Université de Montréal

Learning Neural Ordinary Differential Equations for Optimal Control

par

Nikolaus Harry Reginald Howe

Département d'Informatique et de Recherche Opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

Orientation Intelligence Artificielle

31 août 2021

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Learning Neural Ordinary Differential Equations for Optimal Control

présenté par

Nikolaus Harry Reginald Howe

a été évalué par un jury composé des personnes suivantes :

Gauthier Gidel
(président-rapporteur)

Pierre-Luc Bacon
(directeur de recherche)

Guy Wolf
(membre du jury)

Résumé

Ce mémoire rassemble des éléments d'optimisation, d'apprentissage profond et de contrôle optimal afin de répondre aux problématiques d'apprentissage et de planification dans le contexte des systèmes dynamiques en temps continu. Deux approches générales sont explorées. D'abord, une approche basée sur la méthode du maximum de vraisemblance est présentée. Ici, les trajectoires "d'entrainement" sont échantillonnées depuis la dynamique réelle, et à partir de celles-ci un modèle de prédiction des états observés est appris. Une fois que l'apprentissage est terminé, le modèle est utilisé pour la planification, en utilisant la dynamique de l'environnement et une fonction de coût pour construire un programme non linéaire, qui est par la suite résolu pour trouver une séquence de contrôle optimal. Ensuite, une approche de bout en bout est proposée, dans laquelle la tâche d'apprentissage de modèle dynamique et celle de planification se déroulent simultanément. Ceci est illustré dans le cadre d'un problème d'apprentissage par imitation, où le modèle est mis à jour en rétropropageant le signal de perte à travers l'algorithme de planification. Grâce au fait que l'entrainement est effectué de bout en bout, cette technique pourrait constituer un sous-module de réseau de neurones de plus grande taille, et pourrait être utilisée pour fournir un biais inductif en faveur des comportements optimaux dans le contexte de systèmes dynamiques en temps continu. Ces méthodes sont toutes les deux conçues pour fonctionner avec des modèles d'équations différentielles ordinaires paramétriques et neuronaux. Également, inspiré par des applications réelles pertinentes, un large recueil de systèmes dynamiques et d'optimiseurs de trajectoire, nommé Myriad, est implémenté; les algorithmes sont testés et comparés sur une variété de domaines de la suite Myriad.

Keywords: Apprentissage Profond, Apprentissage Automatique, Contrôle Prédictif par Modèle, ODE Neuronale, Optimisation Non Linéaire, Contrôle Optimal, Apprentissage par Renforcement

Abstract

This thesis brings together elements of optimization, deep learning and optimal control to study the challenge of learning and planning in continuous-time dynamical systems. Two general approaches are explored. First, a maximum likelihood approach is presented, in which training trajectories are sampled from the true dynamics, and a model is learned to accurately predict the state observations. After training is completed, the learned model is then used for planning, by using the dynamics and cost function to construct a nonlinear program, which can be solved to find a sequence of optimal controls. Second, a fully end-to-end approach is proposed, in which the tasks of model learning and planning are performed simultaneously. This is demonstrated in an imitation learning setting, in which the model is updated by backpropagating the loss signal through the planning algorithm itself. Importantly, because it can be trained in an end-to-end fashion, this technique can be included as a sub-module of a larger neural network, and used to provide an inductive bias towards behaving optimally in a continuous-time dynamical system. Both the maximum likelihood and end-to-end methods are designed to work with parametric and neural ordinary differential equation models. Inspired by relevant real-world applications, a large repository of dynamical systems and trajectory optimizers, named Myriad, is also implemented. The algorithms are tested and compared on a variety of domains within the Myriad suite.

Keywords: Deep Learning, Machine Learning, Model Predictive Control, Neural ODE, Nonlinear Programming, Optimal Control, Reinforcement Learning

Contents

List of tables

List of figures

Liste des sigles et des abréviations

AD	Différenciation Automatique
AGDA	Descente-Montée de Gradient en Alternance
BVP	Problème aux Limites
DDP	Programmation Différentielle Dynamique
DL	Apprentissage Profond
(D)RL	Apprentissage par Renforcement (Profond)
EXGD	Extragradient
IVP	Problème à Valeur Initiale
(i)LQR	Régulateur Linéaire Quadratique (Itératif)
ML	Apprentissage Automatique
MBRL	Apprentissage par Renforcement Basé sur un Modèle
MDP	Processus de Décision Markov
MFRL	Apprentissage par Renforcement Sans Modèle
MLE	Estimation du Maximum de Vraisemblance
MLP	Perceptron Multicouche
MPC	Contrôle Prédictif par Modèle
NLP	Optimisation Non Linéaire
OC(P)	(Problème de) Contrôle Optimal
ODE	Équation Différentielle Ordinaire
PMP	Principe du Maximum de Pontryagin
QP	Optimisation Quadratique
RK(4)	Méthode Runge-Kutta (de $4^{\text{ème}}$ Ordre)
RNN	Réseau de Neurones Récurrents
(S)GD	Descente de Gradient (Stochastique)
SQP	Optimisation Quadratique Successive
SysID	Identification de Système

List of acronyms and abbreviations

AD	Automatic Differentiation
AGDA	Alternating Gradient Descent-Ascent
BVP	Boundary Value Problem
DDP	Differential Dynamic Programming
DL	Deep Learning
(D)RL	(Deep) Reinforcement Learning
EXGD	Extragradient
IVP	Initial Value Problem
(i)LQR	(Iterative) Linear Quadratic Regulator
ML	Machine Learning
MBRL	Model-Based Reinforcement Learning
MDP	Markov Decision Process
MFRL	Model-Free Reinforcement Learning
MLE	Maximum Likelihood Estimation
MLP	Multilayer Perceptron
MPC	Model Predictive Control
NLP	Nonlinear Program
OC(P)	Optimal Control (Problem)
ODE	Ordinary Differential Equation
PMP	Pontryagin's Maximum Principle
QP	Quadratic Program
RK(4)	Runge-Kutta (4^{th} Order)
RNN	Recurrent Neural Network
(S)GD	(Stochastic) Gradient Descent
SQP	Sequential Quadratic Programming
SysID	System Identification

Acknowledgements

Thank you to Pierre-Luc Bacon for your kindness, support, and guidance over the past one and a half years; I'm so lucky to have you as an advisor. Thank you to the Mila administration, professors, staff and to my fellow students for making Mila a happy, welcoming, and collaborative space for learning and research. Thank you to my fellow LabReps for all your work for the Mila community, and for being such a pleasure to work with. Thank you to Arinka Jancarik for your support in administrative tasks and scholarship applications. Thank you to Eilif Muller for serving as my mentor over the past year. Thank you to Karina Wolf for your efforts to organize activities for the Mila community. Thank you to the members of the PLURL research group for your friendliness. Thank you to Manuel Del Verme for your leadership in PLURL and for your programming support. Thank you to Simon Dufort-Labbé for your work bringing many new environments to Myriad, and to Nitarshan Rajkumar for helping construct the foundations of the repository. Thank you to Michel Ma for our conversation about truncated backprop, to Ryan D'Orazio for our conversations about extragradient, and to Evgenii Nikishin for your thoughts on warm-starting and fixed points. Thank you to Pierre-Luc Bacon, David Yu-Tung Hui, Mahan Fathi, and Manuel Del Verme for your corrections and suggestions on this document. Thank you to David Kanaa and Julien Roy for helping me translate the abstract. Thank you to my parents and siblings for your love and support.

Chapter 1

Introduction

The construction and application of models to predict system dynamics is a practice central to many fields, including physics, engineering, and economics. In the context of deep learning (DL), recurrent neural networks (RNNs) are a common approach to the dynamics modelling problem, seeing application in physical and engineered systems [**Funahashi and Nakamura, 1993, Bailer-Jones et al., 1998, Connor et al., 1994**], financial markets [**Kamijo and Tanigawa, 1990**], and real-time strategy games [**Vinyals et al., 2019**].

Recently, there has been increasing interest in using neural network models to learn the dynamics of systems in *continuous* time, notably, with the neural ordinary differential equations (Neural ODEs) [**Chen et al., 2018**]. Where RNNs can only work with a discretization of the dynamics at hand, Neural ODEs are inherently continuous-time, making them particularly well-suited for modelling real-world phenomena. Neural ODEs also carry several theoretical benefits over RNN architectures, such as adaptability to different time scales and precision levels, better memory efficiency, and the ability to learn from irregularly spaced data. Building on top of the Neural ODE formulation, several further improvements have also been proposed, including parallelization of training [**Massaroli et al., 2021, Roesch et al., 2021, Lorin, 2020**], compacting of network weights with orthogonal polynomials [**Quaglino et al., 2019**], and adaptation of Neural ODEs to setting with constraints [**Tuor et al., 2020**] and stiff dynamics [**Kim et al., 2021**].

A fundamental aspect of the physical world that is notably absent from the Neural ODE formulation is the ability of an agent to take actions which influence system dynamics. In the context of machine learning (ML), this setting has traditionally been approached from a completely different angle: using the reinforcement learning (RL) paradigm, which models the world as a Markov decision process (MDP). RL algorithms are typically divided into two categories, model-free (MFRL) and model-based (MBRL). MFRL algorithms learn a policy directly through interaction with the environment [**Mnih et al., 2015a,**

Schulman et al., 2017]. MBRL algorithms, on the other hand, construct a dynamics model by interacting with the environment, and use this model to inform future behaviour, updating the model as new information is gathered [**Sutton, 1990, Gu et al., 2016**].

In most modern RL research, both the agent's value function, and, for MBRL, dynamics model, are represented by neural networks. In part due to the representational power of these neural networks, the same RL architecture can be used to learn to perform at or better than human level across a suite of different tasks, such as over the entire Arcade Learning Environment [**Mnih et al., 2015b**]. As the generality of RL models grow, so too does the amount of work required to effectively engineer an agent's reward function to promote desired behaviour [**Dewey, 2014**]. In particular, how to effectively engineer RL agents to behave safely – a prerequisite for application in most real-world tasks – remains a largely unsolved question [**Dulac-Arnold et al., 2020**].

Outside the realm of machine learning, the discipline of control theory also deals with taking actions to influence the evolution of dynamical systems over time, albeit from a very different perspective. Unlike in RL, optimal control (OC) formulations typically assume that the system dynamics are known ahead of time, and planning algorithms are often also given access to the time derivative of the dynamics and cost functions. Within this context, several approaches can be taken. One class of techniques uses local approximations of the dynamics function, resulting in a simplified problem [**Mayne, 1966**]. If the dynamics are highly nonlinear, then local approximations might not lead to a good solution. In such cases, one can solve a sequence of simplified problems, updating the state trajectory to better match the true dynamics at each iteration [**Li and Todorov, 2004**]. An alternative approach which avoids approximating dynamics and cost is to optimize over entire trajectories, solving for controls which satisfy the first-order optimality conditions that are necessary at the solution. These techniques are further divided into two classes: direct and indirect methods. Direct methods, such as single shooting, multiple shooting [**Betts, 2010**], and collocation [**Kelly, 2017**], first discretize the problem at a sequence of points, and then solve a resulting nonlinear program (NLP). Indirect methods instead try to find a solution to Pontryagin's Maximum Principle (PMP) by constructing a boundary value problem (BVP) which can be solved numerically [**Lenhart and Workman, 2007**].

One reason for the great success of optimal control techniques in industrial and commercial settings is the ease with which safety requirements – expressed as constraints on the state trajectory – can be included into the problem formulation [**Betts, 2010**]. Furthermore, some OC formulations provide fast convergence guarantees, enabling use in real-time applications such as controlled rocket landing [**Açıkmeşe et al., 2013**]. Of course, the effectiveness of OC models relies on having an accurate dynamics model. In settings with well-understood dynamics, but unknown parameters of those dynamics, system identification (SysID) can be used to estimate model parameters [**Keesman, 2011**]. Even when the dynamics are

not fully understood, in some cases, model predictive control (MPC), which repeatedly replans for controls at runtime with a receding time horizon, can be used to counteract the accumulation of modelling error [**Camacho and Alba, 2013**].

Several attempts have been made to include OC optimization techniques within a larger neural network architecture, by creating a differentiable control-oriented network subcomponent [**Gevers, 2005**]. In the same way that convolutional neural networks (CNNs) [**Fukushima, 1988, LeCun et al., 1989**] revolutionized computer vision and transformers [**Vaswani et al., 2017**] led to greatly improved performance on natural language tasks, the hope is to create a similarly useful inductive bias [**Mitchell, 1980**] for a network to learn to to behave optimally in a dynamical system. In order to train such a network in an end-to-end fashion, the embedded solver must be differentiable. This can be achieved by performing automatic differentiation (AD) [**Griewank, 2003**] through an unrolled optimal control algorithm [**Okada et al., 2017, Pereira et al., 2018**]. Alternatively, unrolling can be avoided by considering the gradient at the solver's fixed point, which satisfies the first-order necessary optimality conditions [**Amos et al., 2018, Jin et al., 2020**]. So far, these approaches have only only parametric dynamics models in discretized domains, thus simplifying aspects of both model learning and planning. Initial attempts have been made to use expressive and continuous-time Neural ODEs in a setting which includes controls. However, these works have not used the learned model to solve a trajectory optimization problem, instead focusing only on SysID [**Kidger et al., 2020**], or instead using the model in part of an actor-critic MBRL framework [**Alvarez et al., 2020**].

In this work, we develop a new neural network architecture for learning to perform optimally in continuous-time environments with complex dynamics. Our model, which can be trained in a fully end-to-end fashion, combines the modelling expressiveness of Neural ODEs with the inductive bias of a built-in nonlinear trajectory optimization solver. The model can be trained in both imitation learning and MDP settings, and because of the embedded optimal control problem, safety requirements can be easily included in the model's planning procedure.

While there are many possible uses of such models, it is particularly applicable acting in continuous-time physical systems with dynamics which are difficult for an expert to model explicitly. As such, a natural area of application is sustainable energy-related fields, such as optimizing airflow in a wind farm, or performing prediction and control of a smart electricity grid, as well as biological fields such as planning the optimal schedule for administration of medication, or managing population dynamics.

In order to test and compare the effectiveness of different techniques, a benchmark of environments and algorithms can be particularly useful. Indeed, within the context of RL, several benchmarks have been influential in pushing forward the state-of-the art, notably OpenAI Gym [**Brockman et al., 2016**] and the Arcade Learning Environment

[**Bellemare et al., 2013**]. Yet, these environments are inherently discrete time, and focus primarily on abstract game-like settings, so are not the ideal setting to test this research. Other repositories, such as Beluga [**Antony, 2018**] and Wilds [**Koh et al., 2021**], offer interesting real-world tasks, but Beluga is built for optimal control without model learning, and the Wilds domains all fall under the supervised learning paradigm. To this end, this thesis also presents a new benchmark, called *Myriad*, for testing RL and OC algorithms on continuous-time real-world tasks. Myriad offers a large number of dynamical systems spanning biology, medicine, and engineering, created for the dual tasks of model learning and optimal control.

Chapter 2

Background

In this chapter, we introduce the technical background and main algorithmic setup for the problems we are trying to solve. This work is built on a foundation including optimization, control theory, and machine learning.

2.1. Derivatives

The derivative of a function is a linear map best approximating the function at a point [**Spivak, 2018**]. For a scalar function $f : \mathbb{R} \to \mathbb{R}$ this corresponds to the slope of the function. If the function is over vectors, as in $f : \mathbb{R}^m \to \mathbb{R}$, then the output is a column vector of partial derivatives, known as the *gradient*. In a more general case, if the function is $f : \mathbb{R}^m \to \mathbb{R}^n$, then the corresponding notion of derivative is a matrix of partial derivatives, known as the *Jacobian*:

$$(D\ f)(\boldsymbol{x}) = \begin{bmatrix} (D_1\ f_1)(\boldsymbol{x}) & (D_2\ f_1)(\boldsymbol{x}) & \dots & (D_m\ f_1)(\boldsymbol{x}) \\ (D_1\ f_2)(\boldsymbol{x}) & (D_2\ f_2)(\boldsymbol{x}) & \dots & (D_m\ f_2)(\boldsymbol{x}) \\ \vdots & \vdots & \ddots & \vdots \\ (D_1\ f_n)(\boldsymbol{x}) & (D_2\ f_n)(\boldsymbol{x}) & \dots & (D_m\ f_n)(\boldsymbol{x}) \end{bmatrix}$$

Optimization and machine learning techniques often rely on local approximations of functions in order to make progress during training [**Goodfellow et al., 2016**]. This approximation is most commonly performed by taking a first-order Taylor expansion at a point on the function [**Nocedal and Wright, 2006**]. This requires calculation of the derivative.

Leveraging the chain rule of calculus and the fact that most of the functions we deal with can be reduced to composition of elementary functions of which the derivative is known, machine learning frameworks allow programmers to calculate derivatives without requiring the analytical form of the derivative of the function in consideration. This technique is known as *automatic differentiation* (AD).

AD can be performed forwards or backwards. Forward mode allows for derivatives to be calculated without storing any intermediate values along the computation graph, but requires a forward pass for each variable being updated or, alternatively, for a derivative matrix to be accumulated and multiplied at each step. As such, it is often of limited use in deep learning setups, which usually have hundreds to billions of learnable parameters. On the other hand, backward mode, commonly referred to as *backpropagation*, can calculate all parameter updates in a single pass, but requires intermediate values to be stored during the forward calculation.

Alternatives to AD exist, such as symbolic differentiation, which manipulates the symbolic representation of a function itself. It is occasionally, if less commonly, also used in machine learning applications.

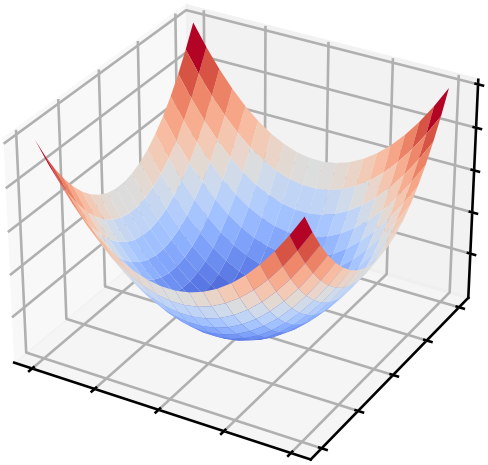
In cases where the techniques described above are not applicable – such as when we do not have access to the function of which we want to take the derivative – it is necessary to resort to other methods. One common approach is that of *finite differences* [**Betts, 2010**], which calculates approximate values of derivatives by measuring the function values some ε away from the point in question. While necessary in some cases, the method of finite differences is often expensive to implement in practice, especially for high-dimensional inputs or when calculating higher-order derivatives. Another approach, which can be used even with stochastic functions, is to use a derivative estimation scheme such as REINFORCE [**Williams, 1992**]. Though some effort is often required in order to get them to work in practice, REINFORCE-like techniques are central to many policy-based RL algorithms.

2.2. Curvature

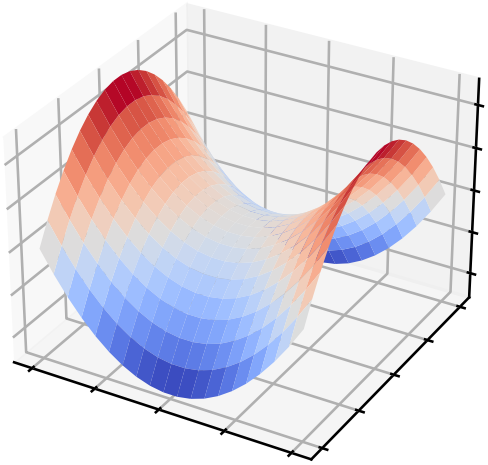
The curvature of a function can be thought of as how much a function deviates from being linear (or affine). One way of measuring the local curvature of a function at a point is to calculate its Hessian, which, for a scalar-output function $f : \mathbb{R}^n \to \mathbb{R}$, evaluated at $\boldsymbol{x}$, is a $n \times n$ matrix of second partial derivatives:

$$(H\ f)(\boldsymbol{x}) = \begin{bmatrix} (D_1^2\ f)(\boldsymbol{x}) & (D_1 D_2\ f)(\boldsymbol{x}) & \dots & (D_1 D_n\ f)(\boldsymbol{x}) \\ (D_2 D_1\ f)(\boldsymbol{x}) & (D_2^2\ f)(\boldsymbol{x}) & \dots & (D_2 D_n\ f)(\boldsymbol{x}) \\ \vdots & \vdots & \ddots & \vdots \\ (D_n D_1\ f)(\boldsymbol{x}) & (D_n D_2\ f)(\boldsymbol{x}) & \dots & (D_n^2\ f)(\boldsymbol{x}) \end{bmatrix}$$

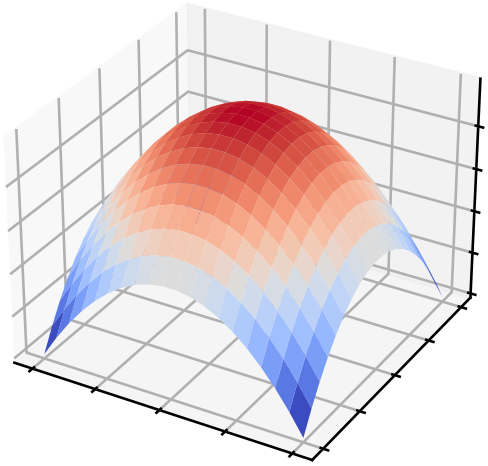
Oftentimes, the eigenvalues of the Hessian matrix give particular insight into the nature of our function. Note that since the Hessian matrix is symmetric, the eigenvalues are always real numbers. To gain intuition about what the eigenvalues tell us, consider the quadric surfaces depicted in Figure 2.1. These surfaces, which are all described by a second-order



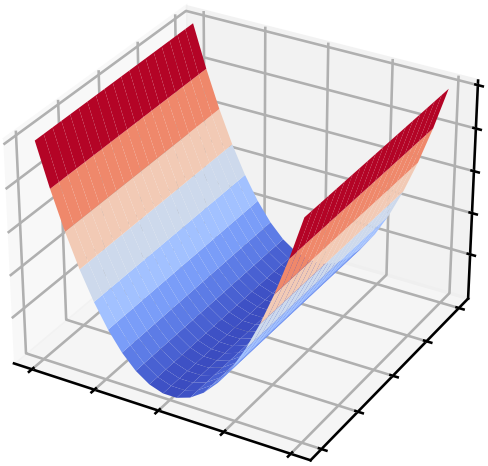
(a) Hyperbolic Ellipsoid (convex). Eigenvalues: (1, 1).



(b) Hyperbolic Paraboloid (not convex, not concave). Eigenvalues: (1, -1).



(c) Hyperbolic Ellipsoid (concave). Eigenvalues: (-1, -1).



(d) Parabolic Cylinder (degenerate, convex). Eigenvalues: (1, 0).

Fig. 2.1. Several quadric surfaces.

polynomial of the form $f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^\top \boldsymbol{Q}\boldsymbol{x} + \boldsymbol{c}^\top \boldsymbol{x}$, are special because their Hessian is the same regardless where it is evaluated, which makes them useful for visualization purposes.

2.3. Convex and Smooth Functions

In machine learning and optimization, we are often particularly happy to deal with certain kinds of well-behaved functions. Two desirable properties are *convexity* and *smoothness*.

A function $f : \mathbb{R}^n \to \mathbb{R}$ is said to be convex on a convex subset $S \subseteq \mathbb{R}^n$ if for every $0 \leq t \leq 1$ and every $\boldsymbol{x}_1, \boldsymbol{x}_2 \in S$ we have that

$$f(t\boldsymbol{x}_1 + (1-t)\boldsymbol{x}_2) \leq tf(\boldsymbol{x}_1) + (1-t)f(\boldsymbol{x}_2).$$

Note that positive semi-definiteness of a function's Hessian at every point in S is a sufficient condition for convexity on S [**Boyd and Vandenberghe, 2004**].

The notion of smoothness is unfortunately somewhat overloaded. One common definition is that a function is called "smooth" if it is twice continuously differentiable, and its Hessian is *Lipschitz continuous*. A function f is said to be Lipschitz continuous if there exists some constant $L \in \mathbb{R}$ such that for any points $\boldsymbol{x}_1, \boldsymbol{x}_2 \in S$, we have

$$\|f(\boldsymbol{x}_1) - f(\boldsymbol{x}_2)\|_2 \leq L \cdot \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|_2.$$

Intuitively, we can think of smooth, convex functions as those which look like the one in Figure 2.1a. In general it is often much easier to perform optimization on convex functions than nonconvex ones. Indeed, whether or not a function is convex can make the difference between easily solving a problem and the problem being completely intractable [**Boyd and Vandenberghe, 2004**], and many techniques in optimization try to 'convexify' a function near a point of interest [**Bertsekas, 1999, Nocedal and Wright, 2006**].

2.4. Finding a Minimum

One of the central goals in optimization is to find a local minimum of a function. For example, if our function represents a cost, then it is desirable to to find the inputs $\boldsymbol{x}$ which will result in the lowest cost. Many practical problems can be translated into the general form of a minimization problem:

$$\min_{\boldsymbol{x}} f(\boldsymbol{x}).$$

2.4.1. Gradient Descent

If our function is continuously differentiable, then we can use gradient descent (GD) to make progress towards its minimum. The idea of gradient descent is to repeatedly take small steps in the direction of greatest descent (that is, in the negative gradient direction). The update step for gradient descent is

$$\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i - \eta \cdot (D\ \ell)(\boldsymbol{x}_i),$$

where $(D\ \ell)(\boldsymbol{x}_i)$ is the gradient of ℓ evaluated at $\boldsymbol{x}_i$, and η is a hyperparameter called the *learning rate*.

A common application of gradient descent in machine learning is to find parameters of a model which perform well for a certain task. In this case, it is common to perform gradient

descent on a loss function which considers the model parameters and the entire training dataset. In many cases, it is impractical to use the whole training dataset simultaneously, and minibatches are selected randomly. Because of this randomness, this approach is referred to as stochastic gradient descent (SGD).

As an aside, we note that gradient descent can be thought of as the application of Euler's method for numerical integration (which will be presented in Section 2.11) on the gradient flow described by $\dot{\boldsymbol{x}}(t) = -(D\,\ell)(\boldsymbol{x}(t))$.

2.4.2. Adam

Many modifications to GD and SGD have been proposed and used in application. These additions have focused on improving convergence speed, and helping the algorithm escape local minima of the loss surface. One particularly effective modification of SGD is the Adam algorithm [**Kingma and Ba, 2015**], which we reproduce in Algorithm 1. Note that $\odot$ is the Hadamard product.

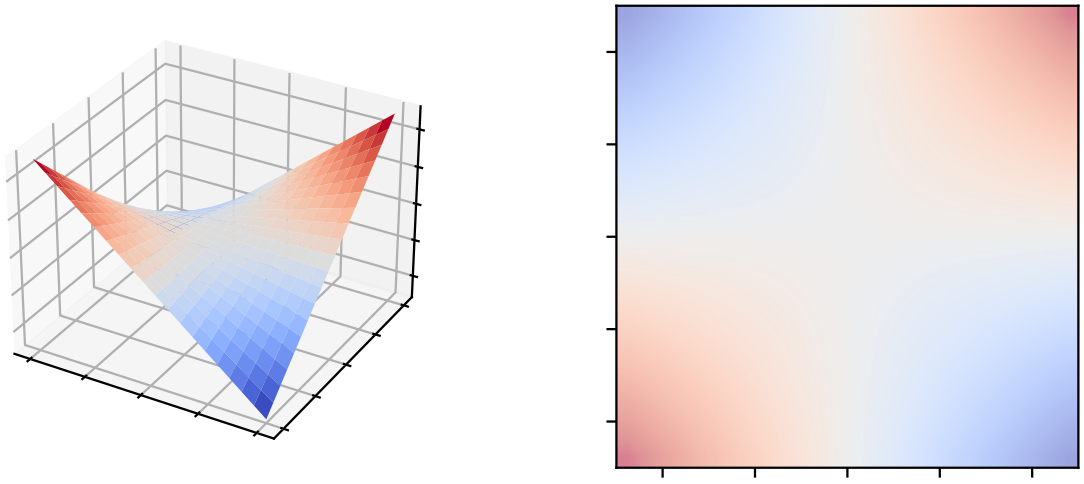
Algorithm 1 Adam

Require: α: step size
Require: $\beta_1, \beta_2 \in [0, 1)$: exponential decay rates for the moment estimates
Require: f: (stochastic) objective function
Require: x_0: initial parameter guess

$m_0 \leftarrow 0$ ▷ Initialize 1st moment vector
$v_0 \leftarrow 0$ ▷ Initialize 2nd moment vector
$t \leftarrow 0$ ▷ Initialize timestep
while x_t not converged **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow (D\,f)(x_{t-1})$ ▷ Get gradients with respect to objective at timestep t
  $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ ▷ Update biased first moment estimate
  $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (g_t \odot g_t)$ ▷ Update biased second raw moment estimate
  $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ ▷ Compute bias-corrected first moment estimate
  $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ ▷ Compute bias-corrected second raw moment estimate
  $x_t \leftarrow x_{t-1} - \eta \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$ ▷ Update parameters
end while
return x_t

2.5. Finding an Equilibrium

So far we have considered problems where we are trying to minimize the value of a single function. In a more general case, we might want to find a point of equilibrium between two competing objectives. In this case, we are no longer trying to find a minimum, but instead, a fixed point of a function of the form $f : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$.



(a) The surface of f.

(b) A contour map of f.

Fig. 2.2. Plots of the function $f(x, y) = xy$.

For visualization purposes, let us consider the case when $m = n = 1$, that is, a function which takes two scalars:

$$f(x, y) = xy.$$

We visualize this function in Figure 2.2. It is clear that f has a single stationary point at $(0, 0)$. If we start somewhere else on the surface of f, what kind of algorithm can we use to approach the fixed point at the origin?

We can describe this task in the language of games [**Koller et al., 1996**]. Say that we have two players, which can choose values x and y respectively. Player 1 wants to minimize the function $f(x, y)$, while player 2 wants to maximize it. Note that this formulation extends to functions with more complicated surfaces, and also to functions for which x and y are vectors.

2.5.1. Alternating Gradient Descent-Ascent

A simple algorithm to find the fixed point is to let players 1 and 2 alternate in choosing the step that leads to the biggest instantaneous improvement towards their objective. That is, a step in the negative gradient direction for player 1, followed by a step in the positive gradient direction for player 2. We call this approach *alternating gradient descent-ascent*[1] (AGDA):

[1] in the context of constrained optimization, which we will see in detail later, this method is known as the *First-Order Lagrangian Approach* [**Bertsekas, 1999**]

$$x_{i+1} \leftarrow x_i - \eta_x \cdot (D_1\ f)(x_i, y_i)$$
$$y_{i+1} \leftarrow y_i + \eta_y \cdot (D_2\ f)(x_{i+1}, y_i).$$

When applying AGDA, the algorithm is sometimes unable to make progress due to oscillatory behaviour. An example of this is shown in see Figure 2.3a. However, given certain assumptions about the function in consideration and a small enough step size, AGDA does converge *on average*, that is, the average of all iterates converges [**Kroer, 2020**].

2.5.2. Extragradient

One way to mitigate the oscillations observed in AGDA is the extragradient method (EXGD) [**Korpelevi, 1976**]. In EXGD, each player first takes a lookahead step, and then takes a true step based on the lookahead. In this way, player 1 chooses what would be the best move if player 2 played the optimal response to the original best move of player 1. Player 2 chooses a move similarly:

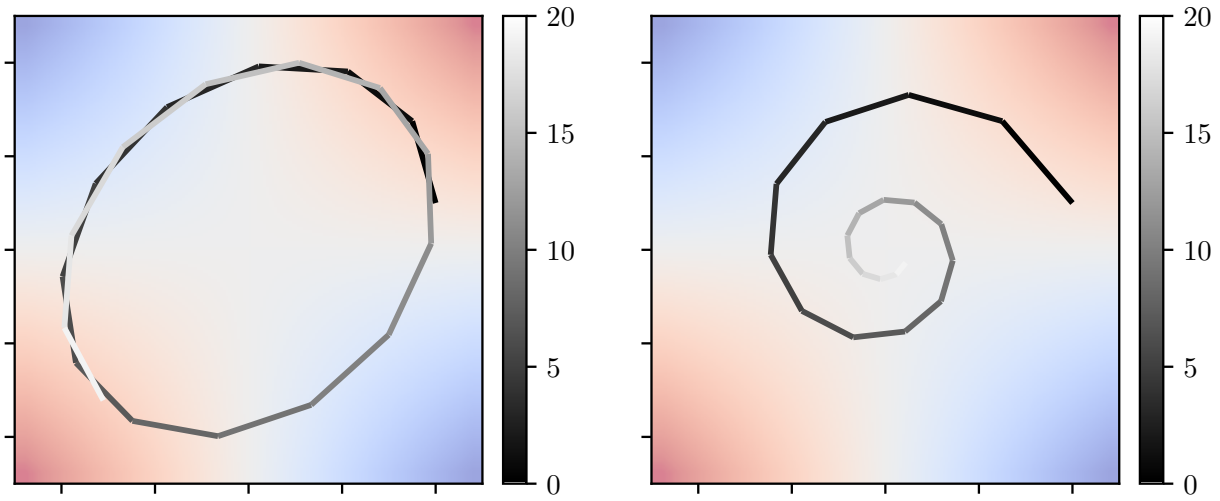
$$\bar{x}_i \leftarrow x_i - \eta_x \cdot (D_1\ f)(x_i, y_i)$$
$$\bar{y}_i \leftarrow y_i + \eta_y \cdot (D_2\ f)(x_i, y_i)$$
$$x_{i+1} \leftarrow x_i - \eta_x \cdot (D_1\ f)(\bar{x}_i, \bar{y}_i)$$
$$y_{i+1} \leftarrow y_i + \eta_y \cdot (D_2\ f)(\bar{x}_i, \bar{y}_i).$$

In practice, the lookahead before taking a step tends to enable the algorithm to progress towards the fixed point, despite there still being some oscillatory behaviour. This can be seen in Figure 2.3b.

Many modifications to EXGD exist, based on the number of lookahead steps and specific interleaving of those steps. Other techniques for overcoming the oscillations can also be used, such as the addition of a negative momentum term [**Gidel et al., 2019**].

2.5.3. Newton's Method

Those familiar with Newton's Method will remember it as a zero-finding algorithm, as opposed to a fixed-point-finding algorithm [**Ortega and Rheinboldt, 2000**]. Assuming that the function f of which we want to find a fixed point is smooth enough (twice continuously differentiable), then we can consider the problem of finding the zeros of its first derivative. The set of points where $\{\boldsymbol{x} \mid f'(\boldsymbol{x}) = 0\}$ is exactly the same set of points where $f(\boldsymbol{x})$ is a fixed point. With this equivalence in mind, let us turn our focus to finding a zero of the function $F = f'$.



(a) 20 steps of alternating gradient descent/ascent.

(b) 20 steps of extragradient.

Fig. 2.3. Comparison of alternating gradient descent-ascent (a) and extragradient (b). The shade of the grey line indicates how many steps have been taken, getting lighter over time. While AGDA displays oscillatory behaviour, it does converge to the fixed point on average. The iterates given by EXGD approach the fixed point.

Consider a function $F : \mathbb{R}^n \to \mathbb{R}^n$ of which we would like to find a zero, that is, an $\boldsymbol{x}$ such that $F(\boldsymbol{x}) = \mathbf{0}$. If we have a guess $\boldsymbol{x}_k$, it is enough to find $\boldsymbol{\Delta}_k$ such that $F(\boldsymbol{x}_k + \boldsymbol{\Delta}_k) = \mathbf{0}$. This is hard to do exactly, but relatively easy to do approximately, by taking a first-order expansion of F at $\boldsymbol{x}_k$ and setting equal to zero:

$$F(\boldsymbol{x}_k + \boldsymbol{\Delta}_k) \approx F(\boldsymbol{x}_k) + (D\ F)(\boldsymbol{x}_k) \cdot \boldsymbol{\Delta}_k = \mathbf{0},$$

where $D\ F$ is the Jacobian of F.

If we rearrange this equation, we can find a step which we can repeatedly apply until we find a zero:

$$\begin{aligned} (D\ F)(\boldsymbol{x}_k) \cdot \boldsymbol{\Delta}_k &= -F(\boldsymbol{x}_k) \qquad (2.5.1) \\ \implies \boldsymbol{\Delta}_k &= -\ [(D\ F)(\boldsymbol{x}_k)]^{-1}\, F(\boldsymbol{x}_k) \quad \text{when } (D\ F)(\boldsymbol{x}_k) \text{ invertible} \end{aligned}$$

In practice, we often prefer to stop at Eq. (2.5.1) and use an iterative solver to find $\boldsymbol{\Delta}_k$, instead of performing a matrix inversion. Once we have $\boldsymbol{\Delta}_k$, we can update our estimate of the zero by setting

$$\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \boldsymbol{\Delta}_k,$$

which we apply repeatedly until this procedure converges [**Nocedal and Wright, 2006**]. In practice, Newton's method comes with several challenges, including the fact that for nonconvex functions, it is difficult to give convergence (or timing) guarantees [**Byrd et al., 2010**].

