0.31	1.23
50%	10.33	3.21	0.83	1.44

Table 4.8: Performance metrics of evaluated function values of coefficient prediction of exact antiderivative in scenario 1 by noise level.

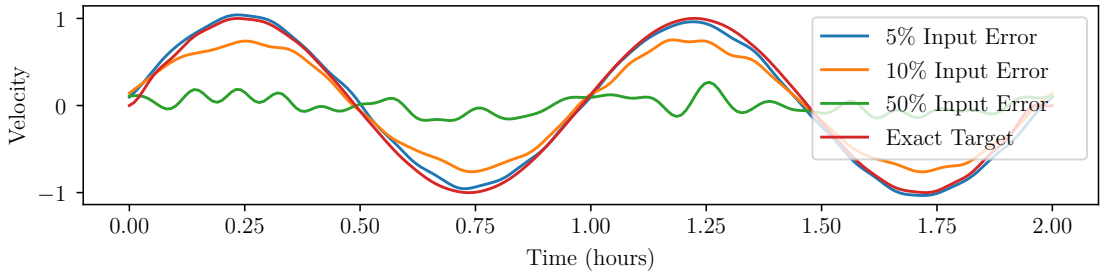
Noise level	MSE	RMSE	MAE	sMAPE
5%	0.00	0.06	0.05	0.23
10%	0.03	0.18	0.15	0.36
50%	0.41	0.64	0.58	1.65

The predictions of the exact equation is shown in figure 4.35. Visually, we can see that as the noise level increases, the model predictions become worse. Another observation is how the higher the noise level, the more the predicted functions become closer to zero. This is explained by the double penalty phenomenon (Lledó et al., 2023). The loss function used for LSSVR penalizes the model for predicting a non-zero value

that turns out to be wrong compared to predicting a zero value. This results in a model with predictions that are close to the mean of the training data.



(a)



(b)

Figure 4.35: (a) The perturbed exact input function values from equation (4.21). (b) Prediction of antiderivative from the input function that was perturbed.

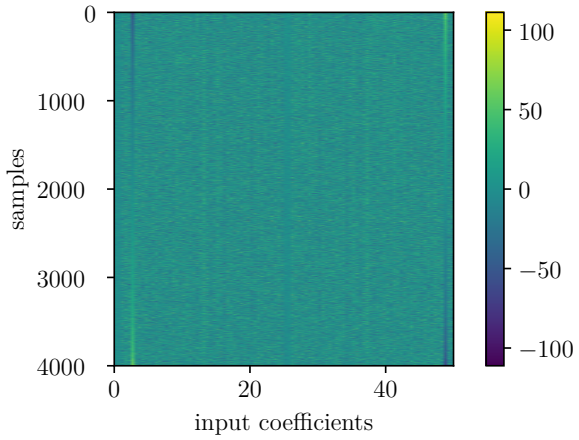
For inverse problems, we simply use the same testing set with the labels now used to predict the features. The model instances are still the same as before. With the implementation shown in figure 4.25, we run the inverse function for predicting coefficients with 2000 epochs, a learning rate of 0.02, and randomized starting point from a normal distribution multiplied by a factor of 0.05. The results of inverse prediction for all three noise levels are shown in table 4.9. These metrics were computed after both the target and predicted input functions are scaled back using the same scaling functions they were scaled down with. Because the scales of the input and output functions are different, the absolute metrics are much larger in these results compared to the forward problem results. For reference, the target input coefficients on average range from -695.65 to 694.9. The relative metrics show that for the lower noise levels, the accuracy is still relatively good. However, the values are lower than the values for the forward test. The difference with the antiderivative prediction becomes more pronounced with the higher noise levels. For the highest noise level, the performance has degraded so far that the predictions are worse than a baseline model from the mean

of the target input functions. One must also note that despite the much worse results of the other metrics for the highest noise level, at this time we're unable to explain why sMAPE results in roughly the same performance as in the forward problem.

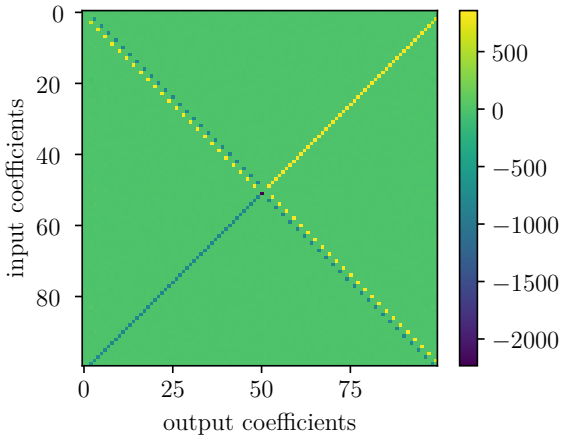
Table 4.9: Performance metrics of coefficient inverse prediction of derivative in scenario 1 by noise level.

Noise level	MSE	RMSE	MAE	R^2	sMAPE
5%	1739.56	41.71	26.44	0.91	0.36
10%	3485.10	59.03	42.19	0.76	0.52
50%	49319.73	222.08	174.48	-0.19	1.05

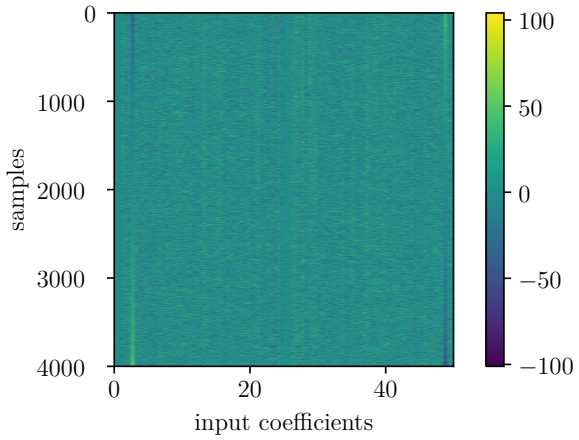
Next, we have the correlation image and p-matrix for the model of each noise level. The results are shown in figure 4.36. Each row in the figure represent the results of a model trained on a different noise level. The correlation image for 5% and 10% noise show clear lines on the coefficient corresponding to the output coefficient that was sorted. We can also see that the negative wave number reflection also being sorted. This is because of the reflection that occurs with complex Fourier coefficients for real-valued functions. There are other faint vertical lines you are able to see for other input coefficients. These faint lines are not sorted like the corresponding coefficients we mentioned before. Meaning, while they don't affect the particular output coefficients that were sorted, they do show that there is information embedded in the kernel matrix for these input coefficients. However, if we look at the 50% noise correlation image, even the corresponding coefficient does not show a clear line or gradient. The lines are more noisy compared to the other noise levels. But the kernel still manages to embed some information. The p-matrices show that The model itself is able to still learn some relationship between the input coefficients and the output coefficients. For the 5% and 10% noise levels, the p-matrices show very clearly the contributions of input coefficients to the output coefficients. However, looking at the p-matrix for 50% noise level, the lower wave numbers show lower contribution of input coefficients to the output coefficients.



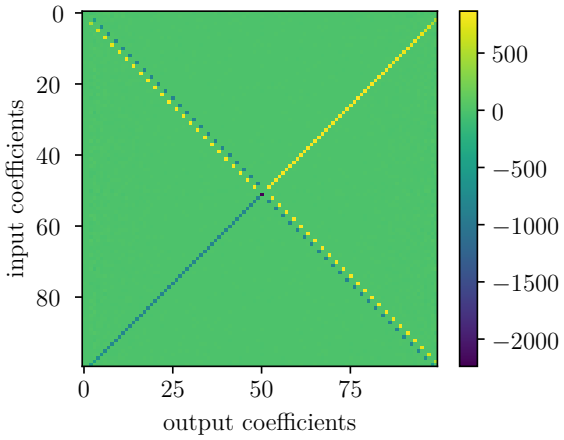
(a) Correlation image 5% noise.



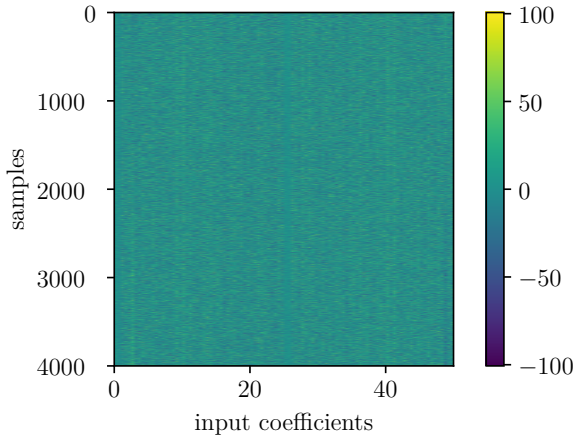
(b) The p-matrix for 5% noise.



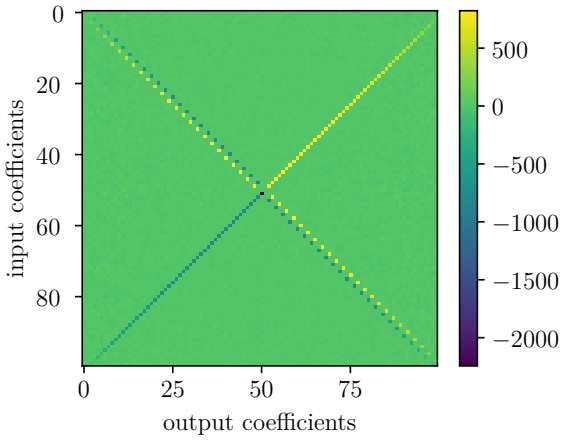
(c) Correlation image 10% noise.



(d) The p-matrix for 10% noise.



(e) Correlation image 50% noise.



(f) The p-matrix for 50% noise.

Figure 4.36: Correlation image (left column) and p-matrix (right column) for each model trained on a different noise level (row). The correlation image was sorted same order the values of the real component of wave number $k = 2$ were sorted in descending order.

The final observation we make is how for all the p-matrices the pronounced contribution of the input wave number to their corresponding output wave number. This means that the majority of contributions to each output coefficients come from the corresponding input coefficients of the same wave number. Our knowledge on how the simple derivative equation for Fourier series relate the coefficients of the derivative function and the antiderivative function is shown in equation (4.5). This aligns with the contributions shown by the p-matrices. This confirms that the model is indeed learning the relations that is defined by the derivative equation.

In summary, the model is able to learn the linear relationship defined by the derivative equation. This is true even for high noise levels. The model also learns the actual relationship between each wave number which is shown in the p-matrix. Inverse prediction of the input function is also successfully done for lower noise levels. At high noise levels, the noise overwhelms the model.

4.6.2 Scenario 2

The second scenario results are presented in two parts which are the coefficient predictions and function value predictions. The functions are first preprocessed in order to format them into current and next values in time as discussed in section 4.3.2. The sample pairs are first divided into two for the training and testing sets. Each function is then discretized into 200 time steps. Then, the features are put together as the coefficients of the solution and forcing term at the current time step. The labels are the solution at the next time step. Since there is too much data to fit into memory all at once at the end of this process, we randomly sample the feature and label pairs. This result in a maximum of 8000 samples for the training set and 2000 samples for the testing set. As mentioned in section 4.3.2, the training features are used to fit a scaling function. And then, all features are scaled using the fitted scaling function.

