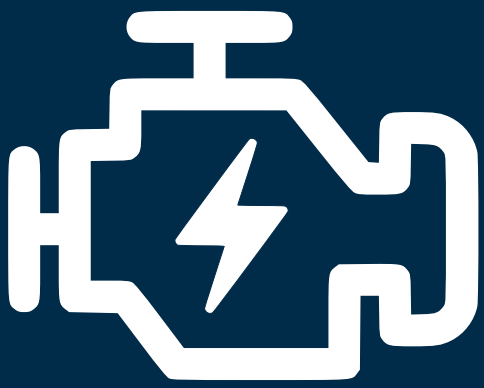
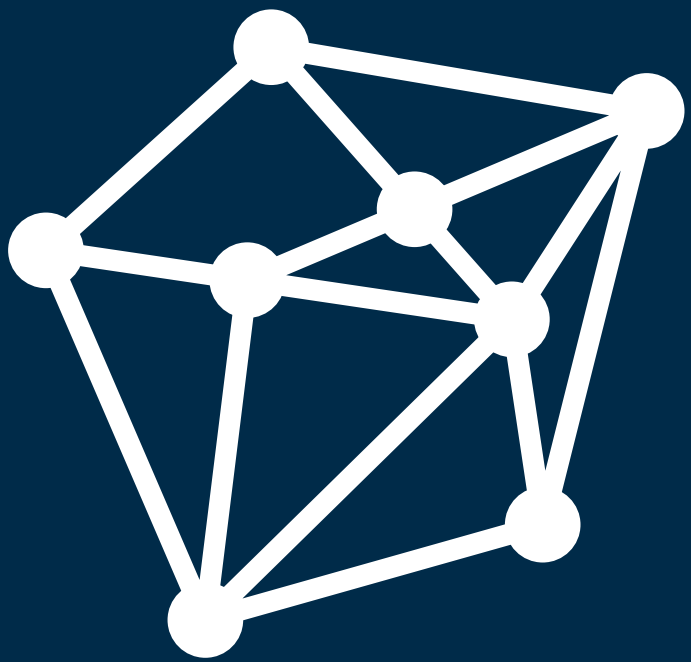
Modern Dynamic Programming Algorithms for Optimal Control Problems in Hybrid Electric Vehicles





Federico Miretti

Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Energy Engineering (34^{th} cycle)

Modern Dynamic Programming Algorithms for Optimal Control Problems in Hybrid Electric Vehicles

Federico Miretti

Supervisor:
Prof. Daniela Anna Misul, Supervisor
Prof. Ezio Spessa, Co-Supervisor

Doctoral Examination Committee:
...

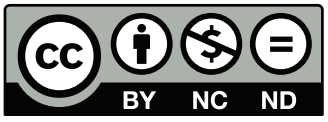
Politecnico di Torino
2022

Declaration

I hereby declare that the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Federico Miretti
2022

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).



Abstract

This thesis deals with the development of computer algorithms and their software implementation for the design of optimal *energy management strategies* (EMSs) for *hybrid electric vehicles* (HEVs).

Currently, optimal control techniques for EMS design are unable to handle complex simulation models because they are computationally over demanding. This limits the ability of automotive manufacturers and researchers to explore the complexity of hybrid electric powertrains and therefore to fully exploit their benefits. The underlying goal of this work is to overcome these shortcomings by developing optimal control methods suitable for higher-fidelity models. This goal is pursued in two research branches, corresponding to the two parts that compose this thesis.

The first part is centered around the well-established technique of dynamic programming. First, an open-source MATLAB toolbox for dynamic programming is developed. The toolbox includes state-of-the-art methods to overcome the potential numerical issues of the technique which typically arise in practical implementations. Secondly, we shift our focus from the algorithmic aspects to the modeling aspects to investigate the interaction between powertrain modeling choices and the algorithm. We conduct a systematic analysis and define ad hoc evaluation criteria. We then develop a case study and we perform extensive numerical experiments to support our analysis and we conclude with a set of recommendations that can be drawn from the evidence. Both of these contributions constitute an improvement to the existing practice of optimal EMS design with dynamic programming.

The second part is centered around a less known but promising technique called differential dynamic programming, with the prospect of overcoming the curse of dimensionality while keeping the benefit of guaranteed optimality. The relevant literature is reviewed to lay out the theoretical foundations of the algorithm. The theory is then used to develop a software implementation making use of modern computational techniques and tools to enable the technique's application to a real-life engineering problem, which constitutes another major contribution of this thesis. Finally, a EMS design application for a series hybrid powertrain is presented using the software, in order to test its robustness and computational capabilities. The results show great promise towards the ambitious goal of developing a broad-purpose EMS design tool capable of handling high-fidelity simulation models.

Contents

List of figures

List of tables

Prologue

1 Introduction

In this chapter, we attempt to briefly frame the issue of optimal *energy management strategies* (EMS) design for *hybrid electric vehicles* (HEVs) and outline a map for navigating this thesis. The reader who is already familiar with EMS design and the most common techniques developed to address it can confidently skip to Part I.

First and foremost, we need to set a starting point for our discussion. Here, we assume that the reader is already well accustomed to the importance of developing solid technology to enable the green transition of the transport sector and the relevance of hybrid electric vehicles to this purpose. What is surely less widely known to the general audience[†] is the importance for HEVs of an energy management strategy to control the various powertrain components so as to meet the driver's power demand while satisfying complex physical constraints associated to the powertrain components, all the while minimizing some important objective such as CO_2 or pollutants emissions, or both.

[†] We refer here to a general audience among engineers and researchers working in the transport sector.

Before we move on, let's start with an informal definition of the energy management strategy. In a conventional vehicle, the driver uses the accelerator pedal to control the acceleration of the vehicle; the pedal position can be converted into a torque demand at the engine and a low-level controller called an *engine control unit* then controls the various engine actuators (injectors, valves, spark plugs, etc.) to meet this demand. Similarly, for a *battery electric vehicle*, the torque demand is fed to a *motor control unit* which controls the power electronics of the electric motor in order to meet this demand.

In an HEV, where there are at least two sources of torque, these low-level control units are not enough. A high-level control unit must also be designed to decide how to satisfy the driver's torque demand using the power sources available, as visualized in Figure 1.1. This controller implements what is generally called the energy management strategy[‡]. For example, in a simple parallel hybrid, the EMS must decide whether to split the torque demand between the engine and the e-machine, satisfy it with one of them only, or use the engine to satisfy it while also providing additional power to the em-machine in order to charge the battery.

[‡] Other common names are *supervisory control* and *hybrid control unit* (HCU).

In doing this, the EMS must take into account the physical constraints of the components and other performance requirements; for example, it should monitor the battery's *state of charge* (SOC) and current to ensure it is not stressed in a way that could lead to premature aging. In addition to fulfilling similar physical, safety- and performance-related constraints, the EMS should also aim at minimizing fuel consumption and/or pollutant emissions in order to maximize the benefits of hybridization. This is where optimal control plays a role in the EMS design.

Properly designing and implementing an energy management strategy are very complex tasks which hide a complex mathematical problem and several practical engineering problems. However, it is of utter importance for at least two main reasons:

- The extent to which the potential decarbonization and emission-reducing potential of an HEV is fulfilled depends on how close the vehicle can get to the true optimal EMS.
- It is hard to predict the theoretical performance of an HEV design in early design phases. This, coupled with the high dimensionality of an HEV powertrain design operation arising from variations in topologies and component characteristics and sizes, makes the optimal design of HEV powertrains a very challenging task.

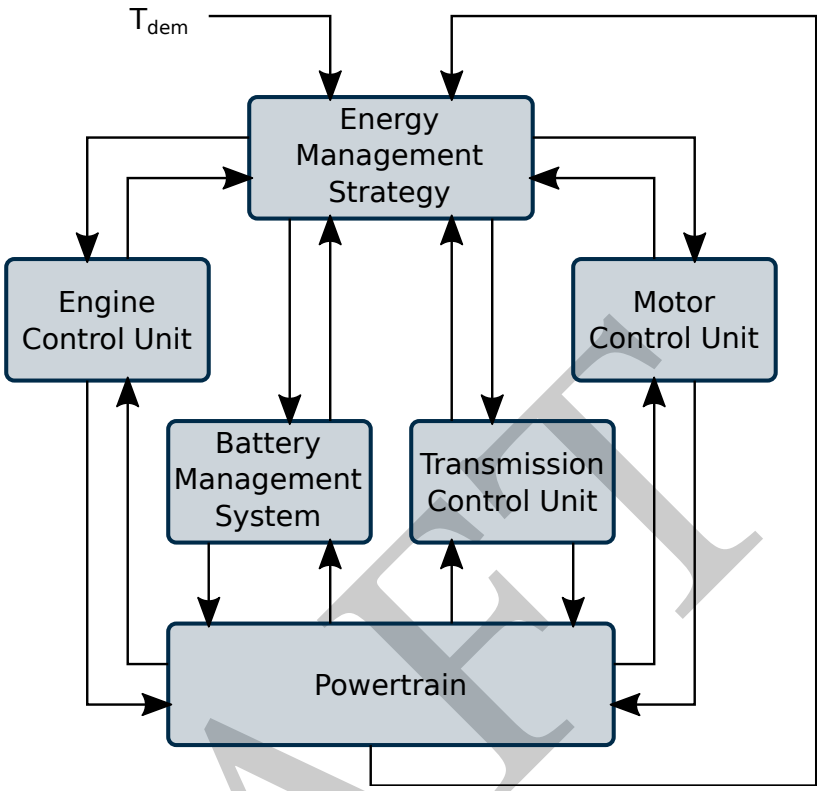


Figure 1.1: The role of the EMS in an hybrid vehicle.

The topics that are then left to discuss before we can dive into the core of this thesis are:

- A formal definition of the optimal design of an EMS.
- An overview of the most common techniques adopted.
- A focus on dynamic programming techniques.

1.1 Energy management strategy optimization

In order to discuss solution methods for the design of energy management strategies, let us first introduce a formal definition of the optimal EMS design problem, formulating it as an optimal control problem. First, we assume that either a set of differential equations or a simulation model is available that describes the evolution of the powertrain's state:

$$\dot{x} = f(x, u; t), \tag{1.1}$$

with some initial conditions

$$x(t_o) = x_o. \tag{1.2}$$

The powertrain's state, described by a vector of state variables $x(t)$, must encode all information required to characterize the physical quantities that are relevant to our problem. For example, variables that should be tracked may include those that are relevant for the definition of the cost functional (1.3) or that must satisfy

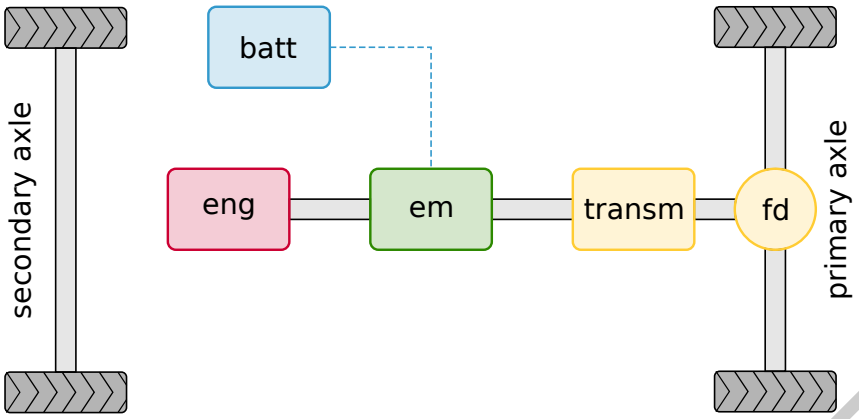


Figure 1.2: Powertrain layout for a simple p2 architecture. ENG: engine, EM: e-machine, BATT: battery, TRANSM: transmission (gearbox), FD: final drive.

some constraint at any time. The control variables $u(t)$ represent those physical quantities we can directly act on in order to influence the system's evolution and therefore the incurred cost.

The objective of optimal EMS design is to minimize a total cost $J(x_o; t_o)$, which is composed by a running cost $L(x, u; t)$ and/or a terminal cost $F(x(t_f); t_f)$:

$$J(x_o; t_o) = \int_{t_o}^{t_f} L(x, u; t)\, dt + F(x(t_f); t_f). \tag{1.3}$$

Finally, the problem is subject to various constraints determined by the physical limitations of the powertrain components or by other requirements. For example, in order for our solution to be useful we have to ensure that we do not ask the thermal engine and e-machines more torque than they can actually provide. Moreover, in order to protect the battery from premature aging, we want to restrict its state of charge (SOC) to stay within a desirable range. Finally, we may want to ensure charge-sustaining operation, that corresponds to ensuring that, after a given driving mission, the battery's state of charge will be equal to its starting value.

In order to clarify the meaning of the definitions we have just given, let us make a simple example. Assume that we want to optimize the fuel economy of the p2 HEV powertrain depicted in Figure 1.2. At any given moment, the driver will transmit an acceleration demand to the vehicle's controller by pushing the accelerator pedal. After transforming this acceleration demand into a torque demand at the powertrain, the question that must be addressed by the EMS is: how to split this torque demand between the thermal engine and the electrical machine in order to minimize fuel consumption while satisfying the aforementioned constraints on the engine, e-machine and battery?

In this example, we would set some torque-related quantity as a control variable. For example, we may define a torque-split ratio as the ratio between the engine torque and the torque demand.

We would then realize that, in order to formulate constraints on the battery's state of charge, we need to be able to track its evolution in time and we would then set it as a state variable. The fuel consumption would obviously be our running cost.

After setting up the problem in this manner, we would have to derive a simulation model to obtain the SOC dynamics and the fuel consumption as a function of the SOC itself, the torque-split ratio and some other time-dependent quan-

tities; the fuel consumption could be our running cost. We would then have a proper formulation of an optimal EMS design problem. What we would need at this point is a technique to solve it.

1.2 Online vs offline EMS

In the problem formulation presented in the previous section, we neglected one major distinction between EMS design problems. In (1.3), we define our total cost over a finite time horizon from t_o to t_f; moreover, although we did not explicitly state it, we implied that we know what the driver's acceleration demand will be over this time horizon. The vehicle controller, in general, does not know this (actually, neither does a human driver); nor do we know what this time horizon is, i.e. when the driver will have reached its destination.

Typically, there are several ways to address this problem:

- Assume that the speed trace of the vehicle is known in advance, defined by a *driving mission*.
- Assume that the speed of the vehicle can be forecast over a short time interval starting from the present. Define this time interval as the control horizon.
- Do not rely on any future information, and treat the problem as an instantaneous optimization.

Since it relies on information that is not actually available in a real driving scenario, the first choice restricts our solution to the class of *offline* control, which means that the method cannot be directly implemented in an actual vehicle controller. The remaining two choices instead allow to design *online* control methods, which can be theoretically implemented in a vehicle controller[†]. Obviously, online control methods must also meet stringent requirements on computational complexity, robustness and reliability in order to be actually implemented in a real-time controller.

[†]Offline and online controllers are sometimes also called *non-causal* and *causal* respectively.

Due to the limited information available, online control methods are generally not able to achieve truly optimal operation; offline control techniques on the other hand can theoretically achieve this goal, at the expense of implementability and, typically but not necessarily, computational cost. While the role of online control methods is clear, the reader who is not familiar with the topic may wonder what the role of offline control methods is. It turns out that there are many.

The first thing that we should consider is that there actually is one situation where the speed trace of the vehicle is known in advance, and that is the case of type-approval testing for the certification of new vehicles over regulatory drive cycles such as those prescribed by the Worldwide Harmonised Light Vehicles Test Procedure in the EU or the Federal Test Procedure in the US. Offline control strategies can therefore provide the theoretical best performance that a given HEV powertrain can achieve on a given regulatory drive cycle, which is an application of obvious interest for vehicle manufacturers.

This also suggests a second application: applying offline control to a given powertrain sets a benchmark that engineers designing the online EMS must target; it provides both a measure of the sub-optimality of their EMS design and an upper threshold which they cannot hope to surpass. Inspecting and understanding the behavior of an optimal controller is also obviously beneficial in order to design better performing controllers.

At the beginning of this chapter, we mentioned that some difficulties in the design of hybrid powertrains are related to the EMS; let us now elaborate a bit

more on that. The design of the EMS is strongly influenced by the characteristics of the powertrain itself, and these characteristics can vary widely in terms of topology[†], components size and technologies. When attempting to design an HEV powertrain, all these variations generate a very large and multi-dimensional design space, which can only be explored by evaluating a large number of design candidates. Typically, amongst the most important metrics by which these design candidates will be evaluated are their fuel consumption and emissions, which depends on the EMS design; but the EMS design depend on the powertrain design itself.

[†]Essentially, the number and position of the electrical machines with respect to the driveshaft.

Hence, since many design candidates must be evaluated, any optimal powertrain design tool needs to embed an automatic EMS design tool. Moreover, this EMS design must ensure a fair comparison between candidates: it is not possible to simply design an EMS for a typical powertrain and then use that EMS with other candidates. The most sound solution is therefore to adopt the optimal EMS for each design candidates, which can be obtained by some offline EMS optimization algorithm which is flexible enough to adapt to any HEV powertrain.

1.3 Optimal energy management strategy methods

In the early era of HEVs commercialization, the focus was on developing heuristic control strategies [4]. A theoretical framework for treating the EMS design as an optimal control problem was lacking and engineers focused on achieving simple yet robust control strategies powered by engineering intuition. Then, the academia picked up the issue and many methods have been developed over the years to design and implement HEV energy management strategies, both by improving on existing methods and by opening entirely new methodologies. In this Section, we list the most widespread approaches that can be found in the literature.

1.3.1 Rule-based control strategies

Rule-based control strategies encompass a variety of methodologies that share the characteristic that they are not designed within an optimal control framework, but rather implement a set of rules that were designed for a particular HEV topology and then calibrated for a specific powertrain.

These rules may be heuristic or obtained through a rule-extraction algorithm from some dataset. Heuristic rules can be implemented in the form of simple formulas purely based on engineering judgment [‡], or as a fuzzy logic controller [71, 73]. Non-heuristic rules on the other hand can be obtained by processing large datasets of simulation results which implement an optimal control-based EMS such as dynamic programming, either by visual investigation [11, 48, 66] or by a machine-learning approach [19, 59]. These types of control methodologies attempt to translate the performance of optimal control techniques into implementable, real-time capable controllers.

[‡]See for example the extensive treatment of such strategies in [20, Chapters 10 and 11]

Nonetheless, they rely heavily on the results of optimal control techniques which must be first run offline in a simulation environment. Therefore, their performance is tied to the implementation of sophisticated optimal control techniques and accurate simulation models.

The most common approach by far is to obtain these optimal results using dynamic programming, as it is the only truly optimal control technique currently available for EMS design. However, as we will later discuss throughout this thesis, dynamic programming is not suitable for high-dimensional simulation models.

1.3.2 Equivalent Consumption Minimization Strategies

Halfway in between heuristic and optimal control are a group of methods going under the name of ECMS, which is the shorthand for *Equivalent Consumption Minimization Strategy*.

These methods were first introduced by Paganelli et al. [65, 63, 64] and are based on the instantaneous minimization of an *equivalent* fuel consumption $\dot{m}_{f,eq}$, which is formulated as the sum of the current fuel flow rate $\dot{m}_f(t)$ and an *electrical* consumption $\dot{m}_{f,el}$.

$$\dot{m}_{f,eq} = \dot{m}_f(t) + \dot{m}_{f,el}. \tag{1.4}$$

The electrical term is formulated as the electric motor power multiplied by a constant equivalence factor:

$$\dot{m}_{f,eq} = \dot{m}_f(t) + sP_{em}. \tag{1.5}$$

This equivalence factor s is defined as the product of the (mean) efficiencies that characterize the conversion of the chemical energy stored in the fuel into electrical energy stored in the battery or vice versa, depending on whether the e-machine is working in motor or generator mode.

This procedure obviously does not ensure that the battery's SOC will stay within feasible bounds or that its value at the end of a given mission will be close to its initial value; several methods have been proposed to overcome this limitation.

The original approach was to penalize deviations from a reference SOC level by adopting an arbitrary penalty function which multiplies the equivalence factor [63] or by shifting the operating point in order to minimize or maximize the battery current as the SOC gets higher or lower than the reference SOC, while not exceeding some maximum extra-cost induced [64].

Another approach consists in using some tuning procedure of the equivalence factors over a given driven cycle until the values which ensure charge-sustaining operation are found. While this approach is preferable in offline control, it is unsuitable for an online implementation for at least two reasons. Firstly, these optimal values are strongly dependent on the initial SOC. Secondly, if constant equivalence factors are used, the SOC trajectory is very sensitive to small variations in the factors [18].

Therefore, online ECMS approaches in the literature generally adopt some form of an *adaptive* or *robust* equivalence factors scheme, in the spirit of the original approach.

Finally, other more complex approaches have been proposed that involve exploiting future driving information in order to periodically adapt the equivalence factor. For example, Musardo et al. [60] propose using a vehicle speed predictor based on GPS data to generate information with which an *adaptor* algorithm can evaluate the equivalence factors that will ensure charge-sustaining operation over a given control horizon.

1.3.3 Model Predictive Control

Model Predictive Control is a general method that relies on a model of the system dynamics which is used to evaluate the state's evolution over a given prediction horizon.

At each time step, a control strategy is evaluated for the whole prediction horizon in order to minimize the cost function. However, only this strategy is only implemented for one time step[†], after which the control strategy is recomputed

[†] Alternatively, over a control horizon shorter than the prediction horizon.

based on the newly observed state. This makes MPC suitable for online control.

In general, one needs at least two ingredients for an MPC: a dynamical system with its cost function and a suitable optimization algorithm. For applications to the EMS problem, where future driving conditions are very relevant to the state dynamics and cost, some form prediction algorithm is also needed.

Often, the state dynamics are linearized to a linear time-varying model with the power demand being treated as a disturbance. If the cost is also formulated as quadratic, this allows to solve the optimization problem with quadratic programming solvers, which are generally fast. Examples of this approach are [15] for a parallel hybrid and [17] for a series hybrid, where the torque demand and vehicle speed or the power demand are treated as measured disturbances, respectively. Others have also experimented with more complex optimization algorithms such as a layered DP-PMP algorithm [62] coupled to a non-linear HEV quasi-static model.

As for the prediction algorithm, a wide range of solutions has been proposed. Two of the most popular approaches are to consider an exponentially decaying torque demand and subsequently evaluate the vehicle speed with the powertrain model [15][†] and Markov chains [38, 58], which can also be trained online by adapting the transition probabilities [17]. Various neural networks architectures have also been proposed [76].

[†] or vice-versa, see e.g. [76].

1.3.4 Pontryagin minimum principle

The minimum principle is a useful tool that can be used to derive necessary conditions of optimality for control problems. While it does not guarantee finding the optimal solution, it can help in greatly narrowing down the solution to a small set. For simple problems, it may even be possible to identify the structure of the optimal solution analytically[‡].

[‡] Notable examples in EMS design are [1, 49].

For more complex problems, the conditions provided by the minimum principle can still be useful by solving them numerically. These conditions generally produce a *two-point boundary value problem*, which is a set of differential equations to be solved coupled with an initial and a final condition (the boundary conditions). Methods following this approach are often called *indirect* methods in optimal control.

Unfortunately, boundary value problems, or BVPs, are generally hard to solve and require iterative techniques which may not converge at all if a good initial guess of the optimal solution is not provided. This may prove in practice to be a daunting task: the minimum principle characterizes the optimal solution in terms of the *co-state* trajectory. Since the co-state may not have a practical physical interpretation, making a good guess of its optimal trajectory is not always possible.

Furthermore, the optimality conditions must still be derived analytically, either by hand or with the assistance of some computer algebra system, which is a cumbersome task. Thus, the method is not very flexible as it must be carefully adapted to the specific control problem at hand and even small variations in the problem may require to re-work the conditions from scratch.

Finally, there is no guarantee that the obtained solution is globally optimal, only that it is locally optimal in some neighborhood of the initial guess; although global optimality can be checked a posteriori using the sufficient condition provided by the Hamilton-Jacobi-Bellman equation[§].

[§] We will introduce this in § 2.3.3.

1.3.5 Dynamic programming

Dynamic programming is perhaps the most widespread technique for offline control, owing to its many advantages. First and foremost, it is the only technique

that guarantees optimality regardless of the structure of the optimal control problem. Complex control problems with highly non-linear dynamics and constraints characterized by both discrete and continuous control variables can be handled. Little to no insight of the underlying physics is required to adapt the optimization algorithm to a given EMS design problem, making it very flexible.

The price to pay for all these advantages is that dynamic programming is computationally expensive for high dimensional control problems; the computational cost explodes as the number of state and control variables increases. This limits our ability to use models that accurately simulate the powertrain behavior. This is a central point in this thesis and the motivating factor for the subject treated in Part I, where we essentially attempt to alleviate the numerical inefficiencies of DP algorithms, and Part II, where we seek an alternative method based on the same principle of standard DP which does not suffer from the *curse of dimensionality*.

Part I

Dynamic Programming

2 Optimal control problems

In this thesis we discuss optimal control techniques for nonlinear dynamic systems and we apply them to develop methods for obtaining optimal energy management strategies for hybrid electric vehicles.

In this first chapter, we lay out a framework for the general description of optimal control problems and some useful terminology that will be used throughout this text. Then, we briefly introduce the two fundamental principles that were developed in optimal control theory: the minimum principle and the principle of optimality. The latter forms the basis for the dynamic programming algorithms developed in this thesis.

2.1 Formulation of an optimal control problem

The objective of the optimal control problems treated in this thesis is to control the evolution in time of a dynamical system, whose state is defined by its state variables x, by acting on one or more control variables u. By controlling u, we want to minimize some additive cost J.

Optimal control problems can be formulated in *discrete* or *continuous* time, depending on whether the system is described as a system which evolves through discrete stages or continuously in time.

2.1.1 Continuous time

In continuous time, we assume that a model is available that characterizes the state dynamics (typically a set of ODEs) of the form

$$\dot{x} = f(x, u; t), \tag{2.1}$$

with some initial conditions

$$x(t_o) = x_o. \tag{2.2}$$

The state dynamics are written as $f(x, u; t)$ if they are explicitly time-dependent or $f(x, u)$ otherwise[†].

The state vector $x(t)$ and the control vector $u(t)$ are n-dimensional and m-dimensional vector functions of time, respectively[‡]. The system's evolution is observed and controlled over a finite control horizon $[t_o, t_f]$.

If, in addition, the terminal state of the system is also defined, i.e.

$$x(t_f) = x_f. \tag{2.3}$$

the problem is said to be *fixed endpoint*, as opposed to *free endpoint*.

The total cost incurred J is composed by an *instantaneous* or *running* cost L, which can be a function of the state and control variables and time, and a terminal cost F, which is a function of the terminal state[§] and final time exclusively. The total cost is additive in that the instantaneous cost is integrated throughout the whole time interval $[t_o, t_f]$.

$$J(x_o, u) = \int_{t_o}^{t_f} L(x, u; t)\, dt + F(x(t_f); t_f). \tag{2.4}$$

Note that, given the initial state, the total cost is a function of the control trajectory, which is itself a function. Hence, $J(x_o, u)$ is a `functional`, a function of

[†] Explicitly time-dependent systems are called *time-varying*, as opposed to *time-invariant* or *autonomous*.

[‡] We often omit the time dependence of $x(t)$ and $u(t)$ for conciseness.

[§] I.e. the value of the state variables at final time.

functions. The objective of an optimal control problem is to find the optimal control trajectory $u^o(t)$ that minimizes the total cost (2.4), where the state trajectory is obtained by applying $u^o(t)$ to the state dynamics (2.1). This optimal solution is also called an *extremum* of the cost functional.

In the most general case, the state dynamics and cost are defined as being explicitly dependent on time, but there may be systems which are only implicitly time dependent through the state and control variables; these are sometimes called *autonomous* systems.

Several constraints can also be formulated on the state variables and variables, which can be formalized in many different ways. One very general notation is to restrict the control variables to belong to a *feasible* control set U, which can be both time- and state-dependent:

$$u(t) \in U(x, u; t). \tag{2.5}$$

[†] As we will do in the context of differential dynamic programming in Part II.

An alternative notation, which may be convenient in some optimal control frameworks[†], is to formulate our constraints in the form of equality or inequality constraints:

- state constraints

$$g(x; t) \leqslant 0, \tag{2.6}$$

- control constraints

$$g(u; t) \leqslant 0, \tag{2.7}$$

- mixed state-control constraints

$$g(x, u; t) \leqslant 0. \tag{2.8}$$

Inequality constraints of the form of (2.6) to (2.8) can be used to set constraints on the state and control variables throughout the system's evolution. Constraints on the terminal state may be formulated by completely constraining it with a terminal condition

$$\psi(x(t_f); t_f) = x(t_f) - x_f = 0, \tag{2.9}$$

or by defining some target set, as we will do in § 3.1.5. We will see in both § 3 and § 5 that this type of constraint is generally more challenging and it requires a separate treatment.

2.1.2 *Discrete-time systems*

In discrete time, the system is characterized by a set of difference equations of the form

$$x_{k+1} = f_k(x_k, u_k), \qquad k = 0, 1, \ldots, N-1, \tag{2.10}$$

and some initial condition x_0. The role of the terminal time t_f is taken by a control horizon (a number of stages) N, which is equal to the number of times the control variables must be selected. The state and control trajectories, rather than functions of time, are now sequences.

The total cost is expressed, for a given control sequence $\pi = \{u_0, u_1, \ldots, u_{N-1}\}$, as

$$J(x_0, \pi) = \sum_{k=0}^{N-1} L_k(x_k, u_k) + F(x_N). \tag{2.11}$$

Note that the state dynamics f_k and the running cost L_k are written as being dependent on the stage k. This is somewhat analogous to defining them as explicitly depending on time in the continuous-time case; similarly, if they do not depend on the stage k the system is said to be autonomous.

The goal for discrete-time problems is to find an optimal control sequence $\pi^o = \{u_0^o, u_1^o, \dots, u_{N-1}^o\}$ that minimizes the total cost. Constraints can be formulated similarly to the continuous-time case, with the only difference that the constraint functions and the feasible control set will be stage-dependent rather than time-dependent, e.g. $u_k \in U_k(x_k)$.

2.2 Pontryagin's minimum principle

The minimum principle (historically developed in its maximum form [14]) is a global optimization technique which is useful to derive necessary conditions for optimality which must be satisfied by any optimal solution [12]. Originally developed in the Soviet Union in the late 1950s [12, Chapter 16], it has since become a milestone in optimal control theory due to its versatility.

Let us define the Hamiltonian[†] of the controlled system (2.1) as:

$$H(x, u, p; t) = L(x, u; t) + p^T f(x, u; t), \tag{2.12}$$

where $p(t)$ is an n-dimensional vector known as the *costate* or *adjoint* vector. The Hamiltonian can be seen as analogous to the Lagrangian used in static optimization, as it combines the cost function with the state equations through a set of multipliers.

[†] This Hamiltonian is also referred to as the *control* Hamiltonian, in order to distinguish it from the Hamiltonian of mechanics from which it was originally inspired.

The minimum principle must be formulated in different ways based on the structure of the problem[‡]. Perhaps the simplest form can be written for a fixed-time, fixed-endpoint problem with a cost function of the form (2.4) and no state or control constraints.

[‡] See e.g. [13, 47]

In this case, if x^o, u^o are optimal trajectories, there must exist a costate vector function $p(t)$ satisfying the adjoint equations

$$\dot{p} = -H_x(x^o, u^o, p; t)^{\S} \tag{2.13}$$

such that the Hamiltonian minimization condition is satisfied:

$$H(x^o, u^o, p; t) = \min_u H(x^o, u, p; t). \tag{2.14}$$

[§] Throughout this chapter, H_x and H_p stand for the Jacobian of H with respect to x and p.

Since $x \in \mathbb{R}^n$ and $p \in \mathbb{R}^n$, the necessary conditions we just wrote generate a set of $2n$ differential equations which therefore require $2n$ boundary conditions. These conditions are given by the initial and terminal state: $x^o(t_o) = x_o$ and $x^o(t_f) = x_f$.

In some sources, the minimum principle is rewritten in terms of the *Hamiltonian form*:

$$\begin{cases} \dot{x}^o = H_p(x^o, u^o, p; t), \quad x^o(t_o = x_o), \\ -\dot{p} = H_x(x^o, u^o, p; t). \end{cases} \tag{2.15}$$

The first expression is simply an alternative way to incorporate the system equations (2.1) and the second constitutes the adjoint equations. This Hamiltonian form coupled with the minimality condition is equivalent to the set of necessary conditions we saw earlier.

If we consider the same problem but let the final state be free, the principle introduces an additional condition. In this case, if x^o, u^o are optimal trajectories, there must exist a costate vector function $p(t)$ satisfying the adjoint equation (2.13) and such that the following necessary conditions are satisfied:

1. Hamiltonian minimization

$$H(x^o, u^o, p; t) = \min_u H(x^o, u, p; t). \tag{2.16}$$

2. Transversality condition

$$p(x(t_f)) - F_x(x^o(t_f)) = 0. \tag{2.17}$$

Note that this transversality condition replaces the n boundary conditions that we lost by removing the terminal state constraint $x^o(t_f) = x_f$.

If the Hamiltonian form of the principle is used, the transversality condition must be included in the set of equations:

$$\begin{cases} \dot{x}^o = H_p(x^o, u^o, p; t), \quad x^o(t_o = x_o), \\ -\dot{p} = H_x(x^o, u^o, p; t). \\ p(x(t_f)) = F_x(x^o(t_f)). \end{cases} \tag{2.18}$$

Many other formulations are possible for problems which include additional constraints or that are defined over a variable time interval (i.e. t_f is not fixed). Since the minimum principle is not the focus of this thesis, we do not report them here.

In any case, we remark that applying the principle generally leads to write a set of $2n$ equations with $2n$ boundary conditions. These boundary equations are not all enforced at either t_o or t_f; rather, they are split between the time interval's endpoints. Hence, the principle translates into a boundary value problem, which are generally hard to solve as we discussed in § 1.3.4.

2.3 Dynamic programming and the HJB equation

Dynamic programming is essentially a technique for solve multi-stage decision problems, i.e. problems where decisions must be made in stages in order to minimize a certain total cost. In the context of optimal control, it can very effectively be used to deal with discrete-time systems of the sort that we described in § 2.1.2; for these problems, the principle of optimality provides a recurrence relation that is well suited for a computer solution.

2.3.1 *The principle of optimality*

The dynamic programming approach can be applied to a very wide class of problems with a variety of algorithms. What these algorithms have in common is that they are all based on the principle of optimality, which was introduced by Richard Bellman [8].

The principle can be formulated in many ways. Perhaps one of the most effective is Bellman's own statement in [9, p. 15]:

> *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

We can intuitively prove the principle by contradiction as follows[†]. Consider the problem of reaching node d from node a in Figure 2.1, and suppose that the optimal path[‡] is to go from a to d through b only; thus incurring in an optimal cost $J^o_{ad} = J_{ab} + J_{bd}$. The principle of optimality then tells us that b-d is the optimal path from b to d, with optimal cost $J^o_{bd} = J_{bd}$.

[†] This derivation is loosely based on [41, ch. 3].

[‡] We refer here to optimal in the sense of minimum cost.

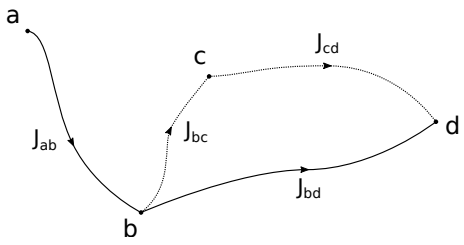


Figure 2.1: Optimal path and suboptimal paths from a to d.

Suppose that this is not true, i.e. there is a path b-c-d, with cost $J_{bc} + J_{cd} < J_{bd}$, that is optimal from b to d. Then:

$$J_{ab} + J_{bc} + J_{cd} < J_{ab} + J_{bd} = J^{o}_{ad}, \tag{2.19}$$

which violates the hypothesis that J^{o}_{ad} is the optimal cost. This simple example also motivates the following equivalent statement of the principle which can be found in [10, p. 7]:

> *The tail of an optimal sequence is optimal for the tail subproblem.*

This statement is perhaps more evocative of a method for iteratively building the optimal sequence by recursively solving increasingly long tail subproblems.

2.3.2 A recursive relationship in dynamic programming

Suppose we have a two-stage decision problem, exemplified in Figure 2.2, where the state evolves from an initial node a to a terminal node h. Depending on the decisions we take on each of the two stages, we may get there by going through nodes b, c or d at the first decision stage and through nodes e, f or g at the second decision stage.

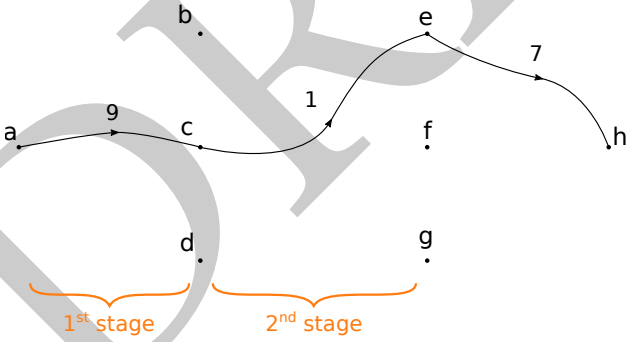


Figure 2.2: A two-stage decision problem with an arbitrary (non-optimal) control sequence.

Depending on the decisions we take, we incur in some *arc-cost* for each decision stage which depends on the decision itself. After the second (and last) stage, we also incur a terminal cost which is the cost of reaching node h. An arbitrary path is depicted in Figure 2.2 to exemplify the process. The first decision leads from a to c with an arc-cost $c_{ac} = 9$, the second decision leads from c to e with cost $c_{ce} = 1$, and finally a terminal cost $c_{eh} = 7$ is incurred to drive the system from e to h, for a total cost of 17.

Suppose we have all the information we need to determine the arc-costs incurred as a result of taking each decision as well as the terminal arc-costs. How can we exploit the principle of optimality to build the optimal path from a to h?

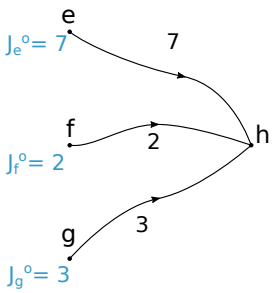


Figure 2.3: Terminal costs for the two-stage problem.

We start by considering the terminal cost which we may incur based on the state (the node) we are at the last stage, which we represent in Figure 2.3. At this time, no decision must be taken as we must simply get to the terminal node h. The optimal cost for each of the nodes we may be at (i.e. e, f or g) is simply the terminal cost we incur to go from that node to h. We call this optimal cost the *cost-to-go* and we mark this with the superscript ^{o} (e.g. J_e^o is the optimal cost to go from e to h).

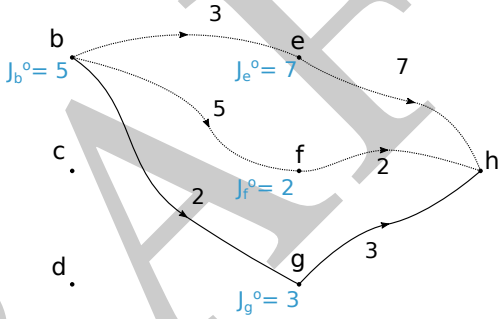


Figure 2.4: Optimal paths for the last stage.

