Open Journal of Mathematical Optimization

Vincent Roulet, Siddhartha Srinivasa, Maryam Fazel & Zaid Harchaoui

Iterative Linear Quadratic Optimization for Nonlinear Control: Differentiable Programming Algorithmic Templates

Volume 5 (2024), article no. 8 (63 pages)

https://doi.org/10.5802/ojmo.32

Article submitted on January 18, 2023, revised on December 6, 2023, accepted on July 3, 2024.

Iterative Linear Quadratic Optimization for Nonlinear Control: Differentiable Programming Algorithmic Templates

Vincent Roulet
Google Brain, Seattle, USA (Work completed at the University of Washington before joining Google)
vroulet@google.com

Siddhartha Srinivasa
Paul G. Allen School of Computer Science and Engineering, University of Washington, Seattle, USA
siddh@cs.washington.edu

Maryam Fazel
Department of Electrical and Computer Engineering, University of Washington, Seattle, USA
mfazel@uw.edu

Zaid Harchaoui
Department of Statistics University of Washington, Seattle, USA
zaid@uw.edu

Abstract

Iterative optimization algorithms depend on access to information about the objective function. In a differentiable programming framework, this information, such as gradients, can be automatically derived from the computational graph. We explore how nonlinear control algorithms, often employing linear and/or quadratic approximations, can be effectively cast within this framework. Our approach illuminates shared components and differences between gradient descent, Gauss–Newton, Newton, and differential dynamic programming methods in the context of discrete time nonlinear control. Furthermore, we present line-search strategies and regularized variants of these algorithms, along with a comprehensive analysis of their computational complexities. We study the performance of the aforementioned algorithms on various nonlinear control benchmarks, including autonomous car racing simulations using a simplified car model. All implementations are publicly available in a package coded in a differentiable programming language.

Digital Object Identifier 10.5802/ojmo.32

Keywords Nonlinear Discrete Time Control, Differentiable Programming, Newton, Gauss–Newton, Dynamic Differentiable Programming.

Acknowledgments This work was supported by NSF DMS-1839371, DMS-2134012, CCF-2019844, CIFAR-LMB, NSF TRIPODS II DMS-2023166 and faculty research awards.

1 Introduction

We consider nonlinear control problems in discrete time with finite horizon, i.e., problems of the form

$$\begin{aligned} \min_{\substack{x_0,\ldots,x_\tau \in \mathbb{R}^{n_x} \\ u_0\ldots,u_{\tau-1} \in \mathbb{R}^{n_u}}} \quad & \sum_{t=0}^{\tau-1} h_t(x_t, u_t) + h_\tau(x_\tau) \\ \text{subject to} \quad & x_{t+1} = f_t(x_t, u_t), \quad \text{for } t \in \{0, \ldots, \tau-1\}, \quad x_0 = \bar{x}_0, \end{aligned} \tag{1}$$

where at time t, $x_t \in \mathbb{R}^{n_x}$ is the state of the system, $u_t \in \mathbb{R}^{n_u}$ is the control applied to the system, $f_t : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x}$ is the discrete dynamic, $h_t : \mathbb{R}^{n_x} \to \mathbb{R}$ is the cost on the state and control variables and $\bar{x}_0 \in \mathbb{R}^{n_x}$ is a given fixed initial state. Problem (1) is entirely determined by the initial state and the controls.

Problems of the form (1) have been tackled in various ways, from direct approaches using nonlinear optimization ([7, 16, 42, 45, 61, 62, 64]) to convex relaxations using semi-definite optimization ([10]). Numerous packages exist for such problems such as CasAdi ([2]), Pyomo ([13]), JumP ([17]), IPOPT ([59]), or SNOPT ([22]), Crocoddyl ([27]), acados ([56]). A popular approach of the former category proceeds by computing at each iteration the linear quadratic regulator associated with a linear quadratic approximation of the problem around the current candidate solutions ([26, 30, 50, 53]). The computed feedback policies are then applied either along the linearized dynamics or along the original dynamics to output a new candidate solution. Such canonical nonlinear control algorithms efficiently incorporate second-order information into the optimization procedure by exploiting the dynamical structure of the problem. This approach lends itself to an integration in a differentiable programming framework to extend this paradigm beyond first-order oracles.

Differentiable programming consists of the implementation of functions in a programming language that enables access to derivatives of these functions by automatic differentiation ([1, 4, 5, 9, 21, 24, 29, 43, 48, 49, 60]). Automatic differentiation itself has roots in the control literature, and its use is pervasive in numerous domains ([24]), in particular deep learning ([23, 65]). Canonical nonlinear control algorithms incorporating second order information can also be integrated in reinforcement learning pipelines ([28, 46]), and may then benefit from a differentiable programming viewpoint to isolate their underlying principles. These algorithms have indeed generally be presented through linear algebraic manipulations instantiated separately for each algorithm, which hinder a global perspective ([30, 38, 42, 50, 53]).

The motivation of this work is to cast all such algorithms in a common differentiable programming viewpoint to delineate the discrepancies between the different algorithms and identify the common subroutines. We review the implementation of (i) a Gauss–Newton method ([50]), a.k.a. Iterative Linear Quadratic Regulator (ILQR), (ii) a Newton method ([16, 31, 42]), (iii) a differential dynamic programming approach based on linear approximations of the dynamics and quadratic approximations of the costs, a.k.a. iterative Linear Quadratic Regulator (iLQR) ([53]), (iv) a differential dynamic programming approach based on quadratic approximations of both dynamics and costs, usually simply called DDP ([26]), and consider regularized variants of the aforementioned algorithms with their corresponding line searches. In turn, the differentiable programming viewpoint informs efficient handling of memory by appropriate check-pointing. An extended related work discussion is in Appendix B.

Outline

In Section 2 we recall how linear quadratic control problems are solved by dynamic programming and used as a building block for nonlinear control algorithms. The implementation of classical optimization oracles such as a gradient step, a Gauss–Newton step, or a Newton step is presented in Section 3. Section 4 details the rationale and implementation of differential dynamic programming approaches. Section 5 presents the computational complexities of each oracle in terms of space and time complexities in a differentiable programming framework. All algorithms are tested on several synthetic problems in Section 6: swinging-up a fixed pendulum, and autonomous car racing with simple dynamics. Code is available at `https://github.com/vroulet/ilqc`.

Appendices A, B, C, D detail notations, related work, proofs and line-search procedures respectively. A summary of all algorithms with detailed pseudocode and computational schemes is given in Appendix E. Alternative implementations using check-pointing and different linear algebra solvers are presented in Appendix F and G respectively. Experimental setups and additional experiments are detailed in Appendix H and I.

Notation

For a sequence of vectors $x_1, \ldots, x_\tau \in \mathbb{R}^{n_x}$, we denote by semicolons their concatenation s.t. $\boldsymbol{x} = (x_1; \ldots; x_\tau) \in \mathbb{R}^{\tau n_x}$. For a function $f : \mathbb{R}^d \to \mathbb{R}^n$, we denote by $\nabla f(x) := (\partial_{x_i} f_j(x))_{1 \leq i \leq d, 1 \leq j \leq n} \in \mathbb{R}^{d \times n}$ the transpose of the Jacobian of f on x. For a function $f : \mathbb{R}^d \times \mathbb{R}^p \to \mathbb{R}^n$, we denote for $x \in \mathbb{R}^d$, $y \in \mathbb{R}^p$, $\nabla_x f(x,y) = (\partial_{x_i} f_j(x,y))_{1 \leq i \leq d, 1 \leq j \leq n} \in \mathbb{R}^{d \times n}$ the partial transpose Jacobian of f w.r.t. x on (x,y).

For a multivariate function $f : \mathbb{R}^d \to \mathbb{R}^n$ composed of coordinates $f_j : \mathbb{R}^d \to \mathbb{R}$ for $j \in \{1, \ldots, n\}$, we denote its Hessian $x \in \mathbb{R}^d$ as a tensor $\nabla^2 f(x) := (\nabla^2 f_1(x), \ldots, \nabla^2 f_n(x)) \in \mathbb{R}^{d \times d \times n}$. For a multivariate function $f : \mathbb{R}^d \times \mathbb{R}^p \to \mathbb{R}^n$ composed of coordinates $f_j : \mathbb{R}^d \times \mathbb{R}^p \to \mathbb{R}$ for $j \in \{1, \ldots, n\}$, we decompose its Hessian on $x \in \mathbb{R}^d$, $y \in \mathbb{R}^p$ by defining, e.g., $\nabla^2_{xx} f(x,y) = (\nabla^2_{xx} f_1(x,y), \ldots, \nabla^2_{xx} f_n(x,y)) \in \mathbb{R}^{d \times d \times n}$. The quantities $\nabla^2_{yy} f(x,y) \in \mathbb{R}^{p \times p \times n}$, $\nabla^2_{xy} f(x,y) \in \mathbb{R}^{d \times p \times n}$, $\nabla^2_{yx} f(x,y) \in \mathbb{R}^{p \times d \times n}$ are defined similarly.

For a function $f : \mathbb{R}^d \to \mathbb{R}^n$, and $x \in \mathbb{R}^d$, we define the finite difference expansion of f around x, the linear expansion of f around x and the quadratic expansion of f around x as, respectively,

$$\delta_f^x(y) := f(x+y) - f(x), \qquad \ell_f^x(y) := \nabla f(x)^\top y, \qquad q_f^x(y) := \nabla f(x)^\top y + \frac{1}{2} \nabla^2 f(x)[y, y, \cdot]. \tag{2}$$

The linear and quadratic approximations of f around x are then $f(x+y) \approx f(x) + \ell_f^x(y)$ and $f(x+y) \approx f(x) + q_f^x(y)$ respectively. Tensor notations, such as $\nabla^2 f(x)[y, y, \cdot]$, inspired from ([39]), are detailed in Appendix A.

2 From Linear Quadratic Control Problem to Nonlinear Control Algorithm

Algorithms for nonlinear control problems revolve around solving linear quadratic control problems by dynamic programming. Therefore, we start by recalling the rationale of dynamic programming and how discrete time control problems with linear dynamics and quadratic costs can be solved by dynamic programming.

2.1 Dynamic Programming

The idea of dynamic programming is to decompose dynamical problems such as (1) into a sequence of nested subproblems defined by the *cost-to-go* c_t, from x_t at time $t \in \{0, \dots, \tau-1\}$:

$$\begin{aligned} c_t(x_t) \coloneqq & \min_{\substack{u_t,\dots,u_{\tau-1}\in\mathbb{R}^{n_u} \\ y_t,\dots,y_\tau\in\mathbb{R}^{n_x}}} \sum_{s=t}^{\tau-1} h_s(y_s, u_s) + h_\tau(y_\tau) \\ & \text{subject to} \quad y_{s+1} = f_s(y_s, u_s) \quad \text{for } s \in \{t, \dots, \tau-1\}, \quad y_t = x_t. \end{aligned}$$

The cost-to-go from x_τ at time τ is simply the last cost, namely, $c_\tau(x_\tau) = h_\tau(x_\tau)$, and the original problem (1) amounts to compute $c_0(\bar{x}_0)$. The cost-to-go functions define nested subproblems that are linked for $t \in \{0, \dots, \tau-1\}$ by *Bellman's equation* ([6])

$$\begin{aligned} c_t(x_t) &= \min_{u_t\in\mathbb{R}^{n_u}} h_t(x_t, u_t) + \min_{\substack{u_{t+1},\dots,u_{\tau-1}\in\mathbb{R}^{n_u} \\ y_{t+1},\dots,y_\tau\in\mathbb{R}^{n_x}}} \sum_{s=t+1}^{\tau-1} h_s(y_s, u_s) + h_\tau(y_\tau) \\ & \qquad \text{subject to} \quad y_{s+1} = f_s(y_s, u_s) \text{ for } s \in \{t+1, \dots, \tau-1\},\ y_{t+1} = f_t(x_t, u_t) \\ &= \min_{u_t\in\mathbb{R}^{n_u}} h_t(x_t, u_t) + c_{t+1}(f_t(x_t, u_t)). \end{aligned} \tag{3}$$

The optimal control at time t from state x_t is given by $u_t = \pi_t(x_t)$, where π_t, called a *policy*, is given by

$$\pi_t(x_t) \coloneqq \operatorname*{arg\,min}_{u_t\in\mathbb{R}^{n_u}} \{h_t(x_t, u_t) + c_{t+1}(f_t(x_t, u_t))\}.$$

Define the procedure that back-propagates (BP) the cost-to-go functions as

$$\mathrm{BP} : f_t, h_t, c_{t+1} \to \begin{pmatrix} c_t : x \to \min_{u\in\mathbb{R}^{n_u}} \{h_t(x, u) + c_{t+1}(f_t(x, u))\}, \\ \pi_t : x \to \operatorname*{arg\,min}_{u\in\mathbb{R}^{n_u}} \{h_t(x, u) + c_{t+1}(f_t(x, u))\} \end{pmatrix}.$$

A dynamic programming approach, formally described in Algorithm 1, solves problems of the form (1) as follows.

1. Compute recursively the cost-to-go functions c_t for $t = \tau, \dots, 0$ using Bellman's equation (3), i.e., compute from $c_\tau = h_\tau$,

   $$c_t, \pi_t = \mathrm{BP}(f_t, h_t, c_{t+1}) \quad \text{for } t \in \{\tau-1, \dots, 0\},$$

   and record at each step the policies π_t.
2. Unroll the optimal trajectory that starts from time 0 at $\bar{x}_0$, follows the dynamics f_t, and uses at each step the optimal control given by the computed policies, that is, starting from $x_0^* = \bar{x}_0$, compute

   $$u_t^* = \pi_t(x_t^*), \qquad x_{t+1}^* = f_t(x_t^*, u_t^*) \qquad \text{for } t = 0, \dots, \tau-1. \tag{4}$$

The resulting command $\boldsymbol{u}^* = (u_0^*; \dots; u_{\tau-1}^*)$ and trajectory $\boldsymbol{x}^* = (x_1^*; \dots; x_\tau^*)$ are then optimal for problem (1). In the following, the dynamic programming (DynProg) procedure, detailed[1] in Algorithm 1 in Appendix E, is denoted

$$\mathrm{DynProg} : (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, \mathrm{BP} \to u_0^*, \dots, u_{\tau-1}^*. \tag{5}$$

The bottleneck of the approach is the ability to solve Bellman's equation (3), i.e., having access to the procedure BP defined above.

[1] For ease of reference and comparisons, all procedures, algorithms, and computational schemes are grouped in Appendix E.

2.2 Linear Dynamic, Quadratic Cost

For linear dynamics and quadratic costs, problem (1) takes the form

$$\begin{aligned} \min_{\substack{x_0,\ldots,x_\tau\in\mathbb{R}^{n_x}\\ u_0\ldots,u_{\tau-1}\in\mathbb{R}^{n_u}}} & \sum_{t=0}^{\tau-1}\left(\frac{1}{2}x_t^\top P_t x_t + \frac{1}{2}u_t^\top Q_t u_t + x_t^\top R_t u_t + p_t^\top x_t + q_t^\top u_t\right) + \frac{1}{2}x_\tau^\top P_\tau x_\tau + p_\tau^\top x_\tau \\ \text{subject to} \quad & x_{t+1} = A_t x_t + B_t u_t, \quad \text{for } t\in\{0,\ldots,\tau-1\}, \quad x_0 = \bar{x}_0. \end{aligned}$$

Namely, we have $h_t(x_t,u_t) = \frac{1}{2}x_t^\top P_t x_t + \frac{1}{2}u_t^\top Q_t u_t + x_t^\top R_t u_t + p_t^\top x_t + q_t^\top u_t$ and $f_t(x_t,u_t) = A_t x_t + B_t u_t$. In that case, under appropriate conditions on the quadratic functions, Bellman's equation (3) can be solved analytically through a linear quadratic back-propagation (LQBP) as recalled in Lemma 1. Note that the operation LQBP defined in (6) amounts to computing the Schur complement of a block of the Hessian of the quadratic $x,u \to q_t(x,u) + c_{t+1}(\ell_t(x,u))$, namely, the block corresponding to the Hessian w.r.t. the control variables (see, e.g., [11, Appendix A.5.5]). The proofs of Lemma 1 and Corollary 2 are standard and are given in Appendix C.