The coefficient value predictions are shown in table 4.10. The RBF kernel parameter computed from the data is once again the same for all versions of the dataset with a value of 5.66. The testing metrics show that the performance of the model vary with the viscosity value ν. Starting with the inviscid Burgers' equation, with $\nu = 0$, the model performs the best with a close second from the model for the $\nu = 0.01$ dataset version. The function value evaluated from the predictions is shown in table 4.11. The metrics across the board shows that the function value relative metrics are slightly better

compared to the coefficients. Comparing the absolute metrics for the same table, we see the same general trend that the higher viscosity show the model performing worse. For reference, the maximum amplitude of the functions on average are 2.56×10^{-2}. This puts the error in an order of magnitude less than the average maximum amplitude. For the coefficients, on average, the maximum absolute value at each time step is 9.29×10^{-2}. This means for the viscosity of 0.0 and 0.01, the error is roughly 1 to 9 of the maximum absolute value.

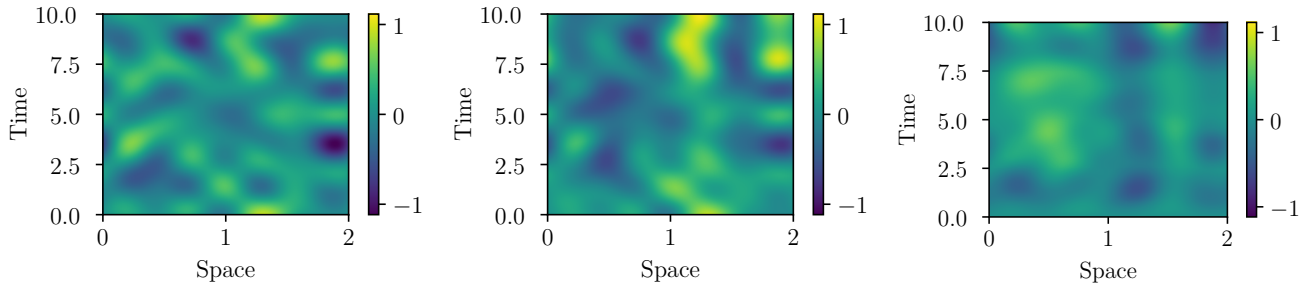
Table 4.10: Performance metrics of coefficient prediction of next time step in scenario 2 by viscosity.

ν	MSE	RMSE	MAE	R^2	sMAPE
0.0	3.46e-04	1.86e-02	1.24e-02	0.79	0.65
0.01	3.56e-04	1.89e-02	1.27e-02	0.78	0.67
0.1	9.17e-04	3.03e-02	2.32e-02	0.46	1.08

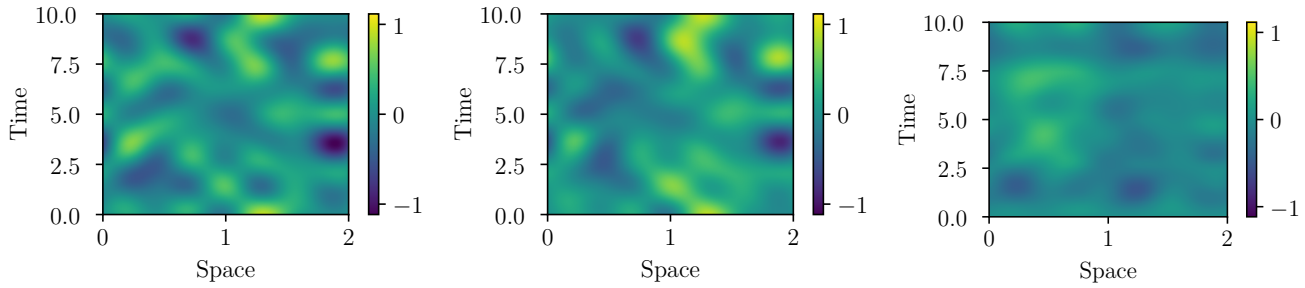
Table 4.11: Performance metrics of function values evaluated from coefficient prediction of next time step in scenario 2 by viscosity.

ν	MSE	RMSE	MAE	R^2	sMAPE
0.0	5.58e-05	7.47e-03	5.89e-03	0.82	0.70
0.01	5.74e-05	7.58e-03	6.01e-03	0.81	0.71
0.1	1.58e-04	1.26e-02	9.98e-03	0.51	1.04

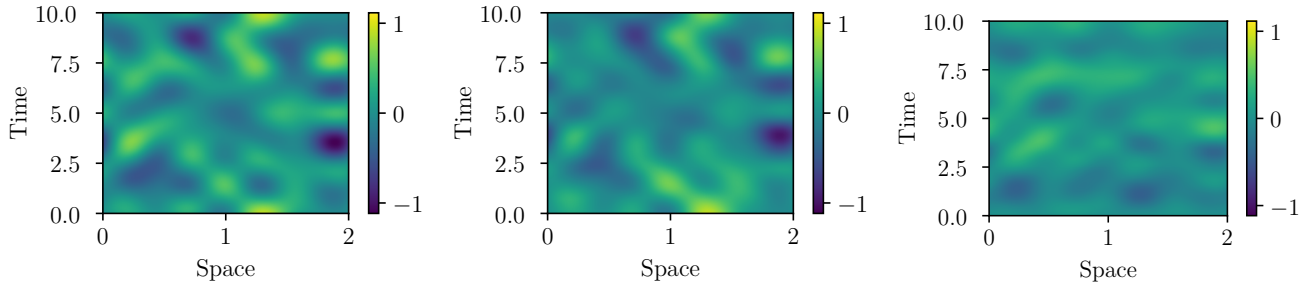
The testing results are complimented with a rollout experiment. The results from the rollout are shown in figure 4.37. The rollout starts from time $t = 0$. In all the plots, this is at the bottom. We have included this initial condition in the plots. From this initial condition in combination with the forcing term at the initial time step, the model predicts the first time step solution. The model then uses the predicted solution combined with the corresponding forcing term to predict the second time step. To see how well the model is performing relative to the actual target values, we compute the difference between the two as shown in figures 4.37c, 4.37f and 4.37i. Observe that errors are introduced relatively early around time $t = 1$. However, the model rollout stays stable until the end despite this error. This results in the model being able to finish the rollout for this sample.



(a) The target for $\nu = 0.0$ (b) The prediction for $\nu = 0.0$ (c) The difference for $\nu = 0.0$



(d) The target for $\nu = 0.01$ (e) The prediction for $\nu = 0.01$ (f) The difference for $\nu = 0.01$



(g) The target for $\nu = 0.1$ (h) The prediction for $\nu = 0.1$ (i) The difference for $\nu = 0.1$

Figure 4.37: The rollout predictions for one of the test function. The difference is calculated as the targets subtracted by the predictions.

Another observation we can see is that the difference between the target and rollout predictions is much more pronounced for lower viscosity values. This result is aligned with metrics shown in tables 4.12 and 4.13. The absolute metrics show that the models are performing several times worse in rollout compared to the single time step tests shown in tables 4.10 and 4.11. This seems to be the same challenge that traditional solvers also face with the shocks that may be present with lower viscosity values. This means that for rollout scenarios, the model has a much harder time with lower viscosity values.

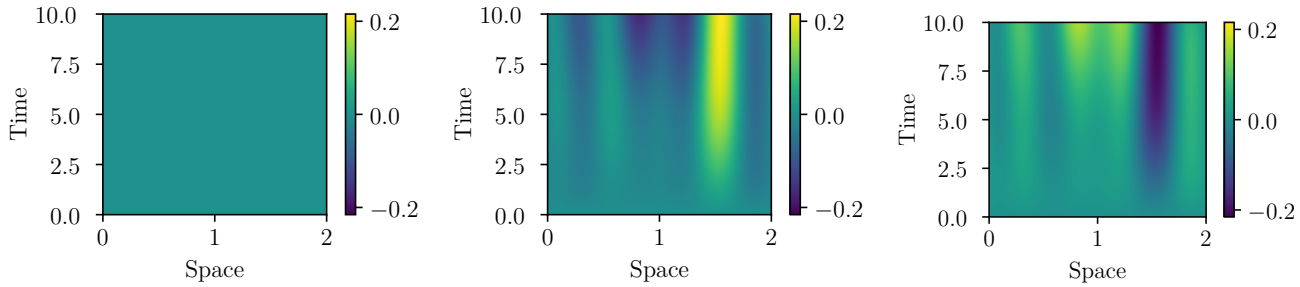
Table 4.12: Performance metrics of coefficient rollout in scenario 2 by viscosity.

ν	MSE	RMSE	MAE	sMAPE
0.0	2.91e-01	5.40e-01	4.07e-01	0.90
0.01	1.61e-01	4.01e-01	2.86e-01	0.75
0.1	1.70e-01	4.12e-01	3.02e-01	0.81

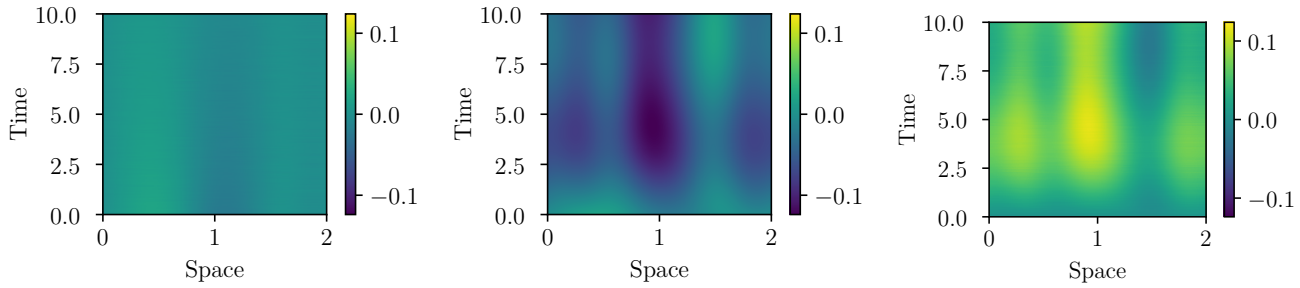
Table 4.13: Performance metrics of function values evaluated from coefficient rollout in scenario 2 by viscosity.

ν	MSE	RMSE	MAE	sMAPE
0.0	5.81e-02	2.41e-01	1.89e-01	1.04
0.01	3.17e-02	1.78e-01	1.40e-01	0.94
0.1	2.91e-02	1.71e-01	1.33e-01	0.92

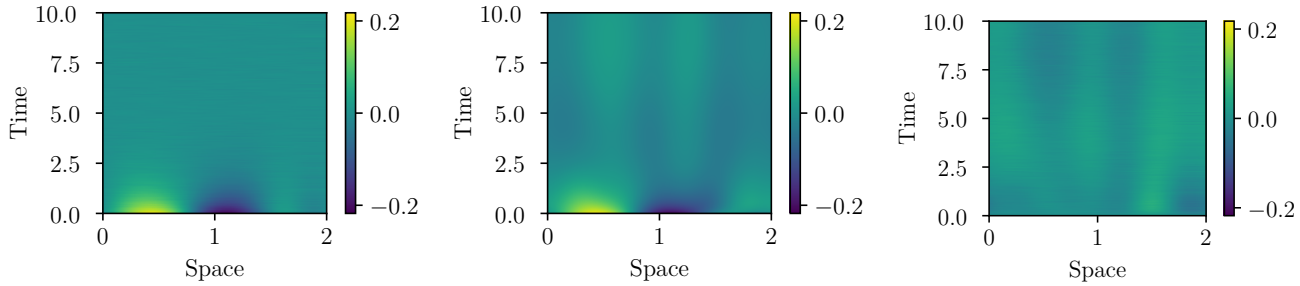
The exact function eq. (2.4) is computed for each viscosity value. The discrete Fourier transform of the values are then used for doing rollout with the model. For the forcing term, we just use a constant function with a value of zero. The rollout predictions are shown in figure 4.38. The model is not successful in the rollout for the inviscid equation and for the viscosity of $\nu = 0.01$. Both of these cases, the model produces too much error that the original function is no longer recognizable in the predictions. For the higher viscosity value of $\nu = 0.1$ the rollout produces a recognizable prediction. The error stays to about half the maximum function value. We do see some more noticeable error in the flatter parts of the function. The error also take a very similar shape across the different viscosity values. Since we essentially used the same solutions for training the model for each viscosity, the similarity in error means that it can be alleviated with more diverse training samples. The error for the flatter functions may also be alleviated by including similar flatter functions during training.