Now let us step back one stage as in Figure 2.4. Suppose that, as a result of our previous choices, we are at node b. We can obtain the optimal cost needed to get to h by trying out all three of the decisions we can take at this stage and sum the corresponding arc-cost and the cost-to-go of the node we end up to as a result of the same decision; we keep the minimum cost and discard the others.

For example, starting from b in Figure 2.4, we may decide to go to e, f or g; the corresponding total costs required to ultimately get to h will be

$$\begin{aligned} c_{be} + J_e^o &= 10, \\ c_{bf} + J_f^o &= 7, \\ c_{bg} + J_g^o &= 5. \end{aligned} \tag{2.20}$$

Hence, the cost-to-go at node b is $J_b^o = 5$.

We may now consider nodes c and d and find the corresponding costs-to-go by repeating the same operation, which we can formalize as:

$$J_\alpha^o = \min_\beta \left[c_{\alpha\beta} + J_\beta^o \right]. \tag{2.21}$$

Equation (2.21) expresses in essence a recursive procedure for constructing the cost-to-go at a given stage as a function of the state of the system. Once we have obtained the costs-to-go for nodes b, c and d, we can step back one stage to the initial node a as in Figure 2.5. Once again, we can apply (2.21) to obtain the

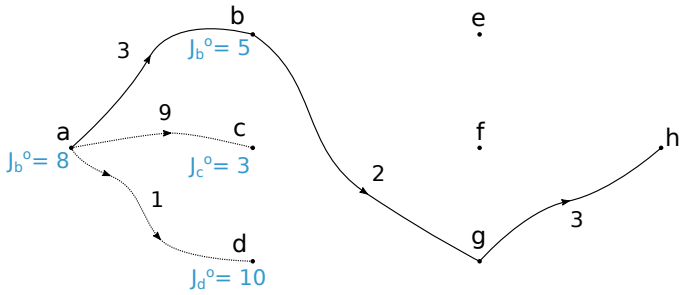


Figure 2.5: Optimal paths for the whole two-stage problem.

cost-to-go for node a, which is also the solution to our original problem, i.e. the optimal cost required to go from a to h.

It is easy to see that this algorithm can be readily extended to any multi-stage decision problem regardless of the number of stages. For each stage (2.21) can be applied to build costs-to-go until the first stage is reached and the optimal solution is found.

The algorithm is also suitable for discrete-time systems, as these can be obviously viewed as multi-stage decision problems. Let us the replace the cost-to-go J° with the value function[†] $V_k(x_k)$, which explicitly expresses its dependence on the value of the state variables x at a given stage k. Let us also replace the arc-cost with the running cost L_k and let us obtain the value of the state at the next stage using the state equations f_k. We then obtain the following recursive relationship:

[†] Note that the terms cost-to-go and value function are essentially synonyms.

$$V_k(x_k) = \min_{u_k} \left[L_k(x_k, u_k) + V_{k+1}\left(f_k\left(x_k, u_k\right)\right) \right]. \tag{2.22}$$

This relationship forms the basis for building a dynamic programming algorithm to deal with discrete-time optimal control systems of the form in § 2.1.2. However, there are some additional challenges in case the state and/or the control variables are not inherently discrete as in this example and in case the problem is constrained. This additional complexity will be treated in § 3.

Continuous-time systems can be treated as well by discretizing the time interval $[t_o, t_f]$ in N time increments Δt. The state equations can then be rewritten in approximate form as

$$x(t + \Delta t) = x(t) + \Delta t\, f(x(t), u(t); t), \tag{2.23}$$

or, using the index k to identify the k-th time increment, so that $t = k\Delta t$:

$$x_{k+1} = f_k(x_k, u_k) \stackrel{\text{def}}{=} x(k\,\Delta t) + \Delta t\, f(x(k\,\Delta t), u(k\,\Delta t); k\,\Delta t). \tag{2.24}$$

Similarly, the running cost is discretized so that the integral cost can be translated into a summation.

2.3.3 The Hamilton-Jacobi-Bellman equation

When dealing with continuous-time systems, the dynamic programming algorithm that we just introduced requires that the state dynamics are approximated by difference equations and that the cost is approximated by a summation. We will now see an alternative way to deal directly with these systems.

The same principle of optimality that lies at the basis of dynamic programming can also be used to derive the Hamilton-Jacobi-Bellman equation, which is essentially the infinitesimal form of (2.22):

$$-\frac{\partial V}{\partial t}(x; t) = \min_u [L(x, u; t) + \langle V_x(x; t), f(x, u; t)\rangle]^{\ddagger} \tag{2.25}$$

with the boundary condition

$$V(x(t_f); t_f) = F(x(t_f); t_f). \tag{2.26}$$

Theoretically, the HJB equation can provide an analytical solution to continuous-time optimal control problems. In practice, this is not possible and some numerical technique must be used; this would also generally involve using some numerical integration scheme. Hence, when dealing with continuous-time systems, two alternative methods based on the principle of optimality arise:

- directly treat the original problem with the HJB equation, which must be (approximately) solved numerically;
- approximate the original control problem with a discrete-time equivalent and obtain the exact solution with dynamic programming.

Hence, as noted by Kirk in [42, ch. 3], we can either obtain an approximate solution to the exact optimal control problem using the HJB equation or obtain the exact solution to a discrete approximation of the original control problem using dynamic programming. In the following chapter we will treat the latter approach, while in the second part of this thesis we will implement a technique based on the former.

‡Throughout this text, $\langle \cdot, \cdot \rangle$ denotes the scalar product of two vectors.

3 Dynamic programming algorithms

In the previous chapter, we introduced the formalism used in optimal control theory to define control problems, we took a quick detour through Pontryagin's minimum principle, and we finally explored the principle of optimality, developing it by intuition. We also saw how the principle of optimality can be used to characterize the optimal solution in terms of *value functions*, which we can build in a dynamic programming algorithm using a convenient recursive relationship or with the more computationally challenging Hamilton-Jacobi-Bellman equation.

In this chapter, we will focus on dynamic programming. In particular, we will formalize a proper dynamic programming algorithm and we will treat the additional complexity that is needed to move from the shortest-path problem we used to illustrate the principle of optimality to the applications we typically find in energy management strategy design. This discussion forms the basis for the development of a flexible open-source toolbox for dynamic programming, which is an original contribution of this thesis work and whose presentation constitutes the second half of this chapter.

3.1 Practical implementation of a DP algorithm

In § 2.3.2, we intuitively introduced the concept which allows us to build recursive dynamic programming algorithms using a shortest-path problem. That example, however, does not immediately resemble the sort of optimal control problems we described in § 2.1. Rather than having a dynamical system whose state evolves in time, we had a problem where the decisions we take define the path we take from an initial node to a final node. Aside from the terminal condition, we did not enforce any constraints.

Additionally, we had an inherently a discrete system, where at each stage[†] we were allowed to select decision (i.e. controls) from a discrete control set which led to one state among a discrete state space; Equation (2.21) is suitable for those problems. If we want to accommodate systems that are characterized by continuous states and/or controls, we need something slightly more elaborate than that.

[†] In most control problems, stages are identified as discrete time steps; in this text, the two terms are used interchangeably.

In this section, we will describe an algorithm to deal with the optimal control problems introduced in § 2.1 and we will discuss some important computational hazards that often arise.

3.1.1 *The fundamental DP algorithm*

Let us first consider discrete-time systems as in § 2.1.2; these problems are easier to deal with DP recursion because they are inherently cast as multi-stage decision processes, where each discrete time step corresponds to one stage.

Suppose that $\pi^o = u_o^o, u_1^o, \dots, u_{N-1}^o$ is the optimal control sequence that minimizes the total cost

$$J(x_o, \pi) = \sum_{k=o}^{N-1} L_k(x_k, u_k) + F(x_N). \tag{3.1}$$

Then, the principle of optimality tells us that the truncated optimal control sequence $u_l^o, \dots, u_{N-1}^o$ is optimal for the tail sub-problem where we start from stage l and state x_l^o, i.e. it is the control sequence that minimizes

$$L_l(x_l^o, u_l) + \sum_{k=l+1}^{N-1} L_k(x_k, u_k) + F(x_N). \tag{3.2}$$

As we anticipated in § 2.3.2, we can use the principle to recursively construct a sequence of value functions

$$V_N(x_N), V_{N-1}(x_{N-1}), \dots, V_o(x_o), \tag{3.3}$$

where each $V_k(x_k)$ expresses the minimum cost required to reach the terminal state starting from state x_k at time k. Note that $V_o(x_o)$ gives the optimal cost of our control problem.

First, we must initialize the value function at stage N. Obviously, this is simply equal to the terminal cost $F(x_N)$:

$$V_N(x_N) = F(x_N). \tag{3.4}$$

Then, we step one stage back and we update the value function. As we did in § 2.3.2, for all possible values of x_k we try all possible controls u_k. We sum the running cost incurred and the value function evaluated at the state $f_k(x_k, u_k)$ that will be reached at the next stage; we keep the minimum over the controls u_k and we assign this value to $V_k(x_k)$.

$$V_k(x_k) = \min_{u_k} \left[L_k(x_k, u_k) + V_{k+1}\left(f_k(x_k, u_k)\right) \right]. \tag{3.5}$$

Although it is more complete, this formulation of the DP recursion algorithm is also a bit cumbersome, and a few clarifications are in order. First, we note that the running cost $L_k(x_k, u_k)$ is expressed as a function of both the current state and the control variables. This running cost plays the same role of the arc-costs in § 2.3.2, but is more general[†].

[†] The arc cost is usually intended as the cost associated with the transition from one discrete state to another, while the running cost may also be associated with transitions to states that do not belong to a discrete computational grid.

Next, we note that the argument of the value function at the next stage is written as $f_k(x_k, u_k)$. This simply the value that the state variables will have at the next stage if the current state is x_k and the control variable is set to u_k for the current stage, evaluated with the state equations (2.10), i.e. $x_{k+1} = f_k(x_k, u_k)$.

Also, note that (3.5) makes it explicit that, once we have constructed the value function $V_{k+1}(x_{k+1})$ in a previous iteration, the argument of the minimization depends only on the value of the current state x_k and the selected control variables u_k. Once we perform the minimization over u_k, it is evident that the result only depends on x_k; which once again justifies our writing of the value function $V_k(x_k)$ as a function of the current state only.

Once we have constructed value functions for all stages k applying (3.5) recursively over the whole state space, we can easily extract the optimal control sequence $u_o^o, u_1^o, \dots, u_{N-1}^o$ and the corresponding optimal state trajectory $x_1^o, \dots, x_N^o$ given an initial state x_o. First, we set $x_o^o = x_o$. Then, we evaluate the optimal control for the first stage using:

$$u_o^o(x_o) = \arg\min_{u_k} \left[L_k(x_k, u_k) + V_{k+1}\left(f_k(x_k, u_k)\right) \right], \tag{3.6}$$

and we advance the simulation by one step by using (2.10):

$$x_{k+1}^o = f_k(x_k^o, u_k^o); \tag{3.7}$$

the process is repeated until the last stage is reached.

This second phase of the algorithm is called the *forward* phase, while the first phase is called the *backward* phase. For the sake of clarity, we reinstate that the output of the backward phase is a sequence of value functions $V_k(x_k)$, one for each stage of the control problem. These value functions can then be used in the forward phase to determine the optimal control sequence with (3.6) while advancing the simulation with the state equations (2.10).

Until now, we have implicitly assumed that:

- There are no constraints on the state and control variables.
- We are dealing with a free-endpoint problem (there is no terminal state constraint).
- We can always somehow represent and store $V_k(x_k)$ exactly.

In the following subsections, we are going to remove these assumptions.

3.1.2 Constrained state and control spaces

Dealing with control and state constraints in a dynamic programming algorithm is actually pretty straightforward in theory. All we have to do is to restrict the minimization (3.5) to:

$$V_k(x_k) = \min_{u_k \in U_k(x_k)} \left[L_k(x_k, u_k) + V_{k+1}\left(f_k\left(x_k, u_k\right)\right) \right]. \tag{3.8}$$

Rather than minimizing over any u_k, we restrict u_k to be in a feasible control set $U_k(x_k)$. The fact that this control set can be state-dependent means that we can incorporate state variable, control variable and mixed constraints in the same way. This possibility is one of the most powerful features of dynamic programming, due to the resulting flexibility in modeling.

If the control variables are continuous, changing (3.5) to a constrained optimization may pose some challenges; although it may be possible to overcome these by using a sufficiently robust NLP (non-linear program) solver, this may prove to be too computationally demanding in practice. On the other hand, if the control variables are discretized, the minimization (3.5) becomes a simple minimization over a discrete set of values and the constraints can be easily dealt with by excluding those values of u_k that violate the constraints from the solution. This is the approach that will be used in this chapter.

3.1.3 Continuous state spaces

In order to construct the value function at a given stage k, we said that we have to apply (3.5) over the whole state space, i.e. for all values of x_k that the system may have. For systems with a discrete state space, this does not pose any theoretical challenge. On the other hand, if the state space is continuous, we are puzzled with the issue of applying (3.5) *for all* x_k; clearly, this is not doable[†]. Therefore, we must discretize the state space and obtain $V_k(x_k)$ for the whole discretized state space.

This in turn poses another issue: if the state space is not inherently discrete but rather a discretized continuous state space, then $x_{k+1} = f_k(x_k, u_k)$ will not in general belong to the discretized state space. Thus, we are unable to evaluate $V_{k+1}\left(f_k\left(x_k, u_k\right)\right)$: at the previous iteration, we only computed its values for each x_k on the discrete state space. In order to deal with these issue we can follow one of two approaches:

Parametric approximation. Approximate $V(x_k)$ with a class of functions $V(x_k, r_k)$ that depend on a set of parameters r_k which are adjusted in order to provide a good approximation of $V(x_k)$[‡].

Interpolation. Compute and store the value function over the discretized state space, which defines a computational grid, and obtain $V(x_k)$ when needed by interpolation between the closest grid points.

Clearly, interpolating the value function means that the corresponding DP algorithm will be an approximate form of the exact DP algorithm, whose accuracy depends on the accuracy of the interpolation itself; which is therefore crucial.

[†] Unless we have an analytical representation of f_k, L_k and V_k, but we assume here that this is not the case.

[‡] See [10, ch. 3] for an extensive treatment.

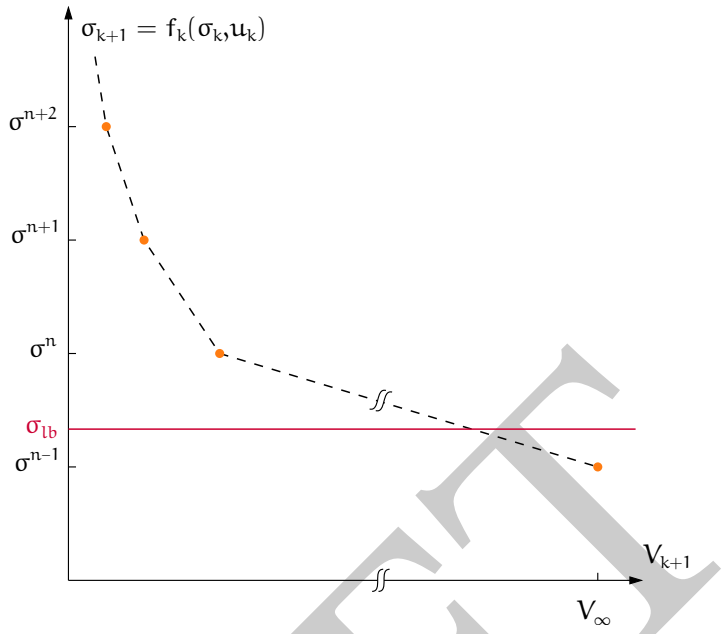


Figure 3.1: Value function interpolation issues near to the feasible state space boundary.

The accuracy of the interpolation depends, in general, on the discretization of the state space. A finer grid will improve accuracy at the cost of increased computational time and required storage. Moreover, the selection of the interpolation scheme may also have an effect; if value function forms a regular shape with respect to the state variables at any given shape, choosing an interpolation scheme that fits that shape well may allow to achieve the same accuracy with less grid points.

For example, Larsson et al. [46] showed that, for a parallel HEV architecture, it may be possible to achieve high accuracy with a reduced grid size by approximating the value function with cubic splines[†]. rather than linear interpolation, to an extent which outweighs the increased cost due to computing splines.

Finally, when dealing with constrained state spaces, interpolation in the proximity of the constraints might be the source of numerical errors and it requires some extra care, as we will see in the next section.

[†] More precisely, the authors approximate the derivative of the value function with respect to the state. That this derivative is what actually drives the optimal trajectory, as claimed by the authors, is also easily deduced by comparing with the V_x term in the HJB equation (2.25)

3.1.4 *Numerical issues associated to constrained state spaces*

When dealing with optimal control problems with continuous state variables and state variable constraints, it may happen that the value of $x_{k+1} = f_k(x_k, u_k)$ while evaluating the value function update (3.5) ends up violating those constraints. Clearly, we want to exclude those points from the minimization so that decisions leading to unfeasible states are avoided. The most common approach in EMS design literature [77, 79] is to assign a very large penalty cost (possibly infinity) to those decisions.

This approach effectively enforces state variable and control-dependent state variable constraints of the forms (2.6) and (2.8); however, it introduces some numerical issues in the practical implementation of the algorithm.

Consider the basic EMS optimal design problem for a P2 HEV introduced in § 1.1. We want to set inequality constraints on the battery SOC σ such that it

does not exceed a lower and an upper threshold σ_{lb} and σ_{ub}:

$$\begin{cases} \sigma_k \leqslant \sigma_{lb}, \\ \sigma_k \geqslant \sigma_{ub}, \end{cases} \tag{3.9}$$

for all stages k[†].

[†] I.e. at any time t.

Now let us use (3.5) at some stage k to update the value function during the backward phase of the DP algorithm. Ideally, $V_{k+1}(\sigma_{k+1})$ should be equal to the minimum fuel consumption required to drive to the end of a driving mission[‡] for σ_{k+1} satisfying (3.9), and equal to infinity otherwise.

[‡] Or some other finite quantity with a physical meaning associated with our cost function.

In practice, since we use an interpolation scheme to obtain $V_{k+1}(\sigma_{k+1})$, its value will be influenced by the infinity value for all σ_{k+1} sufficiently close to σ_{lb} or σ_{ub}; for those values, the interpolation will be greatly inaccurate and the value function update will incorrectly penalize those states.

For example, let us name σ^n the smallest element of the SOC grid higher than σ_{lb}, as in Figure 3.1, and let us approximate the value function with a linear interpolation scheme. Let us use a large cost V_∞ to penalize unfeasible states. For all $\sigma_{k+1} = f_k(x_k, u_k)$ between σ^n and σ_{lb}, the value function V_{k+1} will be obtained by interpolating between $V_{k+1}(\sigma^n)$ and the cost penalty V_∞. The larger V_∞, the more inaccurate the approximation will be.

The problem can be strongly mitigated by selecting a penalty cost which is large enough with respect to typical values taken by the value function to effectively enforce the constraint but small enough that the interpolation inaccuracy at the state space boundary is kept to a minimum; the obvious drawback is that this requires a very thorough understanding of the physical system under investigation.

3.1.5 Treating fixed-endpoint problems

In general, there is no formal way to treat fixed-endpoint problems in a dynamic programming algorithm, nor to accommodate terminal state constraints. However, these can be accommodated by using the terminal cost $F(x_N)$ to penalize undesirable values of the terminal state, i.e. those values that do not meet our constraints.

When dealing with problems with fully discrete state and controls, one can simply exclude at the last stage all actions that do not lead to the desired terminal state, e.g. by assigning an infinite (or very large) terminal cost to all values that violate the constraints and this does not cause any particular issue.

Unfortunately, when we are dealing with continuous state variables with value function interpolation, several issues and limitations arise. The target terminal state cannot be a single point but it must be defined as a set because since the controls are either discrete or discretized, it would be impossible to hit a target point exactly. We will define this target set $T \subseteq \mathbb{R}^n$ as the set of all states for which all state variables x^i are within their lower and upper bound x^i_{lb} and x^i_{ub}[§]:

[§] an equivalent definition is found in [21].

$$T = \left\{ x \in \mathbb{R}^n \mid x^i_{lb} \leqslant x^i \leqslant x^i_{ub}, \quad i = 1, \dots, n \right\}. \tag{3.10}$$

We may then consider using an artificial cost to penalize all terminal states which lie outside of the target set. However, this will inevitably also affect the cost that we evaluate by interpolation for all nearby states.

Define the reachable state space at stage k as the space of all states from which it is possible to reach the terminal set at stage N using feasible controls only. Typically, the reachable state space will grow as we move backwards in time from the last stage[‖]. If we choose to characterize unfeasible states as having infinite cost,

[‖] For example, the reachable SOC range for a typical EMS design problem will grow based on the energy content of the battery and the maximum charge and discharge currents (including the engine and e-machine's limitation) that can drawn.

we can implicitly define the reachable state space at stage k as the space of all states for which the value function is finite:

$$R_k = \{x_k \mid V_k(x_k) < \infty\}. \tag{3.11}$$

The reachable state space then essentially propagates itself backwards in time during the backward phase, starting from $R_N = T$, with the value function update (3.5).

In practice, there are numerical issues that arise due to the discretization of the state space. When a control leads to a state that is very close to the reachable state space, the algorithm will attempt to evaluate the value function at $k+1$ by interpolating between a finite and an infinite value. More precisely, this happens when $f_k(x_k, u_k)$ produces at least one state x^i_{k+1} that lies between an unfeasible and a feasible gridded state.

When this happens, the value function V_{k+1} will be infinite as well even though the state belongs theoretically to the reachable state space. These states and the corresponding controls will be excluded from the value function update (3.5) and they are not allowed to have an influence on V_k. This negatively affects the backward propagation of the value function by unnecessarily excluding feasible controls and by artificially restricting the reachable state space. At the extreme, if the state discretization is very coarse, this may even cause the state space to tend to the null space.

The first way to mitigate this issue is to adopt a large number V_∞, rather than infinity, to penalize unfeasible states, as we discussed when dealing with state constraints. Even then, interpolation close to the boundary of the reachable space will be biased by V_∞; selecting a value that is large enough that the constraints are effectively enforced but small enough that the interpolation issues have a negligible impact may be a very challenging task and must be tailored to the specific optimal control problem at hand.

The problem can be treated more effectively by adopting either the *level set* method [21] or the *boundary line* method [78], which is restricted to scalar state spaces but may yield more satisfactory results. The boundary line method essentially involves pre-calculating the exact[†] boundary of the reachable state space for all stages, which is described by two boundary lines as the state is restricted to be scalar.

[†] In the sense that the state is not discretized.

The level set method uses a level set function for each stage to explicitly characterize the reachable state space and adds a level set update step in the backward phase of the dynamic programming algorithm. The level set functions thus constructed are then used in the forward phase to exclude controls that drive the state outside of the reachable space. Since the implicit characterization of the reachable state space via the value function is replaced by an explicit characterization using the level set functions, there is no need to adopt large penalties to enforce the terminal state constraint.

3.2 THE DYNAPROG TOOLBOX

As we saw in the previous chapters, the apparent simplicity of the dynamic programming principle hides many numerical issues that may arise when translating it into a computer algorithm. The first part of this thesis is mostly concerned with a discussion on how to treat and avoid these issues as well as mitigate the curse of dimensionality.

We also saw that, in general, one needs to tailor the algorithm to a specific application in order to obtain satisfactory results. However, it would not be reasonable to expect that every engineer wanting to exploit dynamic programming

for his own application of interest also becomes an expert in these highly specific topics.

Hence, there is a clear need for a software tool that guides the user in developing dynamic programming applications. For these reasons, a MATLAB toolbox called DynaProg was developed. This section presents some important implementation aspects of the toolbox and then details its inner working. Throughout this section, code listings are extracted from the source code. Note that, for the sake of readability, this listings do not include all code, but only the fundamental steps. The full source code is currently[†] hosted on a GitHub repository at https://github.com/fmiretti/DynaProg.

[†] At the time of this thesis's submission.

3.2.1 *Automatic expansion*

One of the fundamental steps of the backward phase of the DP algorithm is to evaluate the updated state variables $x_{k+1} = f(x_k, u_k)$ and the stage cost $L(x_k, u_k)$ for all gridded x_k and all admissible u_k.

Suppose that the grids for each of the n state variables x^i contains N_i elements and that each of the m control variables u^j is quantized into M_j elements. In any software implementation, a natural representation of x_{k+1} is then as a (n+m)-dimensional array with size $(N_1 \times N_2 \times ... \times N_n \times M_1 \times ... \times M_m)$. Each element of this array is obtained by one evaluation of the state dynamics $f(x_k, u_k)$, which is a user-supplied function.

In any programming language, these evaluations can be easily implemented using loop constructs. In MATLAB, it is far more computationally efficient to vectorize the function call. There are at least two ways to do this:

- Expanding the computational grids for x_k and u_k to (n+m)-dimensional arrays with size $(N_1 \times N_2 \times ... \times N_n \times M_1 \times ... \times M_m)$, and passing these as inputs to the function. We will collectively call these the *full* computational grids.
- Exploiting MATLAB's implicit array expansion.

The first approach is easily understood: we generate all possible combinations of the state and control variables belonging to their computational grids and we evaluate the function for all of these combinations. This approach is implemented, for example, in [79]. However, this approach can be inefficient, especially as the number of state and control variables grows.

In many control problems, not all variables influence all state dynamics and/or the cost. Consider for example, the applications in [74, 82, 40], where we have one state variable σ and one or more state variables to represent the temperature of one or more catalysts, say, T_{SCR}. Suppose also we have one control variable for the gear number γ and one to control the power-split α.

In this problem, each of two the state's dynamics is only influenced by that state itself and the two control variables. In other words, the dynamics of the two states are decoupled. Therefore, if we compute them on the full computational grid, we are wasting $N_1 \times (N_2 - 1) \times M_1 \times M_2$ evaluations of σ_{k+1} and $(N_1 - 1) \times N_2 \times M_1 \times M_2$ evaluations of $T_{SCR,k+1}$ in evaluating exactly the same values.

More generally, the system dynamics will be described by a sequence of equations. Not all these equations have to be evaluated on the full computational grid. Even in a simple application such as the one in § 4, there is a first sequence of steps that is entirely independent of the state and control variables; a second sequence that only depends on the gear number; a third sequence that depends on both control variables; and finally, a final sequence which also depends on the state of charge[‡].

[‡] I.e. the battery model.

It is most efficient to evaluate all these equations only for those combinations of the state and control grids that effectively influence their result, expanding them as needed. This second approach is one of the distinctive features of DynaProg. Rather than expanding x_k and u_k to the full computational grid, we represent the grid for each state variable x^i as an (n+m)-dimensional array were the only non-singleton dimension is the i-th dimension and the grid for each control variable u^j as an (n+m)-dimensional array were the only non-singleton dimension is the j-th dimension.

For example, in a problem with two state variables and one control variables, the grid for x_k^1 would be represented as an array with size $N_1 \times 1 \times 1 \times 1$, the grid for x_k^2 would be represented as an array with size $1 \times N_2 \times 1 \times 1$, and the grid for u_k^2 would be represented as an array with size $1 \times 1 \times 1 \times M_2$.

Then, DynaProg takes advantage of MATLAB's implicit array expansion to automatically expand the variables that are computed within the model function only when needed. Since most binary operators and functions support implicit expansion, this generally requires no code modification by the user. If uncommon functions are needed, these can generally be accommodated easily with little modification.

Additionally, DynaProg also allows to use the first approach using a dedicated option called *safe mode*.

3.2.2 *Exogenous inputs*

Until now, we have expressed the state dynamics as in (2.10):

$$x_{k+1} = f_k(x_k, u_k), \tag{3.12}$$

where f_k is a commonly used formulation that serves to cover all cases where the state dynamics change from one stage to another.

An alternative formulation is to write the state dynamics as being explicitly dependent on a third variable, the *exogenous inputs* w:

$$x_{k+1} = f(x_k, u_k, w_k)^{\dagger}. \tag{3.13}$$

[†]Note how this formulation closely resembles that of a time-varying continuous-time system $f(x, u; t)$.

Clearly, these exogenous inputs must be entirely independent of the state and control variables. This formulation is particularly useful in the context of EMS design, where the exogenous inputs can be used to encode a drive cycle, such as the vehicle's speed and acceleration.

In the same way, the running cost can be reformulate from a generic stage-dependent cost

$$L_k(x_k, u_k) \tag{3.14}$$

to a cost dependent on some exogenous inputs:

$$L(x_k, u_k, w_k). \tag{3.15}$$

The exogenous inputs are essentially external data that have to be loaded or somehow created by the user. An efficient algorithm must have some interface to efficiently pass them to the system dynamics and cost functions as needed with minimal overhead, allowing the user to generate them before the optimization algorithm is run.

3.2.3 *Additional inputs*

In any control problem of real-life interest, the state dynamics and running cost will be dependent on the state and control variables by means of some parameters or other data. For example, any powertrain model used for EMS design will

require a certain amount of data characterizing its components. This data may be needed in the form of simple parameters (e.g. a speed ratio or a constant efficiency) or more complex structures, such as interpolation data for fuel maps or torque limit characteristics.

In any case, this data has to be either created or loaded and then possibly processed before it can be used. A naive approach would be to include these operations in the same context[†] of the state dynamics. Considering that the state dynamics are evaluated by the dynamic programming algorithm a very large number of times, it is evident that this is a very questionable approach[‡].

[†] I.e. in the same function in the code.

[‡] Also, note that data loading operations are typically quite expensive in any programming language.

A much better approach is to generate all relevant parameters and data that are required to evaluate the system dynamics and cost but are not stage-dependent outside of the dynamic programming algorithm. One then needs a method for efficiently passing this data to the function that evaluates f and L.

3.2.4 Model split

The model split is an alternative to implicit expansion to reduce the computational effort by exploiting the problem's structure. The method consists in splitting the system dynamics into an *external*[§] and an *internal* function:

$$x_{k+1} = f_{int}(x_k, u_k, w_k, f_{ext}(u_k, w_k)). \tag{3.16}$$

[§] The reason why they are called this way is that it f_{ext} is run outside of the backward phase of the dynamic programming algorithm, while f_{int} is run within it.

The external function f_{ext} performs all calculations where the state variables are not involved, and it evaluates some *intermediate variables* v:

$$v_k = f_{ext}(u_k, w_k). \tag{3.17}$$

The internal function then evaluates the state update using these intermediate variables, the state variables (as well as u_k and w_k, if needed):

$$x_{k+1} = f_{int}(x_k, u_k, w_k, v_k). \tag{3.18}$$

Similarly, the stage cost can be split by redefining it as a function of the intermediate variables:

$$L(x_k, u_k, w_k, v_k). \tag{3.19}$$

The advantage of splitting the model function is that the external model function, being state-independent, can be run before the backward phase of the dynamic programming algorithm in order to generate the intermediate variables on computational grids that are at most $(M_1 \times ... \times M_m)$-dimensional, for each stage. Then, only the internal model is run on the full computational grids[‖]. Clearly, this approach is only beneficial if the automatic expansion method in § 3.2.1 is not used.

[‖] As defined in § 3.2.1.

3.2.5 The code

This section serves to illustrate the inner workings of the DynaProg toolbox. Although a brief introduction from the user perspective is provided, the main focus of this description is the structure of the source code itself, from a developer perspective. The objective is twofold:

1. to highlight the contributions brought about by this thesis work;
2. to serve as a guide for those researchers who wish to develop their own dynamic programming algorithms, either by branching DynaProg's code (taking advantage of its modular design), or by re-implementing some or all of its features in another software.

The core

DynaProg is implemented as a MATLAB class. An instance of DynaProg represents an optimal control problem; its properties define the algorithm settings and all relevant information that define the problem and methods are available to solve the problem and plot results.

Listing 3.1 shows how a simple optimal control problem is defined as a DynaProg problem. A point unit mass moving on a frictionless plane is subject to an external force $u(t)$ which must be controlled in order to drive the system from an initial position $s_o = 0$ to a final position $s_f = 0.7$ at rest (with initial and final speed $v_o = v_f = 0$) in 1 second, while minimizing the energy spent in applying the external force, i.e.

$$J(u) = \int_{t_o}^{t_f} u^2(t)\,dt. \tag{3.20}$$

The system's state is characterized by the mass position and speed, and the state dynamics are:

$$\dot{x}_1 = x_2, \tag{3.21}$$

$$\dot{x}_2 = u. \tag{3.22}$$

Listing 3.1: Creating a DynaProg problem.

```
%% Set up the problem
% State variables (position, speed) grid
x_grid = {0:0.01:1, -0.2:0.01:1.2};
% Initial state
x_init = {0, 0};
% Final state constraints
x_final = {[0.69, 0.71], [-0.02, 0.02]};
% Control variable (thrust) grid
u_grid = {-5:0.05:5};
% Number of stages (time intervals)
Nint = 10;

% Create DynaProg object
prob = DynaProg(x_grid, x_init, x_final, u_grid, Nint, @cart);

% Solve the problem
prob = run(prob);

% Set some other properties
prob.StateName = {'Position', 'Speed'};
prob.ControlName = 'Thrust';
prob.CostName = 'Energy';

% Plot results
figure
plot(prob);
```

To create the problem object, the class constructor `DynaProg` is called with its mandatory input arguments, that are the essential building blocks of the problem. One important input is a function handle to the *model function*, which in this example is the function `cart` shown in Listing 3.2. The model function is a user-supplied function which takes the state and control variables and exogenous inputs as input arguments and returns the updated state variables, the running cost and an *unfeasibility* array as outputs.

The unfeasibility array is a logical array used to set constraints by specifying which combinations of controls and states must be excluded, setting the corresponding element to `True`.

The model function can also accept additional inputs, as defined in § 3.2.3, and return additional outputs, for which the optimal trajectories are returned by DynaProg after the problem is solved.

Listing 3.2: Sample model function.

```
function [x_next, stageCost, unfeas] = cart(x, u, ~)
dt = 0.1;

x_next{1} = x{1} + x{2}.*dt;
x_next{2} = x{2} + u{1}.*dt;

% Stage cost
stageCost = (u{1}.^2).*dt;

% unfeasibility
unfeas = [];
end
```

The user can then solve the problem using the `run` method, after which he can retrieve the solution of the control problem[†] in the problem's properties and visualize results using the `plot` method. Additional properties can be specified either as additional arguments to the class constructor or later using dot notation, as shown in Listing 3.3 for setting some properties that affect the results plot appearance.

[†] I.e. the optimal state and control trajectories and corresponding cost.

Listing 3.3 lists the most important methods defined by the class. Except for the class constructor, the method definitions shown in Listing 3.3 only define their signature and their attributes[‡]; the method themselves are implemented in separate files with the same name. Not shown in the listing are also the properties' set/get methods.

[‡] Such as the `Hidden` attribute.

Listing 3.3: Main methods of DynaProg.

```
classdef DynaProg

⟨Properties declaration⟩

methods
    % Constructor method
    function obj = DynaProg(StateGrid, StateInitial, StateFinal,
        ControlGrid, Nstages, varargin)
        ⟨...⟩
    end
    % Methods in external files
    obj = run(obj)
    t = plot(obj)
end

% Methods in external files, hidden
methods (Hidden)
    obj = create_grids(obj)
    obj = create_intVars(obj)
    obj = backward(obj)
    obj = forward(obj)
    [states_next, stageCost, unFeas, addOutputs] = model_wrapper(obj,
         state, control, exoInput, IntermediateVars)
```

```
    [obj, cv_opt_idx, cost_opt] = updateVF(obj, k, states_next,
        stageCost, unFeas, vecdim_cv)
    [obj, cv_opt, exoInput, intVars_opt] = optimalControl(obj, k,
        state_next, stageCost, unfeas, vecdim_cv, intVars)
    obj = check_StateFinal(obj)
    obj = checkModelFun(obj, name, mode)
end
```

The method homonymous to the class is the class constructor. This method is invoked whenever an object is instantiated[†] and its inputs include both mandatory and optional arguments. The first five mandatory arguments are:

[†] I.e. a problem is created.

- `StateGrid`: the state variable(s) grid(s).
- `StateInitial`: the initial state.
- `StateFinal`: the target set(s) for the terminal state constraint(s), if present.
- `ControlGrid`: the control variable(s) grid(s).
- `Nstages`: the number of stages N.

In addition to these, there is one additional mandatory argument, that is the system and cost function. If the model split method § 3.2.4 is used, this is replaced by two mandatory arguments for the external and internal function names. In any case, the functions must be passed as function handles. Note that the argument `varargin` is used in MATLAB for specifying a variable number of inputs, in the forms of a cell array. In DynaProg, the first of these arguments is the model function, or if using the model split the first two.

The model split method is automatically enabled by passing two function handles as the sixth and seventh positional arguments (rather than just one as the sixth argument); this sets the hidden `UseSplitModel` property to `true`. In any case, all subsequent arguments must be name-value arguments.

The remaining arguments allow to set other properties by using a name-value pair syntax:

```
DynaProg(_, 'Property1', Value1, ..., 'PropertyN', ValueN)
```

The class constructor and the `run` and `plot` methods are the only non-hidden methods, and they constitute the user interface to the problem together with its public properties. The `run` method (Listing 3.4), is used to run a problem by calling the `create_grids`, `create_intVars` (if needed), `backward` and `forward` methods. The `plot` method, once a problem has been solved, can be used to quickly visualize the optimal state, control and cumulative cost trajectories.

Listing 3.4: `run` method.

```
function obj = run(obj)

% Create computational grids
obj = create_grids(obj);
if obj.UseSplitModel
    obj = create_intVars(obj);
end
% Generate value functions
obj = backward(obj);
% Generate optimal trajectories
obj = forward(obj);

end
```

The remaining methods are hidden methods, meaning that they are not normally visible by the user. The `backward` and `forward` methods are described in § 3.2.5 and § 3.2.5. If the model split method (§ 3.2.4) is used, the `create_intVars` method uses the external function to generate and store the intermediate variables. The methods `updateVF` and `optimalControl` are used in the backward and forward phase respectively to perform the value function update (3.5) and to evaluate the optimal control variable for the current stage $u_k^o(x_k)$ using (3.6).

The `create_grids` method creates the computational grids for the state and control variables that are needed at later stages. These are:

- `StateFullGrid`, `ControlFullGrid`: the full computational grids for the state and control variables; they are only created if safe mode is enabled.
- `StateGridCol`: the state grids in column vector form. This is needed for merely technical reasons to create the value function interpolants in the backward phase, due to MATLAB's built-in interpolants syntax.
- `ControlCombGrid`: the control variables expanded to $(M_1 \times ... \times M_m)$-dimensional grids. These are used in the forward phase to select the optimal control as well as other places if safe mode is enabled. We will sometimes refer to these grids as the *combined* control grids.

Additionally, `create_grids` initializes the terminal value function based on the problem's settings. First, the terminal value function is evaluated for the whole state grids using the terminal cost $F(x_N)$, which can be defined by the user using a function handle and assigning it to the property `TerminalCost`. If this property is left unspecified, $V_N(x_N)$ is set to zero for all x_N.

Then, a penalty term $\Psi(x_N)$ is added to enforce the terminal state constraints (if present):

$$V_N(x_N) = F(x_N) + \Psi(x_N). \tag{3.23}$$

There are currently two built-in methods to define $\Psi(x_N)$, which are selected by setting the `VFPenalty` property to either `'linear'` or `'rift'`. The `'rift'` option sets the penalty term to a large value V_∞ for all values of the terminal state that violate the terminal state constraints set by `StateFinal`:

$$\begin{cases} \Psi(x_N) = V_\infty & \text{if } x_N < x_{lb} \vee x_N > x_{ub}, \\ \Psi(x_N) = 0 & \text{otherwise.} \end{cases} \tag{3.24}$$

The term V_∞ can be modified by the user using the `myInf` property.