Lemma 1. *For linear functions ℓ_t and quadratic functions q_t, c_{t+1} s.t. $q_t(x,\cdot\,) + c_{t+1}(\ell_t(x,\cdot\,))$ is strongly convex for any x, the procedure*

$$\text{LQBP} : (\ell_t, q_t, c_{t+1}) \to \begin{pmatrix} c_t : x \to \min_{u\in\mathbb{R}^{n_u}} \{q_t(x,u) + c_{t+1}(\ell_t(x,u))\} \\ \pi_t : x \to \operatorname*{arg\,min}_{u\in\mathbb{R}^{n_u}} \{q_t(x,u) + c_{t+1}(\ell_t(x,u))\} \end{pmatrix}, \tag{6}$$

can be implemented analytically as detailed in Algorithm 2.

If problem (1) consists of linear dynamics and quadratic costs that are strongly convex w.r.t. the control variable, the procedure LQBP can be applied iteratively in a dynamic programming approach to give the solution of the problem, as formally stated in Corollary 2.

Corollary 2. *Consider problem (1) such that for all $t\in\{0,\ldots,\tau-1\}$, f_t is linear, h_t is convex quadratic with $h_t(x,\cdot\,)$ strongly convex for any x, and h_τ is convex quadratic. Then, the solution of problem (1) is given by*

$$\boldsymbol{u}^* = \text{DynProg}((f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, \text{LQBP}),$$

with DynProg *and* LQBP *implemented in Algorithm 1 and Algorithm 2 respectively.*

2.3 Nonlinear Control Algorithm Example

Nonlinear control algorithms based on nonlinear optimization use linear or quadratic approximations of the dynamics and the costs at a current candidate sequence of controllers to apply a dynamic programming procedure to the resulting problem ([6, 16, 30, 50, 53]). For example, the Iterative Linear Quadratic Regulator (ILQR) algorithm uses linear approximations of the dynamics and quadratic approximations of the costs ([30]). Each iteration of the ILQR algorithm is composed of the three steps below illustrated in Figure 6.

Iterative Linear Quadratic Regulator Iteration

1. Forward pass: Given a set of control variables $u_0,\ldots,u_{\tau-1}$, compute the trajectory $x_1,\ldots,x_\tau$ as $x_{t+1} = f_t(x_t,u_t)$ starting from $x_0 = \bar{x}_0$, and the associated costs $h_t(x_t,u_t), h_\tau(x_\tau)$, for $t\in\{0,\ldots,\tau-1\}$. Record along the computations, i.e., for $t\in\{0,\ldots,\tau-1\}$, the gradients of the dynamics and the gradients and Hessians of the costs.
2. Backward pass: Compute the optimal policies associated with the linear quadratic control problem

$$\begin{aligned} \min_{\substack{y_0,\ldots y_\tau\in\mathbb{R}^{n_x}\\ v_0,\ldots,v_{\tau-1}\in\mathbb{R}^{n_u}}} & \sum_{t=0}^{\tau-1}\left(\frac{1}{2}y_t^\top P_t y_t + \frac{1}{2}v_t^\top Q_t v_t + y_t^\top R_t v_t + p_t^\top y_t + q_t^\top v_t\right) + \frac{1}{2}y_\tau^\top P_\tau y_\tau + p_\tau^\top y_\tau \\ \text{subject to} \quad & y_{t+1} = A_t y_t + B_t v_t, \quad \text{for } t\in\{0,\ldots,\tau-1\}, \quad y_0 = 0, \\ \text{where} \quad & P_t = \nabla^2_{x_t x_t} h_t(x_t,u_t) \;\; Q_t = \nabla^2_{u_t u_t} h_t(x_t,u_t) \;\; R_t = \nabla^2_{x_t u_t} h_t(x_t,u_t) \\ & p_t = \nabla_{x_t} h_t(x_t,u_t) \qquad q_t = \nabla_{u_t} h_t(x_t,u_t) \\ & A_t = \nabla_{x_t} f_t(x_t,u_t)^\top \;\; B_t = \nabla_{u_t} f_t(x_t,u_t)^\top. \end{aligned}$$

The problem above can be written compactly as

$$\min_{\substack{y_0,\ldots y_\tau \in \mathbb{R}^{n_x} \\ v_0,\ldots,v_{\tau-1}\in\mathbb{R}^{n_u}}} \sum_{t=0}^{\tau-1} q_{h_t}^{x_t,u_t}(y_t, v_t) + q_{h_\tau}^{x_\tau}(y_\tau) \tag{7}$$
$$\text{subject to} \quad y_{t+1} = \ell_{f_t}^{x_t,u_t}(y_t, v_t), \quad \text{for } t \in \{0,\ldots,\tau-1\}, \quad y_0 = 0,$$

where $q_{h_\tau}^{x_\tau}(y_\tau) = \frac{1}{2} y_\tau^\top P_\tau y_\tau + p_\tau^\top y_\tau$ and $q_{h_t}^{x_t,u_t}(y_t, v_t) = \frac{1}{2} y_t^\top P_t y_t + \frac{1}{2} v_t^\top Q_t v_t + y_t^\top R_t v_t + p_t^\top y_t + q_t^\top v_t$ are the quadratic expansions of the costs and $\ell_{f_t}^{x_t,u_t}(y_t, v_t) = A_t y_t + B_t v_t$ is the linear expansion of the dynamics, both expansions being defined around the current sequence of controls and associated trajectory. The optimal policies associated to this problem are obtained by computing recursively, starting from $c_\tau = q_{h_\tau}^{x_\tau}$,

$$c_t, \pi_t = \mathrm{LQBP}(\ell_{f_t}^{x_t,u_t}, q_{h_t}^{x_t,u_t}, c_{t+1}) \quad \text{for } t \in \{\tau-1,\ldots,0\},$$

where LQBP presented in Algorithm 2 outputs affine policies of the form $\pi_t : y_t \to K_t y_t + k_t$.

3. Roll-out pass: Define the set of candidate policies as $\{\pi_t^\gamma : y \to K_t y + \gamma k_t \ \text{ for } \gamma \geq 0\}$. The next sequence of controllers is then given as $u_t^{\mathrm{next}} = u_t + v_t^\gamma$, where v_t^γ is given by rolling out the policies π_t^γ from $y_0^\gamma = 0$ along the linearized dynamics as

$$v_t^\gamma = \pi_t^\gamma(y_t^\gamma), \quad y_{t+1} = \ell_{f_t}^{x_t,u_t}(y_t^\gamma, v_t^\gamma), \quad \text{for } t \in \{0,\ldots,\tau\}$$

for γ found by a line-search such that $\sum_{t=0}^{\tau-1}(h_t(x_t + y_t^\gamma, u_t + v_t^\gamma) - h_t(x_t, u_t)) + h_\tau(x_\tau + y_\tau^\gamma) - h_\tau(x_\tau) \leq \gamma c_0(0)$, with $c_0(0)$ the solution of the linear quadratic control problem (7).

The procedure is then repeated on the next sequence of control variables. Ignoring the line-search phase (namely, taking $\gamma = 1$), each iteration can be summarized as computing $\boldsymbol{u}^{\mathrm{next}} = \boldsymbol{u} + \boldsymbol{v}$ where

$$\boldsymbol{v} = \mathrm{DynProg}((\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (q_{h_t}^{x_t,u_t})_{t=0}^{\tau}, y_0, \mathrm{LQBP})$$

for $y_0 = 0$, where DynProg is the dynamic programming procedure implemented in Algorithm 1. Note that for convex costs h_t such that $h_t(x, \cdot)$ is strongly convex, the subproblems (7) satisfy the assumptions of Corollary 2.

The iterations of the following nonlinear control algorithms can always be decomposed into the three passes described above for the ILQR algorithm. The algorithms vary by (i) what approximations of the dynamics and the costs are computed in the forward pass, (ii) how the policies are computed in the backward pass, (iii) how the policies are rolled out.

3 Classical Optimization Oracle

Problem (1) is entirely determined by the choice of the initial state and a sequence of control variables, such that the objective in (1) can be written in terms of the control variables $\boldsymbol{u} = (u_0; \ldots; u_{\tau-1})$ as

$$\mathcal{J}(\boldsymbol{u}) := \sum_{t=0}^{\tau-1} h_t(x_t, u_t) + h_\tau(x_\tau)$$
$$\text{s.t.} \quad x_{t+1} = f_t(x_t, u_t) \quad \text{for } t \in \{0,\ldots,\tau-1\}, \quad x_0 = \bar{x}_0.$$

The objective can be decomposed into the costs and the control of τ steps of a sequence of dynamics defined as follows.

Definition 3. *We define the control of τ discrete time dynamics $(f_t : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x})_{t=0}^{\tau-1}$ as the function $f^{[\tau]} : \mathbb{R}^{n_x} \times \mathbb{R}^{\tau n_u} \to \mathbb{R}^{\tau n_x}$, which, given an initial point $x_0 \in \mathbb{R}^{n_x}$ and a sequence of controls $\boldsymbol{u} = (u_0; \ldots; u_{\tau-1}) \in \mathbb{R}^{\tau n_u}$, outputs the corresponding trajectory $x_1, \ldots, x_\tau$, i.e.,*

$$f^{[\tau]}(x_0, \boldsymbol{u}) := (x_1; \ldots; x_\tau) \tag{8}$$
$$\textit{s.t.} \quad x_{t+1} = f_t(x_t, u_t) \quad \textit{for } t \in \{0,\ldots,\tau-1\}.$$

Overall, problem (1) can be written as the minimization of a composition

$$\min_{\boldsymbol{u}\in\mathbb{R}^{\tau n_u}} \{\mathcal{J}(\boldsymbol{u}) = h \circ g(\boldsymbol{u})\}, \quad \text{where} \quad h(\boldsymbol{x}, \boldsymbol{u}) = \sum_{t=0}^{\tau-1} h_t(x_t, u_t) + h_\tau(x_\tau), \quad g(\boldsymbol{u}) = (f^{[\tau]}(\bar{x}_0, \boldsymbol{u}), \boldsymbol{u}), \tag{9}$$

for $\boldsymbol{x} = (x_1; \ldots; x_\tau)$ and $\boldsymbol{u} = (u_0; \ldots; u_{\tau-1})$. The implementation of classical oracles for problem (9) relies on the dynamical structure of the problem encapsulated in the control $f^{[\tau]}$ of the discrete time dynamics $(f_t)_{t=0}^{\tau-1}$.

3.1 Formulation

Classical optimization algorithms rely on the availability of oracles for the objective. Here, we consider these oracles to compute the minimizer of an approximation of the objective around the current point with an optional regularization term. Formally, at a point $\boldsymbol{u} \in \mathbb{R}^{\tau n_u}$, given a regularization $\nu \geq 0$, for an objective of the form

$$\min_{\boldsymbol{u} \in \mathbb{R}^{\tau n_u}} \quad h \circ g(\boldsymbol{u}),$$

as in (9), we consider

i. a *gradient* oracle to use a linear expansion of the objective, and to output, for $\nu > 0$,

$$\operatorname*{arg\,min}_{\boldsymbol{v} \in \mathbb{R}^{\tau n_u}} \left\{ \ell_{h \circ g}^{\boldsymbol{u}}(\boldsymbol{v}) + \frac{\nu}{2} \|\boldsymbol{v}\|_2^2 \right\} = -\nu^{-1} \nabla (h \circ g)(\boldsymbol{u}), \tag{10}$$

ii. a *Gauss–Newton* oracle to use a linear quadratic expansion of the objective, and to output

$$\operatorname*{arg\,min}_{\boldsymbol{v} \in \mathbb{R}^{\tau n_u}} \left\{ q_h^{g(\boldsymbol{u})}(\ell_g^{\boldsymbol{u}}(\boldsymbol{v})) + \frac{\nu}{2} \|\boldsymbol{v}\|_2^2 \right\} = -(\nabla g(\boldsymbol{u}) \nabla^2 h(g(\boldsymbol{u})) \nabla g(\boldsymbol{u}) + \nu \, \mathrm{I})^{-1} \nabla (h \circ g)(\boldsymbol{u}), \tag{11}$$

iii. a *Newton* oracle to use a quadratic expansion of the objective, and to output

$$\operatorname*{arg\,min}_{\boldsymbol{v} \in \mathbb{R}^{\tau n_u}} \left\{ q_{h \circ g}^{\boldsymbol{u}}(\boldsymbol{v}) + \frac{\nu}{2} \|\boldsymbol{v}\|_2^2 \right\} = -(\nabla^2 (h \circ g)(\boldsymbol{u}) + \nu \, \mathrm{I})^{-1} \nabla (h \circ g)(\boldsymbol{u}), \tag{12}$$

where ℓ_f^x, q_f^x are the linear and quadratic expansions of f around x as defined in the notations in (2).

Gauss–Newton and Newton oracles are generally defined without a regularization, i.e., for $\nu = 0$. However, in practice, a regularization may be necessary to ensure that Gauss–Newton and Newton oracles provide a descent direction. Moreover, the reciprocal of the regularization, $1/\nu$, can play the role of a stepsize as detailed in Appendix D. The regularization ν can then vary with the iterates similarly as in trust region methods ([41, Chapter 4]). Lemma 4 presents how the computation of the above oracles can be decomposed into the dynamical structure of the problem. The proof is detailed in Appendix C.

Lemma 4. *Consider a nonlinear dynamical problem summarized as*

$$\min_{\boldsymbol{u} \in \mathbb{R}^{\tau n_u}} h \circ g(\boldsymbol{u}), \quad \textit{where} \quad h(\boldsymbol{x}, \boldsymbol{u}) = \sum_{t=0}^{\tau-1} h_t(x_t, u_t) + h_\tau(x_\tau), \quad g(\boldsymbol{u}) = (f^{[\tau]}(\bar{x}_0, \boldsymbol{u}), \boldsymbol{u}),$$

with $f^{[\tau]}$ the control of τ dynamics $(f_t)_{t=0}^{\tau-1}$ as defined in Definition 3.

Let $\boldsymbol{u} = (u_0; \ldots; u_{\tau-1})$ and $f^{[\tau]}(\bar{x}_0, \boldsymbol{u}) = (x_1; \ldots; x_\tau)$. Gradient (10)*, Gauss–Newton* (11) *and Newton* (12) *oracles for $h \circ g$ amount to solving for $\boldsymbol{v}^* = (v_0^*; \ldots; v_{\tau-1}^*)$ linear quadratic control problems of the form*

$$\min_{\substack{v_0, \ldots, v_{\tau-1} \in \mathbb{R}^{n_u} \\ y_0, \ldots, y_\tau \in \mathbb{R}^{n_x}}} \quad \sum_{t=0}^{\tau-1} q_t(y_t, v_t) + q_\tau(y_\tau) \tag{13}$$
$$\text{subject to} \quad y_{t+1} = \ell_{f_t}^{x_t, u_t}(y_t, v_t) \quad \text{for } t \in \{0, \ldots, \tau-1\}, \quad y_0 = 0,$$

where for

i. *the gradient oracle* (10)*, $q_\tau(y_\tau) = \ell_{h_\tau}^{x_\tau}(y_\tau)$ and, for $0 \leq t \leq \tau - 1$,*

$$q_t(y_t, v_t) = \ell_{h_t}^{x_t, u_t}(y_t, v_t) + \frac{\nu}{2} \|v_t\|_2^2,$$

ii. *the Gauss–Newton oracle* (11)*, $q_\tau(y_\tau) = q_{h_\tau}^{x_\tau}(y_\tau)$ and, for $0 \leq t \leq \tau - 1$,*

$$q_t(y_t, v_t) = q_{h_t}^{x_t, u_t}(y_t, v_t) + \frac{\nu}{2} \|v_t\|_2^2,$$

iii. *for the Newton oracle* (12)*, $q_\tau(y_\tau) = q_{h_\tau}^{x_\tau}(y_\tau)$ and, defining*

$$\lambda_\tau = \nabla h_\tau(x_\tau), \quad \lambda_t = \nabla_{x_t} h_t(x_t, u_t) + \nabla_{x_t} f_t(x_t, u_t) \lambda_{t+1} \quad \textit{for } t \in \{\tau - 1, \ldots, 1\}, \tag{14}$$

we have, for $0 \leq t \leq \tau - 1$,

$$q_t(y_t, v_t) = q_{h_t}^{x_t, u_t}(y_t, v_t) + \frac{1}{2} \nabla^2 f_t(x_t, u_t)[\,\cdot\,, \cdot\,, \lambda_{t+1}](y_t, v_t) + \frac{\nu}{2} \|v_t\|_2^2,$$

where for $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x}$, $x \in \mathbb{R}^{n_x}$, $u \in \mathbb{R}^{n_u}$, $\lambda \in \mathbb{R}^{n_x}$, we define

$$\nabla^2 f(x, u)[\,\cdot\,, \cdot\,, \lambda] : (y, v) \to \nabla_{xx}^2 f(x, u)[y, y, \lambda] + 2 \nabla_{xu}^2 f(x, u)[y, v, \lambda] + \nabla_{uu}^2 f(x, u)[v, v, \lambda]. \tag{15}$$

From an optimization viewpoint, gradient, Gauss–Newton or Newton oracles are considered as black-boxes. Second order methods such as Gauss–Newton or Newton methods generally require solving a linear system at a cubic cost in the dimension of the problem ([39, Chapter 4]). Here, the dimension of the problem in the control variables is τn_u, with n_u, the dimension of the control variables, usually small (see the numerical examples in Section 6), but τ, the number of time steps, potentially large if, e.g., the discretization time step used to define (1) from a continuous time control problem is small while the original time length of the continuous time control problem is large. A cubic cost w.r.t. the number of time steps τ is then a priori prohibitive.