2.6. Mathematical Programming

A mathematical program is a problem in which we would like to choose some values (known as *decision variables*) so as minimize an objective function, possibly subject to some constraints [**Bertsekas, 1999**]. If either the objective function or constraint function are nonlinear, then the problem is referred to as a nonlinear program. We can write this general kind of problem as

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}) \\ \text{such that} \quad & g(\boldsymbol{x}) \geq \boldsymbol{0} \\ \text{and} \quad & h(\boldsymbol{x}) = \boldsymbol{0}, \end{aligned} \tag{2.6.1}$$

where $\boldsymbol{x} \in \mathbb{R}^X$, $f : \mathbb{R}^X \to \mathbb{R}$, $g : \mathbb{R}^X \to \mathbb{R}^G$, $h : \mathbb{R}^X \to \mathbb{R}^H$, and we let $X, G, H \in \mathbb{N}^+$. We generally assume that the functions f, g and h are at least C^1 continuous. Mathematical programming is a vast field of study, with many techniques used depending on the specific setup of the mathematical program in question.

2.7. Linear Programming

An important special case is the linear program, in which the cost and constraint functions are linear. This field of study is called linear programming. The canonical form of a linear program is

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{such that} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ \text{and} \quad & \boldsymbol{x} \geq \boldsymbol{0}, \end{aligned}$$

where $\boldsymbol{x} \in \mathbb{R}^X$ and $\boldsymbol{A} \in \mathbb{R}^G \times \mathbb{R}^X$. The notation $\boldsymbol{a} \leq \boldsymbol{b}$ means that $a_i \leq b_i \; \forall i$. In general, linear programs are the simplest kind of mathematical program to solve. Two of the standard methods for solving linear programs are the *simplex method*, which examines the points on the boundary of the simplex defined by the feasible half-hyperplanes, and a family of methods known as *interior-point methods*, which instead make progress to the optimal point through the interior of the feasible polytope [**Bertsimas and Tsitsiklis, 1997**].

2.8. Quadratic Programming

Another special case is that of a quadratic program. It is similar to the linear program, except the objective function can be quadratic. We can write this as

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \frac{1}{2}\boldsymbol{x}^\top \boldsymbol{Q}\boldsymbol{x} + \boldsymbol{c}^\top \boldsymbol{x} \\ \text{such that} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}. \end{aligned}$$

Quadratic programs are generally harder than linear programs, but still retain a useful property: both the objective function and the constraints are convex. This allows one to prove guarantees of convergence and speed of various optimization algorithms [**Boyd and Vandenberghe, 2004**].

2.9. Nonlinear Programming

In the general case, f, g and h of the mathematical programming problem described in Equation (2.6.1) are all nonconvex functions. In such a setting, there are no guarantees of reaching a global optimum, and even proving convergence of an algorithm can be a challenge. It is in this setting that we are interested, however, and we must accept the lack of certainty that comes hand-in-hand with the expressive power of nonlinear programs.

Let us begin by defining the Lagrangian of a nonlinear program as

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\mu}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + \boldsymbol{\mu}^\top g(\boldsymbol{x}) + \boldsymbol{\lambda}^\top h(\boldsymbol{x}).$$

The vectors $\boldsymbol{\mu}$ and $\boldsymbol{\lambda}$ are known as the *dual variables* or equivalently *Lagrange multipliers*. In fact, let us simplify the problem by considering an NLP with only equality constraints. After exploring solutions to this simpler problem, we will discuss how the solution techniques can be extended to the general inequality-constrained case. For now, our Lagrangian is simply:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + \boldsymbol{\lambda}^\top h(\boldsymbol{x}).$$

We can think of a solution to this nonlinear program in terms of a min-max formulation, in which we are trying to find an $\boldsymbol{x}$ which minimizes the the Lagrangian, that is, the cost function and the constraint violation, while a the value of $\boldsymbol{\lambda}$ corresponds to the penalty incurred for violating the constraints:

$$\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda})$$

Through Equation 2.9, we see that a solution to the nonlinear programming problem will be a point where, given a $\boldsymbol{\lambda}$, no local change in $\boldsymbol{x}$ can lower the cost; in other words, a

solution will have $\boldsymbol{x}, \boldsymbol{\lambda}$ such that $(D_1\ \mathcal{L})(\boldsymbol{x}, \boldsymbol{\lambda}) = 0$. We will now see several techniques to find such a solution.

2.9.1. SQP

One simple approach to nonlinear programming is sequential quadratic programming (SQP) [**Nocedal and Wright, 2006**]. The general technique of SQP is to use Newton's method on $F = D_1\ \mathcal{L}$. To simplify notation, let $A = D\ h$ (evaluating A at a point gives a Jacobian matrix), and let $g = D\ f$ (evaluating g at a point gives a gradient vector). Then a single Newton step $\boldsymbol{\Delta}_k = [\boldsymbol{d}_k, \boldsymbol{\delta}_k]^\top$ can be calculated as the solution to the following system of equations (known as the primal-dual equations):

$$\begin{bmatrix} W(\boldsymbol{x}_k, \boldsymbol{\lambda}_k) & A(\boldsymbol{x}_k)^\top \\ A(\boldsymbol{x}_k) & 0 \end{bmatrix} \cdot \begin{bmatrix} \hat{d} \\ \hat{\delta} \end{bmatrix} = - \begin{bmatrix} g(\boldsymbol{x}_k) + A(\boldsymbol{x}_k)^\top \boldsymbol{\lambda}_k \\ h(\boldsymbol{x}_k) \end{bmatrix}, \tag{2.9.1}$$

with

$$W(\boldsymbol{x}_k, \boldsymbol{\lambda}_k) \triangleq (D_1^2\ \mathcal{L})(\boldsymbol{x}_k, \boldsymbol{\lambda}_k) = (D_1^2\ f)(\boldsymbol{x}_k) + \sum_{i=1}^{t} \left(\lambda_k^i \cdot (D_1^2\ h^i)(\boldsymbol{x}_k) \right),$$

where h^i is the ith constraint function, and we put hats on $\hat{\boldsymbol{d}}$ and $\hat{\boldsymbol{\delta}}$ to remind ourselves that they are to be found by solving the system of equations in Eq. (2.9.1).

In the case that $A(\boldsymbol{x}_k)$ has full row-rank (that is to say, the gradients of the constraints at this point are linearly independent), and $W(\boldsymbol{x}_k, \boldsymbol{\lambda}_x)$ is positive definite on the null space of $A(\boldsymbol{x}_k)$, there is an equivalence between the problem as stated, and the resolution of the following quadratic program:

$$\begin{aligned} \min_{\boldsymbol{d} \in \mathbb{R}^n} \quad & f(\boldsymbol{x}_k) + g(\boldsymbol{x}_k)^\top \boldsymbol{d} + \frac{1}{2} \boldsymbol{d}^\top W(\boldsymbol{x}_k, \boldsymbol{\lambda}_k) \boldsymbol{d} \\ \text{subject to} \quad & h(\boldsymbol{x}_k) + A(\boldsymbol{x}_k) \boldsymbol{d} = \boldsymbol{0}. \end{aligned} \tag{2.9.2}$$

SQP works by repeatedly solving the quadratic program in Eq. (2.9.2), hence the name *sequential quadratic programming*.

Beyond its theoretical requirements of Hessian positive definiteness (which can be seen as a convexity requirement) and full row-rank of the constraint matrix, SQP also suffers from practical limitations. In particular, it is often necessary to start with a good initial guess in order to observe convergence [**Nocedal and Wright, 2006**]. Furthermore, one is usually required to construct the entire W and A matrices at a sequence of points, which could be computationally and memory-intensive. Due to these challenges, SQP is often used as a base for more sophisticated algorithms which address some of its shortcomings [**Nocedal and Wright, 2006**].

2.9.2. An Inexact Newton Method

One such algorithm is described in [**Byrd et al., 2010**], in which the authors focus on minimizing a linearization of a penalty function:

$$\phi(\boldsymbol{x};\pi) \triangleq f(\boldsymbol{x}) + \pi\|h(\boldsymbol{x})\|,$$

where $\pi \in \mathbb{R}^+$. In particular, a candidate step is calculated by solving Equation (2.9.1). If the step leads to a sufficiently large reduction in a linear approximation of $\phi(\boldsymbol{x};\pi)$, then it is accepted. Otherwise, if the geometry around the current iterate is sufficiently convex, or the candidate step doesn't make the constraint violation significantly worse, then π is increased and the step is tried again. If neither of these conditions is satisfied, then the W matrix is modified to make it closer to positive definite, by adding a small nonzero value to each diagonal element.

One nice advantage of this method is that it is *matrix free*, that is, it does not require the explicit construction of any matrices, and instead only constructs the necessary structure at compute time. This means it could be practical in very high-dimensional settings, where calculating and constructing the whole Hessian or Jacobian matrix would be infeasible.

2.9.3. A Lagrangian Method based on Extragradient

Another alternative to SQP is to apply the Extragradient method discussed in Section 2.5.2 directly to the gradient of the Lagrangian. In this case, at a given iteration k, player 1 chooses $\boldsymbol{x}_{k+1}$, and in response, player 2 chooses $\boldsymbol{\lambda}_{k+1}$.

In practice, this approach, which is fundamentally a first-order method, requires many more iterations to reach convergence when compared with second-order methods like those derived from SQP. However, due to the low cost of computing gradients compared with Hessians, it is competitive in some problem domains, and might out-perform second-order methods in high-dimensional problems.

2.9.4. Other NLP Methods

In addition to those discussed, there are many other methods for solving nonlinear programs. Some of the main categories are:

Barrier and Interior Point methods: In these methods, *barrier functions* are added to the objective function f, which go to infinity as iterates approach the boundary of the feasible set from the inside. Negative logarithmic functions or inverse functions are two possible candidates for barrier methods. Over time, the steepness of the barriers can be changed to allow for convergence to points close to

the edge of the feasible set. In many cases these methods require the initial guess to be contained in the feasible set [**Bertsekas, 1999**].

Augmented Lagrangian and Multiplier methods: In these methods, the Lagrangian is augmented with a term proportional to the violation of the equality constraints

$$L_c(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) + \boldsymbol{\lambda}^\top h(\boldsymbol{x}) + \frac{c}{2}\|h(\boldsymbol{x})\|^2,$$

where $c \in \mathbb{R}$ encodes how much we penalize violating the constraints. Often we increase c to arbitrarily large values as the algorithm progresses, to insure convergence to a feasible point. This approach is effective because the penalty term effectively creates a local convexification of the loss surface [**Bertsekas, 1999**].

2.9.5. Inequality Constraints

We now explore several ways of including inequality constraints in our nonlinear programs. Which technique we choose will depend on the nature of the constraints. In particular, the inequality constraints can either be constant bounds of the form

$$\boldsymbol{x}_{\text{lower}} \leq \boldsymbol{x} \leq \boldsymbol{x}_{\text{upper}},$$

or they can be fully general functional inequality constraints, as expressed in Eq. (2.6.1). We often refer to constant inequality constraints as *bounds* or *bound constraints*.

There are two main approaches we can use to deal with bounds: projection and reparametrization.

Projection: The idea of a projection-based method is to simply take a step as if there were no bounds, and then to project the resulting point back into the *feasible set*, that is, back within the bounds. Because constant bounds give the feasible set a box-like structure, the projection operation is particularly convenient – it is simply a clipping operation in all dimensions [**Bertsekas, 1999**]:

$$\text{clip}(x, x_{\text{lower}}, x_{\text{upper}}) = \begin{cases} x_{\text{lower}} & \text{if } x < x_{\text{lower}} \\ x & \text{if } x_{\text{lower}} \leq x \leq x_{\text{upper}} \\ x_{\text{upper}} & \text{if } x > x_{\text{upper}} \end{cases}$$

Reparametrization: Reparametrization, as described in [**Niculae, 2020**], takes a different approach, by passing our points through a function which forces the result to always be in the feasible set. For example, if our feasible set is $x_{\text{lower}} \leq x \leq x_{\text{upper}}$, then one way to use reparametrization to guarantee feasibility of our iterate is by passing it through a sigmoid:

$$\sigma(x, x_{\text{lower}}, x_{\text{upper}}) = \frac{x_{\text{upper}} - x_{\text{lower}}}{1 + e^{-\alpha x}} - x_{\text{lower}},$$

where α is some temperature constant. We can perform this across every dimension, and might experiment with different temperature values, or even decrease the temperature over time to allow values to more easily approach the boundary.

In some sense, the projection and reparametrization approaches can be thought of as two sides of the same idea: projection takes a step and then restricts it to the feasible set, whereas reparametrization restricts the function to the feasible set, and takes steps in the new space. See [**Niculae, 2020**] for a detailed exploration of projection, reparametrization, and some alternative approaches for dealing with inequality constraints in optimization.

In the more general case, inequality constraints will not be simple bounds, but instead will be (nonlinear) functions of the iterate. In particular, this means that they will not stay fixed as we update the iterate, making them fundamentally different than bound constraints. The approach in this case is generally to keep track of an *active set* of inequality constraints, that is, to record at each step which boundaries of the feasible set the iterate is on [**Betts, 2010**]. Then, at each step, the active inequality constraints are treated as equality constraints, and the inactive ones are ignored. After a step is taken, the active set is recomputed [**Betts, 2010**].

2.10. Dynamical Systems

A dynamical system is a system with a state, represented as a point in some space, that evolves over time.

2.10.1. Discrete Time

In a discrete time setting, one can describe the evolution of a system's state using a difference equation, which says how the state will change as a function of the current timestep $i \in \mathbb{R}$, current state $\boldsymbol{x} \in \mathbb{R}^X$, and a function $f : \mathbb{R}^X \times \mathbb{N} \to \mathbb{R}^X$ describes how the state evolves over time:

$$\boldsymbol{x}_{i+1} = f(\boldsymbol{x}_i, i).$$

If we start at timestep i_s in state x_s and want to find the state at a final timestep i_f, we can calculate it by repeatedly applying the difference equation:

$$\boldsymbol{x}_f = \boldsymbol{x}_s + \sum_{k=i_s}^{i_f} f(\boldsymbol{x}_k, k). \tag{2.10.1}$$

Many ML algorithms for modelling system dynamics do so by learning a difference equation [**Moerland et al., 2020**].

2.10.2. Continuous Time

Difference equations are effective at describing systems that evolve over discrete timesteps, but are less well-suited to continuous-time settings. Indeed, with a difference equation, we can only find the state at a discrete number of timesteps: if asked to find the state at a time between timesteps, we would need to interpolate. Once a difference equation is constructed, the timestep is fixed: if asked to predict a state many timesteps in the future, we would need to sum over many timesteps, even if the underlying dynamics are very simple.

A much more natural approach to model dynamical systems in continuous time is that of ordinary differential equations (ODEs). Instead of our function stepping to the next state, an ODE describes the instantaneous *change* in state, that is $\dot{\boldsymbol{x}} = D\ \boldsymbol{x}$. Now the state is described by $\boldsymbol{x} : \mathbb{R} \to \mathbb{R}^X$, and the function is $f : \mathbb{R}^X \times \mathbb{R} \to \mathbb{R}^X$. A system of differential equations can be used to describe the instantaneous change of a system with vector state:

$$\dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t), t).$$

Suppose we start at time t_s in state $\boldsymbol{x}(t_s)$. If we want to find the value of a state at a future time t_f, we can calculate it as

$$\boldsymbol{x}(t_f) = \boldsymbol{x}(t_s) + \int_{t_s}^{t_f} f(\boldsymbol{x}(t), t)\ dt. \tag{2.10.2}$$

The ODE approach, for its advantages, also comes with a significant disadvantage: computers cannot usually take infinitesimally small steps. Thus, unlike a difference equation where a sum is enough, the act of finding $\boldsymbol{x}_f$ becomes nontrivial. We refer to the problem of finding $\boldsymbol{x}_f$ as in Eq. (2.10.2) as an *initial value problem* (IVP). Depending on the form of the ODE, in some cases, we can use the rules of calculus to derive a closed-form solution for $\boldsymbol{x}(t)$. However, this is not possible in general. As such, we usually turn to numerical integration techniques, which can approximate the integral of f to varying degrees of accuracy.

2.11. Numerical Integration

Before diving into different numerical integration methods, let us establish some notation. When discussing numerical integration, we usually decide to discretize the problem at a certain granularity. To do this, we must choose into how many steps we would like to divide the integration interval[2] $N \in \mathbb{N}^+$. From this we can calculate our step size $h \in \mathbb{R}, h = (t_f - t_s)/N$. For convenience of notation, we let $t_0 \equiv t_s$, $t_N \equiv t_f$, $\boldsymbol{x}_0 \equiv \boldsymbol{x}_s$, and $\boldsymbol{x}_N \equiv \boldsymbol{x}_f$. Then we have that $t_{i+1} = t_i + h$.

[2]More advanced integration techniques use variable step sizes, which we will discuss later.

Remaining agnostic of which numerical integration method is used, we define the notation for a numerical integral of the function f from time t_0 until time t_N, starting in state $\boldsymbol{x}_0$, and using step size h, as

$$ {}_hS_{t_0}^{t_N}(f, \boldsymbol{x}_0) \approx \int_{t_0}^{t_N} f(\boldsymbol{x}(t), t)\, dt. \tag{2.11.1} $$

We now consider different ways of calculating the left-hand side of Equation (2.11.1), with varying levels of simplicity, speed, and accuracy.

2.11.1. Euler's Method

Since the ODE describes the instantaneous change in state, if we could take infinitesimally small steps in time, we could then find the true state trajectory, just as we do when stepping through a difference equation. Although we cannot take infinitesimally small steps, we *can* take very small steps. In that case, we have that

$$ \boldsymbol{x}(t + h) \approx \boldsymbol{x}(t) + h \cdot f(\boldsymbol{x}(t), t). $$

In fact, this corresponds to performing a first-order approximation of the true state trajectory, around the current state and time. The simplest integration method is based on this idea, and is known as Euler's method. We can approximate the integral by starting in state $\boldsymbol{x}_0 = \boldsymbol{x}(t_s)$ and then repeatedly applying the following rule N times:

$$ \boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + h \cdot f(\boldsymbol{x}_i, t_i). $$

This procedure gives us a sequence of points $\boldsymbol{x}_{0:N}$. In particular, each point $\boldsymbol{x}_i$ approximates the value of $\boldsymbol{x}(t_s + ih)$. And so, $\boldsymbol{x}_N \approx \boldsymbol{x}(t_f)$. Figure 2.4 shows a single step of Euler's method.

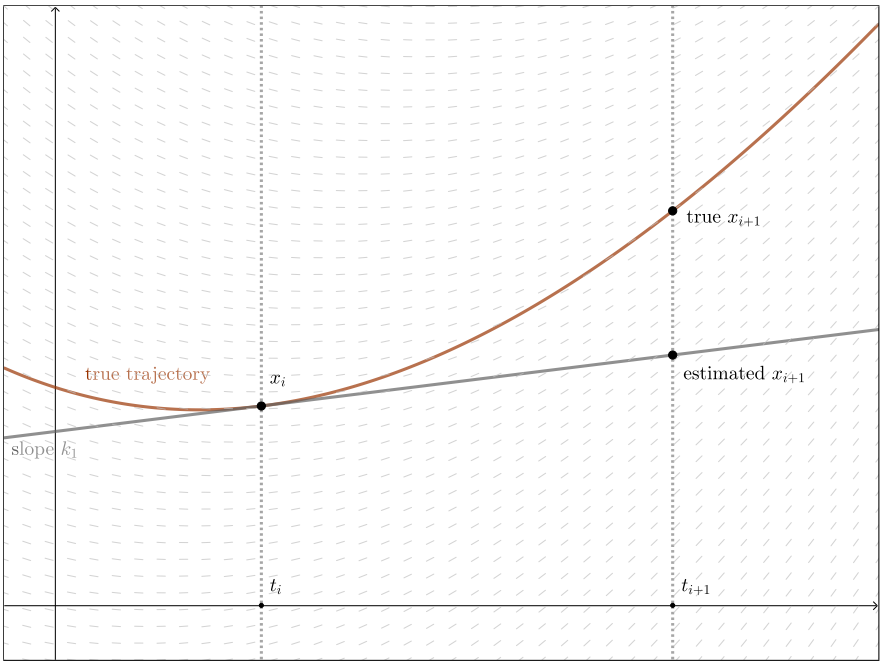


Fig. 2.4. A visualization of the Euler step.

Because it only considers a single gradient, Euler's method will not necessarily give good estimates unless used with many extremely small timesteps, which can be computationally expensive. Thus, when precision is needed, we prefer to use methods which make a better approximation by evaluating the gradient at more points.

2.11.2. Heun's Method

Heun's method is based the intuition that the average value of $f(\boldsymbol{x}(t), t)$ between t_i and t_{i+1} should be close to the average of its values at t_i and t_{i+1}:

$$\boldsymbol{x}(t_{i+1}) \approx \boldsymbol{x}(t_i) + h \cdot \frac{f(\boldsymbol{x}(t_i), t_i) + f(\boldsymbol{x}(t_{i+1}), t_{i+1})}{2}$$

Unfortunately, we can't use this formula as written, since in order to use it we must know $\boldsymbol{x}(t_{i+1})$, which is what we're trying to find in the first place! But, we can first use an Euler approximation for $\boldsymbol{x}(t_{i+1})$:

$$\boldsymbol{x}(t_{i+1}) \approx \boldsymbol{x}(t_i) + h \cdot f(\boldsymbol{x}(t_i), t_i),$$

and this gives us Heun's method:

$$\tilde{\boldsymbol{x}}_{i+1} = \boldsymbol{x}_i + h \cdot f(\boldsymbol{x}_i, t_i)$$
$$\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + h \cdot \frac{f(\boldsymbol{x}_i, t_i) + f(\tilde{\boldsymbol{x}}_{i+1}, t_{i+1})}{2}$$

In general, Heun's method will give significantly more accurate results than Euler's method. Figure 2.5 shows a single step of Heun's method.

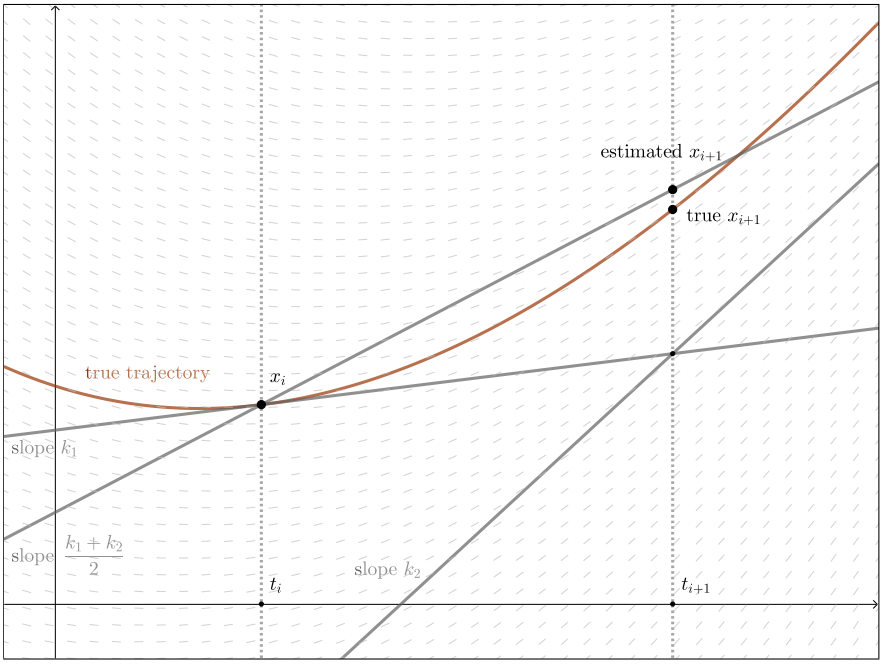


Fig. 2.5. A visualization of the Heun step.

2.11.3. Midpoint Method

The midpoint method is based on the intuition that the value of f at the time halfway through a timestep is a good approximation of its average value through the timestep:

$$\boldsymbol{x}(t_{i+1}) \approx \boldsymbol{x}(t_i) + h \cdot f\left(\boldsymbol{x}\left(t_i + \frac{h}{2}\right), t_i + \frac{h}{2}\right).$$

We again use an Euler approximation, this time to get an estimate of $\boldsymbol{x}(t_i + \frac{h}{2})$, which gives us the Midpoint step:

$$\tilde{\boldsymbol{x}}_{i+0.5} = \boldsymbol{x}_i + h \cdot f(\boldsymbol{x}_i, t_i)$$
$$\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + h \cdot f(\tilde{\boldsymbol{x}}_{i+0.5}, t_{i+0.5}).$$

Figure 2.6 shows an illustration of the Midpoint step.

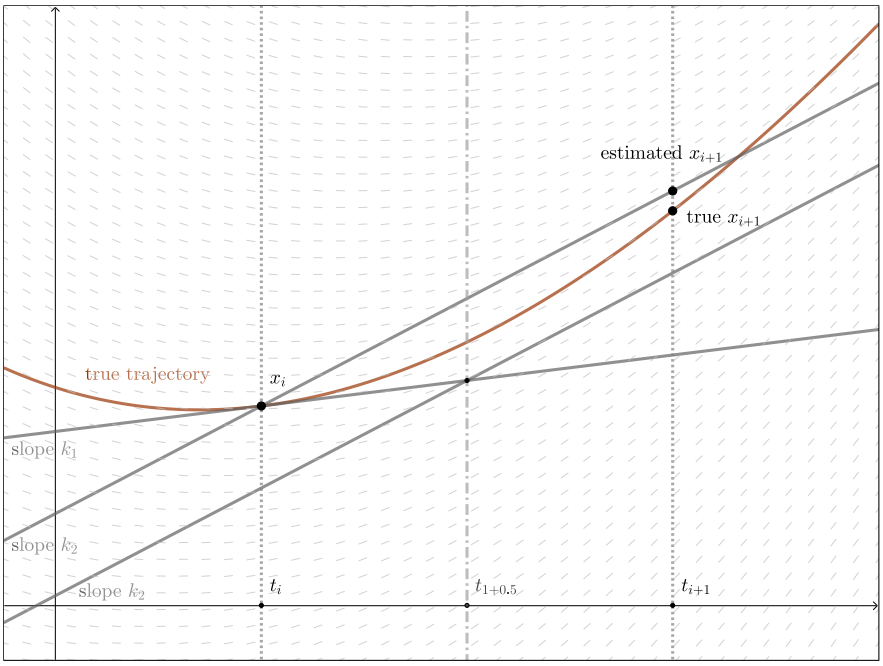


Fig. 2.6. A visualization of the Midpoint step.

2.11.4. Runge-Kutta 4th Order Method

A more elaborate and popular method is the Runge-Kutta 4th order method (RK4), which we write without derivation:

$$\begin{aligned}
\boldsymbol{k}_1 &= f\left(\boldsymbol{x}_i, t_i\right) \\
\boldsymbol{k}_2 &= f\left(\boldsymbol{x}_i + \frac{\boldsymbol{k}_1}{2}h, t_i + \frac{h}{2}\right) \\
\boldsymbol{k}_3 &= f\left(\boldsymbol{x}_i + \frac{\boldsymbol{k}_2}{2}h, t_i + \frac{h}{2}\right) \\
\boldsymbol{k}_4 &= f\left(\boldsymbol{x}_i + \boldsymbol{k}_3 h, t_{i+1}\right) \\
\boldsymbol{x}_{i+1} &\leftarrow \boldsymbol{x}_i + \frac{h}{6}\left(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4\right)
\end{aligned}$$

Figure 2.7 illustrates the RK4 step.

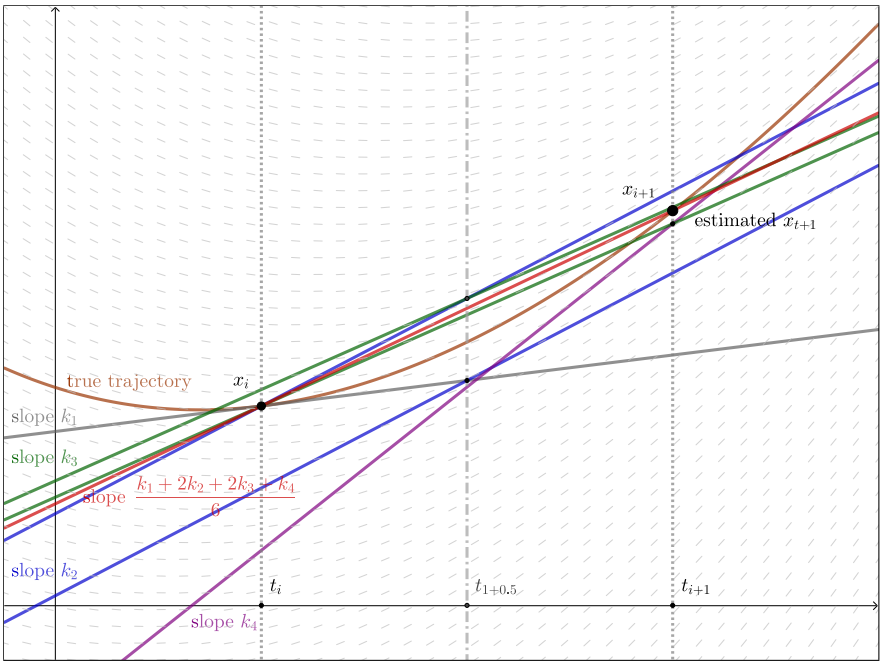


Fig. 2.7. A visualization of the RK4 step.

2.11.5. Implicit Methods

The methods described so far all fall under the category of *explicit methods*, meaning that they express an approximation for the future state in terms of quantities that are already known. Another approach is that of *implicit methods*, which approximate the integral with values which are currently not known, and thus need to be solved for. We now describe two of the simplest implicit methods.

Backward Euler Method: This method is like Euler's Method, but uses the gradient at the future timestep instead of the present one to approximate the dynamics:

$$\boldsymbol{x}(t_{i+1}) \approx \boldsymbol{x}(t_i) + h \cdot f(\boldsymbol{x}_{i+1}, t_{i+1}).$$

Since the value of $\boldsymbol{x}_{i+1}$ is unknown, it must be solved for. One approach is to use fixed-point iteration:

$$\boldsymbol{x}_{i+1}^0 \leftarrow \boldsymbol{x}_i$$

repeat until convergence:

$$\boldsymbol{x}_{i+1}^{k+1} \leftarrow \boldsymbol{x}_i + h \cdot f(\boldsymbol{x}_{i+1}^k, t_{i+1}),$$

where superscript is used to indicate the fixed-point iteration step. Another approach is to use Euler's method to find an approximate value of $\boldsymbol{x}(t_{i+1})$, and then use it as a guess for Newton's method to find the fixed point.

Trapezoid Method: This method is like Heun's Method, except instead of using an Euler estimate of $\boldsymbol{x}_{i+1}$, we solve for it:

$$\boldsymbol{x}(t_{i+1}) \approx \boldsymbol{x}(t_i) + h \cdot \frac{f(\boldsymbol{x}(t_i), t_i) + f(\boldsymbol{x}(t_{i+1}), t_{i+1})}{2}$$

Similar methods as those discussed in Backward Euler can be used here to solve for $\boldsymbol{x}(t_{i+1})$.

Because of the need to solve for an unknown state, implicit methods generally involve more computation than explicit methods. This is compensated by the fact that they often have better stability properties [**Butcher, 2016**].

In Figure 2.8, we show a comparison of the performance of the different methods on a toy dynamics equation $f(x,t) = 0.2x$, starting in state $x_0 = 1$, and taking five steps of stepsize $h = 2$. For the root solve, we used an off-the-shelf root finding algorithm.

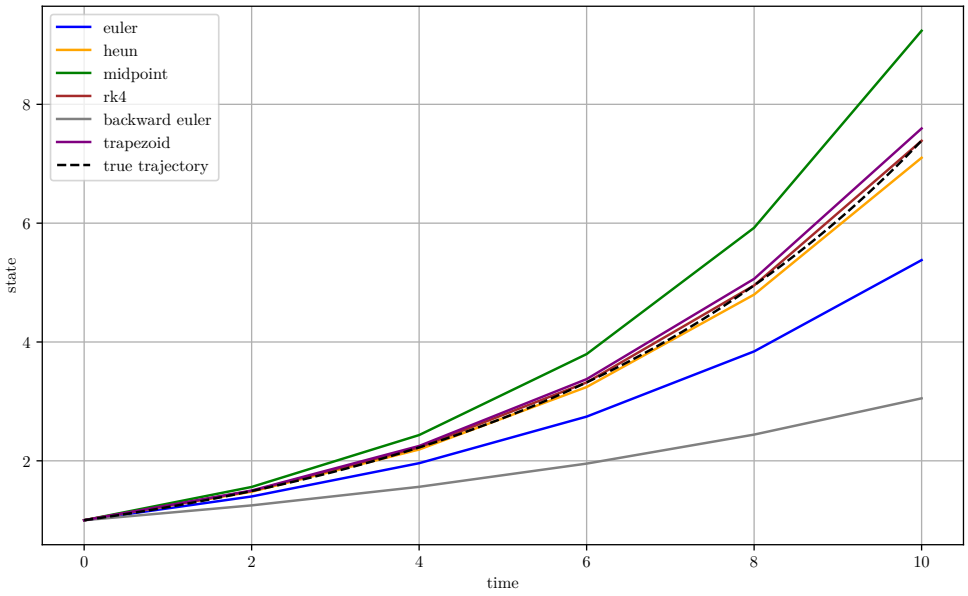


Fig. 2.8. Comparison of the accuracy of different numerical integration methods on the dynamics $f(x,t) = 0.2x$. On this problem, RK4 significantly outperforms the other methods, followed by Heun's method and the Trapezoid method.

Table 2.1 shows the time it took to perform these integrations. As expected, the indirect methods, which require a root solve, take longer. Furthermore, in computer programming frameworks which support just-in-time compilation [**Aycock, 2003**], explicit methods are straightforward to use, because they only depend on already-known values in their computation. Implicit methods, on the other hand, are generally significantly harder to program for just-in-time compilation, due to complications arising from their reliance on a zero-finding algorithm.

Method	Time, unjitted (s)	Time, jitted (s)
Euler	0.004	0.003
Heun	0.008	0.004
Midpoint	0.01	0.004
RK4	0.02	0.003
Backward Euler	0.01	NA
Trapezoid	0.02	NA

Table 2.1. Comparison of the speed of various numerical integration strategies. Times displayed are the total amount of time to integrate the trajectory, averaged over five runs.

It is also important to note that the explicit methods can be differentiated more easily using automatic differentiation. For the implicit methods, differentiation requires either a unrolling a differentiable zero-finding algorithm, or use of the Implicit Function Theorem, which we will see in Section 2.17.1.

2.11.6. Variable-Step Methods

More advanced numerical integration methods allow for the step size h to change depending on the estimated rate of change of dynamics f in the region surrounding the current state. In particular, when the dynamics change slowly, this can allow for fewer, larger steps to be taken, thus speeding up the calculation. Conversely, when the dynamics change quickly, an adaptive method will require finer steps to be taken, thus maintaining a level of precision which might be lost by using a constant stepsize. Of course, variable-step methods are more complicated to implement.

2.12. Boundary Value Problems

So far we have considered initial value problems, in which we are given a start state, and want to estimate how the state evolves over time, and in particular, the value of the state at the final timestep. Here, the main difficulty lies in finding an integration technique (numerical or, in certain cases, analytical) that will give us a solution with the desired speed and accuracy.

There is another kind of problem setup, known as boundary value problems (BVPs). Typically in BVPs, instead of fully specifying the start state $\boldsymbol{x}_0$, some parts of the start state *and* some parts of the end state are specified (they are required to be equal to given *boundary values*). For example, if the state $\boldsymbol{x} = \begin{bmatrix} x & \dot{x} \end{bmatrix}$ is a two-element vector representing position and velocity, then a common two-point BVP would be to have a required start

position and end position. The task is then to find which initial velocity will get us to the final required position at the right time.

More generally, in a BVP we are presented with with one or more boundary constraints (in the example, the start and end position), and one or more unknown parameters (in the example, the start and end velocity), and are asked to find the value of the parameter(s) such that the constraints are satisfied.

Because of the boundary constraints of a BVP, one must consider the entire trajectory at once, as opposed to only performing local calculations as in the IVP solution. A common approach is to make a guess at the unknown parameter(s), estimate the violation of the boundary requirements given that guess, and iteratively adjust the guess until the requirements are satisfied. We will now discuss various methods based on this idea.

2.12.1. Single Shooting

Say we have a dynamics parametrized by parameter(s) $\boldsymbol{\theta}$, subject to initial and final conditions $\boldsymbol{x}(t_s) = \boldsymbol{x}_s$ and $\boldsymbol{x}(t_f) = \boldsymbol{x}_f$ (for simplicity we write the boundary constraints like this, but in general, the initial and final conditions might only apply to a subset of the state dimensions). Then the BVP problem is to find $\boldsymbol{\theta}$ such that

$$\boldsymbol{x}_f = \boldsymbol{x}_s + \int_{t_s}^{t_f} f(\boldsymbol{x}(t), t, \boldsymbol{\theta})\, dt.$$

The single shooting approach is a straightforward way of translating the BVP into an NLP. We make $\boldsymbol{\theta}$ a decision variable of the NLP. Starting from an initial guess, we integrate the dynamics forward to find the final state corresponding to that choice of parameter. Finally, we express the difference between the desired final state and the one arising from that choice of parameter as an equality constraint of the NLP. This gives us the following:

decision variable	$\hat{\boldsymbol{\theta}}$
objective	0
equality constraints	$\boldsymbol{x}_f = \boldsymbol{x}_s + {}_hS_{t_s}^{t_f}(f, \boldsymbol{x}_0, \hat{\boldsymbol{\theta}})$
inequality constraints	None

We can use any off-the-shelf NLP solver to look for a value of $\boldsymbol{\theta}$ which satisfies the boundary conditions.