The `'linear'` option defines a linear penalty term proportional to the distance from the target set:

$$\begin{cases} \Psi(x_N) = V_\infty & \text{if } x_N < x_{lb} \vee x_N > x_{ub}, \\ \Psi(x_N) = p_\Psi^T \min\left((x_N - x_{lb})^{|\cdot|}, (x_N - x_{ub})^{|\cdot|}\right) & \text{otherwise}^\dagger. \end{cases} \tag{3.25}$$

The vector of proportionality factors p_Ψ can be manually set using the `VFPenFactors` property.

†Here, $v^{|\cdot|}$ denotes element-wise absolute value of all elements of the vector v.

The backward phase

The fundamental steps of the backward phase of the DP algorithm are: evaluating the updated state variables $x_{k+1} = f(x_k, u_k)$ and the stage cost $L(x_k, u_k)$, evaluating the value function for stage $k+1$ at state x_{k+1}, evaluating the value function at stage k by minimizing (3.5) for all gridded x_k, and constructing the corresponding approximating function.

In an initialization phase, the appropriate state and control grids are retrieved, depending on whether safe mode is enabled or not, and the dimensions corresponding to control variables are identified; these are later needed for the minimization in the value function update.

The main iteration of the backward phase then starts. First, the intermediate variables and exogenous inputs are retrieved (if present) or set to empty arrays. Then the user's model function is called using as inputs the state and control grids. The `model_wrapper` function is used to ensure that the model function is called correctly regardless of its signature[†]. This enables to handle the flexibility that DynaProg leaves to the user in defining his own model function(s).

This step produces the updated states $f(x_k, u_k)$[‡], stage cost $L(x_k, u_k)$ and unfeasibilties. If the unfeasibilities are not used by the user (and therefore are not an output of the model function), they are set to `false`. As explained in § 3.2.1, these outputs are not by default evaluated on the full computational grids; hence, they are expanded at this point if needed. Note that `states_next` is a cell array where each cell contains the values of $x_{k+1} = f(x_k, u_k)$ for each state variable.

[†]The first line of the function declaration, which defines the number of inputs and outputs.

[‡]Let us neglect the presence of exogenous inputs and/or intermediate variables.

Listing 3.5: The `backward` method.

```
function obj = backward(obj)
% Run the optimization algorithm backward phase

if obj.SafeMode
    state = obj.StateFullGrid;
    control = obj.ControlFullGrid;
else
    state = obj.StateGrid;
    control = obj.ControlGrid;
end

% Vector dimensions corresponding to CVs
vecdim_cv = (length(obj.N_SV)+1):(length(obj.N_CV)+length(obj.N_SV));

% Backward Loop
for k = obj.Nstages:-1:1

    intVars = ⟨Retrieve v_k⟩
    exoInput = ⟨Retrieve w_k⟩

    % State update
    [states_next, stageCost, unfeas] = model_wrapper(obj, state,
        control, exoInput, intVars);

    if ~obj.SafeMode
        % Expand updated states and unfeas to the combined grid
        for n = 1:length(states_next)
            states_next{n} = states_next{n} + zeros([obj.N_SV obj.
                N_CV]);
        end
        stageCost = stageCost + zeros([obj.N_SV obj.N_CV]);
        unfeas = unfeas | false([obj.N_SV obj.N_CV]);
    end

    % Enforce state grids
    if obj.EnforceStateGrid
        for n = 1:length(obj.N_SV)
            unfeas(states_next{n} > obj.StateGrid{n}(end) |
                states_next{n} < obj.StateGrid{n}(1)) = obj.myInf;
        end
```

```
    end

    % Update the value function
    [obj, cv_opt] = updateVF(obj, k, states_next, stageCost, unfeas,
        vecdim_cv);

    % Store cv map
    if obj.StoreControlMap
        obj.ControlMap = ⟨Store u_k^o(x_k)⟩
    end

end

end
```

The user-accessible setting `EnforceStateGrid` determines whether DynaProg should set a constraint on the state variables so that they do not exceed the bounds of the state grids.

Then, the `updateVF` method illustrated in Listing 3.6 is used to perform the value function update (3.5) and, if the level set method is enabled, the level set function update as well. If the level set method is disabled, the first step is to evaluate V_{k+1} (by interpolation) for $f_k(x_k, u_k)$, which was previously stored in a variable called `states_next`. The value functions are stored as interpolants[†] in a cell array `VF` with N cells; essentially, they can be used as functions whose arguments are the query points. The syntax `states_next{:}` is used to generate the arguments as a comma-separated list[‡].

This value is then summed to the running cost (thus obtaining $L_k(x_k, u_k) + V_{k+1}(f_k(x_k, u_k))$) and set to V_∞ for all unfeasible controls. This cost is then minimized over the indexes corresponding to the control variables to obtain the values of $V_k(x_k)$ for all gridded x_k (`cost_opt`). The final step is to use these values construct the value function interpolants. The `minfun` method is merely a wrapper for MATLAB's built-in `min` function that is used for compatibility if an older MATLAB release is detected.

If the level set method is enabled, a few more steps are needed for the level set function update, to determine the set of feasible controls $U_k^R(x_k)$ for which x_{k+1} is within the reachable state space at $k+1$, to evaluate the level set-minimizing controls $u_k^{L_{min}}$ for each gridded x_k, and finally to modify the value function update using $u_k^{L_{min}}$ when $U_k^R(x_k)$ is empty. All these steps are thoroughly described in [21].

The final step of Listing 3.5, if the user requires it by setting `StoreControlMap` property to true, is to store the optimal control map for the current stage:

$$u_k^o(x_k) = \underset{u_k \in U_k(x_k)}{\arg\min} \left[L_k(x_k, u_k) + V_{k+1}\left(f_k\left(x_k, u_k\right)\right)\right]. \tag{3.26}$$

[†] Using MATLAB's `griddedInterpolant` objects.

[‡] For example, if there are two state variables, `obj.VF{k+1}(states_next{:})` is equivalent to `obj.VF{k+1}(states_next{1}, states_next{2})`. See https://www.mathworks.com/help/matlab/matlab_prog/comma-separated-lists.html for more information about comma-separated lists.

Listing 3.6: The `updateVF` method.

```
function [obj, cv_opt_idx, cost_opt] = updateVF(obj, k, states_next,
    stageCost, unFeas, vecdim_cv)
% Update Level-Set function
if obj.UseLevelSet
    % Read L(k+1)
    LevelSet_next = obj.LevelSet{k+1}(states_next{:});
    % Set LevelSet_next to inf for the unfeasible CVs
    LevelSet_next(unFeas) = obj.myInf;
    % Update level-set function and find L-minimizing CVs
    [LevelSetValue, MinLevelSetCV] = obj.minfun(LevelSet_next,
        vecdim_cv);
```

```
    % Check if the set of reachable CVs U^R(x_k) is empty.
    isempty_UR = LevelSetValue>0;
    % Construct L approximating function for the current timestep
    obj.LevelSet{k} = griddedInterpolant(obj.StateGridCol, ...
        LevelSetValue, 'linear');
end

% Read VF(k+1)
VF_next =  obj.VF{k+1}(states_next{:});
cost = stageCost + VF_next;
% Set cost-to-go to inf for the unfeasible/unreachable CVs
cost(unFeas) = obj.myInf;
% Find optimal control as a function of the current state
[cost_opt, cv_opt_idx] = obj.minfun(cost, vecdim_cv);

if obj.UseLevelSet
    % For x_k s.t. U^R(x_k) is empty, calculate the VF based on the
        cv that minimizes the level-set function
    cost_MinLevelSetCV = cost(MinLevelSetCV);
    cost_opt(isempty_UR) = cost_MinLevelSetCV(isempty_UR);
end

% Construct VF approximating function for the current timestep
obj.VF{k} = griddedInterpolant(obj.StateGridCol, cost_opt, 'linear');

end
```

The forward phase

Similarly to the backward phase, the appropriate control grids are retrieved, depending on whether safe mode is enabled or not. Obviously, the state is not gridded and it is rather initialized to x_o.

In the forward simulation loop, the model is evaluated for the current state and the whole control grids, but first:

- the current state is expanded to the same size as the control grids if safe mode is enabled,
- the intermediate variables and external unfeasibilities are retrieved if the model split is used,
- the exogenous inputs are retrieved and also expanded if safe mode is enabled.

The model function is then evaluated with these inputs, generating the updated state x_{k+1}, the stage cost L_k and the unfeasibilities for all controls belonging to the control grids. In addition, x_{k+1}, L_k and the unfeasibilities are expanded to the combined control grids[†] if safe mode is enabled.

[†]As defined in §3.2.5, the combined control grids are the control grids expanded to $(M_1 \times \ldots \times M_m)$-dimensional grids.

Listing 3.7: The `forward` method.

```
function obj = forward(obj)
% Run the optimization algorithm forward phase

if obj.SafeMode
    control = obj.ControlCombGrid;
    % Vector dimensions corresponding to cvs
    vecdim_cv = 1:length(obj.N_CV);
else
```

```
    control = obj.ControlGrid;
    % Vector dimensions corresponding to cvs
    vecdim_cv = (1:length(obj.N_CV)) + length(obj.N_SV);
end

% Initialize the state
state = obj.StateInitial;

for k = 1:obj.Nstages

    % Expand current state to the combined cv grid
    if obj.SafeMode
        for n = 1:length(state)
            state_exp{n} = state{n} + zeros(size(obj.ControlCombGrid
                {1}));
        end
    else
        state_exp = state;
    end

    intVars = ⟨Retrieve v_k⟩
    exoInput = ⟨Retrieve w_k⟩

    % Evaluate state update and stage cost
    [state_next, stageCost, unfeas] = model_wrapper(obj, state_next,
        control, exoInput, intVars);

    % Expand updated states and unfeas to the combined cv grid
    if ~obj.SafeMode
        for n = 1:length(state_next)
            state_next{n} = state_next{n} + zeros([ones(1, length(obj
                .N_SV)) obj.N_CV]);
        end
        stageCost = stageCost + zeros([ones(1, length(obj.N_SV)) obj.
            N_CV]);
        unfeas = unfeas | false([ones(1, length(obj.N_SV)) obj.N_CV])
            ;
    end

    % Find the optimal cvs
    [obj, cv_opt, intVars_opt] = optimalControl(obj, k, state_next,
        stageCost, unfeas, vecdim_cv, intVars);

    % Advance the simulation
    [state, stageCost_opt, unfeas_opt, addout] = model_wrapper(obj,
        state, cv_opt, exoInput, intVars_opt);

    % Update the profiles
    obj.StateProfile(:,k+1) = ⟨Append state⟩
    obj.ControlProfile(:,k) = ⟨Append cv_opt⟩
    obj.CostProfile(k) = stageCost_opt;
    if ~isempty(addout)
            obj.AddOutputsProfile(:,k) = ⟨Append addout⟩
    end
end
end
```

At this point, the `optimalControl` method, shown in Listing 3.8, is used to obtain the optimal control variable for the current stage $u_k^o(x_k)$ using (3.6). If

the level set method is disabled, the first steps are to evaluate the value function $V_k(x_{k+1})$ (by interpolation), add the stage cost L_k and set this sum to V_∞ for unfeasible controls. Then, the index of the controls that minimize this sum is found and used to extract the optimal controls. A final step is also required to extract the corresponding values of the intermediate variables, if the model split is used.

If the level set method is enabled, additional steps are required to check whether the set of feasible controls that lead to reachable states $U_k^R(x_k)$ is empty, and to set u_k^o to the level-set minimizing controls $u_k^{L_{min}}$ if this is the case.

Listing 3.8: The `optimalControl` method.

```
function [obj, cv_opt, intVars_opt] = optimalControl(obj, k,
    state_next, stageCost, unfeas, vecdim_cv, intVars)
%optimaControl find optimal controls for the current stage

% Get level set - minimizing cv
if obj.UseLevelSet
    % Read L(k+1)
    LevelSet_next = obj.LevelSet{k+1}(state_next{:});
    % Set LevelSet_next to inf for the unfeasible cvs
    LevelSet_next(unfeas) = obj.myInf;
    % Determine if U^R(x_k) is empty
    isempty_UR = all(LevelSet_next(:) > 0);
end

% Read VF(k+1)
VF_next =  obj.VF{k+1}(state_next{:});
cost = stageCost + VF_next;
if obj.UseLevelSet
    cost(unfeas) = obj.myInf;
    cost(LevelSet_next > 0) = obj.myInf;
end
% Set cost-to-go to inf for the unfeasible cvs
cost(unfeas) = obj.myInf;

% Find optimal control as a function of the current state
[~, index_opt] = obj.minfun(cost, vecdim_cv);
if obj.UseLevelSet
    % If no reachable cv was found (isempty_UR), then use the L-
      minimizing u
    [~, MinLevelSetCV] = obj.minfun(LevelSet_next, vecdim_cv);
    index_opt(isempty_UR) = MinLevelSetCV;
end
cv_opt =  cellfun(@(x) x(index_opt), obj.ControlCombGrid, '
    UniformOutput', false);

% Extract the intermediate variables for the optimal cv
intVars_opt = ⟨Retrieve w_k(u_k^o)⟩

end
```

4 *Powertrain modeling for dynamic programming*

As we saw in § 1.1, the energy management strategy of a hybrid electric vehicles is usually tasked with controlling at least one control variable that defines the powerflow from the different power sources to the wheels[†].

[†] Possibly, along with other control variables such as the transmission's gear number.

In most cases, the choice of this control variable is not unique and is an important decision for an engineer working on EMS design. In fact, various different models can be found in the literature for this type of architecture. The most popular choices appear to be the engine torque [44, 5, 75, 39, 3], the e-machine torque [45], the e-machine power [67, 33], the battery power [43], an engine torque-split factor, or an e-machine torque-split factor [80].

When we set up the EMS design as an optimal control problem, all these choices alter the structure to some extent and they interact differently with the numerical implementation of any algorithm used to obtain the solution. In this chapter, we want to investigate the interaction between these modeling choices and a dynamic programming algorithm.

As a case study, a p2 parallel hybrid[‡] was identified, whose main parameters are reported in Table 4.1, and modeled with eight different control sets. Dynamic programming was then used for all models to investigate the fuel-optimal EMS.

[‡] We selected the p2 architecture as it is arguably the most commonly studied in the literature.

Part of the contents of this section have been accepted for publication in [57].

Component	Parameter	Value
Vehicle	Mass	1175 kg
	First coast-down coefficient	150 N
	Second coast-down coefficient	2.24 N/(ms)
	Third coast-down coefficient	0.44 N/(ms)^2
	Tyre radius	0.3 m
Engine	Displacement	1.1 l
	Rated power	68 kW
	Maximum torque	130 Nm
E-machine	Rated power	30 kW
	Maximum torque	150 Nm
Battery	Type	Li-ion
	Nominal capacity	5.4 Ah
	Nominal voltage	204 V

Table 4.1: Main vehicle data.

4.1 Simulation model

The scope of this comparison is to compare different modeling choices that can be adopted to describe the power flow in a parallel hybrid.

The eight models characterize the power flow with the following control variables:

A) The engine torque T_{eng}.

B) The e-machine torque T_{em}.

C) The battery current i_b.

D) The normalized engine torque τ_{eng}.

E) The normalized e-machine torque τ_{em}.

F) The normalized battery current ι_b.

G) The engine torque-split factor α_{eng}.

H) The e-machine torque-split factor α_{em}.

Clearly, each one of these control sets introduces changes in the powertrain model. In particular, models based on the battery current, i.e. models C)) and F)), use different equations for the battery model.

Furthermore, each of these control sets requires a different characterization in terms of lower and upper bounds and different constraints to be enforced.

Nonetheless, all models share a common path up to the evaluation of the torque demand. A longitudinal vehicle model was used to evaluate a tractive effort F_{veh} as a function of the vehicle speed v_{veh}. The control effort is then propagated to evaluate a torque demand at the transmission input. Finally, this torque demand is split between the engine and the e-machine based on the power flow control variable.

Omitting the various driveling efficiencies for ease of notation, the torque demand was evaluated as:

$$T_d = \frac{F_{veh}(v_{veh})r_{wh}}{\tau_{fd}\tau_{gb}(\gamma)}, \tag{4.1}$$

where r_{wh} is the wheels' radius, τ_{fd} and τ_{gb} are the final drive and gearbox speed ratios, and γ is the gear number.

The battery was modeled with an internal resistance equivalent circuit model, where the battery power P_b and the battery current i_b are related to each other as follows:

$$P_b = v_b i_b = \left(v_{oc}(\sigma) + R_{eq}(\sigma) i_b\right)\, i_b. \tag{4.2}$$

Note that the open-circuit voltage and internal resistance characteristics $v_{oc}(\sigma)$ and $R_{eq}(\sigma)$ were characterized as SOC-dependent. The battery SOC (σ) is the only state variable for all models.

The engine fuel consumption and e-machine efficiency were obtained by linear interpolation on quasi-static map as a function of their speed and torque, as is common in these sort of models:

$$\dot{m}_f = \dot{m}_f(\omega_{eng}, T_{eng}), \tag{4.3}$$
$$\eta_{em} = \eta_{em}(\omega_{em}, T_{em}). \tag{4.4}$$

The fuel consumption constitutes the running cost of the optimal control problem, so that

$$J(\sigma_o) = \Delta t \sum_{k=o}^{N-1} \dot{m}_f(\omega_{eng,k}, T_{eng,k}). \tag{4.5}$$

4.2 Evaluation criteria

The goal of this section is to provide insight on the advantages of disadvantages of the listed simulation models. To structure our discussion, we identified four areas.

Control bounds definition. To apply dynamic programming, we need to discretize and define lower and upper bounds for all the continuous control variables. For some control sets, these bounds are obvious; for others, they are not. We will sometimes refer to the former as *well-posed* control set bounds.

Numerical efficiency. For each simulation time interval, we exclude all controls that end up violating the constraints that we set on the powertrain components. These controls do not contribute to the value function update. The more controls we have to exclude because of our constraints, the more numerically inefficient the control set is.

Model complexity. The models which directly control the battery current use simpler equations for the battery model, which translate into reduced simulation time.

Interpretability. As engineers, we would like to have a direct correspondence between the value taken by our power flow control variable and the HEV operating modes (i.e. pure electric, torque-split, pure thermal and battery charging)[†]. This may also be very relevant if the results are to be used by some rule-extraction algorithm to obtain an heuristic strategy or to train a machine-learning based strategy[‡].

[†]Pure electric includes regenerative braking, which is the only allowed operating mode when the vehicle is braking.

[‡]In particular, classification algorithms.

4.3 Control sets

We now turn our attention to the definition of the control sets and we discuss their individual advantages and disadvantages.

The normalized control sets, i.e. sets D) to F), are different from models A) to C) in that the control variables are normalized by their maximum values, as imposed by the operational limits of the corresponding components. The torque-split factors G) and H) are defined as the ratio between the engine or e-machine torque and the torque demand (4.1).

4.3.1 *Engine torque and normalized engine torque*

This model has no particular advantages, if not for the fact that no derived quantity needs to be defined.

This control set obviously has a lower bound at $T_{eng} = 0$, and an upper bound at the engine maximum torque. However, the maximum torque is strongly speed-dependent; therefore, the upper bound for the control set must be set to the absolute maximum engine torque, which is available at some speed ω^*_{eng}.

Then, a constraint must be set on the engine torque:

$$T_{eng} \leqslant T_{eng,max}(\omega_{eng}). \tag{4.6}$$

At all times where the engine speed is different from this ω^*_{eng}, we are wasting computations on unfeasible controls.

After setting the engine torque, the e-machine torque is simply evaluated as

$$T_{em} = \frac{T_d - T_{eng}}{\tau_{tc}}. \tag{4.7}$$

The, the e-machine power and, subsequently, the battery power can be computed. The battery current must then be evaluated by solving (4.2), which is quadratic in i_b:

$$i_b = \frac{v_{oc} - \sqrt{v_{oc}^2 - 4R_{eq}P_b}}{2R_{eq}}. \tag{4.8}$$

This is the most computationally expensive equation in the whole powertrain model.

An additional consideration on the lower bound is also in order. Here, we implicitly assumed that the EMS prevents the engine from working at negative torque. For powertrain architectures with a low hybridization ratio, this may not be a reasonable assumption; in that case, the engine torque should be limited to its motoring torque[†] $T_{mot}(\omega_{eng})$, which is strongly speed dependent, leading to the same sort of issues that we have just discussed for the upper torque limit.

[†]The torque absorbed due to pumping and friction losses when no fuel is injected.

Issues related to the definition of the control bounds can be easily overcome by defining a normalized engine torque:

$$\tau_{eng} = \frac{T_{eng}}{T_{eng,max}(\omega_{eng})}. \tag{4.9}$$

With this definition, and by setting $\tau_{eng} = [0, 1]$, the dynamic programming algorithm will never waste time in exploring control variables that are unfeasible because they exceed the maximum engine torque and we can get rid of the corresponding constraint.

Finally, we turn our attention to regenerative braking. With these two control sets, there is no way to directly control the amount of torque demand that gets absorbed by the e-machine to charge the battery. This operating mode is implicitly defined by setting T_{eng} or τ_{eng} to zero and then either let all of the torque demand be absorbed by the e-machine or adopt some other simple rule (i.e. only absorbed a fixed or speed-dependent share).

For the sake of simplicity, we set the e-machine torque in regenerative braking to be only limited by its torque limit curve. In other words, when the torque demand is negative, T_{em} is saturated by:

$$T_{em} = \max\left(\frac{T_d}{\tau_{tc}}, T_{em,min}(\omega_{em})\right). \tag{4.10}$$

Note that $T_{em,min}$, the limit torque in generator mode, is negative by convention.

4.3.2 *E-machine torque and normalized e-machine torque*

This model is analogous to the previous one, and it shares a similar issue in that the maximum torque is speed-dependent; but since the e-machine has two maximum torque curves (for generator and motor mode), the issue affects both the lower and the upper bound for the control set.

The control set must be bounded at the rated minimum and maximum torque. Then, whenever the e-machine is working in the constant power region, the we are wasting computations on unfeasible controls.

Issues related to the definition of the control bounds can be easily overcome by defining a normalized e-machine torque:

$$\tau_{em} = \begin{cases} \frac{T_{em}}{T_{em,min}(\omega_{em})} & \text{if } \tau_{em} < 0, \\ \frac{T_{em}}{T_{em,max}(\omega_{em})} & \text{if } \tau_{em} \geqslant 0. \end{cases} \tag{4.11}$$

With this definition, and by setting $\tau_{em} = [-1, 1]$, the dynamic programming algorithm will never waste time in exploring control variables that are unfeasible because they exceed the e-machine torque limits.

With these two control sets, the amount of torque demand that gets absorbed by the e-machine in regenerative braking can be directly controlled. However, this is not really an advantage: clearly, the optimal decision when the torque demand is negative is to regenerate as much as possible, as for the engine-based models, rather than using the highest feasible torque within our discrete control set. This also ensures a fair comparison with the engine torque-based models.

4.3.3 Battery current and normalized battery current

An important difference between this model and the preceding ones is that directly controlling the battery current means that the battery model no longer requires solving a quadratic equation; rather, the battery power is directly computed from (4.2) and then translated into the e-machine torque which in turn determines the engine torque.

Thus, the most expensive computation of the powertrain model is avoided resulting in a slightly faster-running model.

Note that, since we are setting the battery power based on a control variable and then evaluating the consequent e-machine torque rather than the other way around, we need a characterization of the e-machine efficiency as a function of its speed and electrical power. Usually, the e-machine efficiency is characterized as a function of speed and mechanical torque; transforming from one form to the other is straightforward and should obviously be done outside of the control algorithm.

Whether the lower and upper bounds definition requires the same care as for the previous models depends on how the battery current is limited as well as on the battery technology and performance.

In some cases, the limit current may be determined by thermal or aging aspects, and can generally be set as a constant. In some other cases, the limiting factor may be the battery voltage limits. Then, if the open-circuit voltage and internal resistance are SOC-dependent, so are the limit currents:

$$i_{b,max}(\sigma) = \frac{v_{oc}(\sigma) - v_{b,min}}{R_{eq}(\sigma)}, \tag{4.12}$$

$$i_{b,min}(\sigma) = \frac{v_{oc}(\sigma) - v_{b,max}}{R_{eq}(\sigma)}. \tag{4.13}$$

In these cases, issues related to the definition of the control bounds arise as for the engine and e-machine torque control sets and they can be easily overcome by defining a normalized current:

$$\iota_b = \begin{cases} \frac{i_b}{i_{b,min}(\sigma)} & \text{if } \iota_b < 0, \\ \frac{i_b}{i_{b,max}(\sigma)} & \text{if } \iota_b \geqslant 0. \end{cases} \tag{4.14}$$

With this definition, and by setting $\iota_b = [-1, 1]$, the dynamic programming algorithm will never waste time in exploring control variables that are unfeasible because they exceed the e-machine torque limits.

Regenerative braking is particularly troublesome with these models. In a naive approach, one could simply select the highest current within the control set that meets the battery and e-machine constraints. However, this would make this model unable to reproduce the same behavior as the other ones, since this value of the current will not correspond to the maximum torque the e-machine can absorb.

Suppose that we want to use the approach we have previously used instead. First, we would saturate the e-machine torque to its generator limit torque, and

we would then use (4.8) to evaluate the corresponding maximum charge current; then, we would limit the battery current to this. Doing this however means that we are essentially giving up on the motivating factors for using this model, that is to avoid using (4.8).

An alternative is to pre-calculate a minimum current $\tilde{i}_{b,min}$ that incorporates the e-machine torque limits:

$$\tilde{i}_{b,min}(\sigma, \omega_{em}) = \max \left[i_{b,min}(\sigma), \frac{v_{oc} - \sqrt{v_{oc}^2 - 4R_{eq}\tilde{P}_b}}{2R_{eq}} \right], \tag{4.15}$$

where

$$\tilde{P}_b = \eta_{em}\left(\omega_{em}, T_{em,min}(\omega_{em})\right) \omega_{em} T_{em,min}(\omega_{em}), \tag{4.16}$$

and use that to saturate the battery current. This is the approach that was used for this work.

Even with this trick, it is still not possible to saturate the e-machine torque to the torque demand in regenerative braking and a constraint must be set so that

$$T_{em} \geqslant \frac{T_d}{\tau_{tc}}. \tag{4.17}$$

4.3.4 *Engine torque-split factor*

The engine torque-split factor is defined as the ratio between the engine torque and the torque demand:

$$\alpha_{eng} = \frac{T_{eng}}{T_d}. \tag{4.18}$$

The main advantage of the engine torque-split factor is in its interpretability, in that any value for α_{eng} can be attributed to one of the HEV operating modes as shown in Table 4.2.

One issue with this control set is while there is an obvious lower bound for α_{eng} (i.e. $\alpha_{eng} = 0$), there is no obvious upper bound:

- at times when T_d is small compared to $T_{eng,max}$, a large upper bound would be needed to enable using the engine up to its full power to recharge the battery and a coarse discretization would suffice;
- when T_d is close to $T_{eng,max}$, a small upper bound would be enough but a finer discretization would be needed;
- if T_d is larger than $T_{eng,max}$, an even smaller upper bound (smaller than 1) would suffice.

As a result, this control set generally leads to wasting computations on unfeasible controls whenever the torque demand is relatively high and to an unnecessarily restricted control set for battery charging when the torque demand is relatively low.

Regenerative braking is easily dealt with in the same manner as the engine torque-based models.

value	operating mode
$\alpha_{eng} = 0$	pure electric
$0 < \alpha_{eng} < 1$	torque-split
$\alpha_{eng} = 1$	pure thermal
$\alpha_{eng} > 1$	battery charging

Table 4.2: Correspondence between the engine torque-split factor and the HEV operating mode

4.3.5 *E-machine torque-split factor*

The e-machine torque-split factor is defined as the ratio between the e-machine torque and the torque demand:

$$\alpha_{em} = \frac{T_{em}}{T_d}. \tag{4.19}$$

Similarly to the engine-torque split factor, this control set has an unambiguous relation to the HEV operating modes, as shown in Table 4.3; furthermore, for similar reasons, there is no obvious lower bound.

value	operating mode
$\alpha_{em} < 0$	battery charging
$\alpha_{em} = 0$	pure thermal
$0 < \alpha_{em} < 1$	torque-split
$\alpha_{em} = 1$	pure electric

Table 4.3: Correspondence between the e-machine torque-split factor and the HEV operating mode

Regenerative braking is easily dealt with in the same manner as the e-machine torque-based models.

4.4 Simulation results

To test the ideas discussed in the previous section, all the listed models were implemented and tested with the DynaProg toolbox. We thus compared the control sets in terms of accuracy (which is a product of their numerical efficiency and of the well-posedness of their bounds) and computational time (which is a product of the model complexity).

For all simulations, the SOC grid was defined as ranging from 0.4 to 0.7 with a discretization step of 0.003[†]. The computational grid for power flow control variable was defined by a number m_{PF} of 11, 21 and 61 quantized values in three sets of experiments. Therefore, all simulations had the same number of function evaluations. Model accuracy was measured in terms of the model's ability to achieve the true optimal[‡] fuel consumption while reaching the terminal SOC of $\sigma = 0.6$.

The gear number was set by a simple gear shift schedule as a function of the vehicle speed, to ensure that all models deal with the same torque demand. We will say more about this in § 4.5.

[†] I.e. with $n_\sigma = 101$ values.

[‡] Obtained by running a simulation with extremely fine discretization grids, with $n_\sigma = 2001$ and $m_{PF} = 2001$.

	Model	$m_{PF} = 11$			$m_{PF} = 21$			$m_{PF} = 41$		
		Δm_f	$\psi(\sigma_N)$	t_{sim} (s)	Δm_f	$\psi(\sigma_N)$	t_{sim} (s)	Δm_f	$\psi(\sigma_N)$	t_{sim} (s)
A)	T_{eng}	1.94 %	9.2e-04	2.5	1.26 %	1.3e-03	2.6	0.74 %	1.4e-03	2.9
B)	T_{em}	13.15 %	1.0e-02	2.8	2.15 %	4.9e-03	2.8	1.48 %	5.7e-03	3.0
C)	i_b	3.53 %	3.5e-03	2.8	1.99 %	3.2e-03	3.3	1.17 %	5.2e-03	3.3
D)	τ_{eng}	1.95 %	1.3e-03	2.6	1.25 %	1.2e-03	2.6	0.71 %	1.2e-03	2.9
E)	τ_{em}	2.28 %	4.3e-03	2.6	1.52 %	5.1e-03	2.6	0.95 %	4.7e-03	2.9
F)	ι_b	3.53 %	4.4e-03	3.0	2.01 %	3.8e-03	3.1	1.09 %	3.2e-03	3.3
G)	α_{eng}	1.07 %	4.3e-03	2.8	0.61 %	4.4e-03	2.6	0.32 %	3.2e-03	2.8
H)	α_{em}	0.91 %	1.1e-03	2.5	0.45 %	1.0e-03	2.7	0.33 %	1.0e-03	2.8

Table 4.4: Accuracy and simulation time of the examined models.

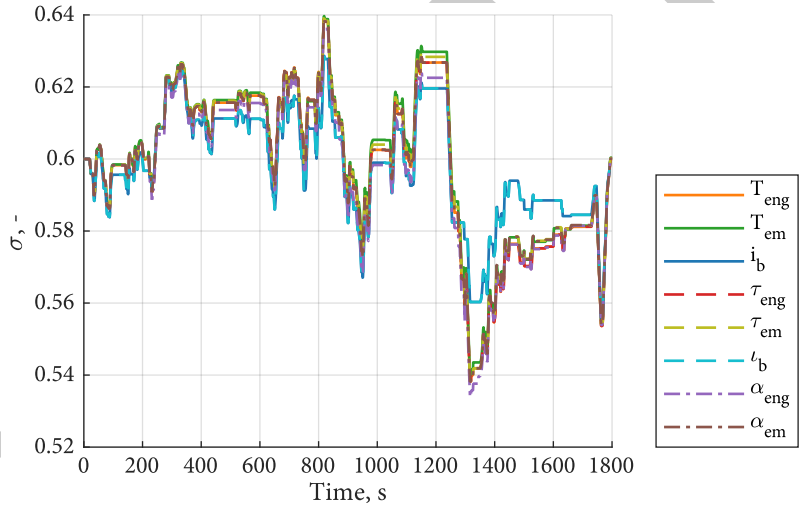


Figure 4.1: SOC trajectories for all models, with $m_{PF} = 2001$.

Accuracy of the models is therefore reported in Table 4.4 for all tested models in terms of two quantities: the difference between the fuel consumption and the true optimal fuel consumption Δm_f and the fixed-endpoint error $\psi(\sigma_N)$ (i.e. the difference between the terminal SOC and the desired value). A third column reports simulation time (t_{sim}). Simulations where the dynamic programming algorithm was unable to find a feasible solution are marked as failed.

The simulation results shown in Table 4.4 raise many points which are worth discussing. Firstly, the torque-split models appear to be the most robust in that they allowed to find a feasible solution with accuracy within 1% even with a coarse discretization grid.

Their good performance is probably explained by the fact that they can accurately reproduce both pure thermal and pure electric modes, thus allowing for stable operation.

As control set discretization is refined all models tend to the same solution, both in terms of fuel consumption and in terms of the state trajectory as shown in Figure 4.1.

Furthermore, inspecting this optimal solution, we also note that it includes some lengthy portions where the optimal operating mode is pure thermal[†] and

[†]Especially in the extra-high phase of the WLTC.

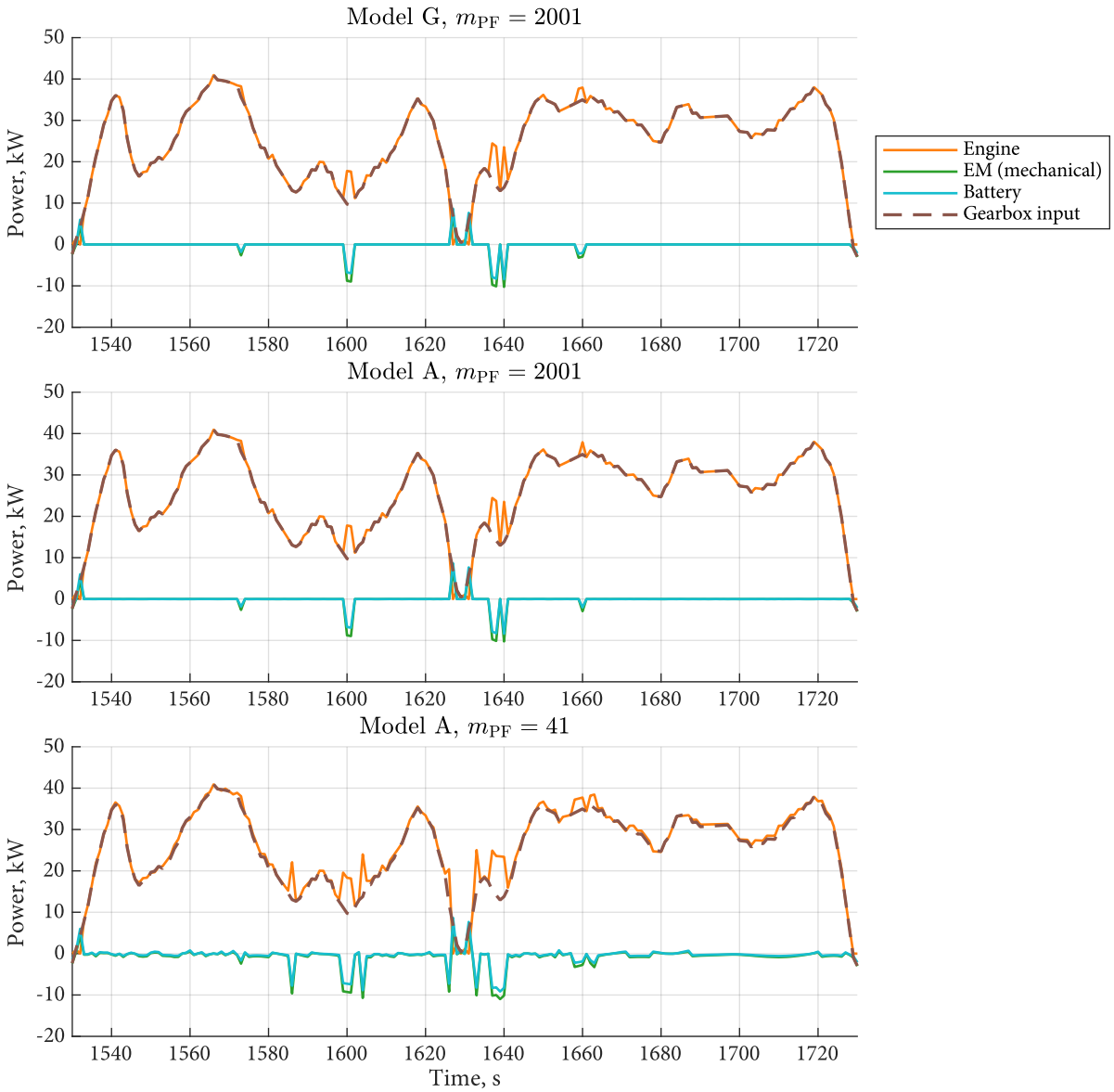


Figure 4.2: Pure thermal segment. The top plot shows model G) with a fine discretization ($m_{PF} = 2001$), which serves as a reference. The middle plot shows how model A) tends to the same solution given the same fine discretization. The bottom plot shows how the same model fails to reproduce the same pure thermal segments accurately with $m_{PF} = 41$.

shorter but frequent portions where the optimal operating mode is pure electric.

However, as we saw earlier, the torque-split models are the only two that can exactly match both these two operating modes. Consider for example the segment shown in Figure 4.2. The figure shows the engine, e-machine and battery power as well as the power demand at the gearbox input.

In this segment, the optimal solution involves going in pure thermal, and both torque-split models are able to accurately reproduce this behavior. When using an extremely fine discretization, the engine-torque based models A) and D) are able to match this behavior almost exactly, as shown in Figure 4.2 for model A).

As we reduce the discretization to $m_{PF} = 41$, we start seeing how the same model is unable to run in pure thermal, although it attempts to do so by making the e-machine torque as small as possible (given the control set quantization).