A closer look at the implementation of all the above oracles (10), (11), (12), shows that they all amount to solving linear quadratic control problems as presented in Lemma 4. Hence, they can be solved by a dynamic programming approach detailed in Section 3.2 at a cost linear w.r.t. the number of time steps τ. As a consequence, if the dimensions n_u, n_x of the control and state variables are negligible compared to the horizon τ, the computational complexities of Gauss–Newton and Newton oracles, detailed in Section 5 are of the same order as the computational complexity of a gradient oracle. This observation was done by [16, 42] for a Newton step and [50] for a Gauss–Newton step. [61] also presented how sequential quadratic programming methods can naturally be cast in a similar way. Lemma 4 casts all classical optimization oracles in the same formulation, including a gradient oracle.

The linear quadratic control problems can be solved by different procedures than dynamic programming such as using Riccati-based or parallel implementations as detailed in Appendix G ([62]). We focus on their resolution by dynamic programming to cast all algorithms in a common framework.

3.2 Implementation

Given Lemma 4, for $f^{[\tau]}(\bar{x}_0, \boldsymbol{u})$ the control of τ dynamics $(f_t)_{t=0}^{\tau-1}$ defined in Definition 3, classical optimization oracles for objectives of the form

$$\mathcal{J}(\boldsymbol{u}) = h \circ g(\boldsymbol{u}), \quad \text{where} \quad h(\boldsymbol{x}, \boldsymbol{u}) = \sum_{t=0}^{\tau-1} h_t(x_t, u_t) + h_\tau(x_\tau), \quad g(\boldsymbol{u}) = (f^{[\tau]}(\bar{x}_0, \boldsymbol{u}), \boldsymbol{u}),$$

can be implemented by (i) instantiating the linear quadratic control problem (13) with the chosen approximations, (ii) solving the linear quadratic control problem (13) by dynamic programming as detailed in Section 2. Precisely, their implementation can be split into the following three phases.

1. Forward pass: All oracles start by gathering the information necessary for the step in a forward pass that takes the generic form of Algorithm 5 and can be summarized as

   $$\mathcal{J}(\boldsymbol{u}), (m_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (m_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, m_{h_\tau}^{x_\tau} = \mathrm{Forward}(\boldsymbol{u}, (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, o_f, o_h)$$

   that compute the objective $\mathcal{J}(\boldsymbol{u})$ associated to the given sequence of controls $\boldsymbol{u}$ and record approximations $(m_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (m_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, m_{h_\tau}^{x_\tau}$ of the dynamics and the costs up to the orders o_f and o_h, respectively as

   $$m_{f_t}^{x_t,u_t} = \begin{cases} \ell_{f_t}^{x_t,u_t} & \text{if } o_f = 1 \\ q_{f_t}^{x_t,u_t} & \text{if } o_f = 2 \end{cases}, \quad m_{h_t}^{x_t,u_t} = \begin{cases} \ell_{h_t}^{x_t,u_t} & \text{if } o_h = 1 \\ q_{h_t}^{x_t,u_t} & \text{if } o_h = 2 \end{cases}, \quad m_{h_\tau}^{x_\tau} = \begin{cases} \ell_{h_\tau}^{x_\tau} & \text{if } o_h = 1 \\ q_{h_\tau}^{x_\tau} & \text{if } o_h = 2. \end{cases} \tag{16}$$

   The orders of approximation o_f, o_h for each algorithm are summarized in Figure 1.
2. Backward pass: Once approximations of the dynamics have been computed, a backward pass on the corresponding linear quadratic control problem (13) can be done as in the linear quadratic case presented in Section 2. The backward passes of the gradient oracle in Algorithm 6, the Gauss–Newton oracle in Algorithm 7 and the Newton oracle in Algorithm 8 take generally the form

   $$(\pi_t)_{t=0}^{\tau-1}, c_0 = \mathrm{Backward}((m_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (m_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, m_{h_\tau}^{x_\tau}, \nu).$$

   Namely, they take as input a regularization $\nu \geq 0$ and some approximations of the dynamics and the costs $(m_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (m_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, m_{h_\tau}^{x_\tau}$ computed in a forward pass, and return a set of policies and the final cost-to-go corresponding to the subproblem (13).
3. Roll-out pass: Given the output of a backward pass defined above, the oracle is computed by rolling out the policies along the linear trajectories defined in the subproblem (13). Formally, given a sequence of policies $(\pi_t)_{t=0}^{\tau-1}$, the oracles are then given as $\boldsymbol{v} = (v_0; \ldots; v_{\tau-1})$ computed, for $y_0 = 0$, by Algorithm 11 as

   $$\boldsymbol{v} = \mathrm{Roll}(y_0, (\pi_t)_{t=0}^{\tau-1}, (\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}).$$

Here the policies $(\pi_t)_{t=0}^{\tau-1}$ are output by one of the backward passes in Algorithm 6, Algorithm 7 or Algorithm 8. For the Gauss–Newton and Newton oracles, an additional procedure checks whether the subproblems are convex at each iteration as explained in more detail in Appendix E.

Gradient, Gauss–Newton, and Newton oracles are implemented by, respectively, Algorithm 12, Algorithm 13, Algorithm 14. Additional line-searches are presented in Appendix D. The computational schemes of a gradient, a Gauss–Newton and a Newton oracle are illustrated in Figures 5, 6 and 8 respectively.

Gradient back-propagation

For a gradient oracle (10), the procedure LQBP normally used to solve linear quadratic control problems simplifies to a linear back-propagation, LBP, presented in Algorithm 3 that implements

$$\mathrm{LBP} : (\ell_t^f, \ell_t^h, c_{t+1}, \nu) \to \begin{pmatrix} c_t : x \to \min_{u \in \mathbb{R}^{n_u}} \left\{ \ell_t^h(x,u) + c_{t+1}(\ell_t^f(x,u)) + \frac{\nu}{2}\|u\|_2^2 \right\} \\ \pi_t : x \to \operatorname*{arg\,min}_{u \in \mathbb{R}^{n_u}} \left\{ \ell_t^h(x,u) + c_{t+1}(\ell_t^f(x,u)) + \frac{\nu}{2}\|u\|_2^2 \right\} \end{pmatrix}, \tag{17}$$

for linear functions $\ell_t^f, \ell_t^h, c_{t+1}$. Plugging into the overall dynamic programming procedure, Algorithm 3, the linearizations of the dynamics and the costs, we get that the gradient oracle, Algorithm 6, computes affine cost-to-go functions of the form $c_t(y_t) = j_t^\top y_t + j_t^0$ with

$$j_\tau = \nabla h_\tau(x_\tau), \quad j_t = \nabla_{x_t} h_t(x_t, u_t) + \nabla_{x_t} f_t(x_t, u_t) j_{t+1} \quad \text{for } t \in \{0, \ldots, \tau-1\}.$$

Moreover, the policies are independent of the state variables, i.e., $\pi_t(y_t) = k_t$, with

$$k_t = -\nu^{-1}(\nabla_{u_t} h_t(x_t, u_t) + \nabla_{u_t} f_t(x_t, u_t) j_{t+1}) = -\nu^{-1} \nabla_{u_t} (h \circ g)(\boldsymbol{u}).$$

The roll-out of these policies is independent of the dynamics and output directly the gradient up to a factor $-\nu^{-1}$. Note that we naturally retrieve the gradient back-propagation algorithm ([24]).

4 Differential Dynamic Programming Oracle

The original differential dynamic programming algorithm was developed by [26] and revisited by, e.g., [32, 36, 38, 54]. The reader can verify from the aforementioned citations that our presentation matches the original formulation in, e.g., the quadratic case, while offering a larger perspective on the method that incorporates, e.g., linear quadratic approximations. Such approaches have also been called *direct multiple shooting* by [8].

4.1 Rationale

Denoting h the total cost as in (9) and $f^{[\tau]}$ the control in τ dynamics $(f_t)_{t=0}^{\tau-1}$, Differential Dynamic Programming (DDP) oracles consist in solving approximately

$$\min_{\boldsymbol{v} \in \mathbb{R}^{\tau n_u}} h(f^{[\tau]}(\bar{x}_0, \boldsymbol{u} + \boldsymbol{v}), \boldsymbol{u} + \boldsymbol{v}),$$

by means of a dynamic programming procedure and using the resulting policies to update the current sequence of controllers. For a consistent presentation with the classical optimization oracles presented in Section 3, we consider a regularized formulation of the DDP oracles, that is,

$$\min_{\boldsymbol{v} \in \mathbb{R}^{\tau n_u}} h(f^{[\tau]}(\bar{x}_0, \boldsymbol{u} + \boldsymbol{v}), \boldsymbol{u} + \boldsymbol{v}) + \frac{\nu}{2}\|\boldsymbol{v}\|_2^2, \tag{18}$$

for some regularization $\nu \geq 0$.

The objective in problem (18) can be rewritten as

$$h(f^{[\tau]}(\bar{x}_0, \boldsymbol{u} + \boldsymbol{v}), \boldsymbol{u} + \boldsymbol{v}) = h(f^{[\tau]}(\bar{x}_0, \boldsymbol{u})) + \delta_h^{f^{[\tau]}(\bar{x}_0, \boldsymbol{u})}(\delta_{f^{[\tau]}}^{\bar{x}_0, \boldsymbol{u}}(0, \boldsymbol{v}), \boldsymbol{v}), \tag{19}$$

where for a function f, δ_f^x is the finite difference expression of f around x as defined in the notations in (2). In particular, $\delta_{f^{[\tau]}}^{\bar{x}_0, \boldsymbol{u}}(0, \boldsymbol{v})$ is the trajectory defined by the finite differences of the dynamics given as

$$\delta_{f_t}^{x_t, u_t}(y_t, v_t) = f_t(x_t + y_t, u_t + v_t) - f_t(x_t, u_t).$$

The dynamic programming approach is then applied on the above dynamics. Namely, the goal is to solve

$$\min_{\substack{v_0,\ldots,v_{\tau-1}\in\mathbb{R}^{n_u}\\ y_0,\ldots,y_\tau\in\mathbb{R}^{n_x}}} \sum_{t=0}^{\tau-1} \delta_{h_t}^{x_t,u_t}(y_t,v_t) + \frac{\nu}{2}\|v_t\|_2^2 + \delta_{h_\tau}^{x_\tau}(y_\tau) \tag{20}$$
$$\text{subject to} \quad y_{t+1} = \delta_{f_t}^{x_t,u_t}(y_t,v_t) \quad \text{for } t\in\{0,\ldots,\tau-1\}, \quad y_0 = 0,$$

by dynamic programming. Denote then c_t^* the cost-to-go functions associated to problem (20) for $t\in\{0,\ldots\tau\}$. These cost-to-go functions satisfy the recursive equation

$$c_t^*(y_t) = \min_{v_t\in\mathbb{R}^{n_u}} \left\{\delta_{h_t}^{x_t,u_t}(y_t,v_t) + \frac{\nu}{2}\|v_t\|_2^2 + c_{t+1}^*(\delta_{f_t}^{x_t,u_t}(y_t,v_t))\right\}, \tag{21}$$

starting from $c_\tau^* = \delta_{h_\tau}^{x_\tau}$ and such that our objective is to compute $c_0^*(0)$. Since the dynamics $\delta_{f_t}^{x_t,u_t}$ are not linear and the costs $\delta_{h_t}^{x_t,u_t}$ are not quadratic, there is no analytical solution for the subproblem (21). To circumvent this issue, the cost-to-go functions are approximated as $c_t^*(y_t)\approx c_t(y_t)$, where c_t is computed from approximations of the dynamics and the costs. The approximation is done around the nominal value of the subproblem (20) which is $\boldsymbol{v}=0$ and corresponds to $\boldsymbol{y}=0$ and no change of the original objective in (19).

Denoting m_f an expansion of a function f around the origin such that $f(x)\approx f(0)+m_f(x)$, the cost-to-go functions are computed with an approximate back-propagation $\widehat{\mathrm{BP}}$ of cost-to-go functions:

$$\widehat{\mathrm{BP}} : \delta_t^f, \delta_t^h, c_{t+1} \to \begin{pmatrix} c_t : y \to (\delta_t^h + c_{t+1}\circ\delta_t^f)(0,0) + \min\limits_{v\in\mathbb{R}^{n_u}} \left\{+m_{\delta_t^h}(y,v) + m_{c_{t+1}\circ\delta_t^f}(y,v) + \frac{\nu}{2}\|v\|_2^2\right\}, \\ \pi_t : y \to \operatorname*{arg\,min}\limits_{v\in\mathbb{R}^{n_u}} \left\{m_{\delta_t^h}(y,v) + m_{c_{t+1}\circ\delta_t^f}(y,v) + \frac{\nu}{2}\|v\|_2^2\right\}\end{pmatrix}, \tag{22}$$

applied to the finite differences $\delta_{f_t}^{x_t,u_t}\to\delta_t^f$ and $\delta_{h_t}^{x_t,u_t}\to\delta_t^h$. A DDP oracle computes then a sequence of policies by iterating in a backward pass, starting from $c_\tau = m_{\delta_{h_\tau}^{x_\tau}}$,

$$c_t, \pi_t = \widehat{\mathrm{BP}}(\delta_{f_t}^{x_t,u_t}, \delta_{h_t}^{x_t,u_t}, c_{t+1}) \quad \text{for } t\in\{\tau-1,\ldots,0\}. \tag{23}$$

Given a set of policies, an approximate solution is given by rolling out the policies along the dynamics defining problem (20), i.e., by computing $v_0,\ldots,v_{\tau-1}$ as

$$v_t = \pi_t(y_t), \qquad y_{t+1} = \delta_{f_t}^{x_t,u_t}(y_t,v_t) = f_t(x_t+y_t, u_t+v_t) - f_t(x_t,u_t) \qquad \text{for } t=0,\ldots,\tau-1. \tag{24}$$

The main difference with the classical optimization oracles lies a priori in the computation of the policies in (23) detailed below and in the roll-out pass that uses the finite differences of the dynamics. The constant part of the cost-to-go functions is used for line-searches as detailed in Appendix D.