(a) The target for $\nu = 0.0$ (b) The prediction for $\nu = 0.0$ (c) The difference for $\nu = 0.0$



(d) The target for $\nu = 0.01$ (e) The prediction for $\nu = 0.01$ (f) The difference for $\nu = 0.01$



(g) The target for $\nu = 0.1$ (h) The prediction for $\nu = 0.1$ (i) The difference for $\nu = 0.1$

Figure 4.38: The rollout predictions for one of the exact function in equation (2.4). The difference is calculated as the targets subtracted by the predictions.

The metrics for the exact rollout further solidifies our observations from the plots. The metrics are shown in tables 4.14 and 4.15. While the absolute metrics are low, this is mostly due to the maximum function values of the exact function also being pretty low. The relative metric sMAPE shows a much clearer picture. As expected, the inviscid prediction is pretty much just error. The next result, for viscosity $\nu = 0.01$, is better. This seems to be from the slight conformance to the target. The plots for this viscosity show that the prediction contains "shapes" similar to the target despite the scale not being close to the target. Finally, the highest viscosity level results show

the best performance. Despite the good performance in the higher detail areas of the function, the error from the flatter areas overwhelm the metrics due to its larger portion of the function.

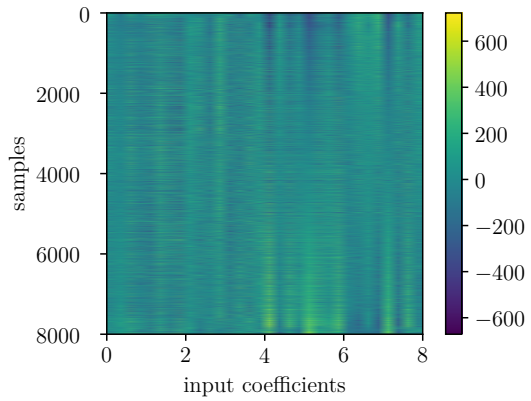
Table 4.14: Performance metrics of coefficient rollout of exact solution in scenario 2 by viscosity.

ν	MSE	RMSE	MAE	sMAPE
0.0	2.16e-02	1.47e-01	1.08e-01	2.00
0.01	1.32e-02	1.15e-01	6.37e-02	1.58
0.1	2.83e-03	5.32e-02	4.00e-02	1.50

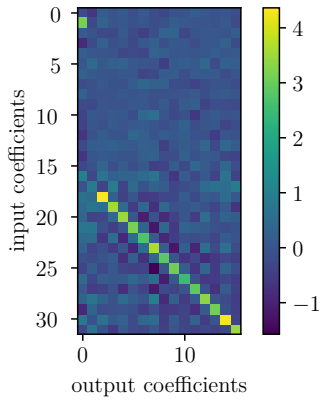
Table 4.15: Performance metrics of function values evaluated from coefficient rollout of exact solution in scenario 2 by viscosity.

ν	MSE	RMSE	MAE	sMAPE
0.0	4.16e-03	6.45e-02	4.56e-02	2.00
0.01	3.23e-03	5.68e-02	4.94e-02	1.73
0.1	3.86e-04	1.97e-02	1.62e-02	1.47

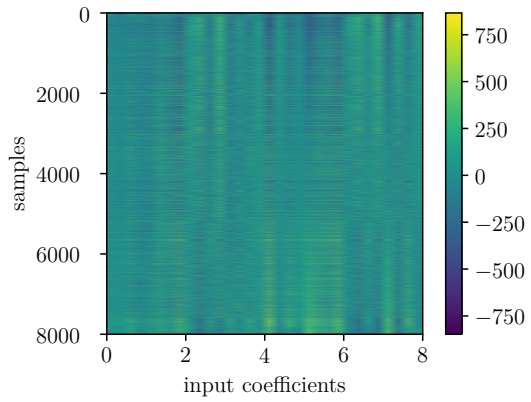
The correlation image and p-matrix for each viscosity can be seen in figure 4.39. Analyzing the correlation images first, we see that some columns are sorted in correlation to how the samples are sorted based on the real component of the 4th output wave number. For the inviscid equation, this correlation is the strongest with multiple columns exhibiting the correlation. The highest and visually most sorted values are from the forcing term portion of the inputs. Different wave numbers in the inputs are also sorted differently, this indicates an inverse correlation. For a higher viscosity of $\nu = 0.01$, a weaker correlation is still present.



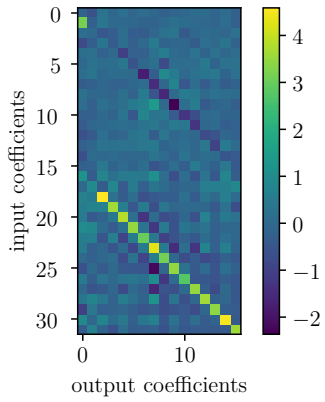
(a) Correlation image $\nu = 0.0$.



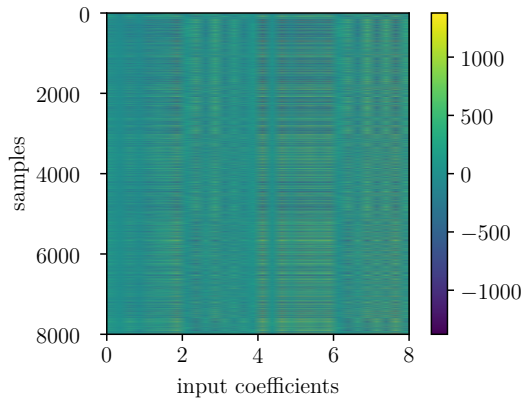
(b) The p-matrix for $\nu = 0.0$.



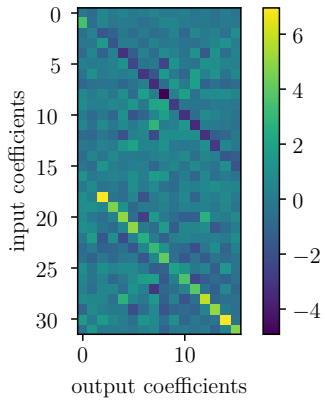
(c) Correlation image $\nu = 0.01$.



(d) The p-matrix for $\nu = 0.01$.



(e) Correlation image $\nu = 0.1$.



(f) The p-matrix for $\nu = 0.1$.

Figure 4.39: Correlation image (left column) and p-matrix (right column) for each model trained on a different viscosity (row). The correlation image was sorted same order the values of the real component of wave number $k = 2$ were sorted in descending order.

Once we get to the highest viscosity, while the order is still present, we see a

more random arrangement of the correlation image values. This observation and the lower scores can both be explained by the level of temporal discretization. This is the same issue we encounter when we use some traditional methods for higher viscosity values with the same time step for lower viscosity values (Kassam & Trefethen, 2005; Seydaoğlu et al., 2016). The issue arises in stiff differential equations. Stiff differential equations refer to the presence of a rapidly varying component. We can see that the larger amplitudes of high frequency components in the forcing term to represent the rapidly varying components as shown in figures 4.2b, 4.2d and 4.2f. Traditionally, stiff equations would require specific numerical methods purpose built to efficiently solve the problem. Otherwise, the using a solver like FDM would require much finer time steps than the one we are using here.

The p-matrix shows the difference in contributions of both input functions. For the inviscid equation, the contributions mainly come from the forcing term. The current solution contributes a smaller amount in comparison. There is similarity with the antiderivative p-matrices in this case. The contributions to each output wave number mainly come from input coefficients with the same wave number. However, there is a noticeable contribution from other wave numbers too unlike the antiderivative case. The biggest contribution from the current solution is the constant/bias term. For higher viscosity values, notice that the contribution of the previous solution increases. The increase is more pronounced for higher frequencies. This aligns with the quadratic scaling from the wave number on the coefficients (Canuto et al., 2007). The p-matrix for lower viscosity values show more structured contribution of other wave numbers in comparison to the higher wave numbers. This structure comes in the form of a grid pattern. For the real components of the output, the contribution of other wave numbers oscillate between higher and lower values depending on whether the input component is real or not. The imaginary output components on the other hand have a more constant contribution across the input components. The contribution once again become more pronounced the higher the absolute value of the wave number.

To summarize, Low viscosity values are a challenge to the model as with traditional methods. Despite that, this scenario confirms the model's ability to learn the nonlinear relationship defined by the Burgers' equation. The model is also able to stay stable with a relatively large time step. This concludes the synthetic data portion of the study.

4.6.3 Scenario 3

This final scenario tests the model on a much more complex dataset. As said previously, this dataset, which is computed using real observations of Earth's weather system, will be used to represent the model learning a very complex system with many unknown contributing variables. For our case, this means the data used will be limited to the 2-meter temperature. The anomaly is computed by subtracting the climatology from the corresponding day of years and hours of the atmospheric state. For example, the temperature on the 6^{th} of January 1998 at 18:00 UTC is subtracted with the average temperature for the 6^{th} day of the year at 18:00 hour UTC. For any anomaly predictions, the corresponding atmospheric state can simply be computed as the inverse process described before by adding the climatology to the predicted anomaly.

The preprocessing of the data is carried out in the same manner as the Burgers' equation experiments. The difference being that for the features we only include the current anomaly of the atmosphere and there are no forcing terms. The targets are the future anomaly of the atmosphere. The reason for choosing to train and predict the anomaly instead of the state valued themselves is to avoid the model from just learning the obvious periodicity that exists in weather such as the day-night cycle or the seasons. Next, Both the current anomaly and future anomaly are Fourier transformed to their 2-dimensional coefficients. This process is done for both the training and testing set. Once the features for both are processed, we fit the standard scaling function on the training features. Using this fitted scaling function, both the training and testing features are scaled to a mean of 0 and standard deviation of approximately 1.

The testing results are shown in tables 4.16 and 4.17. To put the absolute metrics into context, the median test state coefficient' maximum absolute value is $569\,364.25$. With this information, an RMSE value of 64.72 is relatively low error. The relative metric sMAPE also agrees with this, with a value of 0.76. The function values show that the model on average has an error of around 2 degrees kelvin or Celsius. This is again relatively small for the range of temperatures on Earth which span roughly 120 degrees kelvin across the historical maxima and minima. From existing knowledge of the problem, we know these values for the metrics are mainly due to the periodicity of atmospheric states. When we look at the anomaly metrics, the model is still able to learn some things and perform slightly better than a model of anomaly averages. However, since the median maximum absolute value for the state coefficients is 1155.14, the error

rate is not as good as the state predictions.