While single shooting is often a quick and effective way to solve BVPs, we note that at every step of the solve, we have to integrate the dynamics all the way from the start state

to the end state. Furthermore, when calculating the Jacobian of the constraint with respect to the parameter, we must propagate backwards through the entire integration. As such, for problems with long time horizons, this technique might be slow, or in some cases, not converge to a solution.

2.12.2. Multiple Shooting

We just saw how single shooting is fundamentally sequential in nature. We would like the parallelize some of the computation. We can do this by dividing the time horizon into intervals, and then optimizing over each of those intervals using single shooting. This operation can be performed in parallel, both in the forward integration phase, and when we calculate derivatives through the integration process.

The price we must pay for this parallelization is the introduction of additional decision variables and equality constraints to our NLP. In particular, we must know where to start the integration for each of the shooting intervals. Thus, we must add the interval starting states as decision variables to the NLP

Formally, we denote the shooting segment length k as the number of timesteps in each shooting interval. To simplify the presentation, we require that k divide N, so that all the shooting intervals are the same length (though this is not a theoretical requirement for the method). The starting states for each interval, along with the final state, will then be: $\boldsymbol{x}_0, \boldsymbol{x}_k, \boldsymbol{x}_{2k}, \ldots, \boldsymbol{x}_{N-k}, \boldsymbol{x}_N$. The resulting NLP is of the form[3]:

$$
\begin{aligned}
\text{decision variables} \quad & \hat{\boldsymbol{\theta}}, \hat{\boldsymbol{x}}_0, \hat{\boldsymbol{x}}_k, \hat{\boldsymbol{x}}_{2k}, \ldots, \hat{\boldsymbol{x}}_{N-k}, \hat{\boldsymbol{x}}_N \\
\text{objective} \quad & 0 \\
\text{equality constraints} \quad & \hat{\boldsymbol{x}}_k = \hat{\boldsymbol{x}}_0 + {}_hS_{t_0}^{t_k}(f, \hat{\boldsymbol{x}}_0, \hat{\boldsymbol{\theta}}) \\
& \hat{\boldsymbol{x}}_{2k} = \hat{\boldsymbol{x}}_k + {}_hS_{t_k}^{t_{2k}}(f, \hat{\boldsymbol{x}}_k, \hat{\boldsymbol{\theta}}) \\
& \hat{\boldsymbol{x}}_{3k} = \hat{\boldsymbol{x}}_{2k} + {}_hS_{t_{2k}}^{t_{3k}}(f, \hat{\boldsymbol{x}}_{2k}, \hat{\boldsymbol{\theta}}) \\
& \quad \vdots \\
& \hat{\boldsymbol{x}}_N = \hat{\boldsymbol{x}}_{N-k} + {}_hS_{t_{N-k}}^{t_N}(f, \hat{\boldsymbol{x}}_{N-k}, \hat{\boldsymbol{\theta}}) \\
& \hat{\boldsymbol{x}}_0 = \boldsymbol{x}_s \\
& \hat{\boldsymbol{x}}_N = \boldsymbol{x}_f
\end{aligned}
$$

[3]Note that we have included decision variables for the start and end states, even if we know what they must be. The benefit of this is both to simplify the presentation, and to allow for easier vectorized calculations in the resulting code. However, it is also possible to omit the start and end states from the decision variables, and simply use the true start and final state (from the boundary conditions) in their place.

inequality constraints None

2.12.3. Comparing Single and Multiple Shooting

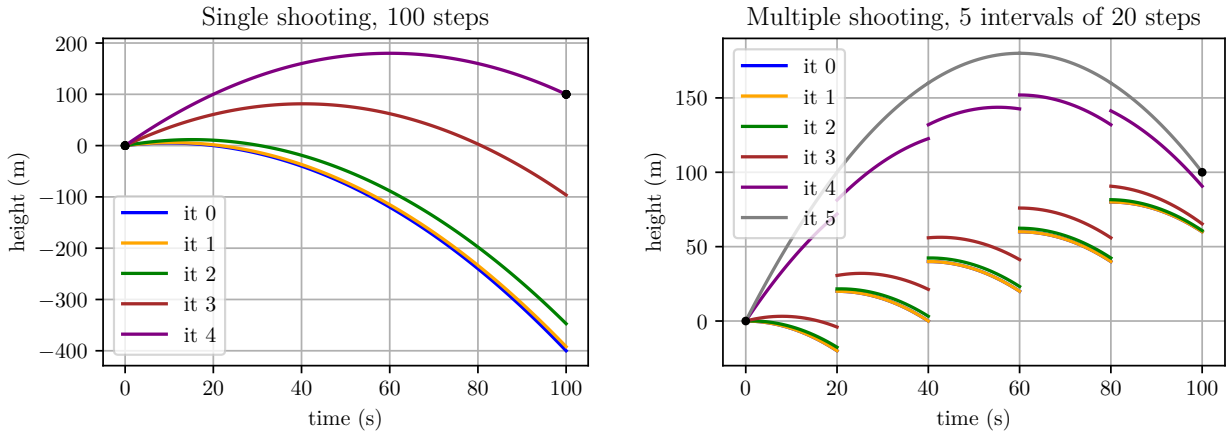
We can visualize the differences between single and multiple shooting by means of a toy problem. In this problem, we start at ground level, and want to launch a projectile up in the air so that after 100s, it is at a vertical location of 100m. We assume constant gravitational acceleration of $g = 10m/s^2$ and no air resistance. The state is a two-dimensional vector of position and velocity:

$$\boldsymbol{x} = \begin{bmatrix} x & \dot{x} \end{bmatrix}^\top.$$

The dynamics equation is simply

$$f(\boldsymbol{x}, t) = \begin{bmatrix} \dot{x} & -g \end{bmatrix}.$$

Figures 2.9a and 2.9b show the results of the first few iterations of single and multiple shooting, respectively.



(a) The trajectories from applying single shooting after 0 to 4 iterations.

(b) The trajectories from applying multiple shooting after 0 to 5 iterations.

Fig. 2.9. Comparison of single shooting (a) and multiple shooting (b), applied to a toy problem with dynamics $f(\boldsymbol{x}, t) = \begin{bmatrix} \dot{x} & -g \end{bmatrix}$.

We see visually that the single shooting algorithm is inherently sequential, while parallel shooting allows each sub-trajectory to be calculated independently, before gathering the results to enforce the constraints at each timestep.

While in this case single shooting does converge one iteration sooner, it is in fact the slower algorithm. Table 2.2 compares the wall-clock time taken to complete the different numbers of iterations from Figure 2.9.

# Iterations	Calculation Time (s)	
	Single Shooting	Multiple Shooting
1	0.29	0.16
2	0.51	0.29
3	0.81	0.42
4	1.02	0.52
5	1.20	0.67
6	1.59	0.78

Table 2.2. Timing of single and multiple shooting on one run of the toy problem with dynamics $f(\boldsymbol{x}, t) = \begin{bmatrix} \dot{x} & -g \end{bmatrix}$.

As we increase the time horizon (and the number of shooting intervals for multiple shooting), we expect this timing discrepancy to increase. However, introducing more shooting intervals does complicate the resulting NLP by creating more equality constraints, so it is not necessarily always better to use multiple shooting.

2.12.4. Trapezoidal Collocation

Multiple shooting is often advantageous because of its increased parallelization compared with single shooting. Taking this parallelization to the limit, we could imagine reducing the shooting intervals to be so small that each interval contains only a single timestep of our numerical integration scheme. If we are using Heun's method as our integration scheme, then the integral over a single interval is

$$ {}_hS_{t_i}^{t_{i+1}}(f, \hat{\boldsymbol{x}}_i, \hat{\boldsymbol{\theta}}) = h \cdot \frac{f(\hat{\boldsymbol{x}}_i, t_i, \hat{\boldsymbol{\theta}}) + f(\tilde{\hat{\boldsymbol{x}}}_{i+1}, t_{i+1}, \hat{\boldsymbol{\theta}})}{2}, $$

where $\tilde{\hat{\boldsymbol{x}}}_{i+1} = \hat{\boldsymbol{x}}_i + h \cdot f(\hat{\boldsymbol{x}}_i, t_i, \hat{\boldsymbol{\theta}})$.

Remember from our discussion of implicit integration methods in Section 2.11.5 that the Heun's method is an explicit equivalent to the implicit trapezoid rule. Yet unlike in the context of implicit numerical integration, where using the trapezoid rule required solving for the state at the next timestep, here we already have a guess of the state at the next step, as it is a decision varaible of our NLP: $\hat{\boldsymbol{x}}_{i+1}$. It is this realization that gives us the following approximation of the integral across one timestep:

$$ {}_TC_{t_i}^{t_{i+1}}(f, \hat{\boldsymbol{x}}_i, \hat{\boldsymbol{x}}_{i+1}, \hat{\boldsymbol{\theta}}) = h \cdot \frac{f(\hat{\boldsymbol{x}}_i, t_i, \hat{\boldsymbol{\theta}}) + f(\hat{\boldsymbol{x}}_{i+1}, t_{i+1}, \hat{\boldsymbol{\theta}})}{2}, $$

where we note that the h is no longer necessary to carry in the lower left subscript because we can recover it via $h = t_{i+1} - t_i$, and we instead replace it with an uppercase T to remind ourselves we're dealing with Trapezoidal collocation.

With this in mind, we can modify our multiple shooting NLP into a completely parallelized one:

$$
\begin{aligned}
\text{decision variables} \quad & \hat{\boldsymbol{\theta}}, \hat{\boldsymbol{x}}_0, \hat{\boldsymbol{x}}_1, \hat{\boldsymbol{x}}_2, \ldots, \hat{\boldsymbol{x}}_N \\
\text{objective} \quad & 0 \\
\text{equality constraints} \quad & \hat{\boldsymbol{x}}_1 = \hat{\boldsymbol{x}}_0 + {}_T C_{t_0}^{t_1}(f, \hat{\boldsymbol{x}}_0, \hat{\boldsymbol{x}}_1, \hat{\boldsymbol{\theta}}) \\
& \hat{\boldsymbol{x}}_2 = \hat{\boldsymbol{x}}_1 + {}_T C_{t_1}^{t_2}(f, \hat{\boldsymbol{x}}_1, \hat{\boldsymbol{x}}_2, \hat{\boldsymbol{\theta}}) \\
& \hat{\boldsymbol{x}}_3 = \hat{\boldsymbol{x}}_2 + {}_T C_{t_2}^{t_3}(f, \hat{\boldsymbol{x}}_2, \hat{\boldsymbol{x}}_3, \hat{\boldsymbol{\theta}}) \\
& \quad \vdots \\
& \hat{\boldsymbol{x}}_N = \hat{\boldsymbol{x}}_{N-1} + {}_T C_{t_{N-1}}^{t_N}(f, \hat{\boldsymbol{x}}_{N-1}, \hat{\boldsymbol{x}}_N, \hat{\boldsymbol{\theta}}) \\
& \hat{\boldsymbol{x}}_0 = \boldsymbol{x}_s \\
& \hat{\boldsymbol{x}}_N = \boldsymbol{x}_f \\
\text{inequality constraints} \quad & \text{None}
\end{aligned}
$$

Trapezoidal collocation performs similarly to completely parallelized multiple shooting with Heun integration, but with the added stability of using an indirect integration method.

2.12.5. Hermite-Simpson Collocation

Trapezoidal collocation is but the first of many different ways of pairing implicit integration methods with fully parallelized problem formulations. We will see one more such method, which uses a more accurate integration scheme: Hermite-Simpson collocation. It uses the Simpson quadrature rule for integration, and gives a state trajectory which can be interpolated with Hermite splines [**Burden et al., 2016**].

Simpson's rule approximates the integral between two points as

$$
{}_{HS} C_{t_i}^{t_{i+1}}(f, \hat{\boldsymbol{x}}_i, \hat{\boldsymbol{x}}_{i+1}, \hat{\boldsymbol{\theta}}) = \frac{h}{6}\left(f(\hat{\boldsymbol{x}}_i, t_i, \hat{\boldsymbol{\theta}}) + 4f(\hat{\boldsymbol{x}}_{i+0.5}, t_{i+0.5}, \hat{\boldsymbol{\theta}}) + f(\hat{\boldsymbol{x}}_{i+1}, t_{i+1}, \hat{\boldsymbol{\theta}})\right),
$$

where the midpoints can be directly calculated as

$$
\begin{aligned}
t_{i+0.5} &= t_i + \frac{h}{2} \\
\hat{\boldsymbol{x}}_{i+0.5} &= \frac{1}{2}(\hat{\boldsymbol{x}}_i + \hat{\boldsymbol{x}}_{i+1}) + \frac{h}{8}(f(\hat{\boldsymbol{x}}_i, t_i, \hat{\boldsymbol{\theta}}) - f(\hat{\boldsymbol{x}}_{i+1}, t_{i+1}, \hat{\boldsymbol{\theta}}))
\end{aligned}
\tag{2.12.1}
$$

Putting these rules together into an NLP gives the following:

$$
\begin{aligned}
\text{decision variables} \quad & \hat{\boldsymbol{\theta}}, \hat{\boldsymbol{x}}_0, \hat{\boldsymbol{x}}_{0.5}, \hat{\boldsymbol{x}}_1, \hat{\boldsymbol{x}}_{1.5}, \hat{\boldsymbol{x}}_2, \ldots, \hat{\boldsymbol{x}}_{N-0.5}, \hat{\boldsymbol{x}}_N \\
\text{objective} \quad & 0 \\
\text{equality constraints} \quad & \hat{\boldsymbol{x}}_1 = \hat{\boldsymbol{x}}_0 + {}_{HS}C_{t_0}^{t_1}(f, \hat{\boldsymbol{x}}_0, \hat{\boldsymbol{x}}_{0.5}, \hat{\boldsymbol{x}}_1, \hat{\boldsymbol{\theta}}) \\
& \hat{\boldsymbol{x}}_2 = \hat{\boldsymbol{x}}_1 + {}_{HS}C_{t_1}^{t_2}(f, \hat{\boldsymbol{x}}_1, \hat{\boldsymbol{x}}_{1.5}, \hat{\boldsymbol{x}}_2, \hat{\boldsymbol{\theta}}) \\
& \hat{\boldsymbol{x}}_3 = \hat{\boldsymbol{x}}_2 + {}_{HS}C_{t_2}^{t_3}(f, \hat{\boldsymbol{x}}_2, \hat{\boldsymbol{x}}_{2.5}, \hat{\boldsymbol{x}}_3, \hat{\boldsymbol{\theta}}) \\
& \quad \vdots \\
& \hat{\boldsymbol{x}}_N = \hat{\boldsymbol{x}}_{N-1} + {}_{HS}C_{t_{N-1}}^{t_N}(f, \hat{\boldsymbol{x}}_{N-1}, \hat{\boldsymbol{x}}_{N-0.5}, \hat{\boldsymbol{x}}_N, \hat{\boldsymbol{\theta}}) \\
& \hat{\boldsymbol{x}}_{0.5} = \frac{1}{2}(\hat{\boldsymbol{x}}_0 + \hat{\boldsymbol{x}}_1) + \frac{h}{8}(f(\hat{\boldsymbol{x}}_0, t_0, \hat{\boldsymbol{\theta}}) - f(\hat{\boldsymbol{x}}_1, t_1, \hat{\boldsymbol{\theta}})) \\
& \hat{\boldsymbol{x}}_{1.5} = \frac{1}{2}(\hat{\boldsymbol{x}}_1 + \hat{\boldsymbol{x}}_2) + \frac{h}{8}(f(\hat{\boldsymbol{x}}_1, t_1, \hat{\boldsymbol{\theta}}) - f(\hat{\boldsymbol{x}}_2, t_2, \hat{\boldsymbol{\theta}})) \\
& \quad \vdots \\
& \hat{\boldsymbol{x}}_{N-0.5} = \frac{1}{2}(\hat{\boldsymbol{x}}_{N-1} + \hat{\boldsymbol{x}}_N) + \frac{h}{8}(f(\hat{\boldsymbol{x}}_{N-1}, t_{N-1}, \hat{\boldsymbol{\theta}}) - f(\hat{\boldsymbol{x}}_N, t_N, \hat{\boldsymbol{\theta}})) \\
& \hat{\boldsymbol{x}}_0 = \boldsymbol{x}_s \\
& \hat{\boldsymbol{x}}_N = \boldsymbol{x}_f \\
\text{inequality constraints} \quad & 0
\end{aligned}
$$

As an aside, note that here we have left the state midpoints as decision variables. Alternatively, we could use Equation (2.12.1) to calculate them at each pass through the NLP solver. This would result is fewer decision variables and fewer constraints, but a slightly more computationally demanding problem.

2.13. Trajectory Optimization

Boundary value problems ask us to find some parameter, such as starting velocity, which will lead to a trajectory which satisfies the boundary conditions.

Trajectory optimization encompasses a slightly different set of problems, in which we are asked to provide a control function, which will be applied over time, to minimize some cost integrated over time. In some cases, our trajectory optimization problem might also have boundary conditions[4].

The components of a trajectory optimization problem are a start state, an (optional) end state, a dynamics function, and a cost function[5]. The problem can be written as

$$\begin{aligned}
\min_{\boldsymbol{u}(t)\ \forall t\in[t_s,t_f]} \quad & \int_{t_s}^{t_f} c(\boldsymbol{x}(t),\boldsymbol{u}(t),t)\ dt \\
\text{such that} \quad & \dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t),\boldsymbol{u}(t),t)\ \forall t \in [t_s,t_f] \\
\text{with} \quad & \boldsymbol{x}(t_s) = \boldsymbol{x}_s \\
\text{*and} \quad & \boldsymbol{x}(t_f) = \boldsymbol{x}_f \\
\text{*and} \quad & \boldsymbol{x}_{\text{lower}}(t) \leq \boldsymbol{x}(t) \leq \boldsymbol{x}_{\text{upper}}(t)\ \forall t \in [t_s,t_f] \\
\text{*and} \quad & \boldsymbol{u}_{\text{lower}}(t) \leq \boldsymbol{u}(t) \leq \boldsymbol{u}_{\text{upper}}(t)\ \forall t \in [t_s,t_f]
\end{aligned}$$

where the lines beginning with an asterisk (*) are optional.

There are two general approaches to solving trajectory optimization problems: direct and indirect methods[6]. Direct methods first discretize the problem, reformulate it as an NLP, and then solve it using an NLP solver. Indirect methods instead manipulate the dynamics and cost functions to create an additional ordinary differential equation known as the adjoint equation. Coupling this with the original dynamics, one can form a BVP, which can in turn be discretized and solved as an NLP.

Direct methods are generally faster and less sensitive to the initial guess. Indirect methods tend to give more accurate solutions, but require a good initial guess, and can take longer. In this thesis, we primarily focus on direct methods.

Before diving into the methods, we note that the formulation of an optimal control problem may also include a *terminal cost*, that is, a cost which is exclusively dependent on the final state and control. It is a well-known result that a problem with an integral cost plus a final cost (known as a *Bolza* problem) can be transformed into one with only an integral cost (known as a *Lagrange* problem). As such, our presentation of methods to solve problems without a terminal cost can also be applied to problems with a terminal cost [**Bliss, 1980**].

[4]BVPs can then be seen as a subset of this general class of problems, with a 'control' that is only applied at the start and end of the trajectory, and there is no cost function.

[5]Since $\boldsymbol{u}$ and $\boldsymbol{x}$ are functions over time, the dynamics and cost are higher-order functions. However, in practice, $\boldsymbol{u}$ and $\boldsymbol{x}$ are always approximated at a finite number of points, in which case the dynamics and cost become first-order functions.

[6]The distinction between direct and indirect methods for approaching an optimal control problem is not to be confused with the distinction between explicit and implicit integration methods – both can be mixed and matched.

Finally, we also note that our formulation can be modified to incorporate the duration of the process itself, T, as a decision variable. This can be useful for applications in which we also want to optimize over the time horizon. Keeping a fixed number of timesteps, this formulation will lead to timesteps which change in size as different values of T are tried by the NLP solver during the solve [**Betts, 2010**].

2.14. Direct Methods

We are now in a setting where the dynamics function also takes a control value:

$$\dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t), \boldsymbol{u}(t), t).$$

When we discretize the problem, we use a numerical integration method to integrate the dynamics. We also choose a discretization of the control signal $\boldsymbol{u}(t)$ which matches the integration scheme. By 'match', we mean that the intervals of the control discretization should align with those of the integration discretization. In this way, the problem of optimizing over all possible control functions simplifies into an optimization over finite control vectors.

Our definitions of Euler, Heun, Midpoint, and RK4 step, as well as our definitions of Trapezoidal and Hermite-Simpson quadrature, also need to be updated to include controls:

Euler step:

$$\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + h \cdot f(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i).$$

Heun step:

$$\begin{aligned}
\tilde{\boldsymbol{x}}_{i+1} &= \boldsymbol{x}_i + h \cdot f(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i) \\
\boldsymbol{x}_{i+1} &\leftarrow \boldsymbol{x}_i + h \cdot \frac{f(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i) + f(\tilde{\boldsymbol{x}}_{i+1}, \boldsymbol{u}_{i+1}, t_{i+1})}{2}
\end{aligned}$$

Midpoint step:

$$\begin{aligned}
\tilde{\boldsymbol{x}}_{i+0.5} &= \boldsymbol{x}_i + h \cdot f(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i) \\
\tilde{\boldsymbol{u}}_{i+0.5} &= \frac{1}{2} \cdot (\boldsymbol{u}_i + \boldsymbol{u}_{i+1}) \\
\boldsymbol{x}_{i+1} &\leftarrow \boldsymbol{x}_i + h \cdot f(\tilde{\boldsymbol{x}}_{i+0.5}, \tilde{\boldsymbol{u}}_{i+0.5}, t_{i+0.5})
\end{aligned}$$

RK4 step:

$$\begin{aligned}
\boldsymbol{k}_1 &= f\left(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i\right) \\
\boldsymbol{k}_2 &= f\left(\boldsymbol{x}_i + \frac{k_1}{2}h, \boldsymbol{u}_{i+0.5}, t_i + \frac{h}{2}\right) \\
\boldsymbol{k}_3 &= f\left(\boldsymbol{x}_i + \frac{k_2}{2}h, \boldsymbol{u}_{i+0.5}, t_i + \frac{h}{2}\right) \\
\boldsymbol{k}_4 &= f\left(\boldsymbol{x}_i + \boldsymbol{k}_3 h, \boldsymbol{u}_{i+1}, t_{i+1}\right) \\
\boldsymbol{x}_{i+1} &\leftarrow \boldsymbol{x}_i + \frac{h}{6}\left(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4\right)
\end{aligned}$$

Trapezoidal quadrature:

$${}_T C_{t_i}^{t_{i+1}}(f, \boldsymbol{x}_i, \boldsymbol{x}_{i+1}, \boldsymbol{u}_i, \boldsymbol{u}_{i+1}) = h \cdot \frac{f(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i) + f(\boldsymbol{x}_{i+1}, \boldsymbol{u}_{i+1}, t_{i+1})}{2},$$

Simpson quadrature:

$${}_{HS} C_{t_i}^{t_{i+1}}(f, \boldsymbol{x}_i, \boldsymbol{x}_{i+1}, \boldsymbol{u}_i, \boldsymbol{u}_{i+1}) = \frac{h}{6}\left(f(\boldsymbol{x}_i, \boldsymbol{u}_i, t_i) + 4f(\boldsymbol{x}_{i+0.5}, \boldsymbol{u}_{i+0.5}, t_{i+0.5}) + f(\boldsymbol{x}_{i+1}, \boldsymbol{u}_{i+1}, t_{i+1})\right)$$

It is worth noting that some implementations of Runge-Kutta 4th Order method approximate the midpoint $\boldsymbol{u}_{i+0.5}$ as $\frac{1}{2} \cdot (\boldsymbol{u}_i + \boldsymbol{u}_{i+1})$, instead of keeping it as a decision variable. While it might not cause problems in practice, it seems conceptually incorrect to interpolate the controls linearly in RK4. This intuition stems from the fact that when f is state-independent (that is, only depends on t and $\boldsymbol{u}(t)$), Runge-Kutta integration is equivalent to Simpson's rule, and Hermite-Simpson collocation assumes a trajectory which uses quadratic of controls, not linear. For this reason, in this thesis, the midpoint control is kept as a decision variable.

With these updated function signatures, we can now go about expressing our trajectory optimization problem as an NLP.

2.14.1. Direct Single Shooting

Direct single shooting for optimal control works in a similar way as single shooting for BVPs. We replace the generic parameter term $\boldsymbol{\theta}$ with a vector of controls $\boldsymbol{u}_{0:N}$, and then apply the controls one-by-one at each timestep.

When we were discussing BVPs, we only needed to find a parameter $\boldsymbol{\theta}$ which satisfied the system dynamics and the endpoint constraints. Now that we are dealing with an optimal control problem, we want to also minimize a cost function which is integrated over time. In direct shooting methods, we do this by augmenting the state at each timestep with the instantaneous cost, and then changing the computer programming expression for the dynamics correspondingly.

If our instantaneous cost function is $c(\boldsymbol{x}(t), \boldsymbol{u}(t), t)$ and our dynamics function is $f(\boldsymbol{x}(t), \boldsymbol{u}(t), t)$, we can combine the two into an augmented dynamics function:

$$f_{\text{aug}}(\boldsymbol{x}(t), \boldsymbol{u}(t), t) = \begin{bmatrix} f(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \\ c(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \end{bmatrix}.$$

In particular, if we take the integral

$$\begin{bmatrix} \boldsymbol{x}_s \\ 0 \end{bmatrix} + \int_{t_s}^{t_f} f_{\text{aug}}(\boldsymbol{x}(t), \boldsymbol{u}(t), t)\ dt = \begin{bmatrix} \boldsymbol{x}_f \\ c_f \end{bmatrix},$$

we see that the final entry in the resulting vector is the total system cost, which we would like to minimize. Of course, in practice, this integral will be calculated with a numerical integration scheme.

We also need to redefine our numerical integration scheme to handle control-dependent dynamics. Let

$${}_h S_{t_s}^{t_f}(f, \boldsymbol{x}_s, \boldsymbol{u}_{0:N}) \approx \int_{t_s}^{t_f} f(\boldsymbol{x}(t), \psi_h(\boldsymbol{u}_{0:N}, t), t)\ dt,$$

where ψ_h is a function which takes a sequence of controls and a time, and, based on the time and timestep, returns an interpolated control value[7]. Then, the total integrated cost can be recovered as

$$c_f = {}_h S_{t_s}^{t_f}\left(f_{\text{aug}}, \begin{bmatrix} \boldsymbol{x}_s \\ 0 \end{bmatrix}, \boldsymbol{u}_{0:N}\right)\texttt{[-1]},$$

where the `[-1]` refers to accessing the final element of the vector, which is this case corresponds to the cost integrated over the entire state trajectory.

All together, our problem so far be written as

$$\begin{aligned} \min_{\boldsymbol{u}_{0:N}} \quad & {}_h S_{t_s}^{t_f}\left(f_{\text{aug}}, \begin{bmatrix} \boldsymbol{x}_s \\ 0 \end{bmatrix}, \boldsymbol{u}_{0:N}\right)\texttt{[-1]} \\ \text{such that} \quad & \boldsymbol{x}_f = \boldsymbol{x}_s + {}_h S_{t_s}^{t_f}(f, \boldsymbol{x}_s, \boldsymbol{u}_{0:N}) \\ \text{and} \quad & \boldsymbol{x}(t_s) = \boldsymbol{x}_s \\ *\text{and} \quad & \boldsymbol{x}(t_f) = \boldsymbol{x}_f \\ *\text{and} \quad & \boldsymbol{x}_{\text{lower}}(t) \leq \boldsymbol{x}(t) \leq \boldsymbol{x}_{\text{upper}}(t)\ \forall t \in [t_s, t_f] \end{aligned}$$

[7]How this interpolation is performed depends on the integration method applied. Matching the control discretization with a fixed integration timestep circumvents the need for interpolation.

$$\text{*and} \quad \boldsymbol{u}_{\text{lower}}(t) \leq \boldsymbol{u}(t) \leq \boldsymbol{u}_{\text{upper}}(t) \; \forall t \in [t_s, t_f]$$

where the lines beginning with an asterisk (*) are optional.

The above problem description is almost, but not quite, an NLP: we don't have a clear way of translating the path constraints on the state into the structure of an NLP. The best we can do in the single shooting formulation is fix the start and end points; the rest we don't have control over. We can do better with the controls though: we can set constraints at all the "knot points" (that is, the points corresponding to timesteps), and feed that into the solver as an inequality constraint.

As such, our NLP becomes

$$\begin{aligned}
\text{decision variables} \quad & \hat{\boldsymbol{u}}_0, \hat{\boldsymbol{u}}_1, \hat{\boldsymbol{u}}_2, \ldots, \hat{\boldsymbol{u}}_N \\
\text{objective} \quad & {}_h S_{t_s}^{t_f} \left(f_{\text{aug}}, \begin{bmatrix} \boldsymbol{x}_s \\ 0 \end{bmatrix}, \hat{\boldsymbol{u}}_{0:N} \right) \texttt{[-1]} \\
\text{*equality constraints} \quad & \boldsymbol{x}_f = \boldsymbol{x}_s + {}_h S_{t_s}^{t_f}(f, \boldsymbol{x}_s, \hat{\boldsymbol{u}}_{0:N}) \\
\text{*inequality constraints} \quad & \boldsymbol{u}_0^{\text{lower}} \leq \hat{\boldsymbol{u}}_0 \leq \boldsymbol{u}_0^{\text{upper}} \\
* \quad & \boldsymbol{u}_1^{\text{lower}} \leq \hat{\boldsymbol{u}}_1 \leq \boldsymbol{u}_1^{\text{upper}} \\
* \quad & \vdots \\
* \quad & \boldsymbol{u}_N^{\text{lower}} \leq \hat{\boldsymbol{u}}_N \leq \boldsymbol{u}_N^{\text{upper}}
\end{aligned}$$

In many applications, it might be desirable to be able to require path constraints on the state trajectory as well as the control trajectory. Furthermore, we might like to parallelize our solving of the problem. Multiple shooting can help us with both these issues.

2.14.2. Direct Multiple Shooting

The translation of our trajectory optimization problem into a multiple shooting problem is analogous to what we saw in the single shooting section, with the addition of decision variables for the states at the boundaries of shooting intervals, and equality constraints ensuring the dynamics properly match up across intervals. The resulting NLP is

$$\begin{aligned}
\text{decision variables} \quad & \hat{\boldsymbol{x}}_0, \hat{\boldsymbol{x}}_k, \hat{\boldsymbol{x}}_{2k}, \ldots, \hat{\boldsymbol{x}}_{N-k}, \hat{\boldsymbol{x}}_N, \hat{\boldsymbol{u}}_0, \hat{\boldsymbol{u}}_1, \hat{\boldsymbol{u}}_2, \ldots, \hat{\boldsymbol{u}}_N \\
\text{objective} \quad & \sum_{j=0}^{\frac{N}{k}-1} {}_h S_{t_{jk}}^{t_{(j+1)k}} \left(f_{\text{aug}}, \begin{bmatrix} \hat{\boldsymbol{x}}_{jk} \\ 0 \end{bmatrix}, \hat{\boldsymbol{u}}_{jk:(j+1)k} \right) \texttt{[-1]}
\end{aligned}$$

$$
\begin{aligned}
\text{equality constraints} \quad & \hat{\boldsymbol{x}}_k = \hat{\boldsymbol{x}}_0 + {}_h S_{t_0}^{t_k}(f, \hat{\boldsymbol{x}}_0, \hat{\boldsymbol{u}}_{0:k}) \\
& \hat{\boldsymbol{x}}_{2k} = \hat{\boldsymbol{x}}_k + {}_h S_{t_k}^{t_{2k}}(f, \hat{\boldsymbol{x}}_k, \hat{\boldsymbol{u}}_{k:2k}) \\
& \hat{\boldsymbol{x}}_{3k} = \hat{\boldsymbol{x}}_{2k} + {}_h S_{t_{2k}}^{t_{3k}}(f, \hat{\boldsymbol{x}}_{2k}, \hat{\boldsymbol{u}}_{2k:3k}) \\
& \vdots \\
& \hat{\boldsymbol{x}}_N = \hat{\boldsymbol{x}}_{N-k} + {}_h S_{t_{N-k}}^{t_N}(f, \hat{\boldsymbol{x}}_{N-k}, \hat{\boldsymbol{u}}_{N-k:N}) \\
& \hat{\boldsymbol{x}}_0 = \boldsymbol{x}_s \\
* \quad & \hat{\boldsymbol{x}}_N = \boldsymbol{x}_f \\
\text{*inequality constraints} \quad & \boldsymbol{x}_0^{\text{lower}} \leq \hat{\boldsymbol{x}}_0 \leq \boldsymbol{x}_0^{\text{upper}} \\
* \quad & \boldsymbol{x}_k^{\text{lower}} \leq \hat{\boldsymbol{x}}_k \leq \boldsymbol{x}_k^{\text{upper}} \\
* \quad & \boldsymbol{x}_{2k}^{\text{lower}} \leq \hat{\boldsymbol{x}}_{2k} \leq \boldsymbol{x}_{2k}^{\text{upper}} \\
* \quad & \vdots \\
* \quad & \boldsymbol{x}_N^{\text{lower}} \leq \hat{\boldsymbol{x}}_N \leq \boldsymbol{x}_N^{\text{upper}} \\
* \quad & \boldsymbol{u}_0^{\text{lower}} \leq \hat{\boldsymbol{u}}_0 \leq \boldsymbol{u}_0^{\text{upper}} \\
* \quad & \boldsymbol{u}_1^{\text{lower}} \leq \hat{\boldsymbol{u}}_1 \leq \boldsymbol{u}_1^{\text{upper}} \\
* \quad & \boldsymbol{u}_2^{\text{lower}} \leq \hat{\boldsymbol{u}}_2 \leq \boldsymbol{u}_2^{\text{upper}} \\
* \quad & \vdots \\
* \quad & \boldsymbol{u}_N^{\text{lower}} \leq \hat{\boldsymbol{u}}_N \leq \boldsymbol{u}_N^{\text{upper}}
\end{aligned}
$$

As hoped, this formulation gives us more ability to enforce constraints on our state trajectory. Indeed, if we want, we can have state bounds at the start and end of every shooting interval. Unfortunately, within each shooting interval, we are still unable to provide guarantees on state values. However, the smaller we make the intervals, the more confident we can be that our states will stay bounded within a reasonable region. This is highly beneficial if we are solving problems in which safety concerns are an issue.

2.14.3. Direct Trapezoidal Collocation

The discretization of the dynamics, and corresponding equality constraints, also match the reasoning on BVPs in Section 2.12.4. An interesting new element, however, is the approximation of the cost integral. In fact, we can use the same trapezoidal collocation

rule to approximate the integral of cost function analogously to how we use this rule for the dynamics function.