4.2 Detailed Derivation of the Backward Passes

Linear Approximation

If we consider a linear approximation for the composition of the cost-to-go function and the dynamics, we have

$$m_{c_{t+1}\circ\delta_{f_t}^{x,u}} = \ell_{c_{t+1}\circ\delta_f^{x,u}} = \ell_{c_{t+1}}^{\delta_f^{x,u}(0,0)} \circ \ell_{\delta_f^{x,u}} = \ell_{c_{t+1}} \circ \ell_f^{x,u},$$

where we denote simply $\ell_f = \ell_f^0$ the linear expansion of a function f around the origin.

Plugging this model into (22) and using linear approximations of the costs, the recursion (23) amounts to computing, starting from $c_\tau = \ell_{\delta_{h_\tau}^{x_\tau,u_\tau}} = \ell_{h_\tau}^{x_\tau,u_\tau}$,

$$\begin{aligned} c_t(y) &= \delta_{h_t}^{x_t,u_t}(0,0) + \min_{v\in\mathbb{R}^{n_u}} \ell_{\delta_{h_t}^{x_t,u_t}}(y,v) + c_{t+1}(\delta_{f_t}^{x_t,u_t}(0,0)) + \ell_{c_{t+1}}(\ell_{f_t}^{x_t,u_t}(y,v)) + \frac{\nu}{2}\|v\|_2^2, \\ &= \min_{v\in\mathbb{R}^{n_u}} \ell_{h_t}^{x_t,u_t}(y,v) + c_{t+1}(\ell_{f_t}^{x_t,u_t}(y,v)) + \frac{\nu}{2}\|v\|_2^2, \end{aligned}$$

where in the last line we used that the cost-to-go functions c_t are necessarily affine, s.t. $c_{t+1}(y) = c_{t+1}(0) + \ell_{c_{t+1}}(y)$. We retrieve then the same recursion as the one used for a gradient oracle (17), with the same policies. Since the computed policies are constant, they are not affected by the dynamics along which a roll-out phase is performed. In other words, the oracle returned by using linear approximations in a DDP approach is just a gradient oracle.

Linear Quadratic Approximation

If we consider a linear quadratic approximation for the composition of the cost-to-go function and the dynamics, we have

$$m_{c_{t+1}\circ\delta_{f^{x,u}}} = q_{c_{t+1}}^{\delta_f^{x,u}(0,0)} \circ \ell_{\delta_f^{x,u}} = q_{c_{t+1}} \circ \ell_f^{x,u},$$

where we denote simply $q_f = q_f^0$ the quadratic expansion of a function f around the origin. Plugging this model into (22) and using quadratic approximations of the costs, the recursion (23) amounts to computing, starting from $c_\tau = q_{\delta_{h_\tau}^{x_\tau,u_\tau}} = q_{h_\tau}^{x_\tau,u_\tau}$,

$$\begin{aligned} c_t(y) &= \delta_{h_t}^{x_t,u_t}(0,0) + \min_{v\in\mathbb{R}^{n_u}} q_{\delta_{h_t}^{x_t,u_t}}(y,v) + c_{t+1}(\delta_{f_t}^{x_t,u_t}(0,0)) + q_{c_{t+1}}^{\delta_f^{x,u}(0,0)} \circ \ell_{\delta_f^{x,u}}^{(0,0)}(y,v) + \frac{\nu}{2}\|v\|_2^2 \\ &= \min_{v\in\mathbb{R}^{n_u}} q_{h_t}^{x_t,u_t}(y,v) + c_{t+1}(0) + q_{c_{t+1}}(\ell_{f_t}^{x_t,u_t}(y,v)) + \frac{\nu}{2}\|v\|_2^2. \end{aligned} \tag{25}$$

If the costs h_t are convex for all t and $q_{h_t}^{x_t,u_t}(y,\cdot\,) + \frac{\nu}{2}\|\cdot\|_2^2$ is strongly convex for all t and all y, then the cost-to-go functions c_t are convex quadratics for all t, i.e., $c_{t+1}(y) = c_{t+1}(0) + q_{c_{t+1}}(y)$. In that case, the recursion (25) simplifies as

$$c_t(y) = \min_{v\in\mathbb{R}^{n_u}} q_{h_t}^{x_t,u_t}(y,v) + c_{t+1}(\ell_{f_t}^{x_t,u_t}(y,v)) + \frac{\nu}{2}\|v\|_2^2, \tag{26}$$

and the policies are given by the minimizer of (26). The recursion (26) is then the same as the recursion done when computing a Gauss–Newton oracle. Namely, the backward pass in this case is the backward pass of a Gauss–Newton oracle. Though the output policies are the same, the output of the oracle will differ since the roll-out phase does not follow the linearized trajectories in the DDP approach. The computational scheme of a DDP approach with linear quadratic approximations presented in Figure 7 is then almost the same as the one of a Gauss–Newton oracle presented in Figure 6, except that in the roll-out phase the linear approximations of the dynamics are replaced by finite differences of the dynamics. This DDP approach amounts to the iterative Linear Quadratic Regulator (iLQR) developed by [53].

Quadratic Approximation

If we consider a quadratic approximation for the composition of the cost-to-go function and the dynamics, we get

$$m_{c_{t+1}\circ\delta_f^{x,u}} = q_{c_{t+1}\circ\delta_f^{x,u}} = \frac{1}{2}\nabla^2 f(x,u)[\,\cdot\,,\cdot\,,\nabla c_{t+1}(0)] + q_{c_{t+1}} \circ \ell_f^{x,u},$$

where $\nabla^2 f(x,u)[\,\cdot\,,\cdot\,,\lambda]$ is defined in (15). Plugging this model into (22) and using quadratic approximations of the costs, the recursion (23) amounts to, starting from $c_\tau = q_{\delta_{h_\tau}^{x_\tau,u_\tau}} = q_{h_\tau}^{x_\tau,u_\tau}$,

$$\begin{aligned} c_t(y) &= \delta_{h_t}^{x_t,u_t}(0,0) + \min_{v\in\mathbb{R}^{n_u}} q_{\delta_{h_t}^{x_t,u_t}}(y,v) + c_{t+1}(\delta_{f_t}^{x_t,u_t}(0,0)) + q_{c_{t+1}\circ\delta_{f_t}^{x,u}}(y,v) + \frac{\nu}{2}\|v\|_2^2 \\ &= \min_{v\in\mathbb{R}^{n_u}} q_{h_t}^{x_t,u_t}(y,v) + c_{t+1}(0) + q_{c_{t+1}} \circ \ell_{f_t}^{x_t,u_t}(y,v) + \frac{1}{2}\nabla^2 f_t(x_t,u_t)[\,\cdot\,,\cdot\,,\nabla c_{t+1}(0)](y,v) + \frac{\nu}{2}\|v\|_2^2. \end{aligned} \tag{27}$$

Provided that the costs are convex and that $q_{h_t}^{x_t,u_t}(y,\cdot\,) + \frac{1}{2}\nabla^2 f_t(x_t,u_t)[\,\cdot\,,\cdot\,,\nabla c_{t+1}(0)](y,\cdot\,) + \frac{\nu}{2}\|\cdot\|_2^2$ is strongly convex for all t and all y, the cost-to-go functions c_t are convex quadratics for all t. In that case, the recursion (27) simplifies as

$$c_t(y) = \min_{v\in\mathbb{R}^{n_u}} q_{h_t}^{x_t,u_t}(y,v) + c_{t+1}(\ell_{f_t}^{x_t,u_t}(y,v)) + \frac{1}{2}\nabla^2 f_t(x_t,u_t)[\,\cdot\,,\cdot\,,\nabla c_{t+1}(0)](y,v) + \frac{\nu}{2}\|v\|_2^2, \tag{28}$$

and the policies are given by the minimizer of (28). The overall backward pass is detailed in Algorithm 9.

Compared to the backward pass of the Newton oracle in Algorithm 8, we note that the additional cost derived from the curvatures of the dynamics is not computed the same way. Namely, the Newton oracle computes this additional cost by using back-propagated adjoint variables in (14), while in the DDP approach the additional cost is directly defined through the previously computed cost-to-go function. Figure 9 illustrates the computational scheme of the implementation of DDP with quadratic approximations and can be compared to the computational scheme of the Newton oracle in Figure 8.

Note that, while we used second order Taylor expansions for the compositions and the costs, the approximate cost-to-go-functions c_t are *not* second order Taylor expansion of the true cost-to-go functions c_t^*, except for c_τ. Indeed, c_t is computed as an approximate solution of the Bellman equation. The true Taylor expansion of the cost-to-go function requires the gradient and the Hessian of the cost and the dynamic in (27) computed at the minimizer of the subproblem. Here, since we only use an approximation of the minimizer, we do not have access to the true gradient and Hessian of the cost-to-go function.

4.3 Implementation

The implementation of the DDP oracles follows the same steps as the ones given for classical optimization oracles as detailed below. The implementation of a DDP oracle with linear quadratic approximations is given in Algorithm 15 and illustrated in Figure 7. The implementation of a DDP oracle with quadratic approximations is given in Algorithm 16 and illustrated in Figure 9.

1. Forward pass: As for the classical optimization methods, the oracles start by gathering the information necessary for the backward pass using Algorithm 5 that computes

   $$\mathcal{J}(\boldsymbol{u}), (m_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (m_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, m_{h_\tau}^{x_\tau} = \mathrm{Forward}(\boldsymbol{u}, (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, o_f, o_h),$$

   where o_f and o_h define the order of the approximations $(m_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (m_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, m_{h_\tau}^{x_\tau}$ of the dynamics and the costs up to the orders o_f and o_h as in (16).
2. Backward pass: As for the classical optimization oracles, the backward pass can generally be written

   $$(\pi_t)_{t=0}^{\tau-1}, c_0 = \mathrm{Backward}((m_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (m_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, m_{h_\tau}^{x_\tau}, \nu),$$

   If linear approximations are used, the backward pass is given in Algorithm 6, if linear quadratic approximations are used, the backward pass is given in Algorithm 7 and if quadratic approximations are used, the backward pass is given in Algorithm 9.
3. Roll-out pass: The roll-out phase differs by using finite differences of the original dynamics of problem (20) rather than the linearized dynamics. Formally, given a sequence of policies $(\pi_t)_{t=0}^{\tau-1}$, the oracles are then given as $\boldsymbol{v} = (v_0; \ldots; v_{\tau-1})$ computed, for $y_0 = 0$, by Algorithm 11 as

   $$\boldsymbol{v} = \mathrm{Roll}(y_0, (\pi_t)_{t=0}^{\tau-1}, (\delta_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}),$$

   where $\delta_{f_t}^{x_t,u_t}(y_t, v_t) = f_t(x_t + y_t, u_t + v_t) - f_t(x_t, u_t)$.

5 Computational Complexity

In Figure 1, we present a summary of the different algorithms presented in this manuscript. We added in parentheses the names usually given for these methods. Additional line-search mechanisms are presented in Appendix D. The overall implementations are detailed in Appendix E. We consider then the computational complexities of the algorithms in a differentiable programming framework.

Formal Computational Complexity

We present in Table 1 the computational complexities of the algorithms following the implementations described in Section 3 and Section 4 and detailed in Appendix E. We ignore the additional cost of the line-searches which requires a theoretical analysis of the admissible stepsizes depending on the smoothness properties of the dynamics and the costs. We consider for simplicity that the cost of evaluating a function $f : \mathbb{R}^d \to \mathbb{R}^n$ is of the order of $O(nd)$, as it is the case if f is linear. For the computational complexities of the core operation of the backward pass, i.e, LQBP in Algorithm 2 or LBP in Algorithm 3, we simply give the leading computational complexities, which, in the case of LQBP, are the matrix multiplications and inversions. The time complexities differ depending on whether linear or quadratic approximations of the costs are used. In the latter case, matrices of size $n_u \times n_u$ need to be inverted and matrices of size $n_x \times n_x$ need to be multiplied. However, all oracles have a linear time complexity with respect to the horizon τ.

We note that the space complexities of the gradient descent and the Gauss–Newton method or the DDP approach with linear quadratic approximations are essentially the same. On the other hand, the space complexity of the Newton oracle is a priori larger.

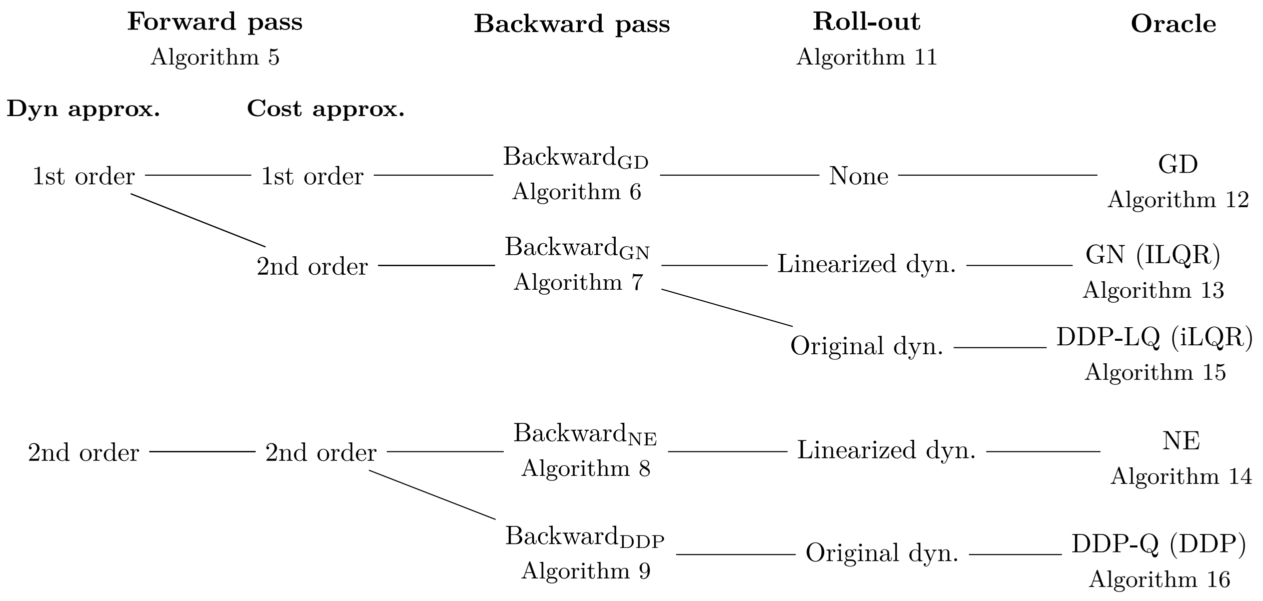


Figure 1 Taxonomy of non-linear control oracles. GD stands for gradient Descent, GN for Gauss–Newton, NE for Newton, DDP-LQ and DDP-Q stand for DDP with linear quadratic or quadratic approx. The iterations of the algorithms use a line-search procedure presented in Algorithm 17 as illustrated in Algorithm 18.

Computational Complexity in a Differentiable Programming Framework

The decomposition of each oracle between forward, backward and roll-out passes has the advantage to clarify the discrepancies between each approach. However, a careful implementation of these oracles only requires storing in memory the function and the inputs given at each time-step. Namely, the forward pass can simply keep in memory h_t, f_t, x_t, u_t for $t \in \{0, \ldots, \tau\}$. The backward pass computes then, on the fly, the information necessary to compute the policies. This amounts to a simple system of check-pointing, a strategy used in differentiable programming to circumvent the memory cost of the reverse-mode of automatic differentiation ([24]).

Such an approach is detailed in Appendix F. In summary, by considering an implementation that simply stores in memory the inputs and the programs that implement the functions, a Newton oracle and an oracle based on a DDP approach with quadratic approximation have the same time and space complexities as their linear quadratic counterparts up to constant factors. This remark was done by [40] for implementing a DDP algorithm with quadratic approximations.

6 Experiments

The control environments considered are thoroughly described in Appendix H. The code is publicly available at `https://github.com/vroulet/ilqc`. Additional experiments are presented in Appendix I, a comparison of all algorithms is presented in Figure 25.

