Optimal trajectory calculation using neural networks

by

Sounak Majumder

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Applied Mathematics

Waterloo, Ontario, Canada, 2023

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Optimal control methods for linear systems have reached a substantial level of maturity, both in terms of conceptual understanding and scalable computational implementation. For non-linear systems, an open-loop feedback control may be calculated using Pontryagin's Maximum Principle. Alternatively, the Hamilton-Jacobi-Bellman (HJB) equation may be used to calculate the optimal control in a state-feedback form. However, it is an established fact that this equation becomes progressively harder to solve as the number of state variables increases. In this thesis, we discuss a Neural Network (NN)-based method [37] to approximate the solution to the HJB equation arising from high-dimensional ODE systems. We leverage the equivalency between the HJB equation and Pontryagin's Principle to generate the training and test datasets and define a physics-based loss function. The NN is then trained using a supervised optimization approach. We also examine an existing toolkit [29] to approximate the optimal control based on a power series expansion of the system around an equilibrium point in an infinite time horizon setting. We examine the possibility of incorporating this toolkit in the NN training procedure at different stages. The proposed methods are applied to three problems: optimal control of a 6 degree-of-freedom rigid body and the stabilization of ODE systems arising from the discretization of a Burgers'-like non-linear PDE and the damped wave equation.

Acknowledgements

Firstly, I would like to express my deepest appreciation to my supervisors, professors Kirsten Morris and Roberto Guglielmi. Their invaluable guidance and insightful discussions on the problems presented in this thesis were instrumental throughout my master's research. Certainly, this thesis would not have reached its final form without their valuable input.

I extend my heartfelt thanks to my undergraduate classmate, Mr. Rajdeep Sardar (Eco-friendly Smart Ship Parts Technology Innovation Center, Pusan National University, Busan, South Korea). I am equally grateful to Mr. Sankar Chakraborty and Mrs. Suman Chakraborty, who are (to date) the only persons I know personally in Canada. We hail from the same locality in India, and I am thankful for the homely refuge that they occasionally gave me thousands of miles away from my original home. During my master's degree, I pursued some extra professional ventures outside of University, such as completing the confirmatory examination program in mechanical engineering required for licensure as a professional engineer with Professional Engineers Ontario. Balancing graduate studies, teaching assignments, and exam preparation was exceptionally challenging, and I am truly thankful for the mental support that Mr. Sardar and Mr. and Mrs. Chakraborty gave me during those tough times. I would also like to thank Mr. Arnab Joardar (University of Waterloo), my flatmate, for his useful advice in those difficult times.

Finally, I am immensely thankful to Mr. Brian Mao, who was of significant help in navigating the co-op process of my master's program. With his support, I was able to get an engineering co-op job, which also happens to be the first industry job that I held. The experience I acquired proved invaluable to me in the early stages of my career.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction to Optimal Control

1.1 Introduction

Optimal Control problems involve the calculation of the admissible control to move a system at a given state to a final target set (an acceptable combination of the final time and system state) while minimizing a cost function along the trajectory. Its study has its roots in the field of Calculus of Variations, which, in turn, started development in the 1600s. Calculus of Variations deals with infinite-dimensional path optimization, but it is primarily concerned with problems in non-dynamic settings. It is generally agreed that the development of the Maximum Principle [40] and that of Dynamic Programming [4] were the most important milestones in the extension of the Calculus of Variations to a control-focused, dynamic setting. A summary of the development of Optimal Control theory and its application to aerospace problems may be found in [8].

In the context of linear systems, Linear Control theory has reached a substantial level of maturity. In the context of Optimal Control, for a quadratic optimization functional in terms of the states and control effort, the optimal controller may be obtained in a state-feedback form. This involves solving either the Differential or Algebraic Riccati Equation, depending on whether the problem is framed in a finite-time or infinite-horizon scenario [32][33]. While an analytical solution is often not possible as these equations are non-linear, the existence and properties of the solution may be guaranteed through relatively simple numerical tests; in fact, the application of concepts like the Maximum Principle is not required in this context. Scalable algorithms and computational routines exist to solve these equations even when the number of state variables is very large [5]. Consequently, methods based on linear control enjoy high confidence, especially in industrial settings [23].

Nonlinear systems may be linearized about an operating point to get an approximate linear system [23]. The nonlinear behavior will, however, become apparent if it operates over a wide range. Through better compensation of these nonlinearities, nonlinear controllers have an expanded range of operability as well as the potential to deliver better performance. Also, it has been shown that certain nonlinearities such as dead zones, hysteresis, coulomb friction, stiction, backlash, and saturation do not present good linear approximations [3]. [23] presents an example of disturbance rejection in a 6-dof manipulator tracking problem. The nonlinear sliding mode controller developed therein is seen to be significantly better and more robust than PID-based control. However, modeling nonlinear systems is mathematically more cumbersome, and its application to real-time systems requires significant computing power. The cost of nonlinear control prototyping hardware is high, and nonlinear control schemes potentially require more expensive and reliable actuators [23]. These limitations have led to their reduced adoption in industrial settings.

Pontryagin's Maximum Principle may be used to determine the optimal control for non-linear systems in an open-loop fashion. The optimal solution is then obtained in the form of a nonlinear boundary-value problem. Consequently, no feedback is solicited at any point in the trajectory to adjust the control effort. If a control adjustment is needed, fresh calculations need to be performed, rendering it unsuitable for real-time control applications. Alternatively, the optimal feedback can be obtained in a state-feedback form through the Hamilton-Jacobi-Bellman (HJB) equation [31]. It is known that these equations become progressively harder to solve as the number of state variables increases, a phenomenon dubbed the "curse of dimensionality". Existing strategies for high dimensional problems often rely on specific, restrictive problem structures or are valid only locally around some nominal trajectory [37].

This thesis aims to discuss the application of Neural Networks (NNs) to approximate the solution to HJB equations arising for general high-dimensional non-linear systems, as originally described in [37]. The training is performed using optimal trajectories calculated from Pontryagin's Maximum Principle. The interrelationships between PMP and HJB equations, as discussed in section 1.7, are used to improve the effectiveness of the training procedure. The trained NN is then used in the feedback control of the non-linear system. An approximate controller to predict the optimal control based on a power-series expansion [29], developed by Krener et al., is studied as well. Its potential application at different stages of the NN training procedure to potentially speed up or improve its convergence is examined subsequently.

This thesis is organized as follows: section 1.2 defines some basic terminologies commonly used in Optimal Control literature. The Pontryagin Maximum Principle (PMP)

is introduced in the context of *fixed time-free state* problems in section 1.3. A derivation of the HJB equation from the principle of optimality is presented in section 1.6. The interrelationship between the two, which forms the basis of the physics-based loss in the NN training process, is described in section 1.7. Chapter 2 describes the computational aspects of the problems discussed in this thesis. Section 2.2 discusses the arising boundary value problem and process-based concurrency techniques in Python, which were used in the data generation process. Sections 2.3 and 2.4 discuss the development of physics-informed neural networks and present an example of their application to solve a parabolic diffusion equation. The application of such concepts to our NN training procedure is discussed as well. Section 2.5 discusses the working of the toolkit developed by Krener et al. [29] in detail. The NN approximation method is applied to the optimal control of a 6-dof rigid body in chapter 3. A potential application of Krener's toolkit in the context of neural network pre-training and selectively choosing the training datapoints is discussed in the same chapter. Chapter 4 presents the application of the NN-based approximation method to the optimal stabilization of a dense ODE system of up to 30 dimensions arising from the discretization of a Burgers'-like PDE using a spectral method. Chapter 5 presents an optimal stabilization problem for an ODE system of up to 60 dimensions arising from the damped wave equation.

1.2 Basic Terminologies in Optimal Control Studies

This section introduces some terms commonly used in optimal control literature and discusses some central assumptions on the class of problems investigated in this thesis. In the most general sense, the dynamical systems examined are of the form $\dot{x} = f(t, x, u), x(t_0) = x_0$, where $x \in \mathbb{R}^n$ is the state, $u \in U \subseteq \mathbb{R}^m$ is the control set, $t \in \mathbb{R}$ is the time and x_0, t_0 is the initial state and time, respectively. Both $x = x(t)$ and $u = u(t)$ are functions of time; the control set U is a closed, time-invariant subset of $\mathbb{R}^m$. A set of constraints on f and admissible controls u must be specified so that for every possible initial condition (t_0, x_0) and every permissible control $u(t)$, the system has a unique solution $x(t)$ on a given time interval $[t_0, t_1]$. One such acceptable set of constraints as per [31, Chapter 3] requires that f is continuous in t and u and at least C^1 in x; furthermore, f_x must be continuous in t and u, and $u(.)$ must be a piecewise continuous function of t. In this thesis, it is assumed that the control systems satisfy this set of requirements. Additionally, the systems studied are time-invariant, such that the governing system f does not depend explicitly on time. One can treat time as an additional independent variable $x_{n+1} := t$ such that $\dot{x}_{n+1} = 1$, although doing so requires additional continuity restrictions on time.

Cost functionals describe the cost associated with the state transition, and these functionals are minimized in optimal control problems. Such functionals can be expressed in three forms, with the most general one being the *Bolza* form:

$$J(u) = \int_{t_0}^{t_f} L(t, x(t), u(t))dt + K(t_f, x_f) \tag{1.1}$$

where t_f and $x_f := x(t_f)$ are the final time and state, $L : \mathbb{R} \times \mathbb{R}^n \times U \to \mathbb{R}$ is the running cost (or Lagrangian), and $K : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$ is the terminal cost. The second form, called *Lagrangian* form involves no terminal cost, so that $K \equiv 0$, and the third form is called the *Mayer* form, in which the running cost $L \equiv 0$. It is possible to transform the cost functional from one form to another. Since

$$\begin{aligned} K(t_f, x_f) &= K(t_0, x_0) + \int_{t_0}^{t_f} \frac{d}{dt}(K(t, x(t))dt \\ &= K(t_0, x_0) + \int_{t_0}^{t_f} (K_t(t, x(t)) + K_x(t, x(t)))dt \end{aligned}$$

and as $K(t_0, x_0)$ is a constant, the Bolza functional is equivalent to the Lagrangian cost function with $\hat{L} = L(t, x(t), u(t)) + K_t(t, x(t)) + K_x(t, x(t))$. On the other hand, if L satisfies the same continuity properties as f discussed previously, one can introduce an extra state $x^0 : \dot{x^0} = L(t, x, u), x^0(t_0) = 0$. Thus $\int_{t_0}^{t_f} L(t, x(t), u(t))dt = x^0(t_f)$, implying a conversion to the Mayer form with terminal cost $\hat{K} = x^0(t_f) + K(t_f, x_f)$.

In order to define acceptable combinations of the final time t_f and the corresponding state x_f, the concept of target sets: $S \subset [t_0, \infty) \times \mathbb{R}^n$ are used. Let t_f be the smallest time such that $(t_f, x_f) \in S$. The set S is assumed to be closed so that if $(t, x(t))$ ever enters S, the time t_f is well defined, and an acceptable final state is reached. For example, the target set $S = [t_0, \infty) \times \{x_1\}$ where x_1 is a free point in $\mathbb{R}^n$ describes a *fixed-point, free-time problem*. One can have target sets such as $S = T \times S_1$, where T is some subset of $[0, \infty)$ and S_1 is some surface in $\mathbb{R}^n$. For example, $S = \{f(t, g(t)) : t \in [t_0, \infty)\}$ for some continuous function $g : \mathbb{R} \to \mathbb{R}^n$ corresponds to hitting a moving target.

This concludes a preliminary description of the problems discussed in optimal control: given a system f which meets the criteria described in the first paragraph and the target set S described before, to obtain the optimal control $u(.)$ that minimizes the cost functional eq. (1.1). In this thesis, *fixed-time, free-endpoint problems* are studied. Such problems are defined by the target set $S = \{t_f\} \times \mathbb{R}^n$. Additional constraints, based on a computational perspective, are described in chapter 2.

1.3 Pontryagin's Maximum Principle (PMP)

We first present the *Basic Variable-Endpoint Control Problem* as outlined in [31, Chapter 4] where the target set is $S = [t_0, \infty) \times S_1$ where S_1 is a k-dimensional surface in $\mathbb{R}^n$ for some non-negative constant $k \leq n$. Such a surface may be defined using equality constraints:

$$S_1 = \{x \in \mathbb{R}^n : h_1(x) = h_2(x) = h_3(x) = ... = h_{n-k}(x) = 0\}$$

where the h_i are C^1 functions from $\mathbb{R}^n$ to $\mathbb{R}$. The terminal cost K is assumed to be zero, and the system $f(x, u)$ and Lagrangian $L(x, u)$ are assumed to be time-independent. The optimality conditions are outlined by the following theorem:

Theorem 1.1. *Let $u^* : [t_0, t_f] \to U$ be an optimal control and let $x^* : [t_0, t_f] \to \mathbb{R}^n$ be the corresponding optimal state trajectory. Then there exists a function $p^* : [t_0, t_f] \to \mathbb{R}^n$ and a constant $p_0^* \leq 0$ satisfying $(p_0^*, p^*(t)) \neq (0, 0)$ for all $t \in [t_0, t_f]$ that has the following properties:*

1. *x^* and p^* satisfy the canonical equations:*

   $$\dot{x} = H_p(x^*, u^*, p^*, p_0^*) \text{ ,and } \dot{p} = -H_x(x^*, u^*, p^*, p_0^*)$$

   with the boundary conditions $x^*(t_0) = x_0$ and $x^*(t_f) \in S_1$, where the Hamiltonian $H : \mathbb{R}^n \times U \times \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$ is defined as

   $$H(x, u, p, p_0) = \langle p, f(x, u) \rangle + p_0 L(x, u)$$

2. *$H(x^*, u^*, p^*, p_0^*) \geq H(x^*, u, p^*, p_0^*)$ for all $t \in [t_0, t_f]$ and $u \in U$*

3. *$H(x^*, u^*, p^*, p_0^*) = 0$ for all $t \in [t_0, t_f]$*

4. *(Transversality condition): $p^*(t_f)$ is orthogonal to the tangent space to S_1 at $x^*(t_f)$:*

   $$\langle p^*(t_f), d \rangle = 0 \forall d \in T_{x^*(t_f)} S_1 \, ; T_{x^*(t_f)} S_1 = \{d \in \mathbb{R}^n : \langle \nabla h_i(x), d \rangle = 0, i = 1, 2, ..., n-k\}$$

Here p_0 is called an abnormal multiplier; it accounts for degenerate cases wherein it is zero, else p_0 is a negative scalar that can be used to normalize the costate vector $(p_0, p_i(t))$, such that $p_0 = -1$. This gives the standard definition of the Hamiltonian as $H = \langle p, f \rangle - L$. Such degenerate cases do not occur in the problems discussed in this thesis, so this normalization is assumed to be valid and $p_0 = -1$. The claim $H(x, u, p) = 0$ for all t is a special consequence of time-invariant problems when t_f is free.

The last condition implies that at t_f, the costate vector $p(t_f)$ can be written as a linear combination of the gradients $\nabla h_i(x^*(t_f))$, $i = 1, 2, ..., n-k$. Consequently if $S_1 = \mathbb{R}^n$, so that the tangent space is all $\mathbb{R}^n$, $p(t_f) = 0$. If, however, $S_1 = \{x_1\}$, it implies that the tangent space is zero, but then $x^*(t_f) = x_1$. Consequently, if f is a system of n ODEs, this basic problem is equivalent to a *boundary value problem* involving $2n$ equations: n for the state dynamics and n for the costate dynamics. If there are k surfaces in S_1, it implies k degrees-of-freedom for $x^*(t_f)$ and $(n-k)$ degrees of freedom for $p^*(t_f)$. The controls are obtained in an open-loop fashion; the solution to the BVP yields the optimal trajectory for a specific initial condition. At no point are errors or feedback metrics evaluated; and if there is a deviation from the initial conditions, fresh computations must be performed.

According to [31, Chapter 4], any other situation involving optimal control can be derived from this case, with appropriate modifications to the conditions. Since the objective of this thesis is to discuss a computational approach to approximate the HJB equation, a formal proof of the above theorem (which can be found in the same chapter) is not discussed. We just state the modified conditions for our *fixed-time, free endpoint problems* with some explanation of the modifications. The target set for such problems is given by $S = \{t_f\} \times \mathbb{R}^n$ and it has a non-zero terminal cost. Taking these facts into account, the following modified theorem is obtained:

Theorem 1.2. *Let $u^* : [t_0, t_f] \to U$ be an optimal control and let $x^* : [t_0, t_f] \to \mathbb{R}^n$ be the corresponding optimal state trajectory. Then there exists a function (called the co-state) $p^* : [t_0, t_f] \to \mathbb{R}^n$ and a constant $p_0^* \leq 0$ satisfying $(p_0^*, p^*(t)) \neq (0, 0)$ for all $t \in [t_0, t_f]$ that has the following properties:*

1. *x^* and p^* satisfy the canonical equations:*

$$\dot{x} = H_p(x^*, u^*, p^*, p_0^*) \text{ ,and } \dot{p} = -H_x(x^*, u^*, p^*, p_0^*)$$

   with the boundary conditions $x^*(t_0) = x_0$ and $x^*(t_f) \in S_1$. The second equation governing the evolution of the co-state is termed the adjoint equation. The Hamiltonian $H : \mathbb{R}^n \times U \times \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$ is defined as

$$H(x, u, p, p_0) = \langle p, f(x, u) \rangle + p_0 L(x, u)$$

2. *$H(x^*, u^*, p^*, p_0^*) \geq H(x^*, u, p^*, p_0^*)$ for all $t \in [t_0, t_f]$ and $u \in U$*
3. *$H(x^*, u^*, p^*, p_0^*)$ is constant for all $t \in [t_0, t_f]$.*
4. *For $x_f \in \mathbb{R}^n$, $p^*(t_f) = -K_x(t_f, x_f)$.*

Remark. *Statement 3 of theorem 1.2 differs from theorem 1.1 as even though* $H(x^*, u^*, p^*, p_0^*)$ *is constant, it cannot be guaranteed to be equal to zero. In [31, Chapter 4],* $\frac{d}{dt}H^*(t)$ *was shown to be equal to zero using topological arguments based on the cone generated from control perturbations along the optimal trajectory, which proves the constancy of* H^**.* $H^*(t_f)$ *is shown to be equal to zero for free-time problems through perturbations of the optimal time at which the target set is reached; this is not permissible if* t_f *is fixed.*

Remark. *The statement for theorem 1.2 is obtained by converting the functional from the Bolza form to the Lagrangian form, which effectively adds a term* $\langle K_x, f\rangle$ *to the running cost. The modified Hamiltonian may be written as* $-L+\langle \bar{p}-K_x, f\rangle$*, where* $\bar{p}$ *is the modified costate. One can consider* $p \equiv \bar{p} - K_x$*, but for this free endpoint problem,* $\bar{p}(t_f) = 0$*, by the transversality condition (theorem 1.1), which still holds from topological considerations in the terminal state. Thus* $p^*(t_f) = -K_x(t_f)$*. A more rigorous justification may be obtained in [31, Section 4.3.1].*

1.4 Dynamic Programming for discrete cases

The maximum principle, as outlined previously, provides the necessary conditions to determine the optimal trajectories to move from a given system state to another, assuming that the initial conditions and the target set are known and fixed. Dynamic programming, on the other hand, attempts to determine the optimal decision to be made at any possible state of the system by decomposing the problem into a sequence of relatively small problems and sequentially solving them, using relationships between the larger and smaller problems [4]. This information can be used to obtain the optimal control in a state-feedback form: an improvement over the maximum principle, as in the event of an incorrect decision, the altered control can be easily re-calculated. Sufficient conditions for optimal control can be obtained through this principle as well.

This concept may be illustrated using the following example from [9, Chapter 4] involving discrete possible states and available controls/paths. It is pictorially represented in fig. 1.1. The objective is to identify the optimal path to traverse from A to M.

The analysis is started from *state* M rather than A. Denote the optimal cost to move from a state S to M as $V(S)$. The obvious minimal costs are given in table 1.1: Subsequently, the minimum cost required to move from M to $\{E, F, G, H, I\}$ is identified. Since a particular state is connected directly to some state $\in \{J, K, M\}$, the cost equals the least possible sum of the cost of all paths available from the selected state to $J/K/L$ and the corresponding V. They are calculated in table 1.2: One can thus observe that V is

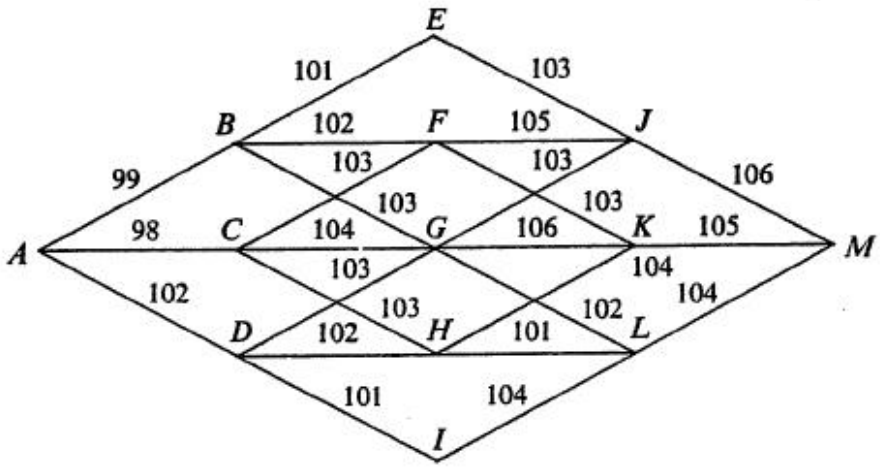


Figure 1.1: Possible paths from state A to M. The numbers are costs to travel between available states.

State	Cost	Path
J	$V(J) = 106$	$\{J, M\}$
K	$V(K) = 105$	$\{K, M\}$
L	$V(L) = 104$	$\{L, M\}$

Table 1.1: Optimal paths from M to $\{J, K, L\}$

a unique function between two points, and *no other path connecting two states can have a lower cost of traversal between them.* Thus, by the principle of optimality, for a given state like B, $V(B) = min\{V(X) + \text{Cost to go from } B \text{ to } X, X \in \{E, F, G, H, I\}\}$. The optimal route from B to M is therefore B to X and the subsequent optimal route from X to M. However, the latter has *already been calculated previously* by sequentially solving a class of relatively smaller problems. This illustrates the concept of dynamic programming. Continuing this procedure, the *net resultant optimal path from A to M* may be identified. This is completed in table 1.3: The optimal path from A to M is thus, $\{A, C, H, L, M\}$ with a total cost of $98 + 308 = 406$ units.

Note that in the process, the optimal path from *every state to state M* was identified, along with the associated costs. One can interpret this to be a form of state-dependent feedback since, given any possible state, the optimal path towards the target can be identified. For example, if there is an initial wrong decision and one moves from A to D rather

State	Cost	Path
E	$V(E) = 209$	$\{E, J, M\}$
F	$V(F) = 208$	$\{F, K, M\}$
G	$V(G) = 206$	$\{G, L, M\}$
H	$V(H) = 205$	$\{H, L, M\}$
I	$V(I) = 208$	$\{I, L, M\}$

Table 1.2: Optimal paths from M to $\{E, F, G, H, I\}$

State	Cost	Path
B	$V(B) = 309$	$\{B, G, K, M\}$
C	$V(C) = 308$	$\{C, H, L, M\}$
D	$V(D) = 307$	$\{D, H, L, M\}$

Table 1.3: Optimal paths from M to $\{B, C, D\}$

than C, the subsequent optimal path is $\{D, H, L, M\}$. No further calculations are needed.