This results in the following NLP:

$$
\begin{aligned}
\text{decision variables} \quad & \hat{\boldsymbol{x}}_0, \hat{\boldsymbol{x}}_1, \hat{\boldsymbol{x}}_2, \ldots, \hat{\boldsymbol{x}}_N, \hat{\boldsymbol{u}}_0, \hat{\boldsymbol{u}}_1, \hat{\boldsymbol{u}}_2, \ldots, \hat{\boldsymbol{u}}_N \\
\text{objective} \quad & \sum_{i=0}^{N-1} {}_T C_{t_i}^{t_{i+1}}(c, \hat{\boldsymbol{x}}_i, \hat{\boldsymbol{x}}_{i+1}, \hat{\boldsymbol{u}}_i, \hat{\boldsymbol{u}}_{i+1}) \\
\text{equality constraints} \quad & \hat{\boldsymbol{x}}_1 = \hat{\boldsymbol{x}}_0 + {}_T C_{t_0}^{t_1}(f, \hat{\boldsymbol{x}}_0, \hat{\boldsymbol{x}}_1, \hat{\boldsymbol{u}}_0, \hat{\boldsymbol{u}}_1) \\
& \hat{\boldsymbol{x}}_2 = \hat{\boldsymbol{x}}_1 + {}_T C_{t_1}^{t_2}(f, \hat{\boldsymbol{x}}_1, \hat{\boldsymbol{x}}_2, \hat{\boldsymbol{u}}_1, \hat{\boldsymbol{u}}_2) \\
& \hat{\boldsymbol{x}}_3 = \hat{\boldsymbol{x}}_2 + {}_T C_{t_2}^{t_3}(f, \hat{\boldsymbol{x}}_2, \hat{\boldsymbol{x}}_3, \hat{\boldsymbol{u}}_2, \hat{\boldsymbol{u}}_3) \\
& \quad \vdots \\
& \hat{\boldsymbol{x}}_N = \hat{\boldsymbol{x}}_{N-1} + {}_T C_{t_{N-1}}^{t_N}(f, \hat{\boldsymbol{x}}_{N-1}, \hat{\boldsymbol{x}}_N, \hat{\boldsymbol{u}}_{N-1}, \hat{\boldsymbol{u}}_N) \\
& \hat{\boldsymbol{x}}_0 = \boldsymbol{x}_s \\
* \quad & \hat{\boldsymbol{x}}_N = \boldsymbol{x}_f \\
\text{*inequality constraints} \quad & \boldsymbol{x}_0^{\text{lower}} \leq \hat{\boldsymbol{x}}_0 \leq \boldsymbol{x}_0^{\text{upper}} \\
* \quad & \boldsymbol{x}_1^{\text{lower}} \leq \hat{\boldsymbol{x}}_1 \leq \boldsymbol{x}_1^{\text{upper}} \\
* \quad & \boldsymbol{x}_2^{\text{lower}} \leq \hat{\boldsymbol{x}}_2 \leq \boldsymbol{x}_2^{\text{upper}} \\
* \quad & \vdots \\
* \quad & \boldsymbol{x}_N^{\text{lower}} \leq \hat{\boldsymbol{x}}_N \leq \boldsymbol{x}_N^{\text{upper}} \\
* \quad & \boldsymbol{u}_0^{\text{lower}} \leq \hat{\boldsymbol{u}}_0 \leq \boldsymbol{u}_0^{\text{upper}} \\
* \quad & \boldsymbol{u}_1^{\text{lower}} \leq \hat{\boldsymbol{u}}_1 \leq \boldsymbol{u}_1^{\text{upper}} \\
* \quad & \boldsymbol{u}_2^{\text{lower}} \leq \hat{\boldsymbol{u}}_2 \leq \boldsymbol{u}_2^{\text{upper}} \\
* \quad & \vdots \\
* \quad & \boldsymbol{u}_N^{\text{lower}} \leq \hat{\boldsymbol{u}}_N \leq \boldsymbol{u}_N^{\text{upper}}
\end{aligned}
$$

Because there are decision variables for control and state at every timestep, collocation provides fine-grained application of path constraints.

2.14.4. Direct Hermite-Simpson Collocation

We can analogously write down the NLP problem from Hermite-Simpson collocation:

$$
\begin{aligned}
\text{decision variables} \quad & \hat{\boldsymbol{x}}_0, \hat{\boldsymbol{x}}_{0.5}, \hat{\boldsymbol{x}}_1, \hat{\boldsymbol{x}}_{1.5}, \hat{\boldsymbol{x}}_2, \ldots, \hat{\boldsymbol{x}}_{N-0.5}, \hat{\boldsymbol{x}}_N, \hat{\boldsymbol{u}}_0, \hat{\boldsymbol{u}}_{0.5}, \hat{\boldsymbol{u}}_1, \hat{\boldsymbol{u}}_{1.5}, \hat{\boldsymbol{u}}_2, \ldots, \hat{\boldsymbol{u}}_{N-0.5}, \hat{\boldsymbol{u}}_N \\
\text{objective} \quad & \sum_{i=0}^{N-1} {}_{HS}C_{t_i}^{t_{i+1}}(c, \hat{\boldsymbol{x}}_i, \hat{\boldsymbol{x}}_{i+0.5}, \hat{\boldsymbol{x}}_{i+1}, \hat{\boldsymbol{u}}_i, \hat{\boldsymbol{u}}_{i+0.5}, \hat{\boldsymbol{u}}_{i+1}) \\
\text{equality constraints} \quad & \hat{\boldsymbol{x}}_1 = \hat{\boldsymbol{x}}_0 + {}_{HS}C_{t_0}^{t_1}(f, \hat{\boldsymbol{x}}_0, \hat{\boldsymbol{x}}_{0.5}, \hat{\boldsymbol{x}}_1, \hat{\boldsymbol{u}}_0, \hat{\boldsymbol{u}}_{0.5}, \hat{\boldsymbol{u}}_1) \\
& \hat{\boldsymbol{x}}_2 = \hat{\boldsymbol{x}}_1 + {}_{HS}C_{t_1}^{t_2}(f, \hat{\boldsymbol{x}}_1, \hat{\boldsymbol{x}}_{1.5}, \hat{\boldsymbol{x}}_2, \hat{\boldsymbol{u}}_1, \hat{\boldsymbol{u}}_{1.5}, \hat{\boldsymbol{u}}_2) \\
& \hat{\boldsymbol{x}}_3 = \hat{\boldsymbol{x}}_2 + {}_{HS}C_{t_2}^{t_3}(f, \hat{\boldsymbol{x}}_2, \hat{\boldsymbol{x}}_{2.5}, \hat{\boldsymbol{x}}_3, \hat{\boldsymbol{u}}_2, \hat{\boldsymbol{u}}_{2.5}, \hat{\boldsymbol{u}}_3) \\
& \quad \vdots \\
& \hat{\boldsymbol{x}}_N = \hat{\boldsymbol{x}}_{N-1} + {}_{HS}C_{t_{N-1}}^{t_N}(f, \hat{\boldsymbol{x}}_{N-1}, \hat{\boldsymbol{x}}_{N-0.5}, \hat{\boldsymbol{x}}_N, \hat{\boldsymbol{u}}_{N-1}, \hat{\boldsymbol{u}}_{N-0.5}, \hat{\boldsymbol{u}}_N) \\
& \hat{\boldsymbol{x}}_{0.5} = \frac{1}{2}(\hat{\boldsymbol{x}}_0 + \hat{\boldsymbol{x}}_1) + \frac{h}{8}(f(\hat{\boldsymbol{x}}_0, \hat{\boldsymbol{u}}_0, t_0) - f(\hat{\boldsymbol{x}}_1, \hat{\boldsymbol{u}}_1, t_1)) \\
& \hat{\boldsymbol{x}}_{1.5} = \frac{1}{2}(\hat{\boldsymbol{x}}_1 + \hat{\boldsymbol{x}}_2) + \frac{h}{8}(f(\hat{\boldsymbol{x}}_1, \hat{\boldsymbol{u}}_1, t_1) - f(\hat{\boldsymbol{x}}_2, \hat{\boldsymbol{u}}_2, t_2)) \\
& \quad \vdots \\
& \hat{\boldsymbol{x}}_{N-0.5} = \frac{1}{2}(\hat{\boldsymbol{x}}_{N-1} + \hat{\boldsymbol{x}}_N) + \frac{h}{8}(f(\hat{\boldsymbol{x}}_{N-1}, \hat{\boldsymbol{u}}_{N-1}, t_{N-1}) - f(\hat{\boldsymbol{x}}_N, \hat{\boldsymbol{u}}_N, t_N)) \\
& \hat{\boldsymbol{x}}_0 = \boldsymbol{x}_s \\
* \quad & \hat{\boldsymbol{x}}_N = \boldsymbol{x}_f \\
\text{*inequality constraints} \quad & \boldsymbol{x}_0^{\text{lower}} \leq \hat{\boldsymbol{x}}_0 \leq \boldsymbol{x}_0^{\text{upper}} \\
* \quad & \boldsymbol{x}_{0.5}^{\text{lower}} \leq \hat{\boldsymbol{x}}_{0.5} \leq \boldsymbol{x}_{0.5}^{\text{upper}} \\
* \quad & \boldsymbol{x}_1^{\text{lower}} \leq \hat{\boldsymbol{x}}_1 \leq \boldsymbol{x}_1^{\text{upper}} \\
* \quad & \boldsymbol{x}_{1.5}^{\text{lower}} \leq \hat{\boldsymbol{x}}_{1.5} \leq \boldsymbol{x}_{1.5}^{\text{upper}} \\
* \quad & \vdots \\
* \quad & \boldsymbol{x}_N^{\text{lower}} \leq \hat{\boldsymbol{x}}_N \leq \boldsymbol{x}_N^{\text{upper}} \\
* \quad & \boldsymbol{u}_0^{\text{lower}} \leq \hat{\boldsymbol{u}}_0 \leq \boldsymbol{u}_0^{\text{upper}} \\
* \quad & \boldsymbol{u}_{0.5}^{\text{lower}} \leq \hat{\boldsymbol{u}}_{0.5} \leq \boldsymbol{u}_{0.5}^{\text{upper}} \\
* \quad & \boldsymbol{u}_1^{\text{lower}} \leq \hat{\boldsymbol{u}}_1 \leq \boldsymbol{u}_1^{\text{upper}} \\
* \quad & \boldsymbol{u}_{1.5}^{\text{lower}} \leq \hat{\boldsymbol{u}}_{1.5} \leq \boldsymbol{u}_{1.5}^{\text{upper}}
\end{aligned}
$$

- $\vdots$
- $\boldsymbol{u}_N^{\text{lower}} \leq \hat{\boldsymbol{u}}_N \leq \boldsymbol{u}_N^{\text{upper}}$

Because the timestep midpoints also have decision variables in this setting, here one can enforce constraints with even greater precision.

2.15. Indirect Methods

The other class of methods used to solve trajectory optimization problems is that of the *indirect* methods. We just saw how direct methods solve the optimal control problem by first discretizing, and then transforming the discretized problem into an NLP. Indirect methods first derive the adjoint equation and construct a BVP, and only then choose a discretization and solve the problem. The distinction between direct and indirect is often summarized as "discretize then optimize" (direct) vs "optimize then discretize" (indirect).

The first step of an indirect method is to derive an adjoint equation which describes the dynamics of the dual variables $\boldsymbol{\lambda}$. To simplify the presentation, let us consider a trajectory optimization problem without final state requirement or bounds. Then, our problem is described by

$$\begin{aligned} \min_{\boldsymbol{u}(t)\ \forall t \in [t_s, t_f]} \quad & \int_{t_s}^{t_f} c(\boldsymbol{x}(t), \boldsymbol{u}(t), t)\ dt \\ \text{such that} \quad & \dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \\ \text{with} \quad & \boldsymbol{x}(t_s) = \boldsymbol{x}_s. \end{aligned}$$

We define the Hamiltonian for the problem as

$$H(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{\lambda}(t), t) = c(\boldsymbol{x}(t), \boldsymbol{u}(t), t) + \boldsymbol{\lambda}(t)^\top f(\boldsymbol{x}(t), \boldsymbol{u}(t), t).$$

By introducing an auxiliary function and massaging the Hamiltonian, we can find a system of equations which must necessarily be satisfied by any optimal solution $\boldsymbol{u}^\star$, for all $t \in [t_s, t_f]$ [**Lenhart and Workman, 2007**]:

$$\dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \tag{2.15.1}$$

$$\begin{aligned}
\boldsymbol{x}(t_s) &= \boldsymbol{x}_s \\
\dot{\boldsymbol{\lambda}}(t) &\equiv g(\boldsymbol{\lambda}(t), (\boldsymbol{x}(t), \boldsymbol{u}(t)), t) \\
&= -(D_1\ H)(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{\lambda}(t), t) \\
&= -\Big((D_1\ c)(\boldsymbol{x}(t), \boldsymbol{u}(t), t) + \boldsymbol{\lambda}(t)^\top (D_1\ f)(\boldsymbol{x}(t), \boldsymbol{u}(t), t)\Big) \qquad (2.15.2) \\
\dot{\boldsymbol{\lambda}}(t_f) &= \mathbf{0} \\
0 &= (D_2\ H)(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{\lambda}(t), t) \\
&= (D_2\ c)(\boldsymbol{x}(t), \boldsymbol{u}(t), t) + \boldsymbol{\lambda}(t)^\top (D_2\ f)(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \qquad (2.15.3)
\end{aligned}$$

We can use Eq. (2.15.3) to find an expression for $\boldsymbol{u}$ in terms of $\boldsymbol{x}$ and $\boldsymbol{\lambda}$. In turn, this expression can be substituted into Eq. (2.15.1) and Eq. (2.15.2), thus forming a BVP with state $\begin{bmatrix}\boldsymbol{x} & \dot{\boldsymbol{x}} & \boldsymbol{\lambda} & \dot{\boldsymbol{\lambda}}\end{bmatrix}^\top$, and boundary conditions on $\boldsymbol{x}(t_s)$ and $\dot{\boldsymbol{\lambda}}(t_f)$. The BVP can then be solved with the methods discussed in Section 2.12.

2.15.1. Forward Backward Sweep Method

Instead of constructing a BVP explicitly and using a BVP solver, it is also possible to apply an iterative method known as the *forward-backward sweep method* (FBSM). FBSM was presented in [**Lenhart and Workman, 2007**], and is similar to an earlier successive approximation method described in [**Mitter, 1966**]. We present FBSM in Algorithm 2.

Algorithm 2 Forward-Backward Sweep Method

Initialize $\hat{\boldsymbol{u}}_{0:N}$ randomly
while $\hat{\boldsymbol{u}}_{0:N}$ not converged **do**
    $\hat{\boldsymbol{x}}_{0:N} \leftarrow \boldsymbol{x}_s + {}_hS^{t_f}_{t_s}(f, \boldsymbol{x}_s, \hat{\boldsymbol{u}}_{0:N})$ ▷ integrate (2.15.1) forwards in time
    $\hat{\boldsymbol{\lambda}}_{0:N} \leftarrow \mathbf{0} + {}_hS^{t_s}_{t_f}(g, \hat{\boldsymbol{\lambda}}_{0:N}, (\hat{\boldsymbol{x}}_{0:N}, \hat{\boldsymbol{u}}_{0:N}))$ ▷ integrate (2.15.2) backwards in time
    $\hat{\boldsymbol{u}}_{0:N} \leftarrow$ solve $(D_2\ H)(\hat{\boldsymbol{x}}_{0:N}, \bullet, \hat{\boldsymbol{\lambda}}_{0:N}, t) = 0$ ▷ solve (2.15.3) for controls
end while

2.16. MLE Model Learning

So far, we have seen how to solve the optimal control problem using trajectory optimization. In particular, the approaches discussed so far all require access to a model of the system dynamics, f. In many settings, we might not know f, but would still like to perform optimal control. Thus, we must first collect data and build a model to approximate f.

A common approach to learning f is to write down a model with the parametric form that we expect the dynamics to follow. We collect trajectories $\boldsymbol{x}_{0:N}$ through state space by

applying controls $\boldsymbol{u}_{0:N}$ over time. We then use these trajectories to fit the parameters of our model in a way which maximizes the likelihood of the trajectories. We refer to this technique as the "maximum likelihood estimation" (MLE) approach.

To simplify the discussion, let us imagine that the true dynamics depend only on the current state and time (no controls), and are parametrized by some parameters $\boldsymbol{\theta}$. In other words, our system evolves as $\dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t), t, \boldsymbol{\theta})$, and we would like to estimate $\boldsymbol{\theta}$ from data. Suppose we have a dataset of trajectories, which were collected by starting in start states $\boldsymbol{x}_0^1, \boldsymbol{x}_0^2, \ldots, \boldsymbol{x}_0^M$ and recording the states at each timestep[8] h as the dynamical system evolves over time:

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}^1 \\ \boldsymbol{x}^2 \\ \vdots \\ \boldsymbol{x}^M \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_0^1 & \boldsymbol{x}_1^1 & \boldsymbol{x}_2^1 & \ldots & \boldsymbol{x}_N^1 \\ \boldsymbol{x}_0^2 & \boldsymbol{x}_1^2 & \boldsymbol{x}_2^2 & \ldots & \boldsymbol{x}_N^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{x}_0^M & \boldsymbol{x}_1^M & \boldsymbol{x}_2^M & \ldots & \boldsymbol{x}_N^M \end{bmatrix}$$

Note that in the general case, X is a tensor of shape (M, N, D), where M is the number of trajectories, N is the number of timesteps, and D is the dimension of the state.

To learn an estimate $\hat{\boldsymbol{\theta}}$ of $\boldsymbol{\theta}$, our approach will be to integrate our model, parametrized by $\hat{\boldsymbol{\theta}}$, forward through time, recording the estimated state at each timestep. We will then compare these trajectories with the true trajectories in our dataset.

How do we integrate our our model forward through time? To see this, let us consider the first training trajectory, $\boldsymbol{x} = \boldsymbol{X}^1$. The starting state of this trajectory is $\boldsymbol{x}_0$. To use this trajectory to help improve our model, we must first calculate our model's prediction of the trajectory:

$$\tilde{\boldsymbol{x}} = {}_hT_{t_0}^{t_N} = \left[\boldsymbol{x}_0 \quad {}_hS_{t_0}^{t_1}(f, \boldsymbol{x}_0, \hat{\boldsymbol{\theta}}) \quad {}_hS_{t_0}^{t_2}(f, \boldsymbol{x}_0, \hat{\boldsymbol{\theta}}) \quad \ldots \quad {}_hS_{t_0}^{t_N}(f, \boldsymbol{x}_0, \hat{\boldsymbol{\theta}})\right].$$

Here we use ${}_hT$ to represent numerical integration where we record the state at each timestep (as opposed to ${}_hS$, where we only recorded the final state). On a practical note, it is not necessary to re-start the integration $N-1$ times, since our numerical integration methods always start from the previous state estimate.

We can then calculate the mean squared error between the predicted trajectory and true trajectory. We illustrate this with an example. Say we have a model with dynamics

$$f(x, t) = \sin(at) - bx,$$

[8]In general it is not a requirement that that we have a measurement for every timestep, or even that the time intervals at which we record be evenly spaced. We make these assumptions here in order to simplify the presentation.

and the true parameters are $a = 0.14$ and $b = 0.1$. Imagine our current guess for the parameters is $a = 0.27$ and $b = 0.25$. Then the trajectory we predict with the parameter guess will be different from the true trajectory, and we can calculate the loss based on this difference. A picture of this example is shown in Figure 2.10.

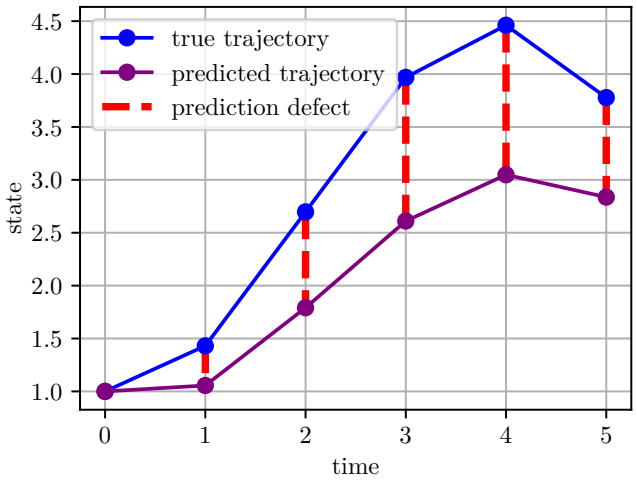


Fig. 2.10. A example of two trajectories of five timesteps of a system with dynamics $f(x,t) = \sin(at) - bx$, with both the true parameters $a = 0.14, b = 0.1$, and a parameter guess $a = 0.27, b = 0.25$. The defect between the true trajectory and the predicted one is measured at each timestep. To calculate the loss, we square and average the red segments.

In practice, we usually want to learn using many different trajectories simultaneously. The step of numerical integration can be performed in parallel over all the trajectories:

$$\begin{aligned}
\tilde{\boldsymbol{X}} &= {}_hT_{t_0}^{t_N}(f, \boldsymbol{X}_0^{1:M}) \\
&= \begin{bmatrix} \boldsymbol{X}_0^{1:M} & {}_hS_{t_0}^{t_1}(f, \boldsymbol{X}_0^{1:M}, \hat{\boldsymbol{\theta}}) & {}_hS_{t_0}^{t_2}(f, \boldsymbol{X}_0^{1:M}, \hat{\boldsymbol{\theta}}) & \dots & {}_hS_{t_0}^{t_N}(f, \boldsymbol{X}_0^{1:M}, \hat{\boldsymbol{\theta}}) \end{bmatrix} \\
&= \begin{bmatrix} \boldsymbol{X}_0^{1} & {}_hS_{t_0}^{t_1}(f, \boldsymbol{X}_0^{1}, \hat{\boldsymbol{\theta}}) & {}_hS_{t_0}^{t_2}(f, \boldsymbol{X}_0^{1}, \hat{\boldsymbol{\theta}}) & \dots & {}_hS_{t_0}^{t_N}(f, \boldsymbol{X}_0^{1}, \hat{\boldsymbol{\theta}}) \\ \boldsymbol{X}_0^{2} & {}_hS_{t_0}^{t_1}(f, \boldsymbol{X}_0^{2}, \hat{\boldsymbol{\theta}}) & {}_hS_{t_0}^{t_2}(f, \boldsymbol{X}_0^{2}, \hat{\boldsymbol{\theta}}) & \dots & {}_hS_{t_0}^{t_N}(f, \boldsymbol{X}_0^{2}, \hat{\boldsymbol{\theta}}) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{X}_0^{M} & {}_hS_{t_0}^{t_1}(f, \boldsymbol{X}_0^{M}, \hat{\boldsymbol{\theta}}) & {}_hS_{t_0}^{t_2}(f, \boldsymbol{X}_0^{M}, \hat{\boldsymbol{\theta}}) & \dots & {}_hS_{t_0}^{t_N}(f, \boldsymbol{X}_0^{M}, \hat{\boldsymbol{\theta}}) \end{bmatrix}
\end{aligned}$$

We also calculate the mean squared error in parallel:

$$L(\hat{\boldsymbol{\theta}}) = \frac{1}{MND} \sum_{\text{batch dim.}} \sum_{\text{timestep}} \sum_{\text{state dim.}} \left(\tilde{\boldsymbol{X}} - \boldsymbol{X}\right) \odot \left(\tilde{\boldsymbol{X}} - \boldsymbol{X}\right).$$

In order to update our parameter guess, we use a gradient-based method on the loss function L. Its gradient can easily be calculated by AD through the integration procedure. We would like to converge on the optimal value of $\hat{\boldsymbol{\theta}}$:

$$\boldsymbol{\theta}^\star = \arg\min_{\hat{\boldsymbol{\theta}}} L(\hat{\boldsymbol{\theta}}).$$

Because of the unconstrained nature of the optimization problem, we can use a stochastic gradient-based method with mini-batching. This will scale better than full-batch methods as we increase dataset size[9]. In this case, we would choose a minibatch $b \subseteq \{1, 2, \ldots, M\}$ and then approximate the loss as

$$L(\hat{\boldsymbol{\theta}}) \approx \frac{1}{|b|ND} \sum_{\text{batch}} \sum_{\text{dim.}} \sum_{\text{timestep}} \sum_{\text{state dim.}} \left({}_hT_{t_s}^{t_f}(f, \boldsymbol{X}_0^b, \hat{\boldsymbol{\theta}}) - \boldsymbol{X}^b\right) \odot \left({}_hT_{t_s}^{t_f}(f, \boldsymbol{X}_0^b, \hat{\boldsymbol{\theta}}) - \boldsymbol{X}^b\right),$$

where $\boldsymbol{X}^b$ selects the rows of $\boldsymbol{X}$ with indices in b. Note that when $|b| = M$, we are back in the full batch setting.

Finally, we note that if we have choice in how the training trajectories are selected, we will want to try a range of different start states, in order to gather as much information as possible about the environment dynamics.

2.16.1. Including Controls

In many cases, we want to learn the dynamics of a system in which we can also apply controls. Then the dynamics are of the form $f(\boldsymbol{x}(t), \boldsymbol{u}(t), t, \boldsymbol{\theta})$, and we want to learn $\hat{\boldsymbol{\theta}}$ which minimizes

$$L(\hat{\boldsymbol{\theta}}) = \frac{1}{MND} \sum_{\text{b.d.}} \sum_{\text{t.s.}} \sum_{\text{s.d.}} \left({}_hT_{t_s}^{t_f}(f, \boldsymbol{X}_0^{1:M}, \boldsymbol{U}, \hat{\boldsymbol{\theta}}) - \boldsymbol{X}\right) \odot \left({}_hT_{t_s}^{t_f}(f, \boldsymbol{X}_0^{1:M}, \boldsymbol{U}, \hat{\boldsymbol{\theta}}) - \boldsymbol{X}\right),$$

where $\boldsymbol{U} \equiv \boldsymbol{U}_{0:N}^{1:M}$ is a (M, N, E) tensor of controls, where E is the dimension of the control. Analogously to the previous section, we can calculate the loss over minibatches by choosing only a subset of the training set at each iteration.

2.16.2. Exploration Strategy

In Section 2.16, we mentioned how it is beneficial to consider trajectories which start from different states, since it will provide information about more parts of the state space. Now that we are in a setting with controls, we can also choose which controls to apply over time when gathering the training trajectories. The problem of choosing controls in order to provide the most useful information about system parameters is an area of study known as Optimum Experimental Design [**Bürger and Lago Garcia, 2014**]. We now list several simple approaches for choosing controls.

[9]In practice, when learning coefficients $\hat{\boldsymbol{\theta}}$ of a parametric model, scaling with data is not of great concern, since we are usually able to learn an accurate guess of $\boldsymbol{\theta}$ with only a small number of training trajectories. When $\hat{\boldsymbol{\theta}}$ parametrizes a more expressive model like a Neural ODE, then data scaling is an important consideration.

Uniform Random: The simplest approach is to sample the controls from the uniform distribution between the lower and upper bounds:

$$\boldsymbol{U}_i^j \sim \mathcal{U}\left[\boldsymbol{u}_{0:N}^{\text{lower}}, \boldsymbol{u}_{0:N}^{\text{upper}}\right] \quad \text{for } i = 0, \ldots, N \text{ and } j = 1, \ldots, M.$$

Each control trajectory can then be integrated over time starting from a (possibly different) start state to generate the corresponding state trajectories.

Random Walk: Another approach is to sample the controls from a Gaussian random walk: first, sample the start state uniformly at random. Then, step forward through time, at each point sampling controls from a Gaussian centered at the previous control. This will lead to controls which are correlated in time. This might be a desirable feature, since it is more likely to explore more regions of state space, compared with uniform random controls which might oscillate rapidly and cancel each other out. This process is summarized in Algorithm 3.

Algorithm 3 Random Walk Trajectory Generation

Initialize s ▷ choose spread hyperparameter
for $j = 1, \ldots, M$ **do**
  $\boldsymbol{U}_0^j \sim \mathcal{U}\left[\boldsymbol{u}_0^{\text{lower}}, \boldsymbol{u}_0^{\text{upper}}\right]$
  for $i = 1, \ldots, N$ **do**
    $\boldsymbol{U}_i^j \sim \text{clip}\left(\mathcal{N}[\boldsymbol{U}_{i-1}^j, (\boldsymbol{u}_i^{\text{upper}} - \boldsymbol{u}_i^{\text{lower}}) \cdot s], \boldsymbol{u}_i^{\text{lower}}, \boldsymbol{u}_i^{\text{upper}}\right)$ ▷ sample then clip
  end for
end for

We can change s depending on how large a deviation from the current control we want to allow at each step. Note that this operation can be vectorized across batch dimension j, but is inherently sequential across timestep dimension i. As in the uniform case, we then use the control trajectories to generate state trajectories.

Best Guess Approach: If we know that the model we are learning will be used for solving an optimal control problem, it might be particularly important for the model to be accurate near the states that are visited when applying the true optimal controls. To ensure this, we might want to observe the dynamics around the optimal trajectory, by applying controls that are close to the optimal controls.

Of course, we usually don't know what those controls are a priori, so we can instead use a bootstrapping approach. After training a model on a dataset generated using one of the previously mentioned techniques, we can then perform trajectory optimization on our learned model, to give us a guess at the optimal trajectory. For future data gathering, we can then sample around the current guess of the optimal trajectory. If all goes well, as our model becomes more refined around the relevant parts of state space, the planned controls will be increasingly accurate.

This loop of alternating data gathering and planning is presented in Algorithm 4. Remember that $\boldsymbol{X}$ has shape (M, N, D), $\boldsymbol{U}$ and $\boldsymbol{U}^\star$ have shape (M, N, E), and $\boldsymbol{X}_0$ has shape (M, D). Over the course of the algorithm, we will choose new controls and corresponding state trajectories, thus augmenting $\boldsymbol{X}$ and $\boldsymbol{U}$ along the first dimension. We use f_{true} to denote the true dynamics.

Algorithm 4 Best-guess updates

Initialize $\boldsymbol{U}$, $\boldsymbol{U}^\star$, $\boldsymbol{X}_0$, $\hat{\boldsymbol{\theta}}$ with random values
while $\boldsymbol{U}^\star$ not converged **do**
    $\boldsymbol{X} \leftarrow {}_hT_{t_s}^{t_f}(f_{\text{true}}, \boldsymbol{X}_0, \boldsymbol{U})$
    while $\hat{\boldsymbol{\theta}}$ not converged **do**
        $\hat{\boldsymbol{X}} \leftarrow {}_hT_{t_s}^{t_f}(f, \boldsymbol{X}_0, \boldsymbol{U}, \hat{\boldsymbol{\theta}})$
        $\hat{\boldsymbol{\theta}} \leftarrow$ update using $(D\ L)(\hat{\boldsymbol{\theta}})$ ▷ L given in 2.16.1
    end while
    $\boldsymbol{U}^\star \leftarrow$ solve OCP with model f parametrized by $\hat{\boldsymbol{\theta}}$
    $\boldsymbol{U}' \sim \text{clip}\Big(\mathcal{N}\left[\boldsymbol{U}^\star, (\boldsymbol{U}^{\text{upper}} - \boldsymbol{U}^{\text{lower}}) \cdot s\right], \boldsymbol{U}^{\text{lower}}, \boldsymbol{U}^{\text{upper}}\Big)$ ▷ sample then clip
    $\boldsymbol{U} \leftarrow \text{concatenate}\,((\boldsymbol{U}, \boldsymbol{U}'), \text{axis} = 0)$
end while

Note that in order to avoid focusing only on the state space around the true optimal trajectory, it is important to keep the original, random dataset in the training set. It would likely also be a good idea to have a proportion of the new controls be sampled uniformly at random. Ensuring that the hyperparmeter s is large enough will also help combat this issue.

2.16.3. Control Smoothing

When choosing which controls to apply for trajectory generation, we also have the option to applying smoothing to the control trajectory. This might be desirable since in most cases, the true optimal controls change gradually over timesteps, while the stochastically generated controls will often lead to quick changes in the controls. In practice, convolution with a small Gaussian filter is enough to remove most of this unwanted behaviour.

2.17. End-to-End Model Learning

In the previous section, data gathering and model training happened largely without consideration of the next step of using the learned model to plan optimal behaviour. What we call the "best guess approach" – where new training data is generated by sampling around the current best guess of controls – is a first step in the direction of choosing which data to generate based on some measure how useful we expect it to be for the model learning. In the best guess approach, the coupling between model building and planning with trajectory

optimization is still quite weak: in particular, the performance of the model in solving the optimal control problem is not used to help select trajectories or update the model parameters.

We will now integrate the data gathering, model learning and planning into one end-to-end-trainable algorithm. We first construct the Lagrangian of the NLP, where we let $\boldsymbol{x}$ be the primal variables (control and state decision variables), $\boldsymbol{\lambda}$ the dual variables (Lagrange multipliers), f, the objective function, and h, the equality constraints (for simplicity, we temporarily ignore inequality constraints). We note that both the objective and constraints functions use the learned dynamics, and thus are parametrized by $\boldsymbol{\theta}$. In order to simplify notation, we let $\boldsymbol{y} = [\boldsymbol{x}, \boldsymbol{\lambda}]^\top$ represent all the decision variables. Then we can write our Lagrangian as

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{y}) = f(\boldsymbol{x}, \boldsymbol{\theta}) + \boldsymbol{\lambda}^\top h(\boldsymbol{x}, \boldsymbol{\theta}).$$

Let υ be the function representing the trajectory optimization solver, taking the parameter values $\hat{\boldsymbol{\theta}}$ as input and giving decision variables $\boldsymbol{y}$ as output. Let J denote the outer objective function, that is, the performance of our controls when acting on the true dynamics. J takes the decision variables as input and gives a real output (in practice, J only depends on the controls $\boldsymbol{u}_{0:N}$, which are a subset of the decision variables).

We would like to propagate the gradient of J with respect to $\hat{\boldsymbol{\theta}}$ through the NLP solve υ, to update our parameter values based on the performance of the controls we solved for.

2.17.1. Basic Algorithm

The basic structure of this end-to-end training is shown in Algorithm 5. The natural

Algorithm 5 Basic End-to-End Idea

1: Initialize $\hat{\boldsymbol{u}}_{0:N}$, $\hat{\boldsymbol{\theta}}$ with random values
2: **while** $\hat{\boldsymbol{u}}_{0:N}$, $\hat{\boldsymbol{\theta}}$ not converged **do**
3: $\quad \hat{\boldsymbol{y}} \leftarrow \upsilon(\hat{\boldsymbol{\theta}})$ ▷ solve NLP to get decision variables
4: $\quad \hat{\boldsymbol{u}}_{0:N}, \_, \_ \leftarrow \hat{\boldsymbol{y}}$ ▷ extract controls
5: $\quad \hat{\boldsymbol{\theta}} \leftarrow$ update using $(D\,(J \circ \upsilon))(\hat{\boldsymbol{\theta}})$
6: **end while**

challenge of this algorithm is the implementation of Line 5 – how can we calculate this gradient? We will now work towards a technique to do just that.

2.17.2. Implementation Theory

By the chain rule we have that

$$(D\,(J \circ \upsilon))(\boldsymbol{\theta}) = (D\,J)(\upsilon(\boldsymbol{\theta})) \cdot (D\,\upsilon)(\boldsymbol{\theta}).$$

The first term, $(D\ J)(\upsilon(\boldsymbol{\theta}))$, can simply be calculated using automatic differentiation if we know the functional form of J. This will be the case when using, for example, an imitation loss to evaluate the controls. If instead we are in more of an RL setting where we have call access to J but don't know its derivative, then we must use a gradient approximation method like REINFORCE.

The calculation of $(D\ \upsilon)(\boldsymbol{\theta})$ is potentially more challenging, since it involves taking a derivative through the NLP solver. Unfortunately, most NLP solvers are not differentiable. Indeed, even if we were to use a differentiable NLP solver, it might take hundreds or thousands of iterations to converge – attempting to use AD through that many iterations might be prohibitively time-consuming. As such, we would like to find a different approach to calculate $(D\ \upsilon)(\boldsymbol{\theta})$.

A natural answer to this challenge is to apply the implicit function theorem (IFT). The IFT says that, given a C^1 function $F : \mathbb{R}^A \times \mathbb{R}^B \to \mathbb{R}$, if we are at a fixed point $F(\boldsymbol{a}, \boldsymbol{b}) = 0$, and if the Jacobian $(D_2\ F)(\boldsymbol{a}, \boldsymbol{b})$ is invertible, then there exists a function $g : \mathbb{R}^A \to \mathbb{R}$, such that for points $(\bar{\boldsymbol{a}}, \bar{\boldsymbol{b}})$ near $(\boldsymbol{a}, \boldsymbol{b})$, we have that $F(\bar{\boldsymbol{a}}, g(\bar{\boldsymbol{b}})) = 0$. Moreover, $(D\ g)(\boldsymbol{a})$ is exactly

$$(D\ g)(\boldsymbol{a}) = -\left(D_2\ F\right)^{-1}(\boldsymbol{a}, \boldsymbol{b}) \cdot \left(D_1\ F\right)(\boldsymbol{a}, \boldsymbol{b}).$$

In order to use the IFT in our problem setting, we must identify our function F, and its value at the root where we wish to apply it. For the function F, we use the gradient of the Lagrangian with respect to the parameters, $(D_1\ \mathcal{L})$. Supposing the NLP solver converged, it is clear that its output decision variables and the current parameters will be a zero of $(D_1\ \mathcal{L})$, since otherwise, we could have made a change to the decision variables to improve performance. Thus, applied to our problem, the IFT says there is a g such that for $(\bar{\boldsymbol{\theta}}, \bar{\boldsymbol{y}})$ close to the zero, we have, $F(\bar{\boldsymbol{\theta}}, g(\bar{\boldsymbol{\theta}})) = 0$, and, more importantly, that

$$(D\ g)(\boldsymbol{\theta}, \boldsymbol{y}) = -\left(D_2 D_1\ \mathcal{L}\right)^{-1}(\boldsymbol{\theta}, \boldsymbol{y}) \cdot \left(D_1^2\ \mathcal{L}\right)(\boldsymbol{\theta}, \boldsymbol{y}).$$

In fact, this quantity is exactly the derivative we were looking for, since g represents the local change in $\boldsymbol{y}$ that we can expect from a small change in $\boldsymbol{\theta}$:

$$(D\ g)(\boldsymbol{\theta}) = (D\ \upsilon)(\boldsymbol{\theta}).$$

With this, we have reduced our problem to one of evaluating derivatives of the Lagrangian at the solution point given by our NLP solver. Fortunately, we can use automatic differentiation to obtain the value of this derivative. The resulting algorithm is shown in Algorithm 6.