All the following plots are in log-scale where on the vertical axis we plot $\log\left(\mathcal{J}(\boldsymbol{u}^{(k)})/\mathcal{J}(\boldsymbol{u}^{(0)})\right)$ with $\mathcal{J}$ the objective, $\boldsymbol{u}^{(k)}$ the set of controls at iteration k. The acronyms (GD, GN, NE, DDP-LQ, DDP-Q) correspond to the taxonomy of algorithms presented in Figure 1. Finally, the algorithms are stopped if no valid stepsizes have been found by line-search beyond machine precision ε, or if the relative difference in terms of costs is smaller than machine precision, where $\varepsilon \approx 10^{-16}$ as we ran these experiments in double precision. Hence, if a curve stops, this means that the linesearch did not find a valid stepsize beyond this point.

6.1 Linear Quadratic Approximation

We compare first a gradient descent and nonlinear control algorithms with linear quadratic approximations, i.e., GN or DDP-LQ with directional or regularized steps. We make the following observations.

1. Cost along iterations (Figure 2)
   - GN and DDP-LQ always outperform GD by several order of magnitudes.

Table 1 Space and time complexities of the oracles of Sections 3 and 4. Acronyms are given in Figure 1.

	Time complexities of the forward pass in Algorithm 5
Function eval. ($o_f = o_h = 0$)	$\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{f_t}+\underbrace{n_x+n_u}_{h_t}\Big) = O(\tau({n_x}^2+n_xn_u))$
Lin. (GD) ($o_f = o_h = 1$)	$\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{f_t,\nabla f_t}+\underbrace{n_x+n_u}_{h_t,\nabla h_t}\Big) = O(\tau({n_x}^2+n_xn_u))$
Lin.-quad. (GN/DDP-LQ) ($o_f = 1, o_h = 2$)	$\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{f_t,\nabla f_t}+\underbrace{n_x+n_u}_{h_t,\nabla h_t}+\underbrace{{n_x}^2+{n_u}^2+n_xn_u}_{\nabla^2 h_t}\Big) = O(\tau(n_x+n_u)^2)$
Quad. (NE/DDP-Q) ($o_f = o_h = 2$)	$\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{f_t,\nabla f_t}+\underbrace{({n_x}^2+{n_u}^2+n_xn_u)n_x}_{\nabla^2 f_t}+\underbrace{n_x+n_u}_{h_t,\nabla h_t}+\underbrace{{n_x}^2+{n_u}^2+n_xn_u}_{\nabla^2 h_t}\Big) = O(\tau n_x(n_x+n_u)^2)$

	Space complexities of the forward pass in Algorithm 5
Function eval. ($o_f = o_h = 0$)	0
Lin. (GD) ($o_f = o_h = 1$)	$\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{\nabla f_t}+\underbrace{n_x+n_u}_{\nabla h_t}\Big) = O(\tau({n_x}^2+n_xn_u))$
Lin.-quad. (GN/DDP-LQ) ($o_f = 1, o_h = 2$)	$\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{\nabla f_t}+\underbrace{n_x+n_u}_{\nabla h_t}+\underbrace{{n_x}^2+{n_u}^2+n_xn_u}_{\nabla^2 h_t}\Big) = O(\tau(n_x+n_u)^2)$
Quad. (NE/DDP-Q) ($o_f = o_h = 2$)	$\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{\nabla f_t}+\underbrace{({n_x}^2+{n_u}^2+n_xn_u)n_x}_{\nabla^2 f_t}+\underbrace{n_x+n_u}_{\nabla h_t}+\underbrace{{n_x}^2+{n_u}^2+n_xn_u}_{\nabla^2 h_t}\Big) = O(\tau n_x(n_x+n_u)^2)$

	Time complexities of the backward passes in Algorithms 6, 7, 8, 9 and the roll-out in Algorithm 11
GD	$\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{\text{Roll}}+\underbrace{{n_x}^2+n_xn_u}_{\text{LBP}}\Big) = O(\tau({n_x}^2+n_xn_u))$
GN/DDP-LQ	$\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{\text{Roll}}+\underbrace{{n_x}^3+{n_u}^3+{n_u}^2n_x}_{\text{LQBP}}\Big) = O(\tau(n_x+n_u)^3)$
NE/DDP-Q	$\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{\text{Roll}}+\underbrace{{n_x}^3+{n_u}^3+{n_u}^2n_x}_{\text{LQBP}}+\underbrace{({n_x}^2+{n_u}^2+n_xn_u)n_x}_{\nabla f_t^2[\cdot,\cdot,\lambda]}\Big) = O(\tau(n_x+n_u)^3)$

- DDP-LQ generally performs better or on par with GN, for the same steps (directional or regularized).
- For GN, regularized steps generally provide a more steady convergence than directional steps. The later do not find a valid stepsize in the real car example, which involves highly nonlinear dynamics (see Appendix H). However, once in a quadratically convergent phase the directional steps can provide faster convergence than the regularized steps (see e.g. GN dir vs GN reg on the pendulum on a cart example).
- For DDP-LQ, similar observations can be done. Only on the simple pendulum problem, directional steps slightly outperform regularized steps

2. Cost along computational time (Figure 19)
   - In terms of time, regularized steps may require fewer evaluations during the line-search as they incorporate previous stepsizes and may provide faster convergence in time.
3. Gradient norm along iterations (Figure 21)
   - Algorithms based on linear quadratic approximations generally display a late quadratic convergence in terms of gradient norm. One exception is the realistic model of the car, where only the regularized steps versions are able to make substantial progress along the iterations and such progress is only linear (in log-log plot scale).

6.2 Quadratic Approximation

We compare now nonlinear control algorithms with quadratic approximations, i.e., NE or DDP-Q.

1. Cost along iterations (Figure 3)
   - As for the linear-quadratic approximations, the DDP approach (here DDP-Q) generally outperforms or performs on par with its Newton (NE) counterpart.
   - For Newton, the regularized steps generally outperform the directional steps.
   - For DDP-Q, the regularized steps outperform the directional steps on all examples but the first pendulum examples.
2. Cost along computational time (Figure 20)
   - In terms of time, all algorithms appear to generally perform on par.
3. Gradient norm along iterations (Figure 22)
   - We generally observe quadratic convergence in gradient norm for all algorithms in late stage of training. However, quadratic convergence of Newton generally appears later than DDP.

Acknowledgments

The authors deeply thank Alexander Liniger for his help on implementing the bicycle model of a car. The authors also thank Dmitriy Drusvyatskiy, Krishna Pillutla and John Thickstun for fruitful discussions on the paper and the code.

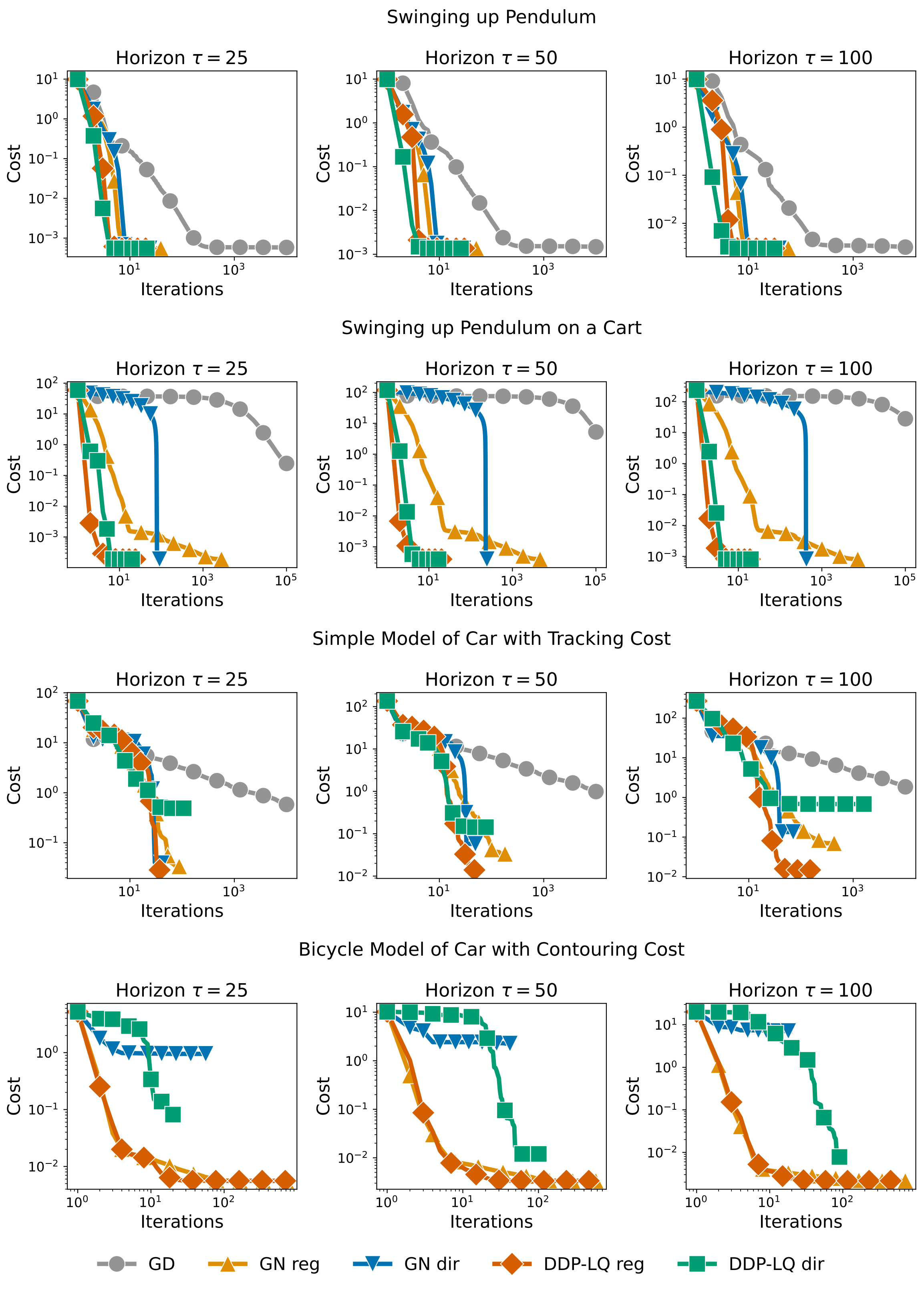


Figure 2 Cost along iterations on various control problems detailed in Appendix H with algorithms using linear (GD) or linear-quadratic approximations (GN, DDP-LQ, see Figure 1 for taxonomy details) and directional (dir (43)) or regularized (reg (45)) steps.

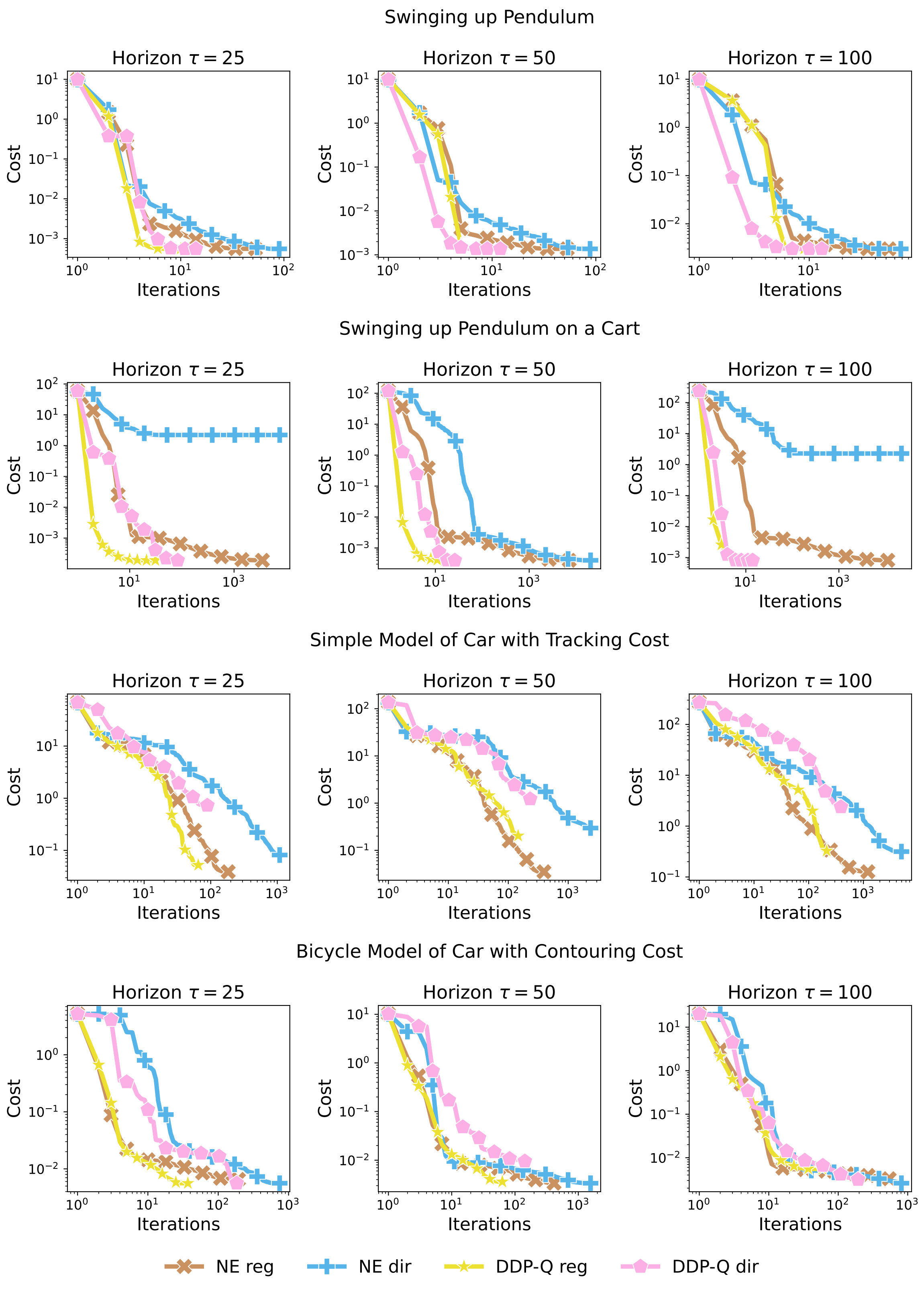


Figure 3 Cost along iterations on various control problems detailed in Appendix H with algorithms using quadratic approximations (NE, DDP-Q, see Figure 1 for taxonomy details) and directional (dir (43)) or regularized (reg (45)) steps.

Appendix

The appendix is organized as follows.

1. Appendix A presents tensor notations used to describe algorithms with second-order information on the dynamics.
2. Appendix B expands the discussion of related work.
3. Appendix C details the proofs of the results claimed in the main text.
4. Appendix D presents line-search mechanisms incorporated in the algorithms to ensure their efficiency.
5. Appendix E details all pseudocode algorithms with associated computational graphs in a differentiable programming framework.
6. Appendix F gives additional complexities when implementing the oracles with checkpointing.
7. Appendix G presents alternative ways to compute oracles using the structure of the subproblems.
8. Appendix H details the control environments on which the algorithms are tested.
9. Appendix I presents additional experiments: the convergence of the algorithms in time, and the stepsize selected along the iterations by a line-search procedure.