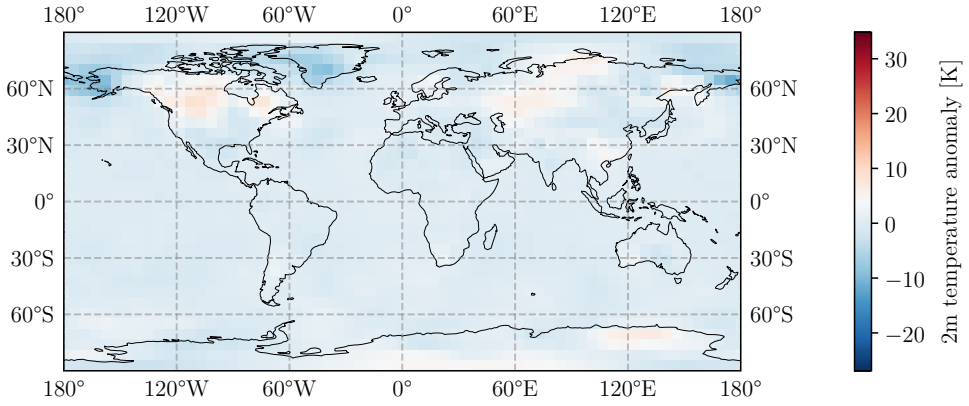
Table 4.16: Performance metrics of anomaly coefficient prediction for scenario 3.

	MSE	RMSE	MAE	R^2	sMAPE
spectral	4189.16	64.72	32.47	0.35	1.19
function value	4.09	2.02	1.44	0.09	1.24

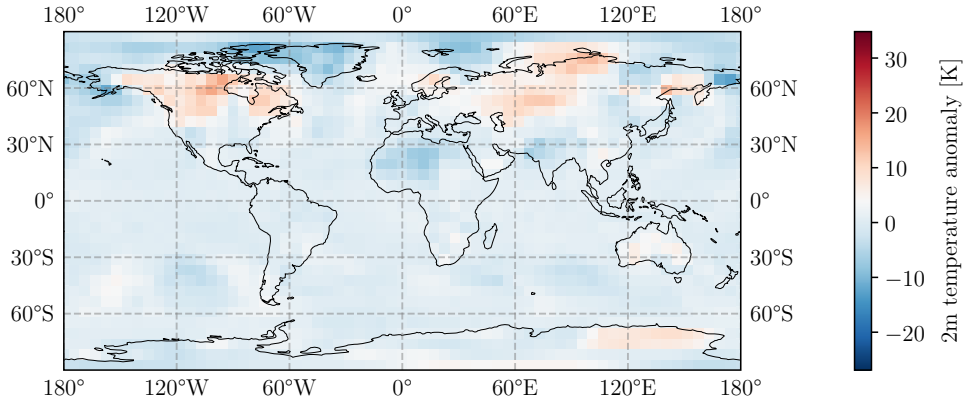
Table 4.17: Performance metrics of state coefficient prediction for scenario 3.

	MSE	RMSE	MAE	R^2	sMAPE
spectral	4189.16	64.72	32.47	0.56	0.74
function value	4.09	2.02	1.44	0.75	0.01

The predictions from test features are processed through the Fourier inverse transform to evaluate the function values. The direct predictions, which is the anomaly, is shown in figure 4.40. The model seem to be predicting less extreme values that are noticeable in the target. Specifically areas over North America, Central Asia, and parts of Antarctica show lower temperature anomaly than expected. We also loose detail in areas which should have a lower anomaly such as northern areas of Africa, Greenland, and South Asia. The lack of detail seems to be a problem of underestimation. We believe this is caused by the so-called "double penalty" issue caused by the loss function used for the LSSVR (Lledó et al., 2023). The predictions tend to be closer to the mean of the data to avoid the extra penalty from a wrong prediction with a high value. One way to allow for finer details that are less accurate is to penalize the higher frequency wave numbers less compared to the lower frequency wave numbers. For our case this means the LSSVR for each output will have their regularization parameter decreased to allow the model to focus less on the errors for higher frequencies.



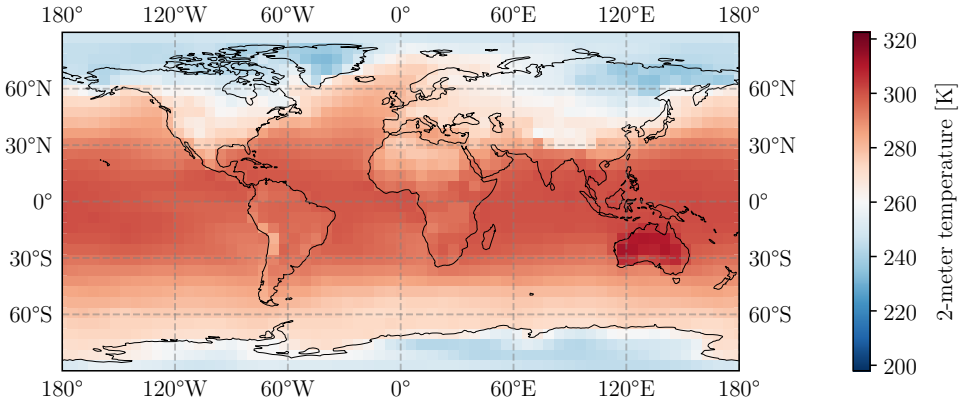
(a) Predicted 2-meter temperature anomaly.



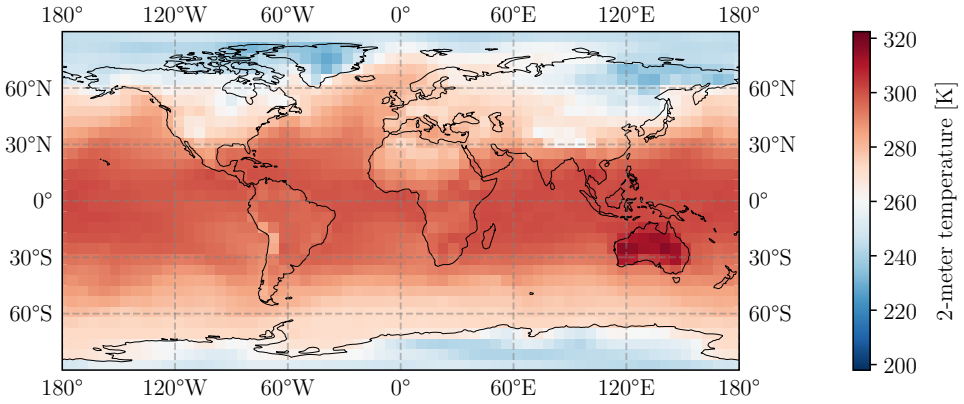
(b) Target 2-meter temperature anomaly.

Figure 4.40: 2-meter temperature anomaly prediction for the 1st of January 2020 at 6:00 UTC based on the previous anomaly at 0:00 UTC.

The state predictions which is obtained by adding the climatology to the predicted anomaly is shown in figure 4.41. This further shows areas of discrepancy between the prediction and the target. The area in the South Pacific show that the target has a higher difference temperature compared to the prediction. This smoothness in the prediction is easier to see for areas in the ocean. Many sharp features are lost and the state prediction start to resemble averaged view of the climatology. We must also take into account that the model only used temperature distribution data. Other pieces of information we know to be crucial to weather prediction such as wind speed or pressure was not included in the inputs. With this in consideration the model was still able to learn some relationships defined by governing equations of weather.



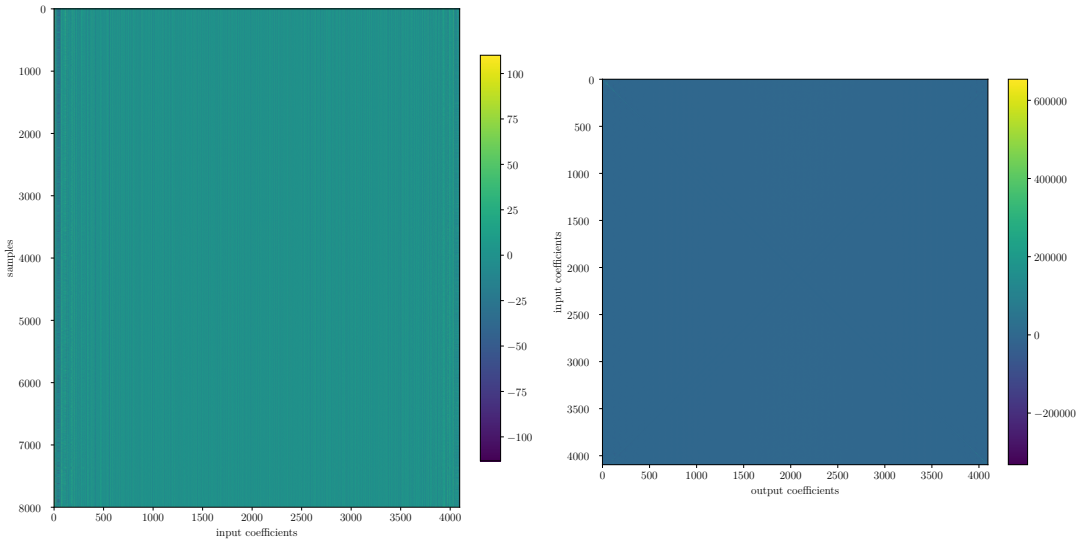
(a) Predicted 2-meter temperature.



(b) Target 2-meter temperature.

Figure 4.41: 2-meter temperature prediction for the 1st of January 2020 at 6:00 UTC based on the previous anomaly at 0:00 UTC.

The p-matrix and correlation image is shown in figure 4.42. For the correlation image, we see that most wave numbers are correlated or inversely correlated with the outputs. The strongest correlation is shown by the features of the same wave number as the sorted target. This is an indication that the equations involved may be nonlinear. If we look at the p-matrix however, it seems very flat save a faint diagonal line from the top left to the bottom right. The values of the p-matrix are also extremely high. With the "flatness" we see, this is indicative of strong contributions by a few inputs.



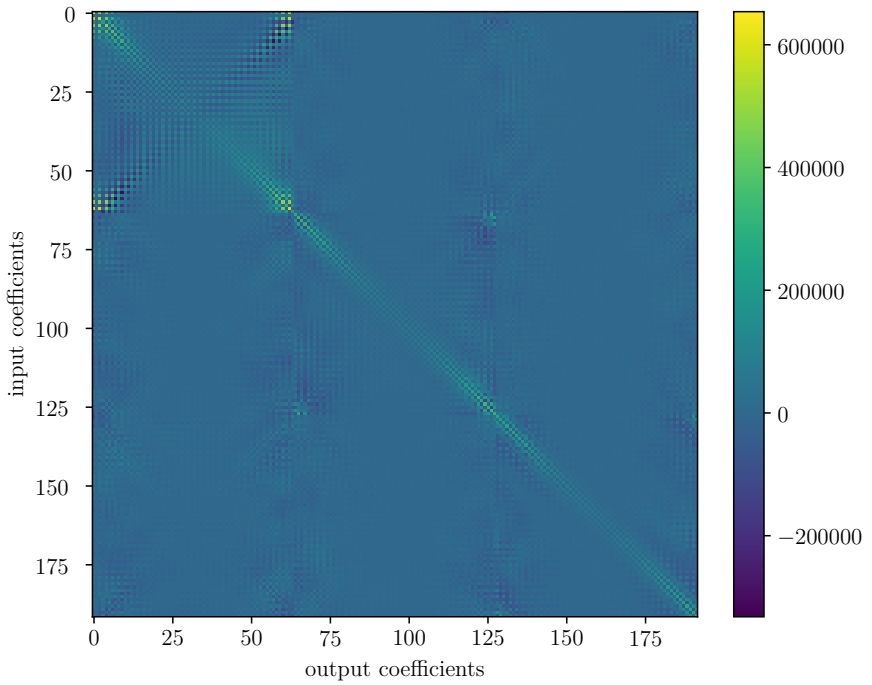
(a) The correlation image.