In practice, the IFT-based approach suffers from at least three drawbacks. First of all, it requires the NLP solver to return the optimal values of the dual variables (a feature implemented by only some solvers), thus limiting the number of off-the-shelf solvers we can use. Second, in order to work, it requires the solver to have found a root of the derivative of

Algorithm 6 End-to-end learning with IFT

Initialize $\hat{\boldsymbol{u}}_{0:N}, \hat{\boldsymbol{\theta}}$ with random values
while $\hat{\boldsymbol{u}}_{0:N}, \hat{\boldsymbol{\theta}}$ not converged **do**
    $\hat{\boldsymbol{y}} \leftarrow \upsilon(\hat{\boldsymbol{\theta}})$
    $\hat{\boldsymbol{u}}_{0:N}, \_, \_ \leftarrow \hat{\boldsymbol{y}}$ ▷ extract controls
    $\texttt{dJ\_dy} \leftarrow (D\ J)(\hat{\boldsymbol{y}})$ ▷ using AD (or REINFORCE)
    $\texttt{dF\_dy} \leftarrow (D_2 D_1\ \mathcal{L})(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{y}})$
    $\texttt{dF\_dtheta} \leftarrow (D_1^2\ \mathcal{L})(\hat{\boldsymbol{\theta}}, \hat{\boldsymbol{y}})$
    $\texttt{dy\_dtheta} \leftarrow -\texttt{dF\_dy}^{-1} \cdot \texttt{dF\_dtheta}$ ▷ apply the IFT
    $\texttt{dJ\_dtheta} \leftarrow \texttt{dJ\_dy} \cdot \texttt{dy\_dtheta}$ ▷ apply the chain rule
    $\hat{\boldsymbol{\theta}} \leftarrow$ update with $\texttt{dJ\_dtheta}$
end while

the Lagrangian. Especially when the dynamics become complicated, there is no guarantee that the NLP solver will converge at all, let alone converge within its maximum allowed iterations. Finally, even when the NLP solver does converge, for anything but the most basic problems, it can take a long time to do so. In our problem setting, we would like to perform hundreds of thousands of gradient descent steps to update our model parameters. Since we get the gradient by completing an NLP solve, anything but a very fast solve will make our algorithm prohibitively slow. With these issues in mind, we would like to devise a practical way to approximate the desired gradients.

2.17.3. Practical Implementation

The reasons we used the IFT were that the NLP solver might not be differentiable, and because even if it was, it would not be feasible to differentiate through the fully unrolled solve. We then saw that in practice, the IFT will be too slow if the NLP solver takes a long time.

One possible compromise is to forgo reaching a root during the NLP solve. If we take only a few steps of a differentiable NLP solver, then we can easily back-propagate through the unrolled loop[10]. While this will only get us an approximate solution at each iteration of the algorithm, if we warm-start each iteration at the previous guess, then we should make progress towards a solution over time. At the same time, in order to not forget the dynamics near the starting point, we must occasionally reset this warm-start to an initial guess.

This practical implementation is presented in Algorithm 7

[10] in practice, it is even faster to accumulate the Jacobian manually during the forward solve, avoiding the need to store temporary values at each iteration of the loop unroll

Algorithm 7 Practical implementation of end-to-end learning

Initialize $\hat{\boldsymbol{u}}_{0:N}, \hat{\boldsymbol{\theta}}$ with random values
while $\hat{\boldsymbol{u}}_{0:N}, \hat{\boldsymbol{\theta}}$ not converged **do**
    $\hat{\boldsymbol{y}}$, `dy_dtheta` $\leftarrow$ simultaneously take several steps of $\upsilon(\hat{\boldsymbol{\theta}})$ and accumulate gradients
    $\hat{\boldsymbol{u}}_{0:N}$, `_`, `_` $\leftarrow \hat{\boldsymbol{y}}$ ▷ extract controls
    `dJ_dy` $\leftarrow (D\ J)(\hat{\boldsymbol{y}})$ ▷ using AD (or REINFORCE)
    `dJ_dtheta` $\leftarrow$ `dJ_dy` $\cdot$ `dy_dtheta` ▷ apply the chain rule
    $\hat{\boldsymbol{\theta}} \leftarrow$ update with `dJ_dtheta`
end while

2.18. Neural ODEs

Traditionally, system identification is performed to learn the parameters of a structured model of which the parameters are unknown [**Keesman, 2011**]. For example, let us again consider the toy model from Section 2.16, with dynamics

$$f(x, t) = \sin(at) - bx,$$

and true parameters $a = 0.14$ and $b = 0.1$.

When performing system identification for this problem, we have access to the parametric form of the dynamics; we just don't know the correct values of a and b. In this thesis, we refer to this kind of model as a "parametric" model[11]. This kind of system identification is well-suited for applications where the dynamics of the systems in question are well-understood enough to write down a reasonable guess at the dynamics.

In most cases, however, we do not have a strong understanding of the dynamics of a system, and cannot even write out the parametric form of the dynamics, let alone use it to learn unknown parameters. In such cases, we would like to instead use a generic class of models which can learn to represent a great variety of functions from data alone. We can use a multilayer perceptron (MLP) for this task. In the case with no controls, the idea of using a neural network to predict dynamics is called Neural ODE, and was first presented in [**Chen et al., 2018**]. In the more general case, we also allow the dynamics to take controls. As such, the neural network will take as input not only the current state, but also the current control. The goal is then to find $\boldsymbol{\theta}$ such that[12]

$$f(\boldsymbol{x}(t), \boldsymbol{u}(t), t, \boldsymbol{\theta}) \equiv \texttt{apply\_net}\left(\boldsymbol{\theta}, \begin{bmatrix} \boldsymbol{x}(t) & \boldsymbol{u}(t) & t \end{bmatrix}^\top\right) \approx f(\boldsymbol{x}(t), \boldsymbol{u}(t), t),$$

where $f(\boldsymbol{x}(t), \boldsymbol{u}(t), t)$ is the true system dynamics.

[11] this is not to be confused with the notion of parametric and nonparametric machine learning models
[12] in practice, we only consider time-independent dynamics, so the network only takes $\boldsymbol{x}$ and $\boldsymbol{u}$ as input

Chapter 3

Myriad: Environments and Optimizers

The Myriad repository[1] represents a significant contribution of this thesis. It was created in the mindset of implementing a repository of different dynamical system environments, which one could interact with and optimize over using a standardized approach. At time of writing, it contains 18 distinct dynamical system environments:

- Bacteria growth
- Bear population management
- Bioreactor management
- Cancer treatment
- Cart-pole swing-up
- Epidemic response (SEIR)
- Glucose level regulation
- Harvest optimization
- HIV treatment
- Invasive plant control
- Mould management
- Mountain car
- Pendulum swing-up
- Pest control in predator-prey setting
- Rocket landing
- Timber investment and harvest
- Tumour treatment
- Forced van der Pol oscillator

Many of these systems are inspired by examples of modelling problems from [**Lenhart and Workman, 2007**].

[1] https://github.com/nikihowe/myriad

The repository also contains implementations of several trajectory optimization techniques:

- Direct single shooting
- Direct multiple shooting
- Direct trapezoidal collocation
- Direct Hermite-Simpson collocation
- Forward-Backward Sweep Method

These trajectory optimization techniques can be mixed and matched with the different environments. Additionally, the user has choice among several numerical integration techniques.

In addition to performing trajectory optimization, one can perform system identification on the environments, either by modelling the environment with a guess of the system parameters, or with a NeuralODE system, which can be integrated through time and differentiated through much like the other systems.

3.1. Extending Myriad

The Myriad repository was created with extensibility in mind. Indeed, it is straightforward to add new environments, new trajectory optimization techniques, and new integration methods to the repository.

3.1.1. Creating a New System

A control environment, or *system*, is characterized by several elements: a dynamics function, a cost function, a start state, and a final time. In addition to these required elements, a system can optionally also include an end state, a terminal cost function, and bounds on the state and controls.

In order to create a new system, one must extend `FiniteHorizonControlSystem`, an abstract class defined in `systems/base.py`. In particular, a dynamics function and cost function must be implemented, along with the class attributes of start state, time horizon, and control and state bounds.

If one would also like to use indirect methods to solve this system, one instead needs to extend the abstract class `IndirectFHCS` (itself a subclass of `FiniteHorizonControlSystem`), which also requires implementation of the adjoint dynamics equation, and the optimality characterization of controls in terms of state and dual variables.

If one would like to be able to perform system identification on the system, one must also implement the `parametrized_dynamics` and `parametrized_cost` functions, which also take parameter values.

3.1.2. Creating a New Trajectory Optimizer

A trajectory optimizer, or simply *optimizer*, can be direct or indirect. All trajectory optimizers have an objective function, a constraint function, control and state bounds, an initial decision variable guess, and an unravel function for sorting the decision variable array into states and controls.

To implement a direct trajectory optimizer, the user must extend the abstract class `TrajectoryOptimizer`, and implement the aforementioned methods and attributes. To implement an indirect trajectory optimizer, the abstract class `IndirectMethodOptimizer` should be extended instead. This has an additional required method, `solve`, which must provide a way to solve the resulting trajectory optimization problem. This is different from the direct optimizer class, which simply calls an NLP solver to perform the solve.

If the new optimizer will be used for system identification, the `parametrized_objective` and `parametrized_constraints` methods must also be implemented. At present, only direct trajectory optimizers support application in the system identification setting.

3.1.3. Creating a New Integration Method

The user can also implement other integration strategies, which are in the `integrate` (and related) functions in the `utils` module.

3.2. Examples

We now present two of the Myriad systems, which we will use as examples throughout the remainder of the thesis. These two systems were chosen due to their ease of visualization, interesting optimal control trajectories, and good performance across all experiments.

3.2.1. Cancer Treatment

This domain, which is presented in [**Lenhart and Workman, 2007**], is one in which we are planning the optimal schedule for administering an anti-cancer drug over a fixed time horizon. The parameters for the model are given in Table 3.1.

Quantity	Symbol	Value
Tumour growth rate	r	0.3
Size of dose	δ	0.45

Table 3.1. Parameters of the Cancer Treatment system.

The state x is the normalized density of the tumour; the control u is the strength of the drug.

The dynamics are

$$f(x,u) = r \cdot x \log \frac{1}{x} - u\delta x.$$

The cost is

$$c(x,u) = a \cdot x^2 + u^2,$$

where a is a parameter encoding the tradeoff between tumour size and drug strength (and thus severity of side effects), by default set to $a = 3$. There is no terminal cost function.

We set the state and control bounds to

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} \preceq \begin{bmatrix} x \\ u \end{bmatrix} \preceq \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

We start the system in $x_0 = 0.975$, with no final state requirement. We perform optimization over a horizon of 20 days. Figure 3.1 shows the resulting optimal trajectory.

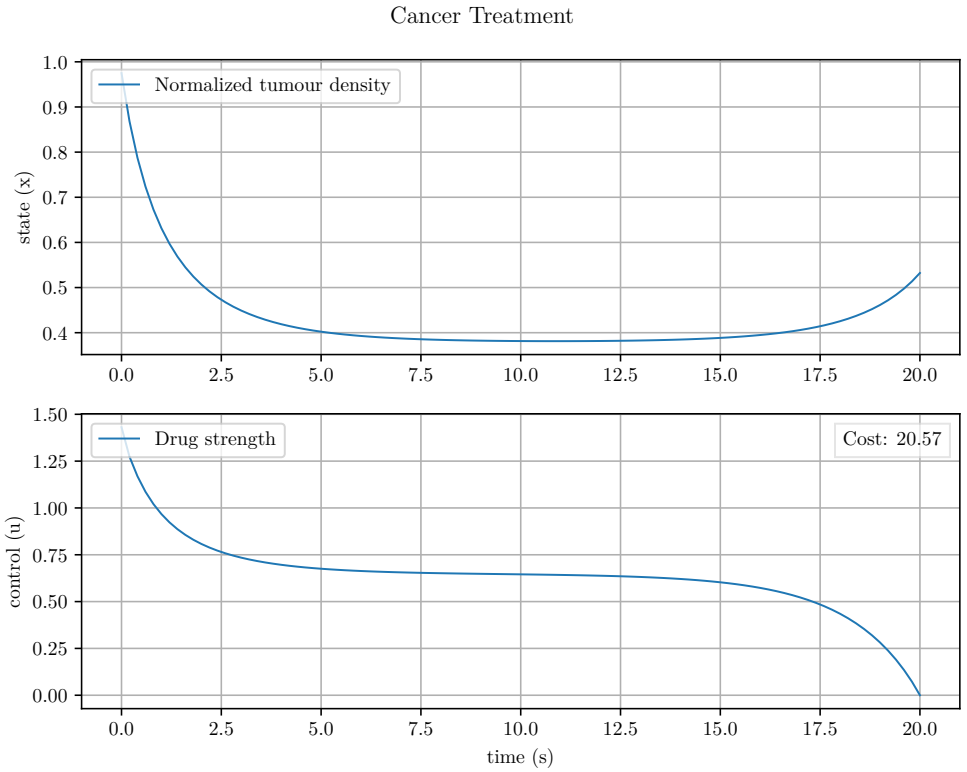


Fig. 3.1. An optimal solution to the Cancer Treatment problem.

3.2.2. Mould Fungicide

This domain, which is also presented in [**Lenhart and Workman, 2007**], is one in which we are trying to reduce the size of a mould population by application of a fungicide. The model parameters are given in Table 3.2.

Quantity	Symbol	Value
Growth Rate	r	0.3
Carrying Capacity	M	10

Table 3.2. Parameters of the Mould Fungicide system.

The state x is the current concentration of mould; the control u is the amount of fungicide we apply to the mould.

The dynamics are

$$f(x, u) = r(M - x) - ux.$$

The cost is

$$c(x, u) = Ax^2 + u^2,$$

where A is a parameter encoding the tradeoff between mould concentration and cost of the fungicide.

We set the state and control bounds to

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} \preceq \begin{bmatrix} x \\ u \end{bmatrix} \preceq \begin{bmatrix} 5 \\ 5 \end{bmatrix}.$$

We start the system in $\boldsymbol{x}_s = 1$, and there is no final state requirement or terminal cost function. We perform optimization over a horizon of 10 days. Figure 3.2 shows the resulting optimal trajectory.

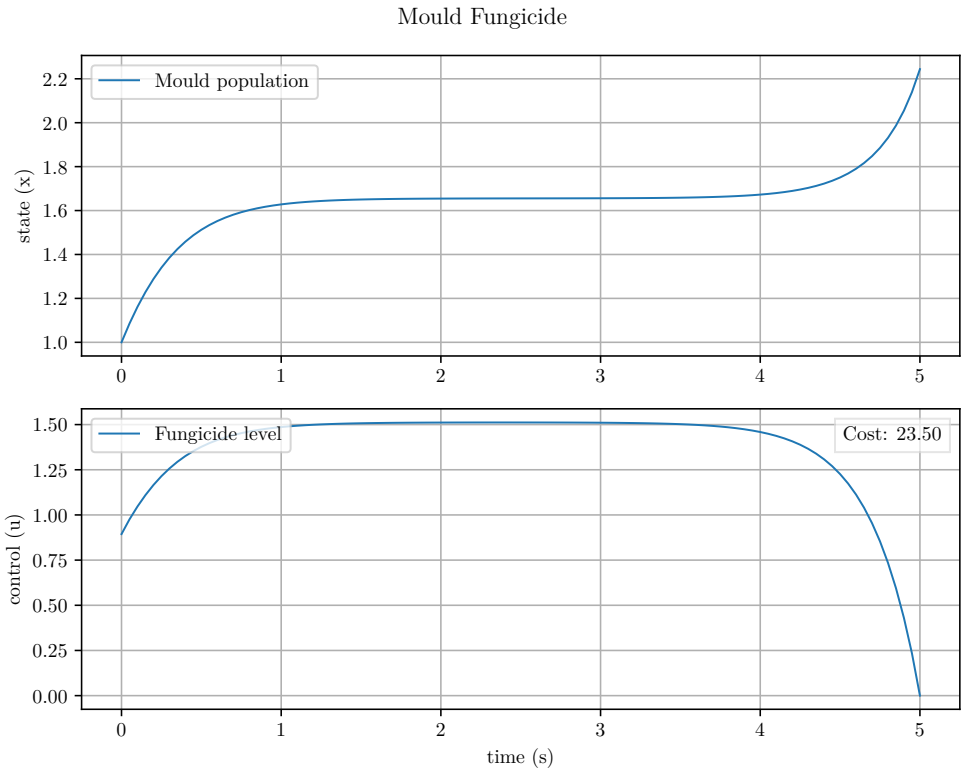


Fig. 3.2. An optimal solution to the Mould Fungicide problem.

3.3. Learning Environment Dynamics

We have just seen two example systems, and the corresponding optimal trajectories that we can find by performing trajectory optimization. We would like a technique to learn the dynamics of these systems (and then use our model of the dynamics to solve the trajectory optimization problem).

The previous chapter described two general flavours of model we can use. In the first and simpler case, we are given the parametric form of the dynamics, and need to estimate the values of several coefficients. For example, in the Cancer Treatment domain, we are given the dynamics equation, but do not know the values of r and δ. Our task is to estimate these quantities.

In the second and more general case, we do not have any information about the parametric form of the dynamics function. As a result, we must use models which can generalize to arbitrary ODE dynamics. In this work we use Neural ODEs to this end. Our task is then to learn weights and biases of the neural network such that the dynamics function resulting from applying the neural network is as close as possible to the true dynamics.

The previous chapter also described two general approaches to learn these parameters, regardless of which of the above cases we're in. The first of these is the MLE approach, in

which model learning and planning are kept separate. Gradients are only propagated through the numerical integration, not through the NLP solver. We will explore this approach in detail in the following chapter.

The second of these is the end-to-end approach, in which model learning and planning are tightly integrated. Gradients are propagated through the numerical integration and the NLP solver. We will examine this approach in the chapter after next.

Chapter 4

Trajectory Optimization with a Learned Model

This chapter studies the case of MLE dynamics learning, where we gather data, fit the model, and once training is complete, use the model for planning. If desired (usually depending on the quality of our controls as evaluated in the true dynamics), we can sample new training trajectories and repeat the process, refining our model over time as we gather more training data.

In the next section, we will evaluate this approach when dealing with the simpler parametric models where we already know the correct functional form of the dynamics. In the section after that, we will explore the more general case in which we use a Neural ODE to learn the dynamics function.

Experiments were performed using a single shooting formulation, with a time discretization of 100 timesteps, and Heun's method used for numerical integration. All training trajectories were generated using controls generated from a Gaussian random walk process. The `ipopt` package was used for the NLP solves [**Wächter and Laird, 2005**].

4.1. Parametric Models

We would like to learn parameters $\boldsymbol{\theta}$ corresponding to the unknown coefficients of a known dynamics function, such that the learned dynamics $f(\boldsymbol{x}(t), \boldsymbol{u}(t), t, \boldsymbol{\theta})$ best approximate the true dynamics. Let's explore this approach on some of the domains in Myriad.

4.1.1. Cancer Treatment

To generate data, we create a Cancer Treatment environment with the default parameters from Table 3.1. We start with an initial parameter guess of $r = 0.1$ and $\delta = 0.1$. We train with early stopping on a dataset of 10 train trajectories and 3 validation trajectories, generated by using random walk controls. The resulting training curve is shown in Figure

4.1. The learned parameters were $\delta : 0.449$ and $r : 0.300$, which are very close to the true values ($\delta : 0.5$ and $r : 0.3$).

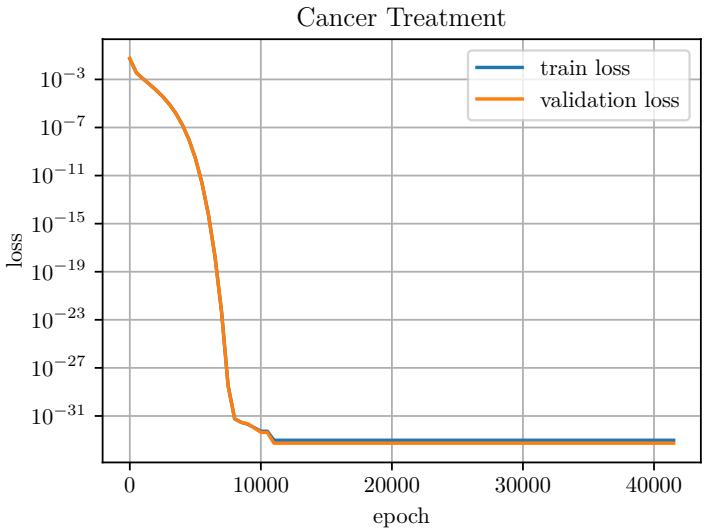


Fig. 4.1. The training curve for the Cancer Treatment problem.

We show the prediction performance on a test trajectory in Figure 4.2. The prediction appears to be perfect.

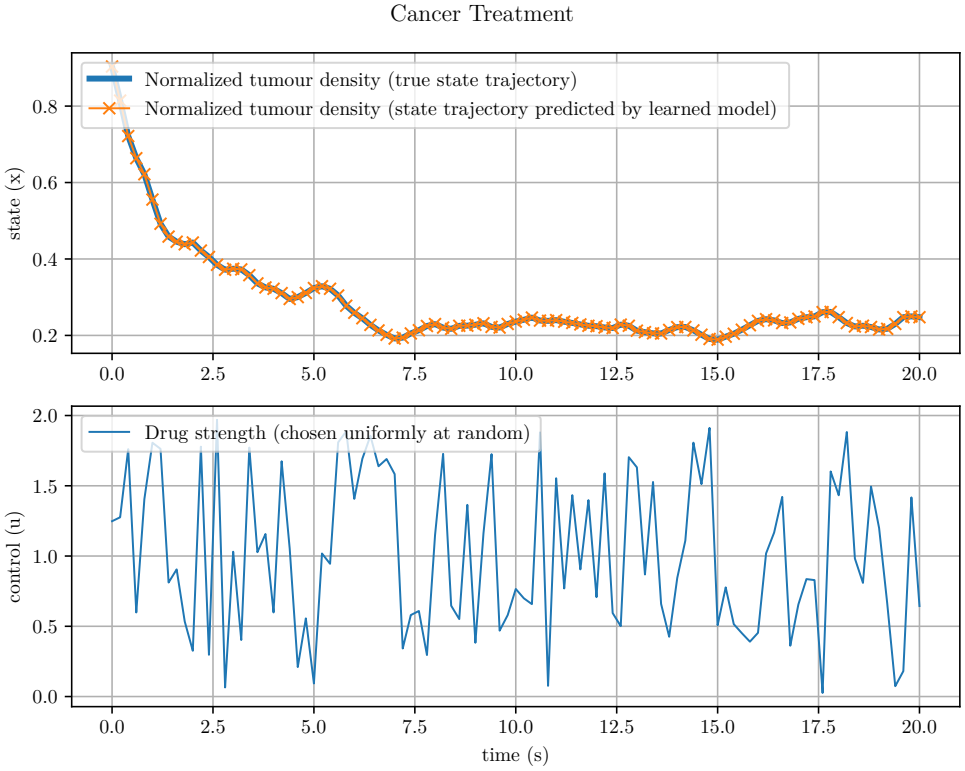


Fig. 4.2. Prediction performance on test trajectory for the Cancer Treatment model.

When we use the learned parameters in the model for planning, we get the trajectory shown in Figure 4.3

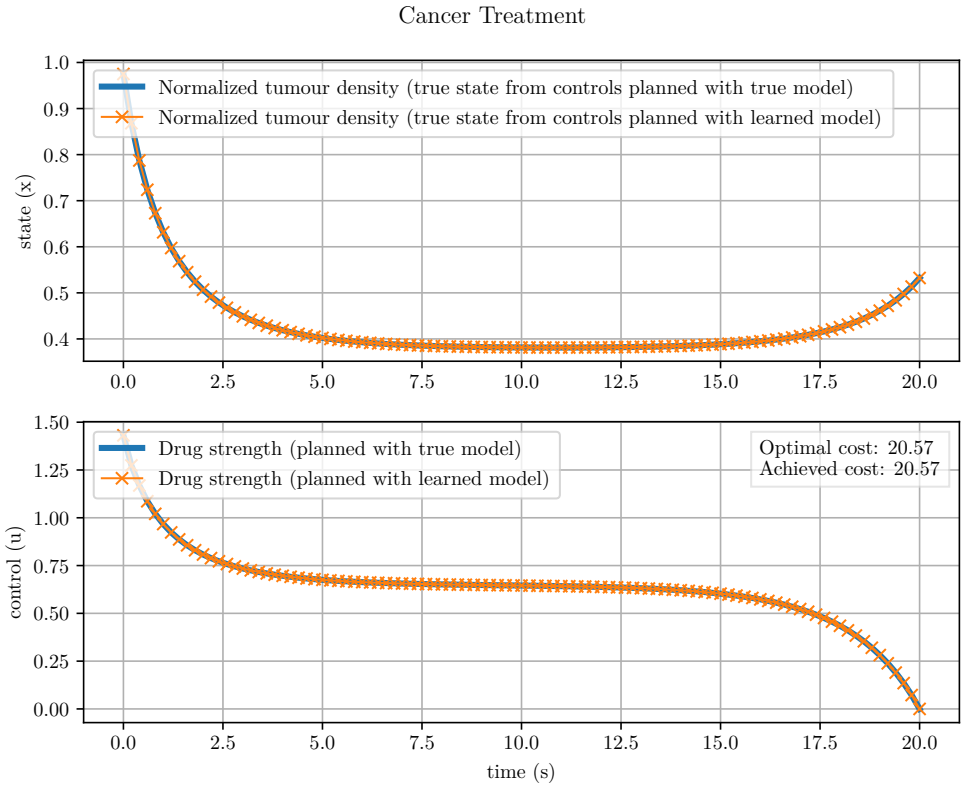


Fig. 4.3. Performance of planning using the learned model, compared with the true optimal controls, on the Cancer Treatment domain.

We see that the cost of planning with the learned model is the same as the optimal one.

4.1.2. Mould Fungicide

To generate data, we create a Mould Fungicide environment with the default parameters from Table 3.2. We start with an initial parameter guess of $r = 0.1$ and $M = 8$. We train with early stopping on a train set of 10 trajectories and validation set of 3 trajectories. The resulting training curve is shown in Figure 4.4. Again, we learn the parameters almost exactly. The learned parameters were r: 0.300 and M: 9.9998; very close to the true ones (r: 0.3 and M: 10).

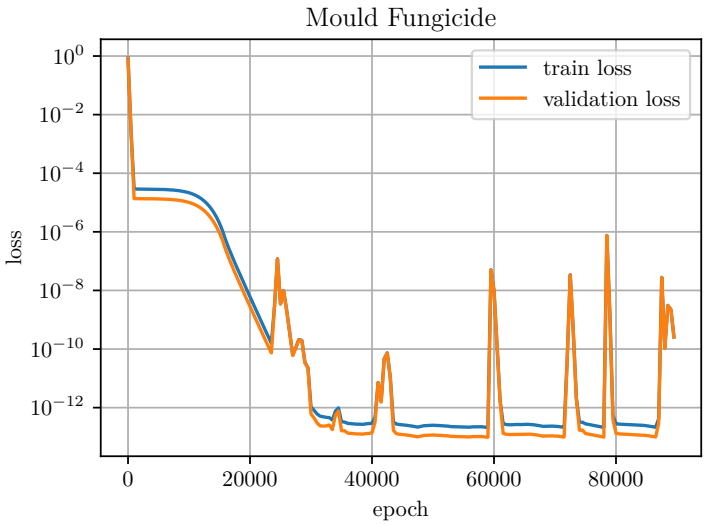


Fig. 4.4. The training curve for learning a parametric model of the Mould Fungicide problem.

We show prediction performance on the test trajectory in Figure 4.5.

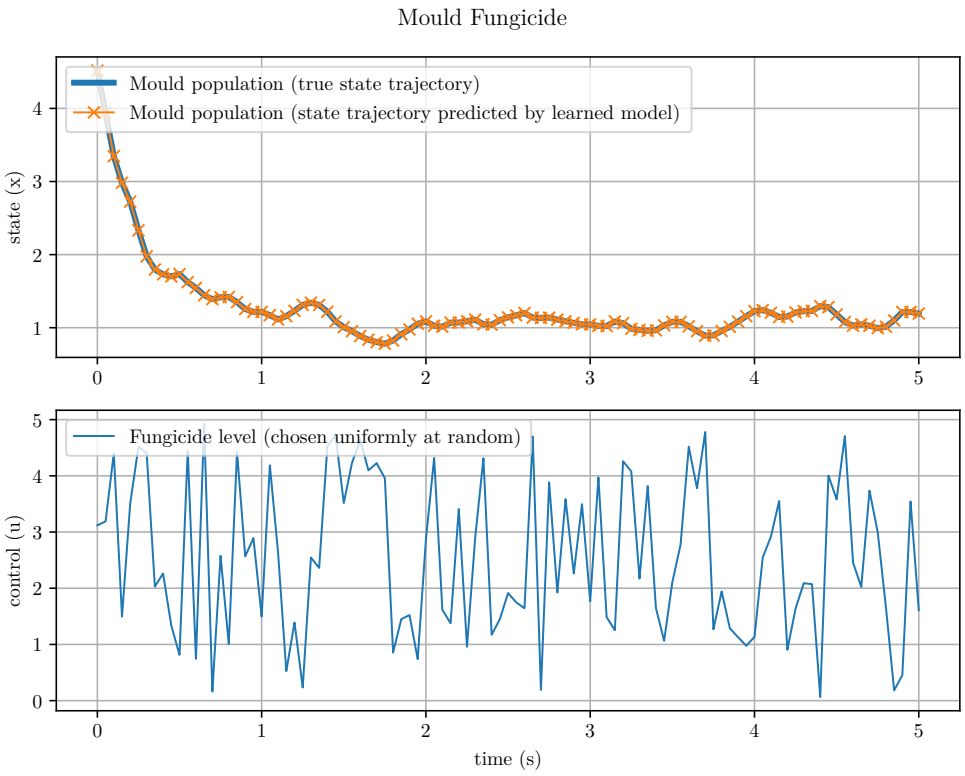


Fig. 4.5. Prediction performance on test trajectory for the Mould Fungicide model.

When we use the learned parameters in the model for planning, we get the trajectory shown in Figure 4.6

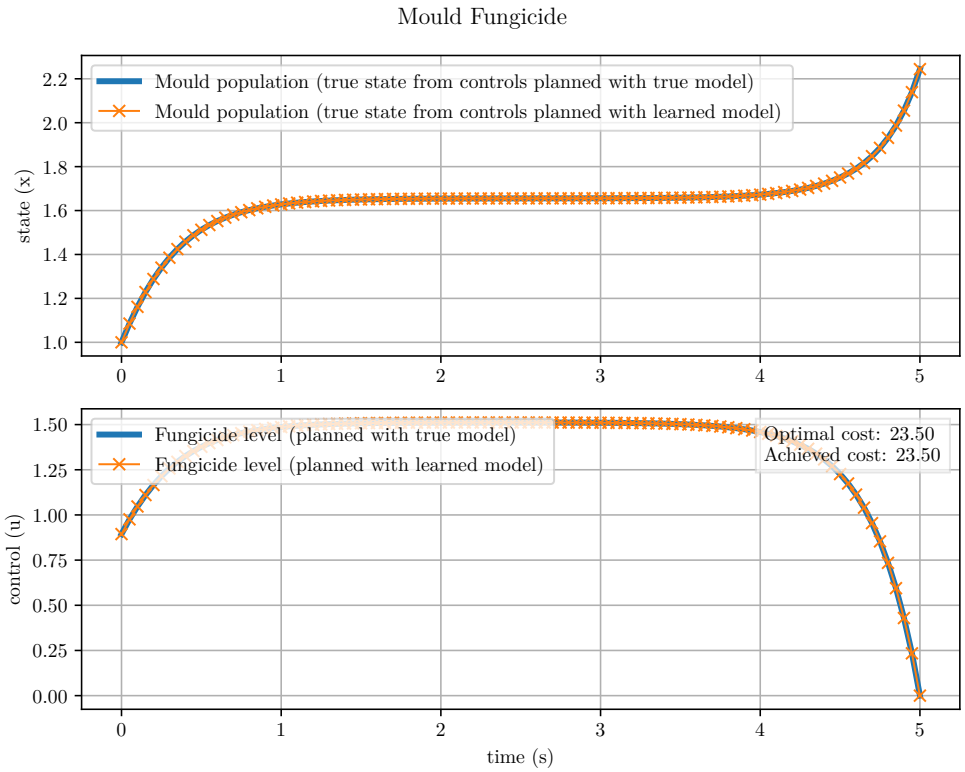


Fig. 4.6. Performance of planning using the learned model, compared with the true optimal controls, on the Mould Fungicide domain.

We see that the resulting cost is the same as the optimal one.

4.1.3. Other Domains

The Myriad repository contains many other systems in which to test the performance of our algorithm. Table 4.1 summarizes the performance on many other domains. In all cases we used 10 training trajectories and 3 validation trajectories.

System	Parameters (true/learned)	Cost (best/achieved)	Defect (best/achieved)
Bacteria	A: 1/1, B: 1/1, C: 1/1	$-7.98/-7.98$	NA
Bear Populations	r: 0.1/0.1, K: 0.75/0.75, m_f: 0.5/0.5, m_p: 0.5/0.5	12.28/12.28	NA

Bioreactor	G: 1/1, D: 1/1	$-1.39/-1.39$	NA
Cancer Treatment	δ: 0.45/0.45, r: 0.3/0.3	20.57/20.57	NA
Cart-Pole Swing-Up	g: 9.81/9.81, m_1: 1/1, m_2: 0.3/0.3, ℓ: 0.5/0.5	87.78/87.78	$\begin{bmatrix} 0/0 & 0/0 & 0/0 & 0/0 \end{bmatrix}^\top$
Glucose	a: 1/1, b: 1/1, c: 1/1	1354.02/1354.02	NA
Mould Fungicide	r: 0.3/0.3, M: 10/10	23.50/23.50	NA
Mountain Car	g: 0.0025/0.0025, p: 0.0015/0.0015	8.57/8.57	$\begin{bmatrix} 0/0 & 0/0 \end{bmatrix}^\top$
Pendulum	g: 10/11.943, m: 1/0.701, ℓ: 1/1.194	25.75/25.49	$\begin{bmatrix} 0/0.04 & 0/0.04 \end{bmatrix}^\top$
Predator Prey	d_1: 0.1/0.1, d_2: 0.1/0.1	1.79/1.79	0/0
Timber Harvest	K: 1/1	$-5104.67/-5104.67$	NA
Tumour	ξ: 0.084/0.084, b: 5.85/5.170, d: 0.00873/0.00873, G: 0.15/0.15, μ: 0.02/-0.660	7571.67/7571.73	NA
Van Der Pol	a: 1/1	2.87/2.87	$\begin{bmatrix} 0/0 & 0/0 \end{bmatrix}^\top$

Table 4.1. Summary of performance of learning and planning with parametric models on a variety of environments. The first column lists the environment, followed by the true and learned parameters, and then the cost (and if applicable, defect) resulting from those parameters.

In most cases, we are able to learn the true parameters exactly. Thus, the result that we get from planning with the learned parameters is exactly the same as that achieved by planning with the true model.

In two cases, namely the Pendulum and Tumour domains, we learn parameters which are significantly different from the true ones. This brings up the question of unidentifiability: if different combinations of parameters lead to exactly the same behaviour of the model, then the true parameters are said to be *unidentifiable*, since there is no way to guess them

from data [**Lehmann and Casella, 2006**]. In the case of Pendulum, the model is clearly unidentifiable. We see this with reference to the Pendulum dynamics, expressed in Eq. (4.1.1):

$$f\left(\begin{bmatrix} x \\ \dot{x} \end{bmatrix}, u, t\right) = \begin{bmatrix} \dot{x} \\ \frac{-g}{2\ell}\sin\theta + \frac{u}{m\ell^2} \end{bmatrix}. \tag{4.1.1}$$

We have two equations in three parameters: we can choose any positive value of ℓ, and g and m can be chosen such that the quantities $\frac{-g}{2\ell}$ and $\frac{1}{m\ell^2}$ do not change, resulting in the same dynamics.

It seems that a similar phenomenon is occurring in the Tumour model. Indeed, the parameters found raise an additional question. Here, the algorithm converged on negative value of $\mu = -0.66$. In the model, μ represents the daily loss of endothelial cells due to natural causes. Does it make physical sense that there be a daily *gain* of these cells? This raises the question of how to best structure a parametric system model for learning parameters. In particular, what might seem obvious when building a model (for example, that the parameters are all positive) might not be respected when learning the dynamics. Structuring the model to avoid learning unintended parameters is a necessary step towards ensuring that the true parameters are identifiable.

Regardless of the parameters learned, we observe that in both the Pendulum and Tumour domains, the learned model performs very well, almost matching the performance of the true model.

Finally, one domain – the HIV Treatment environment – did not work with this algorithm. Regardless of initial parameter guess, the optimization process failed due to numerical issues. Further investigation will be necessary to understand exactly where the problem lies.

4.1.4. Local Minima and Discounting

It is important to report that in three domains – Bacteria, Mountain Car, and Cart-Pole Swing-Up – we applied a slightly different algorithm than that described in Section 2.16.1. When applying the standard algorithm, all three got stuck in local minima. For Mountain Car and Cart-Pole Swing-Up, this is likely due to the oscillatory dynamics: if the guess of parameters is such that the oscillation period is different than those given in the training data, the algorithm might not be able to make progress, because any change in parameter would lead to worse mean squared error because of how the phases line up between the trajectories. This problem is illustrated in Figure 4.7.