1.5 The value function

The idea illustrated above can now be extended to continuous systems, using the concept of a value function. Let us consider a *fixed time, free-endpoint* problem with a target set $S = \{t_1\} \times \mathbb{R}^n$. Consider the following Bolza cost functional, with an explicit reference to the initial state and time:

$$J(t_0, x_0, u) = \int_{t_0}^{t_1} L(t, x(t), u(t))dt + K(x(t_1)). \tag{1.2}$$

Instead of just analyzing the specific functional $J(u, t_0, x_0)$, dynamic programming considers the *family of functionals*:

$$J(t, x, u) = \int_{t}^{t_1} L(s, x(s), u(s))ds + K(x(t_1)) \tag{1.3}$$

where, t ranges in $[t_0, t_1)$ and x ranges in $\mathbb{R}^n$, such that $x(t) = x$. By deriving a *dynamic relationship* in this family, one attempts to solve all of them. In this context the value function is defined as:

$$V(t, x) := \inf_{u_{[t,t_1]}} J(t, x, u) \tag{1.4}$$

such that V satisfies the boundary condition: $V(t_1, x) = K(x)$ for all $x \in \mathbb{R}^n$. This boundary condition arises due to our particular choice of the target set S. Had $S \subset [t_0, \infty) \times \mathbb{R}^n$, the boundary conditions would have been $V(t, x) = K(x)$ for all $(t, x) \in S$.

We propose that the value function necessarily satisfies the following relation for every $(t, x) \in [t_0, t_1) \times \mathbb{R}^n$ and for every $\Delta t \in (0, t_1 - t]$, where $u_{[t,t+\Delta t]}$ is the corresponding control, and the state $x(.)$ is the state trajectory that satisfies $x(t) = x$:

$$V(t, x) = \inf_{u \in [t, t+\Delta t]} \Big\{ \int_t^{t+\Delta t} L(s, x(s), u(s)) ds + V(t + \Delta t, x + \Delta x) \Big\}. \tag{1.5}$$

This claim can be understood from the *principle of optimality*: given an optimal path corresponding to $(x + \Delta x)$, if the state x corresponding to the infimum as expressed above for all permissible controls and small time interval Δt is found, then the augmented path connecting x to the target set will be optimal, with the resultant cost expressed as above. This approach is in alignment with the concept of dynamic programming, i.e., the value function corresponding to all $x \in \mathbb{R}^n$ at time t is identified from that of $V(t + \Delta t, x + \Delta x)$ (already known) and the controls corresponding to the associated transition. Recall that, in the discrete case, we moved from M to $\{J, K, L\}$ then, to $\{E, F, G, H, I\}$ and finally towards A using a similar approach for calculating V: by calculating the V-s for the earlier states in its path. A proof of this relation is presented in the next sections, along with the derivation of a PDE for its calculation.

Consider the following numerical example governed by the dynamics: $\dot{x} = xu$ with $x \in \mathbb{R}$, $t \in [t_0, t_1]$, $u \in U = [-1, 1]$ and the functional to minimize: $J(u) = x(t_1)$. For this case, the optimal control is found from simple inspection: $u = -1$ if $x_0 > 0$, $u = 1$ if $x_0 < 0$. Thus, the final state will be $x(t_1) = x_0 e^{-(t_1 - t_0)} \, if \, x_0 > 0$, and, $x(t_1) = x_0 e^{(t_1 - t_0)} \, if \, x_0 < 0$, and $x(t_1) = 0 \, if \, x_0 = 0$ (Regardless of control applied). The value function, from definition (1.3) is:

$$V(t, x) = \begin{cases} x e^{-(t_1 - t)} & \text{if } x > 0 \\ x e^{(t_1 - t)} & \text{if } x < 0 \\ 0 & \text{if } x = 0. \end{cases}$$

Indeed, let us consider a time t_0 at which the state $x_0 > 0$. Suppose that it is subject to a control $u = u(t) \in U$ for $t \in [t_0, t_0 + \Delta]$, so that $x(t_0 + \Delta) = x_0 e^{\int_{t_0}^{t_0 + \Delta} u(t) dt}$. The value function for $x(t_0 + \Delta)$ is then $x(t_0 + \Delta) e^{-\int_{t_0 + \Delta}^{t_1} dt} = x_0 e^{\int_{t_0}^{t_0 + \Delta} u(t) dt - \int_{t_0 + \Delta}^{t_1} dt}$ as the exponential is always positive. Since $L = 0$, the infinimum for this quantity is possible only when $u(t) = -1$, for which we recover $V(t_0, x_0) = x_0 e^{-(t_1 - t_0)}$ for $x_0 > 0$, as per our expectations from (1.5). Similar arguments may be used for $x_0 < 0$ as well.

1.6 Hamilton-Jacobi-Bellman (HJB) Equation

A rigorous justification of (1.5) is first presented. Let $\bar{V}(t,x)$ denote the RHS, so that:

$$\bar{V}(t,x) := \inf_{u\in[t,t+\Delta t]} \Big\{ \int_t^{t+\Delta t} L(s,x(s),u(s))ds + V(t+\Delta t, x+\Delta x) \Big\}.$$

We first prove that $V(t,x) \geq \bar{V}(t,x)$. By the definition of the infimum and (1.3), it is known that for every $\varepsilon > 0, \exists\, u_\varepsilon$ on $[t,t_1]$, such that:

$$\begin{aligned} V(t,x) + \varepsilon &\geq J(t,x,u_\varepsilon) \\ \text{or}, V(t,x) + \varepsilon &\geq \int_t^{t+\Delta t} L(s,x_\varepsilon(s),u_\varepsilon(s))ds + J(t+\Delta t, x_\varepsilon(t+\Delta t), u_\varepsilon) \\ \implies V(t,x) + \varepsilon &\geq \int_t^{t+\Delta t} L(s,x_\varepsilon(s),u_\varepsilon(s))ds + V(t+\Delta t, x_\varepsilon(t+\Delta t)) \geq \bar{V}(t,x). \end{aligned}$$

Since this inequality holds for all $\varepsilon > 0$, $V(t,x) \geq \bar{V}(t,x)$. To complete this proof, it must be additionally proved that $V(t,x) \leq \bar{V}(t,x)$. From the definition of (1.3), for any $u(t)$ not corresponding to the optimal cost function J as defined in (1.4):

$$V(t,x) \leq J(t,x,u) = \int_t^{t+\Delta t} L(t,x,u)dt + J(t+\Delta t, x+\Delta x, u).$$

Let us now consider $u(t)$ from the state $x+\Delta x$ (possibly deviated from the optimal trajectory) to be the optimal control corresponding to the infimum transition cost from $x+\Delta x$ to the target set. Then we have:

$$V(t,x) \leq \int_t^{t+\Delta t} L(t,x,u)dt + V(t+\Delta t, x+\Delta x).$$

This inequality must hold for any $\Delta t > 0$. Also, the RHS is dependent only on the control acting between $[t, t+\Delta t]$ since it will affect Δx, and the control for the remaining time interval is de-facto assumed to be equal to the control associated with the optimal transition from $x+\Delta x$ to the target set. Therefore, the infimum of the RHS must be greater than the LHS as well:

$$V(t,x) \leq \inf_{u_{[t,t+\Delta t]}} \Big\{ \int_t^{t+\Delta t} L(t,x,u)dt + V(t+\Delta t, x+\Delta x, u) \Big\} = \bar{V}(t,x).$$

Thus, relation (1.5), which is a *necessary condition*, is proved. This relation can now be reformulated in terms of a PDE. Consider the first order Taylor expansion for $V(t,x)$, assuming that the dynamics are governed by the usual expression $\dot{x} = f(t,x,u)$:

$$V(t+\Delta t, x+\Delta x) = V(t,x) + V_t(t,x)\Delta t + \langle V_x(t,x), f(t,x,u)\Delta t\rangle + o(\Delta t).$$

Furthermore, one also has:

$$\int_t^{t+\Delta t} L(s,x(s),u(s))ds = L(t,x(t),u(t))\Delta t + o(\Delta t).$$

Upon making the relevant substitutions to eq. (1.5), the following relation is obtained:

$$V(t,x) = \inf_{u_{[t,t+\Delta t]}} \big\{L(t,x(t),u(t))\Delta t + V(t,x) + V_t(t,x)\Delta t + \langle V_x(t,x), f(t,x,u)\Delta t\rangle + o(\Delta t)\big\}$$
$$\implies 0 = \inf_{u_{[t,t+\Delta t]}} \big\{L(t,x(t),u(t))\Delta t + V_t(t,x)\Delta t + \langle V_x(t,x), f(t,x,u)\Delta t\rangle + o(\Delta t)\big\}.$$

In this case, as the infimum is taken over the instantaneous value of u at a specific t, $V_t(t,x)$ can be pulled outside the infimum brackets since it does not depend on u. Since this result holds for any $\Delta t > 0$, if $\Delta t \to 0$ so that $o(\Delta t)/\Delta t \to 0$, we have the following result in the differential limit:

$$-V_t(t,x) = \inf_{u\in U}\big\{L(t,x,u) + \langle V_x(t,x), f(t,x,u)\rangle\big\}. \tag{1.6}$$

This PDE is the **Hamilton-Jacobi-Bellman (HJB) Equation**. Its boundary conditions need to be formulated according to a known target set. With respect to the fixed-time, free-endpoint problems discussed in this thesis, the boundary conditions are $V(t_1,x) = K(x)$ for all $x \in \mathbb{R}^n$. For the general target set, the condition will be $V(t,x) = K(x)$ for all $(t,x) \in S$.

Consider the standard example of a *free-time fixed endpoint* problem of bringing a body to rest at the minimum possible time, subject to the dynamics $\ddot{x} = u$. The functional $J(u) = \int_0^t dt$, and the state is denoted as $(x_1, x_2) \equiv (x, \dot{x})$. The target set is $[0,\infty) \times (0,0)$, and the control set $U = [-1,1]$. The value function satisfies:

$$-V_t(t,x_1,x_2) = \underset{u\in[-1,1]}{inf}\big\{1 + V_{x_1}(t,x_1,x_2)x_2 + V_{x_2}(t,x_1,x_2)u\big\}.$$

The boundary conditions must be $V(t,0,0) = 0$ for all $t \geq 0$. Evidently, this implies that the optimal control is of a *bang-bang type*, as expected from Pontryagin's Principle:

$$u = -sgn(V_{x_2}(t,x_1,x_2)) = \begin{cases} 1 & \text{if } V_{x_2} < 0 \\ -1 & \text{if } V_{x_2} > 0 \\ ? & \text{if } V_{x_2} = 0. \end{cases}$$

We can simplify the HJB equation obtained to:

$$-V_t(t, x_1, x_2) = 1 + V_{x_1}(t, x_1, x_2)x_2 - |V_{x_2}(t, x_1, x_2)|. \tag{1.7}$$

Section 1.7 discusses the interrelationship between the HJB equation and Pontryagin's Principle, under somewhat strong requirements on V. It also illustrates how the value function may be applied to establish *sufficiency requirements* and explains why the controls synthesized are in a state-feedback form.

1.7 Relationship between HJB and PMP

Equation 1.6 can be reformulated in the following way:

$$V_t(t, x) = \sup_{u \in U} \big\{ -L(t, x, u) - \langle V_x(t, x), f(t, x, u) \rangle \big\} \tag{1.8}$$

This suggests that if the optimal control u^* exists and *if all continuity assumptions are valid*, we can take the costate $p^* = -V_x^*(t, x^*)$, where the optimal state trajectory is denoted as x^*. The reformulated equation is equivalent to the *Hamiltonian Maximization Condition*:

$$\begin{aligned} &V_t^* = \sup_{u \in U} \big\{ H(t, x, -V_x(t, x), u) \big\} \\ &or, \big\{ H(t, x, -V_x(t, x), u) \big\} \leq \big\{ H(t, x^*, -V_x^*(t, x^*), u^*) \big\}. \end{aligned} \tag{1.9}$$

This Hamiltonian $H(.)$ is equivalent to the Hamiltonian function defined in statement 1 of theorem 1.1 and theorem 1.2; the maximization condition is equivalent to statement 2 of the same theorems. It is trivial to obtain the governing equation for the optimal trajectory by simply differentiating the optimal Hamiltonian w.r.t. p (equiv. to $-V_x(t, x)$ in our case). The adjoint equation for the co-state is easily verified as well:

$$\begin{aligned} -p_t^* &= V_{tx}(t, x^*) = V_{xt}(t, x^*) \\ \text{or}, -p_t^* &= \frac{\partial}{\partial x} H(t, x^*, p^*, u^*) = H_x(t, x^*, p^*, u^*) \end{aligned}$$

With respect to the boundary conditions as well $p^* = -V_x(t_f, x) = -K_x(t_f, x)$ for all $(t, x) \in S$. In other words, the conditions of the PMP for the fixed-time free-endpoint problem can be derived from the HJB equation [31, Chapter 4]. For the basic variable endpoint problem with a transversality condition (theorem 1.1), one would expect the

optimal V to be a local minimum at t_f, so that the component of V_x in the tangent space of the spatial surfaces in the target set is zero. Thus $\langle V_x(t_f), d\rangle = 0$ for all $d \in T_{x^*(t_f)}S_1 \,; T_{x^*(t_f)}S_1 = \{d \in \mathbb{R}^n : \langle \nabla h_i(x), d\rangle = 0, i = 1, 2, ..., n-k\}$. Using $p^*(t_f) = -V_x(t_f)$, the transversality condition is recovered.

In section 1.3, it was mentioned that the maximization principle provides an *open-loop* control specification: $u^*(t) = \arg\max_{u\in U} H(t, x^*, p^*, u)$. The optimal control is dependent on the costate p^*, and this needs to be calculated from the adjoint equation. On the other hand, the optimal control from the HJB equation is $u = \arg\max_{u\in U} H(t, x^*, -V_x^*(t, x^*), u)$. This is a *closed-loop* control synthesis in a feedback form since this decision depends only on the current state $x^*(t)$.

The *sufficiency conditions* are now established. Let us assume that the optimal control $u^* : [t_0, t_1] \to U$ exists and let $x^* : [t_0, t_1] \to \mathbb{R}^n$ denote the optimal trajectory. Suppose that $\hat{V} : [t_0, t_1] \times \mathbb{R}^n \to \mathbb{R}$ satisfies the Hamiltonian maximization requirement (1.9) with the boundary condition $\hat{V}(t_1, x) = K(x)$. Then, this value function satisfies:

$$
\begin{aligned}
&-\hat{V}_t(t, x^*(t)) = L(t, x^*(t), u^*(t)) + \langle \hat{V}_x(t, x^*(t), u^*(t)), f(t, x^*(t), u^*(t))\rangle \\
&\implies 0 = L(t, x^*(t), u^*(t)) + \frac{d}{dt}\hat{V}(t, x^*(t)) \\
&\implies 0 = \int_{t_0}^{t_1} L(t, x^*(t), u^*(t))dt - \hat{V}(t_0, x^*(t_0)) + \hat{V}(t_1, x^*(t_1)) \\
&\implies \hat{V}(t_0, x^*(t_0)) = \int_{t_0}^{t_1} L(t, x^*(t), u^*(t))dt + K(x^*(t_1)) \equiv J(t_0, x^*(t_0), u^*).
\end{aligned}
$$

On the other hand, for any other control u, with the corresponding state trajectory x, it is known that:

$$
\begin{aligned}
&-\hat{V}_t(t, x(t)) \leq L(t, x(t), u(t)) + \langle \hat{V}_x(t, x(t), u(t)), f(t, x(t), u(t))\rangle \\
&\implies 0 \leq L(t, x(t), u(t)) + \frac{d}{dt}\hat{V}(t, x(t)) \\
&\implies \hat{V}(t_0, x(t_0)) \leq \int_{t_0}^{t_1} L(t, x(t), u(t))dt + K(x(t_1)) \equiv J(t_0, x(t_0), u).
\end{aligned}
$$

Thus only u^* that corresponds to the Hamiltonian maximization requirement gives the optimal value of $\hat{V}(t_0, x_0)$ and this corresponds to the optimal cost $J(t_0, x^*(t_0), u^*)$; no other control can yield a lower value. Thus, the sufficiency conditions are proved. Also, observe that at since (t_0, x_0) was an arbitrary initial condition, $\hat{V}(t, x)$ would be the *globally optimal cost of transition* (upto a time t_1) for any arbitrary (t, x).

The above analysis showed that the maximum principle may be "derived" from the HJB equations, *provided certain continuity assumptions are valid*. The derivation of the maximum principle discussed above appears to be much simpler compared to that of [31, Chapter 4]. It should be noted that we at least needed $V \in C^1$ to derive the differential form of the HJB equation. Furthermore, in the derivation of the adjoint equation, V_x was required to be differentiable w.r.t time. From a theoretical perspective, these are rather strong assumptions. This is apparent from the numerical example discussed at the end of section 1.6 where the value function is Lipschitz but not necessarily C^1. The original derivation of the maximum principle, however, uses substantially weaker assumptions and, therefore, can be applied to much more general scenarios. The solution to the HJB equation (1.6) must be interpreted in a weaker sense; a systematic procedure in this context is given by *viscosity solutions*. Since we are mostly interested in a computational approach, we do not discuss them further. Some suitable references for viscosity solutions are [17] and [31, Chapter 5].

Chapter 2

Computational Approaches

2.1 General Overview

From a computational perspective, it is non-trivial to numerically solve the HJB equation, as eq. (1.6) is a first-order non-linear, non-conservative PDE with a non-smooth solution in t and the degrees of freedom of the system x. A grid-based method based on the ENO/WENO scheme (a generalized method to solve PDEs) has been developed in [41]. For grid-based methods, the storage requirements and computational costs increase intractably when the number of system variables increases, and the degree of accuracy of the numerical solution is tried to be kept constant. This was recognized when the HJB equation was first formulated and has become notorious as the "Curse of Dimensionality"[4]. This motivated the development of approximation methods and alternatives to grid-based methods to numerically solve the HJB equation.

A sparse grid-based method, in which the initial state variables are based on the Chebychev-Gauss-Lobatto grid was developed in [26]. The PMP-based BVPs are solved for these initial conditions, and the final solution is evaluated through high-dimensional interpolation. Semi-Lagrangian approaches based on this sparse grid method have been developed as well [6][18]. Power series-based controller synthesis methods, which are based on a power-series expansion of the state-dynamic equations have been discussed in [2][25]. [34] examined the convergence of such methods to the true optimal solution. This thesis computationally examines one such toolkit developed by Krener. et al. in MATLAB based on [2]. Patchy methods for solving the HJB PDE are discussed in [1][10][39]. These patchy methods first construct a sufficiently accurate approximation of the HJB equation around the equilibrium location (the first patch). The solution is then extended to new patches

around the first one by picking some boundary points based on the characteristics of the HJB equation. These points define the centers of new neighborhoods that can be used to recalculate the power-series expansion. The solution is then obtained iteratively by fitting together the approximations in all the patches [10]. The fact that the HJB equation has characteristics [31, Chapter 7] has been used with the Hopf-Lax formula [17] to alleviate the curse of dimensionality [12][16][49]. Using the Hopf-Lax formula, the HJB-PDE becomes decoupled, and the solution at each point can be effectively calculated through a minimization problem. Similar to the sparse-grid method, the sub-problems generated in the solution procedure are independent and, therefore, are easily parallelizable [12].

2.2 Boundary Value Problem solution

For generating the training and test data for our neural networks (and general grid-free methods), the two-point boundary value problem (BVP) arising from Pontryagin's Principle must be solved for a number of admissible initial system states. The data is recorded in the form $(t, V(t,x), x(t), p(t))$. We use the standard SciPy BVP solver for this purpose [13]. This solver, based on a fourth-order collocation algorithm, requires an initial guess for the $2n$ system of ODEs in theorem 1.2 (Assuming that there are n state variables). However, the solution to such a nonlinear BVP may be very sensitive to the initial guess supplied. The solver may not potentially converge or can converge to different solutions depending on the guess, as, unlike IVPs, unique solutions to BVPs are not guaranteed. These issues become particularly apparent as the number of system variables increases. [37] suggests a time-marching method to alleviate this issue. In particular, problems such as the optimal stabilization of the ODE system arising from the discretization of a Burgers'-like equation, which involves systems of 20-30 dimensions and calculations involving dense matrices, require such an approach to ensure smooth solver convergence. Suppose that our time domain is $[0, t_f]$. Subsequently, the interval is split into relatively smaller ones $0 < t_1 < t_2 < t_3 < ... < t_f$. Then, we solve the BVP with the same boundary conditions at 0 and t_1 using a basic initial guess (such as $x = p = 0, x(0) = x_0$), as it is expected that solver convergence is better when the time interval is shorter. Once it is solved, we *extrapolate* it to the next time t_2 and use it as the initial guess, e.g.: $x(t) = x(t_1),\ p(t) = p(t_1)$ for $t \in [t_1, t_2)$. In [37], this method is shown to lead to improved convergence rates. After solving the BVP over the whole time interval, the value function is calculated by evaluating the running cost along the optimal trajectory and integrating it with the SciPy quadrature-based function integration method [14].

Process-based concurrency in Python

We now digress a little to discuss concurrency and parallelism in Python. Concurrency is the ability of the program to execute instructions out of order, while parallelism is its ability to execute multiple tasks simultaneously. While generating the training/test datasets, the initial conditions (ICs) are sampled independently from a pre-established allowable domain. An individual BVP must be solved for every IC, but it is not necessary to perform these calculations in order. Consequently, we can deploy thread or process-based concurrency in our programs to efficiently increase the speed of data generation. This subsection is primarily based on the reference [7].

Python provides APIs for both thread and process-based concurrency. The former is a lightweight programming construct, which is fast to start and create. Threads share memory within the same process (an instance of a computer program), so it is easier to share data between them. Processes are a more heavy programming construct, as each process represents a separate instance of the Python interpreter running on an individual CPU core. They are slower to start and allocate, and as they do not share the same memory footprint, data sharing between processes requires explicit mechanisms. However, thread-based concurrency is limited by the Global Interpreter Lock (GIL), which enforces that any given Python process can run only a single thread at a time. Therefore, programs using threading-based concurrency have limited parallelism, and are best suited for IO-bound tasks in which the operation of the program is limited by external devices (such as communication with external hardware, which is much slower than the operation of a CPU core). Process-based concurrency, however, is not restricted by the GIL and executes full parallelism, which makes it suitable for CPU-bound tasks (such as intensive numerical operations). Consequently, we decided to apply process-based concurrency using the multiprocessing class to better utilize our hardware resources (20 CPU cores). It should be noted that this class must be deployed to use resources like multiple CPU cores, as Python programs do not use such resources by default.

Whenever a Python script is executed, it starts a process that is an instance of the Python interpreter. This process is termed the *Main Process*. The Main Process can create child processes through instances of the Process class. The basic API to run a function *task* with arguments *arg1, arg2* in a separate process is *process=Process(target=task, args=(arg1, arg2))*. This child process may then be started in the Main Process using the *process.start()* command (Note, the underlying operating system is responsible for starting a new process; we cannot precisely control its starting). The Main Process can then be blocked until the child process completes using the *process.join()* command.

The above method may be used to execute one-off ad-hoc tasks in a separate process.

Each process created requires computational resources, which can become expensive if they are created and destroyed over and over for individual tasks; for example, in our first problem, which involves the solution of more than 8192 BVPs, each of which is a single task. Process pools are a programming construct that can manage a pool of child worker processes and permit their reuse for multiple tasks. Python process pools can be configured using the *Pool* class, which has an API *pool=Pool(processes=num_processes, maxtasksperchild=tasks)*, where *num_processes* is the number of workers to create and manage within the pool. By default, it is the number of logical CPUs in the system. *maxtasksperchild* is the number of tasks executed by each child process before it is replaced by a new worker. By default, it is none, which implies that a child process will be alive as long as the pool executes, but it is a recommended practice to reduce it to avoid the accumulation of resources in case programming bugs occur [7].