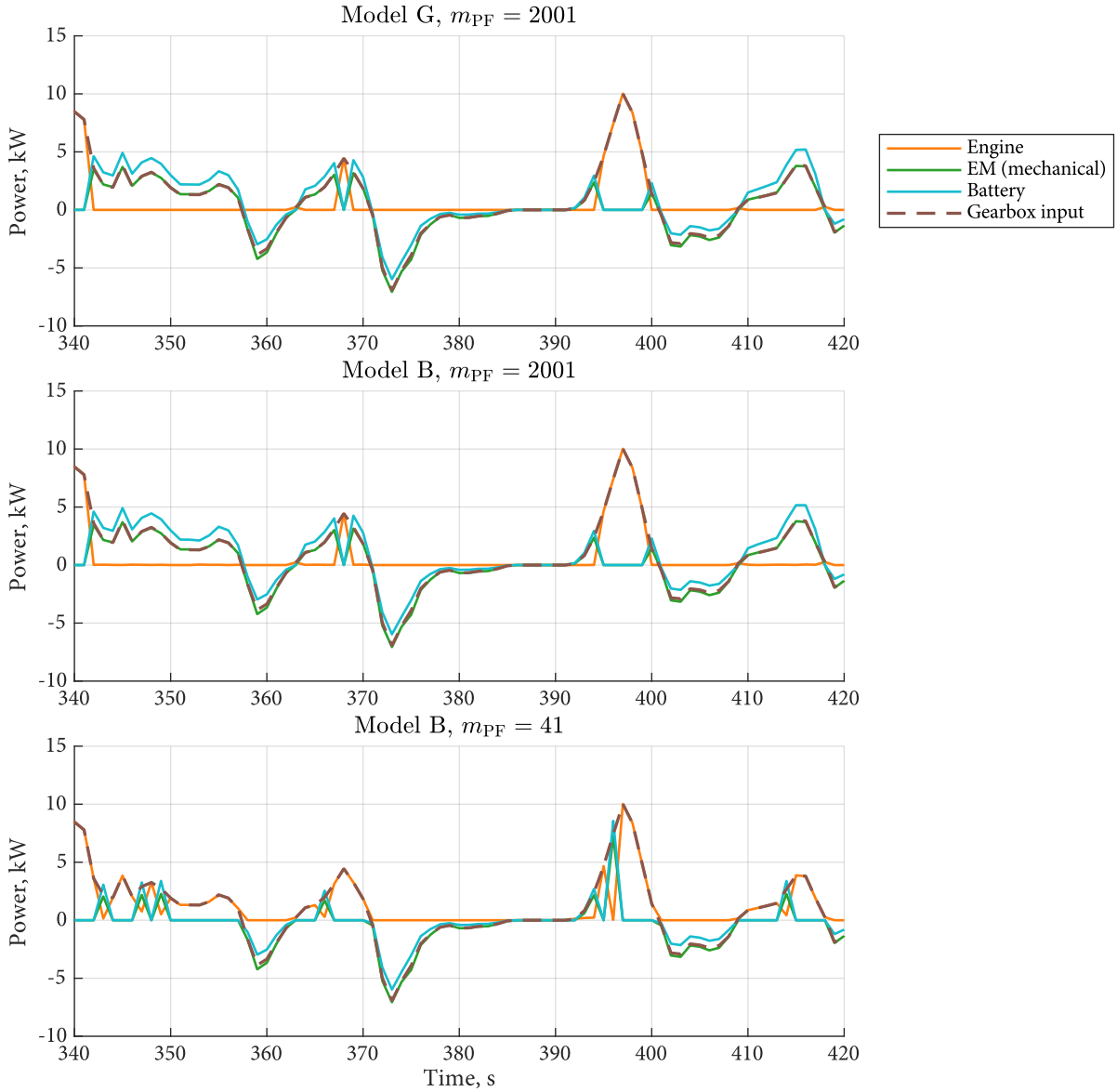


Figure 4.3: Pure electric segment. The top plot shows model G) with a fine discretization ($m_{PF} = 2001$), which serves as a reference. The middle plot shows how model B) tends to the same solution given the same fine discretization. The bottom plot shows how the same model fails to reproduce the same pure electric segments accurately with $m_{PF} = 41$.

A similar issue affects the e-machine- and battery current-based models in that they cannot exactly reproduce pure electric operation. This time, let us consider the segment shown in Figure 4.3, where the optimal solution involves many pure electric portions. While the solution for model B) with $m_{PF} = 2001$ is almost identical, since the fine discretization allows it, the solution for $m_{PF} = 41$ looks very different as the model simply cannot well approximate pure electric.

Instead, it would have used the engine to meet the torque demand, and it does so in an inefficient way as an engine's efficiency is typically low at low load; to the point where it is sometimes more convenient to just run in pure thermal. A clear example of this can be seen by comparing the segments going from 344 to 358 seconds in Figure 4.3.

In addition to that, the batter current-based models are also less effective at

	Model	$m_{PF} = 61$			$m_{PF} = 121$			$m_{PF} = 181$		
		Δm_f	$\psi(\sigma_N)$	t_{sim} (s)	Δm_f	$\psi(\sigma_N)$	t_{sim} (s)	Δm_f	$\psi(\sigma_N)$	t_{sim} (s)
A)	T_{eng}	0.54 %	9.9e-04	3.0	0.31 %	1.0e-03	3.3	0.24 %	1.0e-03	3.6
B)	T_{em}	1.14 %	5.1e-03	3.3	0.77 %	5.2e-03	3.7	0.45 %	2.8e-03	4.0
C)	i_b	0.80 %	3.5e-03	3.7	0.48 %	4.1e-03	4.5	0.34 %	3.7e-03	5.2
D)	τ_{eng}	0.51 %	1.0e-03	3.0	0.31 %	1.0e-03	3.4	0.24 %	1.1e-03	3.8
E)	τ_{em}	0.75 %	5.1e-03	2.9	0.44 %	3.6e-03	3.5	0.38 %	5.0e-03	3.7
F)	ι_b	0.79 %	3.4e-03	3.7	0.46 %	3.9e-03	4.8	0.33 %	3.5e-03	4.9
G)	α_{eng}	0.26 %	3.4e-03	3.0	0.18 %	3.4e-03	3.5	0.17 %	3.9e-03	3.6
H)	α_{em}	0.27 %	1.0e-03	3.0	0.19 %	1.0e-03	3.4	0.20 %	1.0e-03	3.6

Table 4.5: Accuracy and simulation time of the examined models, with finer control set discretization.

regenerative braking, as it is not possible to saturate the e-machine torque to the torque demand. Hence, when the limiting factor is the torque demand and not one of either the e-machine limit torque or the maximum charge current, the current-based models lose some energy with respect to the other ones. This explains why they generally perform worse than model E).

What is less expected is that they are also unable to provide any reduction in computational time. On the contrary, they are the worst performing models in this aspect.

Inspection of the code performance using a dedicated tool[†] revealed that the two most time-consuming operations for these models were the interpolations required to evaluate the e-machine efficiency and engine fuel consumption.

[†] MATLAB's profiler.

While for all other models these two are only a function of the exogenous inputs[‡] and control variables, the fact that we saturate the battery current using $\tilde{\imath}_{b,min}$ defined by (4.15), which is also SOC-dependent, means that we have to do a significantly higher number[§] of interpolations on the two maps.

[‡] The vehicle speed and acceleration.

[§] Precisely n_σ times more.

The corresponding increase in computational cost is enough to overcome the saving induced by avoiding the computation of the battery current with (4.8), as is further confirmed by the simulation results shown in Table 4.5, where two refined control set grids were employed.

4.5 Including the gear number in the control set

In the simulations performed so far, the powerflow control variable was set as the only control variable, while gear number was set with a speed-dependent gearshift schedule. This was done to ensure that all models had to deal with the same torque demand.

If the gear number is set as a second control variable, additional differences between the models may be introduced. For example, the engine torque-based models and torque-split models are able to disengage the engine clutch and run in pure electric with a high gear, whether this is desirable or not. The remaining models on the other hand cannot disengage the engine clutch, as they cannot truly run in pure electric: as we saw earlier, they cannot set the engine torque exactly to zero as they cannot match exactly the torque demand with the e-machine torque only.

Since the engine operating speed range is typically narrower than the e-machine's, especially since it is limited by its idle speed, the latter models hare therefore

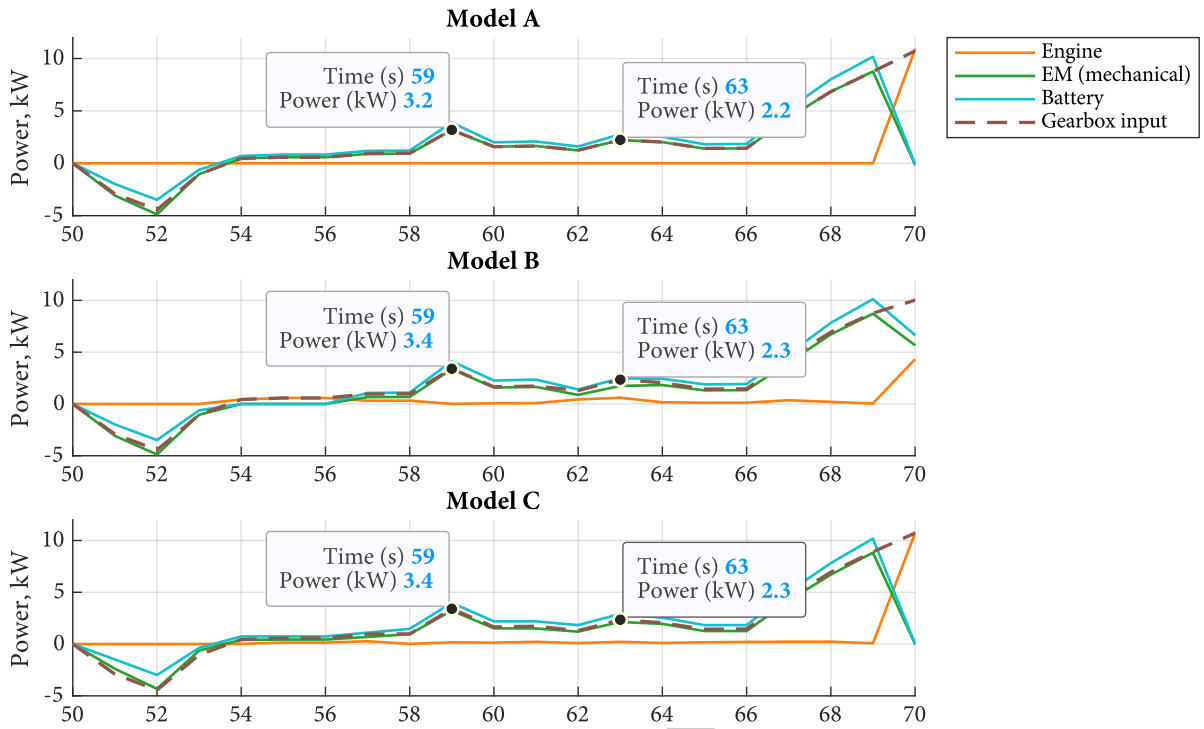


Figure 4.4: Example of the torque demand being slightly higher for the e-machine torque- and battery current- based models when the gear number is controlled by dynamic programming, due to the gearbox efficiency.

	Model	$m_{PF} = 11$			$m_{PF} = 21$			$m_{PF} = 41$		
		Δm_f	$\psi(\sigma_N)$	t_{sim} (s)	Δm_f	$\psi(\sigma_N)$	t_{sim} (s)	Δm_f	$\psi(\sigma_N)$	t_{sim} (s)
A)	T_{eng}	1.94 %	9.2e-04	2.5	1.26 %	1.3e-03	2.6	0.74 %	1.4e-03	2.9
B)	T_{em}	13.15 %	1.0e-02	2.8	2.15 %	4.9e-03	2.8	1.48 %	5.7e-03	3.0
C)	i_b	3.53 %	3.5e-03	2.8	1.99 %	3.2e-03	3.3	1.17 %	5.2e-03	3.3
D)	τ_{eng}	1.95 %	1.3e-03	2.6	1.25 %	1.2e-03	2.6	0.71 %	1.2e-03	2.9
E)	τ_{em}	2.28 %	4.3e-03	2.6	1.52 %	5.1e-03	2.6	0.95 %	4.7e-03	2.9
F)	ι_b	3.53 %	4.4e-03	3.0	2.01 %	3.8e-03	3.1	1.09 %	3.2e-03	3.3
G)	α_{eng}	1.07 %	4.3e-03	2.8	0.61 %	4.4e-03	2.6	0.32 %	3.2e-03	2.8
H)	α_{em}	0.91 %	1.1e-03	2.5	0.45 %	1.0e-03	2.7	0.33 %	1.0e-03	2.8

Table 4.6: Accuracy and simulation time of the examined models, with the gear number controlled by dynamic programming.

more restricted in selecting the gear number. This means that the engine torque-based and torque-split models may have additional ways to run more efficiently, by choosing the gear with the highest transmission efficiency.

The overall effect of this phenomenon on the models' performance can be visualized in Table 4.6.

4.6 Limitations and further work

When considering the results presented in this section, there are many factors that should be taken into account.

Firstly, for simplicity, we did not consider the engine and e-machine's inertia.

Secondly, we did not model any auxiliaries load. This means, for example, that the vehicle can keep the state of charge by simply switching off the engine, and that the e-machine and/or the battery does not have to bear any additional load when the engine is turned off.

Finally, we only considered one architecture with a hybridization ratio of 0.7[†] and one set of mass and road-load coefficients, corresponding to a small-size passenger car.

[†] Defined as: $\frac{P_{eng,max}}{P_{eng,max} + P_{em,max}}$, as in [5]

All these assumptions and data may somewhat alter the structure of the optimal solution, which may in turn stress differently the strengths and weaknesses of each model. For example, the optimal solution that we inspected in the simulation results makes very scarce use of the battery charging operating mode; hence the most important weakness of the torque-split models does not show.

There are numerous extensions to this work. The most obvious are reintroducing the engine inertia and/or accessory loads as well as to test a wider range of vehicles and hybridization ratios. Then, similar analysis may also be repeated for other powertrain configurations, such as power-split hybrids and series hybrids.

Part II

Differential Dynamic Programming

5 *Differential dynamic programming algorithms*

In this chapter, we will first introduce the fundamental ideas behind differential dynamic programming. Then, after a brief overview of its historical development, we illustrate the basic methodology by deriving an algorithm to iteratively improve a nominal control trajectory for unconstrained problems. We then introduce a variant of this base algorithm which is able to handle more complex, constrained optimal control problems.

The starting point of all differential dynamic programming algorithms is the Hamilton-Jacobi-Bellman equation:

$$-\frac{\partial V}{\partial t}(x;t) = \min_{u}[L(x,u;t) + \langle V_x(x;t), f(x,u;t)\rangle] \tag{5.1}$$

The HJB equation can be directly derived using the principle of optimality[†], and it can be seen as its infinitesimal version. Clearly, it constitutes a better starting point for dealing with continuous-time problems; however, it remains quite awkward to use because of the challenges and potentially infinite storage space requirement in representing $V_x(x;t)$.

[†] See Chapter 5 in [47].

The underlying idea of differential dynamic programming is to simplify the problem by using a second-order local expansion of the HJB equation and using it to iteratively improve the control trajectory, while ensuring that conditions are met so that the local expansion is reasonably accurate.

5.1 Historical remarks

The very first work on differential dynamic programming was published by Mayne [53]. Jacobson [31, 29] then developed a series of first-order and second order algorithms for unconstrained problems and problems control inequality constraints, both free and fixed endpoint. Jacobson then went on to develop refinements of his algorithms to improve convergence of the Lagrange multipliers associated with the terminal state constraints in fixed endpoint problems [22, 23] and to better deal with bang-bang control problems [28].

This first set of algorithms, which proved superior to contemporary iterative optimal control techniques, was published in a compact and comprehensive book by Jacobson and Mayne [30] which is one of the main sources for the algorithm presented in this chapter and its derivation.

It should be noted that, at this point, no method for accommodating state variable or state-dependent control variable constraints of the form (2.6) and (2.8) was available. Jacobson and Lele [27] later attempted to address these constraints by augmenting the state of the system with a slack variable in order to convert the constrained problem into an unconstrained one. This method however does not appear to be treatable with a differential dynamic programming approach[‡] and is only applicable to a narrower class of optimal control problems. Moreover, it has the considerable drawback that the transformed problem shows singular arcs where the state constraints are active, which make convergence very hard to achieve.

[‡] In [27], Jacobson and Lele use a conjugate gradient method.

Several extensions and variations to Jacobson's algorithms were proposed in the following decades in order to accommodate constraints on the state variables. Mårtensson proposed a technique to transform any state variable inequality constraint into a state dependent control variable inequality constraints and subse-

quently extended Jacobson's fixed endpoint algorithm to handle this type of constraints [52, 50]. This algorithm is further discussed in § 5.4 and forms the basis for the applications presented in this thesis.

Jarmark experimented with first-order DDP algorithms and introduced a technique, alternative to Jacobson's step-size adjustment procedure, to ensure the convergence of the algorithm [34, 36, 35, 37].

Ruxton introduced another variant using a *multiplier penalty function* (MPF) approach to directly deal with state variable inequality constraints [68, 70, 69]. Compared to Mårtensson's approach, the MPF method does not require transformation of the state constraints into mixed constraints and it modifies the base DDP algorithm to a lower extent. On the downside, the multiplier penalty function is characterized by a set of weights which must be determined by the algorithm.

5.2 DDP ALGORITHM FOR UNCONSTRAINED PROBLEMS

In this section, we will derive an algorithm for iteratively improving a nominal control trajectory until the optimal one is achieved. We will deal with continuous-time problems of the form in § 2.1, but with the assumption that constraints (2.6) to (2.9) are absent.

The content of this section summarizes the basic methodology used by Jacobson and Mayne to derive their first- and second-order algorithms, and it heavily draws from [30]. Although the derivation does not require particularly sophisticated mathematics and is quite straightforward, the notation can become quite cumbersome and it can complicate its understanding. Therefore, we divide the derivation in steps as outlined below.

Expansion of the HJB equation. We redefine the state and control trajectories in terms of deviations δx and δu from a nominal trajectory and we expand the HJB equation to second order in δx.

Introducing the Hamiltonian. We introduce the control Hamiltonian into the second-order expansion of the HJB equation.

Minimization of the RHS. We derive a linear relationship between δx and δu that minimizes the right-hand side of the second-order expansion of the HJB equation.

Derivation of the base equations. We use the second-order expansion of the HJB equation, with the right-hand side minimized, to derive differential equations in time for the value function and its first and second derivatives with respect to x.

5.2.1 *Expansion of the HJB equation*

First, let's introduce a known nominal control trajectory $\bar{u}$ which, applying the state equations (2.1) with initial conditions (2.2) over the time interval $[t_o, t_f]$, produces a nominal state trajectory $\bar{x}$ and a nominal cost $\bar{V}(x;t)$[†].

[†] As defined in (2.4).

Let us now introduce a variation in the nominal control trajectory δu and the corresponding variation in the nominal state trajectory δx; identifying a δu at each iteration that reduces the total cost will be our goal.

We can rewrite the state equations (2.1), total cost (2.4) and the HJB equation (5.1) as a function of the new trajectories $u = \bar{u} + \delta u$ and $x = \bar{x} + \delta x$:

$$\frac{d(\bar{x} + \delta x)}{dt} = f(\bar{x} + \delta x, \bar{u} + \delta u; t); \quad \bar{x}(t_o) + \delta x(t_o) = x_o, \tag{5.2}$$

$$J(x_o; t_o) = \int_{t_o}^{t_f} L(\bar{x} + \delta x, \bar{u} + \delta u; t)\, dt + F(\bar{x}(t_f) + \delta x(t_f); t_f), \tag{5.3}$$

$$-\frac{\partial V}{\partial t}(\bar{x} + \delta x; t) = \min_{\delta u} \big[L(\bar{x} + \delta x, \bar{u} + \delta u; t) + \\ + \langle V_x(\bar{x} + \delta x; t), f(\bar{x} + \delta x, \bar{u} + \delta u; t) \rangle \big]. \tag{5.4}$$

Now, we perform a power series expansion of the optimal cost in δx about $\bar{x}$:

$$V(\bar{x} + \delta x; t) = V(\bar{x}; t) + \langle V_x, \delta x \rangle + \frac{1}{2} \langle \delta x, V_{xx} \delta x \rangle + \text{h.o.t.}^{\dagger}; \tag{5.5}$$

V_x and V_{xx} are evaluated at $\bar{x}; t$. We also expand $V_x(\bar{x} + \delta x; t)$ to

$$V_x(\bar{x} + \delta x; t) = V_x(\bar{x}; t) + V_{xx} \delta x + V_{xxx} \delta x \delta x + \text{h.o.t.}^{\ddagger}, \tag{5.6}$$

where V_x, V_{xx} and V_{xxx} are evaluated at $\bar{x}; t$. For ease of notation, functions whose argument is omitted are evaluated at $(\bar{x} + \delta x; t)$.

Suppose that the the nominal control trajectory is applied from time t_o to t and that the state variables subsequently assume value $\bar{x} = \bar{x}(t)$. Let us define the variable a as the difference between the optimal cost and the nominal cost at $(\bar{x}; t)$, so that:

$$V(\bar{x}; t) = \bar{V}(\bar{x}; t) + a(\bar{x}; t). \tag{5.7}$$

Here, $V(\bar{x}; t)$, the optimal cost at $(\bar{x}; t)$, is the optimal cost incurred by applying the optimal controls $u^o(\tau) = \bar{u} + \delta u(\tau)$ from time t to t_f, i.e. $\tau \in [t, t_f]$; while $\bar{V}(\bar{x}; t)$, the nominal cost at $(\bar{x}; t)$, is the cost incurred by applying the nominal controls $\bar{u}(\tau)$ for $\tau \in [t, t_f]$.

Let us now substitute $V(\bar{x}; t)$ as in (5.7) in our power series expansion of the optimal cost in (5.5):

$$V(\bar{x} + \delta x; t) = \bar{V}(\bar{x}; t) + a(\bar{x}; t) + \langle V_x, \delta x \rangle + \\ + \frac{1}{2} \langle \delta x, V_{xx} \delta x \rangle + \text{h.o.t.} \tag{5.8}$$

If we substituted (5.8) and (5.5) into the HJB equation (5.1), we would still have an exact application of the principle of optimality. However, it is clearly impossible from a practical point of view to store an infinite, or even a high-order, power series expansion of $V(\bar{x} + \delta x; t)$, which is a vector-valued function. We therefore truncate the expansion to second-order and we assume that δx is sufficiently small that the error vanishes. Unless the problem is LQ[§], this is obviously an assumption that does not hold in general.

When we will construct an algorithm for iteratively improving the control trajectory, we will use a method to ensure that δx stays small enough even when the nominal trajectory is far from the optimal trajectory. For the time being, let us postpone this issues to § 5.3. Also, from now on, all terms of order higher than second such as $V_{xxx} \delta x \delta x$ will be neglected with little impact on the algorithms. It is easily shown [30] that this is legit and that it does not have a significant impact as the resulting error in the predicted change in cost $a(t)$ is third-order. Then, our second-order expansion of $V(\bar{x} + \delta x; t)$ and $V_x(\bar{x} + \delta x; t)$ simplifies to:

$$V(\bar{x} + \delta x; t) = \bar{V} + a + \langle V_x, \delta x \rangle + \frac{1}{2} \langle \delta x, V_{xx} \delta x \rangle \tag{5.9}$$

[†] Throughout this text, h.o.t. stands for higher order terms in δx.

[‡] Define

$$V_{xxx} \delta x \delta x \stackrel{\text{def}}{=} \sum_{i=1}^{n} \sum_{j=1}^{n} V_{x x_i x_j} \delta x_i \delta x_j$$

.

[§] In an LQ (linear-quadratic) problem, the cost function is quadratic in the state variables.

and

$$V_x(\bar{x} + \delta x; t) = V_x + V_{xx}\delta x + V_{xxx}\delta x\delta x. \tag{5.10}$$

Plugging the second-order expansions of $V(\bar{x} + \delta x; t)$ and $V_x(\bar{x} + \delta x; t)$ into the HJB equation (5.4) we get:

$$-\frac{\partial \bar{V}}{\partial t} - \frac{\partial a}{\partial t} - \frac{\partial V_x}{\partial t}^T \delta x - \frac{1}{2}\delta x^T \frac{\partial V_{xx}}{\partial t}\delta x = \min_{\delta u}\left[g(\bar{x} + \delta x, \bar{u} + \delta u; t) + (V_x + V_{xx}\delta x)^T f(\bar{x} + \delta x, \bar{u} + \delta u; t)\right] \tag{5.11}$$

If the nominal trajectory is sufficiently close to the optimal one (or if the problem is LQ), the minimizing δu in the right-hand side of the HJB equation (5.11) will be small and so will the resulting δx. If on the other hand the nominal control trajectory is far from optimal, the resulting δx will be too large for the second-order expansion of the optimal cost $V(\bar{x} + \delta x; t)$ to hold. In particular, since δx is produced by integrating the state equations in Equation 5.2 on application of the control $\bar{u} + \delta u$, it can grow larger as the length of the interval $[t, t_f]$ increases.

In our algorithm, we will use a *step-size adjustment procedure* developed by Jacobson and Mayne [30] to restrict the size of δx by applying the minimizing new control $\bar{u} + \delta u$ only to a small time interval. This procedure is also described in § 5.3.

Furthermore, note that in replacing the optimal cost in the HJB equation with its power series expansion, we assume that the cost is smooth enough with respect to the state variables. If the optimal cost presents strong discontinuities with respect to the state variables in the proximity of the optimal trajectory, the algorithm might struggle to improve the trajectory at some point and it might require a large number of iterations to achieve convergence.

5.2.2 *Introducing the Hamiltonian*

In the context of differential dynamic programming, we define the Hamiltonian as

$$H(x, u, V_x; t) = L(x, u; t) + \langle V_x, f(x, u; t)\rangle . \tag{5.12}$$

Note that this formulation of the Hamiltonian is almost identical to one we saw in the context of the minimum principle § 2.2, the only difference being that V_x replaces the co-states.

Introducing (5.12) into the second-order expansion of the HJB equation (5.11), we get

$$-\frac{\partial \bar{V}}{\partial t} - \frac{\partial a}{\partial t} - \left\langle \frac{\partial V_x}{\partial t}, \delta x \right\rangle - \frac{1}{2}\left\langle \delta x, \frac{\partial V_{xx}}{\partial t}\delta x \right\rangle = \min_{\delta u}\left[H(\bar{x} + \delta x, \bar{u} + \delta u, V_x + V_{xx}\delta x; t)\right] \tag{5.13}$$

We now come to another key aspect of the method. So far, we have expressed the new control u in terms of a deviation from the nominal control trajectory. We now replace this definition of δu to express the new control u in terms of a deviation with respect to the control $\hat{u}$

$$u = \hat{u} + \delta u, \tag{5.14}$$

where $\hat{u}$ is defined as the control which minimizes

$$\min_u \left[H(\bar{x}, u, V_x; t) \right]. \tag{5.15}$$

Note that (5.15) is equivalent to considering (5.13) for state $x = \bar{x}$.

In other words, $\hat{u}$ would define the optimal control trajectory if $\bar{x}$ was the optimal state trajectory. We then reintroduce a correction δu to account for the fact that the optimal state trajectory differs from the nominal trajectory by an amount δx. What we want to do next is to find a relationship between this δu and this δx.

Substituting $u = \hat{u} + \delta u$ into (5.13), the HJB becomes:

$$-\frac{\partial \bar{V}}{\partial t} - \frac{\partial a}{\partial t} - \left\langle \frac{\partial V_x}{\partial t}, \delta x \right\rangle - \frac{1}{2} \left\langle \delta x, \frac{\partial V_{xx}}{\partial t} \delta x \right\rangle = \min_{\delta u} \left[H(\bar{x} + \delta x, \hat{u} + \delta u, V_x + V_{xx}\delta x; t) \right]. \tag{5.16}$$

5.2.3 *Minimization of the RHS*

Let us expand the right-hand side of (5.13) to second order about $\bar{x}$, $\hat{u}$:

$$\min_{\delta u} \Big[H + \langle H_u, \delta u \rangle + \langle H_x + V_{xx} f, \delta x \rangle + \left\langle \delta u, (H_{ux} + f_u^T V_{xx}) \delta x \right\rangle + \\ + \frac{1}{2} \langle \delta u, H_{uu} \delta u \rangle + \frac{1}{2} \left\langle \delta x, (H_{xx} + f_x^T V_{xx}) \delta x \right\rangle \Big] \tag{5.17}$$

Now in order to determine δu, we minimize (5.17) by differentiating its argument with respect to δu and equating it to zero

$$H_u + H_{uu}\delta u + (H_{ux} + f_u^T V_{xx})\delta x = 0, \tag{5.18}$$

and, since H_u is obviously zero[†]:

$$H_{uu}\delta u + (H_{ux} + f_u^T V_{xx})\delta x = 0, \tag{5.19}$$

[†] Because we defined $\hat{u}$ to minimize $H(\bar{x}, \hat{u}, V_x, t)$.

Hence, we finally come to establish the following relationship between δu and δx:

$$\delta u = -H_{uu}^{-1}(H_{ux} + f_u^T V_{xx})\delta x, \tag{5.20}$$

which we can also rewrite as

$$\delta u = \beta_1 \delta x, \tag{5.21}$$

with

$$\beta_1 = -H_{uu}^{-1}(H_{ux} + f_u^T V_{xx}). \tag{5.22}$$

For clarity, we remark that what we just derived here is the relationship which enforces the necessary condition of optimality

$$H_u(\bar{x} + \delta x, \hat{u} + \delta u, V_x + V_{xx}\delta x; t) = 0, \tag{5.23}$$

which we wrote based on our second-order expansion of the HJB equation and by neglecting terms of order higher than second; hence, (5.21) with β_1 given by (5.22) is correct for δx sufficiently small.

Equation (5.21) also highlights that the relationship we found between δu and δx is linear. Although this might look arbitrary, a clever point made by Jacobson

and Mayne [30] is that there would be no point in seeking a relationship of higher order. Consider the terms in (5.17) which affect its minimization, that is only the terms involving δu (except for $\langle H_u, \delta u \rangle$ which is zero):

$$\min_{\delta u} \left[\left\langle \delta u, (H_{ux} + f_u^T V_{xx}) \delta x \right\rangle + \frac{1}{2} \left\langle \delta u, H_{uu} \delta u \right\rangle \right]. \tag{5.24}$$

A linear relationship between δu and δx will produce terms that are quadratic in δx, and any relationship of order higher than linear will produce additional terms that are of order higher than quadratic in δx, which are neglected as we expand (5.17) to second order only.

5.2.4 *Derivation of the base equations*

In the previous subsection, we obtained a relationship between δu and δx that allows us to get rid of the minimization in the RHS of our second-order expansion of the HJB equation. We can substitute (5.21) into (5.17) to obtain:

$$\begin{aligned} -\frac{\partial \bar{V}}{\partial t} - \frac{\partial a}{\partial t} - \left\langle \frac{\partial V_x}{\partial t}, \delta x \right\rangle - \frac{1}{2} \left\langle \delta x, \frac{\partial V_{xx}}{\partial t} \delta x \right\rangle = \\ H + \left\langle H_x + V_{xx} f + \beta_1^T H_u, \delta x \right\rangle + \\ + \frac{1}{2} \left\langle \delta x, \left(H_{xx} + f_x^T V_{xx} + V_{xx} f_x - \beta_1^T H_{uu} \beta_1 \right) \delta x \right\rangle \end{aligned} \tag{5.25}$$

Since this equality must hold for all δx (sufficiently small), we can equate the coefficients of the same order in δx, which gives the following set of relationships:

$$-\frac{\partial \bar{V}}{\partial t} - \frac{\partial a}{\partial t} = H \tag{5.26}$$

$$-\frac{\partial V_x}{\partial t} = H_x + \beta_1^T H_u + V_{xx} f \tag{5.27}$$

$$\begin{aligned} -\frac{\partial V_{xx}}{\partial t} = H_{xx} + f_x^T V_{xx} + V_{xx} f_x + \\ - \left(H_{ux} + f_u^T V_{xx} \right)^T H_{uu} \left(H_{ux} + f_u^T V_{xx} \right). \end{aligned} \tag{5.28}$$

At this point, we recall that $V = \bar{V} + a$, V_x and V_{xx} are all functions of $\bar{x}(t)$ and t. We derive their total derivatives with respect to time by applying the chain

rule:

$$\frac{d}{dt}(\bar{V} + a) =$$

$$= \frac{\partial(\bar{V} + a)}{\partial t} + \frac{\partial(\bar{V} + a)}{\partial x}\frac{dx}{dt} =$$

$$= \frac{\partial(\bar{V} + a)}{\partial t} + V_x^T f(\bar{x}, \bar{u}; t)$$

$$\frac{dV_x}{dt} =$$

$$= \frac{\partial V_x}{\partial t} + \frac{\partial V_x}{\partial x}\frac{dx}{dt} = \tag{5.29}$$

$$= \frac{\partial V_x}{\partial t} + V_{xx} f(\bar{x}, \bar{u}; t)$$

$$\frac{dV_{xx}}{dt} =$$

$$= \frac{\partial V_{xx}}{\partial t} + \frac{1}{2}V_{xxx} f(\bar{x}, \bar{u}; t) + \frac{1}{2} f(\bar{x}, \bar{u}; t)^T V_{xxx}.$$

Substituting Equations (5.29) into (5.26) to (5.28) and once again neglecting V_{xxx} and observing that $H_u = 0$, we finally get to the following set of differential equations, which we will call the base equations:

$$-\dot{a} = H - H(\bar{x}, \bar{u}, V_x, t) \tag{5.30}$$

$$-\dot{V}_x = H_x + V_{xx}(f - f(\bar{x}, \bar{u}, t)) \tag{5.31}$$

$$\begin{aligned} -\dot{V}_{xx} = {} & H_{xx} + f_x^T V_{xx} + V_{xx} f_x + \\ & - \left(H_{ux} + f_u^T V_{xx}\right)^T H_{uu} \left(H_{ux} + f_u^T V_{xx}\right). \end{aligned} \tag{5.32}$$

As previously, all quantities here are evaluated at $\bar{x}$, $\hat{u}$ unless otherwise stated.

Assuming that we know a nominal trajectory, we can write a set of boundary conditions for the base equations (5.30) to (5.32). At time $t = t_f$, both the nominal cost and the optimal cost are equal to $V(\bar{x}, t_f) = F(\bar{x}, t_f)$. Hence:

$$a(t_f) = 0 \tag{5.33}$$

$$V_x(t_f) = F_x(\bar{x}, t_f) \tag{5.34}$$

$$V_{xx}(t_f) = F_{xx}(\bar{x}, t_f). \tag{5.35}$$

These base equations constitute one of the main building bricks for the differential dynamic programming algorithm. Assuming a nominal trajectory is available, (5.30) to (5.32) can be integrated backwards in time from t_f to t_0 using the boundary conditions (5.33) to (5.35); note that this also requires the minimization of $H(\bar{x}, u, V_x; t)$ to obtain $\hat{u}$. Once the integration is complete, the state equations can be integrated forward in time starting from an initial state x_0 and by applying the new control $u = \hat{u} + \beta_1 \delta x$.

Furthermore, the integration of the base equation provides a criterion for convergence. By definition, $|a(\bar{x}; t)|$ is the improvement in cost resulting from replacing the nominal control with the optimal control, from time t to t_f[†]. Hence, if the optimal control is exactly equal to the nominal control, $|a(x_0); t_0)|$ will be equal to zero. In practice, in our algorithm, we will assume convergence when $|a(x_0; t_0)|$ becomes smaller than a small quantity η_1.

[†] while still using the nominal control from t_0 to t.

5.2.5 *Extensions of the unconstrained DDP algorithm*

The equations we derived in § 5.2 are useful to treat finite-time, unconstrained optimal control problems. A similar procedure can be used to derive equations for dealing with fixed-endpoint problems with control inequality constraints, which will essentially lead to an extended set of base equations. Problems with unknown final time can also be treated as fixed-endpoint problems by adjoining the final time to the state vector; the base equations remain unaltered, but their boundary conditions become a bit more cumbersome.

Finally, we should mention for the sake of completeness that first-order algorithms can be easily derived as special instances of these second-order algorithms. Although these first-order variants offer lower computational complexity and they also have solid convergence proofs [56], they are generally unable to converge in a reasonable number of iterations apart from specific applications [34, 35]. All these extensions to the base DDP methodology are straightforward[†] and are treated extensively in works by Jacobson [31, 29, 30].

[†]In [27], Jacobson and Lele use a conjugate gradient method.

There is one important piece of the puzzle that was not developed in these works, and that is the ability to incorporate state variable inequality constraints. As we mentioned in § 5.1, a few alternatives were developed to handle them. For this thesis's work, Mårtensson's [52, 50] variant was used as described in § 5.4.

5.3 THE STEP-SIZE ADJUSTMENT METHOD

In § 5.2.3, we derived a relationship to improve the nominal trajectories by applying the new control $u = \hat{u} + \beta_1 \delta x$. However, the derivation relied on the size of the δx produced by the improved controls being small enough, so that our second-order expansion of the HJB equation is accurate.

In general, this does not necessarily apply, especially for long control horizons. In fact, the size of δx is ultimately determined by the length of the time span over which the new control is applied through the state equations:

$$\frac{d(\bar{x} + \delta x)}{dt} = f(\bar{x} + \delta x, \hat{u} + \beta_1 \delta x; t); \quad \bar{x}(t_o) + \delta x(t_o) = x_o. \tag{5.36}$$

This also suggests that the size of δx can be arbitrarily restricted by restricting the time interval over which (5.36) is applied. The step-size adjustment method (proposed by Jacobson [31]) is a method to restrict the size of this time interval until the resulting δx is small enough that a reduction in cost is effectively achieved.

The underlying idea is to generate the improved trajectory by applying the nominal controls from find time t_o to a time t_1 and then use the new control $u = \hat{u} + \beta_1 \delta x$ from t_1 to t_f, and to set t_1 by checking whether a good improvement in cost has been achieved. In order to decide whether a good improvement has been achieved, one compares the actual improvement in cost

$$\Delta V = \bar{V}(\bar{x}; t_1) - V(\bar{x}; t_1) \tag{5.37}$$

with the predicted improvement $|a(\bar{x}; t_1)|$[‡]

[‡]Recall that, by definition, $|a(\bar{x}; t_1)|$ is the improvement in cost resulting from replacing the nominal control with the optimal control from time t_1 to t_f (while still using the nominal control from t_o to t).

Note that, in (5.37), the nominal cost $\bar{V}(\bar{x}; t_1)$ is the cost obtained by running the nominal controls while $V(\bar{x}; t_1)$ is the cost obtained by running the new controls from t_1 to t_f, starting from $\bar{x}(t_1)$.

The goodness of an improvement in cost can be measured by considering if the ratio of the actual and the predicted improvement in cost, is larger than a certain positive quantity C:

$$\frac{\Delta V}{|a(\bar{x}; t_1)|} > C. \tag{5.38}$$

This quantity must clearly be larger than zero, as a negative ΔV would correspond to a degradation of the cost, and it cannot be larger than one. Unfortunately, a good choice of C depends on the structure of the problem. Improvements corresponding to large values may be unattainable if the problem is highly nonlinear and the trajectory is far from optimal. On the other hand, small values may unnecessarily produce a bigger number of iterations for convergence.

In the step-size adjustment method, (5.38) is first tested for $t_1 = t_0$. If the inequality is not satisfied, a new value of t_1 is chosen and the new controls are applied in the interval $[t_1, t_f]$, and (5.38) is checked again.

In the applications presented in this work, we set C to 0.5, as suggested by Jacobson and Mayne [30]. An improvement on this would be to develop a method to adjust the value of C as the algorithm progresses. For example, it may prove to be practical to use a small value at the first iterations, in order to favor global changes to the whole trajectory, and increase its value as the algorithm progresses, to speed up convergence near the optimum by rejecting minor improvements.

Then, one also needs a rule to change t_1 when (5.38) is not satisfied. A simple rule is to try the midpoint of the interval $[t_0, t_f]$, and then the midpoint of $[t_1, t_f]$ if necessary. However, it is possible that a nominal trajectory generated by DDP is also optimal in an interval $[t_{eff}, t_f]$. An improvement to this is therefore to restrict the choice of t_1 so that it never falls $[t_{eff}, t_f]$. Note that this time t_{eff} can be easily identified as the time for which $|a(\bar{x}; t_1)|$ becomes greater than η_1.