A Tensor Notation

A tensor $\mathcal{A} = (a_{i,j,k})_{1\leq i\leq d, 1\leq j\leq p, 1\leq k\leq n} \in \mathbb{R}^{d\times p\times n}$ is represented as a list of matrices $\mathcal{A} = (A_1, \ldots, A_n)$ where $A_k = (a_{i,j,k})_{1\leq i\leq d, 1\leq j\leq p} \in \mathbb{R}^{d\times p}$ for $k \in \{1, \ldots n\}$. Given $\mathcal{A} \in \mathbb{R}^{d\times p\times n}$ and $P \in \mathbb{R}^{d\times d'}, Q \in \mathbb{R}^{p\times p'}, R \in \mathbb{R}^{n\times n'}$, we denote

$$\mathcal{A}[P, Q, R] := \left(\sum_{k=1}^n R_{k,1} P^\top A_k Q, \ldots, \sum_{k=1}^n R_{k,n'} P^\top A_k Q\right) \in \mathbb{R}^{d'\times p'\times n'}.$$

For $\mathcal{A}_0 \in \mathbb{R}^{d_0\times p_0\times n_0}$, $P \in \mathbb{R}^{d_0\times d_1}, Q \in \mathbb{R}^{p_0\times p_1}, R \in \mathbb{R}^{n_0\times n_1}$ denote $\mathcal{A}_1 = \mathcal{A}_0[P, Q, R] \in \mathbb{R}^{d_1\times p_1\times n_1}$. Then, for $S \in \mathbb{R}^{d_1\times d_2}, T \in \mathbb{R}^{p_1\times p_2}, U \in \mathbb{R}^{n_1\times n_2}$, we have $\mathcal{A}_1[S, T, U] = \mathcal{A}_0[PS, QT, RU] \in \mathbb{R}^{d_2\times p_2\times n_2}$. If P, Q or R are identity matrices, we use the symbol " $\cdot$ " in place of the identity matrix. For example, we denote $\mathcal{A}[P, Q, \mathrm{I}_n] = \mathcal{A}[P, Q, \cdot] = \left(P^\top A_1 Q, \ldots, P^\top A_n Q\right)$. If P, Q or R are vectors we consider the flattened object. In particular, for $x \in \mathbb{R}^d, y \in \mathbb{R}^p$, we denote $\mathcal{A}[x, y, \cdot] = \left(x^\top A_1 y, \ldots, x^\top A_n y\right)^\top \in \mathbb{R}^n$, rather than having $\mathcal{A}[x, y, \cdot] \in \mathbb{R}^{1\times 1\times n}$. Similarly, for $z \in \mathbb{R}^n$, we denote $\mathcal{A}[\cdot, \cdot, z] = \sum_{k=1}^n z_k A_k \in \mathbb{R}^{d\times p}$. Such notations follow the ones used by [39, Chapter 5] to study third-order derivatives.

B Related Work

Nonlinear control problems of the form (1) stem from the discretization of generic optimal control problems in continuous time of the form

$$\begin{aligned} \min_{x(\cdot), u(\cdot)} \quad & \int_0^T h(x(t), u(T)) + h_T(x(T)) \\ \text{subject to} \quad & \dot{x}(t) = f(x(t), u(t)), \quad x(0) = \bar{x}_0, \end{aligned} \tag{29}$$

Continuous optimal control problems of the form (29) can be tackled in various ways ([14]). One can approach the problem from a *dynamic programming* perspective to derive the Hamilton–Jacobi–Bellman equation, a partial differential equation in state space ([34]). Alternatively, one can derive necessary optimality conditions for (29) to derive a boundary value problem. Such a method is referred to as an *indirect method* and amounts to a "optimize then discretize" approach ([18]). Finally, problem (29) can be tackled by *direct methods* that consider finite dimensional approximations of the original infinite dimensional problem (29). Direct methods amount to a "discretize then optimize" approach ([14]), they can further be split into different approaches. First, one may consider a finite representation of the continuous control $u(t)$ as piecewise constant functions whose values $q_1, \ldots, q_\tau$ at each piece define the finite number of degrees of freedom. The problem still involves an ODE in the state variable, $\dot{x}(t) = f(x(t), u_{q_{1:\tau}}(t))$, albeit a simpler one. Tackling the problem with such a partial discretization is referred to as a *single shooting* method ([8, 14]). *Collocation methods* ([58]) consider discretizing both the states and controls, leading to a formulation like (1), that can benefit from advanced numerical integration methods. Finally, *multiple shooting* ([8, 14]) combines both approaches. The system is split in multiple windows and for each window a single shooting method is used. We focus solely on the resulting discrete time nonlinear control problems (1) and refer the interested reader to, e.g., [14] for an overview of the approaches mentioned above.

One of the first approaches for nonlinear discrete time control problems (1) appear to be the Differential Dynamic Programming (DDP) methods developed by [26] and further explored by [31, 36, 38]. [8] referred to such approaches as *direct multiple shooting*. Numerous variants of DDP have been developed to account for constraints or noise in the dynamics ([20, 30, 52, 54]).

An implementation of a Newton method for nonlinear control problems of the form (1) was developed after the DDP approach by [16, 42]. A parallel implementation of a Newton step and sequential quadratic programming methods were developed by [61, 62], which led to efficient implementations of interior point methods for linear quadratic control problems under constraints by using the block band diagonal structure of the system of KKT equations solved at each step ([63]). A detailed comparison of the DDP approach and the Newton method was conducted by [32], who observed that the original DDP approach generally outperforms its Newton counterpart. We extend this analysis by comparing regularized variants of the algorithms. Finally, the storage of second order information for DDP and Newton can be alleviated with a careful implementation in a differentiable programming framework as done in our implementation and noted earlier by [40].

Simpler approaches consisting in taking linear approximations of the dynamics and quadratic approximations of the costs were implemented as part of public software ([55]). Two variants have been presented. The Iterative Linear Quadratic Regulator (ILQR) algorithm as originally formulated by [30] amounts naturally to a Gauss–Newton method ([50]). A variant that mixes linear quadratic approximations of the problem with a DDP approach, named iterative Linear Quadratic Regulator (iLQR) was further analyzed empirically by [53]. Here, we detail the line-searches for both approaches and present their regularized variants. We provide detailed computational complexities of all aforementioned algorithms that illustrate the trade-offs between the approaches.

Nonlinear model predictive control methods generally use sparse linear algebra solvers at each iteration ([15]) using solvers like IPOPT ([59]) or SNOPT ([22]). For offline control problems like (1), such sparse linear algebra solvers can also be used to compute the Gauss–Newton or Newton oracles seen as the solutions of a linear quadratic problem with underlying sparse band diagonal structure as first observed by [61, 62]. These sparse linear algebra solvers are an alternative to the dynamic programming procedures, presented in this manuscript, that can be seen as solving Riccatti equations in discrete-time with finite horizon. On the other hand, these sparse linear algebra solvers cannot be used as a black-box to implement DDP methods since they output directly the control variables solutions of the subproblem and do not a priori give access to the policies. They can nevertheless be adapted to record policies ([27, 56]). In this manuscript, we cast both classical optimization oracles and DDP approaches in a common differentiable programming framework to highlight their common ground and discrepancies, which would not be possible from a purely algebraic viewpoint. We aim at comparing these approaches purely in terms of iterations to understand differences in behavior, and leave out the optimization of these implementations in specific frameworks, using e.g. sparse linear algebra solvers to implement each classical optimization oracle. This viewpoint was generalized to handle nonlinear inequalities in model predictive control ([15]) or even generic graphs of computations ([51]). Alternative methods cast as sequential quadratic programming techniques ([19, 25, 37, 57]) are also worth mentioning.

For our experiments, we adapted the bicycle model of a miniature car developed by [33] in Python. We provide an implementation in Python, available at `https://github.com/vroulet/ilqc` for further exploration of the algorithms. Similar implementations have been implemented in the trajax library ([12]). Numerous other packages exist to implement nonlinear control algorithms such as CasAdi ([2]), Pyomo ([13]), JumP ([17]), acados ([56]) that can take advantage of off-the-shelf interior point solvers such as IPOPT ([59]), or SNOPT ([22]). Recently, [3] developed a new solver for quadratic programs with linear constraints using augmented Lagrangian. This solver in turn led to new efficient nonlinear control algorithms such as prox-DDP ([27]).

C Proofs

This section gathers proofs of propositions given in the main text.

C.1 Linear Quadratic Control

Lemma 5. *For linear functions ℓ_t and quadratic functions q_t, c_{t+1} s.t. $q_t(x, \cdot\,) + c_{t+1}(\ell_t(x, \cdot\,))$ is strongly convex for any x, the procedure*

$$\mathrm{LQBP} : (\ell_t, q_t, c_{t+1}) \to \begin{pmatrix} c_t : x \to \min_{u\in\mathbb{R}^{n_u}} \{q_t(x,u) + c_{t+1}(\ell_t(x,u))\} \\ \pi_t : x \to \operatorname*{arg\,min}_{u\in\mathbb{R}^{n_u}} \{q_t(x,u) + c_{t+1}(\ell_t(x,u))\} \end{pmatrix},$$

can be implemented analytically as detailed in Algorithm 2.

Proof. Consider ℓ_t, q_t, c_{t+1} to be parameterized as $\ell_t(x,u) = Ax + Bu$, $q_t(x,u) = \frac{1}{2}x^\top Px + \frac{1}{2}u^\top Qu + x^\top Ru + p^\top x + q^\top u$, $c_{t+1}(x) = \frac{1}{2}x^\top J_{t+1}x + j_{t+1}^\top x + j_{t+1}^0$. The cost-to-go function at time t is

$$\begin{aligned} c_t(x) = {}& \frac{1}{2}x^\top Px + p^\top x + j_{t+1}^0 \\ & + \min_{u\in\mathbb{R}^{n_u}} \left\{ \frac{1}{2}(Ax+Bu)^\top J_{t+1}(Ax+Bu) + j_{t+1}^\top(Ax+Bu) + \frac{1}{2}u^\top Qu + x^\top Ru + q^\top u \right\}. \end{aligned}$$

Since $h(x,\cdot\,) + c_{t+1}(\ell(x,\cdot\,))$ is strongly convex, we have that $Q + B^\top J_{t+1}B \succ 0$. Therefore, the policy at time t is

$$\pi_t(x) = -(Q + B^\top J_{t+1}B)^{-1}[(R^\top + B^\top J_{t+1}A)x + q + B^\top j_{t+1}].$$

Using that $\min_{u\in\mathbb{R}^{n_u}} u^\top Mu/2 + m^\top x = -m^\top M^{-1}m/2$ where, here, $M = Q + B^\top J_{t+1}B$, $m = (R^\top + B^\top J_{t+1}A)x + q + B^\top j_{t+1}$, we get that the cost-to-go function at time t is given by

$$\begin{aligned} c_t(x) = {}& \frac{1}{2}x^\top \left(P + A^\top J_{t+1}A - (R + A^\top J_{t+1}B)(Q + B^\top J_{t+1}B)^{-1}(R^\top + B^\top J_{t+1}A)\right) x \\ & + \left(p + A^\top j_{t+1} - (R + A^\top J_{t+1}B)(Q + B^\top J_{t+1}B)^{-1}(q + B^\top j_{t+1})\right)^\top x \\ & - \frac{1}{2}(q + B^\top j_{t+1})^\top (Q + B^\top J_{t+1}B)^{-1}(q + B^\top j_{t+1}) + j_{t+1}^0. \end{aligned}$$

◀

Corollary 6. *Consider problem* (1) *such that for all $t \in \{0, \ldots, \tau-1\}$, f_t is linear, h_t is convex quadratic with $h_t(x,\cdot\,)$ strongly convex for any x, and h_τ is convex quadratic. Then, the solution of problem* (1) *is given by*

$$\boldsymbol{u}^* = \mathrm{DynProg}((f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, \mathrm{LQBP}),$$

with DynProg *as defined in* (5) *and* LQBP *implemented in Algorithm 2*

Proof. Note that at time $t \in \{0, \ldots, \tau-1\}$ for a given $x \in \mathbb{R}^{n_x}$, if c_{t+1} is convex, then $c_{t+1}(f_t(x,\cdot\,))$ is convex as the composition of a convex function and a linear function and $c_{t+1}(f_t(x,\cdot\,)) + h_t(x,\cdot\,)$ is then strongly convex as the sum of a convex and a strongly convex function. Moreover, $x, u \to c_{t+1}(f_t(x,u)) + h_t(x,u)$ is jointly convex since $x, u \to c_{t+1}(f_t(x,u))$ is the composition of a convex function with a linear function and h_t is convex by assumption. Therefore, $c_t : x \to \min_{u\in\mathbb{R}^{n_u}} c_{t+1}(f_t(x,u)) + h_t(x,u)$ is convex as the partial infimum of jointly convex function. In summary, at time $t \in \{0, \ldots, \tau-1\}$, if c_{t+1} is convex, then (i) $c_{t+1}(f_t(x,\cdot\,)) + h_t(x,\cdot)$ is strongly convex, and (ii) c_t is convex. This ensures that the assumptions of Lemma 1 are satisfied at each iteration of Algorithm 1 (line 4) since $c_\tau = h_\tau$ is convex. ◀

C.2 Oracle Decomposition

Before presenting the proof of Lemma 4, we present below a compact formulation of the first and second order information of $f^{[\tau]}$ with respect to the first and second order information of the dynamics $(f_t)_{t=0}^{\tau-1}$. The decomposition done in this lemma is reused for the proof of Lemma 4.

Lemma 7. *Consider the control $f^{[\tau]}$ of τ dynamics $(f_t)_{t=0}^{\tau-1}$ as defined in Definition 3 and an initial point $x_0 \in \mathbb{R}^{n_x}$. For $\boldsymbol{x} = (x_1; \ldots; x_\tau)$ and $\boldsymbol{u} = (u_0; \ldots; u_{\tau-1})$, define*

$$F(\boldsymbol{x}, \boldsymbol{u}) = (f_0(x_0, u_0); \ldots; f_{\tau-1}(x_{\tau-1}, u_{\tau-1})).$$

The gradient of the control $f^{[\tau]}$ of the dynamics $(f_t)_{t=0}^{\tau-1}$ on $\boldsymbol{u} \in \mathbb{R}^{\tau n_u}$ can be written

$$\nabla_{\boldsymbol{u}} f^{[\tau]}(x_0, \boldsymbol{u}) = \nabla_{\boldsymbol{u}} F(\boldsymbol{x}, \boldsymbol{u})(\mathrm{I} - \nabla_{\boldsymbol{x}} F(\boldsymbol{x}, \boldsymbol{u}))^{-1}.$$

The Hessian of the control $f^{[\tau]}$ of the dynamics $(f_t)_{t=0}^{\tau-1}$ on $\boldsymbol{u} \in \mathbb{R}^{\tau n_u}$ can be written

$$\nabla^2_{\boldsymbol{uu}} f^{[\tau]}(x_0, \boldsymbol{u}) = \nabla^2_{\boldsymbol{xx}} F(\boldsymbol{x}, \boldsymbol{u})[N, N, M] + \nabla^2_{\boldsymbol{uu}} F(\boldsymbol{x}, \boldsymbol{u})[\,\cdot\,, \cdot\,, M] + \nabla^2_{\boldsymbol{xu}} F(\boldsymbol{x}, \boldsymbol{u})[N, \cdot, M] + \nabla^2_{\boldsymbol{ux}} F(\boldsymbol{x}, \boldsymbol{u})[\cdot, N, M],$$

where $M = (\mathrm{I} - \nabla_{\boldsymbol{x}} F(\boldsymbol{x}, \boldsymbol{u}))^{-1}$ and $N = \nabla_{\boldsymbol{u}} f^{[\tau]}(x_0, \boldsymbol{u})^\top$.