One way around this which was found to be effective in practice is to include a discount factor as a weight when summing up the mean squared error of the predicted trajectory from the true trajectory. Let i be the current epoch, let $\boldsymbol{x} \equiv \boldsymbol{x}_{0:N}$ be the true state trajectory with

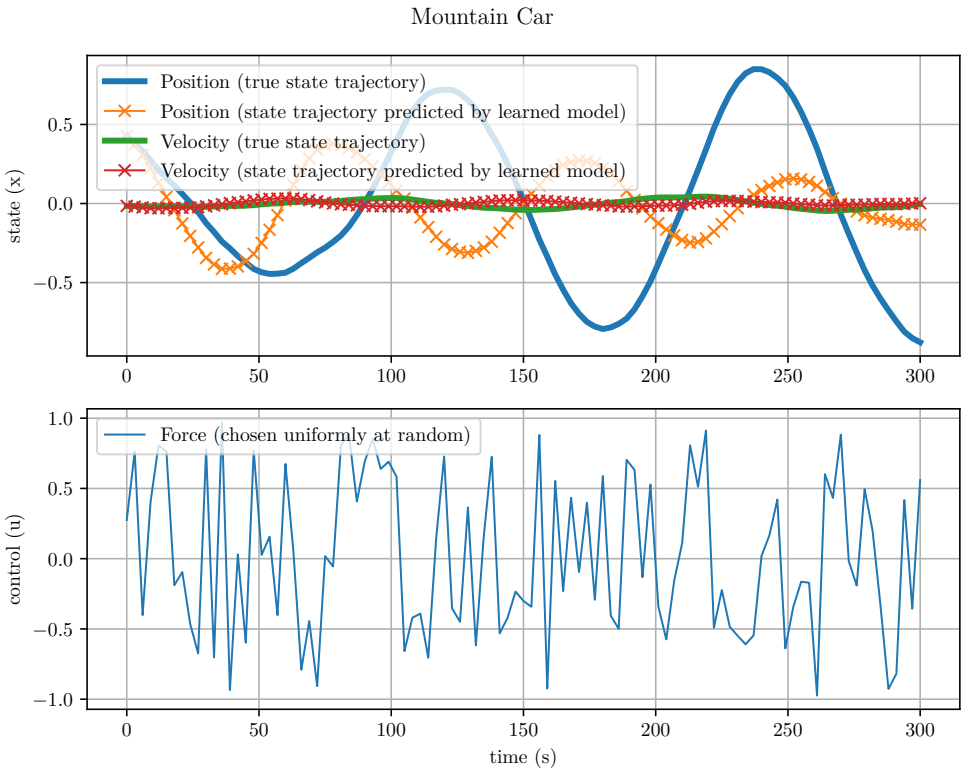


Fig. 4.7. An illustration of how the algorithm can get stuck in local minima in some oscillatory domains. Any change in the parameters would lead to an immediate worsening in prediction, because of the interaction of the phases of the prediction and the training trajectories.

$\boldsymbol{x}_0$ the start state, and define $\tilde{\boldsymbol{x}} \equiv {}_hT_{t_s}^{t_f}(f, \boldsymbol{x}_0, \boldsymbol{u}_{0:N}, \hat{\boldsymbol{\theta}})$ to be the estimated state trajectory. Let

$$\gamma = 1 - \frac{1}{1 + e^{a+i\cdot b}}, \quad \text{and} \quad \boldsymbol{\gamma} = \begin{bmatrix} \gamma^0 & \gamma^1 & \dots & \gamma^N \end{bmatrix}^\top,$$

where a and b are tunable values. The result is that $\boldsymbol{\gamma}$ is a vector of weights starting at 1 and decreasing to close to 0. As we progress in training, and therefore the epoch i increases, the values of $\boldsymbol{\gamma}$ approach 1, until it is approximately equal to the identity vector, weighting all timesteps equally.

Remember, the undiscounted mean squared error for this trajectory can be written as

$$L(\hat{\boldsymbol{\theta}}) = \frac{1}{ND} \sum_{\text{timestep}} \sum_{\text{state dim.}} (\tilde{\boldsymbol{x}} - \boldsymbol{x}) \odot (\tilde{\boldsymbol{x}} - \boldsymbol{x}).$$

With discounting, the mean squared error is

$$L(\hat{\boldsymbol{\theta}}) = \frac{1}{ND} \sum_{\text{timestep}} \boldsymbol{\gamma} \odot \left(\sum_{\text{state dim.}} (\tilde{\boldsymbol{x}} - \boldsymbol{x}) \odot (\tilde{\boldsymbol{x}} - \boldsymbol{x}) \right).$$

If we want to perform this over a batch, then we also need to sum over the batch dimension before applying the discounting. Let $\tilde{\boldsymbol{X}} \equiv {}_hT_{t_s}^{t_f}(f, \boldsymbol{X}_0^{1:M}, \boldsymbol{U}_{0:N}^{1:M}, \hat{\boldsymbol{\theta}})$ denote the estimated state

trajectories. Then we can apply the discounting as

$$L(\hat{\boldsymbol{\theta}}) = \frac{1}{MND} \sum_{\text{timestep}} \boldsymbol{\gamma} \odot \left(\sum_{\text{batch}} \sum_{\text{dim. state}} \left(\tilde{\boldsymbol{X}} - \boldsymbol{X}\right) \odot \left(\tilde{\boldsymbol{X}} - \boldsymbol{X}\right) \text{dim.} \right)$$

In practice we found this allowed us to overcome the local minima caused by oscillatory dynamics, while still enabling us to get to a good final solution, as we reduce the amount of discounting over time. The values $a = 2$ and $b = 10^{-6}$ were found to perform well, though it is likely that more task-specific tuning could lead to faster convergence.

4.1.5. Noise Study

It is natural to wonder how robust our learning algorithm is to the addition of observation noise. Indeed, until now we have considered observations to be exact, which is not realistic when using real-world sensors.

To test this, we perform the same training and planning, but when we generate training and validation sets, we first add Gaussian noise to the state observations. The 'spread' quantity describes how large the standard deviation of our normal distribution is. If our training state trajectory is $\boldsymbol{x}$, then before using it for training we add some $\boldsymbol{\epsilon}$ of noise to it:

$$\boldsymbol{\epsilon} \leftarrow \mathcal{N}\left[0, \text{spread} \cdot (\boldsymbol{x}_{0:N}^{\text{upper}} - \boldsymbol{x}_{0:N}^{\text{lower}})\right]. \tag{4.1.2}$$

Figure 4.8 shows that as observation noise increases, the accuracy that we achieve decreases. However, for small amounts of noise, model performance remains very strong.

4.2. Neural ODE Models

The previous section showed how learning parameters of a model when we know the true functional structure is often very effective, and leads to excellent performance when using the model for planning. In many cases, however, we might not know the functional form of the model we want to learn. In this case, we have no choice but to use a broader class of models in our learning.

As we saw in Section 2.18, one very general class of models for continuous-time dynamics is the Neural ODE [**Chen et al., 2018**]. We are interested in using Neural ODEs as a model class to learn (and plan with) system dynamics. Because Neural ODEs are very general in their modelling ability, it might take more data to effectively learn dynamics. As such, if we are not satisfied with the results after initial training has converged, we can augment our dataset with new data of equal size, and train on the new, augmented dateset.

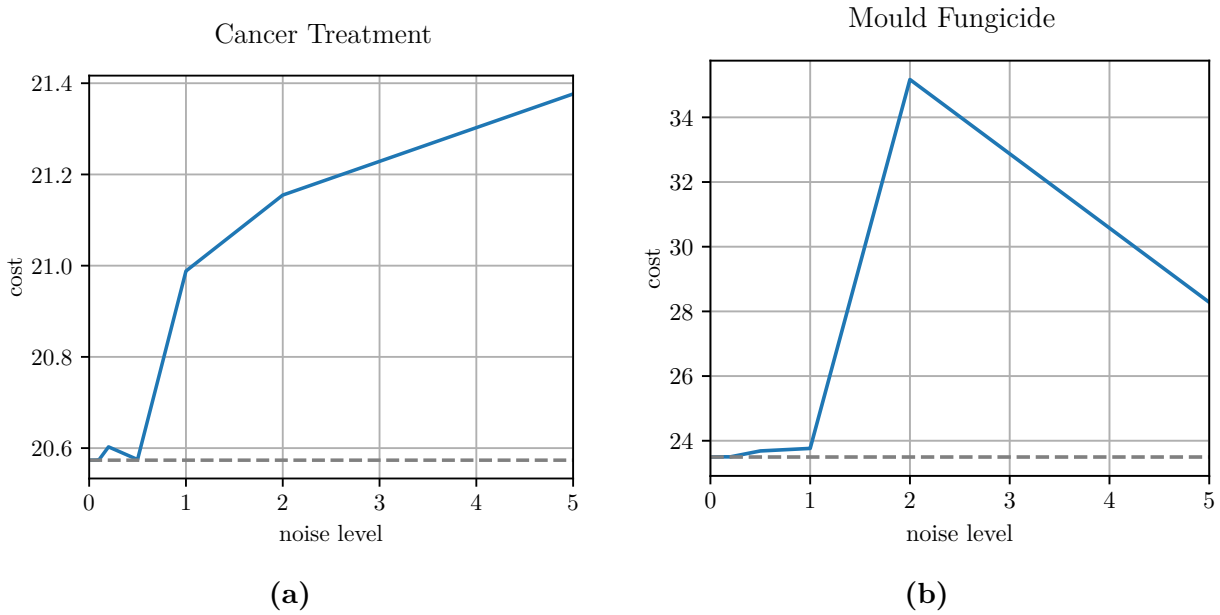


Fig. 4.8. Performance of trajectory optimization using models learned from corrupted observations, on (a) the Cancer Treatment domain and (b) Mould Fungicide domain. The horizontal axis corresponds to the 'spread' parameter in Eq. 4.1.2, which determines the standard deviation of the distribution from which the random values are drawn. The horizontal dashed line shows the best possible cost.

4.2.1. Cancer Treatment

We instantiate a Neural ODE model to match the Cancer Treatment domain. The dynamics function is represented by a MLP with a two-dimensional input (state and control), and a one-dimensional output (time derivative of state). We use a fully-connected network with two hidden layers of 100 neurons each. The cost function, along with the other settings of the domain, are kept the same as in the previous experiments. We train with early stopping on a dataset with 10 train trajectories and 3 validation trajectories.

Figure 4.9 shows the loss on our train and validation sets over the course of training. In this specific experiment, we did not generate any additional datasets after the initial training, since the planning was already good enough after training on one dataset.

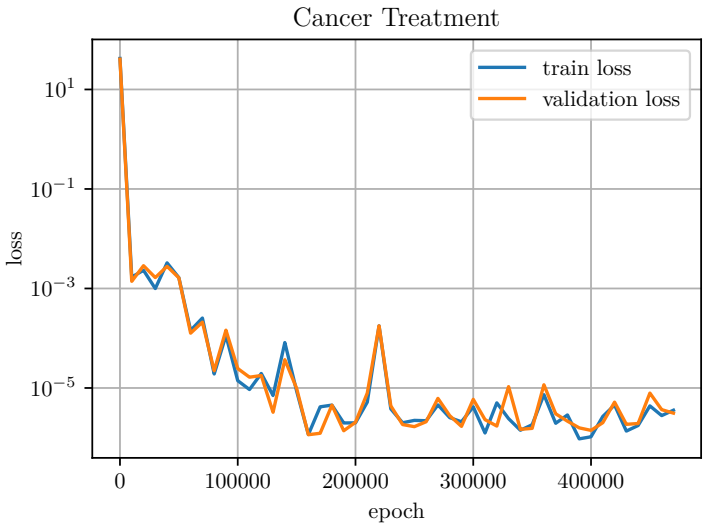


Fig. 4.9. Training curve of a Neural ODE model on the Cancer Treatment domain.

The learned model has excellent prediction performance. An example of this is shown on a test trajectory in Figure 4.10.

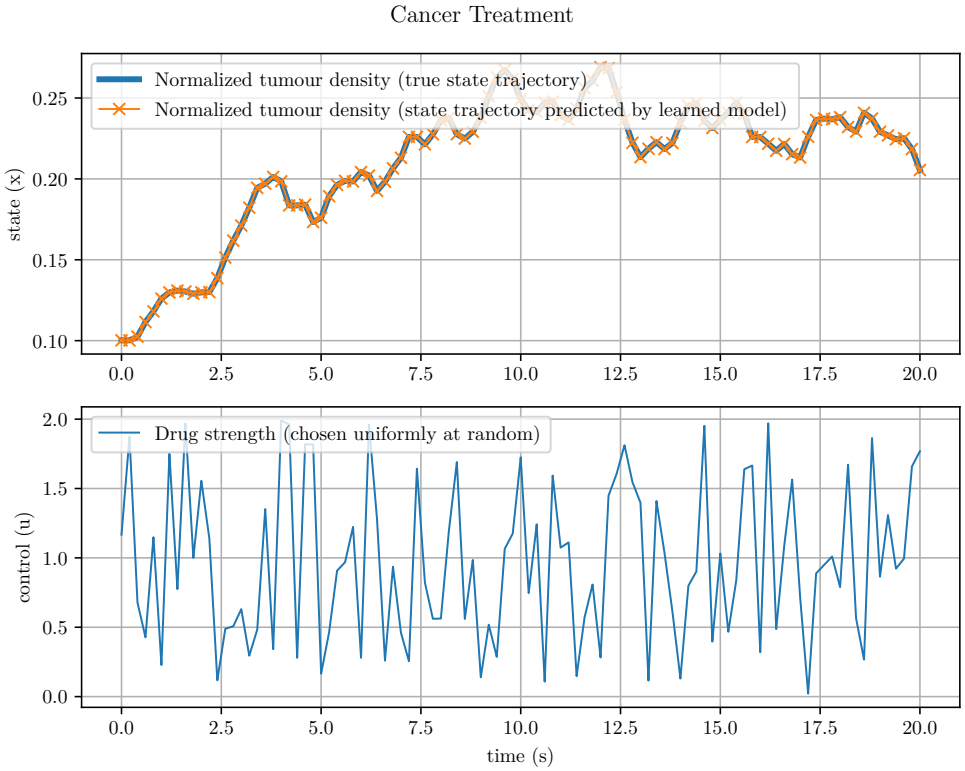


Fig. 4.10. Prediction performance of the learned Neural ODE model on random controls, compared with the true system dynamics, on the Cancer Treatment domain.

Figure 4.11 shows the result of using the learned Neural ODE model for planning.

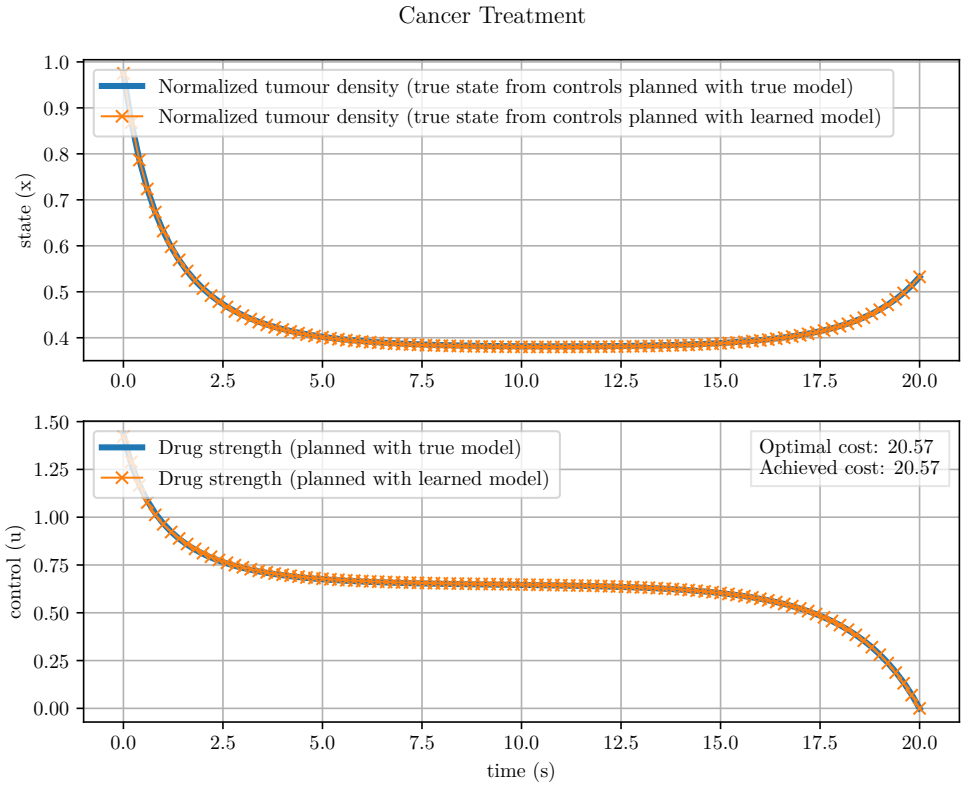


Fig. 4.11. Planning performance of the learned Neural ODE model, compared with the true system dynamics, on the Cancer Treatment domain.

The control trajectory we get from planning with the model is almost exactly the same as that found when planning with the true model. This is surprising, because the Neural ODE we started with had no information about the system dynamics.

4.2.2. Mould Fungicide

We perform the same experiments on the Mould Fungicide domain, training with early stopping on a train set of 10 trajectories and validation set of 3 trajectories. The training curve is shown in Figure 4.12.

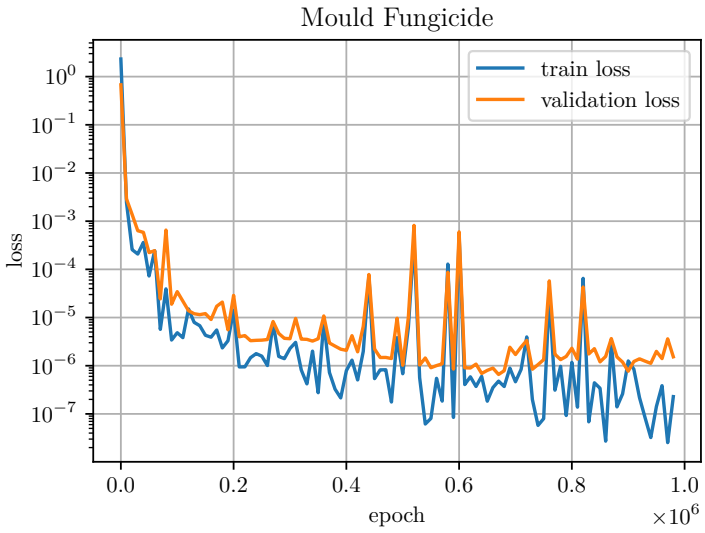


Fig. 4.12. Training curve a Neural ODE model on the Mould Fungicide domain.

Prediction performance is shown in Figure 4.13. The prediction capability is very good.

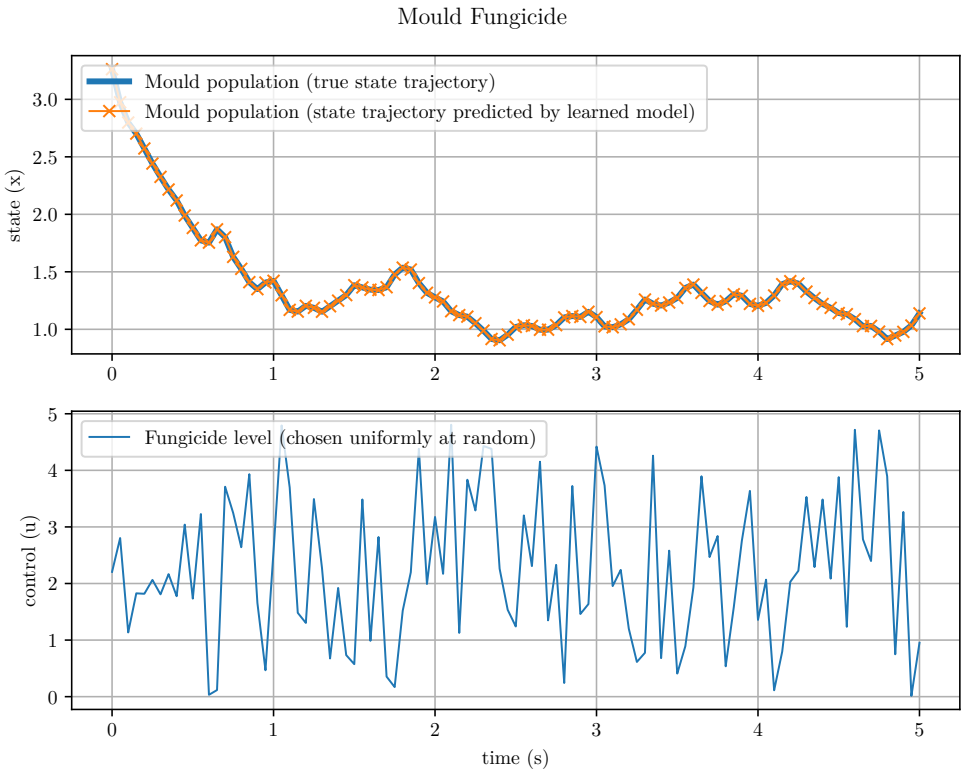


Fig. 4.13. Prediction performance of the learned Neural ODE model on random controls, compared with the true system dynamics, on the Mould Fungicide domain.

Figure 4.14 shows the result of using the learned Neural ODE model for planning. In this domain also, we observe that the learned Neural ODE model performs excellently for both prediction and planning.

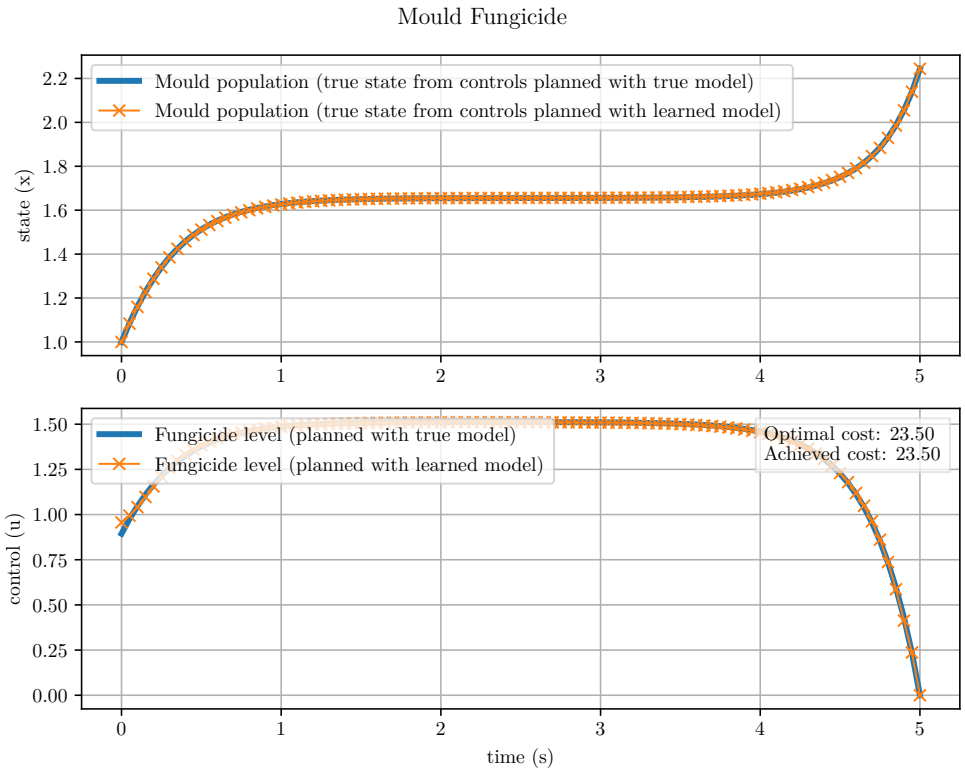


Fig. 4.14. Planning performance of the learned Neural ODE model, compared with the true system dynamics, on the Mould Fungicide domain.

4.2.3. Other Domains

We test the same algorithm the same list of other domains. All models have a neural network of two hidden layers each of 100 neurons, except for the Van Der Pol model, which uses 50 neurons per hidden layer.

System	Cost (best/achieved)	Defect (best/achieved)	# Traj.s/D.set	# D.sets
Bacteria	−7.98/−2.72	NA	10	5
Bear Populations	12.28/12.29	NA	10	4
Bioreactor	−1.39/−1.39	NA	10	1
Cancer Treatment	20.57/20.57	NA	10	1

Cart-Pole Swing-Up	87.78/242.25	$\begin{bmatrix} 0/-0.95 \\ 0/0.23 \\ 0/-0.85 \\ 0/6.63 \end{bmatrix}$	100	7
Glucose	1354.02/1354.02	NA	10	1
Mould Fungicide	23.50/23.50	NA	10	1
Mountain Car	8.57/15.87	$\begin{bmatrix} 0/0.03 \\ 0/-0.01 \end{bmatrix}$	100	5
Pendulum	25.42/20.88	$\begin{bmatrix} -0.01/0.37 \\ -0.01/0.61 \end{bmatrix}$	100	1
Predator Prey	1.79/1.95	0/0.14	100	5
Timber Harvest	$-5104.67/-3928.77$	NA	100	1
Tumour	7571.67/8468.68	NA	100	2
Van Der Pol	2.87/11.12	$\begin{bmatrix} 0/-0.35 \\ 0/-1.76 \end{bmatrix}$	10	3

Table 4.2. Summary of performance of learning and planning with Neural ODE models on a variety of environments. The layout is the same as the previous table, except that parameters are omitted, since there is no way to directly compare the true parameters with the weights and biases of the neural network. Instead, the fourth column shows the number of trajectories per dataset used in training, and the fifth column shows the number of datasets that were used to train the model.

As in the previous experiment, the HIV Treatment model diverged during training, and is not included in the table.

We see that while performance is excellent in many domains, the Neural ODE model was unable to reproduce results on the level of the parametric models. We gain some intuition as to why this might be by studying an example in the next section.

4.2.4. Challenging Environments

In Figure 4.15, we see the performance on the Pendulum domain when planning with the learned Neural ODE model. The control trajectory planned with the learned model suggests that the learned dynamics are quite similar to the true dynamics, yet, the required final

configuration – having the pendulum stand up vertically with zero velocity – is not achieved. This example illustrates how difficult it is to reach a precise final state configuration using only learned dynamics for planning.

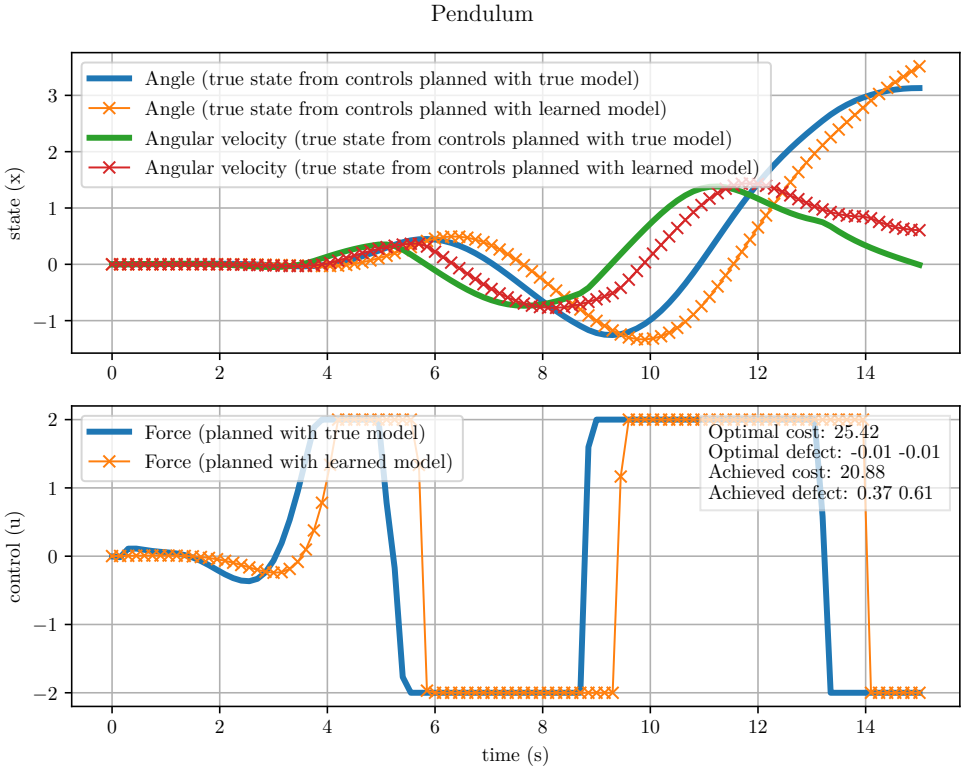


Fig. 4.15. Planning performance using the learned Neural ODE model, compared with using the true system dynamics, on the Pendulum domain.

For models with highly nonlinear dynamics like the Pendulum domain, there are at least two likely factors that make this a difficult problem. First, it is difficult for the model to learn the dynamics near the desired final configuration, because balancing the pole vertically is in itself a difficult state to reach. Thus, it is unlikely that the algorithm saw this state reached many times in its training set. Second, even if the model was able to learn the difficult-to-reach areas of state space, the highly nonlinear nature of domains such as Pendulum means that even a small deviation from the true dynamics will lead to compounding errors over time. One way to mitigate this second issue is to close the loop by applying model predictive control (MPC), that is, to re-plan over the course of the trajectory with a receding time horizon.

4.2.5. Unexpected Solution

There was an unexpected solution found in the Predator Prey environment, shown in Figure 4.16. Planning with the learned model led to a completely different control trajectory

than that found with the true model. Admittedly, the trajectory found with the learned model is significantly more expensive to execute, but it does do a good job at almost satisfying the final state requirement. This result suggests that for some domains, the optimal solution is only achievable with a perfect model, but an imperfect model might still yield acceptable results.

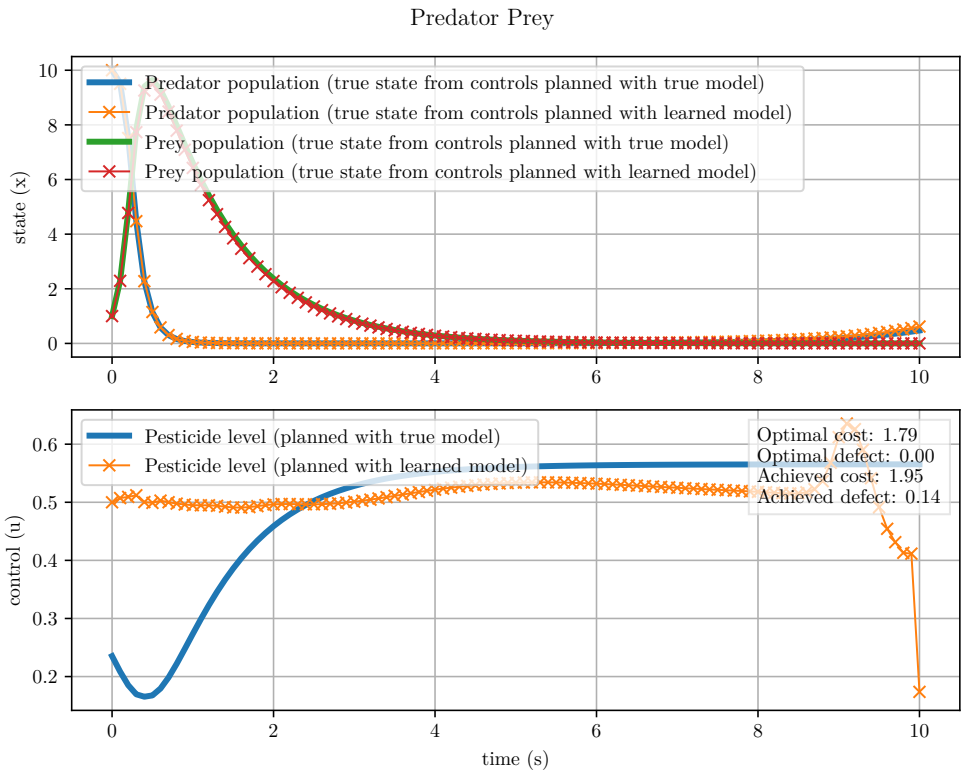


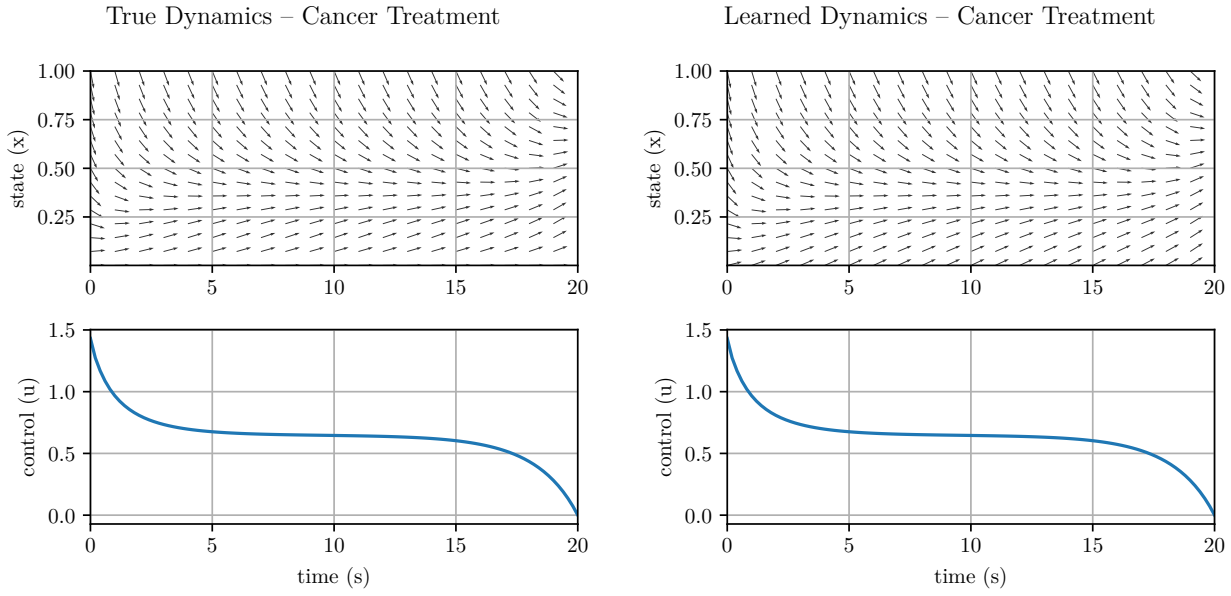
Fig. 4.16. Planning performance using the learned Neural ODE model, compared with using the true system dynamics, on the Predator Prey domain.

4.2.6. Dynamics Study

Since in this setting we are learning weights and biases for a neural network, and not parameters of a given parametric model, we cannot examine the parameters to see if our model has learned the correct system dynamics. What we can do is compare the dynamics across state space for a given control trajectory. In particular, we can visualize the dynamics using a vector field which indicates the value of the dynamics function across states. This is shown for the Cancer Treatment environment in Figure 4.17, and for the Mould Fungicide environment in Figure 4.18.

In these plots, we see that the learned dynamics are generally similar the true dynamics. As we might expect, the learned dynamics differ most in areas of state space which were rarely visited during training. For example, in the Cancer Treatment domain, the Neural

ODE makes an overly large guess for the speed of increase in tumour density (the state) for small density (state) values. Similarly, in the Mould Fungicide domain, it estimates that very large mould populations (state) will shrink less quickly than they do in practice.



(a) **(b)**

Fig. 4.17. Comparison of (a) true dynamics and (b) learned dynamics when applying the true optimal controls in the Cancer Treatment domain.

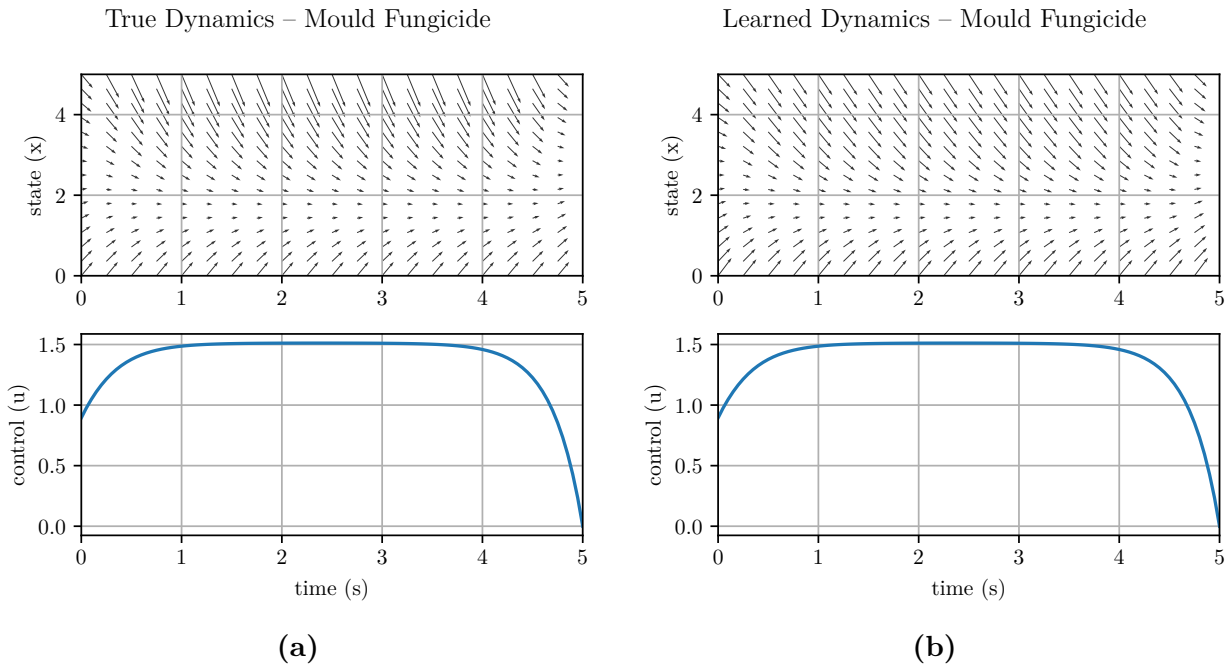


Fig. 4.18. Comparison of (a) true dynamics and (b) learned dynamics when applying the true optimal controls in the Mould Fungicide domain.