The general procedure of the step-size adjustment method can summarized as follows: set $t_1 = 2t_0 - t_f$[†]. Update t_1 to:

$$t_1 = \frac{t_f - t_1}{2} + t_1, \tag{5.39}$$

integrate the state dynamics (5.36) starting from $(\bar{x}(t_1); t_1)$ and evaluate the corresponding change in cost ΔV. If inequality (5.38) is satisfied, replace the nominal trajectories with the new ones and proceed to the next iteration. Else, update t_1 with (5.39) and repeat.

[†] so that the first computed value of t_1 via (5.39) will be t_0.

5.4 DDP algorithm for constrained problems

In § 5.2, we outlined the derivation of the basic DDP algorithm for unconstrained problems. In this section, we briefly describe the main features of an algorithm for fixed endpoint optimal control problems with pure control inequality constraints and mixed state-control inequality constraints of the form $g(x, u; t) \leqslant 0$, developed by Mårtensson [52]. Pure state inequality constraints can then be addressed by transforming them using the constraining hyperplane technique described in § 5.5.

A detailed derivation of the algorithm can be found in [52]. The software implementation that was developed for this thesis, along with its peculiarities (some of which constitute original contributions of this work) is discussed in § 6.

5.4.1 *Terminal state constraints*

To enforce the terminal state constraint $\psi(x(t_f); t_f) = 0$, we adjoin it with a set of Lagrange multipliers b to the cost functional. We therefore define the augmented cost functional

$$J(x_0, b; t_0) = \int_{t_0}^{t_f} L(x, u; t)\, dt + F(x(t_f); t_f) + b^T \psi(x(t_f); t_f). \tag{5.40}$$

Then, there exists a unique set of Lagrange multipliers b for which the solution of the augmented problem corresponds to the solution of the original fixed-endpoint problem.

In fact, there may be problems for which the augmented problem has an extremal but not a minimum. One solution to this problem is to also add a quadratic term $c\psi(x(t_f);t_f)^T\psi(x(t_f);t_f)$ to the terminal cost F [52, Part IV, Sec. 4.4]. This converts the extremal into a minimum and does not alter the optimal solution, as $c\psi(x(t_f);t_f)^T\psi(x(t_f);t_f)$ is obviously zero if the terminal constraints are satisfied. This technique proved crucial for the application presented in § 7.

Clearly, this approach requires that the optimal value of the Lagrange multipliers b is found. The differential dynamic programming algorithm discussed in this section includes a procedure for iteratively improving the fixed endpoint error by updating b, given an initial guess. This modification of the algorithm extends the base equations with three new terms $\dot{V}_b$, $\dot{V}_{xb}$ and $\dot{V}_{bb}$. Also, a new feedback relationship is established between the change in b (with respect to the nominal value) and the new control to be adopted:

$$u(t) = \hat{u}(t) + \beta_1(t)\delta x(t) + \beta_2(t)\delta b. \tag{5.41}$$

The algorithm works in a layered fashion. In an inner layer, the problem is treated as a free endpoint problem and the augmented cost functional (5.40) is minimized for a nominal set of multipliers $\bar{b}$. Within this inner layer, the multipliers are kept fixed and (5.41) therefore simplifies to

$$u(t) = \hat{u}(t) + \beta_1(t)\delta x(t). \tag{5.42}$$

In an outer layer, a new set of multipliers b is generated and the new nominal control trajectory is generated with (5.41), which in turn generates a new nominal state trajectory. The process is repeated until convergence is achieved, i.e. $|a(x_o;t_o)| < \eta_1$[†] and the endpoint error is less than a small quantity η_2, i.e. $\psi(x(t_f);t_f) < \eta_2$.

[†] As for the unconstrained case; see § 5.2.4.

5.4.2 *State and control constraints and the multiplier function approach*

In this section, we introduce changes to the algorithm caused by the introduction of state and control constraints. The first relates to the way we define (and evaluate) $\hat{u}$. The second relates to the way we evaluate β_1 and how this introduces additional terms in the base equations.

Similarly to the unconstrained algorithm, we need to determine the controls $\hat{u}$ that minimize the Hamiltonian evaluated at the nominal state trajectory $\bar{x}$ and with the nominal multipliers $\bar{b}$. This is needed for the integration of the base equations and to determine the new control trajectory.

This time, however, the minimization must be done over u belonging to the set of admissible control variables U, which in our framework is defined by a set of inequality constraints $g(x,u;t) \leqslant 0$:

$$H(\bar{x},\hat{u},Vx;t) = \min_{\substack{u \\ g(x,u;t)\leqslant 0}} [H(\bar{x},u,Vx;t)]. \tag{5.43}$$

For some problems with few control variables and few constraints with a simple structure, the minimization of (5.43) may be easy to perform. In general however, this minimization problem is a complex nonlinear program that requires a dedicated solver. We will deal with this in § 6.4; for now, let us assume we can always obtain $\hat{u}$.

Now let us turn our attention to how β_1 and β_2, used to generate new trajectories, are obtained in the constrained case. To do this, we must first distinguish whether the constraints are active at $(\bar{x},\hat{u};t)$. When minimizing (5.43), we must also note which (if any) of the constraints are active. These constraints will be denoted by $\hat{g}$ and their number by $\hat{p}$.

Assume that a number $\hat{p} > 0$ of the constraints are indeed active. Similarly to the unconstrained case, the algorithm relies on establishing a relationship between δx and δu that enforces the minimization of the RHS of our second-order expansion of the HJB equation (5.17). In § 5.2.3, we did this by enforcing the necessary condition that its derivative with respect to u be equal to zero

$$H_u(\bar{x} + \delta x, \hat{u} + \delta u, V_x + V_{xx}\delta x; t) = 0. \tag{5.44}$$

In the constrained case, we use the *multiplier function* approach introduced by Mårtensson [51]. Similarly to the method of Lagrange multipliers, we reformulate our objective function (the Hamiltonian) by adjoining the constraints via a set of multiplier functions μ:

$$\mathcal{H}(\bar{x}, u, V_x; t) = H(\bar{x}, u, V_x; t) + \langle \mu(\bar{x}, u, \bar{b}; t), \hat{g}(\bar{x}, u; t)\rangle. \tag{5.45}$$

Provided that μ is defined properly, necessary conditions for having a constrained minimum at u are:

$$H_u(\bar{x}, u, \mu, V_x; t) + \mu(\bar{x}, u, \bar{b}; t)^T \hat{g}_u(\bar{x}, u, t) = 0, \tag{5.46}$$

$$\hat{g}(\bar{x}, u, t) = 0. \tag{5.47}$$

These conditions, expanded to first order, provide us with the relationship we were looking for. A rather lengthy derivation leads to obtain:

$$\beta_1 = -(H_{uu} + \mu\hat{g}_{uu})^{-1} Z (H_{ux} + f_u^T V_{xx} + \mu\hat{g}_{ux}) - Q^T \hat{g}_x, \tag{5.48}$$

$$\beta_2 = -(H_{uu} + \mu\hat{g}_{uu})^{-1} Z f_u^T V_{xb}, \tag{5.49}$$

where Q and Z are defined as

$$Q = (\hat{g}_u (H_{uu} + \mu\hat{g}_{uu})^{-1} \hat{g}_u^T)^{-1} \hat{g}_u (H_{uu} + \mu\hat{g}_{uu})^{-1}, \tag{5.50}$$

$$Z = I_m - \hat{g}_u^T Q,^{\dagger} \tag{5.51}$$

and the multiplier function μ is defined as[‡]

$$\mu = -(\hat{g}_u \hat{g}_u^T)^{-1} \hat{g}_u H_u. \tag{5.52}$$

[†] I_m stands for the identity matrix of dimension m, i.e. the number of controls.

[‡] Note that other definitions are possible.

5.4.3 *The base equations*

In the constrained algorithm, we have three additional subsets of equations for $\dot{V}_b$, $\dot{V}_{xb}$, and we have additional terms in the equations that take the constraints into account. These are combinations of the derivatives of the active constraints $\hat{g}$ and the multiplier function μ.

The full set of base equations becomes:

$$-\dot{\alpha} = H - H(\bar{x}, \bar{u}, V_x, t) \tag{5.53}$$

$$-\dot{V}_x = H_x + V_{xx}(f - f(\bar{x}, \bar{u}, t)) + \hat{g}_x^T \mu \tag{5.54}$$

$$-\dot{V}_b = V_{xb}^T (f - f(\bar{x}, \bar{u}, t)) \tag{5.55}$$

$$-\dot{V}_{xb} = (f_x^T + \beta_1^T f_u^T) V_{xb} \tag{5.56}$$

$$-\dot{V}_{bb} = -\beta_2^T (H_{uu} + \mu\hat{g}_{uu})\beta_2,^{\S} \tag{5.57}$$

$$\begin{aligned} -\dot{V}_{xx} = {} & H_{xx} + f_x^T V_{xx} + V_{xx} f_x + \beta_1^T (H_{uu} + \mu\hat{g}_{uu})\beta_1 + \\ & + \beta_1^T (H_{ux} + \mu\hat{g}_{ux} + f_u^T V_{xx})\beta_1 + \mu\hat{g}_{xx}. \end{aligned} \tag{5.58}$$

with boundary conditions:

$$a(t_f) = 0 \tag{5.59}$$

$$V_x(t_f) = F_x(\bar{x}(t_f); t_f) + \psi_x^T(\bar{x}(t_f))\bar{b} \tag{5.60}$$

$$V_b(t_f) = \psi(\bar{x}(t_f)) \tag{5.61}$$

$$V_{xb}(t_f) = \psi_x^T(\bar{x}(t_f)) \tag{5.62}$$

$$V_{bb}(t_f) = 0 \tag{5.63}$$

$$V_{xx}(t_f) = F_{xx}(\bar{x}(t_f); t_f) + \bar{b}\psi_{xx}(\bar{x}(t_f)). \tag{5.64}$$

5.5 CONSTRAINING HYPERPLANE TECHNIQUE

The constrained algorithm that we will implement does not handle state variable constraints directly. It can, however, handle mixed state-control variable inequality constraints of the form of the form $g(x, u; t) \leqslant 0$. In order to deal with pure state variable inequality constraints of the form $g(x; t) \leqslant 0$, we will transform them into mixed inequality constraints using the constraining hyperplane technique developed by Mårtensson [52].

Let $S(x; t) \leqslant 0$ be the feasible region defined by one of our state constraints. The fundamental idea of the technique is to approximate this region with another region that explicitly depends on the control variables. Figure 5.1 illustrates this concept for a first-order constraint.

We define the order of a state constraint S as the lowest order of the time derivatives of S for which this time derivative is explicitly a function of at least one of the control variables. In other words, a constraint S if of order q if $\frac{d^q S}{dt^q}$ is the lowest order time derivative of S which is explicitly a function of at least one of the control variables [50].

For a first-order constraint, the region for which $S(x; t) \leqslant 0$ is replaced by a region in the $\left(\frac{dS}{dt}, S(x; t)\right)$ plane

$$\Omega = \left\{ \left(S(x; t), \frac{dS}{dt} \right) \mid \Pi \leqslant 0 \right\} \tag{5.65}$$

generated by the line[†] Π:

$$\Pi = \left\{ \left(S(x; t), \frac{dS}{dt} \right) \mid \frac{dS}{dt} + a_1 S(x; t) = 0, a_1 > 0 \right\}. \tag{5.66}$$

It is intuitively understood that the Ω approximates well the region $S(x; t) \leqslant 0$ if a_1 is large enough. In practice though, the technique works well even for small values of a_1, which is preferable for a numerical solution.

The constraining hyperplane technique generalizes well to higher-order state constraints. A qth-order state constraint can be transformed into a mixed state-control inequality constraint by defining the constraining hyperplane

$$\Pi = \left\{ \left(S(x; t), \frac{dS}{dt}, \ldots, \frac{d^q S}{dt^q} \right) \mid \frac{d^q S}{dt^q} + a_1 \frac{d^{q-1} S}{dt^{q-1}} + \ldots + a_q S = 0 \right\}, \tag{5.67}$$

where the parameters $a_1, \ldots, a_q$ are all real and must satisfy the condition that the roots p_i^* of the polynomial

$$p^q + a_1 p^{q-1} + \ldots + a_q \tag{5.68}$$

[†] which is the constraining hyperplane for first-order constraints.

[§] We define

$$\mu\hat{g}_{uu} \stackrel{\text{def}}{=} \sum_{k=1}^{\hat{p}}$$

and similarly $\mu\hat{g}_{ux}$ an

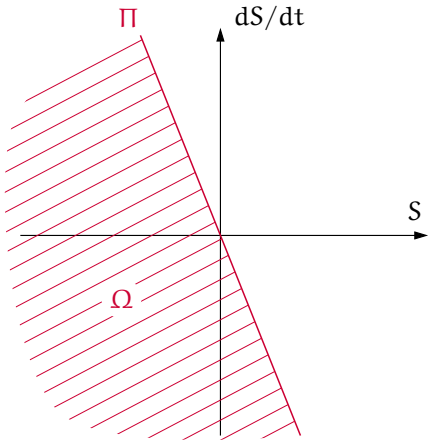


Figure 5.1: Constraining hyperplane for a first-order state constraint. In this case, the hyperplane is a line.

are all real and satisfy

$$p_1^* < p_2^* < \ldots < p_q^*. \tag{5.69}$$

Furthermore, the constraints must be satisfied at (x_o, t_o). This last requirement is clearly a property of any correctly formulated optimal control problem. Readers that are interested in the technical motivation for these requirements should refer to [52, Part II, Ch. 3] and [50].

5.6 The computational trick

So far, we have computed the new controls within one major iteration with (5.42) as:

$$u = \hat{u} + \beta_1 \delta x,$$

where $\beta_1(t)$ is obtained during the integration of the base equations (5.59), (5.60) and (5.64)

According to [30] the term $\beta_1 \delta x$ may in practice become too large for the local expansions in δu to hold, even if δx itself is still small enough for the expansion

$$V_x(\bar{x} + \delta x; t) = V_x + V_{xx}\delta x \tag{5.70}$$

to be accurate. In that case, we could still improve the new control trajectory by storing $V_x(t)$ and $V_{xx}(t)$ and directly minimizing:

$$u = \arg\min_u H(\bar{x} + \delta x, u, V_x + V_{xx}\delta x; t). \tag{5.71}$$

This may improve the convergence of the algorithm, reducing the number of iterations. On the downside, this minimization presents the same challenges that were discussed in § 6.4. Similarly, we replace the evaluation of the new controls between major iterations (5.41):

$$u = \hat{u} + \beta_1 \delta x + \beta_2 \delta b$$

with:

$$u = \arg\min_u H(\bar{x} + \delta x, u, V_x + V_{xx}\delta x + V_{xb}\delta b; t). \tag{5.72}$$

6 Implementation of the DDP algorithm

In this chapter, we describe the differential dynamic programming algorithm that was implemented in this work for solving fixed-endpoint OCPs with state and control variable inequality constraints.

More precisely, the algorithm handles problems with pure control variable or mixed state-control variable inequality constraints of the form

$$g(x, u; t) \leqslant 0. \tag{6.1}$$

Pure state variable inequality constraints of the form $g(x; t) \leqslant 0$ can be transformed into mixed inequality constraints using the constraining hyperplane technique developed by Mårtensson [52].

The theoretical foundation for the algorithm, which were introduced in the previous chapter, as well as the general computational procedure, which will be described in § 6.1, are largely based on the work by Mårtensson [52, 50] and Jacobson and Mayne [31, 30].

The software implementation and some aspects of the algorithm instead constitute original contributions of this thesis' author. In particular, original aspects of this work are the inclusion of a non-linear program solver for the minimization of the Hamiltonian in (5.43) to evaluate $\hat{u}$ (§ 6.4) and the implementation of automatic differentiation (§ 6.3) to evaluate the various derivative terms required for the integration of the base equations.

These features were developed to handle the complexity of the typical optimal control problems that arise in energy management strategy design. Moreover, they constitute the fundamental building bricks that will lead to the culmination of the second part of this thesis, that is the application that will be presented in the next chapter (§ 7).

The contents of this chapter are meant to serve a double purpose. The first is to illustrate how to use the equations and theoretical framework that were introduced in the preceding Chapter to practically build an iterative procedure for the solution of optimal control problems, while also dealing with practical issues such as numerical integration of the base equations, minimization of the Hamiltonian and computing derivatives.

The second purpose is to serve as a guide for the open-source codes that are provided in this thesis' repository. For this reason, this chapter also includes code fragments with break-down of their inner working. Note that these code fragments were largely edited for readability; function names, however, were left unchanged, so that the interested reader can easily refer to the actual working code which can be retrieved from its GitHub repository at https://github.com/fmiretti/ddp-shev.

6.1 Description of the algorithm

In this section, we introduce the computational algorithm in a descriptive manner. Since the algorithm is quite complex, we tackle it at two different levels of abstraction. First, we will look at it from a very broad perspective, which serves to illustrate the general procedure. Then, we will tackle it at a deeper layer of detail. This second description is the foundation for developing our software implementation.

6.1.1 General structure of the algorithm

The general structure of the algorithm is illustrated in Figure 6.1. The first thing to note is that the algorithm works with a layered structure. In an inner layer, we first solve the problem by treating it as a free-endpoint problem. However, rather than minimizing the cost functional

$$J(x_o, u; t_o) = \int_{t_o}^{t_f} L(x, u; t)\, dt + F(x(t_f); t_f), \tag{6.2}$$

we minimize the augmented cost functional:

$$J(x_o, u, b; t_o) = \int_{t_o}^{t_f} L(x, u; t)\, dt + F(x(t_f); t_f) + \bar{b}^T \psi(x(t_f); t_f), \tag{6.3}$$

for a given value of the Lagrange multipliers b, iteratively improving the control trajectory until no further reduction in cost is achievable. Initially, the values for b will have to be guessed.

This procedure is then nested in an outer layer, which reduces the endpoint error while keeping the cost increase to a minimum generating a new set of Lagrange multipliers $\bar{b}$. This is repeated until we meet some convergence criteria on both the cost and endpoint error.

In the remainder of this thesis, we will refer to iterations of the outer layer, which correspond to updates of the Lagrange multipliers, as *major* iterations and to iterations of the inner layer as *minor* iterations.

Before the algorithm can start, an initial guess for the Lagrange multipliers $\bar{b}$ and a nominal control trajectory $\bar{u}$ must be supplied. The nominal control trajectory in turn determines a nominal state trajectory $\bar{x}$.

Each minor iteration involves integrating $\dot{a}$, $\dot{V}_x$, $\dot{V}_{xx}$ backwards in time and evaluating (and storing) $\beta_1(t)$. Convergence of the inner layer is reached if the predicted reduction in cost is smaller than a small quantity, i.e. $|a(\bar{x}_o, \bar{b}; t_o)| < \eta_1$. If it has not, the step-size adjustment method[†] is used to generate a new nominal trajectory with an improved cost, and proceeds to the next minor iteration.

[†] described in § 5.3.

Then, the endpoint error is checked. If it is smaller than a tolerance $et a_2$, the algorithm stops. Else, it moves to the outer layer to update the Lagrange multipliers and then moves back to the inner layer. Updating $\bar{b}$ requires integrating $\dot{V}_b$, $\dot{V}_{xb}$, $\dot{V}_{bb}$ backwards in time and evaluating (and storing) $\beta_2(t)$. At this point, it may also be necessary to use the δb adjustment method described in STEP 7, § 6.1.2.

Summarizing, each minor iteration corresponds to a run of the step-size adjustment method, which updates the nominal state and control trajectories by applying the new controls

$$\hat{u} = \hat{u} + \beta_1 \delta x^{\ddagger} \tag{6.4}$$

over the whole time interval $[t_o, t_f]$ or a part of it.

[‡] Here, $\beta_2 \delta b$ is not present because b is kept fixed within minor iterations.

Each major iteration corresponds to the integration of $\dot{V}_b$, $\dot{V}_{xb}$, $\dot{V}_{bb}$ and a run of the (endpoint) multipliers adjustment method to update the nominal endpoint multipliers $\bar{b}$ as well as the nominal state and control trajectories by applying the new controls

$$\hat{u} = \hat{u} + \beta_1 \delta x + \beta_2 \delta b. \tag{6.5}$$

After each minor or major iteration, $\dot{a}$, $\dot{V}_x$, $\dot{V}_{xx}$ are integrated to then check convergence and to generate $\beta_1(t)$ and $\beta_2(t)$ which are needed if convergence has not been achieved. This is also done before the very first iteration to start the algorithm.

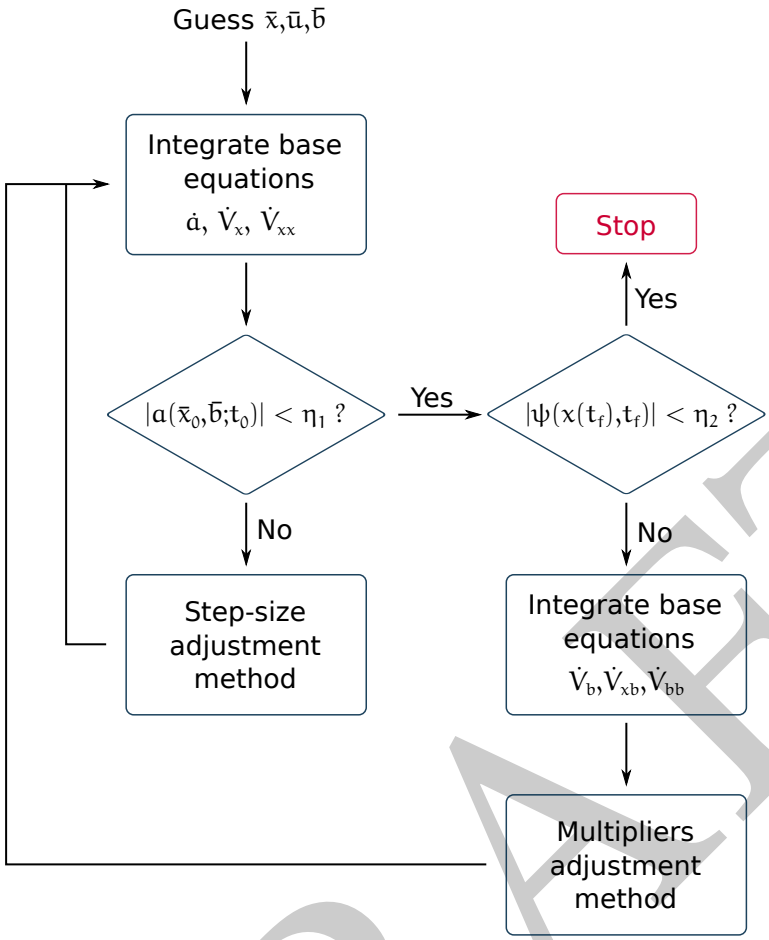


Figure 6.1: General outline of the differential dynamic programming algorithm.

6.1.2 *Detailed structure of the algorithm*

In this section, we describe the computational procedure as a detailed sequence of steps to be taken. Each of these steps expands the flowchart in Figure 6.1.

STEP 1 **Initialization.** Define a nominal control trajectory $\bar{u}(t)$ with $t \in [t_o, t_f]$ and a set of nominal Lagrange multipliers $\bar{b}$. Integrate the state equations to calculate the nominal state trajectory $\bar{x}(t)$

$$\frac{d\bar{x}}{dt} = f(\bar{x}, \bar{u}; t), \qquad \bar{x}(t_o) = x_o. \tag{6.6}$$

Calculate the nominal cost $\bar{V}(x_o, \bar{b}; t)$ with (5.40):

$$\bar{V}(x_o, \bar{b}; t) = \int_{t_o}^{t_f} L(\bar{x}, \bar{u}; t)\, dt + F(\bar{x}(t_f); t_f) + \bar{b}^T \psi(\bar{x}(t_f); t_f). \tag{6.7}$$

STEP 2 **Base equations (first set).** Integrate $\dot{a}$, $\dot{V}_x$, $\dot{V}_{xx}$ from (5.53), (5.54) and (5.58)

backward in time from t_f to t_o:

$$-\dot{a} = H - H(\bar{x}, \bar{u}, V_x, t),$$

$$-\dot{V}_x = H_x + V_{xx}(f - f(\bar{x}, \bar{u}, t)) + \hat{g}_x^T \mu,$$

$$-\dot{V}_{xx} = H_{xx} + f_x^T V_{xx} + V_{xx} f_x + \beta_1^T (H_{uu} + \mu \hat{g}_{uu}) \beta_1 + \\ + \beta_1^T (H_{ux} + \mu \hat{g}_{ux} + f_u^T V_{xx}) \beta_1 + \mu \hat{g}_{xx},$$

with boundary conditions (5.59), (5.60) and (5.64):

$$a(t_f) = o,$$

$$V_x(t_f) = F_x(\bar{x}(t_f); t_f) + \psi_x^T(\bar{x}(t_f))\bar{b},$$

$$V_{xx}(t_f) = F_{xx}(\bar{x}(t_f); t_f) + \bar{b}\psi_{xx}(\bar{x}(t_f)),$$

while obtaining $\hat{u}$ by minimizing $H(\bar{x}, u, V_x; t)$ subject to $g(\bar{x}, u; t) \leqslant o$.

If one or more inequality constraints $g(\bar{x}, u; t)$ are active, compute Z with (5.51) and Q with (5.50):

$$Q = (\hat{g}_u (H_{uu} + \mu \hat{g}_{uu})^{-1} \hat{g}_u^T)^{-1} \hat{g}_u (H_{uu} + \mu \hat{g}_{uu})^{-1}, \tag{6.8}$$

$$Z = I_m - \hat{g}_u^T Q; \tag{6.9}$$

else, set $Z = I_m$ and $Q = o$.

Compute $\beta_1(t)$ with (5.22):

$$\beta_1 = -(H_{uu} + \mu \hat{g}_{uu})^{-1} Z (H_{ux} + f_u^T V_{xx} + \mu \hat{g}_{ux}) - Q^T \hat{g}_x.$$

Store $\hat{u}$, $\beta_1(t)$, Z and $a(\bar{x}, \bar{b}; t)$ [†].

[†] In practice, it is also useful to store $(H_{uu} + \mu \hat{g}_{uu})^{-1}$, for later use in STEP 6.

STEP 3 **Cost convergence.** Check convergence of the inner layer: if the predicted change in cost $|a(\bar{x}_o, \bar{b}; t_o)|$ is smaller than a small quantity η_1, then $\bar{u}(t)$ is the optimal solution of the free endpoint problem with the augmented cost (5.40); proceed to STEP 5. Otherwise, proceed to STEP 4.

STEP 4 **Step-size adjustment method.** Start with $t_1 = t_o$ and integrate the state equations with the new controls $u = \hat{u} + \beta_1 \delta x$ over the whole time interval $[t_1, t_f]$.

While integrating the new state trajectory, ensure that the inequality constraints are not violated. If this happens, determine u by setting $\hat{g}(\bar{x}, u; t) = o$.

If the actual improvement in cost

$$\Delta V = \bar{V}(\bar{x}, \bar{b}; t_1) - V(\bar{x}, \bar{b}; t_1) \tag{6.10}$$

is close to the predicted improvement $|a(\bar{x}, \bar{b}; t_1)|$ in the sense defined by (5.38), i.e.

$$\frac{\Delta V}{|a(\bar{x}; t_1)|} > C^{\ddagger}, \tag{6.11}$$

replace the nominal trajectories with the new trajectories $x(t)$ and $u(t)$ and go to STEP 2. Else, change t_1 and repeat until (6.11) is satisfied, and then go to STEP 2.

[‡] C should be chosen according to the discussion in § 5.3.

STEP 5 **Endpoint convergence.** Check whether the endpoint error is within the specified tolerance, i.e.

$$\psi(x(t_f); t_f) < \eta_2. \tag{6.12}$$

If so, the algorithm has converged and $\bar{x}$, $\bar{u}$, $\bar{b}$ constitute the optimal solution of the problem. Else, go to STEP 6.

STEP 6 **Base equations (second set).** Integrate $\dot{V}_b, \dot{V}_{xb}, \dot{V}_{bb}$ from (5.55) to (5.56) backward in time from t_f to t_o:

$$-\dot{V}_b = V_{xb}^T \left(f - f\left(\bar{x}, \bar{u}, t\right)\right),$$

$$-\dot{V}_{xb} = (f_x^T + \beta_1^T f_u^T) V_{xb},$$

$$-\dot{V}_{bb} = -\beta_2^T (H_{uu} + \mu \hat{g}_{uu}) \beta_2,$$

with boundary conditions (5.61) to (5.63):

$$V_b(t_f) = \psi(\bar{x}(t_f)),$$

$$V_{xb}(t_f) = \psi_x^T(\bar{x}(t_f)),$$

$$V_{bb}(t_f) = 0.$$

Compute and store $\beta_2(t)$ with (5.49):

$$\beta_2 = -(H_{uu} + \mu \hat{g}_{uu})^{-1} Z f_u^T V_{xb}.$$

STEP 7 **Multipliers adjustment method.** Start with $\varepsilon = 1$. Evaluate

$$\delta b = -\varepsilon V_{bb}^{-1}(t_o) V_b^T(t_o) \tag{6.13}$$

and integrate the state equations with the new controls $u = \hat{u} + \beta_1 \delta x + \beta_2 \delta b$ over the whole time interval $[t_o, t_f]$.

First, check whether there was an improvement in the endpoint error, i.e.

$$\|\psi(x(t_f); t_f)\| < \|\psi(\bar{x}(t_f); t_f)\|. \tag{6.14}$$

If it does, check also that the actual change in cost

$$\Delta V = V(x_o, \bar{b} + \delta b; t_o) - \bar{V}(x_o, \bar{b}; t_o) \tag{6.15}$$

is close enough to the predicted change in cost.

$$\widetilde{\Delta V} = a(x_o, \bar{b}; t) - (\varepsilon - \frac{1}{2}\varepsilon^2) V_b(t_o)^T V_{bb}(t_o) V_b(t_o), \tag{6.16}$$

i.e. check that

$$\gamma_1 \leqslant \frac{\Delta V}{\widetilde{\Delta V}} \leqslant \gamma_2, \tag{6.17}$$

where $0 < \gamma_1 < 1$ and $\gamma_2 > 1$.

If both (6.14) and (6.17) are satisfied, replace the nominal trajectories with the new trajectories $x(t)$ and $u(t)$ and the nominal multipliers with the new multipliers $\bar{b} + \delta b$ and go to STEP 2.

Else, set $\varepsilon = \frac{\varepsilon}{2}$ and repeat this step. If (6.14) and (6.17) cannot be satisfied within a certain number of iterations, repeat this step without checking (6.17).

Algorithm 1: Fourth order Runge-Kutta integration routine.

function RK4(f, y_0, t_0, t_f, N)
  $h \leftarrow \frac{t_f - t_0}{N}$ ▷ Timestep length.
  $t \leftarrow t_0$
  $y \leftarrow y_0$
  for $n \leftarrow 1$ **to** N **do**
    $k_1 = f(t, y)$
    $k_2 = f\left(t + \frac{h}{2}, y + k_1\frac{h}{2}\right)$
    $k_3 = f\left(t + \frac{h}{2}h, y + k_2\frac{h}{2}\right)$
    $k_4 = f\left(t + h, y + k_3 h\right)$
    $y(t+h) = y(t) + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}h$
  end for
end function

This procedure entails using the feedback relationship given by $\beta_1(t)$ and $\beta_2(t)$ to evaluate new controls. If the computational trick described in § 5.6 is to be used instead, the following modifications must be made:

- STEP 2 must be modified to store $V_x(t)$ and $V_{xx}(t)$ rather than $\beta_1(t)$.
- In STEP 4, the new controls must be determined by

$$u = \arg\min_u H(\bar{x} + \delta x, u, V_x + V_{xx}\delta x; t) \tag{6.18}$$

  when integrating the state equations.
- STEP 6 must be modified to store $V_x b(t)$ rather than $\beta_2(t)$.
- In STEP 7, the new controls must be determined by

$$u = \arg\min_u H(\bar{x} + \delta x, u, V_x + V_{xx}\delta x + V_{xb}\delta b; t) \tag{6.19}$$

  when integrating the state equations.

6.2 INTEGRATION OF THE BASE EQUATIONS

The first practical issue that must be addressed is the integration in time of the base equations in STEP 2 and STEP 6. A robust numerical integration scheme must be selected in order to handle the corner points in the control trajectory that may arise on application of the step-size adjustment procedure[†] (see § 5.3).

[†]Discontinuities in the time derivative of $u(t)$.

Another issue is that it may not be possible to use one of the many available libraries of ode solvers. The base equations of our DDP algorithm are peculiar in that we must integrate them while simultaneously obtaining $\hat{u}$ by minimizing the Hamiltonian. If this cannot be done analytically, this means that a constrained NLP problem must be solved at each integration time step. Embedding this into the equations to be integrated may be impossible or impractical with available general-purpose ode solvers.

For our software, a specific fourth-order Runge-Kutta [26] solver was implemented as in Algorithm 1, which solves the Hamiltonian minimization NLP at each time step. Here, the integration variables y can as a vectorized form of a and

the entries of V_x, V_{xx} for the integration run in STEP 2 and the vectorized entries of V_b, V_{xb} and V_{bb} for the integration run in STEP 6.

The same integration routine has also been used to integrate the state equations forward in time. In this case, the integration variables y are the state variables.

Listing 6.1 shows the code for the integration of the base equations in STEP 2. A time vector `ts`, defined by the user, is used to define the integration steps. Each row of the integration variable `y` contains the elements of a, V_x and V_{xx} for the corresponding integration step. The last row is initialized to the boundary conditions, which were previously evaluated as in Listing 6.2.

Listing 6.1: Function for the integration of the base equations $\dot{a}$, $\dot{V}_x$, $\dot{V}_{xx}$.

```
function [ts, y, u_hat, beta1, Z, A] = Vx_bwd(ts, yf, x_nom, u_nom,
    hessMinSolver, f, H, g, gb, nx, nu)

% Initial conditions
y(end, :) = yf;

for n = length(ts):-1:1
    yy = y(n, :);
    xx_nom =  x_nom(n, :)';
    uu_nom = u_nom(n, :)';
    Vx = reshape(yy(2:(1+nx)), [1 nx]);

    [uu_hat, g_hat] = get_u_hat( xx_nom, Vx, ts(n), hessMinSolver,
        uu_nom, g, gb);
    u_hat(n,:) = uu_hat;

    [k_1, beta1(n,:,:), Z(n,:,:), A(n,:,:)] = ...
        Vx_odefun( ts(n), yy, xx_nom, uu_nom, uu_hat, f, H, g_hat, nx
            , nu);
    if n>1
        % Timestep
        h = ts(n-1) - ts(n);
        % Fourth order runge kutta
        k_2 = ...
            Vx_odefun( ts(n) + h/2, yy + k_1*h/2, xx_nom, uu_nom,
                uu_hat, f, H, g_hat, nx, nu);
        k_3 = ...
            Vx_odefun( ts(n) + h/2, yy + k_2*h/2, xx_nom, uu_nom,
                uu_hat, f, H, g_hat, nx, nu);
        k_4 = ...
            Vx_odefun( ts(n) + h, yy + k_3*h, xx_nom, uu_nom, uu_hat,
                f, H, g_hat, nx, nu);
        y(n-1, :) = yy + (1/6)*( k_1 + 2*k_2 + 2*k_3 + k_4 )*h;

    end

end

end
```

Listing 6.2: Boundary conditions $a(t_f)$, $V_x(t_f)$, $V_{xx}(t_f)$.

```
% Evaluate boundary conditions
a_f = 0;
Vx_f = F.Fx( x_nom(end,:), ts(end) ) + psi.psix( x_nom(end,:), ts(end
    ) ) .' *  b;
```

```
Vxx_f = F.Fxx( x_nom(end,:), ts(end) ) + tensor_times_vector(zeros(
    npsi,nx,nx), b, 1);
yf = [a_f; Vx_f(:); Vxx_f(:)];
```

Then, the integration loop is run. First, the relevant quantities for the current integration step are retrieved and renamed for convenience. The nominal state and controls as well as V_x are also put in column vector form, coherently with the conventions we adopted throughout this text.

The function `get_u_hat` is then used to obtain $\hat{u}$ by minimizing $H(\bar{x}, u, V_x; t)$ subject our state and control constraints, which are stored in `g` and `gb`[†]. This will be described in § 6.4; for now, let us note that, in addition to $\hat{u}$, it also returns the active constraints (`g_hat`); and that $\hat{u}$ is also stored for each integration step for later use.

[†] `gb` stores the box constraints on u, as these are generally treated differently from the other constraints by NLP solvers.

Finally, the function integrates one step forward as we just illustrated in Algorithm 1. The function `Vx_odefun`, shown in Listing 6.3, evaluates the time derivatives $\dot{a}$, $\dot{V}_x$ and $\dot{V}_{xx}$ for a given time; hence, it can be used by any integrator. Additionally, it can be used to return β_1, Z and $A = (H_{uu} + \mu\hat{g}_{uu})^{-1}$, so that they can be stored for later usage; obviously, this is done for the time nodes of the integration steps only and not for intermediate evaluation steps.

Listing 6.3: Function evaluating the time derivatives $\dot{a}$, $\dot{V}_x$, $\dot{V}_{xx}$.

```
function [dydt, beta1, Z, A] = Vx_odefun(t, y, x_nom, u_nom, u_hat, f
    , H, g_hat, nx, nu)

    Vx = reshape(y(2:(1+nx)), [1 nx]);
    Vxx = reshape(y(2+nx:end), [nx nx]);

    % System dynamics and Hamiltonian
    f = f.fun(x_nom, u_hat, t);
    H = H.fun(x_nom, u_hat, Vx, t);
    f_nom = f.fun(x_nom, u_nom, t);
    H_nom = H.fun(x_nom, u_nom, Vx, t);
    % System dynamics and Hamiltonian derivatives
    fx = f.fx(x_nom, u_hat, t);
    fu = f.fu(x_nom, u_hat, t);
    Hx = H.Hx(x_nom, u_hat, Vx, t)';
    Hxx = H.Hxx(x_nom, u_hat, Vx, t);
    Hux = H.Hux(x_nom, u_hat, Vx, t);
    Huu = H.Huu(x_nom, u_hat, Vx, t);

    % Number of active constraints
    p_hat = length(g_hat);

    if p_hat > 0
        % Constraints derivatives
        for n = 1:p_hat
            gx_hat(n,:) = g_hat{n}.gx(x_nom, u_hat, t);
            gu_hat(n,:) = g_hat{n}.gu(x_nom, u_hat, t);
            gxx_hat(n,:,:) = g_hat{n}.gxx(x_nom, u_hat, t);
            gux_hat(n,:,:) = g_hat{n}.gux(x_nom, u_hat, t);
            guu_hat(n,:,:) = g_hat{n}.guu(x_nom, u_hat, t);
        end

        % Multiplier function
        Hu = full( H.Hu(x_nom, u_nom, Vx, t) )';
        mu = - inv(gu_hat * gu_hat.') * gu_hat * Hu;

        mu_gxx = tensor_times_vector(gxx_hat, mu, 1);
```

```
        mu_guu = tensor_times_vector(guu_hat, mu, 1);
        mu_gux = tensor_times_vector(gux_hat, mu, 1);

        % Evaluate Q, Z, beta1
        A = inv(Huu +  mu*guu_hat);
        Q = inv(gu_hat * A *  gu_hat.') * gu_hat * A;
        Z = eye(nu) - gu_hat.' * Q ;
        beta1 = - A * Z * (Hux + mu_gux + fu.'*Vxx) - Q .' * gx_hat;

    else
        % No constraints are active
        A = inv(Huu);
        Z = eye(nu);
        mu = 0;
        gx_hat = 0;
        mu_gxx = 0;
        mu_guu = 0;
        mu_gux = 0;

        % Evaluate beta1
        beta1 = - (Z * A) * (Hux + fu.'*Vxx);
    end

    % Evaluate da/dt, dVx/dt, dVxx/dt
    a_dot = H - H_nom;
    Vx_dot = Hx + Vxx * ( f - f_nom ) + (gx_hat.' * mu);
    Vxx_dot = Hxx + mu_gxx + fx.' * Vxx + Vxx * fx + ...
        + beta1.' * (Huu + mu_guu) * beta1 + ...
        + beta1.' * (Hux + mu_gux + fu.'*Vxx) + ...
        + (Hux + mu_gux + fu.'*Vxx).' * beta1;

    dydt = - [a_dot; Vx_dot(:); Vxx_dot(:)];

end
```

Similarly, a function `Vb_bwd` was defined to integrate the base equations in STEP 6, using another function `Vb_odefun` to evaluate $\dot{V}_b$, $\dot{V}_{xb}$ and $\dot{V}_{bb}$ and optionally to return β_2 to be stored for later use in the multipliers adjustment method.