Proof. Denote simply, for $\boldsymbol{u} \in \mathbb{R}^{\tau n_u}$, $\phi(\boldsymbol{u}) = f^{[\tau]}(x_0, \boldsymbol{u})$ with x_0 a fixed initial state. By definition, the function ϕ can be decomposed, for $\boldsymbol{u} \in \mathbb{R}^{\tau n_u}$, as $\phi(\boldsymbol{u}) = (\phi_1(\boldsymbol{u}); \ldots; \phi_\tau(\boldsymbol{u}))$, such that

$$\phi_{t+1}(\boldsymbol{u}) = f_t(\phi_t(\boldsymbol{u}), E_t^\top \boldsymbol{u}) \quad \text{for } t \in \{0, \ldots, \tau-1\}, \tag{30}$$

with $\phi_0(\boldsymbol{u}) = x_0$ and for $t \in \{0, \ldots, \tau-1\}$, $E_t = e_t \otimes \mathrm{I}_{n_u}$ is such that $E_t^\top \boldsymbol{u} = u_t$, with e_t the $t+1^{\text{th}}$ canonical vector in $\mathbb{R}^\tau$, $\otimes$ the Kronecker product and $\mathrm{I}_{n_u} \in \mathbb{R}^{n_u \times n_u}$ the identity matrix. By taking the derivative of (30), we get, denoting $x_t = \phi_t(\boldsymbol{u})$ for $t \in \{0, \ldots, \tau\}$ and using that $E_t^\top \boldsymbol{u} = u_t$,

$$\nabla \phi_{t+1}(\boldsymbol{u}) = \nabla \phi_t(\boldsymbol{u}) \nabla_{x_t} f_t(x_t, u_t) + E_t \nabla_{u_t} f_t(x_t, u_t) \quad \text{for } t \in \{0, \ldots, \tau-1\}.$$

So, for $\boldsymbol{v} = (v_0; \ldots; v_{\tau-1}) \in \mathbb{R}^{\tau n_u}$, denoting $\nabla \phi(\boldsymbol{u})^\top \boldsymbol{v} = (y_1; \ldots; y_\tau)$ s.t. $\nabla \phi_t(\boldsymbol{u})^\top \boldsymbol{v} = y_t$ for $t \in \{1, \ldots, \tau\}$, we have, with $y_0 = 0$,

$$y_{t+1} = \nabla_{x_t} f_t(x_t, u_t)^\top y_t + \nabla_{u_t} f_t(x_t, u_t)^\top v_t \quad \text{for } t \in \{0, \ldots, \tau-1\}. \tag{31}$$

Denoting $\boldsymbol{y} = (y_1; \ldots; y_\tau)$, we have then

$$(\mathrm{I} - A)\boldsymbol{y} = B\boldsymbol{v}, \quad \text{i.e.,} \quad \nabla \phi(\boldsymbol{u})^\top \boldsymbol{v} = (\mathrm{I} - A)^{-1} B \boldsymbol{v},$$

where $A = \sum_{t=1}^{\tau-1} e_t e_{t+1}^\top \otimes A_t$ with $A_t = \nabla_{x_t} f_t(x_t, u_t)^\top$ for $t \in \{1, \ldots, \tau-1\}$ and $B = \sum_{t=1}^{\tau} e_t e_t^\top \otimes B_{t-1}$ with $B_t = \nabla_{u_t} f_t(x_t, u_t)^\top$ for $t \in \{0, \ldots, \tau-1\}$, i.e.

$$A = \begin{pmatrix} 0 & A_1 & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ & & & \ddots & 0 \\ \vdots & & & \ddots & A_{\tau-1} \\ 0 & \ldots & & \ldots & 0 \end{pmatrix}, \quad B = \begin{pmatrix} B_0 & 0 & \ldots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \ldots & 0 & B_{\tau-1} \end{pmatrix}.$$

By definition of F in the claim, one easily check that $A = \nabla_{\boldsymbol{x}} F(\boldsymbol{x}, \boldsymbol{u})^\top$ and $B = \nabla_{\boldsymbol{u}} F(\boldsymbol{x}, \boldsymbol{u})^\top$. Therefore, we get

$$\nabla_{\boldsymbol{u}} f^{[\tau]}(x_0, \boldsymbol{u}) = \nabla \phi(\boldsymbol{u}) = \nabla_{\boldsymbol{u}} F(\boldsymbol{x}, \boldsymbol{u})(\mathrm{I} - \nabla_{\boldsymbol{x}} F(\boldsymbol{x}, \boldsymbol{u}))^{-1}.$$

For the Hessian, note that for $g : \mathbb{R}^d \to \mathbb{R}^p$, $f : \mathbb{R}^p \to \mathbb{R}$, $x \in \mathbb{R}^d$, we have $\nabla^2 (f \circ g)(x) = \nabla g(x) \nabla^2 f(x) \nabla g(x)^\top + \nabla^2 g(x)[\,\cdot\,, \cdot\,, \nabla f(x)] \in \mathbb{R}^{d \times d}$. If $f : \mathbb{R}^p \to \mathbb{R}^n$, we have

$$\nabla^2 (f \circ g)(x) = \nabla^2 f(x)[\nabla g(x)^\top, \nabla g(x)^\top, \cdot\,] + \nabla^2 g(x)[\,\cdot\,, \cdot\,, \nabla f(x)] \in \mathbb{R}^{d \times d \times n}.$$

Applying this on $f_t \circ g_t$ for $g_t(\boldsymbol{u}) = (\phi_t(\boldsymbol{u}), E_t^\top \boldsymbol{u})$, we get from (30), using that $\nabla g_t(\boldsymbol{u}) = (\nabla \phi_t(\boldsymbol{u}), E_t)$,

$$\begin{aligned} \nabla^2 \phi_{t+1}(\boldsymbol{u}) = {} & \nabla^2 \phi_t(\boldsymbol{u})[\,\cdot\,, \cdot\,, \nabla_{x_t} f_t(x_t, u_t)] \\ & + \nabla^2_{x_t x_t} f_t(x_t, u_t)[\nabla \phi_t(\boldsymbol{u})^\top, \nabla \phi_t(\boldsymbol{u})^\top, \cdot\,] + \nabla^2_{u_t u_t} f_t(x_t, u_t)[E_t^\top, E_t^\top, \cdot\,] \\ & + \nabla^2_{x_t u_t} f_t(x_t, u_t)[\nabla \phi_t(\boldsymbol{u})^\top, E_t^\top, \cdot\,] + \nabla^2_{u_t x_t} f_t(x_t, u_t)[E_t^\top, \nabla \phi_t(\boldsymbol{u})^\top, \cdot\,], \end{aligned}$$

for $t \in \{0, \ldots, \tau-1\}$, with $\nabla^2 \phi_0(\boldsymbol{u}) = 0$. Therefore, for $\boldsymbol{v} = (v_0; \ldots; v_{\tau-1}), \boldsymbol{w} = (w_0; \ldots; w_{\tau-1}) \in \mathbb{R}^{\tau n_u}$, $\boldsymbol{\mu} = (\mu_1; \ldots; \mu_\tau) \in \mathbb{R}^{\tau n_x}$, we get

$$\begin{aligned} \nabla^2 \phi(\boldsymbol{u})[\boldsymbol{v}, \boldsymbol{w}, \boldsymbol{\mu}] &= \sum_{t=0}^{\tau-1} \nabla^2 \phi_{t+1}(\boldsymbol{u})[\boldsymbol{v}, \boldsymbol{w}, \mu_{t+1}] \\ &= \sum_{t=0}^{\tau-1} \Big(\nabla^2_{x_t x_t} f_t(x_t, u_t)[y_t, z_t, \lambda_{t+1}] + \nabla^2_{u_t u_t} f_t(x_t, u_t)[v_t, w_t, \lambda_{t+1}] \\ &\qquad + \nabla^2_{x_t u_t} f_t(x_t, u_t)[y_t, w_t, \lambda_{t+1}] + \nabla^2_{u_t x_t} f_t(x_t, u_t)[v_t, z_t, \lambda_{t+1}] \Big), \end{aligned} \tag{32}$$

where $\boldsymbol{y} = (y_1; \ldots ; y_\tau) = \nabla\phi(\boldsymbol{u})^\top \boldsymbol{v}$, $\boldsymbol{z} = (z_1; \ldots ; z_\tau) = \nabla\phi(\boldsymbol{u})^\top \boldsymbol{w}$, with $y_0 = z_0 = 0$ and $\boldsymbol{\lambda} = (\lambda_1; \ldots ; \lambda_\tau) \in \mathbb{R}^{\tau n_x}$ is defined by

$$\lambda_t = \nabla_{x_t} f_t(x_t, u_t)\lambda_{t+1} + \mu_t \qquad \text{for } t \in \{1, \ldots , \tau - 1\}, \quad \lambda_\tau = \mu_\tau .$$

On the other hand, denoting $F_t(\boldsymbol{x}, \boldsymbol{u}) = f_t(x_t, u_t)$ for $t \in \{0, \ldots , \tau - 1\}$, the Hessian of F with respect to the variables $\boldsymbol{u}$ can be decomposed as

$$\nabla^2_{\boldsymbol{uu}} F(\boldsymbol{x}, \boldsymbol{u})[\boldsymbol{v}, \boldsymbol{w}, \boldsymbol{\lambda}] = \sum_{t=0}^{\tau-1} \nabla^2_{\boldsymbol{uu}} F_t(\boldsymbol{x}, \boldsymbol{u})[\boldsymbol{v}, \boldsymbol{w}, \lambda_{t+1}] = \sum_{t=0}^{\tau-1} \nabla^2_{u_t u_t} f_t(x_t, u_t)[v_t, w_t, \lambda_{t+1}].$$

The Hessian of F with respect to the variable $\boldsymbol{x}$ can be decomposed as

$$\nabla^2_{\boldsymbol{xx}} F(\boldsymbol{x}, \boldsymbol{u})[\boldsymbol{y}, \boldsymbol{z}, \boldsymbol{\lambda}] = \sum_{t=0}^{\tau-1} \nabla^2_{\boldsymbol{xx}} F_t(\boldsymbol{x}, \boldsymbol{u})[\boldsymbol{y}, \boldsymbol{z}, \lambda_{t+1}] = \sum_{t=1}^{\tau-1} \nabla^2_{x_t x_t} f_t(x_t, u_t)[y_t, z_t, \lambda_{t+1}].$$

Finally, the second cross-derivatives of F w.r.t. $\boldsymbol{x}$ and $\boldsymbol{u}$ can be decomposed as

$$\nabla^2_{\boldsymbol{xu}} F(\boldsymbol{x}, \boldsymbol{u})[\boldsymbol{y}, \boldsymbol{w}, \boldsymbol{\lambda}] = \sum_{t=0}^{\tau-1} \nabla^2_{\boldsymbol{xu}} F_t(\boldsymbol{x}, \boldsymbol{u})[\boldsymbol{y}, \boldsymbol{w}, \lambda_{t+1}] = \sum_{t=1}^{\tau-1} \nabla^2_{x_t u_t} f_t(x_t, u_t)[y_t, w_t, \lambda_{t+1}].$$

From (32), we then get

$$\nabla^2\phi(\boldsymbol{u})[\boldsymbol{v}, \boldsymbol{w}, \boldsymbol{\mu}] = \nabla^2_{\boldsymbol{xx}} F(\boldsymbol{x}, \boldsymbol{u})[\boldsymbol{y}, \boldsymbol{z}, \boldsymbol{\lambda}] + \nabla^2_{\boldsymbol{uu}} F(\boldsymbol{x}, \boldsymbol{u})[\boldsymbol{v}, \boldsymbol{w}, \boldsymbol{\lambda}] + \nabla^2_{\boldsymbol{xu}} F(\boldsymbol{x}, \boldsymbol{u})[\boldsymbol{y}, \boldsymbol{w}, \boldsymbol{\lambda}] + \nabla^2_{\boldsymbol{ux}} F(\boldsymbol{x}, \boldsymbol{u})[\boldsymbol{v}, \boldsymbol{z}, \boldsymbol{\lambda}].$$

Finally, by noting that $\boldsymbol{y} = (\nabla_{\boldsymbol{u}} F(\boldsymbol{x}, \boldsymbol{u})(\mathrm{I} - \nabla_{\boldsymbol{x}} F(\boldsymbol{x}, \boldsymbol{u}))^{-1})^\top \boldsymbol{v}$, $\boldsymbol{z} = (\nabla_{\boldsymbol{u}} F(\boldsymbol{x}, \boldsymbol{u})(\mathrm{I} - \nabla_{\boldsymbol{x}} F(\boldsymbol{x}, \boldsymbol{u}))^{-1})^\top \boldsymbol{w}$, and $\boldsymbol{\lambda} = (\mathrm{I} - \nabla_{\boldsymbol{x}} F(\boldsymbol{x}, \boldsymbol{u}))^{-1}\boldsymbol{\mu}$, the claim is shown. ◀

Lemma 8. *Consider a nonlinear dynamical problem summarized as*

$$\min_{\boldsymbol{u}\in\mathbb{R}^{\tau n_u}} h \circ g(\boldsymbol{u}), \quad \textit{where} \quad h(\boldsymbol{x}, \boldsymbol{u}) = \sum_{t=0}^{\tau-1} h_t(x_t, u_t) + h_\tau(x_\tau), \quad g(\boldsymbol{u}) = (f^{[\tau]}(\bar{x}_0, \boldsymbol{u}), \boldsymbol{u}),$$

with $f^{[\tau]}$ the control of τ dynamics $(f_t)_{t=0}^{\tau-1}$ as defined in Definition 3.

Let $\boldsymbol{u} = (u_0; \ldots ; u_{\tau-1})$ and $f^{[\tau]}(\bar{x}_0, \boldsymbol{u}) = (x_1; \ldots ; x_\tau)$. Gradient (10)*, Gauss–Newton* (11) *and Newton* (12) *oracles for $h \circ g$ amount to solving for $\boldsymbol{v}^* = (v_0^*; \ldots ; v_{\tau-1}^*)$ linear quadratic control problems of the form*

$$\begin{aligned} \min_{\substack{v_0,\ldots,v_{\tau-1}\in\mathbb{R}^{n_u}\\ y_0,\ldots,y_\tau\in\mathbb{R}^{n_x}}} \quad & \sum_{t=0}^{\tau-1} q_t(y_t, v_t) + q_\tau(y_\tau) \\ \textit{subject to} \quad & y_{t+1} = \ell_{f_t}^{x_t,u_t}(y_t, v_t) \quad \textit{for } t \in \{0, \ldots , \tau - 1\}, \quad y_0 = 0, \end{aligned}$$

where for

i. *the gradient oracle* (10)*, $q_\tau(y_\tau) = \ell_{h_\tau}^{x_\tau}(y_\tau)$ and, for $0 \le t \le \tau - 1$,*

$$q_t(y_t, v_t) = \ell_{h_t}^{x_t,u_t}(y_t, v_t) + \frac{\nu}{2}\|v_t\|_2^2,$$

ii. *the Gauss–Newton oracle* (11)*, $q_\tau(y_\tau) = q_{h_\tau}^{x_\tau}(y_\tau)$ and, for $0 \le t \le \tau - 1$,*

$$q_t(y_t, v_t) = q_{h_t}^{x_t,u_t}(y_t, v_t) + \frac{\nu}{2}\|v_t\|_2^2,$$

iii. *for the Newton oracle* (12)*, $q_\tau(y_\tau) = q_{h_\tau}^{x_\tau}(y_\tau)$ and, defining*

$$\lambda_\tau = \nabla h_\tau(x_\tau), \quad \lambda_t = \nabla_{x_t} h_t(x_t, u_t) + \nabla_{x_t} f_t(x_t, u_t)\lambda_{t+1} \quad \textit{for } t \in \{\tau - 1, \ldots , 1\},$$

we have, for $0 \le t \le \tau - 1$,

$$q_t(y_t, v_t) = q_{h_t}^{x_t,u_t}(y_t, v_t) + \frac{1}{2}\nabla^2 f_t(x_t, u_t)[\,\cdot\,, \cdot\,, \lambda_{t+1}](y_t, v_t) + \frac{\nu}{2}\|v_t\|_2^2,$$

where for $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x}$, $x \in \mathbb{R}^{n_x}$, $u \in \mathbb{R}^{n_u}$, $\lambda \in \mathbb{R}^{n_x}$, we define

$$\nabla^2 f(x, u)[\,\cdot\,, \cdot\,, \lambda] : (y, v) \to \nabla^2_{xx} f(x, u)[y, y, \lambda] + 2\nabla^2_{xu} f(x, u)[y, v, \lambda] + \nabla^2_{uu} f(x, u)[v, v, \lambda].$$

Proof. In the following, we denote for simplicity $\phi(\boldsymbol{u}) = f^{[\tau]}(\bar{x}_0, \boldsymbol{u})$. The optimization oracles can be rewritten as follows.