4.2.7. Noise Study

As when using parametric models, we can examine how much worse our models become as we add more noise to our training set. With reference to Figure 4.19, we see again that for low noise levels, performance remains good. However, in the Cancer Treatment domain, it appears that medium noise levels lead to somewhat worse performance.

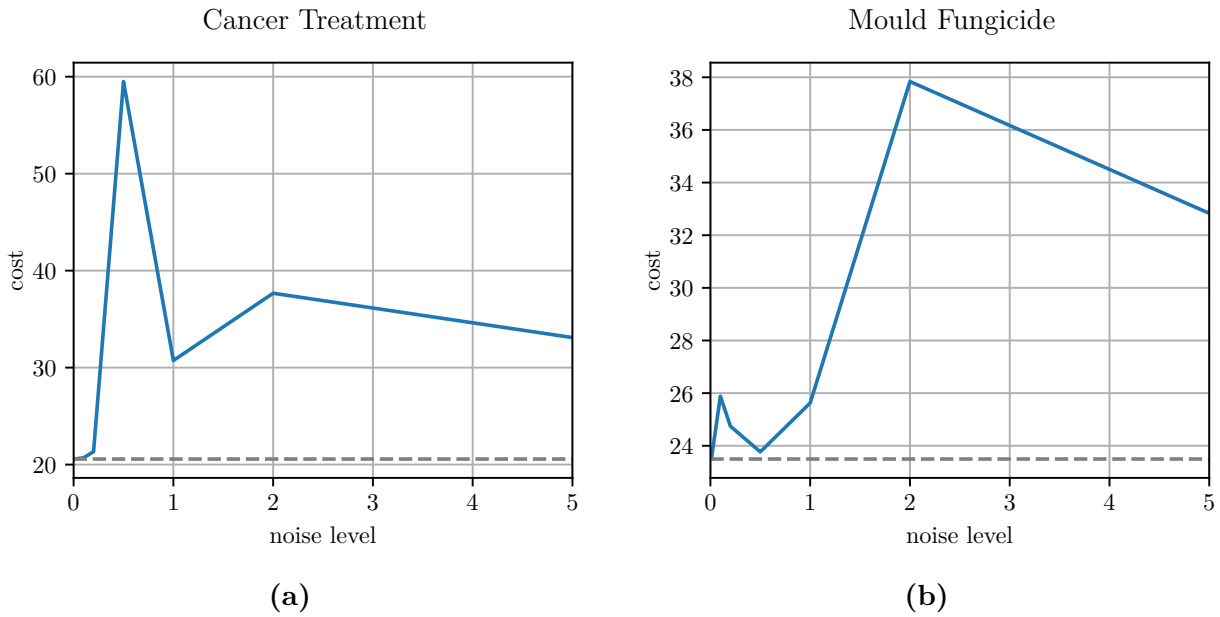


Fig. 4.19. Performance of trajectory optimization using Neural ODE models learned from corrupted observations on (a) the Cancer Treatment domain and (b) the Mould Fungicide domain. The horizontal dashed line shows the best possible cost.

4.3. Summary

In this chapter, we explored the problem setting of learning dynamics from trajectories in a MLE approach, and then using the learned model to plan optimal controls through trajectory optimization. When learning parameters of simple parametric models, we found that the learning worked very well, leading to perfect system identification in several cases. In other cases, the introduction of a discount factor was necessary to overcome local minima. Regardless, practically all parametric models converged to a useful model

The performance of Neural ODE models was also good. In several domains, planning performance on par with that when using a true model was achieved, showing effective learning of the system's dynamics. The algorithm struggled more in other domains, particularly those with highly nonlinear dynamics. In these cases, planning with the learned model led to controls which performed significantly worse than the optimal ones. It is likely that with larger training sets, hyperparameter tuning, and the application of MPC, performance could be improved significantly.

Chapter 5

Learning End-to-End Models for Trajectory Optimization

The previous chapter explored the performance of learning models to reproduce dynamics from sampled trajectories, and then using the learned model to plan a sequence of controls. The alternative method is to learn the model dynamics and plan controls with them at the same time. This technique is interesting because it can be trained in an end-to-end fashion, updating model parameters directly based on the performance of the resulting controls. Its end-to-end nature makes it amenable to use as a sub-component of a larger learning algorithm – one which includes the inductive bias of solving a trajectory optimization problem.

As we discussed in Section 2.17, the major challenge of this approach is differentiating through the planning algorithm. It is not feasible to use AD through a fully unrolled NLP solve, nor does the IFT represent a feasible solution due to the relative slowness of the NLP solver. A compromise solution is to manually unroll only a few steps of the NLP solve, and use AD on that small unrolled segment. Crucially, we warm-start from where the previous partial solve left off, occasionally resetting to an initial guess.

The end-to-end approach requires an outer loss describing how good the current controls are. Here we use the loss of an imitation learning problem, in which an expert's optimal control trajectory is provided to us. This is analogous to the approach used in [**Amos et al., 2018**]. As in the previous chapter, we will explore the performance of this approach on a variety of domains, using both parametric and Neural ODE models.

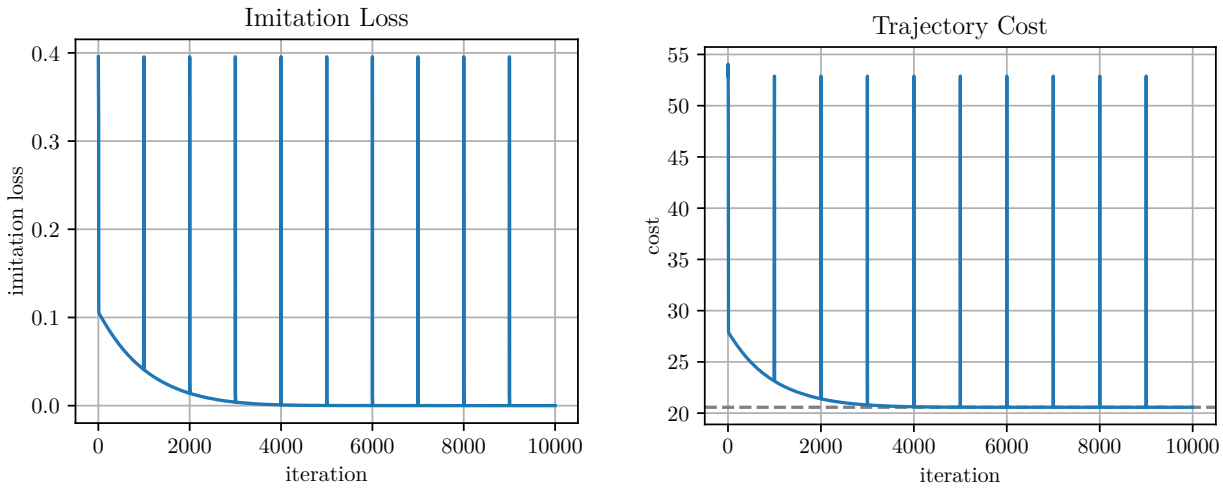
5.1. Parametric Models

We would like to imitate an optimal control trajectory, while differentiating our learning signal through a parametric dynamical system model. We start with an initial guess for the parameters, and then update the parameters by differentiating through 10 steps of unrolled Extragradient applied to the Lagrangian of the NLP resulting from the trajectory

optimization problem. At each step, we warm-start the guess of optimal controls. We reset the warm-start every 1000 iterations.

5.1.1. Cancer Treatment

We begin with the Cancer Treatment domain. In Figure 5.1a we observe the imitation loss over the course of training, while Figure 5.1b shows the integrated cost of applying the current guess of controls. As we would hope, both these quantities decrease over the course of training. This suggests that the controls guess is approaching the expert trajectory.



(a) **(b)**

Fig. 5.1. We observe (a) the imitation loss and (b) the trajectory cost over the course of end-to-end training on the Cancer Treatment domain. The spike every thousand iterations happens after we reset the warm-start to an initial random guess of controls. The values quickly return back to where they were as the training procedure continues. In (b), the horizontal line indicates the best possible performance.

We can view how the controls guess, and corresponding state, evolve over the course of training. This is shown in Figure 5.2. We see that after starting with a random guess, the controls slowly approach the optimal control trajectory, slowing down as they get closer. By the end of training, the overlap with the optimal controls. Since imitation loss is the outer loss of the problem, the algorithm can be considered successful.

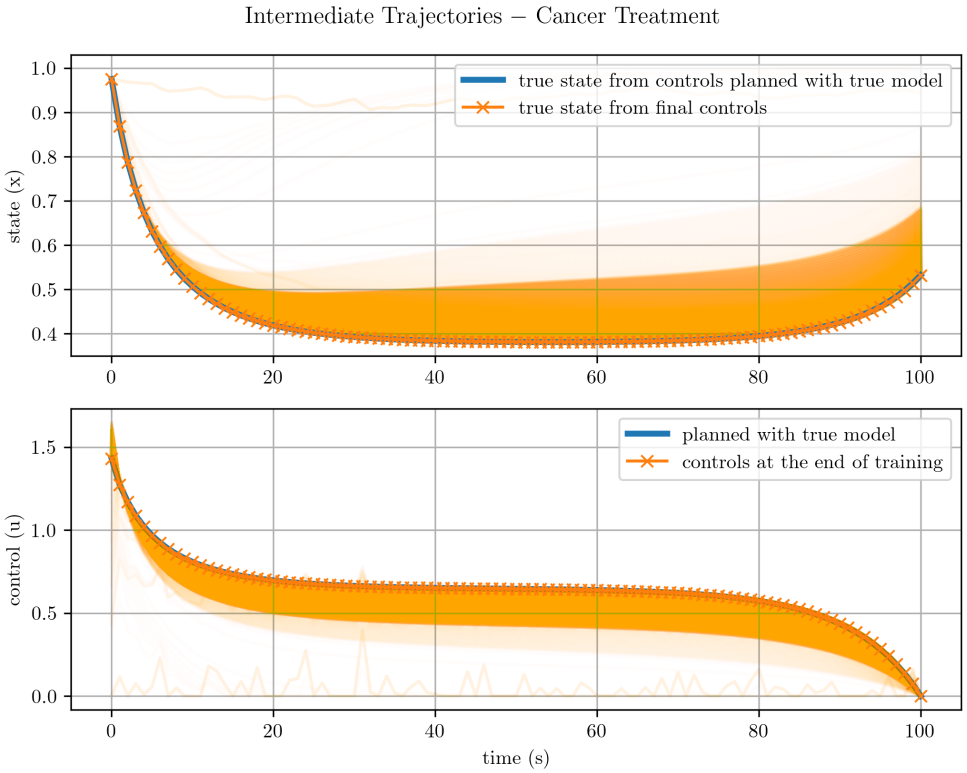


Fig. 5.2. Visualization of how the controls guess and corresponding state evolves over time when learning end-to-end on the Cancer Treatment domain. Each intermediate guess is plotted with the same high transparency value. Trajectories which occur more often during the training procedure therefore appear darker. The true optimal trajectory and final control (and corresponding state) guess are plotted with no transparency, on top of the guesses.

An interesting question to ask is what the learned model looks like. It was repeatedly used and updated during training; can we also use it for planning once training has finished? We answer this question with the plot in Figure 5.3: we see that when used in a trajectory optimization setting, the learned model performs well.

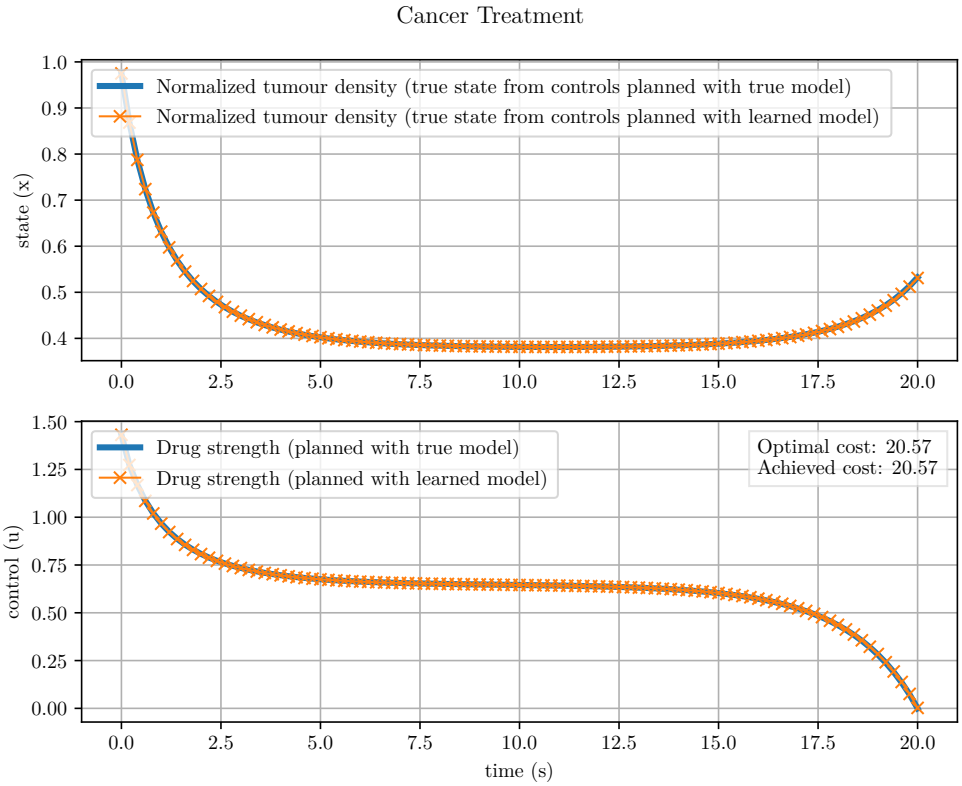
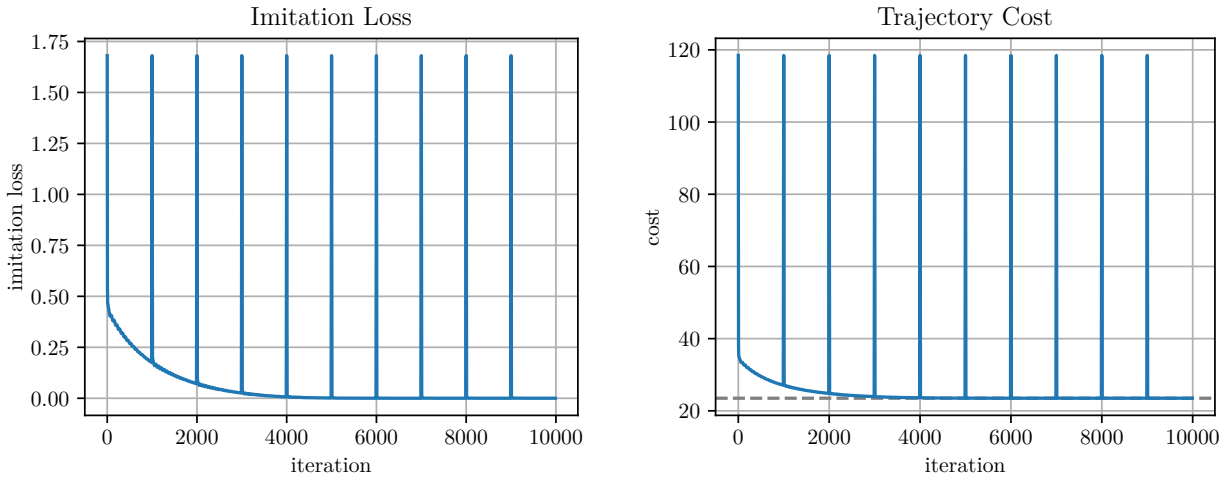


Fig. 5.3. Result of planning using the end-to-end learned model on the Cancer Treatment domain. Evaluated in the true system, we achieve the same performance as the true optimal controls.

Since we used a parametric model of the system dynamics, we can also report the learned parameters. They were δ: 0.4518 and r: 0.3011. This is quite close to the true parameters of δ: 0.45 and r: 0.3.

5.1.2. Mould Fungicide

We perform the same procedure in the Mould Fungicide domain. Figures 5.4a and 5.4b show imitation loss and performance of planned controls, as in the previous section.



(a) **(b)**

Fig. 5.4. We plot (a) the imitation loss and (b) the trajectory cost over the course of end-to-end training on the Mould Fungicide domain.

In Figure 5.5, we show how our guess of controls, and resulting state, changes during the training. Here too, we see that the algorithm converges to a trajectory very close to optimal.

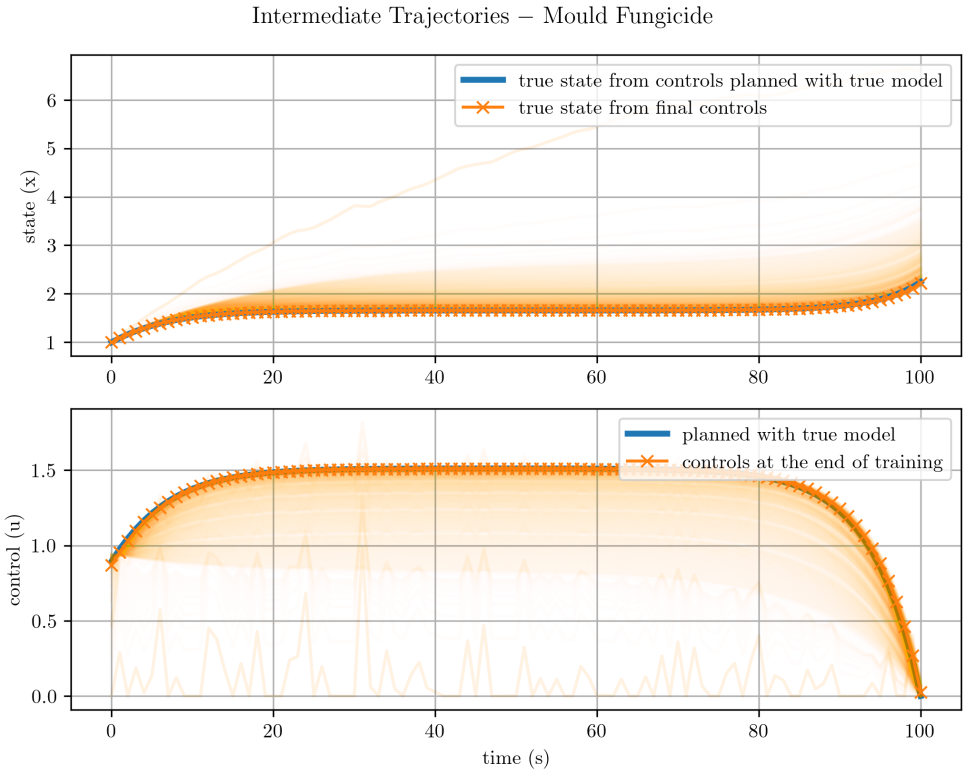


Fig. 5.5. Visualization of how the controls guess, and corresponding state, evolves over time when learning end-to-end on the Mould Fungicide domain.

Finally, Figure 5.6 shows the performance of using the learned model for planning.

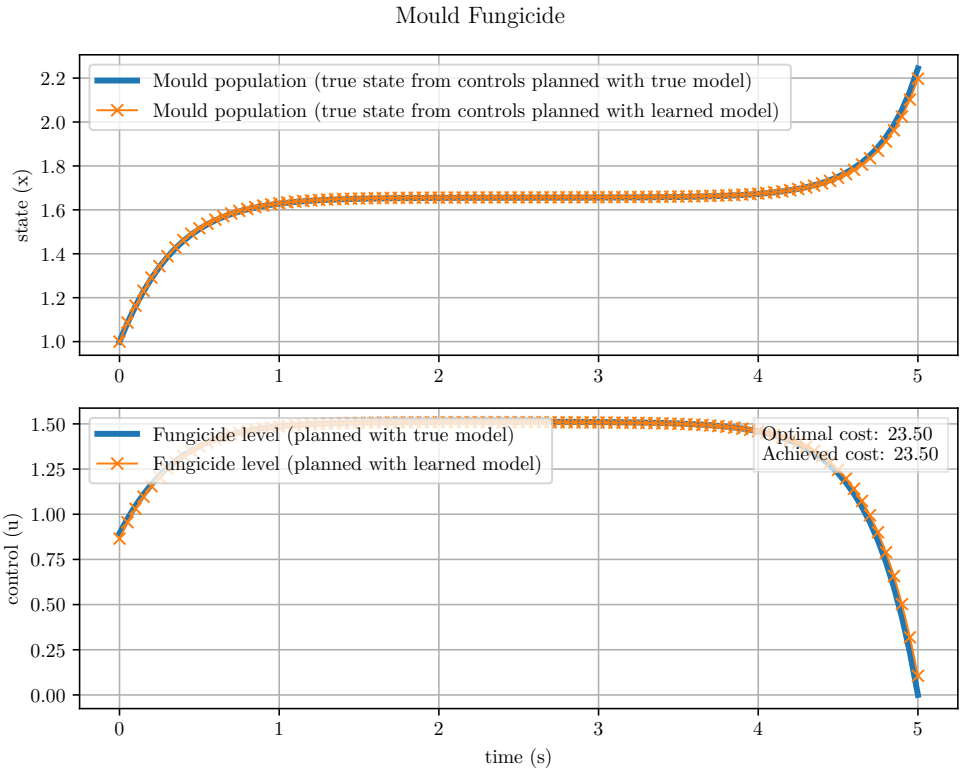


Fig. 5.6. Result of planning using the end-to-end learned model on the Mould Fungicide domain. Evaluated in the true system, we achieve the same performance as the true optimal controls.

The final learned parameters were r: 0.381 and M: 8.388. These are quite close to the true parameters of r: 0.3 and M: 10, but somewhat worse than those found when using the MLE approach. Of course, the parameter values are only a secondary concern – what matters is the performance of the controls at the end of training.

5.1.3. Other Domains

We record the performance of the end-to-end training approach with parametric models on a number of different domains. All learned models were trained end-to-end for 10000 iterations, with a lookahead horizon of 10 steps of extragradient, and a warm-start reset every 1000 iterations. The results are summarized in Table 5.1.

System	Parameters (true/learned)	Cost (best/achieved)	Defect (best/achieved)

Bacteria	A: 1/1.543, B: 1/1.660, C: 1/1.15	$-7.98/-4.83$	NA
Bear Populations	r: 0.1/0.165, K: 0.75/0.134, m_f: 0.5/0.195, m_p: 0.5/0.666	12.28/5606.96	NA
Bioreactor	G: 1/0.976, D: 1/1.007	$-1.39/-1.39$	NA
Cancer Treatment	δ: 0.45/0.452, r: 0.3/0.301	20.57/20.57	NA
Cart-Pole Swing-Up	g: 9.81/9.704, m_1: 1/0.553, m_2: 0.3/0.000, ℓ: 0.5/0.151	87.78/0.52	$\begin{bmatrix} 0/-1.59 \\ 0/-3.10 \\ 0/-0.56 \\ 0/-0.11 \end{bmatrix}$
Glucose	a: 1/0.570, b: 1/0.308, c: 1/0.600	1354.02/1354.48	NA
Mould Fungicide	r: 0.3/0.381, M: 10/8.388	23.50/23.50	NA
Mountain Car	g: 0.0025/0.0025, p: 0.0015/0.0015	8.57/8.57	$\begin{bmatrix} 0/0 & 0/0 \end{bmatrix}^\top$
Pendulum	g: 10/11.943, m: 1/0.701, ℓ: 1/1.194	25.75/25.49	$\begin{bmatrix} 0/-0.04 & 0/-0.04 \end{bmatrix}^\top$
Predator Prey	d_1: 0.1/0.117, d_2: 0.1/0.105	1.79/1.79	0/0
Timber Harvest	K: 1/0.456	$-5104.67/-3516.24$	NA
Tumour	ξ: 0.084/0.133, b: 5.85/4.51, d: 0.00873/$-$0.000542, G: 0.15/0.226, μ: 0.02/0.000	7571.67/7614.09	NA

Van Der Pol	a: 1/0.504	2.87/33.82	$\begin{bmatrix}0/1.18 & 0/-0.84\end{bmatrix}^\top$

Table 5.1. A summary of the performance of end-to-end learning with parametric models on a variety of environments. The first column lists the environment. The second column indicates the value of the parameters. The third column indicates the cost of applying the controls solved for using the parameters in that row, applied in the true environment. The fourth column shows the defect of the final state from the desired final state, if any.

5.2. Neural ODE Models

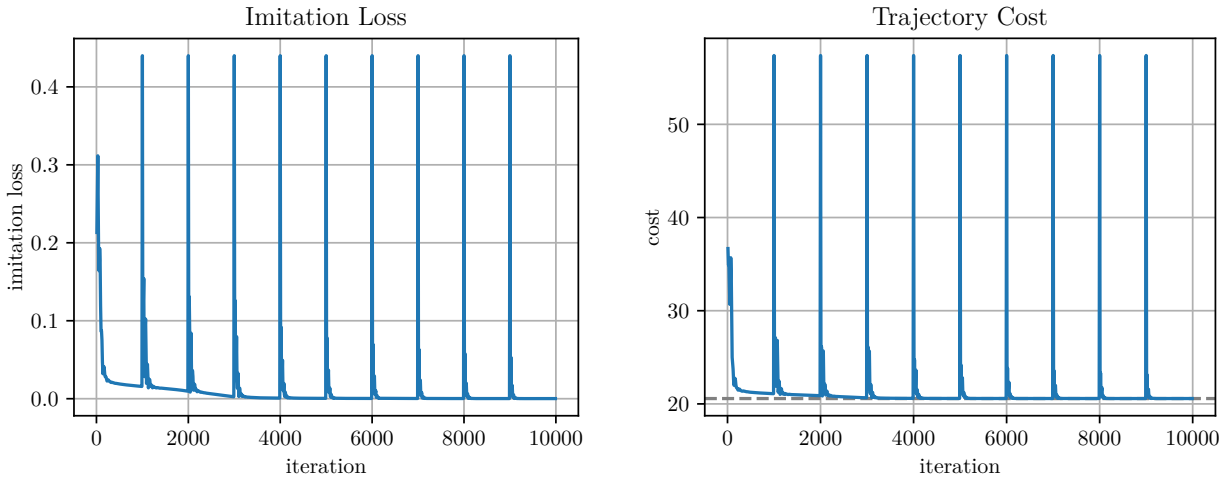
In this section, we will apply the same end-to-end learning approach, but with a much more general class of model, namely Neural ODEs. As in the previous experiments, we evaluate performance on several different domains.

5.2.1. Cancer Treatment

We instantiate a Neural ODE to match the Cancer Treatment domain, with 2D input, 1D output, and two hidden layers of 50 neurons each.

We randomly initialize the weights and biases, and repeatedly update our parameters by differentiating through 5 steps of unrolled Extragradient applied to the Lagrangian of the NLP resulting from the trajectory optimization problem. We used only 5 steps, down from 10, because of the additional time it takes to jit the pipeline, given that we are multiplying tensors of many more parameters than in the parametric case. Our initial experiments suggested that the number of lookahead steps did not have a large impact on the effectiveness of the learned model, only on compilation time. As in the previous section, we warm-start the decision variable guess, and reset the warm-start ever 1000 iterations. We train for a total of 10000 iterations.

In Figures 5.7a and 5.7b, we observe the imitation loss, and integrated cost of applying the current guess of controls. As we would hope, both these quantities decrease over the course of training.



(a) **(b)**

Fig. 5.7. We observe (a) the imitation loss and (b) the trajectory cost over the course of end-to-end Neural ODE model training on the Cancer Treatment domain.

We also visualize how our guess of controls, and corresponding state, changes over the course of training. This is shown in Figure 5.2. We see that the final control trajectory is very similar to the true optimal trajectory.

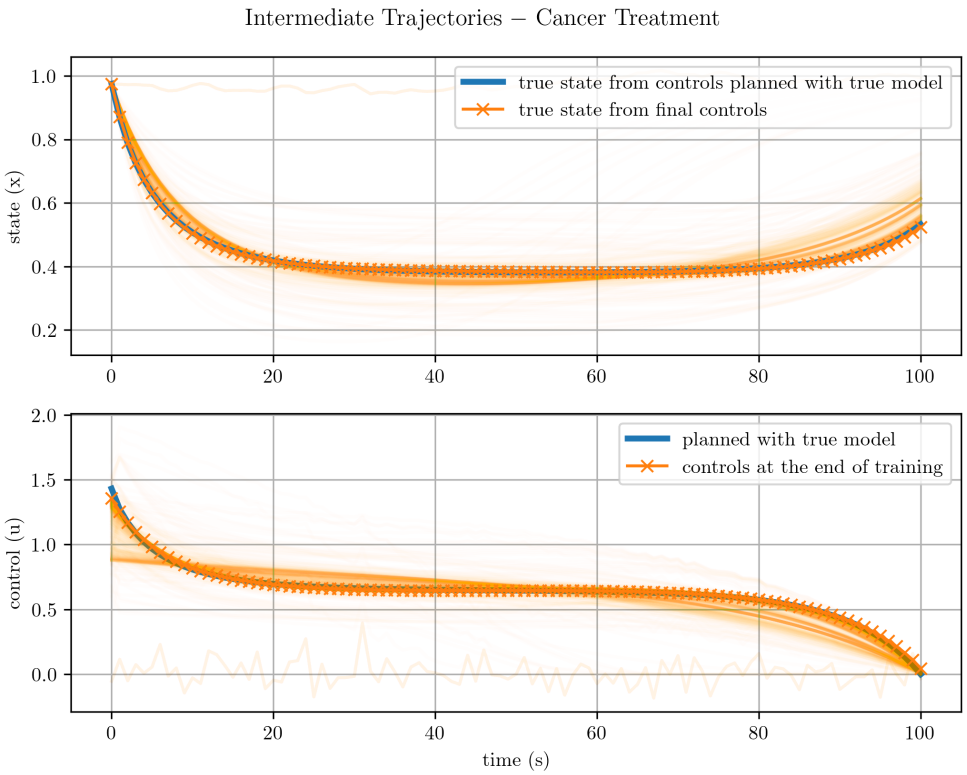


Fig. 5.8. Visualization of how the controls guess, and corresponding state, evolve over time when learning a Neural ODE model end-to-end on the Cancer Treatment domain.

As with the parametric models, we explore whether the final model can also be used for planning. In fact, planning works quite well, though not quite as well as in the parametric case. This is shown in Figure 5.9.

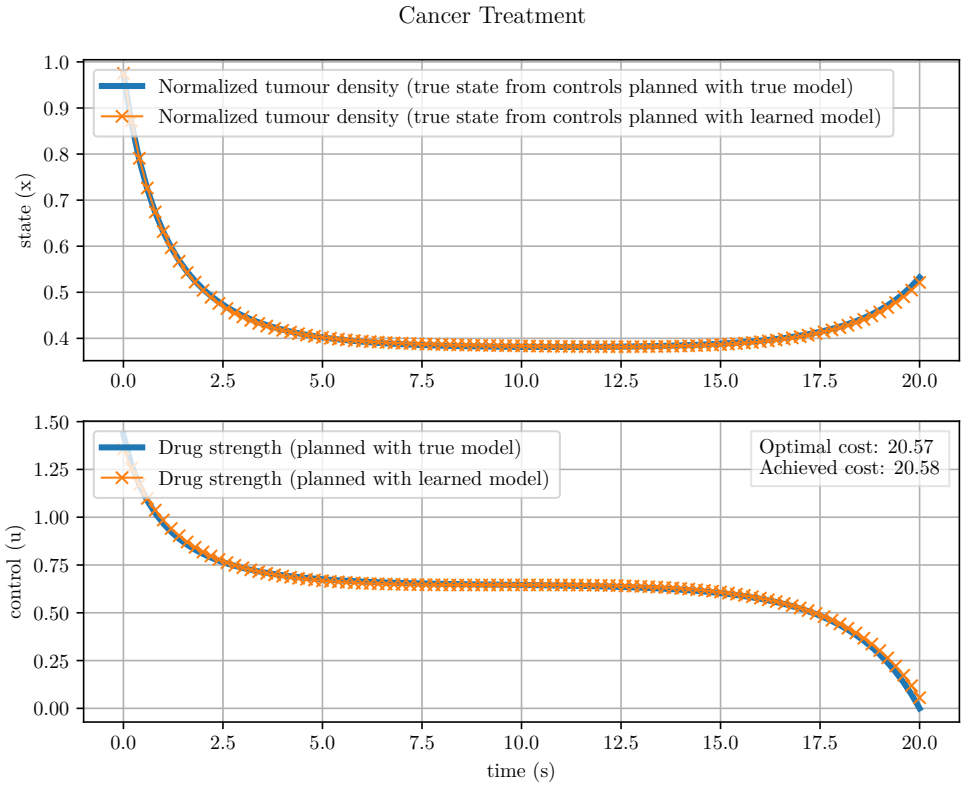
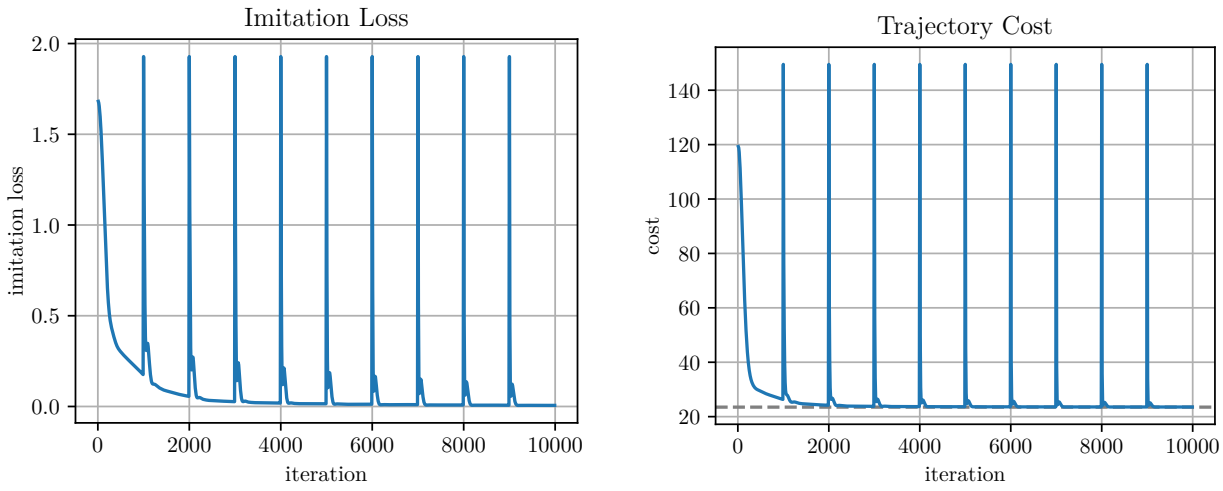


Fig. 5.9. Result of planning using the end-to-end learned Neural ODE model on the Cancer Treatment domain.

5.2.2. Mould Fungicide

We perform the same procedure on the Mould Fungicide domain, with the same hyper-parameters as in the Cancer Treatment experiments.

Figures 5.10a and 5.10b show the imitation loss and integrated cost of applying the current guess of controls. They decrease over time, suggesting the algorithm is learning well.



(a) **(b)**

Fig. 5.10. We plot (a) the imitation loss and (b) the trajectory cost over the course of end-to-end Neural ODE model training on the Mould Fungicide domain.

We visualize how the controls and state evolve over the course of training in Figure 5.11.