(b) The p-matrix.

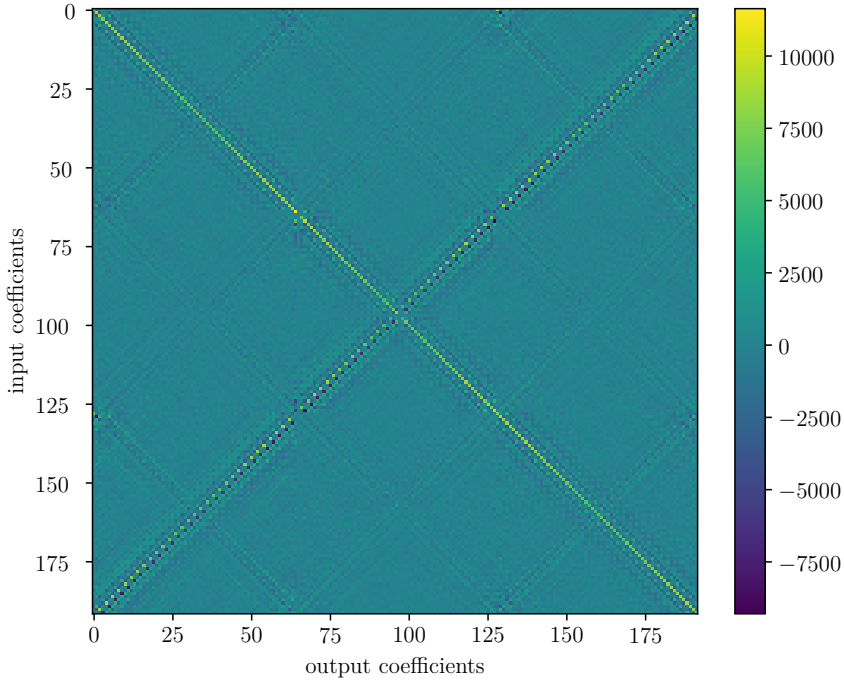
Figure 4.42: Correlation image and p-matrix for WeatherBench 2 model. The correlation image was sorted same order the values of the real component of wave number $(k_{longitude} = 0, k_{latitude} = 2)$ were sorted in descending order.

To see more details of the p-matrix, we show a zoomed in version in figures 4.43a and 4.43b. The relationships are much more clear between features and outputs that do not share the same wave number. The coefficients with the same longitude wave numbers appear in the same "square" shape. Within the "square", coefficients close to the same latitude wave number show much stronger contribution. For low wave numbers, shown in figure 4.43a, the contribution is spread over more latitude wave numbers. The higher wave numbers, on the other hand, has the contributions much more concentrated on the coefficients with the same latitude wave numbers. In addition, the value of contribution differ by an order of magnitude with lower longitude wave numbers having higher maximum contribution. This observation suggests that the nonlinearity and magnitude are inversely proportional to the absolute value of the longitude wave number. This is in contrast to when we discussed the forcing term for our Burgers' equation scenario. The low frequencies in longitude may indicate that the model has learned some feature of the anomaly that is specific to different times of day. This is because the lowest longitudinal wave number would correspond to when different parts of the world is in nighttime or in daytime. Other possible explanations may be due to the fact that the earth more closely resemble a sphere compared to the toroidal shape that the 2-dimensional Fourier transform assumes. This would result

in the representation latitude-wise being distorted. For this issue, the use of spherical harmonics may be a better fit to rule out the possibility of any distortions the Fourier transform may introduce.



(a) The p-matrix zoomed in to $(k_{longitude} = \{0, 1, 2\}, k_{latitude} = [-16, 16))$.



(b) The p-matrix zoomed in to $(k_{longitude} = \{\ \ 31, -32, \ \ 32\}, k_{latitude} = [-16, 16))$.

Figure 4.43: The p-matrix of the WeatherBench 2 models zoomed in to specific wave numbers.

The results of our experiments show that the proposed model has the ability to learn the relationships that PDEs and other governing equations define. The interpretation methods for support vector regression that we used, namely the correlation image and p-matrix developed by Üstün et al. (2007), allows us to confirm what the model has learned. In the case of the WeatherBench2 model, it has also allowed us to glean some insight into some mechanisms that the model has learned. This information was used to verify the learned relationships for the simple derivative equation. Together with the performance metrics on three different problems, we have shown the proposed model to be a viable tool for situations with limited knowledge and a need for some level of interpretability of the model.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

5.1 Conclusion

Based on the results of the research into approximating solutions of governing equations with fourier transform and support vector machine, several conclusions can be drawn.

1. The proposed SpectralSVR model is able to learn the relationships defined by PDEs. This is done with a computational model that with four phases which are data preparation, data transformation, model training, and finally model evaluation.

2. Interpretation of the proposed model can be done to a certain extent using two tools which are the correlation image and p-matrix developed by Üstün et al. (2007).

3. The proposed model is also capable in dealing with noise and partial data as shown in scenario 1 and 3.

4. The model's capabilities are also verified using exact solution. This revealed that for some configurations the model is able to successfully generalize to exact solutions.

5.2 Future Work

During the work on this study, some limitations were imposed, and many new questions also went unanswered. Here, we would like to list some potential avenues for future work:

- The synthetic data used in scenarios 1 and 2 were generated using MMS. While mathematically correct, this data is not realistic. Incorporating more realistic data from traditional numerical methods or other sources could enhance the model's ability to predict more realistic solutions.

- Test how the auto-regression of weather compares to the target data.

- Compare inverse prediction results with traditional derivatives, such as the spectral method or FDM.
- Compare the proposed method with other methods in comparable situations.
- Measure and compare the efficiency (time) of computing solutions against traditional and machine learning-based methods.
- Add more basis functions, such as spherical harmonics or wavelet bases.
- Incorporate more regression models, specifically more support vector regression models, to continue utilizing the correlation image and p-matrix.
- Use better metrics for measuring performance, such as the "standard error of regression".
- Analyze the performance with different hyperparameters or data sizes and present the results in charts.
- Perform an analysis of the error by wave number.
- Solve the problem of double penalties that cause the model to favor smooth solutions (Brown et al., 2011; Srivastava et al., 2017). Try using stronger regularization (a higher regularization parameter) for higher wave numbers.
- Add support for sparse data by employing data assimilation techniques, such as the Lomb-Scargle periodogram (VanderPlas, 2018), and 4D-Var data assimilation (Park & Xu, 2013; Pu & Kalnay, 2018).
- Study the contribution patterns for different terms in a PDE using the p-matrix or other methods.

Bibliography

Aarts, L. P., & Van Der Veer, P. (2001). Neural Network Method for Solving Partial Differential Equations. *Neural Processing Letters*, *14*(3), 261–271. https://doi.org/10.1023/A:1012784129883

Abramowitz, M., & Stegun, I. A. (with Conference on mathematical tables, National science foundation, & Massachusetts institute of technology). (1972). *Handbook of mathematical functions: With formulas, graphs and mathematical tables [conference under the auspices of the National science foundation and the Massachussetts institute of technology]* (Unabridged, unaltered and corr. republ. of the 1964 ed). Dover publ.

Ahsan, M., Mahmud, M., Saha, P., Gupta, K., & Siddique, Z. (2021). Effect of Data Scaling Methods on Machine Learning Algorithms and Model Performance. *Technologies*, *9*(3), 52. https://doi.org/10.3390/technologies9030052

Alpaydın, E. (2020). *Introduction to machine learning* (Fourth edition). The MIT Press.

Anderson, J. D. (1992). Governing Equations of Fluid Dynamics. In J. F. Wendt (Ed.), *Computational Fluid Dynamics* (pp. 15–51). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-11350-9_2

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., Chauhan, G., Chourdia, A., Constable, W., Desmaison, A., DeVito, Z., Ellison, E., Feng, W., Gong, J., Gschwind, M., … Chintala, S. (2024). PyTorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. https://doi.org/10.1145/3620665.3640366

Anselmo, L., & Pardini, C. (2005). Computational methods for reentry trajectories and risk assessment. *Advances in Space Research*, *35*(7), 1343–1352. https://doi.org/10.1016/j.asr.2005.04.089

Anwar, M. R., Liu, D. L., Macadam, I., & Kelly, G. (2013). Adapting agriculture to climate change: A review. *Theoretical and Applied Climatology*, *113*(1–2), 225–245. https://doi.org/10.1007/s00704-012-0780-1

Astitha, M., & Nikolopoulos, E. (Eds.). (2023). *Extreme weather forecasting : State of the science, uncertainty and impacts*. Elsevier.
OCLC: 1347429101.

Baker, N., Alexander, F., Bremer, T., Hagberg, A., Kevrekidis, Y., Najm, H., Parashar, M., Patra, A., Sethian, J., Wild, S., Willcox, K., & Lee, S. (2019, February 10). *Workshop Report on Basic Research Needs for Scientific Machine Learning: Core Technologies for Artificial Intelligence* (None, 1478744). https://doi.org/10.2172/1478744

Banks, J., Hittinger, J., Connors, J., & Woodward, C. (2012). Numerical error estimation for nonlinear hyperbolic PDEs via nonlinear error transport. *Computer Methods in Applied Mechanics and Engineering*, *213–216*, 1–15. https://doi.org/10.1016/j.cma.2011.11.021

Barter, G. E. (2008). *Shock capturing with PDE-based artificial viscosity for an adaptive, higher-order discontinuous Galerkin finite element method* [Doctoral dissertation, Massachusetts Institute of Technology]. Retrieved November 9, 2024, from http://dspace.mit.edu/handle/1721.1/7582

Basir, S., & Senocak, I. (2022). Critical Investigation of Failure Modes in Physics-informed Neural Networks. *AIAA SCITECH 2022 Forum*. https://doi.org/10.2514/6.2022-2353

Bec, J., & Khanin, K. (2007). Burgers turbulence. *Physics Reports*, *447*(1–2), 1–66. https://doi.org/10.1016/j.physrep.2007.04.002

Ben-Hur, A., & Weston, J. (2010). A User's Guide to Support Vector Machines. In O. Carugo & F. Eisenhaber (Eds.), *Data Mining Techniques for the Life Sciences* (pp. 223–239, Vol. 609). Humana Press. https://doi.org/10.1007/978-1-60327-241-4_13

Benton, E. R., & Platzman, G. W. (1972). A table of solutions of the one-dimensional Burgers equation. *Quarterly of Applied Mathematics*, *30*(2), 195–212. https://doi.org/10.1090/qam/306736

Berliner, L. M. (2003). Physical-statistical modeling in geophysics. *Journal of Geophysical Research: Atmospheres*, *108*(D24), 2002JD002865. https://doi.org/10.1029/2002JD002865

Bernardi, M., Sánchez, H. D., Sheth, R. K., Brownstein, J. R., & Lane, R. R. (2022). Stellar population analysis of MaNGA early-type galaxies: IMF dependence and

systematic effects. *Monthly Notices of the Royal Astronomical Society*, *518*(3), 4713–4733. https://doi.org/10.1093/mnras/stac3287

Bi, K., Xie, L., Zhang, H., Chen, X., Gu, X., & Tian, Q. (2023). Accurate medium-range global weather forecasting with 3D neural networks. *Nature*, *619*(7970), 533–538. https://doi.org/10.1038/s41586-023-06185-3

Bittner, K., & Spence, I. (2006). *Managing Iterative Software Development Projects*. Pearson Education, Limited.
OCLC: 1348491270.