At this point, it is worthwhile to discuss an implementation issue which arises in the integration of the base equations. The RK4 integrator involves evaluating our function[†] at intermediate times within each integration interval to estimate slopes. This is true in general of all integration methods of order higher than one. In principle, this means that in order to accurately integrate the base equations one should evaluate $\hat{u}$ at these intermediate times.

[†] i.e. $\dot{\alpha}(t)$, $\dot{V}_x(t)$ and $\dot{V}_{xx}(t)$.

However, unless one is able to evaluate $\hat{u}(t)$ analytically, this would be impractical. First, it should be noted that the obtaining by minimizing the Hamiltonian with numerical methods is one of the most computationally expensive tasks of the algorithm. If we were to adopt this strategy with the RK4 integrator, we would increase the number of NLP solutions by a factor of four, which will have a strong impact on the overall computational time.

Secondly, this would introduce a discrepancy between the accuracy of the computed $\alpha(t)$, $V_x(t)$ and $V_{xx}(t)$ and the accuracy of $\hat{u}(t)$ and $\beta_1(t)$, since the latter would only be stored for the nodes of the integration steps. This will make it hard to compare[‡] the true reduction in cost (which is a product of $\hat{u}(t)$ and $\beta_1(t)$) and the predicted change in cost (which is represented by $\alpha(t)$), with a negative impact on our ability to control the convergence of the algorithm.

[‡] for example, when evaluating new trajectories within the step-size adjustment method (STEP 4) or when checking the cost convergence in STEP 3.

One way to avoid this issue altogether would be to adopt a first-order integration routine, such as Euler's method, and using an increased number of inte-

gration steps. In our experiments, this approach proved ineffective and was subsequently discarded; this is likely attributable to the bad performance of Euler's method in the presence of discontinuities or rapidly changing integration variables.

Hence, in our algorithm, we always evaluate the base equations using $\hat{u}$ as evaluated at the left node of each integration step, thus introducing a small error when using the RK4 integrator. While in our experience this did not prevent the algorithm from effectively solving optimal control problems, as demonstrated by the toy example in § 6.7 and the applications in § 7, it is the author's opinion that the convergence of the algorithm might be greatly improved if a method to overcome this issue was devised.

6.3 Taking derivatives

An important feature of the DDP algorithm is the integration of the base equations, which require the evaluation of the first and second derivatives of the system dynamics, Hamiltonian and constraints.

Similarly to the minimization of the Hamiltonian, some simple problems may allow an analytical derivation of these derivatives, either by hand or by using a computer algebra system (CAS) software[†].

[†] Some popular examples are Maple, Mathematica or Matlab's Symbolic Computation Toolbox

Unless the problem is fully scalar[‡], these derivatives are Jacobian (such as f_x, H_x) or Hessian matrices (such as H_{uu}); the entries may be far too many and their derivatives far too complex to use an analytical approach. In this case, two derivation methods can be employed: numerical differentiation and automatic differentiation.

[‡] I.e. the system has only one state variable and one control variable.

Numerical differentiation is perhaps the best known approach of the two. At its core, it consists in using a finite differences approximation. For example, first and second derivatives of a function $f(x)$ at some point x_0 may be calculated using central differences scheme:

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0 - h)}{h}. \tag{6.20}$$

$$f''(x_0) \approx \frac{f(x_0 + h) - 2f(x_0)f(x_0 - h)}{h^2}. \tag{6.21}$$

where h is a suitable (small) step size. More sophisticated techniques can be used to improve on the accuracy of the solution, such as the adoption of higher-order differences to obtain a better approximation (at the cost of more function evaluations) or using Richardson extrapolation [6, Chapter 14].

Nonetheless, numerical differentiation is relatively computationally expensive and inaccurate because of cancellation and round-off errors. As h gets smaller, $f(x_0 + h)$ gets closer to $f(x_0 - h)$ and the difference between the two may have very few significant digits, meaning that its floating-point representation is more prone to numerical errors such has cancellation [24].

[§] Also know as algorithmic differentiation.

Automatic differentiation[§] is an approach to obtain first and higher order derivatives which at its core is based on systematic application of the chain rule. The main practical advantages of AD is that it is accurate to machine precision and has a significantly lower computational cost with respect to numerical differentiation [25].

Automatic differentiation is a very wide and ever-evolving field which comprises many variants and techniques which are beyond the scope of this thesis. Readers are referred to [25, 61] for an extensive treatment.

[‖] https://web.casadi.org/

For this work, Automatic Differentiation was implemented by using CasADi[‖], an open-source C++ software framework with MATLAB and Python interfaces [2].

In particular, CasADi was used to obtain the derivatives of all quantities that appear in the base equations.

Listing 6.4 illustrates how this is set up using CasADi commands and stored. For conciseness, we only show how derivatives of the system dynamics $\mathtt{f}(\mathtt{x}, \mathtt{u}; \mathtt{t})$ and the Hamiltonian $\mathtt{H}(\mathtt{x}, \mathtt{u}, V_{\mathtt{x}}; \mathtt{t})$ are defined. First, the dependent variables $\mathtt{x}$, $\mathtt{u}$, $V_{\mathtt{x}}$ and $\mathtt{t}$ must be defined as symbolic variables using the `MX` symbolics[†] and their dimension[‡] must be specified[§].

Then, a structure `f` is created to store CasADi functions to evaluate `f` (`f.fun`), $\mathtt{f}_{\mathtt{x}}$ (`f.fx`) and $\mathtt{f}_{\mathtt{u}}$ (`f.fu`). CasADi functions are objects that are created using the `Function()` constructor, where the first input is a tag for the function's name, the second input specifies the dependent variables (which must be CasADi symbolic variables, such as MX symbolics). The third input is a CasADi symbolic expression which defines the function. These functions can then be used in `Vx_odefun` to numerically evaluate them for given values of their dependent variables.

The command `fexpr = fexpr(x,u,t)`, which may look redundant, is used to convert the user-supplied function handle for `f` into a CasADi symbolic expression. This is simply done by calling the function handle using the `MX` symbolic variables as inputs.

This is needed because DDPtoolbox requires users to specify the system dynamics, running cost and terminal cost as function handles (a built-in MATLAB type). This choice was made so that new potential users do not have to learn any additional syntax in addition to pure MATLAB and to maintain a high level of modularity, so that CasADi may be easily replaced with other tools if needed in future developments.

In defining the derivatives of `f`, the symbolic expressions are generated starting from the symbolic expression for `f` itself by using the `jacobian` command, where the second input specifies the derivation variables. Note that these symbolic expressions do not actually contain analytic expressions for these derivatives; rather, they store the computational graphs that are needed to evaluate derivatives by automatic differentiation.

Similarly to the manner in which `f` and the graphs for evaluating its derivatives are created and store in a structure `f`, the Hamiltonian and its derivatives are stored in a structure `H`. This time however the second-order derivatives are also generated using the `hessian`, whose syntax is analogous to `jacobian`. A similar procedure is repeated for the terminal cost, endpoint error and constraints, which are stored in the structures `F`, `psi`, `g` and `gb`. The reason why the box constraints are stored in a different structure `gb` with respect to the other general constraints `g` is that NLP solvers generally treat these two differently, as is the case for the IPOPT solver shown in § 6.4.

[†] `MX` stands for *matrix expression*, as these objects are intended for matrix and vector algebra.

[‡] `nx` and `nu` represent the number of states and controls $\mathtt{n}$ and $\mathtt{m}$.

[§] Users that are familiar with computer algebra systems and symbolic computation will find this syntax familiar. This is due to the fact that CasADi's interface was specifically designed to emulate CAS syntax, hence the first part of its name.

Listing 6.4: Setup of the system dynamics and Hamiltonian derivatives functions.

```
%% Set up Casadi Stuff
import casadi.*

x = MX.sym('x', nx);
Vx = MX.sym('Vx', nx);
u = MX.sym('u', nu);
t = MX.sym('t', 1);

% State equations
fexpr = fexpr(x,u,t);
f.fun = Function('f', {x, u, t}, {fexpr});
% Create the Jacobian expressions
f.fx = Function('fx', {x, u, t}, {jacobian(fexpr, x)});
f.fu = Function('fu', {x, u, t}, {jacobian(fexpr, u)});

```

```
% Hamiltonian
Hexpr = L(x,u,t) + Vx.' * fexpr;
H.expr = Hexpr;
H.fun = Function('H', {x, u, Vx, t}, {Hexpr});
% Create the Jacobian expressions
H.Hx = Function('Hx', {x, u, Vx, t}, {jacobian(Hexpr, x)});
H.Hu = Function('Hu', {x, u, Vx, t}, {jacobian(Hexpr, u)});
% Create the Hessian expressions
Hexpr = hessian(Hexpr, [x; u]);
H.Hxx = Function('Hxx', {x, u, Vx, t}, {Hexpr(1:nx, 1:nx)});
H.Hux = Function('Hux', {x, u, Vx, t}, {Hexpr(nx+1:end, 1:nx)});
H.Huu = Function('Huu', {x, u, Vx, t}, {Hexpr(nx+1:end, nx+1:end)});
```

6.3.1 *Limitations of CasADi*

When building the computational graphs for derivatives for a certain function, one must ensure that all operations used to define that function are supported by the AD engine.

While CasADi supports most of MATLAB's common operators, there is one type that is frequently used in powertrain modeling that requires some additional care, and that is interpolant objects. Interpolant objects (such as `griddedInterpolant`) are frequently used to characterize the behavior of map-based components. For example, in § 7, linear interpolants will be used to characterize the battery's open-circuit voltage v_{oc} as a function of its state of charge σ.

Unfortunately, CasADi does not support MATLAB's interpolant object. However, it does provide its own interpolant class. Therefore, the current way to use interpolant objects in DDPtoolbox is to define them using CasADi; for example, the $v_{oc}(\sigma)$ characteristic for § 7 was implemented as

```
batt.ocVolt = casadi.interpolant('ocVolt', 'linear', socData, ocVoltData);
```

6.4 The Hamiltonian minimization

Another important step of the computational procedure is the minimization of the Hamiltonian with respect to u along the nominal state trajectory that is needed to obtain $\hat{u}$.

$$\hat{u} = \underset{g(\bar{x},u;t)\leqslant 0}{\arg\min}\ H(\bar{x},u,Vx;t). \tag{6.22}$$

In some cases, it may be possible to perform this minimization analytically. For more complex applications, this is generally not possible in practice. Therefore, a suitable algorithm is necessary to solve this sub-problem.

This Hamiltonian minimization is naturally cast as a nonlinear optimization problem and can be solved with a suitable non-linear programming (NLP) solver. In the software developed for this work, the widely popular IPOPT solver [83] was used.

IPOPT uses the gradient and hessian of the objective function (in our case, the Hamiltonian) and constraints to generate solutions. If these are not available, it estimates them using a quasi-Newton method; however, it is greatly beneficial to directly provide them to the solver. To this end, we can once again resort to automatic differentiation.

In particular, CasADi includes an interface to IPOPT[†] which allows to easily use the same AD tools to efficiently exploit the solver's full capabilities. The NLP is represented by an object which defines the objective functions and constraints, possibly depending on some parameters.

[†]Other popular NLP solvers can be interfaced to CasADi, both commercial and freely distributed.

In Listing 6.5, the basic definition of an NLP problem using CasADi is illustrated. A problem structure `prob` defines the objective function, the optimization variable and the parameters by which the objective function is parameterized. For our Hessian minimization problem, in the unconstrained case, the objective function would be the Hessian, the parameters would be the x, V_x and t and the optimization variable is u.

Then, the problem structure is passed as an input to the `nlpsol` constructor, specifying the solver to be used and, if needed, additional settings using a dedicated structure.

Listing 6.5: NLP solver setup

```
% Objective function
prob.f = H.fun(x,u,Vx,t);
% Optimization variable
prob.x = u;
% Parameters
prob.p = vertcat(x, Vx, t);
% Constraints
prob.g = gexprs;
% Additional settings
opts = ⟨Define options structure⟩
% Create the NLP solver
hessMinSolver = nlpsol('solver', 'ipopt', prob, opts);
```

For constrained problems, the constraints expressions are added in an additional field `g` of the problem structure. Note that these do not include the box constraints, which are passed as inputs when the solver is used.

The solver can be run as in Listing 6.6 by specifying an initial guess for $\hat{u}$, the value of the parameters (in our case, $\bar{x}(t)$, $V_x(t)$ and t) and lower and upper bounds for u and the constraints $g(x, u, t)$. Since in our DDP framework the constraints are all defined as $g(x, u, t) \leqslant 0$, the lower and upper bounds for these constraints are always $-\infty$ and 0. The lower and upper bounds for u on the other hand are simply our box constraints, whose numerical values were previously stored in the cell `gb3`.

Listing 6.6: NLP solver usage

```
sol = hessMinSolver('x0', u_guess, 'p', [x_nom(n); Vx(n); ts(n)], '
    lbx', gb{3}(1), 'ubx', gb{3}(2), 'lbg', -inf, 'ubg', 0);
u_hat = sol.x;
```

An important limitation to keep in mind is that IPOPT, like most efficient NLP solvers, is not a global solver. This means that if the objective function is particularly challenging, it may converge to a local minimum and not the global minimum.

Although it is impossible to guarantee the absolute optimality of the solution with a local optimizer, there are some ways to reduce the chances of converging to a local minimum. The first and most effective is to provide a good first guess of the solution, although this may be challenging in practice because it requires some understanding of the shape of the objective function.

Another effective method is resorting to *multistart*, that is running the solver several times with different starting points and the selecting the best solution. For example, one might start the algorithm using the lower and upper bound of the optimization variable and an intermediate value. A three-point multistart can be selected in DDPtoolbox using a dedicated `NLPmultiStart` option.

6.5 PUTTING IT ALL TOGETHER

In the previous sections, we focused our attention to specific aspects of the algorithm that the author deems particularly important and where some original contributions of this thesis work were developed. In this section, a general outline of the computer program and the parts that were not addressed yet is offered.

6.5.1 *The forward integrator*

The `sys_fwd` function in Listing 6.7 is used to integrate the state equations, which is needed in several different steps of the algorithm:

- To generate the first nominal state trajectory given the nominal control trajectory;
- To generate new trajectories in the step-size adjustment method;
- To generate new trajectories in the multipliers adjustment method.

In order to accommodate all these different usages, the function can be called with different signatures[†]. In practice, name-value pair arguments are specified after the mandatory positional arguments to alter the behavior of the function. The positional arguments are the state dynamics `f`, the initial state `x_0`, the nominal state trajectory `x_nom` and the $\hat{u}$ trajectory `u_hat`. The name-value pair arguments that can be specified are defined by an `arguments` block[‡], which is not shown in Listing 6.7 for conciseness.

[†] Different input arguments.

[‡] See https://www.mathworks.com/help/matlab/matlab_prog/function-argument-validation-1.html for more information on argument validation in MATLAB.

Listing 6.7: The state dynamics integrator.

```
function [ts, x, u] = sys_fwd(f, x0, ts, x_nom, u_hat, nvp)

% Initial conditions
x(1, :) = x0;

% Integration routine
for n = 1:length(ts)
    xx = x(n, :);
    xx_nom = x_nom(n,:)';
    uu_hat = u_hat(n,:)';

    % Get new controls
    u(n, :) = get_u;

    % Advance the simulation
    if n<length(ts)
        % Timestep
        h = ts(n+1) - ts(n);
        % Integrate one step fwd
        k_1 = f.fun( xx, u(n, :), ts(n) );
        k_2 = f.fun( xx + k_1*h/3, u(n, :), ts(n) + h/3 );
        k_3 = f.fun( xx - k_1*h/3 + k_2*h, u(n, :), ts(n) + h*2/3 );
        k_4 = f.fun( xx + k_1*h - k_2*h + k_3*h, u(n, :), ts(n) + h )
            ;
        x(n+1, :) = xx + (1/8)*( k_1 + 3*k_2 + 3*k_3+k_4 )*h;

    end
end

function u_new = get_u
    ⟨...⟩
```

```
end

end
```

The function itself is fairly simple: it sets the initial state and uses an RK4 integration scheme while using the function `get_u` to generate the new control to be adopted. Note that this function, shown in Listing 6.8, is nested to `sys_fwd`, meaning that it shares its workspace[†].

[†] In other words, it `get_u` has access to the variables passed to and those defined in `sys_fwd`.

Listing 6.8: Function to generate the new controls at a given time step.

```
function u_new = get_u
% Get new controls

if nvp.initialization
    % State traj initialization: u_hat is u_nom
    u_new = uu_hat';
elseif nvp.trick
    % Use the computational trick
    dx = ( xx - xx_nom );
    Vx = reshape(nvp.Vx(n,:), nx, 1);
    Vxx = reshape(nvp.Vxx(n,:), nx, nx);
    if isempty(nvp.Vxb)
        sol = nvp.hessMinSolver('x0', uu_hat, 'p', [xx; Vx + Vxx*dx;
            ts(n)], 'lbx', nvp.gb{3}(1), 'ubx', nvp.gb{3}(2), ...
            'lbg', -inf, 'ubg', 0);
    else
        Vxb = reshape(nvp.Vxb(n,:), nx, length(nvp.db));
        sol = nvp.hessMinSolver('x0', uu_hat, 'p', [xx; Vx + Vxx*dx +
             Vxb*nvp.db; ts(n)], 'lbx', nvp.gb{3}(1), 'ubx', nvp.gb
            {3}(2), ...
            'lbg', -inf, 'ubg', 0);
    end
    u_new = sol.x;
else
    % Use beta1 and beta2
    if isempty(nvp.beta2)
        u_new = uu_hat' + ( reshape(nvp.beta1(n,:,:), nu, nx) * ( xx
            - xx_nom ) )';
    else
        u_new = uu_hat' + ( reshape(nvp.beta1(n,:,:), nu, nx) * ( xx
            - xx_nom ) )' + (reshape(nvp.beta2(n,:,:), nu, npsi) *
            nvp.db)';
    end
end

% Enforce constraints on u(t)
% General constraints
if ~isempty(nvp.g)
    % Check constraints violation
    constrValue = cellfun( @(c) full( c.fun(xx, u_new, ts(n)) ), nvp.
        g );
    activeConstraints = constrValue > 0;
    % If the constraints were violated, set u to satisfy them
    if any(activeConstraints)
        [~, activeConstraints] = max(constrValue);
        u_new =  fsolve(@(u) nvp.g{activeConstraints}.fun(xx, u, ts(n
            )) , 0);
    end
end
```

```
% Box constraints
if ~isempty(nvp.gb)
    % Enforce box constraints
    u_new = min(max(u_new, nvp.gb{3}(1)), nvp.gb{3}(2));
end

end
```

The function `get_u` can use different methods to evaluate the new controls. First, if the flag `initialization` is set to `true`, the new control is simply copied from `u_hat`, and then adjusted so that the constraints are not violated. This is used during the algorithm's initialization when generating the nominal state trajectory with the nominal control trajectory.

If the computational trick is enabled (the flag `compTrick` is `true`), then $V_x(t)$, $V_{xx}(t)$ and the NLP solver for the Hamiltonian minimization must be among the inputs of `sys_fwd` so that the new controls $u(t)$ can be determined by (5.71). If $V_{xb}(t)$ was also provided as an input, then (5.72) is used instead. These two alternatives cover the usage of the computational trick while integrating the state dynamics during the step-size and the multiplier adjustment method respectively.

If the computational trick[†] is disabled, then $\hat{u}(t)$ and $\beta_1(t)$ must be among the inputs of `sys_fwd` so that the new controls can be computed with (5.42), and this covers the case where `sys_fwd` is called within the step-size adjustment method. If $\beta_2(t)$, then (5.41) is used instead, and this covers the case where `sys_fwd` is called within the multipliers adjustment method.

In any case, constraints violation due to the new controls is always checked. If this happens, then $u(t)$ is determined by the condition $\hat{g}(\bar{x}, u; t) = 0$.

[†] See § 5.6.

6.5.2 *The main function*

The main function controlling the flow of the optimization algorithm is the function `diffDP`, whose implementation is sketched in Listing 6.9. The inputs to the function are:

- Expressions (function handles) for the state dynamics `f`, running cost `L` and terminal cost `F`.
- Expressions for the inequality constraints `g` and the box constraints for the controls[‡].
- The initial state `x0` and, optionally, the terminal state `xf`.
- A discretized time vector `ts`.
- Initial values of the nominal control trajectory `u_nom` and the nominal multipliers `b`.
- An options structure for defining the algorithm's settings.

[‡] Lower and upper bounds for u.

First, the options structure is parsed and unspecified settings are set to their default values. The state dynamics, running cost, terminal cost and constraints are then transformed into CasADi functions and their derivatives are generated and ehe endpoint error and its derivatives are also generated, as described in § 6.3 and Listing 6.4. Furthermore, the NLP solver for the Hamiltonian minimization is created as shown in Listing 6.5.

The initialization phase ends after generating the nominal state trajectory with the state dynamics integrator `sys_fwd`.

Listing 6.9: Main function for the differential dynamic programming algorithm.

```
function [x_nom, u_nom, V0] = diffDP(f, L, F, g, u_bound, x0, xf, ts,
     u_nom, b, options)

% Parse options
⟨Parse options and set defaults⟩

% Set up CasADi
⟨Listing 6.4⟩

% Nominal state traj
[ts, x_nom, u_nom] = sys_fwd(f, x0, ts, zeros(length(ts), nx), u_nom,
     options, 'g', g, 'gb', gb);

% Main loop
for iter2 = 1:options.maxIter2
    for iter1 = 1:options.maxIter1
        % Integrate a, Vx, Vxx base equations
        % Evaluate boundary conditions
        a_f, Vx_f, Vxx_f = ⟨Boundary conditions (5.59), (5.60) and (5.64)⟩
        yf = [a_f; Vx_f(:); Vxx_f(:)];

        % Integrate backwards in time
        % ys: a, Vx, Vxx
        [ts, ys, u_hat, beta1, Z, A] = Vx_bwd(ts, yf, x_nom, u_nom,
            hessMinSolver, f, H, g, gb, nx, nu, options);

        a = ys(:,1);

        % Time t_eff
        N_eff = find(abs(a) < options.eta1, 1, 'first');

        % Check inner layer convergence
        if N_eff <= 1
            % Inner layer converged
            break
        end

        % Step-Size Adjustment
        if options.compTrick
            [x_nom, u_nom, halt] = stepSizeAdj(f, L, F, psi, ys, ts,
                ...
                x_nom, u_nom, u_hat, b, N_eff, options, ...
                'trick', true, 'g', g, 'gb', gb, ...
                'hessMinSolver', hessMinSolver);
        else
            [x_nom, u_nom, halt] = stepSizeAdj(f, L, F, psi, ys, ts,
                ...
                x_nom, u_nom, u_hat, b, N_eff, options, ...
                'beta1', beta1, 'g', g, 'gb', gb);
        end

        if halt
            % Failed to converge
            break
        end

    end

```

```
    % Check convergence
    if full( norm(psi.fun(x_nom(end,:), tf)) ) < options.eta2 ||
         options.maxIter2 == 1
        % The algorithm converged
        break
    end

    % Integrate Vb, Vxb, Vbb base equations
    % Evaluate boundary conditions
    Vb_f, Vxb_f, Vbb_f = ⟨Boundary conditions (5.61) to (5.63)⟩
    yf = [Vb_f(:); Vxb_f(:); Vbb_f(:)];

    % Integrate backwards in time
    % ys: Vb, Vxb, Vbb
    [~, ys, beta2] = Vb_bwd(ts, yf, x_nom, u_nom, u_hat, beta1, A, Z,
         f, nx, nu, npsi, options);

    % Multipliers adjustment method
    if options.compTrick
        [x_nom, u_nom, b] = mltpAdj(f, L, F, psi, ys, ts, ...
            x_nom, u_nom, u_hat, b, options, ...
            'a', a, 'g', g, 'gb', gb, ...
            'hessMinSolver', hessMinSolver, 'Vx', Vx, 'Vxx', Vxx);
    else
        [x_nom, u_nom, b] = mltpAdj(f, L, F, psi, ys, ts, ...
            x_nom, u_nom, u_hat, b, options, ...
            'a', a, 'g', g, 'gb', gb, 'beta1', beta1, 'beta2', beta2)
                ;
    end

end

end
```

The main algorithm then runs with two nested loops. The inner loop obviously contains what we called the inner layer in § 6.1.1. The base equations for a, V_x and V_{xx} are integrated backwards in time using the function `Vx_bwd` shown in Listing 6.1. Convergence of the inner layer is examined by checking whether the time interval N_{eff} where the predicted change in cost $|a(\bar{x}_o \bar{b}; t_o)|$ is smaller than the tolerance η_1 is the first time interval, which is equivalent to the check in STEP 3 of the algorithm's outline (§ 6.1.2).

If the algorithm hasn't converged, the `stepSizeAdj` function is used to generate new control and state trajectories; if needed, the step-size adjustment method is used, as described in § 6.5.3. The inner layer is repeated until convergence, at which point the algorithm moves to the outer layer.

Convergence of the outer layer is checked by also checking whether the endpoint error is smaller than the specified tolerance η_2. If this is verified, the the algorithm has converged; if not, the base equations for V_b, V_{xb} and V_{bb} are integrated backwards in time using the function `Vb_bwd` and the multipliers adjustment method § 6.5.4 is used to generate new multipliers b as well as new control and state trajectories.

6.5.3 *The step-size adjustment method*

First, the parameter C used in check (5.38) and two flow control flags are initialized. An outer loop is used to repeat the whole step-size adjustment method with a less stringent C if needed. The inner loop is used to implement the method it-

self. N_1 is initialized so that it will be equal to 1 at the first iteration; then, it is used to split the nominal state and control trajectories into two vectors.

Note that `stepSizeAdj` can be called with different signatures depending on whether the computational trick must be used or not to generate new control trajectories. This in turn changes the way in which `stepSizeAdj` calls `sys_fwd`, as explained in § 6.5.1. After using `sys_fwd` to integrate the state dynamics for t_1 to t_f, the whole new state and control trajectories are created by concatenating $\bar{x}(t)$ and $\bar{u}(t)$ for $t < t_1$ and the newly generated $x(t)$ and $u(t)$ for $t \geqslant t_1$.

Listing 6.10: Step size adjustment method.

```
function [x_nom, u_nom, halt] = stepSizeAdj(f, L, F, psi, ys, ts,
    x_nom, u_nom, u_hat, b, N_eff, options, nvp)

% Acceptance criterion parameter
C = 0.5;
% Initialize flags
to_next_iter = false;
halt = false;

while true
    N1 = 2 - N_eff;
    while N1 < (N_eff-1)
        % Adjust N1
        N1 = ceil( (N_eff - N1) / 2 + N1 );

        % (Re-)evaluate new nominal trajectories
        ts_t0_t1 = ts( 1:N1-1 )'; % Time nodes from t0 to t1
        ts_t1_tf = ts( N1:end )'; % Time nodes from t1 to tf

        x_nom_t0_t1 = x_nom( 1:N1-1, : );
        u_nom_t0_t1 = u_nom( 1:N1-1, : );
        x_nom_t1_tf = x_nom( N1:end, : );
        u_hat_t1_tf = u_hat( N1:end, : );

        % Apply the new control trajectory from t1 to the end
        if options.compTrick
            % Obtain the new control trajectory with the
              computational trick
            Vx = ys( N1:end, 2:nx+1 );
            Vxx = ys( N1:end, nx+2:end );
                    [ts_t1_tf, x_nom_new_t1_tf, u_nom_new_t1_tf]
                      = ...
                sys_fwd(f, x_nom(N1,:), ts_t1_tf, x_nom_t1_tf,
                    u_hat_t1_tf, options, 'g', nvp.g, 'gb', nvp.gb, '
                    hessMinSolver', nvp.hessMinSolver, 'Vx', Vx, 'Vxx
                    ', Vxx);

        else
            % Obtain the new control trajectory with beta1 * dx
            beta1_t1_tf = nvp.beta1( N1:end, : );
            [ts_t1_tf, x_nom_new_t1_tf, u_nom_new_t1_tf] = sys_fwd(f,
                 x_nom(N1,:), ts_t1_tf, x_nom_t1_tf, u_hat_t1_tf,
                options, 'beta1', beta1_t1_tf, 'g', nvp.g, 'gb', nvp.
                gb);
        end

        x_nom_new = [x_nom_t0_t1; x_nom_new_t1_tf];
        u_nom_new = [u_nom_t0_t1; u_nom_new_t1_tf];

```

```
        % Actual improvement in cost
        V_nom = totalCost(x_nom_t1_tf, u_nom( N1:end, : ), ts_t1_tf,
            L, F, b, psi);
        V = totalCost(x_nom_new_t1_tf, u_nom_new_t1_tf, ts_t1_tf, L,
            F, b, psi);
        DV1 = V_nom - V;

        % Predicted improvement in cost
        a1 = abs( ys(N1,1) );

        % Check improvement
        if DV1/a1 > C
            % Good improvement in cost achieved. Move to the next
                iter.
            to_next_iter = true;
            break
        end
    end

    %% Flow control
    if to_next_iter
        % Good improvement in cost achieved. Move to the next iter.
        x_nom = x_nom_new;
        u_nom = u_nom_new;
        return
    end

    % No satisfactory t1 found.
    if C == 0
        % No improvement in trajectory attainable.
        warning('Unable to solve the free endpoint problem.')
        halt = true;
        return
    else
        % Retry the step-size adjustment method with less stringent
            acceptance criterion
        C = 0;
    end
end

end
```

The `totalCost` method is then used to evaluate the actual improvement in cost ΔV with (5.37), which can then be compared to the predicted change in cost $|a(\bar{x}; t_1)|$. If it is close enough, in the sense that (5.38) is satisfied, then the flag `to_next_iter` is set to true to break out of the step-size adjustment method after replacing the nominal trajectories with the new trajectories. If not, the process is repeated after changing N_1, unless N_1 has reached N_{eff}. In this case, the method is repeated after changing C to zero, which relaxes the iteration acceptance criterion (5.38). If $C = 0$ has already been tried, then the algorithm has failed to converge. The flag `halt` is used to break out of the method and to issue a warning to the user (not shown in the listing).

6.5.4 *The multipliers adjustment method*

In the multipliers adjustment method described in STEP 7, § 6.1.2, the variation of the Lagrangian multipliers δb is evaluated as

$$\delta b = -\varepsilon V_{bb}^{-1}(t_0) V_b^T(t_0), \tag{6.23}$$

where $\varepsilon = 1$ is attempted first and then adjusted until a satisfactory improvement in the endpoint error is obtained, if possible without producing large changes in the cost.

This procedure is implemented by the function `mltpAdj` shown in Listing 6.11. First, $V_{bb}(t_o)$ and $V_b(t_o)$ are retrieved from `ys`, which contains the time profiles for V_b, V_{xb} and V_{bb}[†]. If the computational trick is used, $V_{xb}(t)$ is also retrieved for later use. The nominal endpoint error $\psi(\bar{x}(t_f); t_f)$ is also computed and some flow control flags are initialized.

[†]As previously computed by `Vb_bwd`.

Listing 6.11: Multipliers adjustment method.

```
function [x_nom, u_nom, b] = mltpAdj(f, L, F, psi, ys, ts, x_nom,
    u_nom, u_hat, b, options, nvp)

% Retrieve Vb(t_0) and Vbb(t_0)
Vb0 = ys(1, 1:length(b));
Vbb0 = ys(1, end-length(b)^2+1:end);
VbbVb0 = Vbb0 \ Vb0;
% Retrieve Vxb
if options.compTrick
    Vxb = ys(:, (npsi+1):(npsi+npsi*nx));
end
% Nominal endpoint error
psi_nom = psi.fun(x_nom(end,:), ts(end));

% Initialize flags
test2_enable = true;
success = false;

% Loop until satisfactory db is found
while true
    % (Re-)initialize epsilon
    epsilon = 2;

    for n = 1:10
        % Reduce db
        epsilon = epsilon/2;
        db = - epsilon * VbbVb0;

        % New trial trajectories
        if options.compTrick
            % Obtain the new control trajectory with the
                computational trick
            [~, x_nom_new, u_nom_new] = ...
                sys_fwd(f, x_nom(1,:), ts, x_nom, u_hat, options, 'db
                    ', db, 'g', nvp.g, 'gb', nvp.gb, 'hessMinSolver',
                     nvp.hessMinSolver, 'Vx', nvp.Vx, 'Vxx', nvp.Vxx,
                     'Vxb', Vxb);
        else
            % Obtain the new control trajectory with beta1 and beta2
            [~, x_nom_new, u_nom_new] = ...
                sys_fwd(f, x_nom(1,:), ts, x_nom, u_hat, options, ...
                'db', db, 'g', nvp.g, 'gb', nvp.gb, ...
                'beta1', nvp.beta1, 'beta2', nvp.beta2, 'db', db);
        end

        % Test 1/2
        if full( norm( psi.fun(x_nom_new(end,:), ts(end) ) ) ) > norm
            ( psi_nom )
            % Test 1 failed; reject db
```

```
            continue
        end

        if test2_enable
            % Test 2/2

            % Actual improvement in cost
            V_nom = totalCost(x_nom, u_nom, ts, L, F, b, psi);
            V = totalCost(x_nom_new, u_nom_new, ts, L, F, b + db, psi
                );
            DV = V - V_nom;

            % Predicted improvement in cost
            DVtilde = nvp.a(1) - (epsilon - 0.5 * epsilon^2) * Vb0.'
                * VbbVb0;

            gamma1 = 0.8;
            gamma2 = 1.2;
            if DV/DVtilde > gamma1 && DV/DVtilde < gamma2
                % Both tests passed; Accept db
                success = true;
                break
            else
                % Test 2 failed; reject db
                continue
            end
        else
            % Test 1 passed and test 2 disabled; accept db
            success = true;
            break
        end
    end

    if test2_enable && ~success
        % Try again with test 2 disabled
        test2_enable = false;
        continue
    elseif success
        x_nom = x_nom_new;
        u_nom = u_nom_new;
        b = b + db;
        break
    else
        warning('Multipliers adjustment method failed.')
        break
    end
end

end
```

The rest of the function is composed by two loops. The outer loop runs at most twice and is used to repeat the whole multipliers adjustment method with relaxed acceptance criteria, if needed. In the inner loop, ε is halved[†] to produce a new δb. The state equations are integrated using either (5.41)

$$u = \hat{u} + \beta_1 \delta x + \beta_2 \delta b$$

or (5.72)

$$u = \hat{u} + \beta_1 \delta x + \beta_2 \delta b,$$

[†] ε is initialized to 2 so that the first iteration sets $\varepsilon = 1$.

if the computational trick is enabled.

Two tests are then used to determine whether to accept or reject the change δb and the corresponding new trajectories. First, the endpoint error is checked, and if there is no improvement ((6.14) is not satisfied) the iterate is rejected. Then, the change in cost is also checked and if the actual change in cost is not close enough to the predicted change in cost ((6.17) is not satisfied), the iterate is rejected. If after ten of these iterations a value of δb that satisfies both tests is not found, the whole procedure is repeated without checking the test on the cost, by setting the flag `test2_enable` to false.

6.6 Modeling for DDP

In this section, we discuss some aspects of the algorithm that must be kept in mind when defining the optimal control problem. Some of these are related to the theoretical foundations of the algorithm, while some others are related to practical aspects of its numerical implementation.

The first requirement is that the Hamiltonian of the problem must be well defined for any value of the controls that lies within its box constraints; otherwise, the NLP solver will fail. Suppose for example that we define a typical energy management strategy design problem where we attempt to minimize the fuel consumption of a parallel hybrid as the one presented in § 4, and that we define a torque-split ratio as our control variable u. With this setup, there will certainly be values of u for which the corresponding engine torque is higher that its limit torque; physically speaking, fuel consumption is undefined for these values and hence the Hamiltonian. Thus, the algorithm will certainly fail.

Thus, for this and similar problems, the best course of action is to redefine whatever model is being used to evaluate fuel consumption as a function of torque so that it allows extrapolation for any value of the torque. This does not have an impact on the optimal solution as we would obviously set a constraint on $g(u,;t)$ so that the engine torque never exceeds its limit.

A theoretical aspect of any second-order differential dynamic programming algorithm to be considered is that it relies on the inversion of the term

$$(H_{uu} + \mu \hat{g}_{uu}), \tag{6.24}$$

which must therefore be locally nonzero (for scalar controls) or nonsingular (for non-scalar controls), when evaluated at $(\bar{x}, \hat{u}, V_x; t)$. This may be the source of some numerical issues; to illustrate this, let us focus on the case where no constraints are active. In this case, we must ensure invertibility of $H_{uu}(\bar{x}, \hat{u}, V_x; t)$. Since $\hat{u}$ minimizes the Hamiltonian, we are requiring H_{uu} to be positive definite; note that this does not mean that the Hamiltonian must be convex everywhere. Rather, we simply require that the Hamiltonian $H(\bar{x}, u, V_x; t)$ is continuous[†] at its global minimum with respect to u.

[†] More rigorously, we required that the Hamiltonian is twice continuously differentiable.

While this is not a very restrictive requirement, there may still be some numerical issues if H_{uu} is close to singular. In practice, the algorithm may struggle if, for a certain nominal state trajectory, the Hamiltonian is almost linear at its minimum, i.e. if $H_{uu}(\bar{x}, \hat{u}, V_x; t)$ is close to zero.

Hence, the differential dynamic programming developed for this work, which is of second-order, is suitable for highly nonlinear problems but it may struggle for problems where the Hamiltonian is almost linear. Note that, for these problems, there exist first-order variants of the differential dynamic programming algorithm [30].

6.7 TESTING THE ALGORITHM

In order to test the software implementation of the differential dynamic programming algorithm, it was tested on a toy problem taken from [50]. A two-state system characterized by the state dynamics

$$\dot{x}_1 = x_2 \tag{6.25}$$

$$\dot{x}_2 = u, \tag{6.26}$$

must be driven from the initial state $x_o = \left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right)$ at time $t_o = 0$ to the final state $x_f = \left(\begin{smallmatrix} 0 \\ -1 \end{smallmatrix}\right)$ at time $t_f = 1$, while minimizing the cost

$$V = \frac{1}{2}\int_{t_o}^{t_f} u^2 dt \tag{6.27}$$

and subject to the state constraint

$$S(x;t) = x_1 - \frac{1}{9} \leqslant 0. \tag{6.28}$$

[†]I.e. its lowest-order time derivative for which it is explicitly a function of u is of second order.