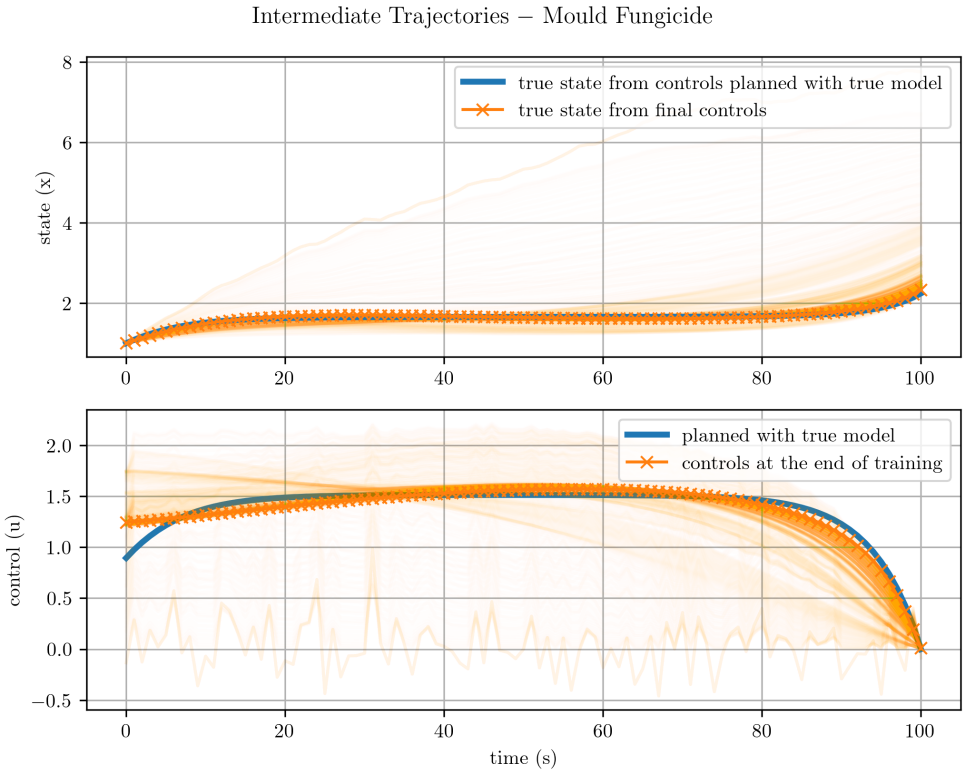


Fig. 5.11. Visualization of how the controls guess, and corresponding state, evolve over time.

Surprisingly, the first 20s of control look quite different from the true optimal control trajectory. It is unclear why the neural network had difficulty matching this part of the trajectory specifically. Despite this, the resulting state trajectory is very similar to the true optimal trajectory.

Finally, we also use the Neural ODE model for planning. Figure 5.12 shows the performance of the resulting control trajectory. Despite being somewhat different from the true optimal trajectory, it is only slightly worse in terms of cost, and results in a very similar state trajectory to the true optimal one.

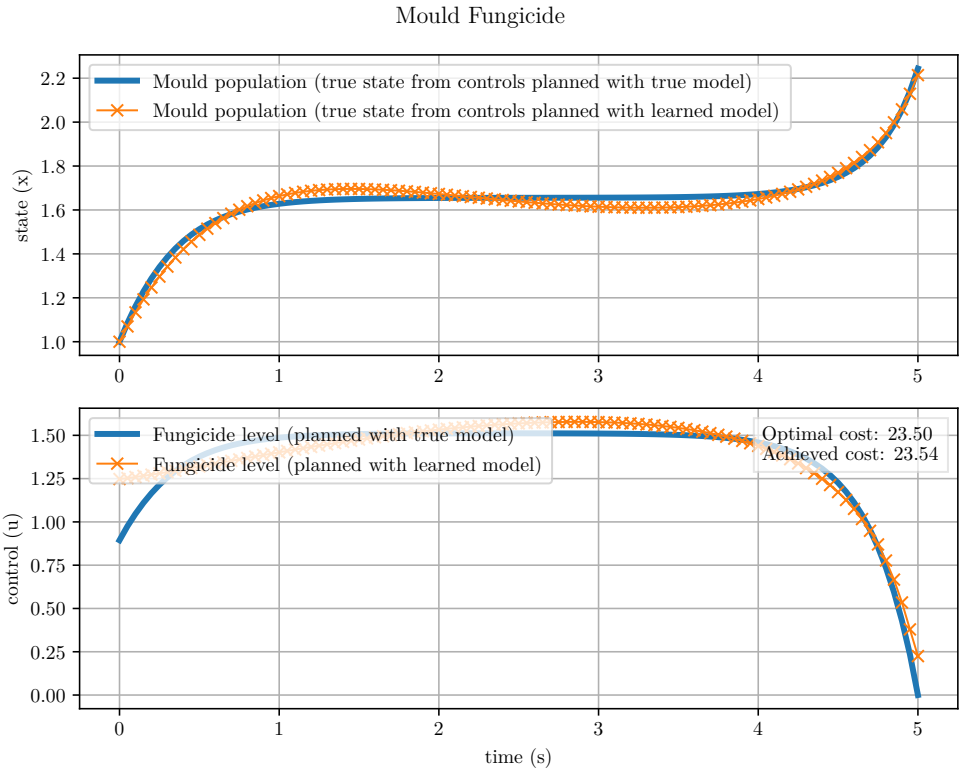


Fig. 5.12. Result of planning using the end-to-end learned Neural ODE model on the Mould Fungicide domain.

5.2.3. Other Domains

We explore the performance of the end-to-end Neural ODE algorithm on a number of different domains. All models were trained over up to 10000 iterations, with 5 lookahead steps of extragradient.

System	Cost (best/achieved)	Defect (best/achieved)	Neurons/Layer
Bacteria	$-7.98/-7.60$	NA	100
Bear Populations	12.28/12.40	NA	100
Bioreactor	$-1.39/-1.39$	NA	100
Cancer Treatment	20.57/20.58	NA	50
Cart-Pole Swing-Up	87.78/36.90	$\begin{bmatrix} 0/1.21 \\ 0/-4.88 \\ 0/-0.29 \\ 0/-3.28 \end{bmatrix}$	100
Glucose	1354.02/1354.12	NA	100
HIV Treatment	$-823.13/-822.96$	NA	100
Mould Fungicide	23.50/23.54	NA	50
Mountain Car	8.57/3000	$\begin{bmatrix} 0/-0.39 \\ 0/-0.02 \end{bmatrix}$	50
Pendulum	25.53/1.90	$\begin{bmatrix} 0/-2.69 \\ 0/-0.08 \end{bmatrix}$	100
Predator Prey	1.79/1.04	$0/-1.84$	100
Timber Harvest	$-5104.67/-500.00$	NA	50
Tumour	7571.67/9161.23	NA	50
Van Der Pol	2.87/16.81	$\begin{bmatrix} 0/0.54 \\ 0/-1.75 \end{bmatrix}$	100

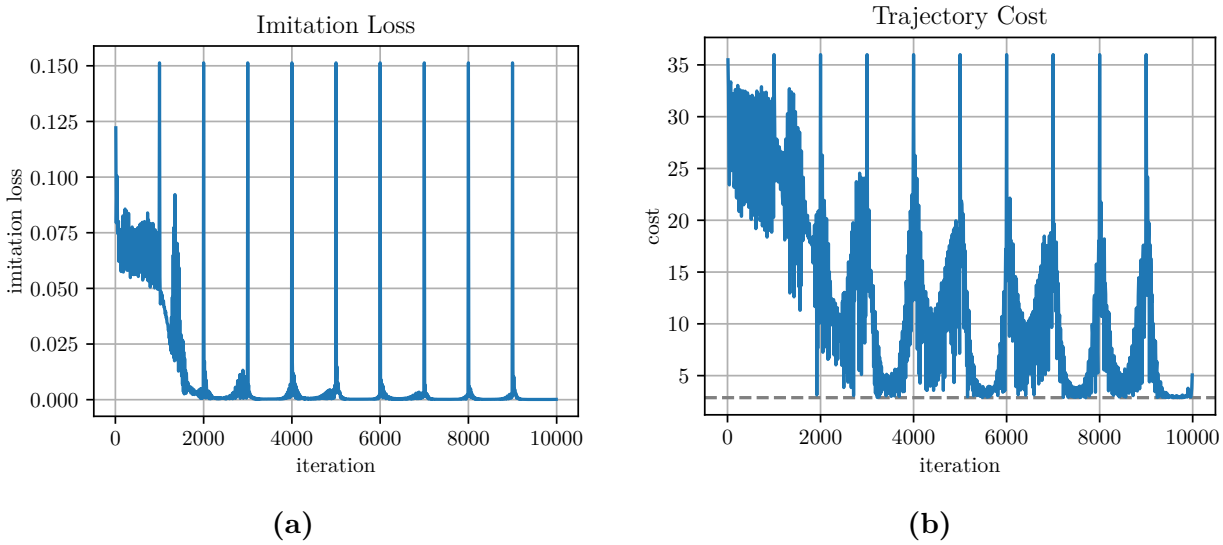
Table 5.2. Summary of performance of end-to-end learning and planning with Neural ODE models on a variety of environments. The first column lists the environment. The second column indicates the cost of applying the controls solved for with the model, applied in the true environment. The third column shows the defect of the final state from the desired final state, if any. The fourth column shows the size of the hidden layers of the neural network that was used for the model.

Note the presence of the HIV Treatment domain in this table. Unlike the previous three approaches discussed, the Neural ODE end-to-end method did not diverge during training on this domain. This suggests that it might be able to avoid numerical instabilities where other approaches fail.

At the same time, several environments proved particularly difficult for this approach. We will discuss some of them now.

5.2.4. Challenging Environments

On several domains with very nonlinear dynamics, the Neural ODE end-to-end approach did not fare well. This is likely for one of the same reasons that they struggled in the MLE setting: small model error can lead to large trajectory differences. An an example, we look at the performance achieved on the Van Der Pol domain. Figure 5.13 shows that the imitation loss goes to zero during training, suggesting that the algorithm learned well.



(a) **(b)**

Fig. 5.13. We observe (a) the imitation loss and (b) the trajectory cost over the course of end-to-end training of a Neural ODE model on the Van Der Pol domain.

Indeed, as shown in Figure 5.14 the control trajectory found during the algorithm is very close to the true optimal trajectory. However, because of the highly nonlinear dynamics of the Van Der Pol system, the resulting trajectories is very different from that achieved by the slightly different true optimal controls.

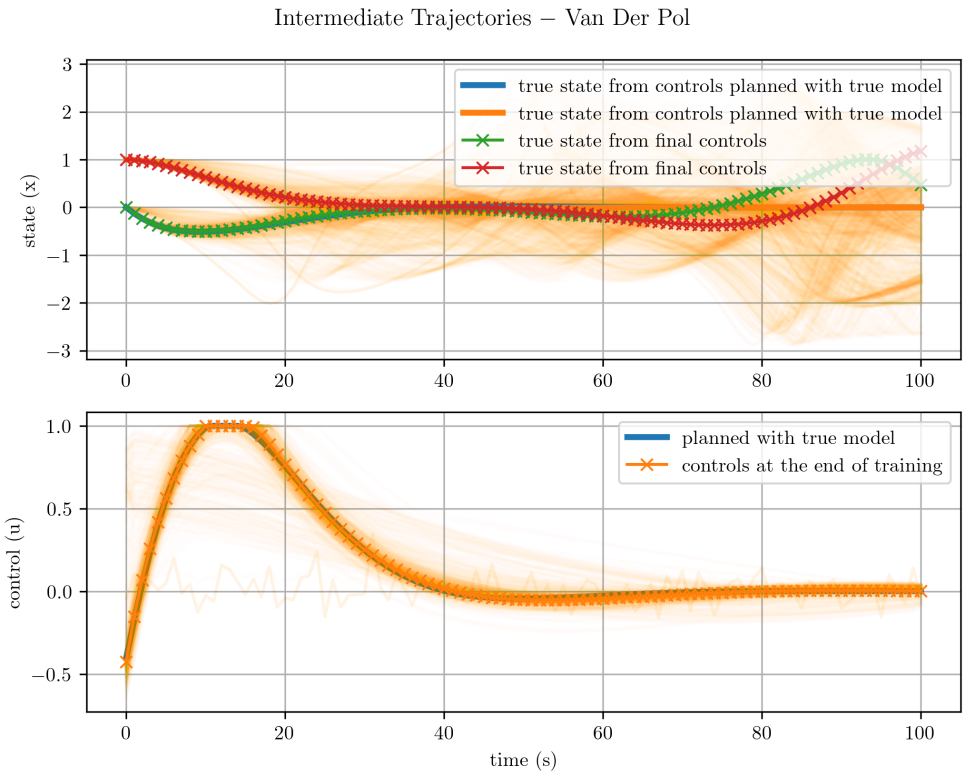


Fig. 5.14. Visualization of how the controls guess, and corresponding state, evolve over the course of end-to-end training of a Neural ODE model on the Van Der Pol domain.

The same dilemma is faced when using the model for planning, as shown in Figure 5.15. As mentioned in Section 4.2.4, one practical solution would be to use MPC.

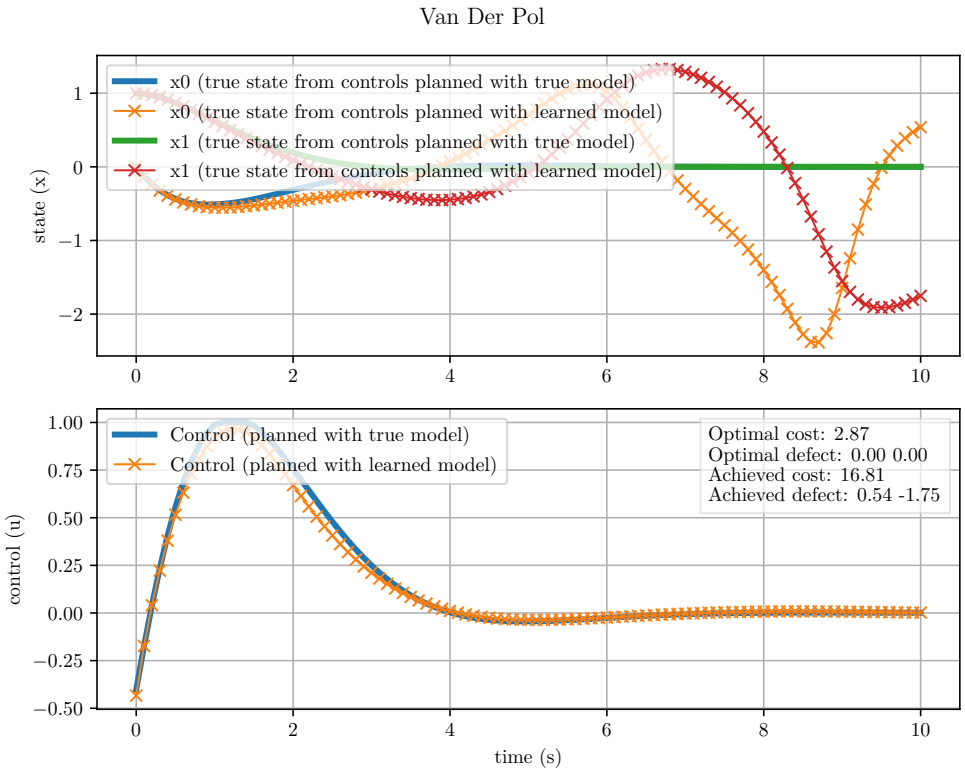


Fig. 5.15. Result of planning using the end-to-end learned Neural ODE model on the Van Der Pol domain.

An additional challenge experienced in training the end-to-end Neural ODE models was that of vanishing gradients. In several different domains, the gradients vanished after several tens of iterations. It seems that the architecture is somewhat sensitive to hyperparameters, including network size. One interesting way to address the vanishing gradient problem would be to pre-train the neural network in an MLE fashion. Then, with this initialization, we could apply the end-to-end approach for further refinement. This could also work in a setting where we first train an MLE model in a simulator, and then refine it end-to-end once applied to the real-world task.

5.2.5. Dynamics Study

As in the MLE case, we cannot directly compare the learned parameters of our Neural ODEs with the true model parameters. In order to get an idea of how the learned dynamics compare with the true dynamics, we can plot vector fields.

These are shown for the Cancer Treatment environment in Figure 5.16, and for the Mould Fungicide environment in Figure 5.17.

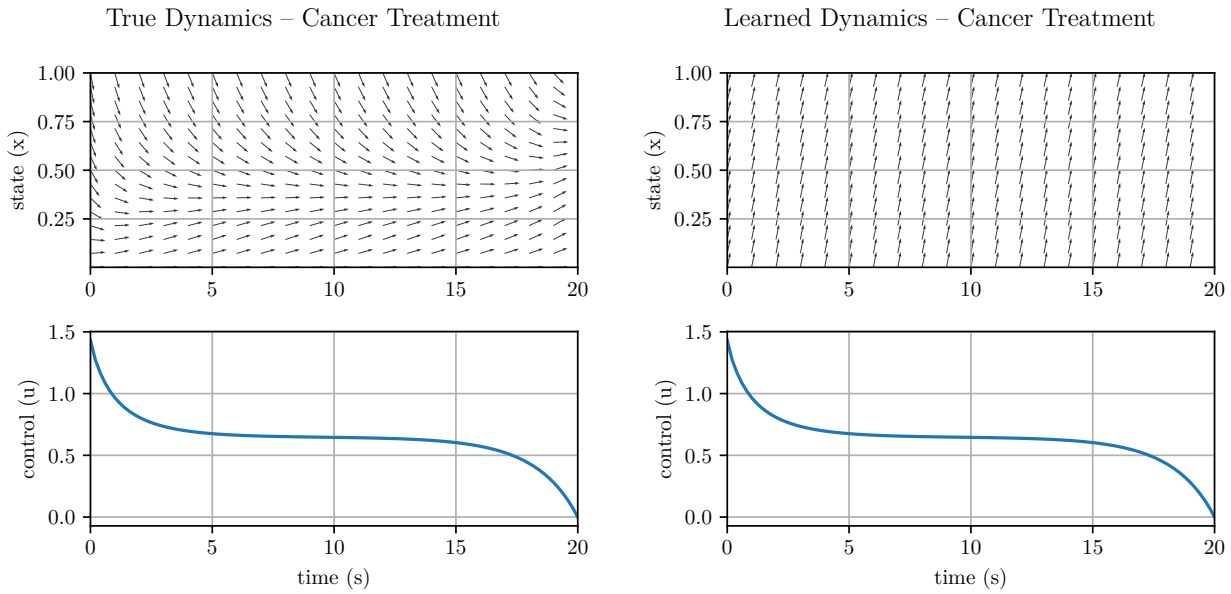


Fig. 5.16. Comparison of (a) true dynamics and (b) end-to-end learned Neural ODE dynamics when applying the true optimal controls in the Cancer Treatment domain.

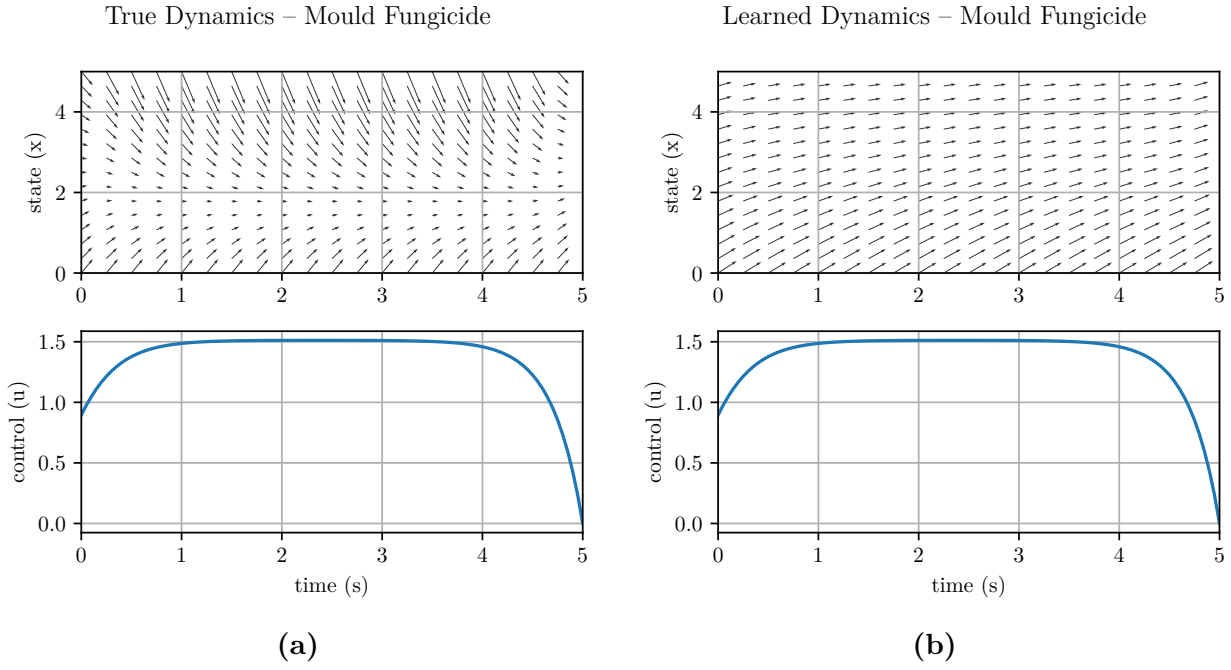


Fig. 5.17. Comparison of (a) true dynamics and (b) end-to-end learned Neural ODE dynamics when applying the true optimal controls in the Mould Fungicide domain.

As both figures show, the dynamics learned by the end-to-end approach differ significantly from the true dynamics. This is in contrast with the dynamics learned with the MLE approach, which looked much more similar to the true dynamics.

Because of the way that the end-to-end models were trained, this is not overly surprising. Since the algorithm was instructed to minimize the imitation loss on the controls, there was no penalty for learning an unrealistic dynamics model – the only role of the dynamics model is to lead to the same controls while using the model for planning over the course of training. In general, we can expect there to be many dynamics models which all lead to the same planned controls, yet only one of them corresponds to the true model. It would be interesting to compare this approach with a similar approach in which we perform imitation learning on the optimal state trajectory instead of the optimal control trajectory. In this case, it seems likely that the learned dynamics model would be much closer to the true dynamics.

5.3. Summary

In this chapter, we explored the problem setting of simultaneously learning, and planning with, a dynamics function, in an end-to-end fashion. We found the task to be somewhat more challenging than the MLE approach of the previous chapter. Despite this, in both the parametric and Neural ODE case, several models achieved excellent performance. We also explored some of the cases where training did not perform well, along with possible explanations and solutions, such as MPC and pre-training.

Chapter 6

Conclusion

In this thesis we presented the groundwork for building machine learning architectures which couple the expressiveness of Neural ODEs with model-based planning techniques from optimal control. We presented several approaches, building up to a Neural ODE-based model learning and planning algorithm which can be trained in a fully end-to-end fashion. With the construction and presentation of the Myriad repository, we saw that many relevant real-world settings can be modeled by systems of ODEs, and thus that this problem formulation is useful for a number of practical applications.

We focused on two general approaches to the SysID and control problems. In Chapter 4, we used a modular approach, separating the system identification and trajectory optimization into two distinct phases. First, we generated training trajectories by applying sequences of controls in the true model. With these data, we fit our model parameters in a maximum likelihood fashion, building a model that performed well at predicting the state trajectories. Then, we used the learned model in a classical trajectory optimization setting, applying techniques such as single shooting to choose the optimal control trajectory. Depending on the performance of our learned model, in the more complex environments, we iteratively added data to our training and validation sets. Here we saw various possible techniques of choosing controls to generate new trajectories.

In Chapter 5, we merged learning dynamics and planning controls together into an end-to-end trainable formulation. At each step of the algorithm, we partially solved a trajectory optimization problem, while also updating the model parameters to result in better future planning. We discussed different ways of differentiating through the optimal control problem, and presented a practical implementation which provided a compromise between speed and precision.

In parallel with the two learning formulations, we also considered different model classes. To begin with, we learned the parameters of a dynamics model of fixed parametric form. In several cases of both the MLE and end-to-end learning formulations, we were able to recover

the exact parameter values of the true models. In others, the parameter values were not exactly the same as the true ones, but were still useful in the planning process. It was in this context that we introduced a heuristic to overcome local minima, by applying a discount factor, dependent on both timestep and epoch, to the loss function.

We then explored the use of much more general models for describing continuous-time dynamics, in the form of Neural ODEs. Due to their more general expressive power, we found that Neural ODEs were able to consistently make accurate predictions in many of our test environments. At the same time, as is to be expected, we found that they required more data in order to effectively learn dynamics, and even then, struggled to accurately model difficult-to-reach portions of state space. In the fully end-to-end setting, some environments also led to vanishing gradients part-way through the training procedure.

Overall, this thesis laid the groundwork for training algorithms to learn and plan in continuous-time environments. Furthermore, it proposed a Neural ODE and extragradient-based approach for end-to-end training a network which could be used to provide an inductive bias for behaving optimally in a dynamical system.

6.1. Future Work

This work represents only the first steps into a wide array of related research. Practically every aspect of our approach could be studied and improved individually. At the same time, modifications to the model architecture and gradient pipeline could also lead to significant improvements in performance. Some specific examples follow.

6.1.1. Technical Directions

It would be desirable to have an adaptive numerical integration technique, to take advantage of regions of low curvature. By including a penalty for unnecessarily complicated dynamics as presented in [**Kelly et al., 2020**], this could lead to faster integration times. A challenge is that this integration technique must be very fast to differentiate through.

To improve the stability of the dynamics, it would be desirable to implement variable scaling as we construct the NLP.

So far, we have only explored using general-purpose NLP solvers, as well as extragradient, which is a simple first-order algorithm. It would be beneficial to write our own high-performance NLP solver. Specifically, we could take advantage of the structure arising from shooting and collocation formulations of the trajectory optimization problem. Furthermore, we could optimize the solve to take advantage of the GPUs we are using for training the underlying Neural ODE.

6.1.2. Conceptual Directions

While we presented various approaches to solving trajectory optimization problems, the experiments only made use of one, namely single shooting. Indeed, replacing Neural ODEs with a parallelized version has been studied with some success in [**Massaroli et al., 2021**]. Initial results suggest that in our setting of a large NLP, multiple shooting and collocation might be harder to get working due to the large number of resulting inequality constraints, and the resulting larger number of steps necessary for extragradient to converge. However, using these highly parallel methods in place of the sequential single shooting would lead to much shorter backpropagation horizons, which might help overcome the vanishing gradients experienced in some settings. Furthermore, this parallelization could also lead to significant improvements in computation time, unlocking the application of these algorithms in higher data regimes.

This work focused on learning dynamics functions from data and/or interaction with the environment, while keeping a fixed cost function. As is done in [**Jin et al., 2020**], it would be interesting to expand this technique to learning cost functions as well.

It would be good to explore the performance of these algorithms in an RL context, where we do not have access to analytic gradients and instead need to perform derivative estimation. For example, we could consider a setting in which the Neural ODE-based agent learns to maximize the return given by the distribution over trajectories induced by the joint interaction of the NLP solver, the learned dynamics, and the true dynamics.

The problem of Optimum Experiment Design – how to choose controls to best learn the dynamics in areas of state space which are hard to access – remains an interesting open question.

It would be interesting to explore the performance of other methods for escaping local minima, which are specifically suited for time-series prediction. In conjunction with this, it would be beneficial to switch from using mean squared error to a more informative loss function, which also takes into account the temporal nature of the trajectories.

Finally, an important future step will be to explore the performance of our learned models in the setting of model predictive control. Indeed, even imperfect models may achieve drastically better performance when replanning is allowed. Furthermore, an improvement to the approach could be to encode the fact that the model will be used in MPC into the learning pipeline itself, thus encouraging the learned representations to be specifically useful for MPC.

Bibliography

[Açıkmeşe et al., 2013] Açıkmeşe, B., Carson, J. M., and Blackmore, L. (2013). Lossless convexification of nonconvex control bound and pointing constraints of the soft landing optimal control problem. *IEEE Transactions on Control Systems Technology*, 21(6):2104–2113.

[Alvarez et al., 2020] Alvarez, V. M. M., Rosca, R., and Falcutescu, C. G. (2020). Dynode: Neural ordinary differential equations for dynamics modeling in continuous control. *arXiv preprint arXiv:2009.04278*.

[Amos et al., 2018] Amos, B., Rodriguez, I. D. J., Sacks, J., Boots, B., and Kolter, J. Z. (2018). Differentiable mpc for end-to-end planning and control. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 8299–8310, Red Hook, NY, USA. Curran Associates Inc.

[Antony, 2018] Antony, T. (2018). *Large Scale Constrained Trajectory Optimization Using Indirect Methods*. PhD thesis, Purdue University.

[Aycock, 2003] Aycock, J. (2003). A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113.

[Bailer-Jones et al., 1998] Bailer-Jones, C. A., MacKay, D. J., and Withers, P. J. (1998). A recurrent neural network for modelling dynamical systems. *network: computation in neural systems*, 9(4):531.

[Bellemare et al., 2013] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

[Bertsekas, 1999] Bertsekas, D. P. (1999). *Nonlinear programming*. Athena Scientific.

[Bertsimas and Tsitsiklis, 1997] Bertsimas, D. and Tsitsiklis, J. (1997). *Introduction to Linear Optimization*. Athena Scientific, 1st edition.

[Betts, 2010] Betts, J. T. (2010). *Practical methods for optimal control and estimation using nonlinear programming*. Siam.

[Bliss, 1980] Bliss, G. A. (1980). *Lectures on the calculus of variations*. University of Chicago Press.

[Boyd and Vandenberghe, 2004] Boyd, S. and Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

[Burden et al., 2016] Burden, R. L., Faires, J. D., and Burden, A. M. (2016). *Numerical analysis*. Cengage Learning.

[Butcher, 2016] Butcher, J. C. (2016). *Numerical methods for ordinary differential equations*. Wiley.

[Byrd et al., 2010] Byrd, R. H., Curtis, F. E., and Nocedal, J. (2010). An inexact newton method for nonconvex equality constrained optimization. *Math. Program.*, 122(2):273–299.

[Bürger and Lago Garcia, 2014] Bürger, A. and Lago Garcia, J. (2014). Casadi interface for optimum experimental design and parameter estimation and identification applications.

[Camacho and Alba, 2013] Camacho, E. F. and Alba, C. B. (2013). *Model predictive control*. Springer science & business media.

[Chen et al., 2018] Chen, T. Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D. (2018). Neural ordinary differential equations. *CoRR*, abs/1806.07366.

[Connor et al., 1994] Connor, J., Martin, R., and Atlas, L. (1994). Recurrent neural networks and robust time series prediction. *IEEE Transactions on Neural Networks*, 5(2):240–254.

[Dewey, 2014] Dewey, D. (2014). Reinforcement learning and the reward engineering principle. In *2014 AAAI Spring Symposium Series*.

[Dulac-Arnold et al., 2020] Dulac-Arnold, G., Levine, N., Mankowitz, D. J., Li, J., Paduraru, C., Gowal, S., and Hester, T. (2020). An empirical investigation of the challenges of real-world reinforcement learning. *arXiv preprint arXiv:2003.11881*.

[Fukushima, 1988] Fukushima, K. (1988). Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119–130.

[Funahashi and Nakamura, 1993] Funahashi, K.-i. and Nakamura, Y. (1993). Approximation of dynamical systems by continuous time recurrent neural networks. *Neural networks*, 6(6):801–806.

[Gevers, 2005] Gevers, M. (2005). Identification for control: From the early achievements to the revival of experiment design. *European journal of control*, 11(4-5):335–352.

[Gidel et al., 2019] Gidel, G., Hemmat, R. A., Pezeshki, M., Le Priol, R., Huang, G., Lacoste-Julien, S., and Mitliagkas, I. (2019). Negative momentum for improved game dynamics. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1802–1811. PMLR.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

[Griewank, 2003] Griewank, A. (2003). A mathematical view of automatic differentiation. *Acta Numerica*, 12:321–398.

[Gu et al., 2016] Gu, S., Lillicrap, T., Sutskever, I., and Levine, S. (2016). Continuous deep q-learning with model-based acceleration. In Balcan, M. F. and Weinberger, K. Q., editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2829–2838, New York, New York, USA. PMLR.

[Jin et al., 2020] Jin, W., Wang, Z., Yang, Z., and Mou, S. (2020). Pontryagin differentiable programming: An end-to-end learning and control framework. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

[Kamijo and Tanigawa, 1990] Kamijo, K. and Tanigawa, T. (1990). Stock price pattern recognition-a recurrent neural network approach. In *1990 IJCNN International Joint Conference on Neural Networks*, pages 215–221 vol.1.

[Keesman, 2011] Keesman, K. J. (2011). *System identification: an introduction*. Springer Science & Business Media.

[Kelly et al., 2020] Kelly, J., Bettencourt, J., Johnson, M. J., and Duvenaud, D. (2020). Learning differential equations that are easy to solve. *CoRR*, abs/2007.04504.

[Kelly, 2017] Kelly, M. (2017). An introduction to trajectory optimization: How to do your own direct collocation. *SIAM Review*, 59(4):849–904.

[Kidger et al., 2020] Kidger, P., Morrill, J., Foster, J., and Lyons, T. (2020). Neural controlled differential equations for irregular time series. *arXiv preprint arXiv:2005.08926*.

[Kim et al., 2021] Kim, S., Ji, W., Deng, S., Ma, Y., and Rackauckas, C. (2021). Stiff neural ordinary differential equations.

[Kingma and Ba, 2015] Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y., editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[Koh et al., 2021] Koh, P. W., Sagawa, S., Marklund, H., Xie, S. M., Zhang, M., Balsubramani, A., Hu, W., Yasunaga, M., Phillips, R. L., Gao, I., Lee, T., David, E., Stavness, I., Guo, W., Earnshaw, B. A., Haque, I. S., Beery, S., Leskovec, J., Kundaje, A., Pierson, E., Levine, S., Finn, C., and Liang, P. (2021). WILDS: A benchmark of in-the-wild distribution shifts. In *International Conference on Machine Learning (ICML)*.

[Koller et al., 1996] Koller, D., Megiddo, N., and Von Stengel, B. (1996). Efficient computation of equilibria for extensive two-person games. *Games and economic behavior*, 14(2):247–259.

[Korpelevi, 1976] Korpelevi, G. (1976). An extragradient method for finding saddle points and for other problems, konom. i mat.

[Kroer, 2020] Kroer, C. (2020). Computing nash equilibrium via regret minimization.

[LeCun et al., 1989] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551.

[Lehmann and Casella, 2006] Lehmann, E. L. and Casella, G. (2006). *Theory of point estimation*. Springer Science & Business Media.

[Lenhart and Workman, 2007] Lenhart, S. and Workman, J. T. (2007). *Optimal control applied to biological models*. Chapman Hall/CRC.

[Li and Todorov, 2004] Li, W. and Todorov, E. (2004). Iterative linear quadratic regulator design for non-linear biological movement systems. In *ICINCO (1)*, pages 222–229. Citeseer.

[Lorin, 2020] Lorin, E. (2020). Derivation and analysis of parallel-in-time neural ordinary differential equations. *Annals of Mathematics and Artificial Intelligence*, 88(10):1035–1059.

[Massaroli et al., 2021] Massaroli, S., Poli, M., Sonoda, S., Suzuki, T., Park, J., Yamashita, A., and Asama, H. (2021). Differentiable multiple shooting layers. *CoRR*, abs/2106.03885.

[Mayne, 1966] Mayne, D. (1966). A second-order gradient method for determining optimal trajectories of non-linear discrete-time systems. *International Journal of Control*, 3(1):85–95.

[Mitchell, 1980] Mitchell, T. M. (1980). *The need for biases in learning generalizations*. Department of Computer Science, Laboratory for Computer Science Research ....

[Mitter, 1966] Mitter, S. (1966). Successive approximation methods for the solution of optimal control problems. *Automatica*, 3(3-4):135–149.

[Mnih et al., 2015a] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015a). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

[Mnih et al., 2015b] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015b). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

[Moerland et al., 2020] Moerland, T. M., Broekens, J., and Jonker, C. M. (2020). Model-based reinforcement learning: A survey. *CoRR*, abs/2006.16712.

[Niculae, 2020] Niculae, V. (2020). Optimizing with constraints: reparametrization and geometry.

[Nocedal and Wright, 2006] Nocedal, J. and Wright, S. (2006). *Numerical Optimization: Springer Series in Operations Research and Financial Engineering*. Springer.

[Okada et al., 2017] Okada, M., Rigazio, L., and Aoshima, T. (2017). Path integral networks: End-to-end differentiable optimal control. *arXiv preprint arXiv:1706.09597*.

[Ortega and Rheinboldt, 2000] Ortega, J. M. and Rheinboldt, W. C. (2000). *Iterative solution of nonlinear equations in several variables*. SIAM.

[Pereira et al., 2018] Pereira, M., Fan, D. D., An, G. N., and Theodorou, E. (2018). Mpc-inspired neural network policies for sequential decision making. *arXiv preprint arXiv:1802.05803*.

[Quaglino et al., 2019] Quaglino, A., Gallieri, M., Masci, J., and Koutník, J. (2019). Snode: Spectral discretization of neural odes for system identification. *arXiv preprint arXiv:1906.07038*.

[Roesch et al., 2021] Roesch, E., Rackauckas, C., and Stumpf, M. P. (2021). Collocation based training of neural ordinary differential equations. *Statistical Applications in Genetics and Molecular Biology*, 20(2):37–49.

[Schulman et al., 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

[Spivak, 2018] Spivak, M. (2018). *Calculus on manifolds: a modern approach to classical theorems of advanced calculus*. CRC press.

[Sutton, 1990] Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine learning proceedings 1990*, pages 216–224. Elsevier.

[Tuor et al., 2020] Tuor, A., Drgona, J., and Vrabie, D. (2020). Constrained neural ordinary differential equations with stability guarantees.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

[Vinyals et al., 2019] Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. (2019). Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354.

[Williams, 1992] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256.

[Wächter and Laird, 2005] Wächter, A. and Laird, C. (2005). Ipopt.