Tasks may be submitted to the pool using the *pool.apply(task)* or *pool.apply_async(task)* method. The former will block the main thread until the task is executed while the latter is of a non-blocking kind. Another way of submitting tasks for execution is through the *map()* method: it takes the name of the target function and an iterable. The input iterable is traversed and a task is created in the pool to call the target function. It then returns an iterable over the return values from each call to the target function. However, as all the tasks are issued simultaneously to the pool, it might result in computational issues if the iterable contains several hundreds/thousands of items. Consequently, for our intents and purposes, we decided to apply the *imap()* method to our pool, which is a lazy version of the *map()* method: It applies a target function to each item in the input iterable every time one of the workers become available. Additionally the returned iterable contains the output of the task function in the order in which they are issued to the process pool.

To summarize, our data generation programs use process-based concurrency through a *process-pool imap*. The input iterable is a list of numbers up to the number of trajectories that we want to evaluate. For each such item, a task function is called which generates a permissible initial state and applies the BVP solver to it. Subsequently, any necessary post-processing is performed in the same function, and the output is returned in the form of a numpy vector. After executing all the tasks in the pool, we use the returned iterable to record the output for subsequent applications in the neural network.

2.3 Physics-informed Neural Networks: An overview

Physics-Informed Neural Networks (PINNs) are a scientific machine learning technique used to solve problems involving PDEs [15]. The neural network minimizes a loss function,

which includes terms reflecting the initial and boundary conditions along the space-time domain boundary and the PDE residual at selected points in the domain (called collocation points). After training, given an input point in the integration domain, an estimated solution of the PDE at that point is produced. Consequently, this approach is equivalent to a mesh-free method that finds PDE solutions by converting the problem of directly solving the governing equations into a loss function optimization problem [15]. An overview of the various architectures, loss functions, and optimization methods used for PINNs may be found in [15, Section 2.1], although a simple feedforward NN with 64 neurons in 3 hidden layers was numerically found to be satisfactory for our primary reference [37]. Some of the earliest applications of PINNs involved an NN with a few hidden layers to solve differential equations [30]. [43] presents examples of PINNs to solve commonly solved PDEs such as the Allen-Cahn, Schröedinger and Burgers' equation, as well as examples of parameter estimation problems. Some of the areas where PINNs have found applications are in fluid mechanics [44], turbulence research [35], hematology [27], and electromagnetism [19][28]. [19] is also a work that uses a convolutional neural network as an alternative to the commonly used feedforward networks.

[11] and [46] are early examples of the application of custom NNs to solve the HJB equation for optimal control purposes. [11] used a single layer of neurons with polynomial activation functions and time-varying weights to solve a system of up to 3 state variables. [46] also used a custom NN architecture and presented numerical strategies to improve the convergence of the optimization algorithm, using a Levenburg-Marquardt method. Such an approach requires an explicit implementation of the optimization algorithm, especially if the suggested modifications are considered. It is difficult, if not impossible, to implement such networks using established packages such as TensorFlow and PyTorch, as they effectively encapsulate commonly used NN architectures and optimization algorithms. [24] presented an application of a recurrent NN architecture to compute the HJB equation for a 3-dof nonlinear system. [45] presents examples of solving high-dimensional PDEs, including an example of an HJB equation corresponding to the optimal control of a stochastic heat equation using multilayered neural networks. [37], our primary reference, presented a supervised PINN-based approach using a feedforward architecture to approximate the HJB equation, in which trajectories computed using PMP were used to train the NN. In this thesis, we have computationally examined the methodology proposed in [37] to study optimal control problems for high-dimensional systems.

2.4 Neural Network formulation methodology

We now describe the formulation of one such physics-informed neural network using a fully connected feedforward architecture through the example presented in [47]. Let $F()$ be the function we wish to approximate and $F^{NN}()$ be its NN representation. Feedforward NNs approximate complicated nonlinear functions by a composition of simpler functions [15][37], namely,

$$F(t, \boldsymbol{x}) \approx F^{NN}(t, \boldsymbol{x}) = g_L \bigcirc g_{L-1} \bigcirc ...g_l... \bigcirc g_1(t, \boldsymbol{x}), \tag{2.1}$$

where each layer $g_l(.)$ is defined as:

$$g_l(\boldsymbol{y}) = \sigma(\boldsymbol{W}_l \boldsymbol{y} + \boldsymbol{b}_l). \tag{2.2}$$

Here $\boldsymbol{W}_l$ and $\boldsymbol{b}_l$ are the weight matrices and bias vectors, respectively. $\sigma_l()$ represents a nonlinear activation function applied component-wise to its argument. The final layer is typically linear, so $\sigma_L()$ is the identity function. If $\boldsymbol{\theta}$ denotes the collection of the parameters of the NN, i.e.,

$$\boldsymbol{\theta} := \{\boldsymbol{W}_l, \boldsymbol{b}_l\}_{l=1}^{L} \tag{2.3}$$

the NN is trained by optimizing over the parameters $\boldsymbol{\theta}$ to best approximate $F(t, \boldsymbol{x})$ by $F^{NN}(t, \boldsymbol{x}; \boldsymbol{\theta})$.

Consider the following PDE:

$$\begin{aligned} &\frac{\partial y}{\partial t} = \frac{\partial^2 y}{\partial x^2} - e^{-t}(\sin(\pi x) - \pi^2 \sin(\pi x)) \\ &x \in [-1, 1], \quad t \in [0, 1] \\ &y(x, 0) = \sin(\pi x), \quad y(-1, t) = y(1, t) = 0. \end{aligned} \tag{2.4}$$

This parabolic PDE has the analytic solution $y(t, x) = e^{-t} \sin(\pi x)$. Let the neural network $F^{NN}(t, x) \approx y(t, x)$ approximate the solution to this PDE. It is apparent from [47] that the mean-squared loss function has two components:

$$\begin{aligned} MSE_f &= \frac{1}{N_f} \sum_i^{N_f} |f(x_f^i, t_f^i)|^2 \\ f &= \frac{\partial F^{NN}(t, x)}{\partial t} - \frac{\partial^2 F^{NN}(t, x)}{\partial x^2} + e^{-t}(\sin(\pi x) - \pi^2 \sin(\pi x)) \\ MSE_u &= \frac{1}{N_u} \sum_i^{N_u} |y(x_u^i, t_u^i) - F^{NN}(x_u^i, t_u^i)|^2. \end{aligned} \tag{2.5}$$

Here, there are N_f collocation points inside the domain $D : (x_f^i, t_f^i) \in (-1, 1) \times (0, 1)$ and N_u collocation points based on the boundary conditions $C : (x_u^i, t_u^i) \in \{-1, 1\} \times [0, 1] \cup (-1, 1) \times \{0\}$ at which $y(t, x)$ is known from the boundary conditions. In summary, the procedure involves creating a vector $(x_{Nu}, t_{Nu}, y_{Nu})_{Nu\times 3}$, where the (x_{Nu}, t_{Nu}) are randomly sampled from C and consequently, $MSE_u = MSE(F^{NN}(x_{Nu}, t_{Nu}), y_{Nu})$. Another vector comprising of the training points $(x_f, t_f)_{N_f\times 2}$ is used, where the elements are randomly sampled from D. Some more details about the physics-based loss calculation (MSE_f), along with some computational aspects of PyTorch, are presented below:

1. The forward pass is performed as $F^{NN} = NN(x_f, t_f)$, where (x_f, t_f) is a PyTorch tensor located in the appropriate device (CUDA or CPU) with a gradient attribute=True to enable gradient-tracking. For PyTorch-based applications, the number of rows of the input equals the number of datapoints, and the number of columns equals the number of features, with the output being a matrix having the number of rows equal to the number of datapoints, and the number of columns equal to the number of output features.

2. Let $F^{NN} = \vec{f}$ and $(x_f, t_f) = \vec{g}$ for coding purposes. *F_x_t* is then calculated using the autograd.grad command:

   F_x_t = autograd.grad(f,g,torch.ones([g.shape[0], 1]).to(device), retain_graph=True, create_graph=True)[0]

   As explained in [20], if $\vec{y} = F(\vec{x})$, then the autograd.grad command first internally calculates the Jacobian Matrix $J : J_{ij} = \frac{\partial y_i}{\partial x_j}$. A *gradient argument*=$\vec{v}$ must be supplied, and the output is $J^T\vec{v}$. This is equal to the vector generated from *torch.ones* statement.

   For our example, for each input (x_f^i, t_f^i), J would be a $N_f \times 2$ matrix. However, as calculations for each element $(x_f^i, t_f^i) =\in \vec{g} \in D$, is done independently in the back end, the only non-zero entries possible in the Jacobian are (J_{i1}, J_{i2}). Consequently, when our gradient argument $\vec{v}_{Nf\times 2} : v_{ij} = 1$, the output vector F_x_t is such that the $i - th$ row is $\big(\frac{\partial F^{NN}(x_f^i, t_i^f)}{\partial x_f^i}, \frac{\partial F^{NN}(x_f^i, t_i^f)}{\partial t_f^i}\big)$. Thus, $F_t(x_f^i, t_i^f)$ equals the 2-nd column of the resulting vector.

3. As explained in [20], autograd keeps a record of data (tensors) and all executed operations (along with the resulting new tensors) in a directed acyclic graph (DAG) consisting of Function objects. In this DAG, leaves are the input tensors, roots are the output tensors. By tracing this graph from roots to leaves, one can automatically compute the gradients using the chain rule. However, during a backward pass, this

graph gets deleted. Consequently, one must set the *create_graph* and *retain_graph* argument in the autograd.grad command to true in order to create a computational graph of the derivatives [21]: this is essential in order to calculate higher-order derivatives, as well as allow the PyTorch backend to appropriately evaluate the gradients with respect to the neural network parameters when the .backward() method is called on the loss-function in the optimization step.

With the above two explanations, one can see how *F_xx_tt* was calculated:

F_xx_tt = autograd.grad(F_x_t,g,torch.ones(g.shape).to(device), create_graph=True)[0]

F_x_t is automatically differentiated using the internally calculated computational graph from the previous step. F_{xx} now equals the first column of this vector.

4. One can now evaluate the tensor, $F_t^i - F_{xx}^i = \frac{\partial F^{NN}(x_f^i, t_f^i)}{\partial t_f^i} - \frac{\partial^2 F^{NN}(x_f^i, t_f^i)}{\partial (x_f^i)^2}$. Subsequently, the source term $e^{-t_f^i}(\sin(\pi x_f^i) - \pi^2 \sin(\pi x_f^i))$ can be added, and the resulting tensor used in the physics-based loss function MSE_f.

We recommend the above approach for calculating the loss function for optimization. The torch.autograd package encapsulates algorithms for the auto-differentiation as a batch process in the back-end, enabling faster computations and more efficient use of resources, compared to an approach using explicit loops. This is especially true if a GPU is used in the calculations, as the clock for GPU processors is slower than that of a CPU.

From this example, it is seen that PINNs incorporate losses based on physics-expected constraints, which potentially involve derivatives calculated from back-propagation. In an analogous fashion, our loss function for the Hamilton-Jacobi-Bellman equation is based on the physics-based constraints described in section 1.7, where the fact that $p = -V_x$ (assuming that the derivatives exist pointwise) is used. Thus, our loss function is:

$$\begin{aligned} \text{loss}(\boldsymbol{\theta}; \mathcal{D}) &= \underset{V}{\text{loss}}(\boldsymbol{\theta}; \mathcal{D}) + \mu . \underset{\boldsymbol{p}}{\text{loss}}(\boldsymbol{\theta}; \mathcal{D}) \\ \underset{V}{\text{loss}}(\boldsymbol{\theta}; \mathcal{D}) &= \frac{1}{N_d} \sum_{i=1}^{N_d} \left[V^{(i)} - V^{NN}(t^{(i)}, x^{(i)}; \boldsymbol{\theta}) \right]^2 \\ \underset{p}{\text{loss}}(\boldsymbol{\theta}; \mathcal{D}) &= \frac{1}{N_d} \sum_{i=1}^{N_d} ||p^{(i)} + V_x^{NN}(t^{(i)}, x^{(i)}; \boldsymbol{\theta})||_2^2 \end{aligned} \tag{2.6}$$

where $\boldsymbol{\theta}, \mathcal{D}$ denotes the weights of the neural network and the training dataset, respectively. μ here acts as a hyper-parameter; its influence on the performance of the neural

network is examined in the first problem in chapter 3. The principal difference between [47] and our approach to solving the HJB equation is that the costates (and the optimal Hamiltonian, if needed) are evaluated at the data-generation phase instead of the loss-calculation stage. From the viewpoint of [15], this is a supervised learning paradigm, while the example presented for the diffusion equation is an example of unsupervised learning.

It will be subsequently mentioned that the first problem involves the design of an infinite-horizon controller, such that the control depends only on the current state and not on the time. Consequently, the neural network does not take time as an input. The other problems involve moving horizon controllers that do take time into account. An additional term based on eq. (1.6) based on the fact that $V_t = H^*(t, x, u)$ can be proposed for the loss function. Note: The optimal Hamiltonian for a fixed-time optimal control problem involving time-invariant dynamics, is known to be a constant, and it can be calculated during the data-generation phase without a significant time overhead. We examine the effect of including this term for the second problem in chapter 4.

In summary, the basic training pipeline for the neural network would be:

1. A number of initial conditions (ICs) $\{X^{(i)}\}_{i=1}^{i=N_d}$ are sampled from a known domain for which the controller is to be designed. The BVP is formulated from the PMP.

2. For every IC, the BVP must be solved to generate the training dataset $\mathcal{D}$. The data is recorded in the form $\{t^{(i)}, x^{(i)}(t^{(i)}), V^{(i)}(t^{(i)}, t_f), p^{(i)}(t^{(i)})\}$ by solving the BVP from step 1. Note: for time-dependent controllers, the solver will have a series of default time instants for each IC. The value function $V^{(i)}(t^{(i)}, t_f)$ is evaluated by integrating the running cost from $t^{(i)}$ to t_f and adding the terminal cost.

3. Define the NN Architecture. In this thesis, we have used a simple feedforward NN having three internal layers of 64 neurons each with tanh activation functions to fit the value function, unless mentioned otherwise. PyTorch was used to develop the NN framework as well as perform the optimization for NN training.

4. Train the network with the selected PINN-based cost function, using the training dataset $\mathcal{D}$ generated from PMP. Since our datasets are relatively small (This is even more true if adaptive data-collection is used), the second-order optimizer L-BFGS is used as it is a quasi-newton method that uses data from the entire dataset. For the small NNs and relatively small datasets used, L-BFGS is expected to give better error metrics [15][37] compared to first-order optimizers such as Adam or SGD.

5. Use the trained NN in conjunction with the Hamiltonian maximization condition to calculate the optimal control.

In general, a given dataset is randomly split into a training set and a test set. During training, the loss function is calculated with respect to the training dataset, and the NN performance is measured against the test dataset, which was not applied in the training process. In this thesis, the validation test is made more stringent by generating the test and training data from independently drawn initial conditions so that the two datasets do not share any part of the same trajectories when time instants are considered in the data generation process. We use the following metrics based on [37] to examine the NN performance. The $RMAE$ is termed the relative mean absolute error, while RML^2 is the relative mean L^2 error in gradient prediction. According to [37], these metrics are a better representation of the NN approximation in regions where a lot of control effort is needed, instead of pointwise errors.

$$\begin{aligned} RMAE(\boldsymbol{\theta}, \boldsymbol{\mathcal{D}}) &= \frac{\sum_{i=1}^{N_d} |V^{(i)} - V^{NN}(\boldsymbol{x}^i; \boldsymbol{\theta})|}{\sum_{i=1}^{N_d} |V^{(i)}|} \\ RML^2(\boldsymbol{\theta}, \boldsymbol{\mathcal{D}}) &= \frac{\sum_{i=1}^{N_d} ||\boldsymbol{p}^{(i)} + V_{\boldsymbol{x}}^{NN}(\boldsymbol{x}^i; \boldsymbol{\theta})||_2}{\sum_{i=1}^{N_d} ||\boldsymbol{p}^{(i)}||_2}. \end{aligned} \tag{2.7}$$

Finally, the application of our trained NN in the context of the optimal controller is discussed. It is very easy to calculate $V_x(t, x(t))$ from the network using *backpropagation*. Since we know that $p = -V_x$; we can use it in the *Hamiltonian maximization equation* $H_u = 0$. While the NN-based approach is theoretically applicable to any general dynamic system described by the first paragraph of section 1.2, all our systems are effectively described through the relations:

$$\begin{aligned} \dot{\boldsymbol{x}} &= \boldsymbol{f}(t, \boldsymbol{x}) + \boldsymbol{g}(t, \boldsymbol{x})\boldsymbol{u} \\ \mathcal{L}(t, \boldsymbol{x}, \boldsymbol{u}) &= h(t, \boldsymbol{x}) + \boldsymbol{u}^T \boldsymbol{W} \boldsymbol{u} \\ \mathcal{H}(t, \boldsymbol{p}, \boldsymbol{x}, \boldsymbol{u}) &= -(h(t, \boldsymbol{x}) + \boldsymbol{u}^T \boldsymbol{W} \boldsymbol{u}) + \boldsymbol{p}^T (\boldsymbol{f}(t, \boldsymbol{x}) + \boldsymbol{g}(t, \boldsymbol{x})\boldsymbol{u}). \end{aligned} \tag{2.8}$$

This implies though, that for a symmetric $\boldsymbol{W}$, the control $\boldsymbol{u}$ can be explicitly calculated as $\boldsymbol{u} = \frac{1}{2}\boldsymbol{W}^{-1}\boldsymbol{g}(t, \boldsymbol{x})^T \boldsymbol{p} \approx -\frac{1}{2}\boldsymbol{W}^{-1}\boldsymbol{g}(t, \boldsymbol{x})^T V_{\boldsymbol{x}}^{NN}$. From a theoretical perspective, it allows the explicit calculation of the control corresponding to the infimum in eq. (1.6) in terms of $V_x(t, x)$ to get an explicit PDE involving $V(t, x)$, similar to eq. (1.7). No iterative method is needed to calculate the control from an implicit equation. The optimal control can thus be better predicted since derivative calculations using backpropagation are much more efficient and accurate than numerical approximations which would have been necessary had grid-based methods been applied to the HJB equation.

2.5 Toolkit of Krener et. al.

Krener et al. developed a MATLAB toolkit based on [2] for approximating the optimal controller for time-invariant non-linear infinite-horizon problems [29]. This method involves expanding the system dynamics and running cost about its equilibrium operating point as a (convergent) power series and using this to systematically obtain the value function and optimal control as a power series expansion as well. This toolkit may be used to approximate the optimal control to stabilize a smooth non-linear system at its stable equilibrium point.

We investigated this toolkit in detail computationally to examine its behavior. As expected from theory, this toolkit was found to approximate the value function using a multivariate polynomial function of the deviation of the state variables from the system equilibrium. As an infinite-horizon problem is considered, time is not taken into account. If the control is to be approximated using a polynomial of order d, the value function contains multivariate monomials of the order 2 to $d+1$. The optimal feedback is approximated using polynomials of order 1 to d. The meaning and arrangement of these monomials are explained in detail below.

If there are n state variables and Δx_j denotes the deviation of the j-th state from its equilibrium value, we can have a monomial expression $M(\Delta x_j, \alpha_j) = \prod_{i=1}^{n} \Delta x_j^{\alpha_j}$. The sum of the powers $\sum_{j=1}^{n} \alpha_j$ is termed the order of the monomial. Let us denote the set of all combinations $M(\Delta x_j, \alpha_j)$ such that the order of the combination is constant by $C(k) : M(\Delta x_j, \alpha_j) \in C(k) \iff \sum_{j=1}^{n} \alpha_j = k$. This set will contain $\binom{k+n-1}{k}$ elements. These may be sorted in a lexicographic fashion similar to words in a dictionary: compare the coefficients of Δx_1, and if they match, compare that of Δx_2, and so forth. For a given set of deviations Δx_j the expressions $M(\Delta x_j, \alpha_j) \in C(k)$ may be numerically evaluated and arranged in this fashion. If $2 \leq k \leq d+1$, the toolkit calculates the weightage of each expression $M(\Delta x_j, \alpha_j)$ in the approximate value function in this order. For a range of $C(k)$, the monomial coefficients are evaluated in order of increasing k. The optimal value of each control is approximated in a similar fashion, with the exception that the weights of an expression $M(\Delta x_j, \alpha_j) \in C(k)$ is determined if $1 \leq k \leq d$. If the system has multiple independent controls, the weights for a given $M(\Delta x_j, \alpha_j)$ are determined independently.

A numerical example is now presented as a further illustration. Consider the following system based on the MATLAB documentation for the Algebraic Riccati Equation [22]:

$$\begin{aligned}
\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\\ \dot{x}_3\end{bmatrix} &= \begin{bmatrix}-1 & 2 & 3\\ 4 & 5 & -6\\ 7 & -8 & 9\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix} + \begin{bmatrix}5\\ 6\\ -7\end{bmatrix} u \\
L(x_1, x_2, x_3, u) &= \begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix}^T Q \begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix} + Ru^2 \\
Q &= C^T C; \quad C = \begin{bmatrix}7 & -8 & 9\end{bmatrix}; \quad R = 1
\end{aligned} \tag{2.9}$$

The pair of matrices (A, B) is controllable, and (A, C) is observable. It is known from the linear control theory [33] that the infinite horizon controller for such systems is governed by the unique solution P to the Algebraic Riccati Equation (ARE):

$$A^T P + PA + Q - PBR^{-1}B^T P = 0$$

with the optimal control given by the function $u = -(R^{-1}B^T P)x$. The optimal cost incurred for an initial state X_0 is $X_0^T P X_0$. Consequently, the optimal cost is a quadratic function of these state variables. One can further see that for this system with a single control input, the control is expected to be a linear function of the state variables (these are equivalent to the deviations used in the earlier paragraphs, as the equilibrium point for a linear system is zero).

We implemented the toolkit of Krener et al. for this system for a degree of approximation = 3 and compared the output with the MATLAB solver for the ARE (icare, [22]). The value function polynomial has $\binom{3+2-1}{2} + \binom{3+3-1}{3} + \binom{3+4-1}{4} = 6 + 10 + 15 = 31$ entries, as expected from the preceding paragraphs. Also, only the first 6 entries, corresponding to the quadratic cost are non-zero. The output cost, interpreted in the lexicographic sense, is $15.32x_1^2 + 8.474x_1x_2 + 34.018x_1x_3 + 2.625x_2^2 + 8.825x_2x_3 + 19.038x_3^2$. This can be rearranged in terms of a symmetric matrix $P = \begin{bmatrix}15.32 & 4.237 & 17.009\\ 4.237 & 2.625 & 4.413\\ 17.009 & 4.413 & 19.038\end{bmatrix}$, such that the cost $= [x_1\, x_2\, x_3]^T P [x_1\, x_2\, x_3]$ The feedback control is a matrix with 1 row and $\binom{3+1-1}{1} + \binom{3+2-1}{2} + \binom{3+3-1}{3} = 3 + 6 + 10 = 19$ columns, with only the first three-entries non-zero, as expected from theory. The entries match the MATLAB results for both K and P, with the exception of the sign as MATLAB just calculates $K = R^{-1}B^T P$, such that $u = -Kx$, while this toolkit directly returns $-K$, so that an extra change of sign is not needed. From this example, we can claim the validity of our statements on the working of Krener's toolkit at the beginning of this section.

It is apparent that solving a full two-point boundary value problem as described in section 2.2 is a computationally expensive procedure. If the terminal time t_f is very large compared to the state dynamics, then one can claim that the optimal trajectory for this finite-horizon problem approaches that of its infinite-horizon counterpart, analogous to how the time-varying solution to the Differential Riccati Equation approaches the Algebraic Riccati Equation (a constant) in the infinite-time limit for optimal control calculations for LTI systems [31, Chapter 6]. Consequently, we examined whether we could use the approximate results from this toolkit to our advantage when training the neural network or calculating the optimal trajectory of the actual non-linear system, as the application of this toolkit simply involves polynomial calculations. We discuss these experiments in detail in section 3.2 and section 4.2.