Bonev, B., Kurth, T., Hundt, C., Pathak, J., Baust, M., Kashinath, K., & Anandkumar, A. (2023, July 23–29). Spherical Fourier neural operators: Learning stable dynamics on the sphere. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, & J. Scarlett (Eds.), *Proceedings of the 40th international conference on machine learning* (pp. 2806–2823, Vol. 202). PMLR. https://proceedings.mlr.press/v202/bonev23a.html

Bonkile, M. P., Awasthi, A., Lakshmi, C., Mukundan, V., & Aswin, V. S. (2018). A systematic literature review of Burgers' equation with recent advances. *Pramana*, *90*(6), 69. https://doi.org/10.1007/s12043-018-1559-4

Botchkarev, A. (2019). A New Typology Design of Performance Metrics to Measure Errors in Machine Learning Regression Algorithms. *Interdisciplinary Journal of Information, Knowledge, and Management*, *14*, 045–076. https://doi.org/10.28945/4184

Boullé, N., & Townsend, A. (2024). A mathematical guide to operator learning. In *Handbook of Numerical Analysis* (pp. 83–125, Vol. 25). Elsevier. https://doi.org/10.1016/bs.hna.2024.05.003

Boyd, J. P. (2001). *Chebyshev and Fourier spectral methods* (1. publ., rev. and enlarged 2. ed. of the work by Springer, Berlin, 1969). Dover Publ.

Brauer, F., Castillo-Chávez, C., & Feng, Z. (2019). *Mathematical models in epidemiology*. Springer.

Braun, M. (1993). *Differential Equations and Their Applications: An Introduction to Applied Mathematics* (Fourth Edition). Springer. https://doi.org/10.1007/978-1-4612-4360-1

Brezis, H., & Browder, F. (1998). Partial Differential Equations in the 20th Century. *Advances in Mathematics*, *135*(1), 76–144. https://doi.org/10.1006/aima.1997.1713

Brown, B. G., Gilleland, E., & Ebert, E. E. (2011, December 16). Forecasts of Spatial Fields. In I. T. Jolliffe & D. B. Stephenson (Eds.), *Forecast Verification* (1st ed., pp. 95–117). Wiley. https://doi.org/10.1002/9781119960003.ch6

Buoni, M., & Petzold, L. (2007). An efficient, scalable numerical algorithm for the simulation of electrochemical systems on irregular domains. *Journal of Computational Physics*, *225*(2), 2320–2332. https://doi.org/10.1016/j.jcp.2007.03.025

C3S. (2018). *ERA5 hourly data on single levels from 1940 to present*. https://doi.org/10.24381/CDS.ADBB2D47

Canuto, C., Hussaini, M. Y., & Quarteroni, A. (2007). *Spectral Methods: Evolution to Complex Geometries and Applications to Fluid Dynamics*. Springer e-books.

Chen, J.-S., Hillman, M., & Chi, S.-W. (2017). Meshfree Methods: Progress Made after 20 Years. *Journal of Engineering Mechanics*, *143*(4), 04017001. https://doi.org/10.1061/(ASCE)EM.1943-7889.0001176

Chen, R. T. Q. (2021, June). *Torchdiffeq* (Version 0.2.2). https://github.com/rtqichen/torchdiffeq

Chen, T., & Chen, H. (1995). Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *IEEE Transactions on Neural Networks*, *6*(4), 911–917. https://doi.org/10.1109/72.392253

Chu, E., & George, A. (2000). *Inside the FFT black box: Serial and parallel fast Fourier transform algorithms*. CRC Press.

Climate change 2022: Impacts, adaptation and vulnerability : Working Group II contribution to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change. (2023). Cambridge University Press.
OCLC: 1391150208.

Cogato, A., Meggio, F., De Antoni Migliorati, M., & Marinello, F. (2019). Extreme Weather Events in Agriculture: A Systematic Review. *Sustainability*, *11*(9), 2547. https://doi.org/10.3390/su11092547

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, *2*(4), 303–314. https://doi.org/10.1007/BF02551274

Du, Y., Chalapathi, N., & Krishnapriyan, A. S. (2024). Neural spectral methods: Self-supervised learning in the spectral domain. *The Twelfth International*

Conference on Learning Representations. https://openreview.net/forum?id=2DbVeuoa6a

Eustace, S. (with The Poetry contributors). (2024, May 8). *Poetry: Python packaging and dependency management made easy* (Version 1.8.3). Retrieved December 9, 2024, from https://python-poetry.org

Fanaskov, V. S., & Oseledets, I. V. (2023). Spectral Neural Operators. *Doklady Mathematics*, *108*(S2), S226–S232. https://doi.org/10.1134/S1064562423701107

Florêncio, A., Gabriel Rudloff, & Madson Dias. (2020, December 10). *Lssvr* (Version 0.1.0). Retrieved July 9, 2023, from https://github.com/zealberth/lssvr

Galkin, I. A., Reinisch, B. W., Vesnin, A. M., Bilitza, D., Fridman, S., Habarulema, J. B., & Veliz, O. (2020). Assimilation of sparse continuous near-earth weather measurements by NECTAR model morphing. *Space weather : the international journal of research & applications*, *18*(11), e2020SW002463. https://doi.org/10.1029/2020SW002463

Gao, H., Sun, L., & Wang, J.-X. (2021a). PhyGeoNet: Physics-informed geometry-adaptive convolutional neural networks for solving parameterized steady-state PDEs on irregular domain. *Journal of Computational Physics*, *428*, 110079. https://doi.org/10.1016/j.jcp.2020.110079

Gao, H., Sun, L., & Wang, J.-X. (2021b). Super-resolution and denoising of fluid flow using physics-informed convolutional neural networks without high-resolution labels. *Physics of Fluids*, *33*(7), 073603. https://doi.org/10.1063/5.0054312

Gaul, L., Klein, P., & Plenge, M. (1991). Simulation of wave propagation in irregular soil domains by BEM and associated small scale experiments. *Engineering Analysis with Boundary Elements*, *8*(4), 200–205. https://doi.org/10.1016/0955-7997(91)90014-K

GCSFS — GCSFs 2023.12.2post1+1.g8e500c6.dirty documentation. (n.d.). Retrieved November 26, 2024, from https://gcsfs.readthedocs.io/en/latest/

Getting Started on Kaggle | Kaggle. (n.d.). Retrieved September 9, 2024, from https://www.kaggle.com/docs/notebooks#technical-specifications

Givoni, M. (2006). Development and Impact of the Modern High-speed Train: A Review. *Transport Reviews*, *26*(5), 593–611. https://doi.org/10.1080/01441640600589319

Gong, X., Herty, M., Piccoli, B., & Visconti, G. (2023). Crowd Dynamics: Modeling and Control of Multiagent Systems. *Annual Review of Control, Robotics, and Autonomous Systems*, *6*(1), 261–282. https://doi.org/10.1146/annurev-control-060822-123629

Gopalsamy, K. (1992). *Stability and oscillations in delay differential equations of population dynamics*. Springer Science+Business Media, B.V.

Govind, R., Garg, N., & Mittal, V. (2020). Weather, Affect, and Preference for Hedonic Products: The Moderating Role of Gender. *Journal of Marketing Research*, *57*(4), 717–738. https://doi.org/10.1177/0022243720925764

Gregor, K., Danihelka, I., Mnih, A., Blundell, C., & Wierstra, D. (2014, June 22–24). Deep AutoRegressive networks. In E. P. Xing & T. Jebara (Eds.), *Proceedings of the 31st international conference on machine learning* (pp. 1242–1250, Vol. 32). PMLR. https://proceedings.mlr.press/v32/gregor14.html

Groch, M. W. (1998). Radioactive decay. *RadioGraphics*, *18*(5), 1247–1256. https://doi.org/10.1148/radiographics.18.5.9747617

Haifeng Wang & Dejin Hu. (2005). Comparison of SVM and LS-SVM for Regression. *2005 International Conference on Neural Networks and Brain*, *1*, 279–283. https://doi.org/10.1109/ICNNB.2005.1614615

Herrera, M., Natarajan, S., Coley, D. A., Kershaw, T., Ramallo-González, A. P., Eames, M., Fosas, D., & Wood, M. (2017). A review of current and future weather data for building simulation. *Building Services Engineering Research and Technology*, *38*(5), 602–627. https://doi.org/10.1177/0143624417705937

Holmes, E. E., Lewis, M. A., Banks, J. E., & Veit, R. R. (1994). Partial Differential Equations in Ecology: Spatial Interactions and Population Dynamics. *Ecology*, *75*(1), 17–29. https://doi.org/10.2307/1939378

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, *2*(5), 359–366. https://doi.org/10.1016/0893-6080(89)90020-8

Hoyer, S., & Hamman, J. (2017). Xarray: N-D labeled arrays and datasets in Python. *Journal of Open Research Software*, *5*(1). https://doi.org/10.5334/jors.148

Hoyer, S., Roos, M., Joseph, H., Magin, J., Cherian, D., Fitzgerald, C., Hauser, M., Fujii, K., Maussion, F., Imperiale, G., Clark, S., Kleeman, A., Nicholas, T., Kluyver, T., Westling, J., Munroe, J., Amici, A., Barghini, A., Banihirwe, A., … Littlejohns,

O. (2024, September 11). *Xarray* (Version v2024.09.0). https://doi.org/10.5281/ZENODO.13750907

Huestis, S. P. (2002). Understanding the Origin and Meaning of the Radioactive Decay Equation. *Journal of Geoscience Education*, *50*(5), 524–527. https://doi.org/10.5408/1089-9995-50.5.524

Hughes, R. (2000). The flow of large crowds of pedestrians. *Mathematics and Computers in Simulation*, *53*(4–6), 367–370. https://doi.org/10.1016/S0378-4754(00)00228-7

Ian, B. (n.d.). *Zarr encoding attributes persist after slicing data, raising error on 'to_zarr' · Issue #5219 · pydata/xarray · GitHub*. Retrieved January 5, 2025, from https://github.com/pydata/xarray/issues/5219

Jameson, A. (2007). Energy Estimates for Nonlinear Conservation Laws with Applications to Solutions of the Burgers Equation and One-Dimensional Vicous Flow in a Shock Tube by Central Difference Schemes. *18th AIAA Computational Fluid Dynamics Conference*. https://doi.org/10.2514/6.2007-4620

Jetbrains. (2023). *Languages - The State of Developer Ecosystem in 2023 Infographic | JetBrains: Developer Tools for Professionals and Teams*. Retrieved January 9, 2025, from https://www.jetbrains.com/lp/devecosystem-2023/languages/

Jia, Y., Liu, K., & Zhang, X. S. (2024). Modulate stress distribution with bio-inspired irregular architected materials towards optimal tissue support. *Nature Communications*, *15*(1), 4072. https://doi.org/10.1038/s41467-024-47831-2

Jim Zemlin. (2022, September 12). *Welcoming PyTorch to the Linux Foundation - Linux Foundation*. Retrieved January 9, 2025, from https://www.linuxfoundation.org/blog/blog/welcoming-pytorch-to-the-linux-foundation

Jin, M., Wang, L., Ge, F., & Yan, J. (2023). Detecting the interaction between urban elements evolution with population dynamics model. *Scientific Reports*, *13*(1), 12367. https://doi.org/10.1038/s41598-023-38979-w

Kapoor, S., & Narayanan, A. (2023). Leakage and the reproducibility crisis in machine-learning-based science. *Patterns*, *4*(9), 100804. https://doi.org/10.1016/j.patter.2023.100804

Kassam, A.-K., & Trefethen, L. N. (2005). Fourth-Order Time-Stepping for Stiff PDEs. *SIAM Journal on Scientific Computing*, *26*(4), 1214–1233. https://doi.org/10.1137/S1064827502410633

Katok, A., Hasselblatt, B., & Mendoza, L. (2009). *Introduction to the modern theory of dynamical systems: With a supplement by Anatole Katok and Leonardo Mendoza* (1. paperback ed., 10. print). Cambridge Univ. Press.