Upon inspection, we see that $S(x;t)$ is of second-order[†], and it can therefore be transformed into a mixed state-control inequality constraint by replacing it with the constraining hyperplane

$$\frac{d^2S}{dt^2} + \alpha_1\frac{dS}{dt} + \alpha_2 S = 0, \tag{6.29}$$

which gives the constraint

$$g(x,u;t) = u + \alpha_1 x_2 + \alpha_2\left(x_1 - \frac{1}{9}\right) \geqslant 0. \tag{6.30}$$

The problem was solved with our software implementation using the same parameters reported in [50]: the cost and endpoint tolerance were set to $\eta_1 = \eta_2 = 0.002$ and the steepness factors of the hyperplane were set to $\alpha_1 = 45$ and $\alpha_1 = 500$. The nominal control was initialized to a constant value $\bar{u} = 5$ and the Lagrange multipliers were initially set to $\bar{b} = \left(\begin{smallmatrix} -5 \\ 5 \end{smallmatrix}\right)$. An additional balancing term $c\psi^2(\sigma(t_f))$, with c equal to 20, was added to the terminal cost[‡]; additional tests on this same problem proved this not to be crucial but helpful in slightly reducing the number of iterations required for convergence.

[‡]As we mentioned in § 5.4.1, this generally improves the convergence of the algorithm with fixed-endpoint problems.

As shown in Figures 6.2 and 6.3, the algorithm converges taking four minor iterations followed by six major iterations, with no minor iterations in between. Specifically, the algorithm uses the four minor iterations to rapidly reduce the augmented cost[§] by reducing the endpoint error. Then, the following major iterations update the Lagrange multipliers b to hit the terminal constraints up to the required tolerance. Since these iterations do not cause large deviations in the state trajectories, the algorithm does not need to use any additional minor iterations in between as δx remains small enough that

[§]I.e. the sum of the total cost $\int L dt$, terminal cost F and endpoint error term $b\psi$.

$$u = \hat{u} + \beta_1\delta x + \beta_2\delta b. \tag{6.31}$$

satisfies the convergence criterion $|\alpha(x_o;t_o)| < \eta_1$.

The optimal solution that was thus obtained has a total cost of 4.007 with a terminal state of $x(t_f) = \left(\begin{smallmatrix} -0.0007 \\ -0.9995 \end{smallmatrix}\right)$, and the optimal Lagrange multipliers were found to be $b = \left(\begin{smallmatrix} -17.7 \\ 6.00 \end{smallmatrix}\right)$.

These results match well with those reported by Mårtensson [50], except from the fact that one more major iteration was required. This is probably attributable to the numerical errors resulting from the fact that our implementation makes use

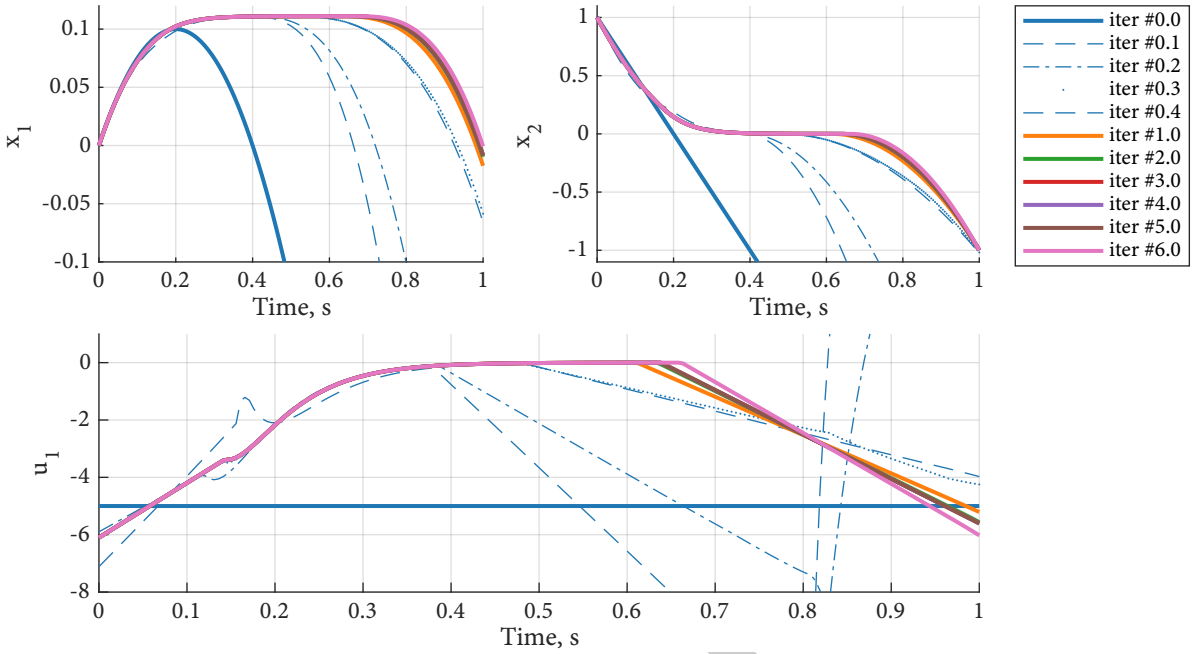


Figure 6.2: State and control trajectories at each iteration.

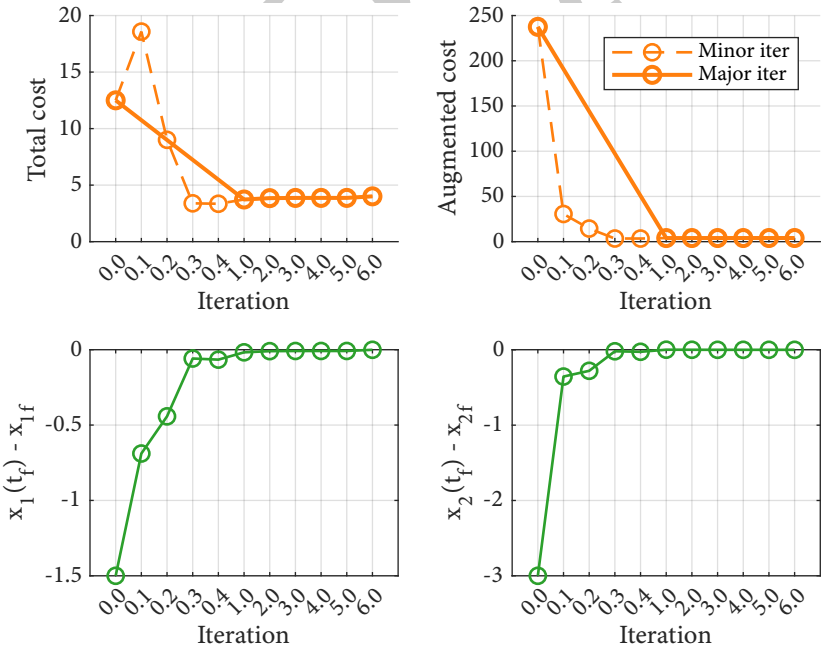


Figure 6.3: Cost, augmented cost and endpoint error per iteration.

of automatic differentiation and uses an NLP solver to obtain $\hat{u}$, whereas Mårtensson provides analytic derivatives to the algorithm as well as an analytic expression for $\hat{u}$.

This toy example thus shows that the software implementation developed for this thesis maintains a good performance despite the replacement of analytical derivatives with automatic differentiation and of an analytical minimization of the Hamiltonian (5.43) with an NLP solver. These two unique [†] features of the software implementation developed for this work are very important as they enable the solution of problems involving complex models such as those arising in energy management strategy design.

[†]To the best of the author's knowledge.

7 DDP application: series hybrid

Theories stand or fall, ultimately, upon numbers.

Richard Bellman [7]

In this chapter, we consider a series hybrid powertrain as depicted in Figure 7.1. We are going to tackle the classical issue of controlling the generator set in order to minimize the fuel consumption as the vehicle completes a driving cycle, while enforcing charge-sustaining operation.

An electric motor drives the vehicle's primary axle, drawing power from the battery, the generator set or both. The generator set, composed of a thermal engine which powers an electrical generator, can also charge the battery while powering the motor. The main vehicle parameters are reported in Table 7.1.

Component	Parameter	Value
Vehicle	Mass	1200 kg
	First coast-down coefficient	76.11 N
	Second coast-down coefficient	2.957 N/(ms)
	Third coast-down coefficient	0.3664 N/(ms)^2
	Tyre radius	0.3 m
Gen-set	Maximum power	55 kW
Motor	Rated power	85 kW
	Maximum torque	300 Nm
Battery	Type	LiFePO4
	Nominal capacity	21.6 Ah
	Nominal voltage	173.2 V
	Nominal energy	3.74 kWh

Table 7.1: Main vehicle data.

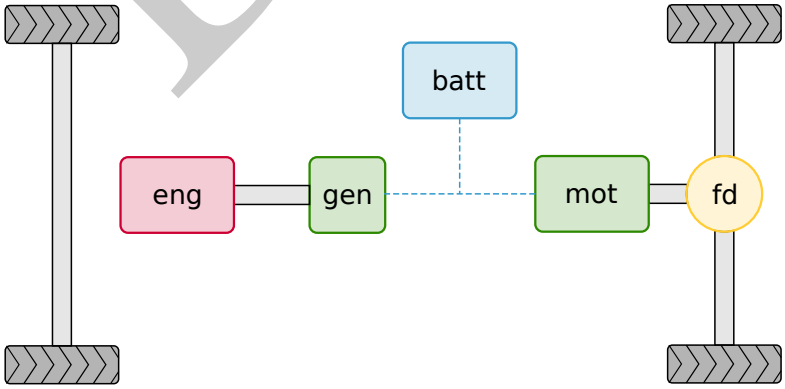


Figure 7.1: Series hybrid architecture.

7.1 Powertrain model

7.1.1 State dynamics

Having defined a driving cycle, the motor torque and speed can be evaluated using the same backward facing approach as we described in other parts of this thesis.

The motor torque and speed were evaluated as:

$$T_{mot} = \frac{F_{veh}(v_{veh})r_{wh}}{\tau_{fd}}, \qquad \omega_{mot} = \frac{v_{veh}}{r_{wh}}. \tag{7.1}$$

And the motor electrical power was then evaluated using an efficiency map $\eta_{mot}(\omega_{mot}, T_{mot})$

$$\begin{cases} P_{mot} = \frac{1}{\eta_{mot}}\omega_{mot}T_{mot}, & \text{if } \omega_{mot}T_{mot} \geqslant 0, \\ P_{mot} = \eta_{mot}\omega_{mot}T_{mot}. & \text{if } \omega_{mot}T_{mot} < 0. \end{cases} \tag{7.2}$$

This motor power profile $P_{mot}(t)$ can be treated as an exogenous input to our optimization problem.

Our control variable is the electrical power generated by the gen-set. In practice, for improved flexibility of the software, a normalized gen-set power was defined as

$$\tau_{gen} = \frac{P_{gen}}{P_{gen,max}}. \tag{7.3}$$

Theoretically, we could directly control both the speed and torque of the engine; however, since they can be set independently of the driving conditions, there is no point in controlling both in our optimal control problem.

We can identify the electrical power generated by the gen-set as our control variable and set the engine speed and torque[†] to minimize fuel consumption. This allows us to characterize the fuel consumption as a function of the gen-set electrical power only.

[†] and consequently the e-machine speed and torque.

Specifically, the engine's fuel consumption map and the generator's efficiency map were used to identify the minimum fuel operating points for various values of the generated electrical power and the resulting fuel consumption values used to fit a quadratic model $\dot{m}_f(\tau_{gen})$, as described in § 7.3[‡]. This function, which we will refer to as the gen-set characteristic, defines our running cost:

[‡] This approach was inspired by [72].

$$L(x, u, t) = \dot{m}_f(\tau_{gen}) \tag{7.4}$$

The system's state is characterized by the battery's state of charge σ and temperature T_b. The state of charge must be considered in order to formulate the charge-sustaining constraint as the terminal state constraint $\sigma(t_f) = \sigma(t_0) = \sigma_0$.

The battery temperature can be used to improve the accuracy of the equivalent circuit model by considering battery-dependent open circuit voltage and equivalent resistance characteristics. In this work, for the sake of simplicity, the equivalent resistance $R_{eq}(T_b)$ was modeled as being purely temperature dependent and the open circuit voltage $v_{oc}(\sigma)$ as being dependent on the state of charge only. However, the differential dynamic programming algorithm developed for this application could handle the characteristics being dependent on both σ and T_b with no modification.

The SOC dynamics were determined by according to a typical equivalent circuit model:

$$\dot{\sigma} = -\frac{i_b}{C_b}, \tag{7.5}$$

where the battery current i_b is evaluated as

$$i_b = \frac{v_{oc} - \sqrt{v_{oc}^2 - 4R_{eq}P_b}}{2R_{eq}}, \tag{7.6}$$

and the battery power P_b as

$$P_b = P_{mot} - \tau_{gen}P_{gen,max}. \tag{7.7}$$

The battery thermal dynamics were characterized by a lumped thermal capacity $C_{th,b}$. Aside from the heat generation due to the Joule losses, the only mode of heat transfer considered was convective heat transfer with the surrounding environment with a constant heat transfer coefficient h_{conv} and a constant environment temperature T_{env}.

$$\dot{T}_b = \frac{1}{C_{th,b}}\left(h_{conv}\left(T_b - T_{env}\right) + R_{eq}i_b^2\right), \tag{7.8}$$

7.1.2 Constraints and initial conditions

The normalized gen-set power was obviously constrained to

$$0 \leqslant \tau_{gen} \leqslant 1, \tag{7.9}$$

and the battery current was constrained within minimum and maximum limits

$$i_{b,min} \leqslant i_b \leqslant i_{b,max}. \tag{7.10}$$

Finally, the battery state of charge was constrained within an upper and lower bound of $\sigma_{lb} = 0.4$ and $\sigma_{ub} = 0.8$

$$\sigma_{lb} \leqslant \sigma \leqslant \sigma_{ub}. \tag{7.11}$$

In order to implement the differential dynamic programming algorithm, we must formulate all constraints (except for the terminal state constraints) to be either in the form of box constraints for the control variable or in the form of inequality constraints $g(x, u, t) \leqslant 0$.

The constraints on τ_{gen} are therefore readily accommodated. The battery current constraints must be reformulated as two mixed state-control variable[†] inequality constraints:

[†]because i_b is a function of σ and τ_{gen} (as well as time).

$$i_b - i_{b,max} \leqslant 0, \tag{7.12}$$
$$-i_b + i_{b,min} \leqslant 0. \tag{7.13}$$

The state of charge constraints are two state variable inequality constraints:

$$S_1(x;t) = \sigma - \sigma_{ub} \leqslant 0, \tag{7.14}$$
$$S_2(x;t) = -\sigma + \sigma_{lb} \leqslant 0, \tag{7.15}$$

and they must be transformed into mixed state-control variable using the constraining hyperplane technique discussed in § 5.5, so that we can handle them with the algorithm developed in § 6.

These constraints are of first order, in the sense that the first-order time derivative is the first that explicitly contains the control variables. In fact, the corresponding constraining hyperplanes are:

$$\frac{dS_1}{dt} + a_1(\sigma - \sigma_{ub}) = 0, \tag{7.16}$$
$$\frac{dS_2}{dt} + a_2(-\sigma + \sigma_{lb}) = 0, \tag{7.17}$$

and these allow us to define the transformed inequality constraints:

$$\dot{\sigma} + a_1(\sigma - \sigma_{ub}) \leqslant 0, \tag{7.18}$$

$$\dot{\sigma} + a_2(-\sigma + \sigma_{lb}) \leqslant 0, \tag{7.19}$$

where $\dot{\sigma}$ is evaluated as in (7.5). The parameters a_1 and a_2 [†], as discussed in § 5.5, must be positive. Also, they should be large enough that they actually enforce the constraint; but they should not be set to high or it may become too hard for the algorithm to generate a new acceptable trajectory in the first iterations [‡]. For our application, we used $a_1 = a_2 = 100$.

Finally, for all simulations, the initial state of charge σ_0 was set to 0.6 and the terminal state of charge $\sigma(t_f)$ was constrained to be equal to 0.6 as well. For those simulations where the battery thermal dynamics were also modeled, the initial battery temperature was set to be equal to the environment temperature T_{env}. No terminal constraint was set on the battery temperature.

[†] i.e. the steepness of the constraining hyperplanes.

[‡] According to [52], it may even be desirable to start with a moderate steepness of the hyperplanes and increase them near convergence

7.2 Numerical experiments

The differential dynamic programming algorithm was used to run several experiments in order to test its strengths and weaknesses. In this section, we are going to address many discussion points.

First, we will evaluate the algorithm with a scalar state, by neglecting the battery temperature and assuming a constant equivalent resistance, and we will compare the results with the solution obtained with dynamic programming using DynaProg.

This will also provide us with a starting point to discuss the algorithm's performance and convergence as we vary the initial guesses for the nominal control trajectory $\bar{u}$ and the Lagrange multipliers $\bar{b}$.

We will then evaluate the algorithm with two states, considering the battery's thermal dynamics as presented in § 7.1.1, and we will once again compare the results with the solution obtained with DynaProg.

Finally, we will highlight a potential challenge that may arise in enforcing the state constraints, by testing the algorithm on a powertrain with a reduced battery size.

7.2.1 Scalar state

The first experiment is a simple run to evaluate the optimality of the solution with a reference solution obtained with dynamic programming.

The series hybrid model with scalar state, neglecting the battery's thermal dynamics, was implemented using the DDP algorithm. The running cost was set to the $\dot{m}_f(\tau_{gen} P_{gen,max})$, in kg/s. The nominal control trajectory $\bar{u}$ was set to a constant value $\tau_{gen} = 0.06$. This of course is a very naive approach, and many smarter initialization strategies could be adopted by exploiting engineering understanding of the control problem. The nominal Lagrange multiplier was set to -1 kg[§].

Furthermore, a terminal cost $c\psi^2(\sigma(t_f))$, with c equal to 1 kg, was set as a balancing term. As we discussed in § 5.4.1, introducing such a balancing term to the terminal cost may help improving the convergence of the fixed-endpoint algorithm. In practice, this was found to be beneficial for the problem in this chapter. The convergence parameters for the augmented cost η_1 and for the endpoint error η_2 were set to 0.001 kg and 0.001.

With these settings, the algorithm converged in three iterations: two minor iterations iterations followed by one major iteration[‖] were sufficient. As can be seen in Figure 7.2, the algorithm is able to produce a new, improved control trajectory

[§] Since σ is dimensionless, so is ψ. Hence, b must have the same unit of measurement of the running cost L.

[‖] Recall from § 6.1.1 that we denote *minor* iterations the iterations of the algorithm's inner layer, which improve the trajectory for a fixed $\bar{b}$, and we denote *major* iterations those of the outer layer which update $\bar{b}$ while keeping the cost increase to a minimum.

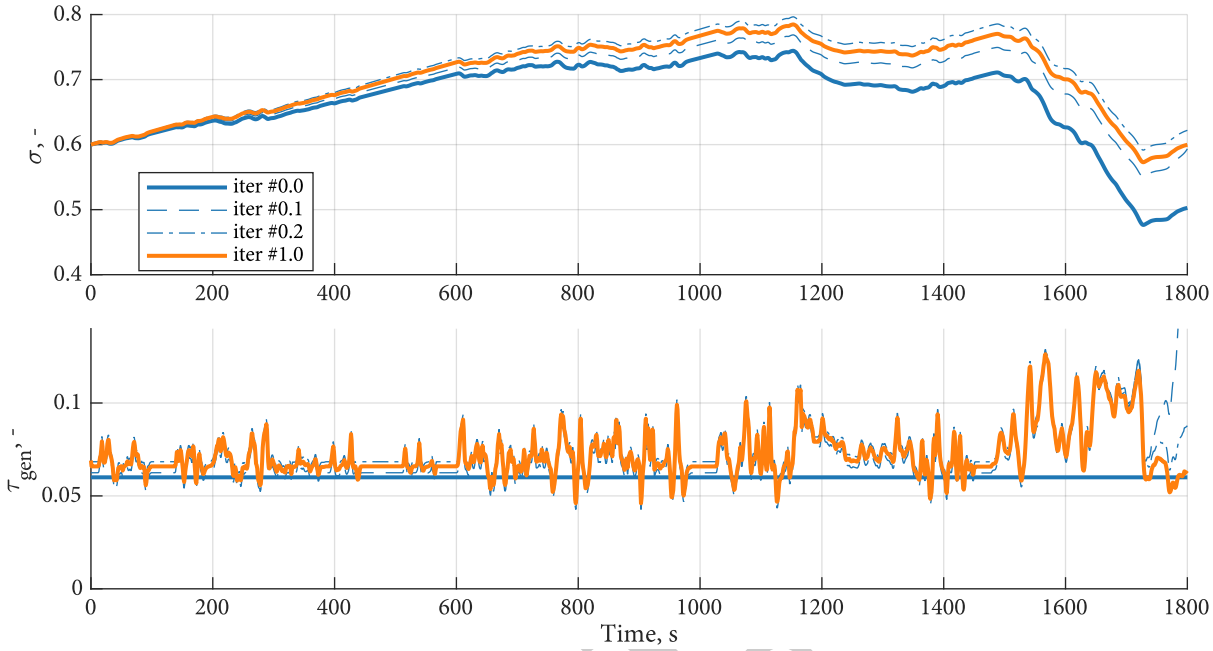


Figure 7.2: State and control trajectories as the algorithm runs. Iteration 0.0 corresponds to the nominal trajectories, iterations 0.1 and 0.2 correspond to two minor iterations and iteration 1.0 corresponds to a major iteration (an update of the Lagrange multiplier).

with large variations with respect to the nominal one with the very first minor iteration. A second iteration further reduces the augmented cost, but it induces a slight overshoot of the terminal SOC constraints. Finally, a single major iteration updates the Lagrange multiplier to -0.51 kg. The resulting fuel consumption was 575.5 g, with a terminal SOC of 0.5997.

Figure 7.3 shows the total cost, the augmented cost and endpoint error[†] per iteration.

The total cost for this problem is the total fuel consumption. The augmented cost is the sum of the total fuel consumption, the terminal cost $F(\sigma(t_f))$[‡] and the endpoint error $b\psi(\sigma(t_f))$. As expected, the augmented cost is always reduced between minor iterations. It can, on the other hand, increase as a result of an update of the Lagrange multipliers, which is exactly what happens.

Looking at the total cost alone, we see that it grows during the first two minor iterations as the algorithm tries to improve the endpoint error. This is a result of the initial nominal trajectory producing a terminal state of charge lower than the terminal constraint. Since iteration 0.2 produces a terminal SOC higher than 0.6, it is no surprise that the subsequent Lagrange multipliers update results in a slight reduction in total cost.

Let us now compare the performance of the algorithm with the solution obtained with dynamic programming. The same model was implemented in DynaProg (§ 3.2), using a uniform SOC grid with 801 elements ranging from 0.4 to 0.8 and a uniform τ_{gen} grid with 81 elements ranging from 0 to 1. To ensure a fair comparison, the cost of the dynamic programming solution was obtained by integrating the state dynamics with the control sequence obtained with DynaProg using the same fourth-order Runge-Kutta integration routine that is used by the differential dynamic programming algorithm.

The resulting fuel consumption was 575.8 g with a terminal SOC of 0.6001, which matches well with the results of the differential dynamic programming algorithm. As can be seen from Figure 7.4, the state and control trajectories are

[†]I.e. the deviation of the terminal state from its constraint.

[‡]Which in this problem is only composed by the balancing term $c\psi^2(\sigma(t_f))$.

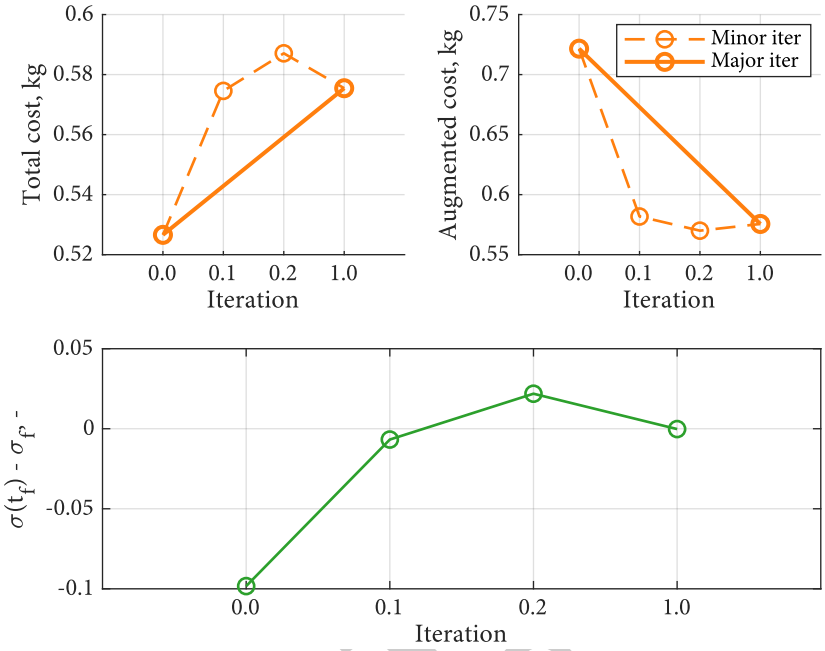


Figure 7.3: Total cost, augmented cost and endpoint error per iteration.

also very similar, the differences between them being attributable to the quantization required by the dynamic programming algorithm. This also highlights a major advantage of the differential dynamic programming in that, treating τ_{gen} as a continuous variable, it generates a smoother trajectory.

7.2.2 *Downsized battery pack*

In this section, we will evaluate the algorithm's performance with a reduced battery size with halved capacity, bringing its nominal capacity to 10.8 Ah and its nominal energy to 1.87 kWh. The reason is that in this case the optimal SOC trajectory hits the upper bound (more than once). Hence, it is a good test for the algorithm's performance in dealing with the state constraint.

The same settings for the initial Lagrange multiplier and convergence parameters as in § 7.2.1 were used. The nominal control $\bar{u}$ had to be changed in order to ensure that it does not violate the state constraints, as this is a requirement of the algorithm. Hence, a simple piece-wise constant function was defined as:

$$\bar{\tau}_{gen}(t) = \begin{cases} 0.05 & \text{if } t < 1545 \text{ s}, \\ 0.17 & \text{if } t \geqslant 1545 \text{ s}. \end{cases} \tag{7.20}$$

This trajectory was set based simply based on the (very inaccurate) idea that the extra high phase would require about three times as much power as the previous part of the mission. Much better initialization strategies could be conceived with little effort. However, we deliberately avoided to do so in order to test the robustness of the algorithm in the case that the initial guess for $\bar{u}$ is not good.

Compared to the previous case, the algorithm required more iterations to converge. In particular, the first major iteration required 5 minor iterations rather than just two. This is expected as it is harder to adjust the control trajectory when state constraints are involved. After this, two major iterations (with no minor iterations in between) were enough for the algorithm to converge, producing a fuel consumption of 580.4 g and a final SOC of 0.5998.

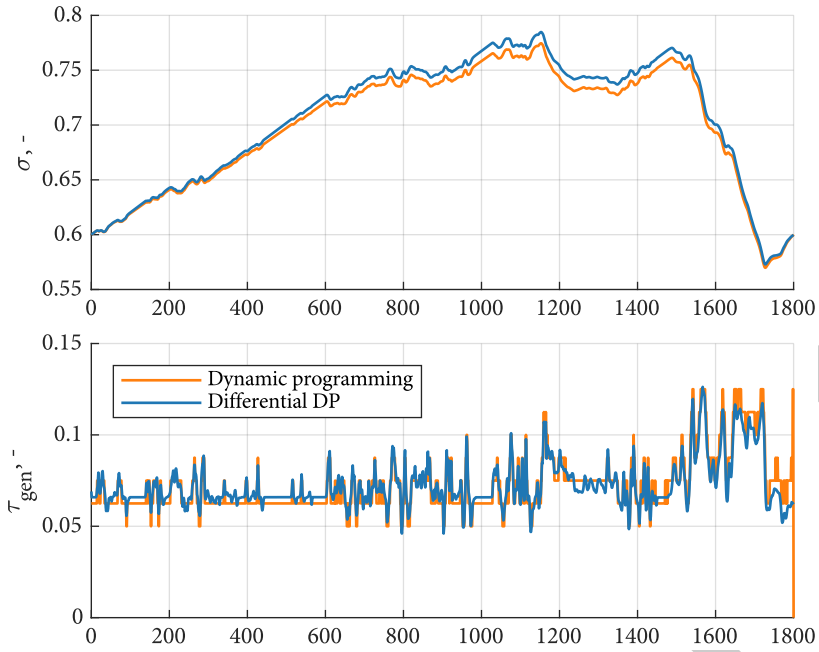


Figure 7.4: Comparison between the optimal solution obtained with differential dynamic programming and DynaProg.

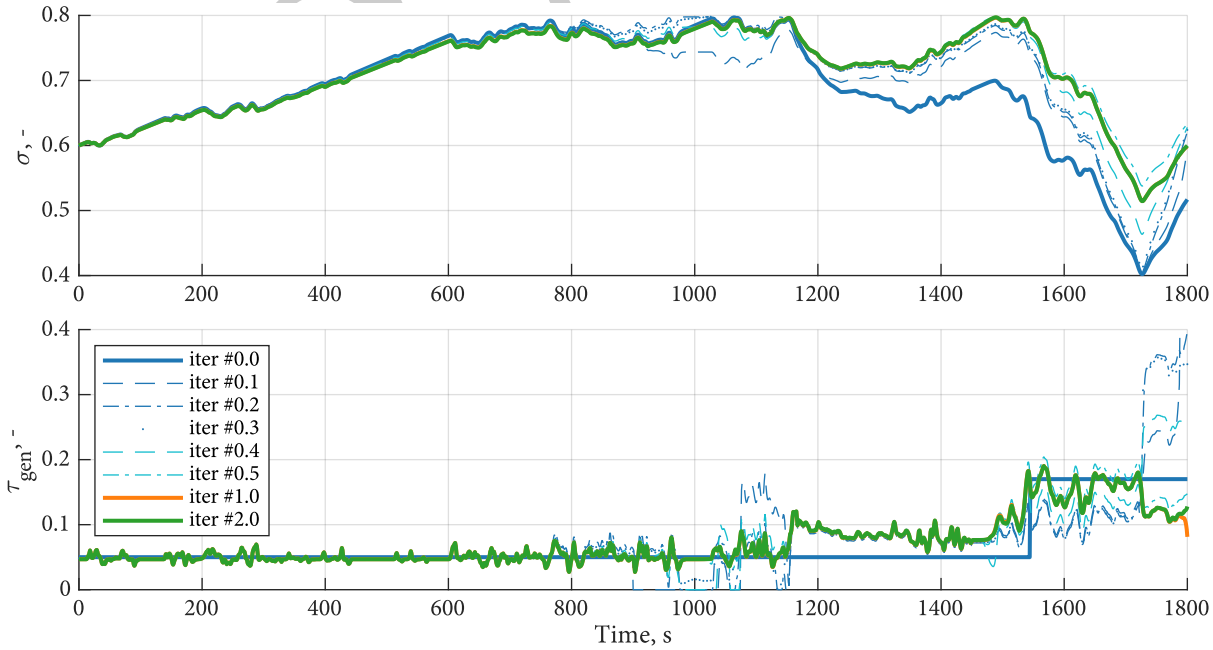


Figure 7.5: State and control trajectories as the algorithm runs, with the reduced battery. For this problem, the optimal SOC trajectory hits the upper bound several times.

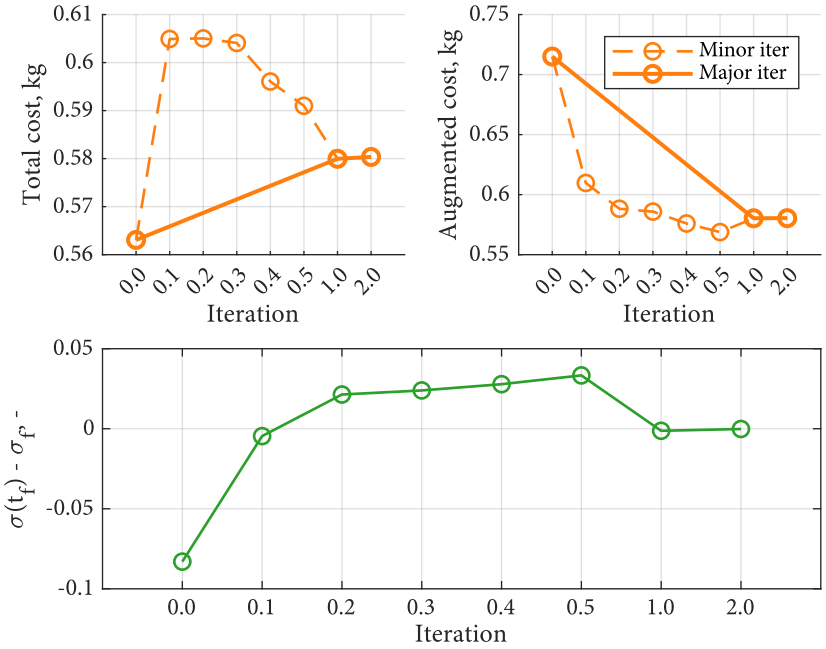


Figure 7.6: Total cost, augmented cost and endpoint error per iteration, with the reduced battery.

Figure 7.6 shows the total cost, the augmented cost and endpoint error per iteration. Similarly to the problem in § 7.2.1, the nominal SOC trajectory, generated by the first guess for the control trajectory, leads to a strong violation of the terminal SOC. Consequently, the very first iterations are used by the algorithm to reduce the associated cost, even if that produces an increase in fuel consumption. This brings the nominal state trajectory to hit the upper SOC bound around at about 1000 s.

Then, the algorithm reduces the cost as much as possible while avoiding to violate the upper SOC bound, and this requires a few more minor iterations. Finally, two major iterations are needed to update the Lagrange multipliers so as to satisfy the terminal state constraint with the required accuracy [†].

[†] That is, the parameter we named η_2.

Once again, the solution was compared with one obtained using DynaProg. The settings were the same as in § 7.2.1. The resulting fuel consumption of 580.3 g and a final SOC of 0.6002. Hence, this numerical experiment confirms that the differential dynamic programming algorithm is capable of obtaining the optimal solution with high accuracy even when the optimal state trajectory hits some state constraints. This is an important feature in the context of energy management strategy design as it often happens that the optimal SOC trajectory hits the battery's lower and/or upper SOC bounds, especially with small-sized battery packs.

Modeling issues

In a first attempt to solve the problem presented in § 7.2.2, the algorithm failed to converge. An important feature of the algorithm is that the base equations for the constrained algorithm presented in § 6 were developed under the assumption that the nominal trajectories do not violate the constraints[‡]. In practice, the algorithm proves to be tolerant to small violations.

[‡] although they can hit them.

Large violations on the other hand may cause the integration of the base equations to produce meaningless time profiles for α, V_x and V_{xx} as well as V_{β_1}. When this happens, the algorithm cannot improve the nominal trajectories. For this

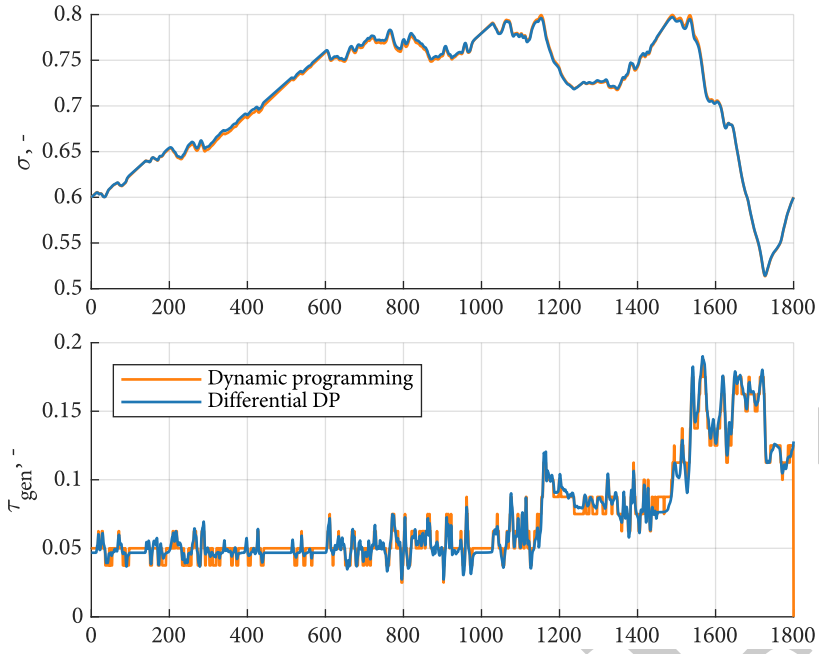


Figure 7.7: Comparison between the optimal solution obtained with differential dynamic programming and DynaProg, with the reduced battery.

reason, when generating new nominal trajectories if the new control trajectory $u = \bar{u} + \beta_1 \delta x$ induces violation of a constraint $g(x, u; t)$, it is replaced by the control that satisfies the condition $g(x, u; t) = 0$. It is an important requirement for the convergence of the algorithm that such a control exists.

In the problem presented in this chapter this means that, when generating new trajectories, if the new control trajectory $u = \bar{u} + \beta_1 \delta x$ violates the upper SOC threshold σ_{ub}, it is replaced by the control that satisfies $\sigma = \sigma_{ub}$. In our formulation of the problem, this control may not always exist. In fact, in braking phases, $\dot{\sigma}$ is positive regardless of τ_{gen} because we assume that the vehicle performs regenerative braking.

A potential solution to this issue would be to reformulate the control problem so that we are also able to control the amount of braking torque to be absorbed by the electric motor. This would introduce additional complexity to the problem with no benefit on the optimal solution, as it is safe to assume that the optimal control trajectories would ultimately involve doing as much regenerative braking as allowed by the electric motor.

A workaround, which we used in our simulation, is to modify the state dynamics instead so that $\dot{\sigma}$ simply vanishes when the state of charge is above σ_{ub}. The fact that this does not reflect physical behavior is not an issue as neither the optimal trajectories nor any other nominal trajectory generated by the algorithm is allowed to violate the SOC constraints.

This trick however requires some mathematical judgment. Because the base equations rely on first and second order derivatives of the state equations, it must be at least twice differentiable. Therefore, modifying the SOC dynamics (7.5) to

$$\dot{\sigma} = \begin{cases} -\dfrac{i_b}{C_b} & \text{if } \sigma \leqslant \sigma_{ub}, \\ 0 & \text{if } \sigma > \sigma_{ub}, \end{cases} \tag{7.21}$$

would cause all sorts of numerical issues as it is non-differentiable at $\sigma = \sigma_{ub}$.