Chapter 3

Aircraft-orientation control: A 6-dof system

We independently reconstruct the first numerical example discussed in [37]. It pertains to the control of the attitude/orientation of a spacecraft using three momentum wheels. Momentum wheels constitute a set of rotating wheels aligned in at least three different directions. In flight, by changing the rotation rate, the angular orientation of the object can be changed (by the principle of conservation of Angular Momentum).

The system is governed by the following non-linear equations consisting of six state variables $\boldsymbol{x} = \{\boldsymbol{v}^T, \boldsymbol{\omega}^T\}^T$; $\boldsymbol{v} = (\phi, \theta, \psi)^T$, $\boldsymbol{\omega} = (\omega_1, \omega_2, \omega_3)^T$:

$$\begin{bmatrix} \dot{\boldsymbol{v}} \\ \dot{\boldsymbol{\omega}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{E}(\boldsymbol{v})\boldsymbol{\omega} \\ \boldsymbol{J}^{-1}\boldsymbol{S}(\boldsymbol{\omega})\boldsymbol{R}(\boldsymbol{v})\boldsymbol{h} + \boldsymbol{J}^{-1}\boldsymbol{B}(\boldsymbol{u}) \end{bmatrix} \tag{3.1}$$

where, the functions $\boldsymbol{E}(\boldsymbol{v})$, $\boldsymbol{S}(\boldsymbol{\omega})$, $\boldsymbol{R}(\boldsymbol{v}) : \mathbb{R}^{3\times 3} \to \mathbb{R}^3$ are:

$$\begin{aligned} \boldsymbol{E}(\boldsymbol{v}) &= \begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi/\cos\theta & \cos\phi/\cos\theta \end{bmatrix} \quad \boldsymbol{S}(\boldsymbol{\omega}) = \begin{bmatrix} 0 & \omega_3 & -\omega_2 \\ -\omega_3 & 0 & \omega_1 \\ \omega_2 & -\omega_1 & 0 \end{bmatrix} \\ \boldsymbol{R}(\boldsymbol{v}) &= \begin{bmatrix} \cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta \\ \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \cos\theta\sin\phi \\ \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi & \cos\theta\cos\phi \end{bmatrix}. \end{aligned} \tag{3.2}$$

Matrix $\boldsymbol{B}$ is a $3 \times m$ constant matrix, where m is the number of momentum wheels. For the nonlinear control problem posed, $m = 3$. Matrix $\boldsymbol{J}$ is a 3×3 constant matrix, and $\boldsymbol{h}$

is a 3×1 vector. Their values are as follows:

$$\boldsymbol{J} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 4 \end{bmatrix} \quad \boldsymbol{B} = \begin{bmatrix} 1 & 1/20 & 1/10 \\ 1/15 & 1 & 1/10 \\ 1/10 & 1/15 & 1 \end{bmatrix} \quad \boldsymbol{h} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}. \tag{3.3}$$

The control objective optimized is:

$$\begin{aligned} \underset{\boldsymbol{u(.)}}{\text{minimize}} \quad J[\boldsymbol{u(.)}] &= \int_t^{t_f} \mathcal{L}(\boldsymbol{v}, \boldsymbol{\omega}, \boldsymbol{u}) d\tau + \frac{W_4}{2}||\boldsymbol{v}(t_f)||^2 + \frac{W_5}{2}||\boldsymbol{\omega}(t_f)||^2 \\ \mathcal{L}(\boldsymbol{v}, \boldsymbol{\omega}, \boldsymbol{u}) &= \frac{W_1}{2}||\boldsymbol{v}||^2 + \frac{W_2}{2}||\boldsymbol{\omega}||^2 + \frac{W_3}{2}||\boldsymbol{u}||^2 \\ W_1 = 1, \quad W_2 = 10 \quad W_3 = 1/2 \quad W_4 = 1 \quad W_5 = 1 \quad t_f &= 20. \end{aligned} \tag{3.4}$$

It is also assumed that at $t = 0$, the state $\boldsymbol{x}$ belongs to the following set:

$$\mathbb{X}_0 = \left\{ \boldsymbol{v}, \boldsymbol{\omega} \in \mathbb{R}^3 \middle| -\frac{\pi}{3} \leq \phi, \theta, \psi \leq \frac{\pi}{3} \text{ and } -\frac{\pi}{4} \leq \omega_1, \omega_2, \omega_3 \leq \frac{\pi}{4} \right\}. \tag{3.5}$$

As per PMP, the costates and optimal control are governed by the following BVP:

$$\begin{aligned} \dot{p}_\phi &= W_1\phi - [p_\phi, p_\theta, p_\psi]\frac{\partial \boldsymbol{E}(\boldsymbol{v})}{\partial \phi}\boldsymbol{\omega} - [p_{\omega_1}, p_{\omega_2}, p_{\omega_3}]\boldsymbol{J^{-1}}\boldsymbol{S}(\boldsymbol{\omega})\frac{\partial \boldsymbol{R}(\boldsymbol{v})}{\partial \phi}\boldsymbol{h} \\ \dot{p}_\theta &= W_1\theta - [p_\phi, p_\theta, p_\psi]\frac{\partial \boldsymbol{E}(\boldsymbol{v})}{\partial \theta}\boldsymbol{\omega} - [p_{\omega_1}, p_{\omega_2}, p_{\omega_3}]\boldsymbol{J^{-1}}\boldsymbol{S}(\boldsymbol{\omega})\frac{\partial \boldsymbol{R}(\boldsymbol{v})}{\partial \theta}\boldsymbol{h} \\ \dot{p}_\psi &= W_1\psi - [p_{\omega_1}, p_{\omega_2}, p_{\omega_3}]\boldsymbol{J^{-1}}\boldsymbol{S}(\boldsymbol{\omega})\frac{\partial \boldsymbol{R}(\boldsymbol{v})}{\partial \psi}\boldsymbol{h} \\ \dot{p}_{\omega_1} &= W_2\omega_1 - [p_\phi, p_\theta, p_\psi]col_1(\boldsymbol{E}(\boldsymbol{v})) - [p_{\omega_1}, p_{\omega_2}, p_{\omega_3}]\boldsymbol{J^{-1}}\frac{\partial \boldsymbol{S}(\boldsymbol{\omega})}{\partial \omega_1}\boldsymbol{h} \\ \dot{p}_{\omega_2} &= W_2\omega_3 - [p_\phi, p_\theta, p_\psi]col_2(\boldsymbol{E}(\boldsymbol{v})) - [p_{\omega_1}, p_{\omega_2}, p_{\omega_3}]\boldsymbol{J^{-1}}\frac{\partial \boldsymbol{S}(\boldsymbol{\omega})}{\partial \omega_2}\boldsymbol{h} \\ \dot{p}_{\omega_3} &= W_2\omega_3 - [p_\phi, p_\theta, p_\psi]col_3(\boldsymbol{E}(\boldsymbol{v})) - [p_{\omega_1}, p_{\omega_2}, p_{\omega_3}]\boldsymbol{J^{-1}}\frac{\partial \boldsymbol{S}(\boldsymbol{\omega})}{\partial \omega_3}\boldsymbol{h} \\ \boldsymbol{u}^*(t) &= \frac{1}{W_3}([p_{\omega_1}, p_{\omega_2}, p_{\omega_3}]\boldsymbol{J^{-1}}\boldsymbol{B})^T \\ p_v(t_f) &= -W_4 v(t_f), \quad v \in (\phi, \theta, \psi) \\ p_\omega(t_f) &= -W_5\omega(t_f), \quad \omega \in (\omega_1, \omega_2, \omega_3). \end{aligned} \tag{3.6}$$

The matrices used in the above equations are given as follows:

$$\frac{\partial \boldsymbol{R}(\boldsymbol{v})}{\partial \phi} = \begin{bmatrix} 0 & 0 & 0 \\ \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi & \cos\theta\cos\phi \\ -\sin\phi\sin\theta\cos\psi + \cos\phi\sin\psi & -\sin\phi\sin\theta\sin\psi - \cos\phi\cos\psi & -\cos\theta\sin\phi \end{bmatrix}$$

$$\frac{\partial \boldsymbol{R}(\boldsymbol{v})}{\partial \theta} = \begin{bmatrix} -\sin\theta\cos\psi & -\sin\theta\sin\psi & -\cos\theta \\ \sin\phi\cos\theta\cos\psi & \sin\phi\cos\theta\sin\psi & -\sin\theta\sin\phi \\ \cos\phi\cos\theta\cos\psi & \cos\phi\cos\theta\sin\psi & -\sin\theta\cos\phi \end{bmatrix}$$

$$\frac{\partial \boldsymbol{R}(\boldsymbol{v})}{\partial \psi} = \begin{bmatrix} -\cos\theta\sin\psi & \cos\theta\cos\psi & 0 \\ -\sin\phi\sin\theta\sin\psi - \cos\phi\cos\psi & \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & 0 \\ -\cos\phi\sin\theta\sin\psi + \sin\phi\cos\psi & \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi & 0 \end{bmatrix}$$

$$\frac{\partial \boldsymbol{E}(\boldsymbol{v})}{\partial \phi} = \begin{bmatrix} 0 & \cos\phi\tan\theta & -\sin\phi\tan\theta \\ 0 & -\sin\phi & -\cos\phi \\ 0 & \cos\phi/\cos\theta & -\sin\phi/\cos\theta \end{bmatrix} \quad \frac{\partial \boldsymbol{E}(\boldsymbol{v})}{\partial \theta} = \begin{bmatrix} 0 & \sin\phi\sec^2\theta & \cos\phi\sec^2\theta \\ 0 & 0 & 0 \\ 0 & \sin\phi\sec\theta\tan\theta & \cos\phi\sec\theta\tan\theta \end{bmatrix}$$

$$\frac{\partial \boldsymbol{S}(\boldsymbol{\omega})}{\partial \omega_1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix} \quad \frac{\partial \boldsymbol{S}(\boldsymbol{\omega})}{\partial \omega_2} = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \frac{\partial \boldsymbol{S}(\boldsymbol{\omega})}{\partial \omega_3} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{3.7}$$

3.1 Basic Neural-Network formulation

In our problem, the Hamiltonian is time-invariant, and the time horizon is rather large. Consequently, the controller may be *approximated with a time-independent moving horizon controller* rather than a time-dependent controller. This turns out to be analogous to an infinite time-horizon controller. Consequently, the neural network only approximates the value function at $t = 0$, i.e., $V(0, \boldsymbol{v}(0), \boldsymbol{\omega}(0))$ and *does not* take time as an input variable. At each time t when the control is evaluated, we assume the current system state $(\boldsymbol{v}(t), \boldsymbol{\omega}(t))$ is the initial condition and evaluate the value function as $\boldsymbol{V}(t) = V^{NN}(\boldsymbol{v}(t), \boldsymbol{\omega}(t))$ for calculating the optimal control. One can consider an analogous case for linear systems: the Differential Riccati Equation for the finite-time horizon, which governs optimal control for such systems, reduces to the Algebraic Riccati Equation in which feedback gain is independent of time, as the time horizon becomes very large [31, Chapter 6].

As time is not considered as an input to the neural network, the training pipeline changes slightly from the one discussed in section 2.4. The problem-specific training pipeline is:

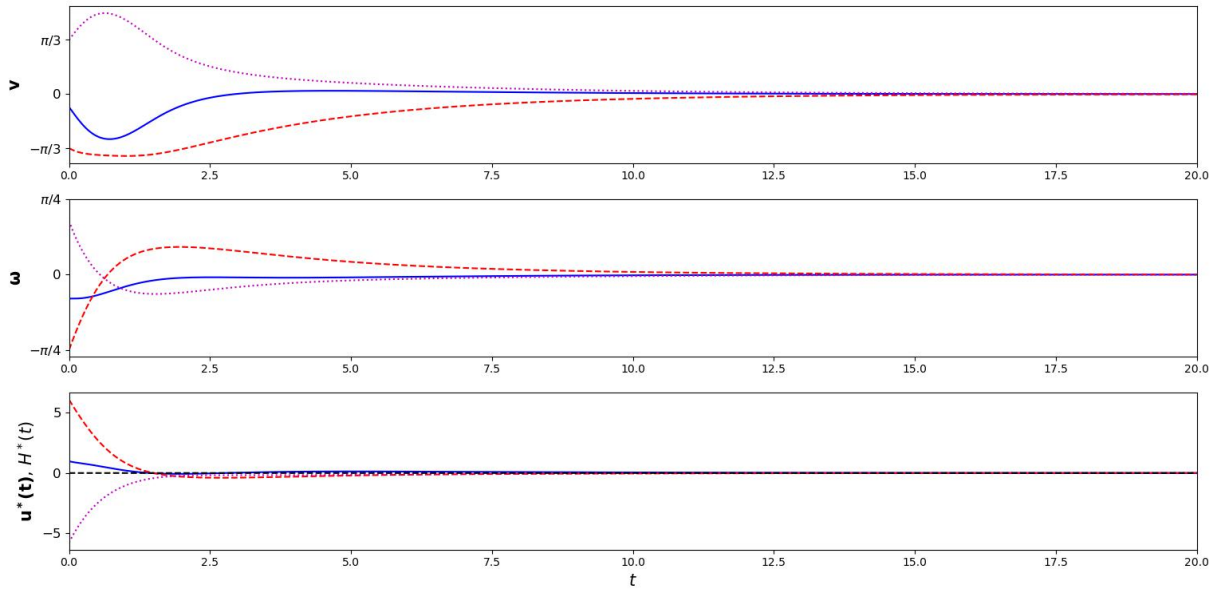


Figure 3.1: Optimal trajectories and controls for the initial state calculated from PMP: The initial state $\boldsymbol{v}(0) = (\phi(0), \theta(0), \psi(0)) = (-1/4, -\pi/3, \pi/3)$ and $\boldsymbol{\omega}(0) = (\omega_1(0), \omega_2(0), \omega_3(0)) = (-1/4, -\pi/4, 0.56)$. The optimal control vector $\boldsymbol{u}^*(t) = (u_1^*(t), u_2^*(t), u_3^*(t))$. Solid blue line: ϕ, ω_1, $u_1^*(t)$; Dashed red line: θ, ω_2, $u_2^*(t)$; Dotted purple line: ψ, ω_3, $u_3^*(t)$. Dashed black line: Optimal Hamiltonian. From theorem 1.2, it is known that such trajectories have a constant Hamiltonian, and since this particular system is stable, the states and the control tend to zero due to the large time horizon. Consequently, the Hamiltonian obtained is nearly zero.

1. Generate a set of data points $\{\boldsymbol{x}^{(i)}\}_{i=1}^{N_d}$ such that $\boldsymbol{x}^{(i)} \in \mathbb{X}_0$ and are independent and identically distributed.

2. Define the BVP from Pontryagin's Principle. Solve the BVP for the relevant $\boldsymbol{x}^{(i)}$ using the SciPy BVP solver (optionally using process-based concurrency) to generate training dataset $\mathcal{D} = \{(\boldsymbol{x}^{(i)}), (V^{(i)}, \boldsymbol{p}^{(i)})\}_{i=1}^{N_d}$. The value function $V^{(i)} = V(0, t_f)$ and the initial costate $p^{(i)} = p^*(0)$ is recorded as $(V^{(i)}, \boldsymbol{p}^{(i)})$.

3. Define the Neural Network. As mentioned before in section 2.4, a standard feed-forward architecture using 3 Hidden Layers of 64 Neurons each was used with tanh activation functions. The entire framework was constructed and optimized in Py-Torch ([37] used a TensorFlow framework with a scikit-learn-based optimizer). Some recommended practices in this aspect were discussed in section 2.4.

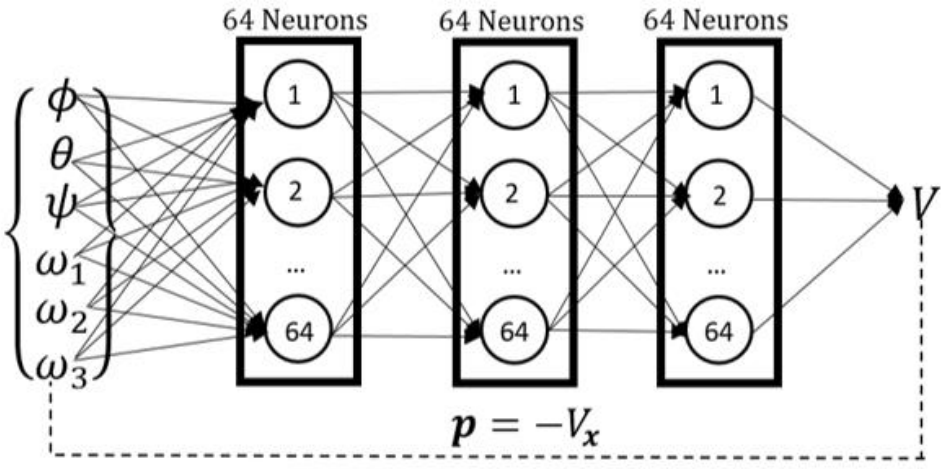


Figure 3.2: A visual representation of our problem-specific NN

4. Train the network with the physics-based loss function eq. (2.6). As mentioned in section 2.4, we have used a Pytorch-based L-BFGS optimization method.

At this point, some important differences between the computational approach developed in this thesis and that of our primary reference [37] should be mentioned. As explained in section 2.2, solving BVPs generally involves high computational costs. Consequently, it could be time-consuming to generate large causality-free datasets (where the trajectories are generated from independently sampled initial conditions). As an attempt to mitigate this issue, [37] recommended a number of strategies that involve NN training and data generation in a closed loop. This involves training a low-fidelity network with a small dataset and gradually increasing the dataset size based on the losses calculated during the training. In [37, section 4], a statistical method is described to examine the overall method convergence and the extent to which the training dataset must be expanded. The lower fidelity neural network is used in two ways: firstly, to identify regions in the state-variable space in which a larger control effort is needed, as estimated by $||V_{\boldsymbol{x}}(0)||_2$ evaluated from the NN. When the dataset is expanded, one initially considers a large number of candidate points selected from the state-variable domain using Monte Carlo sampling. An estimate of $||V_{\boldsymbol{x}}(0)||_2$ is quickly evaluated from the NN, and as $\boldsymbol{p}(0) = -V_{\boldsymbol{x}}(0)$, a larger value implies a larger initial control effort. Consequently, these points can be selectively included in the expanded dataset and their optimal trajectories used in the NN training, potentially increasing its robustness. Secondly, this low-fidelity NN may be used to generate a guess for the BVP solver, and it is expected that this initial guess would allow the solver to converge faster and in a more stable way. This serves as an alternative to the time-marching trick

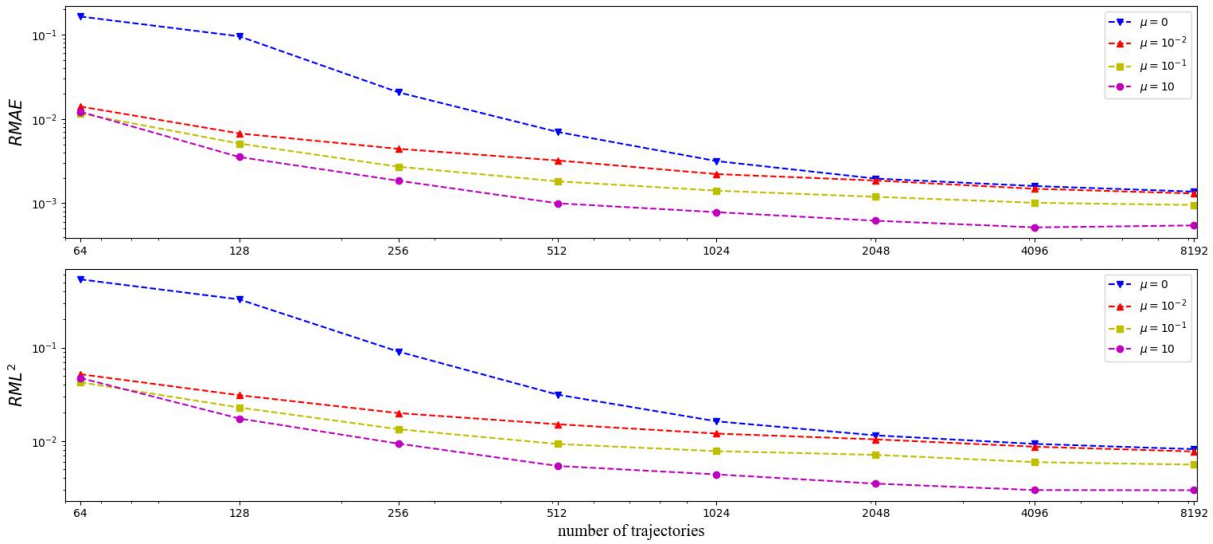


Figure 3.3: NN performance metrics at the conclusion of 500 training iterations for the optimal aircraft attitude control problem. The performances obtained are similar to that of [37], and serves to validate our implementation of the basic NN method. For consistency reasons, all the NNs have the same initial parameter initialization.

described in section 2.2, in which the time domain was split into smaller intervals with an extrapolation to later times. The NN is then re-trained using the expanded dataset, closing the loop. Thus, this procedure is expected to yield a high-fidelity NN as well as an optimal training dataset.

In this thesis, we have not implemented such a closed-loop approach. Our approach is "open-loop", in the sense that the training dataset is generated independent of the intermediate performance of the NN and that it is not expanded further. However, for this problem, we examine the effect of pre-training the NN using Krener's toolkit. We also examine the effect of selectively choosing points based on control effort estimated from Krener's toolkit on the overall performance of the NN in the same context. We examine the effect of using this toolkit to generate an initial guess for the second problem in section 4.2. Nevertheless, even if a closed-loop approach, as discussed in [37, section 4] is used, including a penalty based on V_t in the loss function (for time-dependent controllers) is still expected to result in improved NN convergence metrics.