Kaufman, S., Rosset, S., Perlich, C., & Stitelman, O. (2012). Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data*, *6*(4), 1–21. https://doi.org/10.1145/2382577.2382579

Kovachki, N., Li, Z., Liu, B., Azizzadenesheli, K., Bhattacharya, K., Stuart, A., & Anandkumar, A. (2023). Neural operator: Learning maps between function spaces with applications to pdes. *Journal of Machine Learning Research*, *24*(89), 1–97. http://jmlr.org/papers/v24/21-1524.html

Krishnapriyan, A., Gholami, A., Zhe, S., Kirby, R., & Mahoney, M. W. (2021). Characterizing possible failure modes in physics-informed neural networks. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, & J. W. Vaughan (Eds.), *Advances in neural information processing systems* (pp. 26548–26560, Vol. 34). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2021/file/df438e5206f31600e6ae4af72f2725f1-Paper.pdf

Kurth, T., Subramanian, S., Harrington, P., Pathak, J., Mardani, M., Hall, D., Miele, A., Kashinath, K., & Anandkumar, A. (2023). FourCastNet: Accelerating Global High-Resolution Weather Forecasting Using Adaptive Fourier Neural Operators. *Proceedings of the Platform for Advanced Scientific Computing Conference*, 1–11. https://doi.org/10.1145/3592979.3593412

Kwasniok, F. (2012). Data-based stochastic subgrid-scale parametrization: An approach using cluster-weighted modelling. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *370*(1962), 1061–1086. https://doi.org/10.1098/rsta.2011.0384

Lang, S., Alexe, M., Chantry, M., Dramsch, J., Pinault, F., Raoult, B., Clare, M. C. A., Lessig, C., Maier-Gerber, M., Magnusson, L., Bouallègue, Z. B., Nemesio, A. P., Dueben, P. D., Brown, A., Pappenberger, F., & Rabier, F. (2024, August 7). *AIFS – ECMWF's data-driven forecasting system*. arXiv: 2406.01465 [physics]. Retrieved August 24, 2024, from http://arxiv.org/abs/2406.01465

Larios, A. (2021, April 28). *MATH 934 – BURGERS EQUATION PROJECT* [handout]. Retrieved September 16, 2024, from https://www.math.unl.edu/~alarios2/courses/2017_spring_M934/documents/burgersProject.pdf

Leake, C., Johnston, H., Smith, L., & Mortari, D. (2019). Analytically Embedding Differential Equation Constraints into Least Squares Support Vector Machines Using the Theory of Functional Connections. *Machine Learning and Knowledge Extraction*, *1*(4), 1058–1083. https://doi.org/10.3390/make1040060

Lepage, S., & Morency, C. (2021). Impact of Weather, Activities, and Service Disruptions on Transportation Demand. *Transportation Research Record: Journal of the Transportation Research Board*, *2675*(1), 294–304. https://doi.org/10.1177/0361198120966326

Li, Z., Kovachki, N. B., Azizzadenesheli, K., liu, B., Bhattacharya, K., Stuart, A., & Anandkumar, A. (2021). Fourier neural operator for parametric partial differential equations. *International Conference on Learning Representations*. https://openreview.net/forum?id=c8P9NQVtmnO

Lledó, L., Thomas Haiden, Josef Schröttle, & Richard Forbes. (2023, January). *Scale-dependent verification of precipitation and cloudiness at ECMWF*. Scale-dependent verification of precipitation and cloudiness at ECMWF. Retrieved December 26, 2024, from https://www.ecmwf.int/en/newsletter/174/earth-system-science/scale-dependent-verification-precipitation-and-cloudiness

Lu, L., Jin, P., Pang, G., Zhang, Z., & Karniadakis, G. E. (2021). Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, *3*(3), 218–229. https://doi.org/10.1038/s42256-021-00302-5

Maccarrone, G., Morelli, G., & Spadaccini, S. (2021). GDP Forecasting: Machine Learning, Linear or Autoregression? *Frontiers in Artificial Intelligence*, *4*, 757864. https://doi.org/10.3389/frai.2021.757864

Malhi, G. S., Kaur, M., & Kaushik, P. (2021). Impact of Climate Change on Agriculture and Its Mitigation Strategies: A Review. *Sustainability*, *13*(3), 1318. https://doi.org/10.3390/su13031318

Martin, J. E. (2014). *Introduction to weather and climate science*. Cognella Inc. OCLC: 880693165.

MathWorks. (2024). *Discrete Fourier Transform*. Retrieved December 11, 2024, from https://www.mathworks.com/help/signal/ug/discrete-fourier-transform.html

Matloff, N. (2017, September 19). *Statistical Regression and Classification: From Linear Models to Machine Learning* (1st ed.). Chapman and Hall/CRC. https://doi.org/10.1201/9781315119588

Mehrkanoon, S., & Suykens, J. A. (2015). Learning solutions to partial differential equations using LS-SVM. *Neurocomputing*, *159*, 105–116. https://doi.org/10.1016/j.neucom.2015.02.013

Mengaldo, G., Wyszogrodzki, A., Diamantakis, M., Lock, S.-J., Giraldo, F. X., & Wedi, N. P. (2019). Current and Emerging Time-Integration Strategies in Global Numerical Weather and Climate Prediction. *Archives of Computational Methods in Engineering*, *26*(3), 663–684. https://doi.org/10.1007/s11831-018-9261-8

Meuris, B., Qadeer, S., & Stinis, P. (2023). Machine-learning-based spectral methods for partial differential equations. *Scientific Reports*, *13*(1), 1739. https://doi.org/10.1038/s41598-022-26602-3

Monmonier, M. S. (1999). *Air apparent: How meteorologists learned to map, predict, and dramatize weather*. Univ. of Chicago Press.

Moon, S., Kang, M. Y., Bae, Y. H., & Bodkin, C. D. (2018). Weather sensitivity analysis on grocery shopping. *International Journal of Market Research*, *60*(4), 380–393. https://doi.org/10.1177/1470785317751614

Mukherjee, S., Goswami, D., & Chatterjee, S. (2015). A Lagrangian Approach to Modeling and Analysis of a Crowd Dynamics. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, *45*(6), 865–876. https://doi.org/10.1109/TSMC.2015.2389763

Muller, A. P. O., Costa, J. C., Bom, C. R., Klatt, M., Faria, E. L., de Albuquerque, M. P., & de Albuquerque, M. P. (2023). Deep pre-trained FWI: Where supervised learning meets the physics-informed neural networks. *Geophysical Journal International*, *235*(1), 119–134. https://doi.org/10.1093/gji/ggad215

Nash, C. (1991, January). *The case for high speed rail* (Working Paper No. Working Paper 323). Institute of Transport Studies, University of Leeds / ARRAY(0x5581272c4498). Leeds, UK. https://eprints.whiterose.ac.uk/2236/

Nelsen, N. H., & Stuart, A. M. (2024). Operator Learning Using Random Features: A Tool for Scientific Computing. *SIAM Review*, *66*(3), 535–571. https://doi.org/10.1137/24M1648703

Ni, P., Sun, L., Yang, J., & Li, Y. (2022). Multi-End Physics-Informed Deep Learning for Seismic Response Estimation. *Sensors*, *22*(10), 3697. https://doi.org/10.3390/s22103697

Nurmi, P., Perrels, A., & Nurmi, V. (2013). Expected impacts and value of improvements in weather forecasting on the road transport sector. *Meteorological Applications*, *20*(2), 217–223. https://doi.org/10.1002/met.1399

Olver, P. J. (2014). *Introduction to Partial Differential Equations*. Springer International Publishing. https://doi.org/10.1007/978-3-319-02099-0

Orlandi, P. (2000). The Burgers equation. In P. Orlandi (Ed.), *Fluid Flow Phenomena* (pp. 40–50, Vol. 55). Springer Netherlands. https://doi.org/10.1007/978-94-011-4281-6_4

Orszag, S. A. (1971). On the Elimination of Aliasing in Finite-Difference Schemes by Filtering High-Wavenumber Components. *Journal of the Atmospheric Sciences*, *28*(6), 1074–1074. https://doi.org/10.1175/1520-0469(1971)028<1074:OTEOAI>2.0.CO;2

Orszag, S. A. (1972). Comparison of Pseudospectral and Spectral Approximation. *Studies in Applied Mathematics*, *51*(3), 253–259. https://doi.org/10.1002/sapm1972513253

Park, S. K., & Xu, L. (Eds.). (2013). *Data Assimilation for Atmospheric, Oceanic and Hydrologic Applications (Vol. II)*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-35088-7

Preston-Werner, T. (2023, December 19). *Semantic Versioning 2.0.0*. Semantic Versioning. Retrieved September 9, 2024, from https://semver.org/

Pu, Z., & Kalnay, E. (2018). Numerical Weather Prediction Basics: Models, Numerical Methods, and Data Assimilation. In Q. Duan, F. Pappenberger, J. Thielen, A. Wood, H. L. Cloke, & J. C. Schaake (Eds.), *Handbook of Hydrometeorological Ensemble Forecasting* (pp. 1–31). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-40457-3_11-1

Raissi, M., Perdikaris, P., & Karniadakis, G. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, *378*, 686–707. https://doi.org/10.1016/j.jcp.2018.10.045

Rasp, S., Hoyer, S., Merose, A., Langmore, I., Battaglia, P., Russel, T., Sanchez-Gonzalez, A., Yang, V., Carver, R., Agrawal, S., Chantry, M.,

Bouallegue, Z. B., Dueben, P., Bromberg, C., Sisk, J., Barrington, L., Bell, A., & Sha, F. (2023). WeatherBench 2: A benchmark for the next generation of data-driven global weather models.

Rathore, P., Lei, W., Frangella, Z., Lu, L., & Udell, M. (2024). *Challenges in Training PINNs: A Loss Landscape Perspective* (2). https://doi.org/10.48550/ARXIV.2402.01868

Reinhardt, J. C., Chen, X., Liu, W., Manchev, P., & Paté-Cornell, M. E. (2016). Asteroid Risk Assessment: A Probabilistic Approach. *Risk Analysis*, *36*(2), 244–261. https://doi.org/10.1111/risa.12453

Roache, P. J. (2002). Code Verification by the Method of Manufactured Solutions. *Journal of Fluids Engineering*, *124*(1), 4–10. https://doi.org/10.1115/1.1436090

Roberts, M., & Bowman, J. C. (2011). Dealiased convolutions for pseudospectral simulations. *Journal of Physics: Conference Series*, *318*(7), 072037. https://doi.org/10.1088/1742-6596/318/7/072037

Rossum, G. van. (1998, January 6–8). *Glue It All Together With Python | Python.org*. Retrieved January 9, 2025, from https://www.python.org/doc/essays/omg-darpa-mcc-position/

Rossum, G. V. (2009, January 20). *The History of Python: A Brief Timeline of Python*. The History of Python. Retrieved January 9, 2025, from https://python-history.blogspot.com/2009/01/brief-timeline-of-python.html

Sadiku, M. N. O., & Obiozor, C. N. (2000). A Simple Introduction to the Method of Lines. *International Journal of Electrical Engineering & Education*, *37*(3), 282–296. https://doi.org/10.7227/IJEEE.37.3.8

Salari, K., & Knupp, P. (2000, June 1). *Code Verification by the Method of Manufactured Solutions* (SAND2000-1444, 759450). Sandia National Laboratories. https://doi.org/10.2172/759450

Schiesser, W. E. (2012). *The Numerical Method of Lines: Integration of Partial Differential Equations*. Elsevier Science.