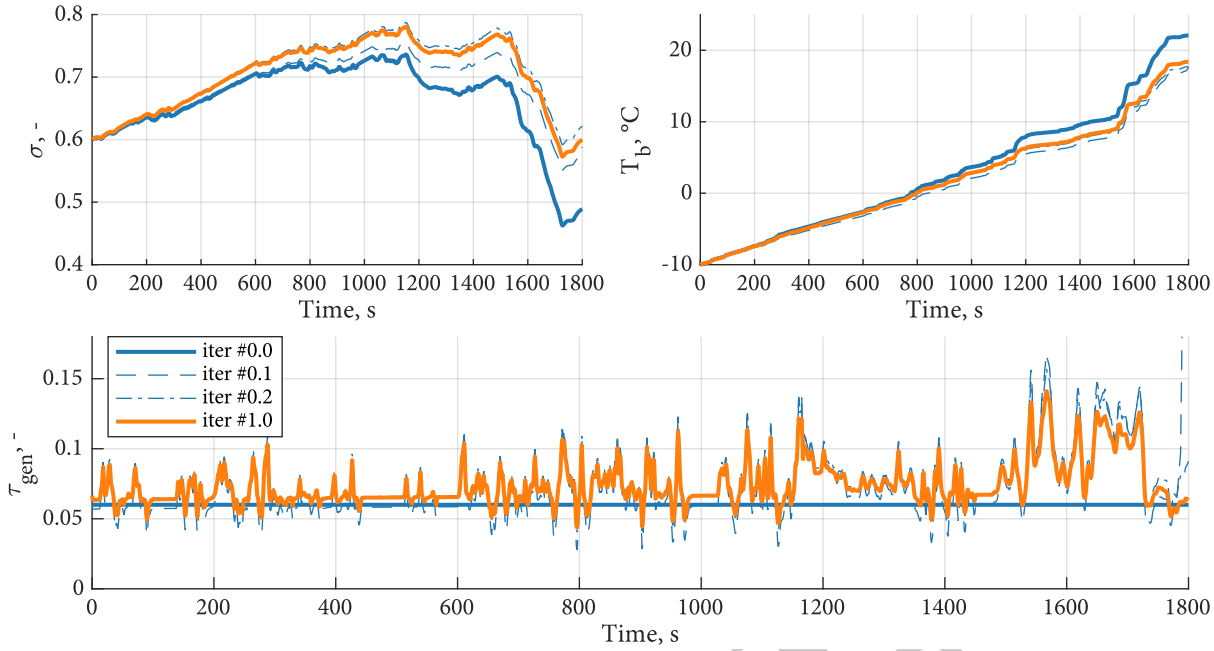


Figure 7.8: State and control trajectories as the algorithm runs, with modeled battery temperature dynamics.

For our simulation, we adopted a sigmoid function, specifically the logistic function:

$$\Lambda(\sigma) = \frac{1}{1 + e^{-10^2(\sigma_{ub} - \sigma)}}, \tag{7.22}$$

and redefined the SOC dynamics as:

$$\dot{\sigma} = -\frac{i_b}{C_b}\Lambda(\sigma). \tag{7.23}$$

7.2.3 *Temperature-dependent battery pack*

In this section, we evaluate the performance with the full two-state model, the battery temperature is considered as a second state variable whose dynamics are described by (7.8). The reason for this numerical experiment is to test the capability of the differential dynamic programming algorithm to endure the curse of dimensionality. Specifically, we are going to compare the simulation times of the algorithm with that required by DynaProg to obtain a solution with the same accuracy.

First, let's observe the behavior of the differential dynamic programming algorithm alone by running a simulation with the same settings as in § 7.2.1.

From Figures 7.8 and 7.9, we observe that the performance of the algorithm in terms of number of iterations is basically the same as the scalar case. This is likely due to the fact that the slow dynamics of the battery temperature do not make the solution of the problem more challenging to a significant extent. The fuel consumption was 582.0 g with a final SOC of 0.5997.

The same problem was solved using DynaProg. As in § 7.2.1, uniform SOC grid with 801 elements ranging from 0.4 to 0.8 and a uniform τ_{gen} grid with 81 elements ranging from 0 to 1 were used. In addition, a uniform grid with 301 elements ranging from -10 °C to 20 °C was used for the battery temperature. This discretization level proved necessary to obtain the optimal fuel consumption; tests conducted with a coarser grid produced a significantly higher cost with respect

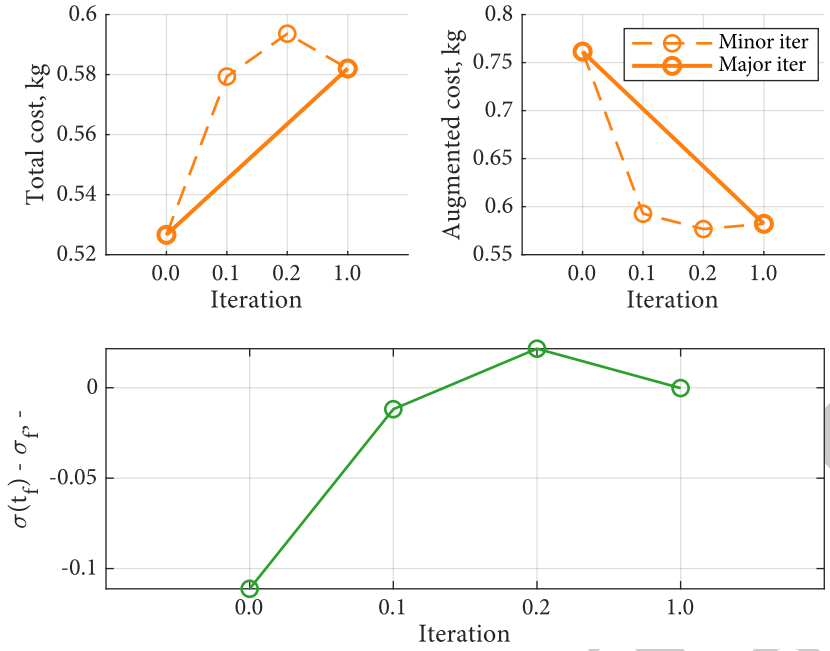


Figure 7.9: Total cost, augmented cost and endpoint error per iteration, with modeled battery temperature dynamics.

to differential dynamic programming. With these settings, the resulting fuel consumption was 582.3 g, similar to the differential dynamic programming solution.

We now turn our attention to simulation time, which is the focal point of this experiment. While the dynamic programming solution required 1226 seconds, or 20 minutes and 26 seconds, of simulation time, the differential dynamic programming algorithm converged in 214 seconds, or 3 minutes and 34 seconds, about five times less. This stands in stark contrast with the two algorithm's performance for the scalar case § 7.2.1, where simulation times were 3 seconds for dynamic programming and 198 seconds, or 3 minutes and 18 seconds, for differential dynamic programming.

These results are particularly remarkable when considering that, while DynaProg was specifically optimized for speed, the differential dynamic programming algorithm developed for this thesis was designed as a demonstrator with no particular effort towards code optimization.

In conclusion, the introduction of an additional state variable did not change significantly the computational effort on the differential dynamic programming, while it obviously increased the dynamic programming algorithm's simulation time according to the well-known curse of dimensionality. It is reasonable to expect that adding even more state variables to the problem would make the differential dynamic programming algorithm even more superior in terms of computational time with respect to dynamic programming.

7.3 Creating the gen-set characteristic

This section illustrates how the gen-set characteristic $\dot{m}_f(\tau_{gen})$ was built starting from the engine's fuel consumption map and the generator's efficiency map.

First, a set of discrete values for the gen-set power P_{gen} was generated. Then, for each of these points, the following minimization problem was set up:

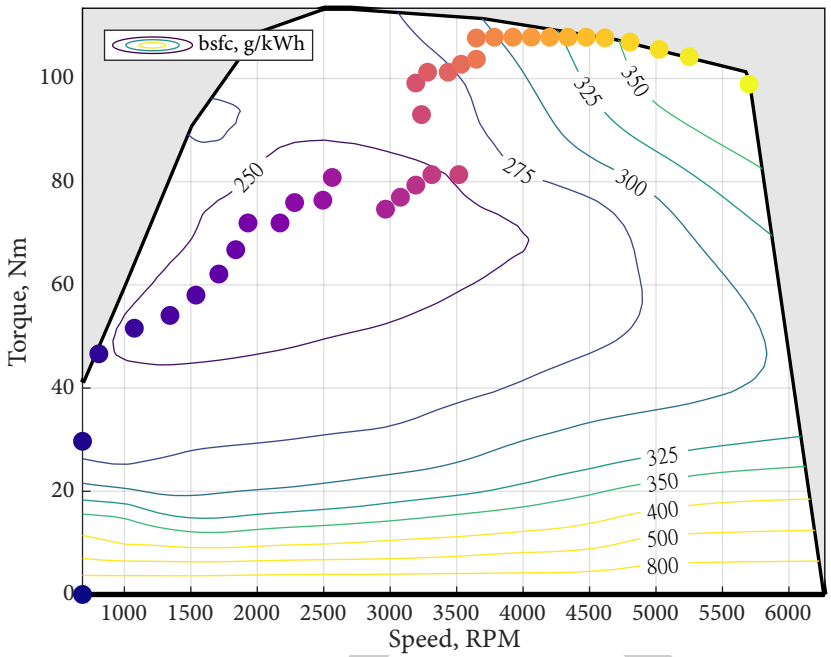


Figure 7.10: Engine operating points. Increasing values of gen-set power are marked by brighter colors.

$$\begin{aligned} \underset{\omega_{eng},T_{eng}}{\text{minimize}} \quad & \dot{m}_f(\omega_{eng}, T_{eng}) \\ \text{subject to} \quad & \omega_{eng,idle} \leqslant \omega_{eng} \leqslant \omega_{eng,max}, \\ & 0 \leqslant T_{eng} \leqslant T_{eng,max}(\omega_{eng}), \\ & P_{gen} = \eta_{em}\left(\omega_{eng}\tau_{tc}, \frac{T_{eng}}{\tau_{tc}}\right). \end{aligned} \tag{7.24}$$

The problem was solved using an interior-point method implemented with the `fmincon` function developed by MathWorks [†] and the corresponding engine and e-machine operating points can be visualized in Figures 7.10 and 7.11.

[†] See [81] for a general outline of the algorithm's implementation or [16] for a more rigorous treatment of the matter.

Finally, these operating points were used to fit a quadratic model of the fuel consumption as a function of the gen-set power, to be used as the running cost for the EMS design presented in this section.

Listing 7.1: aaa

```
1
2  % Number of discretized gen-set power values
3  numPoints = 40;
4  % Discretized gen-set power values
5  elPwr = linspace(0, eng.maxPwr, 40);
6  % Initial guesses for the engine speed
7  engSpd0 = linspace(eng.idleSpd, eng.maxSpd, 40);
8  % Lower and upper bounds for the engine speed and torque
9  lb = [eng.idleSpd; 0];
10 ub = [eng.maxSpd; inf];
11 % Solver options
12 options = optimoptions("fmincon",...
13 "Algorithm","interior-point",...
```

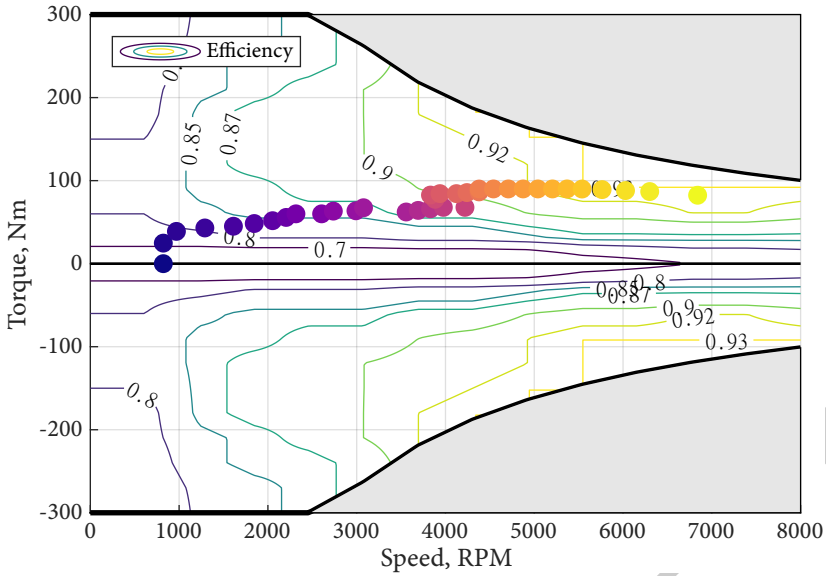


Figure 7.11: Generator operating points. Increasing values of gen-set power are marked by brighter colors.

```
"EnableFeasibilityMode",true,...
"SubproblemAlgorithm","cg");

% Run the solver
for n = 1:length(elPwr)
[optPoint(n,:), ~, flag(n)] = fmincon(@(x) eng.fuelMap(x(1), x
    (2)), [engSpd0(n), 10], [], [], [], [], lb, ub, ...
  @(x) elPwrCon(x, eng, em, elPwr(n)), options);
end

function [c,ceq] = elPwrCon(x, eng, em,  elPwr)
  % x(1): engSpd, x(2): engTrq
  % Maximum torque (inequality) constraint
  c = x(2) - eng.maxTrq(x(1));
  % Power balance (equality) constraint
  ceq = elPwr - em.effMap(x(1) .* em.tcSpdRatio, ...
    x(2) ./ em.tcSpdRatio) .* x(1) .* x(2);
end
```

Epilogue

8 Conclusion

8.1 Looking back

In this work, we discussed the implementation of two optimal control techniques and their application to energy management strategy design for hybrid electric vehicles:

- dynamic programming,
- differential dynamic programming.

Dynamic programming is a widely known technique with widespread application to EMS design, because of its unmatched versatility in handling virtually any conceivable optimal control problem regardless of the nature of the state dynamics, cost and constraints; although this comes at a computational cost that may be unbearable as the problem complexity grows due to the infamous curse of dimensionality.

The first contributions of this work, presented in § 3, are a discussion on practical implementation of dynamic programming algorithms and the development of a dedicated software. Many aspects of the numerical implementation of the technique were discussed as well as potential numerical issues and ways to avoid them, providing some guidelines on how to go from theory to practice. An open-source toolbox for dynamic programming was then presented, which combines state-of-the-art techniques with new unique features to obtain a numerically robust, flexible, user-friendly and computationally efficient software.

Having dealt with these general algorithmic aspects, § 4 focuses on powertrain modeling and its interaction with dynamic programming. Since these aspects are specific in their nature to the goal and requirements of energy management strategy design, a specific application was developed and a thorough investigation was conducted. The discussion, supported by numerical experiments, contained in this chapter offer some useful guidelines for researchers and engineers who wish to utilize dynamic programming for EMS design.

The second part of this thesis is centered on differential dynamic programming. While being far from recent, this technique has not been translated yet into publicly available software capable of dealing with applications of real-life interest, probably due to the inherent complexity of its numerical aspects.

In this dissertation, § 5 is devoted to summarizing the theoretical foundations for the algorithm, based on the existing literature. Then, § 6 introduces another major contribution of this work: that is, the development of a modern software tool to implement differential dynamic programming for complex control problems. Both the standard aspects and the unique aspects of this algorithm, such as the application of automatic differentiation and of an interior-point non-linear program solver, were discussed.

Finally, the second part culminates in § 7, where the differential dynamic programming software is effectively used to solve an EMS design problem for a series-hybrid vehicle. Numerical experiments were conducted to explore the software's strengths and weaknesses. One remarkable result is that the software already runs five times faster for a moderate model complexity, i.e. for a two-state, scalar control problem.

8.2 Looking ahead

As is typical in academic research, the objectives that were originally set to guide the research work of this thesis are far from being met. As is also typical, the results of this work open up numerous investigation topics, where we originally saw one.

8.2.1 Expanding DynaProg

The motivating factor for the development of DynaProg were to create a universal tool for dynamic programming that could solve any problem with arbitrary accuracy.

While the toolbox does provide some sensible improvements, it is still relatively narrow in scope with regard to the wide range of variations that can be made to dynamic programming algorithms to exploit a specific problem's structure. With this in mind, the toolbox was developed with the highest regard to code readability and modularity. The author's hope is that this effort, combined with its open-source nature, will foster contributions from a wide community that incrementally expands DynaProg into a swiss army knife for dynamic programming.

We must note however that even if such a tool became available many questions would still be unanswered about the interaction between the algorithm and a particular simulation model. This is a particularly broad field even if we restrict our attention to the topic of EMS design, due to the extremely wide variety of hybrid powertrains that can be conceived in terms of topology, components sizing and technology as well as control objectives. Therefore, it is yet to be seen if to what extent the considerations and experiments in § 4 generalize to other powertrains and control objectives.

8.2.2 Differential dynamic programming

No matter how much effort is put into developing computationally efficient implementations of a dynamic programming algorithm, the curse of dimensionality will always make it virtually impossible to use complex, multi-dimensional powertrain models. Nonetheless, it is hard to give up on its strengths, which are fundamentally the guarantee of optimality and the ability to handle strongly nonlinear problems.

The differential dynamic programming algorithm developed in this work was developed with the ultimate goal of keeping these important features while overcoming the curse of dimensionality, thus enabling the usage of high-dimensional powertrain models in optimal EMS design. Differential dynamic programming algorithms appears to be an excellent candidate due to their solid theoretical foundation laid out in the literature (e.g. [53, 31, 29, 32, 55, 54, 52, 50]).

Nonetheless, their practical implementation for complex problems spurred many issues that require thorough investigation. As a result, more work is needed before a robust software that is capable of solving problems of arbitrary complexity; it is the author's opinion, however, that the encouraging results obtained in the application in § 4 fully justifies further investigation.

The lines of research are numerous. First, there is the issue of setting the various algorithm parameters, such as the initial guess for the nominal control and endpoint multipliers, the cost convergence parameter η_1, the acceptance parameters used in the step-size adjustment and multipliers adjustment method (C, γ_1, γ_2), and the steepness of the constraining hyperplanes when dealing with state constraints. All these parameters may disrupt the algorithm's convergence if not

properly set. However, since all of them also have at least some relationship with the physics of the problem, it is likely that good rules of thumb can be developed.

Second, there are other variants of the algorithm that were not thoroughly tested in this work and which may prove more robust or computationally efficient. To name a few that are certainly worthy of investigation:

- In our implementation, we focused on continuous-time differential dynamic programming algorithms. There also exist discrete-time variants [84, 30], which are inherently designed to deal with difference equations rather than differential equations. Since these are typically easier to treat numerically, it is possible that transforming a continuous time problem into a discrete time formulation may turn out to be more accurate than attempting to directly deal with the original continuous time problem.

- The algorithm presented in this thesis does not handle discrete control variables. Hence, some method to adapt differential dynamic programming to a switched system should be conceived to treat control problems where the transmission's gear number is set as a control variable.

- Jarmark's convergence control parameter [34, 36, 35] is an alternative to the step-size adjustment method that may be more efficient for problems defined over long time intervals.

- There are alternative approaches to the constraining hyperplane technique that directly deal with state constraints. In particular, Ruxton [70] claims that a more efficient implementation can be obtained with multiplier penalty functions.

Bibliography

[1] Daniel Ambühl et al. "Explicit optimal control policy and its practical application for hybrid electric powertrains". In: *Control Engineering Practice* 18.12 (Dec. 2010), pp. 1429–1439. DOI: 10.1016/j.conengprac.2010.08.003.

[2] Joel A. E. Andersson et al. "CasADi: a software framework for nonlinear optimization and optimal control". In: *Mathematical Programming Computation* 11.1 (July 2018), pp. 1–36. DOI: 10.1007/s12532-018-0139-4.

[3] Pier Giuseppe Anselma. "Computationally efficient evaluation of fuel and electrical energy economy of plug-in hybrid electric vehicles with smooth driving constraints". In: *Applied Energy* 307 (Feb. 2022), p. 118247. DOI: 10.1016/j.apenergy.2021.118247.

[4] Pier Giuseppe Anselma and Giovanni Belingardi. "Next generation HEV powertrain design tools: roadmap and challenges". In: *SAE Technical Paper Series*. SAE International, Oct. 2019. DOI: 10.4271/2019-01-2602.

[5] Pier Giuseppe Anselma et al. "Rapid assessment of the fuel economy capability of parallel and series-parallel hybrid electric vehicles". In: *Applied Energy* 275 (Oct. 2020), p. 115319. DOI: 10.1016/j.apenergy.2020.115319.

[6] Uri M. Ascher and Chen Greif. *A first course in numerical methods*. Society for Industrial and Applied Mathematics, Feb. 2011. DOI: 10.1137/9780898719987.

[7] Richard Bellman. *Eye of the hurricane*. WSPC, June 1984. 356 pp. ISBN: 9971966018.

[8] Richard E. Bellman. "The theory of dynamic programming". In: *Bulletin of the American Mathematical Society* 60.6 (1954), pp. 503–515. DOI: 10.1090/s0002-9904-1954-09848-8.

[9] Richard E. Bellman and Stuart E. Dreyfus. *Applied dynamic programming*. Santa Monica, CA: RAND Corporation, 1962.

[10] Dimitri Bertsekas. *Reinforcement learning and optimal control*. Belmont, Massachusetts: Athena Scientific, 2019. ISBN: 1886529396.

[11] Domenico Bianchi et al. "A rule-based strategy for a series/parallel hybrid electric vehicle: an approach based on dynamic programming". In: *ASME 2010 Dynamic Systems and Control Conference, Volume 1*. ASMEDC, Jan. 2010. DOI: 10.1115/dscc2010-4233.

[12] Vladimir Boltyanski, Horst Martini, and V. Soltan. *Geometric methods and optimization problems*. Springer US, Jan. 2014. 444 pp. ISBN: 1461374278. URL: https://www.ebook.de/de/product/22339874/vladimir_boltyanski_horst_martini_v_soltan_geometric_methods_and_optimization_problems.html.

[13] Vladimir Boltyanski and Alexander Poznyak. *The robust maximum principle*. Springer Basel AG, Nov. 2011. ISBN: 0817681515. URL: https://www.ebook.de/de/product/15341917/vladimir_g_boltyanski_alexander_s_poznyak_the_robust_maximum_principle.html.

[14] Vladimir Boltyanski et al. "The maximum principle in the theory of optimal processes of control". In: *IFAC Proceedings Volumes* 1.1 (Aug. 1960), pp. 464–469. DOI: 10.1016/s1474-6670(17)70089-4.

[15] Hoseinali Borhan et al. "MPC-based energy management of a power-split hybrid electric vehicle". In: *IEEE Transactions on Control Systems Technology* 20.3 (May 2012), pp. 593–603. DOI: 10.1109/tcst.2011.2134852.

[16] Richard H. Byrd, Mary E. Hribar, and Jorge Nocedal. "An interior point algorithm for large-scale nonlinear programming". In: *SIAM Journal on Optimization* 9.4 (Jan. 1999), pp. 877–900. DOI: 10.1137/s1052623497325107.

[17] Stefano Di Cairano et al. "Stochastic MPC with learning for driver-predictive vehicle control and its application to HEV energy management". In: *IEEE Transactions on Control Systems Technology* 22.3 (May 2014), pp. 1018–1031. DOI: 10.1109/tcst.2013.2272179.

[18] A. Chasse, A. Sciarretta, and J. Chauvin. "Online optimal control of a parallel hybrid with costate adaptation rule". In: *IFAC Proceedings Volumes* 43.7 (July 2010), pp. 99–104. DOI: 10.3182/20100712-3-de-2013.00134.

[19] Zheng Chen et al. "Energy management for a power-split plug-in hybrid electric vehicle based on dynamic programming and neural networks". In: *IEEE Transactions on Vehicular Technology* 63.4 (May 2014), pp. 1567–1580. DOI: 10.1109/tvt.2013.2287102.

[20] Mehrdad Ehsani et al. *Modern electric, hybrid electric, and fuel cell vehicles*. Third Edition. Boca Raton: CRC Press, an imprint of Taylor & Francis Group, 2019. ISBN: 9781138330498.

[21] Philipp Elbert, Soren Ebbesen, and Lino Guzzella. "Implementation of dynamic programming for n-dimensional optimal control problems with final state constraints". In: *IEEE Transactions on Control Systems Technology* 21.3 (May 2013), pp. 924–931. DOI: 10.1109/tcst.2012.2190935.

[22] Stanley B. Gershwin and David H. Jacobson. *A discrete-time differential dynamic programming algorithm with application to optimal orbit transfer*. Tech. rep. No. 566. Harvard University, Aug. 1968.

[23] Stanley B. Gershwin and David H. Jacobson. "A discrete-time differential dynamic programming algorithm with application to optimal orbit transfer". In: *AIAA Journal* 8.9 (Sept. 1970), pp. 1616–1626. DOI: 10.2514/3.5955.

[24] David Goldberg. "What every computer scientist should know about floating-point arithmetic". In: *ACM Computing Surveys* 23.1 (Mar. 1991), pp. 5–48. DOI: 10.1145/103162.103163.

[25] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, Nov. 2008. 460 pp. ISBN: 0898716594. URL: https://www.ebook.de/de/product/23743815/andreas_griewank_andrea_walther_evaluating_derivatives_principles_and_techniques_of_algorithmic_differentiation.html.

[26] Ernst Hairer, Syvert P. Nørsett, and Gerhard Wanner. *Solving ordinary differential equations i: nonstiff problems*. Springer Berlin Heidelberg, Apr. 2008. 548 pp. ISBN: 3540566708. URL: https://www.ebook.de/de/product/1327253/ernst_hairer_syvert_p_noersett_gerhard_wanner_solving_ordinary_differential_equations_i.html.

[27] D. Jacobson and M. Lele. "A transformation technique for optimal control problems with a state variable inequality constraint". In: 14.5 (Oct. 1969), pp. 457–464. DOI: 10.1109/tac.1969.1099283.

[28] D. H. Jacobson. "Differential dynamic programming methods for solving bang-bang control problems". In: 13.6 (Dec. 1968), pp. 661–675. DOI: 10.1109/tac.1968.1099026.

[29] D. H. Jacobson. "New second-order and first-order algorithms for determining optimal control: a differential dynamic programming approach". In: 2.6 (Nov. 1968), pp. 411–440. DOI: 10.1007/bf00925746.

[30] David H. Jacobson and David Q. Mayne. *Differential dynamic programming*. Ed. by Richard Bellman. Vol. 24. Modern analytic and computational methods in science and mathematics. New York: American Elsevier Publishing Company, 1970. ISBN: 9780444000705.

[31] David Harris Jacobson. "Differential dynamic programming methods for determining optimal control of non-linear systems". PhD thesis. Imperial College London, Oct. 1967.

[32] David Harris Jacobson. "Second-order and second-variation methods for determining optimal control: a comparative study using differential dynamic programming". In: 7.2 (Feb. 1968), pp. 175–196. DOI: 10.1080/00207176808905594.

[33] Bram de Jager, Maarten Steinbuch, and Thijs van Keulen. "An adaptive sub-optimal energy management strategy for hybrid drive-trains". In: *IFAC Proceedings Volumes* 41.2 (2008), pp. 102–107. DOI: 10.3182/20080706-5-kr-1001.00017.

[34] Bernt S. A. Jarmark. "Convergence control in differential dynamic programing applied to air-to-air combat". In: *AIAA Journal* 14.1 (Jan. 1976), pp. 118–121. DOI: 10.2514/3.61338.

[35] Bernt S. A. Jarmark. "Differential dynamic programming techniques in differential games". In: *Control and Dynamic Systems*. Ed. by C. T. Leondes. Vol. 17. Academic Press, 1981, pp. 125–160. DOI: 10.1016/b978-0-12-012717-7.50010-3.

[36] Bernt S. A. Jarmark. "On convergence control in differential dynamic programming applied to realistic aircraft and differential game problems". In: *IEEE Conference on Decision and Control*. IEEE, Dec. 1977. DOI: 10.1109/cdc.1977.271620.

[37] Bernt S. A. Jarmark and Claes Hillberg. "Pursuit-evasion between two realistic aircraft". In: *Journal of Guidance, Control, and Dynamics* 7.6 (Nov. 1984), pp. 690–694. DOI: 10.2514/3.19914.

[38] Lars Johannesson, Mattias Asbogard, and Bo Egardt. "Assessing the potential of predictive control for hybrid vehicle powertrains using stochastic dynamic programming". In: *IEEE Transactions on Intelligent Transportation Systems* 8.1 (Mar. 2007), pp. 71–83. DOI: 10.1109/tits.2006.884887.

[39] Lars Johannesson and Bo Egardt. "Approximate dynamic programming applied to parallel hybrid powertrains". In: *IFAC Proceedings Volumes* 41.2 (2008), pp. 3374–3379. DOI: 10.3182/20080706-5-kr-1001.00573.

[40] J. T. B. A. Kessels et al. "Integrated energy & emission management for hybrid electric truck with SCR aftertreatment". In: *2010 IEEE Vehicle Power and Propulsion Conference*. IEEE, Sept. 2010. DOI: 10.1109/vppc.2010.5728990.

[41] Donald E. Kirk. "An introduction to dynamic programming". In: *IEEE Transactions on Education* 10.4 (Dec. 1967), pp. 212–219. DOI: 10.1109/te.1967.4320291.

[42] Donald E. Kirk. *Optimal control theory; An introduction*. Englewood Cliffs, N.J: Prentice-Hall, 1970. ISBN: 9780136380986.

[43] M. Koot et al. "Energy management strategies for vehicular electric power systems". In: *IEEE Transactions on Vehicular Technology* 54.3 (May 2005), pp. 771–782. DOI: 10.1109/tvt.2005.847211.

[44] Dongsuk Kum, Huei Peng, and Norman K. Bucknor. "Supervisory control of parallel hybrid electric vehicles for fuel and emission reduction". In: *Journal of Dynamic Systems, Measurement, and Control* 133.6 (Nov. 2011). DOI: 10.1115/1.4002708.

[45] Viktor Larsson, Lars Johannesson, and Bo Egardt. "Analytic solutions to the dynamic programming subproblem in hybrid vehicle energy management". In: *IEEE Transactions on Vehicular Technology* 64.4 (Apr. 2015), pp. 1458–1467. DOI: 10.1109/tvt.2014.2329864.

[46] Viktor Larsson, Lars Johannesson, and Bo Egardt. "Cubic spline approximations of the dynamic programming cost-to-go in HEV energy management problems". In: *2014 European Control Conference (ECC)*. IEEE, June 2014. DOI: 10.1109/ecc.2014.6862404.

[47] Daniel Liberzon. *Calculus of variations and optimal control theory*. Princeton University Press, Jan. 2012. 256 pp. ISBN: 0691151873.

[48] Chan-Chiao Lin et al. "Power management strategy for a parallel hybrid electric truck". In: *IEEE Transactions on Control Systems Technology* 11.6 (Nov. 2003), pp. 839–849. DOI: 10.1109/tcst.2003.815606.

[49] Jose Manuel Lujan et al. "Analytical optimal solution to the energy management problem in series hybrid electric vehicles". In: *IEEE Transactions on Vehicular Technology* 67.8 (Aug. 2018), pp. 6803–6813. DOI: 10.1109/tvt.2018.2821265.

[50] Krister Mårtensson. "A constraining hyperplane technique for state variable constrained optimal control problems". In: *Journal of Dynamic Systems, Measurement, and Control* 95.4 (Dec. 1973), pp. 380–389. DOI: 10.1115/1.3426739.

[51] Krister Mårtensson. "A new approach to constrained function optimization". In: 12.6 (Dec. 1973), pp. 531–554. DOI: 10.1007/bf00934776.

[52] Krister Mårtensson. "New approaches of the numerical solution of optimal control problems". PhD thesis. Department of Automatic Control, Lund Institute of Technology (LTH), 1972.

[53] David Mayne. "A second-order gradient method for determining optimal trajectories of non-linear discrete-time systems". In: 3.1 (Jan. 1966), pp. 85–95. DOI: 10.1080/00207176608921369.

[54] David Q. Mayne. "Differential dynamic programming - a unified approach to the optimization of dynamic systems". In: *Control and Dynamic Systems*. Ed. by C. T. Leondes. Vol. 10. Academic Press, 1973, pp. 179–254. ISBN: 9780120127108. DOI: 10.1016/b978-0-12-012710-8.50010-8.

[55] David Q. Mayne. "Strong variation algorithms for optimal control problems with control and terminal inequality constraints". In: IEEE, Dec. 1973. DOI: 10.1109/cdc.1973.269169.

[56] David Q. Mayne Mayne and E. Polak. "First-order strong variation algorithms for optimal control". In: 16.3-4 (Aug. 1975), pp. 277–301. DOI: 10.1007/bf01262937.

[57] Federico Miretti and Daniela Misul. "Robust modeling for optimal control of parallel hybrids with dynamic programming". In: *2022 IEEE/AIAA Transportation Electrification Conference and Electric Aircraft Technologies Symposium (ITEC+EATS), to appear*. June 2022.

[58] Scott Jason Moura et al. "A stochastic optimal control approach for power management in plug-in hybrid electric vehicles". In: *IEEE Transactions on Control Systems Technology* 19.3 (May 2011), pp. 545–555. DOI: 10.1109/tcst.2010.2043736.

[59] Y. L. Murphey et al. "Intelligent hybrid vehicle power control - part i: machine learning of optimal vehicle power". In: *IEEE Transactions on Vehicular Technology* 61.8 (Oct. 2012), pp. 3519–3530. DOI: 10.1109/tvt.2012.2206064.

[60] Cristian Musardo et al. "A-ECMS: an adaptive algorithm for hybrid electric vehicle energy management". In: *European Journal of Control* 11.4-5 (Jan. 2005), pp. 509–524. DOI: 10.3166/ejc.11.509-524.

[61] Uwe Naumann. *The art of differentiating computer programs: an introduction to algorithmic differentiation*. Philadelphia: Society for Industrial and Applied Mathematics, 2011. ISBN: 9781611972061.

[62] V. Ngo et al. "Predictive gear shift control for a parallel hybrid electric vehicle". In: *2011 IEEE Vehicle Power and Propulsion Conference*. IEEE, Sept. 2011. DOI: 10.1109/vppc.2011.6043185.

[63] G. Paganelli. "General supervisory control policy for the energy optimization of charge-sustaining hybrid electric vehicles". In: *JSAE Review* 22.4 (Oct. 2001), pp. 511–518. DOI: 10.1016/s0389-4304(01)00138-2.

[64] G. Paganelli et al. "Equivalent consumption minimization strategy for parallel hybrid powertrains". In: *Vehicular Technology Conference. IEEE 55th Vehicular Technology Conference. VTC Spring 2002 (Cat. No.02CH37367)*. Vol. 4. IEEE, Aug. 2002, pp. 2076–2081. DOI: 10.1109/vtc.2002.1002989.

[65] G. Paganelli et al. "Simulation and assessment of power control strategies for a parallel hybrid car". In: *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering* 214.7 (July 2000), pp. 705–717. DOI: 10.1243/0954407001527583.

[66] Jiankun Peng, Hongwen He, and Rui Xiong. "Rule based energy management strategy for a series–parallel plug-in hybrid electric bus optimized by dynamic programming". In: *Applied Energy* 185 (Jan. 2017), pp. 1633–1643. DOI: 10.1016/j.apenergy.2015.12.031.

[67] Pierluigi Pisu and Giorgio Rizzoni. "A comparative study of supervisory control strategies for hybrid electric vehicles". In: *IEEE Transactions on Control Systems Technology* 15.3 (May 2007), pp. 506–518. DOI: 10.1109/tcst.2007.894649.

[68] David J. W. Ruxton. "Differential dynamic programming and optimal control of inequality constrained continuous dynamic systems". MA thesis. Department of Mathematics and Computing, University of Central Queensland, Dec. 1991.

[69] David J. W. Ruxton. "Differential dynamic programming and state variable inequality constrained problems". In: Springer US, 1994, pp. 223–234. DOI: 10.1007/978-1-4615-2425-0_19.

[70] David J. W. Ruxton. "Differential dynamic programming applied to continuous optimal control problems with state variable inequality constraints". In: 3.2 (Apr. 1993), pp. 175–185. DOI: 10.1007/bf01968530.

[71] Farzad Rajaei Salmasi. "Control strategies for hybrid electric vehicles: evolution, classification, comparison, and future trends". In: *IEEE Transactions on Vehicular Technology* 56.5 (Sept. 2007), pp. 2393–2404. DOI: 10.1109/tvt.2007.899933.

[72] Marcelino Sánchez, Sébastien Delprat, and Theo Hofman. "Energy management of hybrid vehicles with state constraints: a penalty and implicit hamiltonian minimization approach". In: *Applied Energy* 260 (Feb. 2020), p. 114149. DOI: 10.1016/j.apenergy.2019.114149.

[73] N. J. Schouten, M. A. Salman, and N. A. Kheir. "Fuzzy logic control for parallel hybrid vehicles". In: *IEEE Transactions on Control Systems Technology* 10.3 (May 2002), pp. 460–468. DOI: 10.1109/87.998036.

[74] A. Simon et al. "Optimal supervisory control of a diesel HEV taking into account both DOC and SCR efficiencies". In: *IFAC-PapersOnLine* 51.9 (2018), pp. 323–328. DOI: 10.1016/j.ifacol.2018.07.053.

[75] Matteo Spano et al. "Exploitation of a particle swarm optimization algorithm for designing a lightweight parallel hybrid electric vehicle". In: *Applied Sciences* 11.15 (July 2021), p. 6833. DOI: 10.3390/app11156833.

[76] Chao Sun et al. "Velocity predictors for predictive energy management in hybrid electric vehicles". In: *IEEE Transactions on Control Systems Technology* 23.3 (2015), pp. 1197–1204. DOI: 10.1109/TCST.2014.2359176.

[77] Olle Sundström. "Optimal control and design of hybrid-electric vehicles". en. PhD thesis. 2009. DOI: 10.3929/ETHZ-A-005902040.

[78] Olle Sundström, Daniel Ambühl, and Lino Guzzella. "On implementation of dynamic programming for optimal control problems with final state constraints". In: *Oil & Gas Science and Technology–Revue de l'Institut Français du Pétrole* 65.1 (Feb. 2010), pp. 91–102. DOI: 10.2516/ogst/2009020.

[79] Olle Sundström and Lino Guzzella. "A generic dynamic programming matlab function". In: *2009 IEEE International Conference on Control Applications*. IEEE, July 2009. DOI: 10.1109/cca.2009.5281131.

[80] Olle Sundström, Lino Guzzella, and Patrik Soltic. "Torque-assist hybrid electric powertrain sizing: from optimal control towards a sizing law". In: *IEEE Transactions on Control Systems Technology* 18.4 (July 2010), pp. 837–849. DOI: 10.1109/tcst.2009.2030173.

[81] The MathWorks, Inc. *Constrained nonlinear optimization algorithms*. URL: https://web.archive.org/web/20220408124800/https://www.mathworks.com/help/optim/ug/constrained-nonlinear-optimization-algorithms.html (visited on 04/08/2022).

[82] Florian Tschopp et al. "Optimal energy and emission management of a diesel hybrid electric vehicle equipped with a selective catalytic reduction system". In: *SAE Technical Paper Series*. SAE International, Sept. 2015. DOI: 10.4271/2015-24-2548.

[83] Andreas Wächter and Lorenz T. Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". In: *Mathematical Programming* 106.1 (Apr. 2005), pp. 25–57. DOI: 10.1007/s10107-004-0559-y.

[84] Sidney Yakowitz. "Control and dynamic systems: advances in aerospace systems dynamics and control systems". In: *Control and Dynamic Systems*. Ed. by C. T. Leonides. Vol. 31. 1 of 3. Elsevier, 1989. Chap. Algorithms and Computational Techniques in Differential Dynamic Programming, pp. 75–91. DOI: 10.1016/b978-0-12-012731-3.50008-1.