We present numerical examples of the basic NN performance in fig. 3.3 and fig. 3.4. From fig. 3.3, it is apparent that the inclusion of the costate-based term with the hyper-

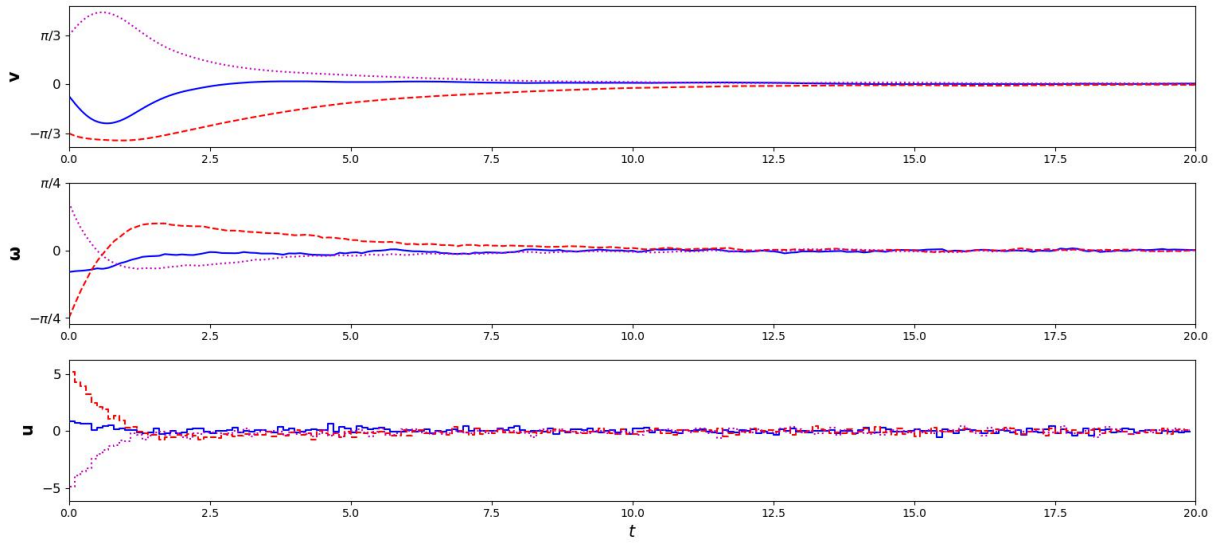


Figure 3.4: Performance of the NN in the presence of noise. The NN applied was trained with $\mu = 10$ and involved 8192 sample points. The default initial condition is the same as that of fig. 3.1: $\boldsymbol{v}(0) = (\phi(0), \theta(0), \psi(0)) = (-1/4, -\pi/3, \pi/3)$ and $\boldsymbol{\omega}(0) = (\omega_1(0), \omega_2(0), \omega_3(0)) = (-1/4, -\pi/4, 0.56)$. Sampling was done at a frequency of 10 Hz and corrupted with white Gaussian noise of standard deviation 0.01π. Solid blue line: ϕ, ω_1, $u_1^*(t)$; Dashed red line: θ, ω_2, $u_2^*(t)$; Dotted purple line: ψ, ω_3, $u_3^*(t)$.

parameter μ in the loss function eq. (2.6) always improves the NN performance. It was observed in [37] that the accuracy improved as μ was increased, with diminishing returns from $\mu = 10$. Figure 3.4 shows an application of a trained neural network to predict the optimal control when the readings are corrupted with noise. This simulation entailed an RK-4 integration scheme with a constant step size of 0.01s. The state variables were sampled at a frequency of 10 Hz, but the input to the NN-based controller was corrupted with white Gaussian noise of standard deviation 0.01π. Such problems cannot be handled directly with PMP because the control is obtained in an open-loop fashion as discussed in section 1.7. No feedback is solicited at any point in the trajectory, and any control adjustment due to noise will need a re-evaluation of the BVP.

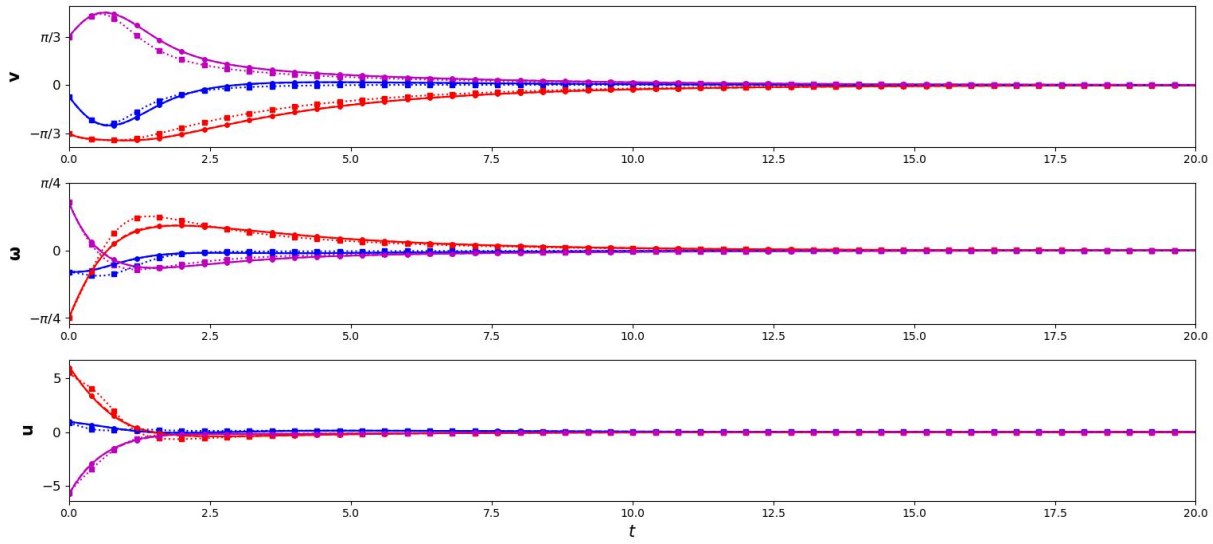


Figure 3.5: Performance of the approximated controller generated from Krener's Toolkit. The initial condition is the same as fig. 3.1: $\boldsymbol{v}(0) = (\phi(0), \theta(0), \psi(0)) = (-1/4, -\pi/3, \pi/3)$ and $\boldsymbol{\omega}(0) = (\omega_1(0), \omega_2(0), \omega_3(0)) = (-1/4, -\pi/4, 0.56)$. Solid blue line: ϕ, ω_1, $u_1^*(t)$; Dashed red line: θ, ω_2, $u_2^*(t)$; Dotted purple line: ψ, ω_3, $u_3^*(t)$. The relevant quantities, as evaluated using Krener's toolkit, are indicated with square markers.

3.2 Application of Krener's Toolkit

As explained in the preceding section, we were interested in examining the feasibility of applying Krener's toolkit to the NN training process. The toolkit approximated the control with a polynomial of degree 3. Figure 3.5 compares the control response from this approximated controller with that of the PMP for a sample initial condition.

We first examined if pre-training the neural network would lead to improved simulation accuracy. Towards this end, we first compared trajectories generated from the approximated controller and the corresponding one from PMP and attempted to measure the average deviation between them. We reiterate that Krener's toolkit only approximates infinite-horizon optimal controllers so that the approximated value function (denoted by V_K) is only a function of the initial condition $\boldsymbol{x}$. Also, for infinite-horizon control design, the value function from PMP, denoted by V_P is calculated only at $t = 0$. Likewise, the constant $\boldsymbol{p}$ is only recorded at $t = 0$. The results are discussed in table 3.1. In table 3.1, the Mean % error for $V = \sqrt{\frac{1}{N}\sum\left(\frac{V_K - V_P}{V_P}\right)^2}$ and the MSE for $V = \sqrt{\frac{1}{N}\sum(V_K - V_P)^2}$. The Mean %

error for $V_{\boldsymbol{x}} = \sqrt{\frac{1}{N}\sum\left(\frac{||\nabla(V_K)-\boldsymbol{p})||_2}{||\boldsymbol{p}||_2}\right)^2}$ and the MSE for $V_{\boldsymbol{x}} = \sqrt{\frac{1}{N}\sum(||\nabla(V_K)-\boldsymbol{p}||_2)^2}$.

We believe that the MSE errors are the most significant metric for comparing the approximate controller with actual PMP results due to the form of the NN loss function eq. (2.6). It is apparent from table 3.1 that the simulation accuracy increases as the domain of the permissible state-space variables is constrained. This is expected as Krener's toolkit is based on a local power-series expansion of the system dynamics and running cost about the equilibrium point and is therefore, expected to be more accurate within a restricted range of perturbation about the equilibrium point.

Ranges ($t_f = 20$)	MSE-error (V)	Mean % sq. error (V) $\times 10^{-2}$	MSE-error ($V_{\boldsymbol{x}}$)	Mean % sq. error ($V_{\boldsymbol{x}}$) $\times 10^{-2}$
$\{\phi,\theta,\psi\} \in R_1$ $\{\omega_1,\omega_2,\omega_3\} \in R_2$	0.18234	4.3619×10^{-4}	1.1828	0.0059063
$\{\phi,\theta,\psi\} \in R_1/1.5$ $\{\omega_1,\omega_2,\omega_3\} \in R_2/1.5$	0.021374	3.4919×10^{-4}	0.39130	0.0035072
$\{\phi,\theta,\psi\} \in R_1/2$ $\{\omega_1,\omega_2,\omega_3\} \in R_2/2$	0.0059298	4.93357×10^{-5}	0.21211	6.9851×10^{-4}
$\{\phi,\theta,\psi\} \in R_1/5$ $\{\omega_1,\omega_2,\omega_3\} \in R_2/5$	8.6378×10^{-5}	9.0489×10^{-7}	0.031175	1.3868×10^{-4}
$\{\phi,\theta,\psi\} \in R_1/10$ $\{\omega_1,\omega_2,\omega_3\} \in R_2/10$	4.7624×10^{-6}	7.2955×10^{-7}	7.917×10^{-3}	2.9484×10^{-4}

Table 3.1: Deviation of Krener's toolkit from PMP, based on 2000 datapoints. $R_1 = [-\pi/3, \pi/3]$, $R_2 = [-\pi/4, \pi/4]$.

The pre-training step involves training the NN using a relatively large amount of data evaluated using Krener's toolkit and then retraining it using numerically correct data from PMP. The hyperparameter μ in the loss function eq. (2.6) is an tunable quantity that must be fixed when the NN is optimized in the pre-training step. We investigated cases for $\mu = 0$ and $\mu = 10$. The dataset for the pre-training involved 8192 trajectories, and to maintain fairness, each NN had the same initialization as the basic NNs, which did not have the pre-training step. This seems to be important to ensure consistency in the observations. The results are discussed in fig. 3.6. For $\mu = 0$ in the pre-training step, improvements in the final performance were observed only for relatively large-sized datasets. A possible reason behind this is that $\mu = 0$, and it is known from fig. 3.3 that neural networks trained this way tend to perform relatively poorly. Substantially better results are obtained when $\mu = 10$ in the pre-training step.

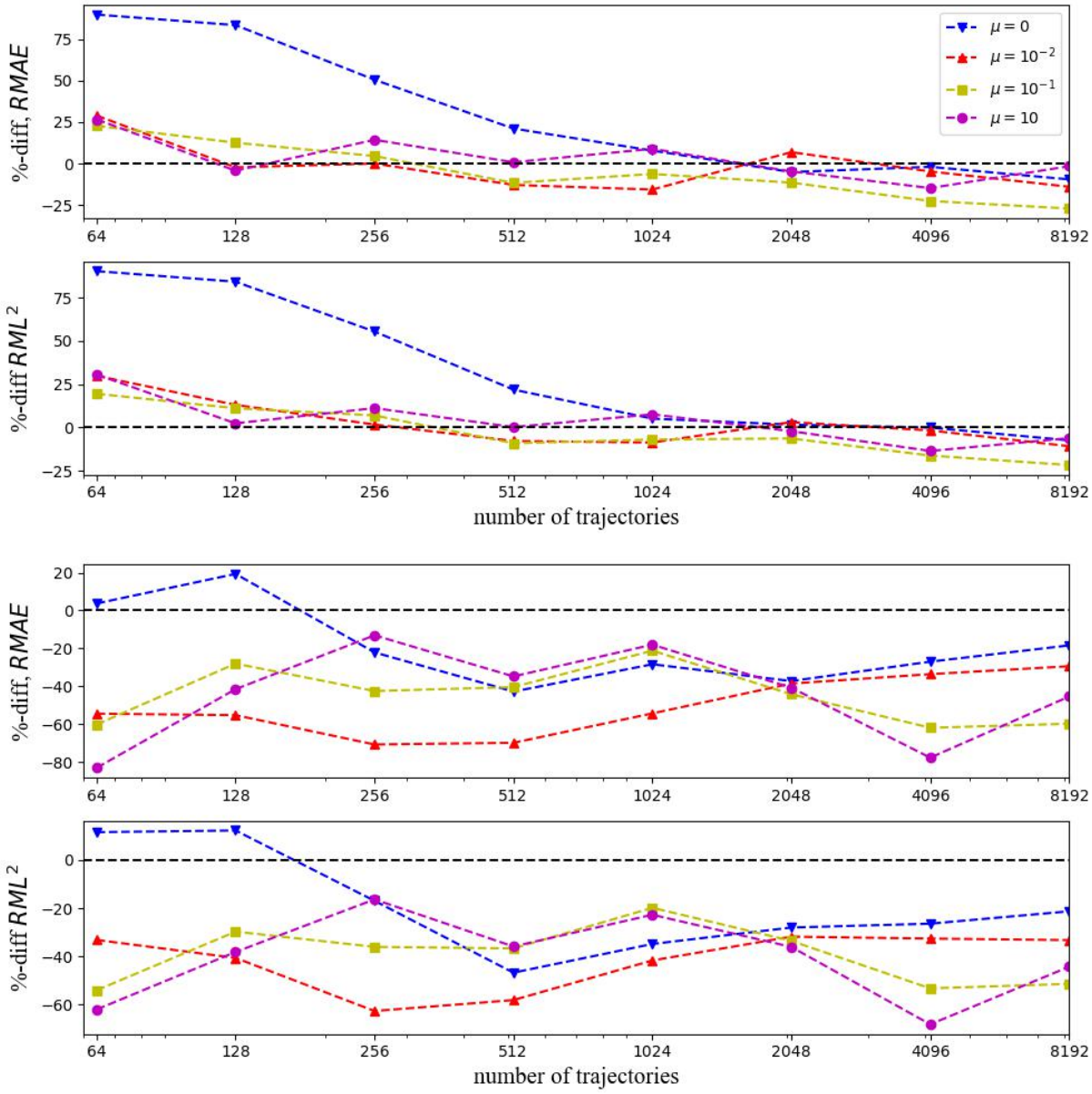


Figure 3.6: % difference in performance metrics for NNs with and without pre-training at the conclusion of 500 training iterations, with initial conditions sampled from eq. (3.5). If X_p and X are a metric with and without pre-training, respectively, then the % error=$(X - X_p)/X \times 100\%$. Top: $\mu = 0$; Bottom: $\mu = 10$ at the pre-training step.

From table 3.1, it is expected that if a more limited range for the initial states is considered, the effect of the pre-training would be much better. To justify this statement further, we repeated this experiment for the following restricted range of permissible values, same as the fourth row of table 3.1:

$$\mathbb{X}_0 = \left\{ \boldsymbol{v}, \boldsymbol{\omega} \in \mathbb{R}^3 \middle| -\frac{\pi}{15} \leq \phi, \theta, \psi \leq \frac{\pi}{15} \text{ and } -\frac{\pi}{20} \leq \omega_1, \omega_2, \omega_3 \leq \frac{\pi}{20} \right\}. \tag{3.8}$$

Fresh datasets were generated using Pontryagin's Principle for this restricted range, following the procedure outlined in the previous section. A dataset containing 8192 trajectories based on this range was generated using Krener's toolkit and used for pre-training purposes. For this step, however, we just used $\mu = 10$ in the pre-training step. Fresh NNs were trained for these datasets (both with and without pre-training). The results are presented in fig. 3.7. Clearly, pre-training has a significant advantage in this case, especially for the hyper-parameters $\mu = 0.1, 10$.

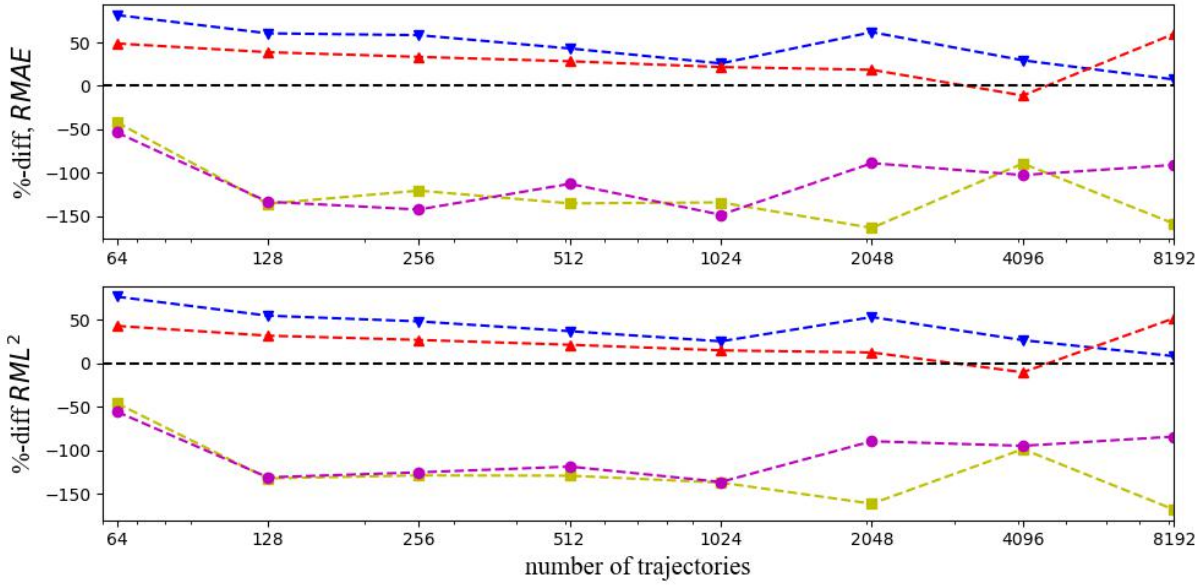


Figure 3.7: % difference in performance metrics for NNs with and without pre-training at the conclusion of 500 training iterations, with initial conditions sampled from the restricted eq. (3.8). If X_p and X are a metric with and without pre-training, respectively, then the % error=$(X - X_p)/X \times 100\%$. The color code is the same as fig. 3.6, with blue: $\mu = 0$, red: $\mu = 0.01$, yellow: $\mu = 0.1$, purple: $\mu = 10$.

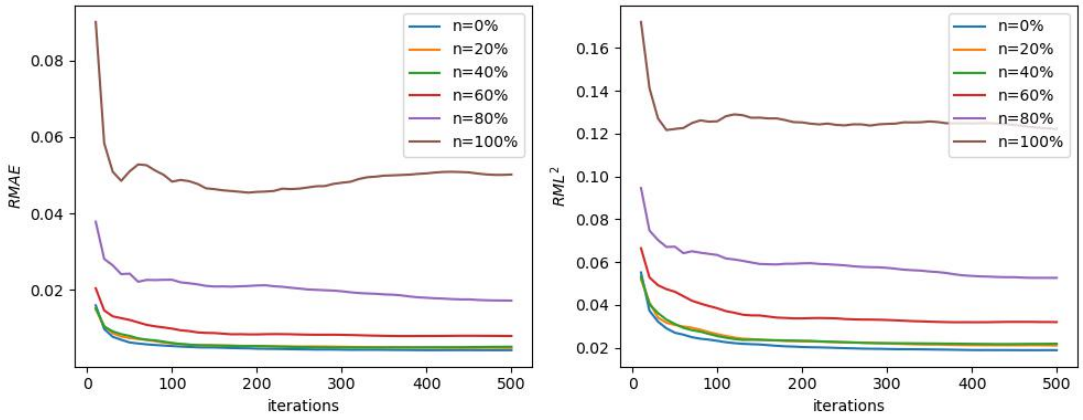


Figure 3.8: NN performance with a selectively chosen dataset consisting of 128 datapoints

We subsequently examined the effect of selectively choosing datapoints based on the estimated magnitudes of $||\nabla V||_2$, as obtained from Krener's toolkit, on the NN performance. In this series of experiments, we generated a fixed set of 8192 datapoints through random sampling from eq. (3.5), evaluated the requisite V and $||\nabla V||_2$ through Krener's toolkit. The datapoints were then sorted as per the magnitudes of $||\nabla V||_2$. Subsequently, for generating a dataset of k datapoints, of which say $n\%$ are to be selected randomly, the first $\lceil nk/100 \rceil$ datapoints are taken from the fixed sorted dataset, while the remaining points are randomly sampled again from eq. (3.5). For the datapoints thus obtained, the BVP is solved to generate the final dataset on which the NN can be subsequently trained. The hyperparameter μ was equal to 10 in the loss function eq. (2.6) throughout the course of this series of experiments. For fairness, all the NNs were subject to the same model parameter initialization as the NNs in fig. 3.3. The performance metrics were measured through an independently generated dataset randomly sampled from eq. (3.5). The results for different percentages n of selectively chosen points with varying dataset sizes are presented in figures 3.8 to 3.12. We observed, however, that such a selective choice actually tended to degrade the final performance of the NN, particularly for very high percentages of the selective choosing and when the number of trajectories is less. In fact, we observed significant convergence issues for $n = 60\%, 80\%$ for the datasets containing 128 and 256 trajectories. Comparable performances are obtained only for relatively large-sized datasets, and for relatively lower selection percentages. This observation is probably from the fact that if the percentage of selectively chosen datapoints is high, the NN tends to better approximate those regions of the admissible set of initial states for which the optimal control requirement is high rather than the entire design domain. When tested on a dataset that

has been uniformly sampled from eq. (3.5), the performance is degraded. The performance improves as the dataset cardinality increases because the region covered by the selectively chosen data points effectively increases.

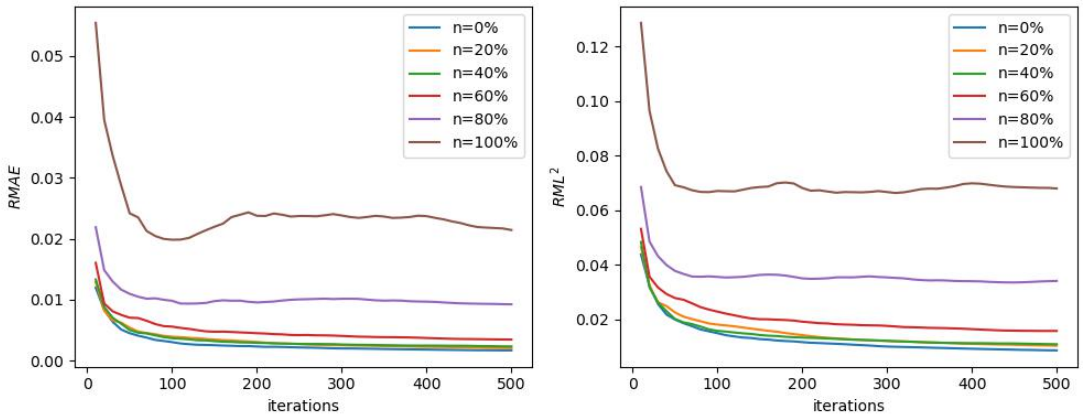


Figure 3.9: NN performance with a selectively chosen dataset consisting of 256 datapoints

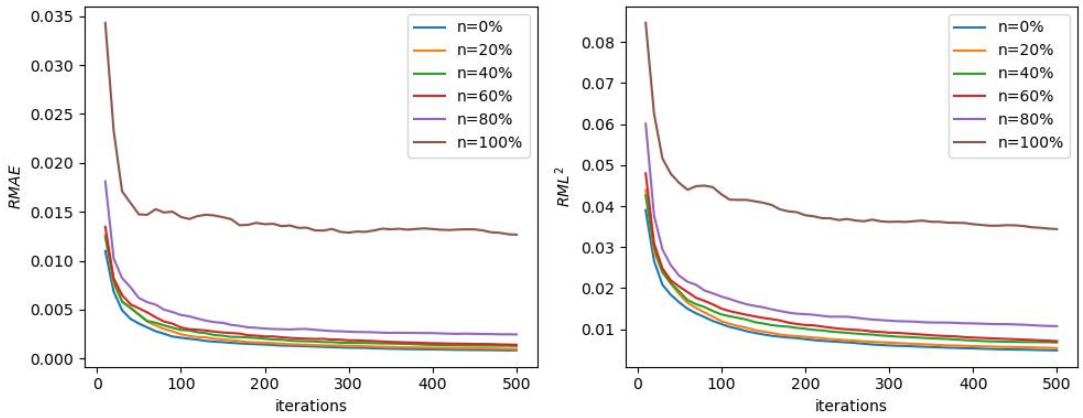


Figure 3.10: NN performance with a selectively chosen dataset consisting of 512 datapoints

Summary: In this chapter, we have examined the optimal control of a 6-dof rigid body. We discussed the application of Krener's toolkit in two stages of the NN training pipeline. Its application to pre-training the NN, in which a large number of datapoints generated from the toolkit is used to train it before retraining using PMP-based data, was found to have promising directions in terms of improved convergence metrics. This is especially

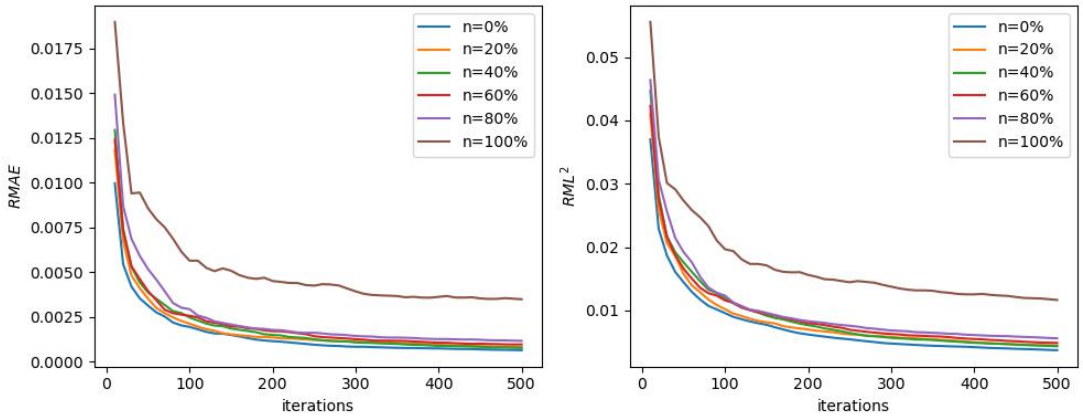


Figure 3.11: NN performance with selectively chosen dataset consisting of 1024 datapoints

true if the admissible range of the initial states is restricted to a small range around its equilibrium location. Its application to selectively identify training datapoints, however, led to less promising results as comparable performance (with little to no improvement) with the base case was noticed only for larger-sized datasets.