Selvadurai, A. P. S. (2000). *Partial Differential Equations in Mechanics 2: The Biharmonic Equation, Poisson's Equation*. Springer. https://doi.org/10.1007/978-3-662-09205-7

Seydaoğlu, M., Erdoğan, U., & Öziş, T. (2016). Numerical solution of Burgers' equation with high order splitting methods. *Journal of Computational and Applied Mathematics*, *291*, 410–421. https://doi.org/10.1016/j.cam.2015.04.021

Shen, J. (with Tang, T., & Wang, L.-L.). (2011). *Spectral Methods: Algorithms, Analysis and Applications*. Springer. https://doi.org/10.1007/978-3-540-71041-7

Shrestha, A., & Mahmood, A. (2019). Review of Deep Learning Algorithms and Architectures. *IEEE Access*, *7*, 53040–53065. https://doi.org/10.1109/ACCESS.2019.2912200

Smith, J. O. (2007). *Mathematics of the discrete Fourier transform (DFT): With audio applications* (2. ed). BookSurge.

Soydaner, D. (2020). A Comparison of Optimization Algorithms for Deep Learning. *International Journal of Pattern Recognition and Artificial Intelligence*, *34*(13), 2052013. https://doi.org/10.1142/S0218001420520138

Spanos, A. (2006). Where do statistical models come from? Revisiting the problem of specification. In *Institute of Mathematical Statistics Lecture Notes - Monograph Series* (pp. 98–119). Institute of Mathematical Statistics. https://doi.org/10.1214/074921706000000419

Srivastava, A., Valkov, L., Russell, C., Gutmann, M. U., & Sutton, C. (2017). VEEGAN: Reducing mode collapse in gans using implicit variational learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), *Advances in neural information processing systems* (Vol. 30). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/44a2e0804995faf8d2e3b084a1e2db1d-Paper.pdf

stackoverflow. (2024). *Technology | 2024 Stack Overflow Developer Survey*. Retrieved January 9, 2025, from https://survey.stackoverflow.co/2024/technology

Stott, P. A., Christidis, N., Otto, F. E. L., Sun, Y., Vanderlinden, J.-P., van Oldenborgh, G. J., Vautard, R., von Storch, H., Walton, P., Yiou, P., & Zwiers, F. W. (2016). Attribution of extreme weather and climate-related events. *WIREs Climate Change*, *7*(1), 23–41. https://doi.org/10.1002/wcc.380

Suykens, J. A. K. (2005). *Least squares support vector machines* (Repr). World Scientific.

Tabatabaei, A. H. A., Shakour, E., & Dehghan, M. (2007). Some implicit methods for the numerical solution of Burgers' equation. *Applied Mathematics and Computation*, *191*(2), 560–570. https://doi.org/10.1016/j.amc.2007.02.158

Tian, J., Zhang, Y., & Zhang, C. (2018). Predicting consumer variety-seeking through weather data analytics. *Electronic Commerce Research and Applications*, *28*, 194–207. https://doi.org/10.1016/j.elerap.2018.02.001

Tian, X., Cao, S., & Song, Y. (2021). The impact of weather on consumer behavior and retail performance: Evidence from a convenience store chain in China. *Journal of Retailing and Consumer Services*, *62*, 102583. https://doi.org/10.1016/j.jretconser.2021.102583

Turchin, P. (2001). Does population ecology have general laws? *Oikos*, *94*(1), 17–26. https://doi.org/10.1034/j.1600-0706.2001.11310.x

Üstün, B., Melssen, W., & Buydens, L. (2007). Visualisation and interpretation of Support Vector Regression models. *Analytica Chimica Acta*, *595*(1–2), 299–309. https://doi.org/10.1016/j.aca.2007.03.023

Uzan, J.-P. (2003). The fundamental constants and their variation: Observational and theoretical status. *Reviews of Modern Physics*, *75*(2), 403–455. https://doi.org/10.1103/RevModPhys.75.403

VanderPlas, J. T. (2018). Understanding the Lomb–Scargle Periodogram. *The Astrophysical Journal Supplement Series*, *236*(1), 16. https://doi.org/10.3847/1538-4365/aab766

Vapnik, V. N. (2000). *The Nature of Statistical Learning Theory* (2nd ed). Springer New York. https://doi.org/10.1007/978-1-4757-3264-1

Vapnik, V. N., & Chervonenkis, A. Y. (1964). On a perceptron class. *Avtomatika i Telemkhanika*, *25*(1), 112–120.

Vapnik, V. N., & Lerner, A. Y. (1963). Recognition of patterns with help of generalized portraits. *Avtomat. i Telemekh*, *24*(6), 774–780.

Vedovoto, J. M., Silveira Neto, A. D., Mura, A., & Figueira Da Silva, L. F. (2011). Application of the method of manufactured solutions to the verification of a pressure-based finite-volume numerical scheme. *Computers & Fluids*, *51*(1), 85–99. https://doi.org/10.1016/j.compfluid.2011.07.014

Wang, R., Kashinath, K., Mustafa, M., Albert, A., & Yu, R. (2020). Towards Physics-informed Deep Learning for Turbulent Flow Prediction. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 1457–1466. https://doi.org/10.1145/3394486.3403198

Wang, X., & Zhang, W. (2019). Efficiency and Spatial Equity Impacts of High-Speed Rail on the Central Plains Economic Region of China. *Sustainability*, *11*(9), 2583. https://doi.org/10.3390/su11092583

Wazwaz, A.-M. (2010). *Partial Differential Equations and Solitary Waves Theory*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-00251-9

Wilby, R. L., & Yu, D. (2013). Rainfall and temperature estimation for a data sparse region. *Hydrology and Earth System Sciences*, *17*(10), 3937–3955. https://doi.org/10.5194/hess-17-3937-2013

Wood, W. L. (2006). An exact solution for Burger's equation. *Communications in Numerical Methods in Engineering*, *22*(7), 797–798. https://doi.org/10.1002/cnm.850

Youxi Wu, Ying Li, Lei Guo, Weili Yan, Xueqin Shen, & Kun Fu. (2005). SVM for Solving Forward Problems of EIT. *2005 IEEE Engineering in Medicine and Biology 27th Annual Conference*, 1559–1562. https://doi.org/10.1109/IEMBS.2005.1616732

Zhang, R., Liu, Y., & Sun, H. (2020). Physics-guided convolutional neural network (PhyCNN) for data-driven seismic response modeling. *Engineering Structures*, *215*, 110704. https://doi.org/10.1016/j.engstruct.2020.110704

Zhao, X., Gong, Z., Zhang, Y., Yao, W., & Chen, X. (2023). Physics-informed convolutional neural networks for temperature field prediction of heat source layout without labeled data. *Engineering Applications of Artificial Intelligence*, *117*, 105516. https://doi.org/10.1016/j.engappai.2022.105516

APPENDIX A

EXAMPLE COMPUTATION OF LSSVR

In this example we will be using the function $2x^2 + 4$. The values of this function can be seen in table 1.1.

No	x	y
1	0.0	4.0
2	0.33…	4.22…
3	0.66…	4.88…
4	1.0	6.0

Table 1.1: Example data of function $2x^2 + 4$

For $\Omega_{i,j} = K(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$

For example, with $\sigma = 1$, $x_i = 0.0$, & $x_j = 0.33\ldots$

$$\begin{aligned} K(0.0, 0.33) &= \exp\left(-\frac{\|0.0 - 0.33\|^2}{2(1)^2}\right) \\ &= \exp\left(-\frac{0.33^2}{2}\right) \\ &= 0.9460 \end{aligned} \tag{1.1}$$

$$\Omega \leftarrow \begin{bmatrix} 1.0000 & 0.9460 & 0.8007 & 0.6065 \\ 0.9460 & 1.0000 & 0.9460 & 0.8007 \\ 0.8007 & 0.9460 & 1.0000 & 0.9460 \\ 0.6065 & 0.8007 & 0.9460 & 1.0000 \end{bmatrix} \tag{1.2}$$

$$\mathbf{I}\frac{1}{C} \rightarrow \mathbf{I}\frac{1}{5} \rightarrow \begin{bmatrix} 0.2000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.2000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.2000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.2000 \end{bmatrix} \tag{1.3}$$

$$\Omega + \mathbf{I}\frac{1}{5} \rightarrow H \rightarrow \begin{bmatrix} 1.2000 & 0.9460 & 0.8007 & 0.6065 \\ 0.9460 & 1.2000 & 0.9460 & 0.8007 \\ 0.8007 & 0.9460 & 1.2000 & 0.9460 \\ 0.6065 & 0.8007 & 0.9460 & 1.2000 \end{bmatrix} \tag{1.4}$$

$$A \rightarrow \begin{bmatrix} 0.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 \\ 1.0000 & 1.2000 & 0.9460 & 0.8007 & 0.6065 \\ 1.0000 & 0.9460 & 1.2000 & 0.9460 & 0.8007 \\ 1.0000 & 0.8007 & 0.9460 & 1.2000 & 0.9460 \\ 1.0000 & 0.6065 & 0.8007 & 0.9460 & 1.2000 \end{bmatrix} \tag{1.5}$$

$$B \rightarrow \begin{bmatrix} 0.0000 \\ 4.0000 \\ 4.2222 \\ 4.8889 \\ 6.0000 \end{bmatrix} \tag{1.6}$$

$$A^{\dagger} \rightarrow \begin{bmatrix} -0.8994 & 0.4348 & 0.0652 & 0.0652 & 0.4348 \\ 0.4348 & 2.0686 & -1.6490 & -0.5292 & 0.1096 \\ 0.0652 & -1.6490 & 3.3774 & -1.1992 & -0.5292 \\ 0.0652 & -0.5292 & -1.1992 & 3.3774 & -1.6490 \\ 0.4348 & 0.1096 & -0.5292 & -1.6490 & 2.0686 \end{bmatrix} \tag{1.7}$$

$$A^{\dagger}B \rightarrow S \rightarrow \begin{bmatrix} 4.9421 \\ -0.6177 \\ -1.3737 \\ -0.5625 \\ 2.5538 \end{bmatrix} \tag{1.8}$$

$$b \to 4.9421 \tag{1.9}$$

$$\alpha \to \begin{bmatrix} -0.6177 \\ -1.3737 \\ -0.5625 \\ 2.5538 \end{bmatrix} \tag{1.10}$$

Prediction

$$U \to \begin{bmatrix} 0.3 \\ 0.2 \\ 0.5 \end{bmatrix} \tag{1.11}$$

$$\Omega \to \begin{bmatrix} 0.9560 & 0.9994 & 0.9350 & 0.7827 \\ 0.9802 & 0.9912 & 0.8968 & 0.7261 \\ 0.8825 & 0.9862 & 0.9862 & 0.8825 \end{bmatrix} \tag{1.12}$$

$$\Omega\alpha \to \begin{bmatrix} -0.4904 \\ -0.6170 \\ -0.2008 \end{bmatrix} \tag{1.13}$$

$$\Omega\alpha + b\mathbf{1}_m \to v \to \begin{bmatrix} 4.4516 \\ 4.3251 \\ 4.7413 \end{bmatrix} \tag{1.14}$$

Where $\mathbf{1}_m$ is a vector of 1s with the length of U.