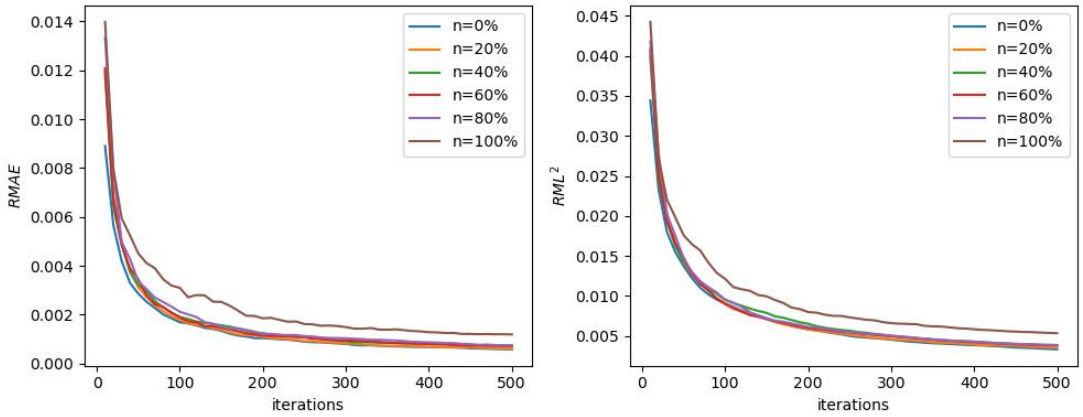


Figure 3.12: NN performance with selectively chosen dataset consisting of 2048 datapoints

Chapter 4

Optimal control of the Burgers' Equation

4.1 Preliminary optimality conditions from PMP

This section is concerned with the optimal control of a Burgers'-like partial differential equation, based on [37, Section 6]. Let $X(t,\xi) : [0,t_f] \times [-1,1]$ satisfy the following PDE:

$$X_t = XX_\xi + \nu X_{\xi\xi} + \alpha X e^{\beta X} + I(\xi)u \tag{4.1}$$

subject to the boundary conditions and numerical parameters:

$$X(t,-1) = X(t,1) = 0 \quad t_f = 8, \quad \nu = 0.2, \quad \alpha = 1.5, \quad \beta = -0.1 \tag{4.2}$$

with a scalar control $u(t)$ acting on the support Ω of the indicator function $I_\Omega(\xi) = 1$ if $-0.5 \leq \xi \leq -0.2$ and 0 otherwise. Consider the optimization of the following quadratic functional:

$$\begin{aligned} \underset{\boldsymbol{u(.)}}{\text{minimize}} \quad J[\boldsymbol{u(.)}] &= \int_t^{t_f} \mathcal{L}(X,u)d\tau + \frac{W_2}{2}||X(t_f,\xi))||^2_{L^2_{(-1,1)}} \\ \mathcal{L}(X,u) &= \frac{1}{2}||X(\tau,\xi)||^2_{L^2_{(-1,1)}} + \frac{W_1}{2}u(\tau)^2 \\ W_1 = 0.1, \quad W_2 = 1 \quad t_f &= 8. \end{aligned} \tag{4.3}$$

Equation 4.1 is discretized using the Chebyshev pseudospectral collocation method [48] in order to obtain a high-dimensional ODE system. This discrete system is then used to

optimize the cost functional eq. (4.3) using PMP. If there are $N_c + 1$ collocation points, the set of Chebyshev points on $[-1, 1]$ is given by [48, Chapter 5]:

$$\xi_j = \cos(j\pi/N_c), \qquad j = 0, 1, 2, ..., N_c. \tag{4.4}$$

Since $\xi_0 = \cos(0) = 1, \xi_{N_c} = \cos(\pi) = -1$, we get $X(t, \xi_0) = X(t, \xi_{N_c}) = 0$ from the boundary conditions.

Consequently, our discretized state effectively used in our simulations is given by:

$$\boldsymbol{x}(t) := [X(t, \xi_1),\, X(t, \xi_2),\, X(t, \xi_3), ...,\, X(t, \xi_n)]^T : [0, t_f] \to \mathbb{R}^n, \quad n = N_c - 1. \tag{4.5}$$

Let $\odot$ represent element-wise multiplication (Hadamard product) and $\mathbb{I}_\Omega$ be the discretized indicator function from eq. (4.1). For a Chebyshev grid containing $N_c + 1$ points, Theorem 7 of [48, Chapter 6] describes a matrix $\tilde{\boldsymbol{D}} \in \mathbb{R}^{N_c+1} \times \mathbb{R}^{N_c+1}$ which approximates the differentiation operator. Thus, if $\boldsymbol{v}$ represents a function discretized on an $N_c + 1$ Chebyshev grid, its discrete first derivative is given by $\tilde{\boldsymbol{D}}\boldsymbol{v}$, and the discrete second derivative is given by $\tilde{\boldsymbol{D}}^2\boldsymbol{v}$. Taking into account the singular Dirichlet boundary conditions, the first spatial differentiation operator for our discretized domain, $\boldsymbol{D}_1$, is obtained from the interior of $\tilde{\boldsymbol{D}}$ by removing its first and last rows along with its first and last column. Likewise, the second spatial differentiation operator, $\boldsymbol{D}_2$, is obtained from the interior of $\tilde{\boldsymbol{D}}^2$ by removing its first and last rows along with its first and last column. Note that as per [48, Chapter 13], if the boundary conditions were non-singular, additional source terms would be present in the discretized equation. Neumann boundary conditions will require additional modifications to the operators $\boldsymbol{D}_1$ and $\boldsymbol{D}_2$. In summary, eq. (4.1) may be discretized into the following system of ODEs:

$$\dot{\boldsymbol{x}} = 0.5\boldsymbol{D}_1(\boldsymbol{x} \odot \boldsymbol{x}) + \nu\boldsymbol{D}_2\boldsymbol{x} + \alpha\boldsymbol{x} \odot e^{\beta\boldsymbol{x}} + \mathbb{I}_\Omega u(t). \tag{4.6}$$

[48, Chapter 12] describes the Clenshaw-Curtiss quadrature formula to integrate a function discretized on a Chebyshev grid on the standard domain $[-1, 1]$. The nodes in this context correspond to the Chebyshev gridpoints, and like any quadrature formula, weights are assigned such that the integral may be approximated as $\sum_1^{N_c+1} w_i X(\xi_i)$, where $N_c + 1$ refers to the number nodes, equivalent to the collocation points here. Since we deal with Dirichlet boundary conditions, the weight vector $\boldsymbol{w} \in \mathbb{R}^n$ is simply obtained by neglecting the first and last weight. Thus, the following approximation for $||X(\tau, \xi)||^2_{L^2_{(-1,1)}}$ is obtained:

$$||X(\tau, \xi)||^2_{L^2_{(-1,1)}} \approx \boldsymbol{w}^T(\boldsymbol{x}(\boldsymbol{\tau}) \odot \boldsymbol{x}(\boldsymbol{\tau})). \tag{4.7}$$

Consequently, the discretized cost functional is:

$$\underset{\boldsymbol{u(.)}}{\text{minimize}} \quad J[\boldsymbol{u(.)}] = \frac{1}{2}\int_t^{t_f} \left(\boldsymbol{w}^T(\boldsymbol{x(\tau)} \odot \boldsymbol{x(\tau)}) + \frac{W_1}{2} u(\tau)^2 \right) d\tau + \frac{W_2}{2} \boldsymbol{w}^T(\boldsymbol{x}(t_f) \odot \boldsymbol{x}(t_f)). \tag{4.8}$$

From PMP, the equations for the co-state dynamics are defined by:

$$\begin{aligned} \dot{\boldsymbol{p}} &= \boldsymbol{w} \odot \boldsymbol{x} - \boldsymbol{x} \odot \boldsymbol{D}_1^T \boldsymbol{p} - \nu \boldsymbol{D}_2^T \boldsymbol{p} - (\alpha e^{\beta \boldsymbol{x}} + \alpha\beta \boldsymbol{x} \odot e^{\beta \boldsymbol{x}}) \odot \boldsymbol{p} \\ u^*(t) &= \frac{1}{W_1} \boldsymbol{p}^T \mathbb{I}_\Omega \\ \boldsymbol{p}(t_f) &= -W_2 \boldsymbol{w}^T \odot \boldsymbol{x}(t_f). \end{aligned} \tag{4.9}$$

This completes the details of the problem studied in this chapter. We do not address the impact of the discretization scheme on the optimal control. However, problems based on discretized PDEs provide an opportunity to examine the scalability of the algorithms discussed in this thesis. We consider discretizations in $n = 20$ and $n = 30$ dimensions. One should note that since we apply a spectral method, the matrices used in eq. (4.6) and eq. (4.9) are full, dense matrices, as compared to the sparse matrices obtained from standard discretization schemes such as forward Euler/Backward Euler. Consequently, this problem is the most computationally intensive problem examined in this thesis.

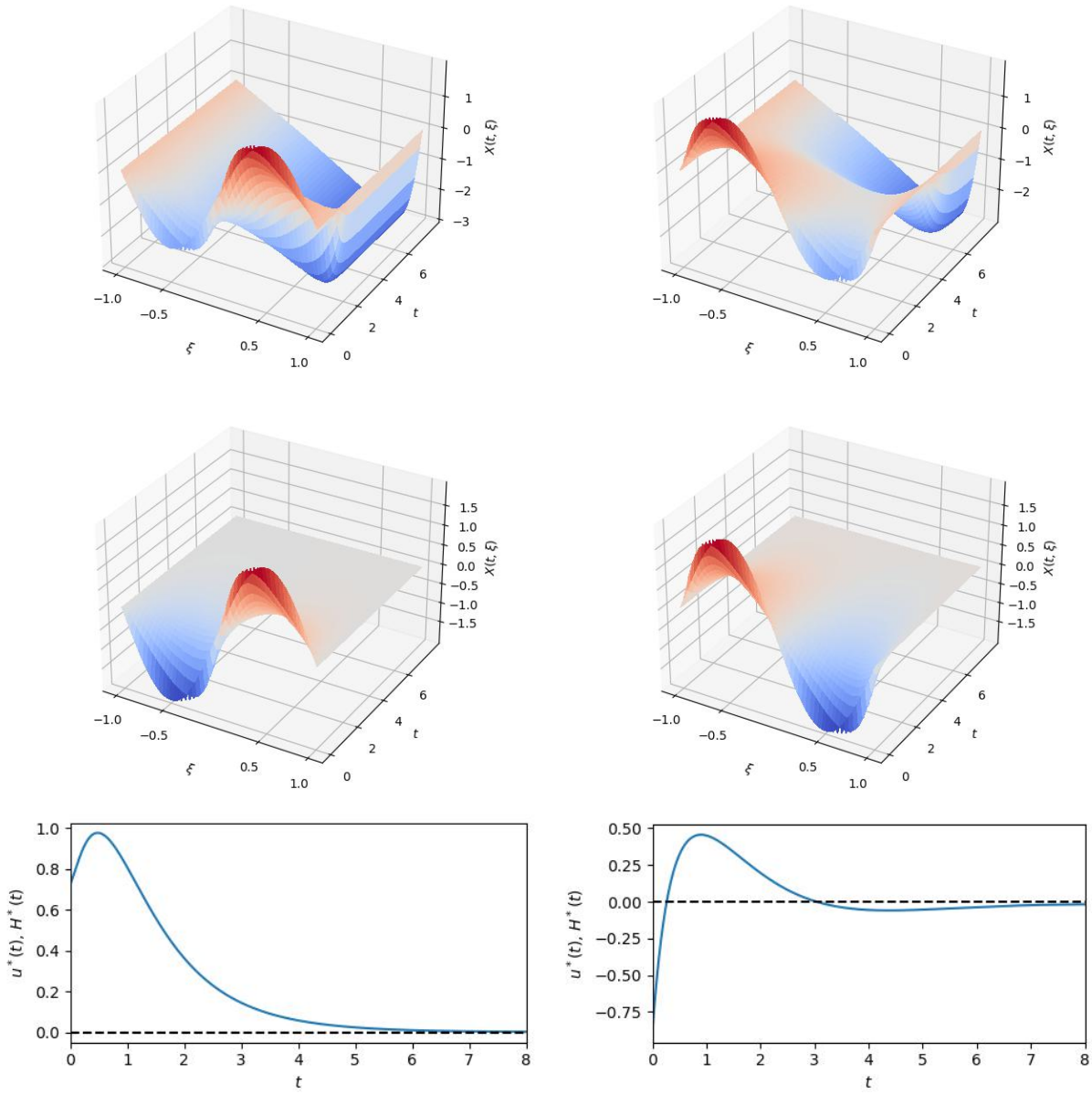


Figure 4.1: Simulations of (4.1) in $n = 20$ dimensions. Left Column: Simulations for initial condition $X_0 = 2\sin(\pi\xi)$ and right Column: $X_0 = -2\sin(\pi\xi)$. Top Row: Uncontrolled dynamics; Middle: Controlled optimal dynamics; Bottom: Respective optimal controls (solid blue) and Hamiltonian (dashed black) from PMP.

4.2 Application of Krener's toolkit

We implemented Krener's toolkit for eq. (4.1) for $n = 30$ dimensions (having 29 state variables), with degree of optimal feedback= 1 and 2. Figure 4.2 compares the controls obtained from PMP and Krener's toolkit for a sample initial condition:

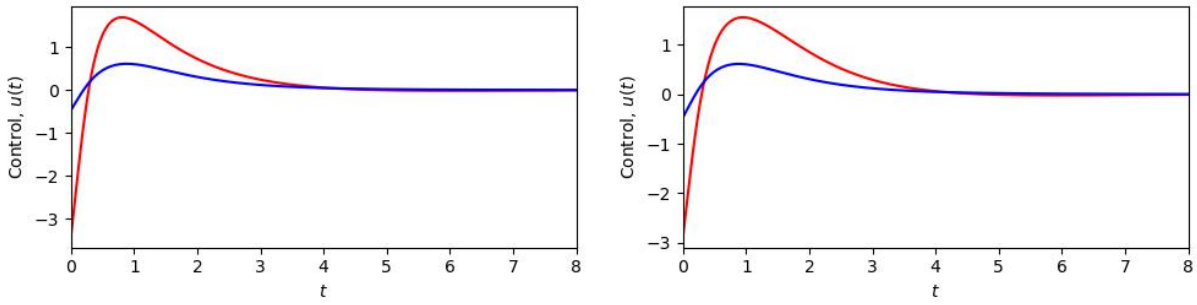


Figure 4.2: Comparison of the controls obtained from PMP (Blue) and Krener's toolkit (Red) for an initial condition $X_0 = 2\sin(\pi\xi)$. Left: degree of optimal feedback=2, Right: degree of optimal feedback=1.

From fig. 4.2, it is apparent that a significant difference exists between the actual controls from PMP and the approximation from Krener's toolkit. Indirectly this implies that a substantial difference exists between the costate indices corresponding to the non-zero values of the discretized indicator function, as evident from (4.9). Furthermore, the loss function described in eq. (2.6) penalizes the entire costate vector. Owing to a potentially large difference in the predicted costates, and based on our observations from section 3.2, we believe that it is infeasible to apply pre-training as performed in chapter 3 to this problem.

From fig. 4.2, one can see that there is only a slight change in the controls obtained as the degree of the optimal control is increased from 1 to 2. On the other hand, as expected from section 2.5, the approximated value function for a degree of optimal control=2 will have $\binom{30}{2} + \binom{31}{3} = 435 + 4495 = 4930$ monomial terms, while the approximated control will $\binom{29}{1} + \binom{30}{2} = 29 + 435 = 464$ monomial terms. Consequently, the number of terms in the expression increases exponentially for high-dimensional systems, and in our computational experiments, we do not go beyond a degree of approximation 2. Implicitly, this is a reflection of the curse of dimensionality. As mentioned earlier, it seems infeasible to apply pre-training to the neural network. However, it is natural that solving the BVP for this high-dimensional coupled system is substantially more expensive computationally

compared to the 6-dof system discussed in chapter 3. Consequently, as an alternative to the *neural network warm-start* approach discussed in [37], in which a neural network trained on a relatively fewer number of optimal trajectories is used to generate an initial guess for the BVP solver, one can use this toolkit to potentially reduce the data generation time.

While investigating this direction, a number of subtle problems arose. The initial guess for the BVP solver requires both the states and the costates. As explained in section 2.5, the value function and the optimal control are approximated as a polynomial expression: not the costates. As a result, a method to appropriately differentiate the polynomial expression for the value function in order to evaluate the costates as per $p = -V_x$ had to be devised. Recall that the set of monomials of degree=k having n system variables has $\binom{k+n-1}{k}$ terms. Our Python code is able to evaluate these terms sequentially using a for-loop-based method that involves a number of condition checks. While it is difficult to vectorize this operation, it is still cheaper to evaluate the value function in this way compared to solving the BVP. The partial differentiation needed for the costate calculations, however, will involve $n.\binom{k+n-1}{k}$ additional calculations, and this was found experimentally to be computationally prohibitive. If a neural network is used, however, these calculations can be performed far more efficiently using backpropagation.

Owing to an exponential increase in the number of terms and unsatisfactory improvement in the approximation of the costates (as well as difficulties in their calculations), in our opinion, it is infeasible to use Krener's toolkit beyond a degree of optimal control=1 for such high dimensional systems. We modified Krener's toolkit to return the approximate (quadratic) optimal cost in the standard form, $\boldsymbol{x}^T\boldsymbol{P}\boldsymbol{x}$, where $\boldsymbol{P}$ is a symmetric matrix so that the approximated costates are given by $-2\boldsymbol{P}\boldsymbol{x}$. This was used to calculate the co-states along the approximated trajectory calculated from Krener's toolkit in the initial guess. Subsequently, we examined the performance of the BVP solver when this approximated trajectory was used as the initial guess for approximately 100 random initial states. Compared to the basic guess (only the initial state at $t = 0$ is non-zero), this method resulted in an average calculation time reduction of 18 seconds. However, we were not able to guarantee that this led to a uniform reduction in computational time, as in some cases, this initial guess performed poorer than the basic guess. Additionally, one must note that CPU times measured in this way have the potential to be corrupted due to the presence of background tasks.

4.3 Neural Network Formulation

For the data generation process, the initial conditions (ICs) are sampled from the following set:

$$\mathbb{X}_0 = \{\boldsymbol{x} \in \mathbb{R}^n | -2 \leq x_j \leq 2, j = 1, 2, 3, ..., n\} \tag{4.10}$$

For each of the discretized problems involving $n = 20$ and $n = 30$ states, training datasets are generated from $128, 145, 163$ initial conditions, while the test dataset is generated from 100 initial conditions, each sampled independently of one another. As the controller designed for this problem is a moving horizon controller, time is taken as an input into the neural network. For each IC, a BVP is solved (optionally using process-based concurrency techniques), and the solution obtained from the Python solver generally contains more than 1000 default time instants. These default time instants and the corresponding solutions are used in the generated dataset; the solutions are not interpolated onto a uniform time grid. For each time instant, the value function is evaluated as described in section 2.2. For sanity purposes, the optimal Hamiltonian is evaluated along the trajectory, even though it is known to be a constant from theory. The data format is, therefore, of type $(t, \boldsymbol{x}^*(t), \boldsymbol{p}^*(t), H^*(t), V(t, t_f))$. Due to the large number of time instants returned by default, it is sufficient to solve a fewer number of trajectories, even though these BVPs are the most computationally intensive problems considered in this thesis. The NN architecture is the same as problem 1 (a standard feedforward network of 3 hidden layers, each having 64 neurons with tanh activation functions). Similar to problem 1, we have not implemented adaptive data generation into the NN training process. Our implementation is based on an open-loop approach in which the trajectories are generated from the initial conditions prior to starting the training; it is not expanded subsequently based on the NN performance. Likewise, we do not consider a training termination criteria and allow the training to continue for 500 epochs.

In section 2.4, the rationale behind the loss function for PINNs was discussed. In [37], the principal reference on which this thesis is based, the loss function is given by eq. (2.6). However, for moving-horizon controllers, one can think of an additional constraint relationship between the gradient of the value function with respect to time, V_t, and the optimal Hamiltonian, based on eq. (1.9). Consequently, we investigated the inclusion of this loss through a term $\mu \times (V_t - H^*(t, \boldsymbol{x}^*, \boldsymbol{p}^*, \boldsymbol{u}^*))^2$, where μ is the same hyper-parameter used in the standard loss function. For fairness, all NNs having the same n have the same initialization. Table 4.1 and fig. 4.3 compare the NN performance when this additional term is taken into account. Overall, an improvement in the NN performance was observed: in particular, the response is seen to be much more stable.

	Metric	Trajectories=128		Trajectories=145		Trajectories=163	
		No V_t	With V_t	No V_t	With V_t	No V_t	With V_t
n=20	$RMAE \times 10^3$	9.37	7.98	8.77	7.13	8.1	6.39
	$RML^2 \times 10^2$	2.43	2.39	2.52	2.50	2.32	2.18
	Time(s)	1245.22	1284.57	1622.55	1382.29	1798	1509.53
n=30	$RMAE \times 10^3$	12.5	11.9	10.865	9.72	9.26	9.19
	$RML^2 \times 10^2$	3.44	3.44	3.4	3.06	3.07	2.93
	Time(s)	1549.51	1281.44	1453	1448	1881	1622

Table 4.1: Relevant metrics at the end of 500 training iterations when the term involving V_t is included in the loss function for problem 2. The metrics used are defined in eq. (2.7). Note: The time readings have a strong tendency to be altered from background tasks running in the machine, nevertheless, the training times obtained are comparable.

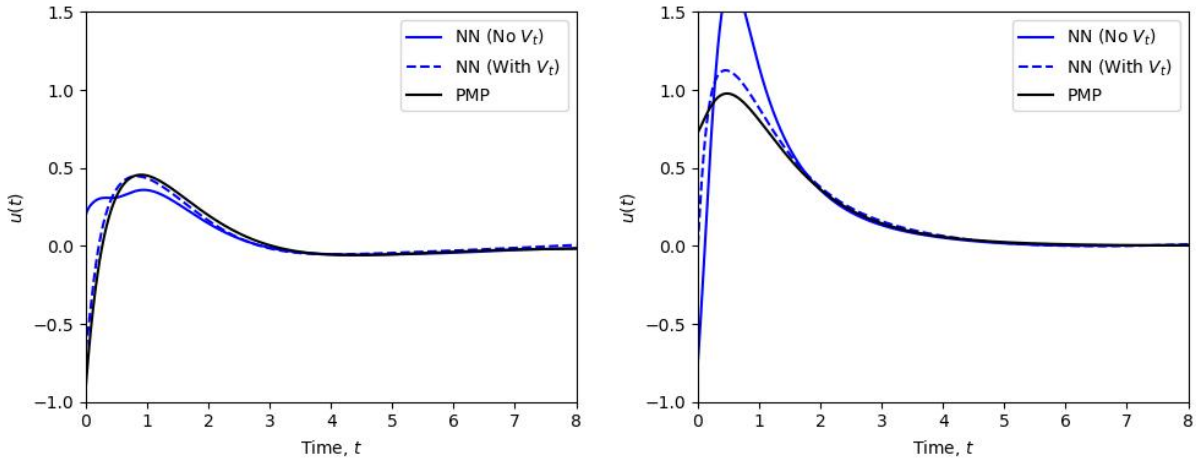


Figure 4.3: Comparison of the controls obtained from the neural networks (with and without the V_t-based losses, for an $n = 20$ grid discretization. Left: $X_0 = -2\sin(\pi\xi)$, Right: $X_0 = 2\sin(\pi\xi)$.

Summary: In this chapter, we have examined the optimal stabilization of a high-dimensional dense ODE system arising from the discretization of a Burgers'-like PDE. The application of Krener's toolkit to generate a guess for the BVP solver was found to result in improvements in the average time of solver convergence. We examined the effect of the addition of a loss based on V_t and the optimal Hamiltonian to the usual loss function eq. (2.6). This was found to result in small improvements in the NN convergence and a relatively more stable controller output.

Chapter 5

Optimal Control of the damped wave equation

5.1 A preliminary discussion

We first discuss the motivation behind the study of this problem. Let us first consider the 1-D uncontrolled heat equation with homogeneous Dirichlet boundary conditions, satisfied by $X(t,\xi) : [0,t_f] \times [0,1] \to \mathbb{R}$:

$$X_t = X_{\xi\xi}, \qquad X(t,0) = X(t,1) = 0.$$

On performing a separation of variables and solving the PDE, the solution obtained of the form: $X(t,\xi) = \sum_{n=1}^{\infty} c_n(0)\phi_n(\xi)e^{-\lambda_n^2 t}$, where the eigenfunctions are given by $\phi_n(\xi) = \sqrt{2}\sin(n\pi\xi)$ and the eigenvalues are $\lambda_n = n\pi$. Clearly, all the eigenvalues are real and exhibit a rapid decay. Their contribution to the state as time progresses becomes negligible.

The viscous Burgers' equation with homogeneous boundary conditions is similar to the problem addressed in chapter 4. Let $X(t,\xi) : [0,t_f] \times [0,1] \to \mathbb{R}$ satisfy the PDE:

$$X_t + XX_\xi = \nu X_{\xi\xi}, \qquad X(t,0) = X(t,1) = 0.$$

This equation is non-linear, and eigenfunctions or eigenvalues cannot be directly calculated. The Cole-Hopf transformation can, however, be used to obtain an analytic solution. Its study in the context of Koopman Theory and data-driven methods like Dynamic Mode Decomposition [38][42] indicates that it is possible to represent the system state with a small number of transient dynamic structures.

Let us now consider the uncontrolled damped wave equation with homogeneous Dirichlet conditions, satisfied by $X(t,\xi) : [0,t_f] \times [0,1] \to \mathbb{R}$:

$$X_{tt} + \gamma X_t = X_{\xi\xi} \qquad X(t,0) = X(t,1) = 0.$$

The eigenfunctions of $\partial^2/\partial\xi^2$ in this context are still given by $\phi_n(\xi) = \sqrt{2}\sin(n\pi\xi)$, but the eigenvalues are now $-\gamma/2 + i\sqrt{(n\pi)^2 - (\gamma/2)^2} \approx -\gamma/2 + in\pi$. The damping coefficient $\approx \gamma/(2n\pi)$ actually decreases for higher spectral modes. Thus, while the wave equation is stable, the higher spectral modes do not exhibit a similar rapid decay as the heat equation. Neither can it be assumed that the state evolution is governed by a small number of underlying dynamic structures, similar to the Burgers' equation. Consequently, we examine the application of the method discussed thus far to the optimal control of the 1-D damped wave equation.

5.2 Optimality conditions from PMP

We first develop the optimal control for this equation with uniformly distributed viscous damping. Let $X(t,\xi) : [0,t_f] \times [0,1] \to \mathbb{R}$ satisfy the following PDE:

$$X_{tt} + \gamma X_t = X_{\xi\xi} + \mathbb{I}_\Omega(\xi)u(t) \tag{5.1}$$

subject to the conditions:

$$X(t,0) = X(t,1) = 0, \qquad t_f = 10, \qquad \gamma = 0.1 \tag{5.2}$$

and subject to a scalar control $u(t)$ on the support Ω of indicator function $\mathbb{I}_\Omega(\xi) = 1$ if $0.35 \leq \xi \leq 0.55$ and 0 otherwise.

To develop a discrete version of eq. (5.1), we employ a Galerkin method with the orthonormal set of eigenfunctions $\{\phi_n(\xi)\}_{n=1}^{n=k} = \{\sqrt{2}\sin(n\pi\xi)\}_{n=1}^{n=k}$ as the Galerkin basis. Consequently, we claim that $X(t,\xi) \approx \sum_{n=1}^{k} c_n(t)\phi_n(\xi)$, where for a given n, c_n is termed the $n-th$ spectral coefficient. The approximation on substitution into the PDE yields the following:

$$\sum_{n=1}^{k} \ddot{c_n}(t)\phi_n(\xi) + \gamma \sum_{n=1}^{k} \dot{c_n}(t)\phi_n(\xi) = -\sum_{n=1}^{k} n^2\pi^2 c_n(t)\phi_n(\xi) \tag{5.3}$$

where we use the fact that $\phi_n''(\xi) = -n^2\pi^2\phi_n(\xi)$. Using the orthonormality conditions, we get an *uncoupled system of ODEs* for the spectral coefficients $c_n(t)$:

$$\ddot{c_n}(t) + \gamma \dot{c_n}(t) = -n^2\pi^2 c_n(t) \tag{5.4}$$

We henceforth consider systems of $k = \{5, 10, 15, 20, 25, 30\}$ spectral coefficients. In state-space representations of linear systems, the matrix relating the uncontrolled system dynamics $\boldsymbol{A} : \dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x}$ is called the state matrix. If we take the system variables to be $(\boldsymbol{x_1}, \boldsymbol{x_2})$ such that their n-th elements are c_n and $\dot{c}_n$, then the uncontrolled dynamics are:

$$\frac{d}{dt}\begin{pmatrix}\boldsymbol{x_1}\\ \boldsymbol{x_2}\end{pmatrix} = \begin{pmatrix}\boldsymbol{0}_{n\times n} & \boldsymbol{I}_{n\times n}\\ -\boldsymbol{\mathcal{M}} & -\gamma\boldsymbol{I}_{n\times n}\end{pmatrix}\begin{pmatrix}\boldsymbol{x_1}\\ \boldsymbol{x_2}\end{pmatrix}.$$

Here $\boldsymbol{\mathcal{M}}$ is a diagonal matrix defined as $M_{ii} = n^2\pi^2; M_{ij} = 0$. The state matrix obtained, however, has high condition numbers of order $\sim \mathcal{O}(n^2\pi^2)$. Consequently, for numerical stability purposes, it is recommended to represent this system with the independent variables $(\boldsymbol{x_1}, \boldsymbol{x_2})$, such that their $n-th$ elements are $n\pi c_n$ and $\dot{c_n}$ respectively. The (controlled) dynamics are then given by:

$$\frac{d}{dt}\begin{pmatrix}\boldsymbol{x_1}\\ \boldsymbol{x_2}\end{pmatrix} = \begin{pmatrix}\boldsymbol{0}_{k\times k} & \boldsymbol{M}\\ -\boldsymbol{M} & -\gamma\boldsymbol{I}_{k\times k}\end{pmatrix}\begin{pmatrix}\boldsymbol{x_1}\\ \boldsymbol{x_2}\end{pmatrix} + \begin{pmatrix}\boldsymbol{0}_{k\times 1}\\ \boldsymbol{B}\end{pmatrix}u(t) \tag{5.5}$$

where, $B_n = \int_0^1 \mathbb{I}_\Omega(\xi)\phi_n(\xi)d\xi = \int_0^1 \sqrt{2}\mathbb{I}_\Omega(\xi)\sin(n\pi\xi)d\xi = \frac{\sqrt{2}}{n\pi}(\cos(0.35n\pi) - \cos(0.55n\pi))$. $\boldsymbol{M}$ is a diagonal matrix defined as $M_{ii} = i\pi$; $M_{ij} = 0$. The state matrix in this form has condition numbers of order $\sim \mathcal{O}(n)$, which are much lower than the earlier representation. As a (successful) validation step, the eigenvalues of the state matrix were calculated using SciPy and compared to the analytical ones from the characteristic equations of eq. (5.4); their approximate value for small γ is $-\gamma/2 \pm i(n\pi)$, as expected from section 5.1.

It is assumed that the initial condition $X(0, \xi) = f(\xi)$ is a continuous function, rather than a discrete one specified only at certain points. This enables the evaluation of the $n-th$ initial spectral coefficient $c_n(0) = \int_0^1 f(\xi)\phi_n(\xi)d\xi$ using standard SciPy quadrature-based integration routines. As a computational simplification, only cases where the initial perturbed state is at rest are considered. Consequently, the initial velocity $X'(0, \xi) = 0$.

The following quadratic cost functional is optimized:

$$\begin{aligned}\underset{\boldsymbol{u(.)}}{\text{minimize}} \quad J[\boldsymbol{u(.)}] &= \int_t^{t_f} \mathcal{L}(X, X', u)d\tau + \frac{W_4}{2}||X(t_f, \xi)||^2_{L^2_{(-1,1)}} + \frac{W_4}{2}||X'(t_f, \xi)||^2_{L^2_{(-1,1)}}\\ \mathcal{L}(X, X', u) &= \frac{W_1}{2}||X(\tau, \xi)||^2_{L^2_{(-1,1)}} + \frac{W_2}{2}||X'(\tau, \xi)||^2_{L^2_{(-1,1)}} + \frac{W_3}{2}u(\tau)^2\\ W_1 = 1, \quad W_2 = 1 \quad W_3 = 0.01 \quad W_4 = 1 \quad t_f &= 10.\end{aligned} \tag{5.6}$$

We note that as an orthonormal basis for this set of equations, they satisfy the property $\langle x, x\rangle = \langle \sum_n c_n\phi_n, \sum_j c_j\phi_j\rangle = \sum_n c_n^2$. Since we consider a *Galerkin basis* of k eigenfunctions, we need to provide a discrete version of this cost functional in terms of $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$.

As $x_{1n} = n\pi c_n$, $c_n^2 = x_{1n}^2/(n^2\pi^2)$, so that $||X||^2 \approx \sum_1^k x_{1n}^2/(n^2\pi^2)$. As $x_{2n} = \dot{c}_i$ is an independent vector, $||X'||^2 \approx \sum_1^k x_{2n}^2$. Consequently, the approximated running and terminal costs are:

$$\begin{aligned} \mathcal{L}(\boldsymbol{x_1}, \boldsymbol{x_2}, u) &= \frac{W_1}{2} \sum_{n=1}^{k} \frac{x_{1n}^2}{n^2\pi^2} + W_2 \sum_{n=1}^{k} \frac{x_{2n}^2}{2} + \frac{W_3}{2} ||\boldsymbol{u}||^2 \\ K(\boldsymbol{x_1}, \boldsymbol{x_2}) &= \frac{W_4}{2} \sum_{n=1}^{k} \frac{x_{1n}^2}{n^2\pi^2} + W_4 \sum_{n=1}^{k} \frac{x_{2n}^2}{2}. \end{aligned} \tag{5.7}$$

The equations for the co-state, along with the boundary conditions at t_f, thus become:

$$\begin{aligned} \dot{p}_{x_{1i}} &= \frac{W_1 x_{1i}}{n^2\pi^2} + i\pi p_{x_{2i}} \\ \dot{p}_{x_{2i}} &= W_2 x_{2i} + \gamma p_{x_{2i}} - i\pi p_{x_{1i}} \\ p_{x_{1i}}(t_f) &= -W_4 \frac{x_{1i}}{i^2\pi^2} \\ p_{x_{2i}}(t_f) &= -W_4 x_{2i}. \end{aligned} \tag{5.8}$$

We numerically examine the controls obtained for the initial condition (fig. 5.1):

$$X(0) = \begin{cases} 4\xi - 1 & \forall\, 0.25 \leq \xi \leq 0.50 \\ 3 - 4\xi & \forall\, 0.50 \leq \xi \leq 0.75 \\ 0 & \forall\, \xi \in [0, 0.25] \cup [0.75, 1]. \end{cases} \tag{5.9}$$

The results obtained from PMP for this initial condition when $k = 30$ spectral coefficients are considered are shown in fig. 5.2 and fig. 5.3.

5.3 Neural Network Implementation and Performance

As explained in section 5.1, the most interesting feature of the wave equation is that the higher system modes do not display a rapid time decay similar to the heat or Burgers' equation. The system eigenvalues (for our boundary conditions) for small γ are given by $\approx -\gamma/2 \pm i(n\pi)$. The damping coefficient $\approx \gamma/(2n\pi)$ decreases for higher spectral modes. Another interesting feature is that while the PDE 5.1 is originally described in the physical space, all the relevant calculations are performed in an orthonormal Galerkin basis obtained from the system eigenfunctions.

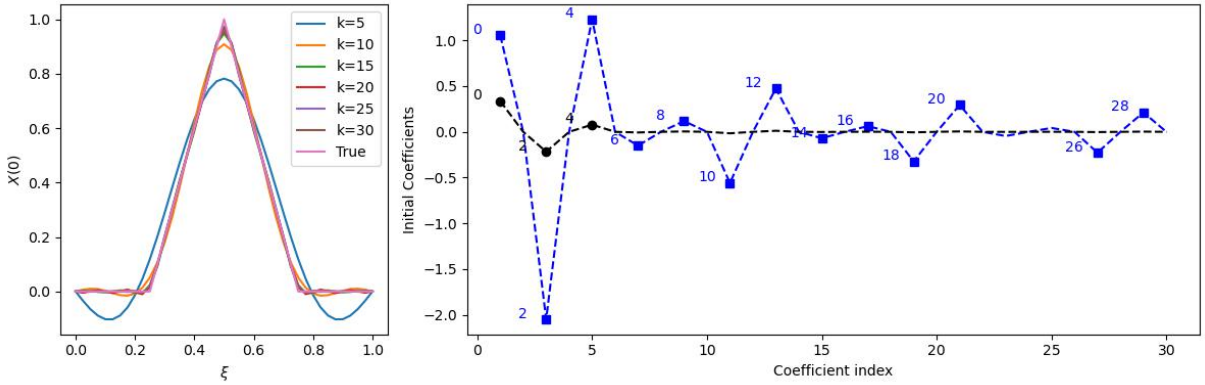


Figure 5.1: Numerical and spectral aspects of eq. (5.9). Right fig, black line: $c_n(0)$; blue line: $n\pi c_n(0)$. Blue square markers indicate indices where $|c_n(0)| > 0.05$, while black ones indicate indices where $|n\pi c_n(0)| > 0.05$. While $|c_n(0)| < 0.05 \forall n > 4$, this is not true for $|n\pi c_n(0)|$. Left fig: $f_k(\xi) = \sum_{n=1}^{k} \sqrt{2} c_n \sin(n\pi\xi)$. If $E_k = \sqrt{\int_0^1 (X(0) - f_k)^2 d\xi}$ and $X(0)$ is given by eq. (5.9), $E_5 = 0.0817$, $E_{10} = 0.022$, $E_{15} = 0.0099$, $E_{20} = 0.0086$, $E_{25} = 0.0068$, $E_{30} = 0.0049$

,

Owing to our use of the Galerkin approximation, changes must be made in the data generation process. The first problem we discussed involved an ODE, and no transformation was applied to the initial conditions for which the optimal trajectories were calculated using PMP. Even for the second problem involving a Burgers'-like PDE, the initial states used in the data generation process were obtained in a discrete form directly from a known domain. However, for the wave equation, the initial spectral coefficients must be evaluated through an integration with the appropriate eigenfunction. To use more accurate quadrature-based integration routines, the initial condition $f(\xi)$ must be provided as a continuous, rather than a discrete function. Consequently, points are sampled from the following set:

$$\mathbb{X}_0 = \{\boldsymbol{x} \in \mathbb{R}^{21} | -1 \leq x_j \leq 1, j = 1, 2, 3, ..., 19, \, x_0 = x_{20} = 0\} \tag{5.10}$$

and the spatial domain $[0, 1]$ is partitioned into an equispaced grid of 20 elements, comprising of 21 gridpoints $\xi_i = i/20$, $i = 0, 1, 2, ..., 20$. We make $f(\xi_i) = x_i$ and define it for intermediate $\xi : \xi_i \leq \xi \leq \xi_{i+1}$ using the simple piecewise linear scheme: $f(\xi) = \frac{\xi - \xi_i}{\xi_{i+1} - \xi_i}(f(\xi_{i+1}) - f(\xi_i)) + f(\xi_i)$.

The *continuous function* $f(\xi)$ is now used in the data-generation process, as the spec-

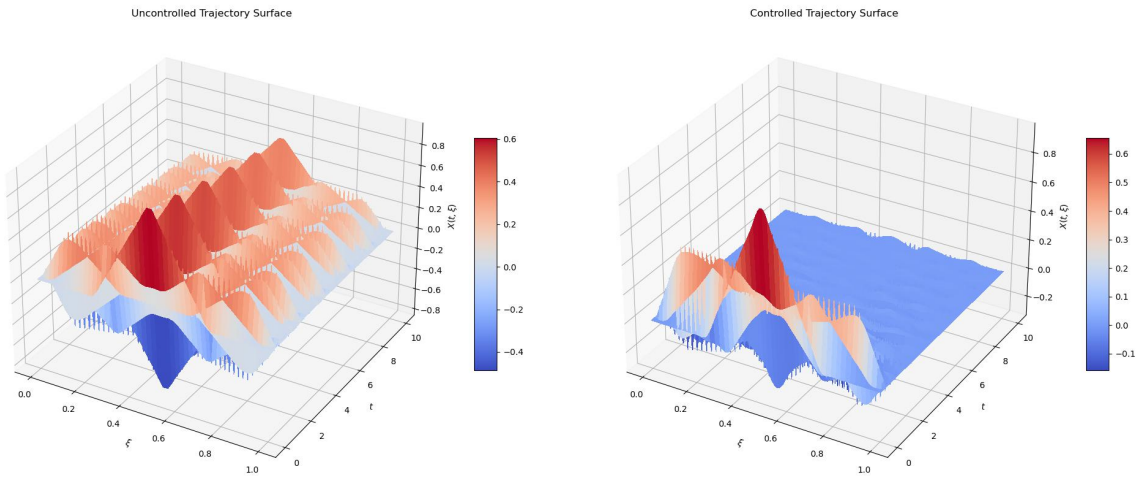


Figure 5.2: Left: Simulated trajectory involving $k = 30$ spectral coefficients for the initial state eq. (5.9) with no acting control. Right: Simulated trajectory for the same initial state when subject to optimal control with cost functional eq. (5.6).

tral coefficients $c_n(0) = \langle f(\xi), \phi_n(\xi) \rangle$ may now be calculated. The Pontryagin BVP, as described in section 5.2, is subsequently solved to generate datasets for training the NN. We generate individual training (128 trajectories) and test (100 trajectories) datasets for initial conditions having $k = \{5, 10, 15, 20, 25, 30\}$ non-zero spectral coefficients. We re-iterate that our training procedure is open-loop and that the training dataset is generated independent of the intermediate performance of the neural network. It is also important to note that although $-1 \leq f(\xi) \leq 1$ for all $\xi \in [0, 1]$, such a bound cannot be obtained for the spectral coefficients.

The BVP described in section 5.2 is dependent on the number of non-zero spectral coefficients. If we consider problems involving k such coefficients, it implies that the system is governed by $4k$ equations. Consequently, the NN architecture must change as the number of coefficients changes. Thus, while working with a dataset having simulations involving k spectral coefficients, the NN has 4 hidden layers of 64 Neurons each, with Tanh activation functions, with the first layer having $2k$ inputs.

In order to examine the performance of the NN-based optimal control scheme, we trained separate NNs using individually generated datasets consisting of simulations having different numbers of spectral coefficients, as discussed earlier. The loss function has the hyperparameter $\mu = 10$. The V_t based loss as described in section 4.3 is excluded, as this was found to cause numerical convergence issues, especially for higher values of k. A

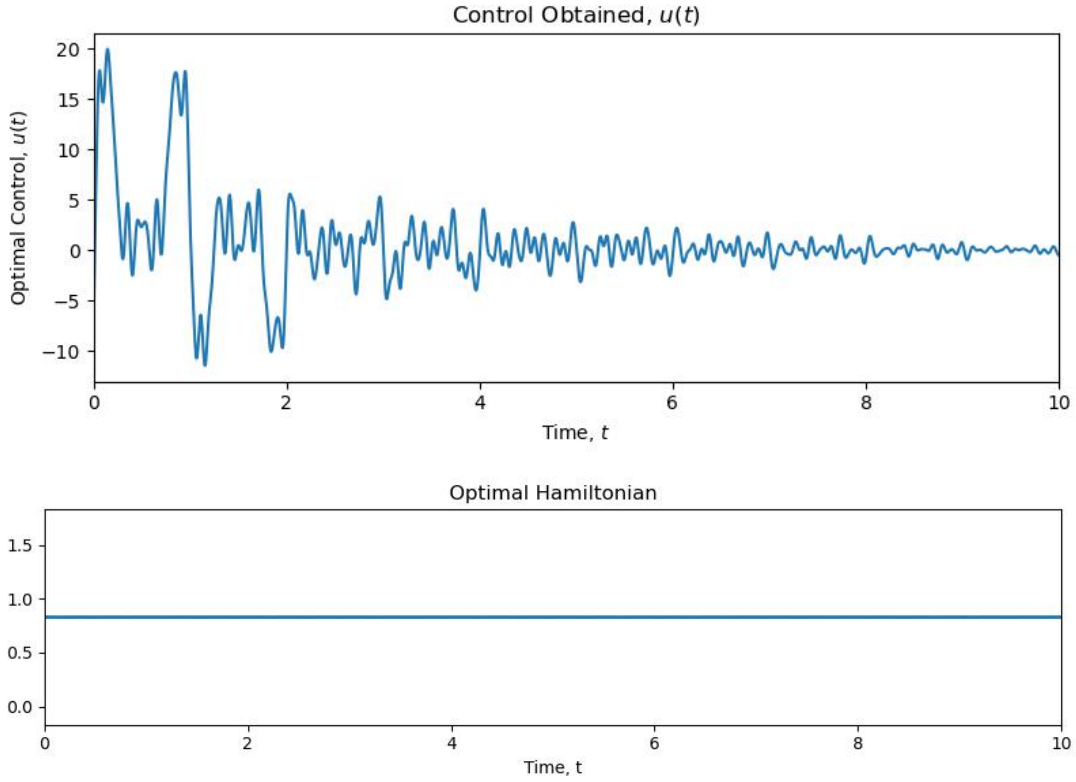


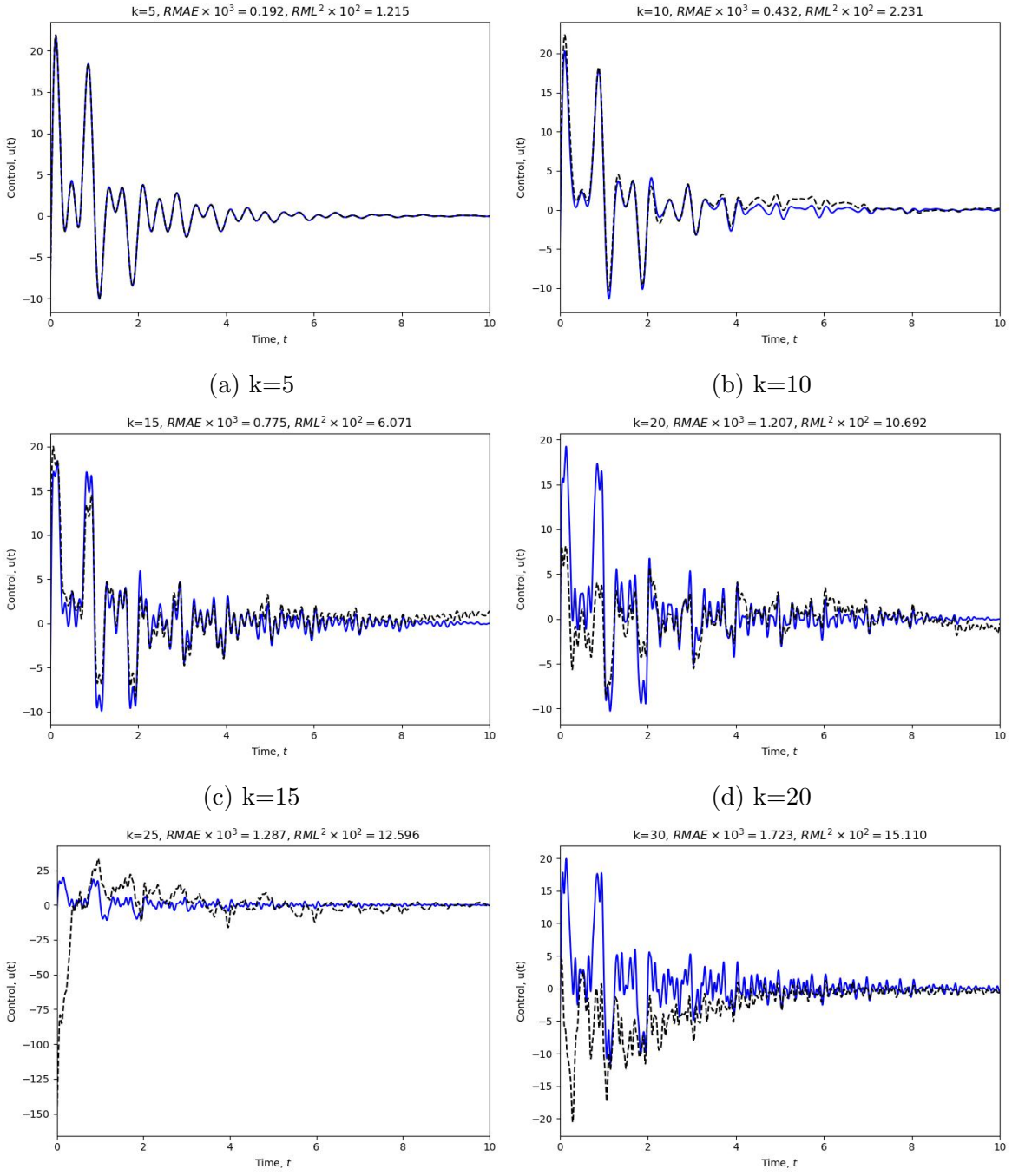
Figure 5.3: Top: Optimal control $u(t)$ obtained for the Cost functional eq. (5.6) with $k = 30$ spectral coefficients, and Bottom: The Optimal Hamiltonian obtained on solving the boundary value problem. It is seen to be numerically constant, as expected from theory, but its value is not zero.

potential reason behind this is that the NN is unable to satisfactorily approximate the controls required by the higher system modes; and the presence of such modes tends to adversely affect the NN training, as will be explained below. We then considered the expansion of eq. (5.9) in terms of the eigenfunctions and truncated it at k terms. The NN trained on the dataset involving simulations with k spectral coefficients was applied to this modified initial condition. For fairness, the PMP-based BVP was solved for this modified initial condition as well and compared to the NN-based control. The results are shown in fig. 5.4. Satisfactory controls were obtained for the NNs trained on simulations involving low $k = 5, 10, 15$ spectral coefficients. However, for higher $k = 20, 25, 30$, the controls obtained were unsatisfactory.

Taking advantage of the fact that the ODE system eq. (5.4) is decoupled, we examined the performance of the NN relative to the PMP for the set of ICs based on the eigenfunctions, $X(0) = \phi_n(\xi) = \sqrt{2}\sin(n\pi\xi)$. Through this examination, we attempted to identify the cause behind the unsatisfactory performance of the NNs involving larger numbers of spectral coefficients. We first considered the case for $k = 5$, for which the best performance was observed. For this case, we observed significant divergence for $n = 4, 5$. For larger $k > 15$, we observed that the NNs performed significantly poorer compared to PMP, even for the lower eigenfunctions. Consequently, we believe that the NN-based methods, as examined in this thesis, are best applicable to systems that can be approximated with a small number of decaying system modes. If higher system modes that do not decay faster than the lower ones are present (similar to the wave equation examined here), they can potentially corrupt the NN training.

To justify our claim further, we generated new datasets involving $k = 15, 20, 25, 30$ spectral coefficients such that any initial coefficient corresponding to $n \geq 5$ is truncated to zero. This does not imply that these datasets are equivalent to the earlier dataset based on 5 non-zero spectral coefficients: the higher order coefficients are altered during the simulation through the control term $\boldsymbol{B}$ as the scalar control $u(t)$ acts on all the non-zero entries. On retraining the NN based on this dataset, the convergence of the initial conditions given by the lower eigenfunctions was observed to be much better. However, such a controller designed primarily using a number of lower system eigenmodes is susceptible to *spillover* [36], as the control still affects the higher system modes, potentially leading to system instabilities.

Consequently, in the validation step, eq. (5.9) was expanded in terms of the eigenfunctions and *truncated at k terms, not 5*. Thus, all the initial spectral coefficients $c_n(0) : 5 \leq n \leq k$ are non-zero. The corresponding NN was then applied to predict the optimal control for this modified initial condition. The numerical results are presented in fig. 5.5. Clearly, substantially better results are obtained fig. 5.4 without spillover effects. A potential reason behind this observation is that the initial condition eq. (5.9) does not have a substantial contribution from higher-order spectral components (evident from fig. 5.1, $|c_n(0)| < 0.05$ for $n > 5$). Also, while the higher spectral coefficients for $n > 5$ are initially zero, they are modified in the simulations as the corresponding control term B is non-zero. This potentially improves the robustness of the controller. However, a similarly good performance cannot still be assured for initial conditions that have a substantial amount of high-order spectral components (physically, this would correspond to non-smooth/irregular initial conditions), as a good controller could only be designed using datasets in which their initial value was zero.



(a) k=5

(b) k=10

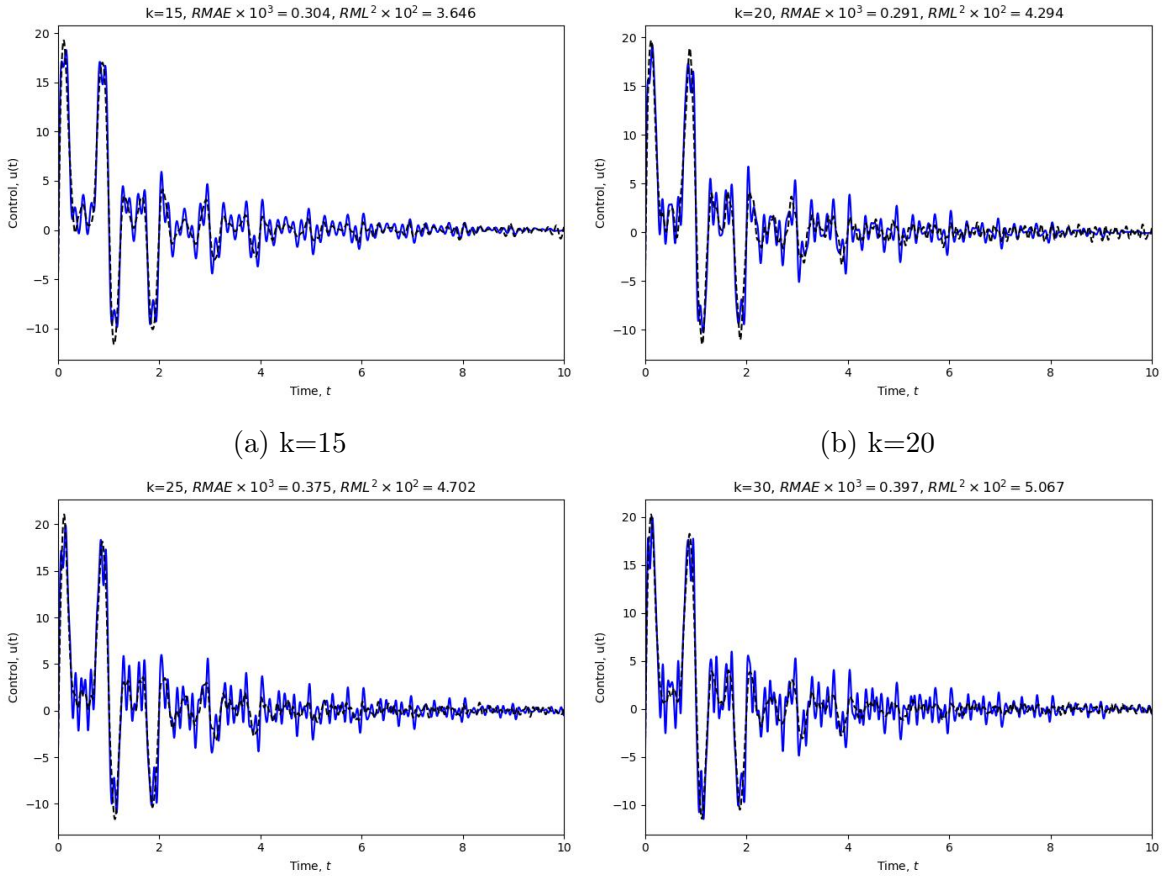
(c) k=15

(d) k=20

(e) k=25

(f) k=30

Figure 5.4: Comparison of control obtained from PMP and the NN trained on datasets having k non-zero spectral coefficients, in which no higher-order spectral coefficient was truncated to zero. The results are presented for the initial condition eq. (5.9). Blue lines: controls from PMP, Black lines: controls from NN.



(a) k=15

(b) k=20

(c) k=25

(d) k=30

Figure 5.5: Comparison of control obtained from PMP and the NN trained on the modified datasets having k time-varying spectral coefficients, with all initial spectral coefficients $c_n(0) : n > 5$ in the dataset truncated to zero. The initial condition eq. (5.9) used in this validation step is, however, still expanded to k non-zero spectral coefficients. Blue lines: controls from PMP, Black lines: controls from NN.

Summary: In this chapter, we examined the optimal stabilization of an ODE system arising from the damped wave equation. As described in section 5.1, the higher system modes of this system do not display an exponential decay, and is therefore challenging to control. A Galerkin method based on the system eigenfunctions was used to optimize and solve the BVP. We observed that a non-zero initial value for these system modes has the potential to corrupt the NN-training. A good approximation for the optimal control could only be obtained for the lower system modes.

Chapter 6

Conclusions and Future Work

In this thesis, we have examined a data-driven approach for approximating the HJB equation arising from general high-dimensional optimal control problems. Feedforward neural networks were used to fit the value function and calculate the optimal control in a feedback form for generally stabilizable systems. The training datasets were generated based on initial conditions sampled from a restricted set of admissible initial conditions. Optimal trajectories were calculated using Pontryagin's Maximum Principle and various intermediate quantities such as the costates and optimal Hamiltonian tracked for training purposes. These intermediate quantities were subsequently applied to a physics-based loss function with a supervised optimization approach to greatly improve the neural network performance. These trained neural networks were utilized to evaluate the optimal control for the system in a closed-loop fashion. We also discussed in detail a toolkit for approximating the infinite-horizon optimal controller based on a power-series expansion of the system dynamics about the equilibrium point.

We numerically examined this method using three problems. The first involved the optimal control of the orientation of a six-degree of freedom rigid body. In the context of this problem, we also examined the feasibility of pre-training the neural network and selectively choosing training datapoints based on calculations from an approximated optimal controller. We confirmed that the approximate controller is most accurate around the stable equilibrium point, due to which pre-training was most effective when the controller was designed for a limited range of the initial states around the equilibrium point. Selective choice of initial conditions, however, did not lead to satisfactory results. The NN performance was seen to be comparable to the basic neural network only for the larger datasets, and with no significant performance improvements.

The second problem involved an optimal control of a high-dimensional non-linear system arising from the discretization of a Burgers'-like PDE. The problem was made computationally intensive by using spectral methods in the spatial discretization step: though spectral methods lead to improved simulation accuracy, they comprise operations involving dense matrix operators. The approximate optimal controller developed from the power-series expansion was used as an initial guess for the SciPy boundary value solver, and improvements in the convergence time were observed. We also examined the effect of an additional physics-based loss, based on the optimal Hamiltonian, on the performance of the neural network. The controller thus designed was found to have slightly improved convergence metrics and a more stable controller response.

The third problem concerned the optimal control of the damped wave equation, for which the higher system modes do not decay exponentially compared to the lower modes. For such systems, we found that the performance of the neural network degrades if a higher number of non-zero initial system modes are considered in the training dataset. Only the controls for the lower system modes could be properly approximated by the neural network. This potentially suggests that the supervised neural network-based approximation approach as examined in this thesis is well suited mostly for systems whose dynamics can be well represented by a smaller number of slowly decaying transient structures, such as the heat or Burgers' equation.

In the context of these neural network-based approaches to the HJB equation, current literature has primarily focused on fixed-time free-endpoint problems. An extension for different target sets, such as free-time problems with final-state constraints would be an important development. Such problems present computational difficulties at the data-generation step as unlike fixed-time problems, the time-instant at which the boundary conditions need to be imposed by the BVP solver is not known a-priori. Iterative methods such as the shooting method must be employed in conjunction with the BVP solver to solve them. Additionally, such problems can potentially give rise to non-unique solutions to Pontryagin's equations and non-smooth value functions, which present further challenges in the neural network optimization stage [37]. A further problem was discovered while applying this method to the wave equation, in which the higher system modes do not decay exponentially compared to the lower modes. Such components have the potential to corrupt or degrade the neural network performance. For the wave equation, we were able to take advantage of a Galerkin method based on the system eigenfunctions to obtain a set of ODEs that effectively filtered out the higher modes; this is not possible for general non-linear systems. A different method, potentially employing different neural network architectures, may be required to approximate the HJB equation in such cases, and is a subject of future research.

References

[1] Cesar O Aguilar and Arthur J Krener. Numerical solutions to the Bellman equation of optimal control. *Journal of optimization theory and applications*, 160:527–552, 2014.

[2] EG Al'Brekht. On the optimal stabilization of nonlinear systems. *Journal of Applied Mathematics and Mechanics*, 25(5):1254–1266, 1961.

[3] Alfonso Baños, Françoise Lamnabhi-Lagarrigue, and Francisco J Montoya. *Advances in the control of nonlinear systems*, volume 264. Springer Science & Business Media, 2001.

[4] Richard Bellman. *Dynamic Programming*. Princeton university press, Princeton, N.J., 1957.

[5] Peter Benner and Jens Saak. Numerical solution of large and sparse continuous time algebraic matrix Riccati and Lyapunov equations: a state of the art survey. *GAMM-Mitteilungen*, 36(1):32–52, 2013.

[6] Olivier Bokanowski, Jochen Garcke, Michael Griebel, and Irene Klompmaker. An adaptive sparse grid semi-Lagrangian scheme for first order Hamilton-Jacobi-Bellman equations. *Journal of Scientific Computing*, 55:575–605, 2013.

[7] Jason Brownlee. *Python Multiprocessing Jump-Start*. 2022.

[8] Arthur E Bryson. Optimal control-1950 to 1985. *IEEE Control Systems Magazine*, 16(3):26–33, 1996.

[9] Arthur E Bryson and Yu-Chi Ho. *Applied optimal control: optimization, estimation, and control*. Washington: Hemisphere Pub. Corp, 1975.

[10] Simone Cacace, Emiliano Cristiani, Maurizio Falcone, and Athena Picarelli. A patchy dynamic programming scheme for a class of Hamilton-Jacobi-Bellman equations. *SIAM Journal on Scientific Computing*, 34(5):A2625–A2649, 2012.

[11] Tao Cheng, Frank L. Lewis, and Murad Abu-Khalaf. Fixed-final-time-constrained optimal control of nonlinear systems using neural network HJB approach. *IEEE Transactions on Neural Networks*, 18(6):1725–1737, 2007.

[12] Yat Tin Chow, Jérôme Darbon, Stanley Osher, and Wotao Yin. Algorithm for overcoming the curse of dimensionality for state-dependent Hamilton-Jacobi equations. *Journal of Computational Physics*, 387:376–409, 2019.

[13] The Scipy Community. Scipy.integrate.solve_bvp. Accessed: 2023-09-30.

[14] The Scipy Community. Scipy.integrate.quad. Accessed: 2023-10-10.

[15] Salvatore Cuomo, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi, and Francesco Piccialli. Scientific machine learning through physics-informed neural networks: Where we are and what's next. *Journal of Scientific Computing*, 92(3):88, 2022.

[16] Jérôme Darbon and Stanley Osher. Algorithms for overcoming the curse of dimensionality for certain Hamilton-Jacobi equations arising in control theory and elsewhere. *Research in the Mathematical Sciences*, 3(1):19, 2016.

[17] Lawrence C. Evans. *Partial differential equations*. American Mathematical Society, Providence, R.I., 2010.

[18] Maurizio Falcone and Roberto Ferretti. *Semi-Lagrangian approximation schemes for linear and Hamilton-Jacobi equations*. SIAM, 2013.

[19] Zhiwei Fang. A high-efficient hybrid physics-informed neural networks based on convolutional neural network. *IEEE Transactions on Neural Networks and Learning Systems*, 33(10):5514–5526, 2021.

[20] The PyTorch Foundation. A Gentle Introduction to Autograd - PyTorch Tutorials. Accessed: 2023-09-30.

[21] The PyTorch Foundation. torch.autograd.grad-PyTorch 2.0 documentation. Accessed: 2023-09-30.

[22] MathWorks Inc. Implicit solver for continuous-time algebraic Riccati equations. Accessed 2023-10-01.

[23] Jamshed Iqbal, Mukhtar Ullah, Said Ghani Khan, Baizid Khelifa, and Saša Ćuković. Nonlinear control systems-a brief overview of historical and recent advances. *Nonlinear Engineering*, 6(4):301–312, 2017.

[24] Frank Jiang, Glen Chou, Mo Chen, and Claire J. Tomlin. Using neural networks to compute approximate and guaranteed feasible Hamilton-Jacobi-Bellman PDE solutions. https://doi.org/10.48550/arXiv.1611.03158.

[25] Wei Kang, PK De, and A Isidori. Flight control in a windshear via nonlinear H_∞ methods. In *[1992] Proceedings of the 31st IEEE Conference on Decision and Control*, pages 1135–1142. IEEE, 1992.

[26] Wei Kang and Lucas Wilcox. A causality free computational method for HJB equations with application to rigid body satellites. In *AIAA Guidance, Navigation, and Control Conference*, page 2009, 2015.

[27] Georgios Kissas, Yibo Yang, Eileen Hwuang, Walter R Witschey, John A Detre, and Paris Perdikaris. Machine learning in cardiovascular flows modeling: Predicting arterial blood pressure from non-invasive 4D flow MRI data using physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 358:112623, 2020.

[28] Alexander Kovacs, Lukas Exl, Alexander Kornell, Johann Fischbacher, Markus Hovorka, Markus Gusenbauer, Leoni Breth, Harald Oezelt, Masao Yano, Noritsugu Sakuma, et al. Conditional physics informed neural networks. *Communications in Nonlinear Science and Numerical Simulation*, 104:106041, 2022.

[29] Arthur J. Krener. Nonlinear Systems Toolbox v.1.0, 1997. MATLAB based toolbox available by request from ajkrener@ucdavis.edu.

[30] Isaac E Lagaris, Aristidis Likas, and Dimitrios I Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, 9(5):987–1000, 1998.

[31] Daniel Liberzon. *Calculus of variations and optimal control theory*. Princeton university press, Princeton, N.J., 2011.

[32] Jun Liu. *AMATH 456/656: Calculus of Variations Lecture Notes*. 2021.

[33] Jun Liu. *AMATH 455/655: Control Theory Lecture Notes*. 2022.

[34] Dahlard L Lukes. Optimal regulation of nonlinear dynamical systems. *SIAM Journal on Control*, 7(1):75–100, 1969.

[35] Abhilash Mathews, Manaure Francisquez, Jerry W Hughes, David R Hatch, Ben Zhu, and Barrett N Rogers. Uncovering turbulent plasma dynamics via deep learning from partial observations. *Physical Review E*, 104(2):025205, 2021.

[36] Kirsten A Morris. *Controller design for distributed parameter systems*. Springer, 2020.

[37] Tenavi Nakamura-Zimmerer, Qi Gong, and Wei Kang. Adaptive deep learning for high-dimensional Hamilton-Jacobi-Bellman Equations. *SIAM Journal of Scientific Computing*, 43(2):A1221–A1247, 2021.

[38] J Nathan Kutz, Joshua L Proctor, and Steven L Brunton. Applied Koopman theory for partial differential equations and data-driven modeling of spatio-temporal systems. *Complexity*, 2018:1–16, 2018.

[39] Carmeliza Navasca and Arthur J Krener. Patchy solutions of Hamilton-Jacobi-Bellman partial differential equations. In *Modeling, Estimation and Control: Festschrift in Honor of Giorgio Picci on the Occasion of his Sixty-Fifth Birthday*, pages 251–270. Springer, 2007.

[40] Lucien W Neustadt, Lev Semenovic Pontryagin, and KN Trirogoff. *The mathematical theory of optimal processes*. Interscience, 1962.

[41] Stanley Osher and Chi-Wang Shu. High-order essentially nonoscillatory schemes for Hamilton-Jacobi equations. *SIAM Journal on numerical analysis*, 28(4):907–922, 1991.

[42] Jacob Page and Rich R Kerswell. Koopman analysis of Burgers equation. *Physical Review Fluids*, 3(7):071901, 2018.

[43] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. 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, 2019.

[44] Maziar Raissi, Alireza Yazdani, and George E Karniadakis. Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations. *Science*, 367(6481):1026–1030, 2020.

[45] Justin Sirignano and Konstantinos Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. *Journal of computational physics*, 375:1339–1364, 2018.

[46] Yuval Tassa and Tom Erez. Least squares solutions of the HJB equation with neural network value-function approximators. *IEEE Transactions on Neural Networks*, 18(4):1031–1041, 2007.

[47] Juan Diego Toscano. Learning-PIML-in-Python. Accessed: 2023-09-30.

[48] Lloyd N Trefethen. *Spectral methods in MATLAB*. SIAM, 2000.

[49] Ivan Yegorov and Peter M Dower. Perspectives on characteristics based curse-of-dimensionality-free numerical approaches for solving Hamilton-Jacobi equations. *Applied Mathematics & Optimization*, 83:1–49, 2021.