1. The gradient oracle (10) is given by
$$v^* = \operatorname*{arg\,min}_{\boldsymbol{v}\in\mathbb{R}^{\tau n_u}} \left\{ \nabla h(g(\boldsymbol{u}))^\top \nabla g(\boldsymbol{u})^\top \boldsymbol{v} + \frac{\nu}{2}\|\boldsymbol{v}\|_2^2 \right\}. \tag{33}$$
2. The Gauss–Newton oracle (11) is given by
$$v^* = \operatorname*{arg\,min}_{\boldsymbol{v}\in\mathbb{R}^{\tau n_u}} \left\{ \frac{1}{2}\boldsymbol{v}^\top \nabla g(\boldsymbol{u})\nabla^2 h(g(\boldsymbol{u}))\nabla g(\boldsymbol{u})^\top \boldsymbol{v} + \nabla h(g(\boldsymbol{u}))^\top \nabla g(\boldsymbol{u})^\top \boldsymbol{v} + \frac{\nu}{2}\|\boldsymbol{v}\|_2^2 \right\}. \tag{34}$$
3. The Newton oracle (12) is given by
$$v^* = \operatorname*{arg\,min}_{\boldsymbol{v}\in\mathbb{R}^{\tau n_u}} \left\{ \frac{1}{2}\boldsymbol{v}^\top \nabla g(\boldsymbol{u})\nabla^2 h(g(\boldsymbol{u}))\nabla g(\boldsymbol{u})^\top \boldsymbol{v} + \frac{1}{2}\nabla^2 g(\boldsymbol{u})[\boldsymbol{v},\boldsymbol{v},\nabla h(g(\boldsymbol{u}))] + \nabla h(g(\boldsymbol{u}))^\top \nabla g(\boldsymbol{u})^\top \boldsymbol{v} + \frac{\nu}{2}\|\boldsymbol{v}\|_2^2 \right\}. \tag{35}$$

We have, denoting $\boldsymbol{x} = \phi(\boldsymbol{u})$,
$$\begin{aligned}
\nabla h(g(\boldsymbol{u}))^\top \nabla g(\boldsymbol{u})^\top \boldsymbol{v} &= \nabla_{\boldsymbol{x}} h(\boldsymbol{x},\boldsymbol{u})^\top \nabla\phi(\boldsymbol{u})^\top \boldsymbol{v} + \nabla_{\boldsymbol{u}} h(\boldsymbol{x},\boldsymbol{u})^\top \boldsymbol{v} \\
\boldsymbol{v}^\top \nabla g(\boldsymbol{u})\nabla^2 h(g(\boldsymbol{u}))\nabla g(\boldsymbol{u})^\top \boldsymbol{v} &= \boldsymbol{v}^\top \nabla\phi(\boldsymbol{u})\nabla^2_{\boldsymbol{xx}} h(\boldsymbol{x},\boldsymbol{u})\nabla\phi(\boldsymbol{u})^\top \boldsymbol{v} + \boldsymbol{v}^\top \nabla^2_{\boldsymbol{uu}} h(\boldsymbol{x},\boldsymbol{u})\boldsymbol{v} + 2\boldsymbol{v}^\top \nabla\phi(\boldsymbol{u})\nabla^2_{\boldsymbol{xu}} h(\boldsymbol{x},\boldsymbol{u})\boldsymbol{v} \\
\nabla^2 g(\boldsymbol{u})[\boldsymbol{v},\boldsymbol{v},\nabla h(g(\boldsymbol{u}))] &= \nabla^2\phi(\boldsymbol{u})[\boldsymbol{v},\boldsymbol{v},\nabla_{\boldsymbol{x}} h(\boldsymbol{x},\boldsymbol{u})].
\end{aligned}$$

For $\boldsymbol{v} = (v_0;\ldots;v_{\tau-1}) \in \mathbb{R}^{\tau n_u}$, denoting $\boldsymbol{y} = \nabla\phi(\boldsymbol{u})^\top \boldsymbol{v} = (y_1;\ldots;y_\tau)$, with $y_0 = 0$, we have then
$$\begin{aligned}
\nabla h(g(\boldsymbol{u}))^\top \nabla g(\boldsymbol{u})^\top \boldsymbol{v} &= \sum_{t=0}^{\tau-1} \left[\nabla_{x_t} h_t(x_t,u_t)^\top y_t + \nabla_{u_t} h_t(x_t,u_t)^\top v_t\right] + \nabla h_\tau(x_\tau)^\top y_\tau && (36)\\
&= \sum_{t=0}^{\tau-1} \ell_{h_t}^{x_t,u_t}(y_t,v_t) + \ell_{h_\tau}^{x_\tau}(y_\tau). && (37)
\end{aligned}$$

Following the proof of Lemma 7, we have that $\boldsymbol{y} = \nabla\phi(\boldsymbol{u})^\top \boldsymbol{v} = (y_1;\ldots;y_\tau)$ satisfies
$$y_{t+1} = \nabla_{x_t} f_t(x_t,u_t)^\top y_t + \nabla_{u_t} f_t(x_t,u_t)^\top v_t = \ell_{f_t}^{x_t,u_t}(y_t,v_t), \quad \text{for } t\in\{0,\ldots,\tau-1\}, \tag{38}$$
with $y_0 = 0$. Hence, plugging (37) and (38) into (33) we get the claim for the gradient oracle.

The Hessians of the total cost are block diagonal with, e.g., $\nabla^2_{\boldsymbol{uu}} h(\boldsymbol{x},\boldsymbol{u})$ being composed of τ diagonal blocks of the form $\nabla^2_{u_t u_t} h_t(x_t,u_t)$ for $t\in\{0,\ldots,\tau-1\}$. Therefore, we have
$$\begin{aligned}
&\frac{1}{2}\boldsymbol{v}^\top \nabla g(\boldsymbol{u})\nabla^2 h(g(\boldsymbol{u}))\nabla g(\boldsymbol{u})^\top \boldsymbol{v} \\
&= \sum_{t=0}^{\tau-1}\left[\frac{1}{2}y_t^\top \nabla^2_{x_t x_t} h_t(x_t,u_t) y_t + \frac{1}{2}v_t^\top \nabla^2_{u_t u_t} h_t(x_t,u_t) v_t + y_t^\top \nabla^2_{x_t u_t} h_t(x_t,u_t) v_t\right] + \frac{1}{2}y_\tau^\top \nabla^2 h_\tau(x_\tau) y_\tau.
\end{aligned}$$

The linear quadratic approximation in (34) can then be written as
$$\frac{1}{2}\boldsymbol{v}^\top \nabla g(\boldsymbol{u})\nabla^2 h(g(\boldsymbol{u}))\nabla g(\boldsymbol{u})^\top \boldsymbol{v} + \nabla h(g(\boldsymbol{u}))^\top \nabla g(\boldsymbol{u})^\top \boldsymbol{v} = \sum_{t=0}^{\tau-1} q_{h_t}^{x_t,u_t}(y_t,v_t) + q_{h_\tau}^{x_\tau}(y_\tau). \tag{39}$$

Hence, plugging (39) and (38) into (34) we get the claim for the Gauss–Newton oracle.

For the Newton oracle, denoting $\boldsymbol{\mu}{=}\nabla_{\boldsymbol{x}} h(\boldsymbol{x},\boldsymbol{u}) = (\nabla_{x_1} h_1(x_1,u_1);\ldots;\nabla_{x_{\tau-1}} h_{\tau-1}(x_{\tau-1},u_{\tau-1});\nabla h_\tau(x_\tau))$, and defining adjoint variables λ_t as
$$\lambda_\tau = \nabla h_\tau(x_\tau) \qquad \lambda_t = \nabla_{x_t} h_t(x_t,u_t) + \nabla_{x_t} f_t(x_t,u_t)\lambda_{t+1} \qquad \text{for } t\in\{1,\ldots,\tau-1\},$$
we have, as in the proof of Lemma 7,
$$\begin{aligned}
\nabla^2\phi(\boldsymbol{u})[\boldsymbol{v},\boldsymbol{v},\nabla_{\boldsymbol{x}} h(\boldsymbol{x},\boldsymbol{u})] &= \sum_{t=0}^{\tau-1} \nabla^2\phi_{t+1}(\boldsymbol{u})[\boldsymbol{v},\boldsymbol{v},\mu_{t+1}] \\
&= \sum_{t=0}^{\tau-1}\Big(\nabla^2_{x_t x_t} f_t(x_t,u_t)[y_t,y_t,\lambda_{t+1}] + \nabla^2_{u_t u_t} f_t(x_t,u_t)[v_t,v_t,\lambda_{t+1}] \\
&\qquad + 2\nabla^2_{x_t u_t} f_t(x_t,u_t)[y_t,v_t,\lambda_{t+1}]\Big). \qquad (40)
\end{aligned}$$

Hence, plugging (39), (40) and (38) into (35) we get the claim for the Newton oracle. ◀

D Line-search

So far, we defined procedures that, given a command and some regularization parameter, output a direction that minimizes an approximation of the objective or approximately minimizes a shifted objective. Given access to such procedures, the next command can be computed in several ways. The main criterion is to ensure that the value of the objective decreases along the iterations, which is generally done by a line-search.

In the following, we only consider oracles based on linear quadratic or quadratic approximations of the objective such as Gauss–Newton and Newton, and refer the reader to [41] for classical line-searches for gradient descent.

D.1 Rule

We start by considering the implementation of line-searches for classical optimization oracles which can again exploit the dynamical structure of the problem and are mimicked by differential dynamic programming approaches. We consider, as in Section 3, that we have access to an oracle for an objective $\mathcal{J}$, that, given a command $\boldsymbol{u} \in \mathbb{R}^{\tau n_u}$ and any regularization $\nu \geq 0$, outputs

$$\operatorname{Oracle}_\nu(\mathcal{J})(\boldsymbol{u}) = \operatorname*{arg\,min}_{\boldsymbol{v}\in\mathbb{R}^{\tau n_u}} m_{\mathcal{J}}^{\boldsymbol{u}}(\boldsymbol{v}) + \frac{\nu}{2}\|\boldsymbol{v}\|_2^2, \tag{41}$$

where $m_{\mathcal{J}}^{\boldsymbol{u}}$ is a linear quadratic or quadratic expansion of the objective $\mathcal{J}$ around $\boldsymbol{u}$ s.t. $\mathcal{J}(\boldsymbol{u}+\boldsymbol{v}) \approx \mathcal{J}(\boldsymbol{u})+m_{\mathcal{J}}^{\boldsymbol{u}}(\boldsymbol{v})$. Given such an oracle, we can define a new candidate command that decreases the value of the objective in several ways.

Directional Step

The next iterate can be defined along the direction provided by the oracle, as long as this direction is a descent direction. Namely, the next iterate can be computed as

$$\boldsymbol{u}^{\text{next}} = \boldsymbol{u} + \gamma\boldsymbol{v}, \quad \text{with } \boldsymbol{v} = \operatorname{Oracle}_\nu(\mathcal{J})(\boldsymbol{u}) \text{ for } \nu \geq 0 \text{ s.t. } \nabla\mathcal{J}(\boldsymbol{u})^\top \boldsymbol{v} < 0, \tag{42}$$

where the stepsize γ is chosen to satisfy, e.g., an Armijo condition ([41, Chapter 3]), that is,

$$\mathcal{J}(\boldsymbol{u}+\gamma\boldsymbol{v}) \leq \mathcal{J}(\boldsymbol{u}) + \frac{\gamma}{2}\nabla\mathcal{J}(\boldsymbol{u})^\top\boldsymbol{v}. \tag{43}$$

In this case, the search is usually initialized at each step with $\gamma = 1$. If condition (43) is not satisfied for $\gamma = 1$, the stepsize is decreased by a factor $\rho_{\text{dec}} < 1$ until condition (43) is satisfied. If a stepsize $\gamma = 1$ is accepted, then the linear quadratic or quadratic algorithms may exhibit a quadratic local convergence ([41, Chapter 3, 10]). Alternative line-search criterions such as Wolfe's curvature condition or trust-region methods can also be implemented ([41, Chapter 3]).

Regularized Step

Given a current iterate $\boldsymbol{u} \in \mathbb{R}^{\tau n_u}$, we can find a regularization such that the current iterate plus the direction output by the oracle decreases the objective. Namely, the next command can be computed as

$$\boldsymbol{u}^{\text{next}} = \boldsymbol{u} + \boldsymbol{v}^\gamma, \quad \text{where} \quad \boldsymbol{v}^\gamma = \operatorname{Oracle}_{1/\gamma}(\mathcal{J})(\boldsymbol{u}) = \operatorname*{arg\,min}_{\boldsymbol{v}\in\mathbb{R}^{\tau n_u}} m_{\mathcal{J}}^{\boldsymbol{u}}(\boldsymbol{v}) + \frac{1}{2\gamma}\|\boldsymbol{v}\|_2^2, \tag{44}$$

where the parameter $\gamma > 0$ acts as a stepsize that controls how large should be the step (the smaller the parameter γ, the smaller the step $\boldsymbol{v}^\gamma$). The stepsize γ can then be chosen to satisfy

$$\mathcal{J}(\boldsymbol{u}+\boldsymbol{v}^\gamma) \leq \mathcal{J}(\boldsymbol{u}) + m_{\mathcal{J}}^{\boldsymbol{u}}(\boldsymbol{v}^\gamma) + \frac{1}{2\gamma}\|\boldsymbol{v}^\gamma\|_2^2, \tag{45}$$

which ensures a sufficient decrease of the objective to, e.g., prove convergence to stationary points ([47]). In practice, as for the line-search on the descent direction, given an initial stepsize for the iteration, the stepsize is either selected or reduced by a factor ρ_{dec} until condition (45) is satisfied. However, here, we initialize the stepsize at each iteration as $\rho_{\text{inc}}\gamma_{prev}$ where γ_{prev} is the stepsize selected at the previous iteration and $\rho_{\text{inc}} > 1$ is an increasing factor. By trying a larger stepsize at each iteration, we may benefit from larger steps in some regions of the optimization path. Note that such an approach is akin to trust region methods which increase the radius of the trust region at each iteration depending on the success of each iteration ([41]).

In practice, we observed that, when using regularized steps, acceptable stepsizes for condition (45) tend to be arbitrarily large as the iterations increase. Namely, we tried choosing $\rho_{\text{inc}} = 10$ and observed that the acceptable stepsizes tended to plus infinity with such a procedure. To better capture this tendency, we consider regularizations that may depend on the current state and of the form $\nu(\boldsymbol{u}) \propto \|\nabla h(\boldsymbol{x}, \boldsymbol{u})\|_2$, i.e., stepsizes of the form $\gamma(\boldsymbol{u}) = \bar{\gamma}/\|\nabla h(\boldsymbol{x}, \boldsymbol{u})\|_2$. The line-search is then performed on $\bar{\gamma}$ only. Intuitively, as we are getting closer to a stationary point, quadratic models are getting more accurate to describe the objective. By scaling the regularization with respect to $\|\nabla h(\boldsymbol{x}, \boldsymbol{u})\|_2$, which is a measure of stationarity, we may better capture such behavior. Note that for $\nu = 0$, we retrieve the iteration with a descent direction of stepsize $\gamma = 1$ described above.

D.2 Implementation

Directional Step

The Armijo condition (43) can be computed directly from the knowledge of a gradient oracle and the chosen oracle (such as Gauss–Newton or Newton). We present here the implementation of the line-search in terms of the dynamical structure of the problem. Denote

$$(\pi_t)_{t=0}^{\tau-1}, c_0 = \text{Backward}((m_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (m_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, m_{h_\tau}^{x_\tau}, \nu)$$

the policies and the value of the cost-to-go function output by the backward pass of the considered oracle, i.e., Gauss–Newton or Newton.

By definition, $c_0(0)$ is the minimum of the corresponding linear quadratic control problem (13). Moreover, the linear quadratic control problem can be summarized as a quadratic problem of the form $\min_{\boldsymbol{v}} m_{\mathcal{J}}(\boldsymbol{v}) + \frac{\nu}{2}\|\boldsymbol{v}\|_2^2 = \min_{\boldsymbol{v}} \frac{1}{2}\boldsymbol{v}^\top(Q + \nu\,\mathrm{I})\boldsymbol{v} + \nabla\mathcal{J}(\boldsymbol{u})^\top\boldsymbol{v}$ with Q a quadratic that is either the Hessian of $\mathcal{J}$ for a Newton oracle or an approximation of it for a Gauss–Newton oracle. Therefore, we have that, for a Newton or a Gauss–Newton oracle $\boldsymbol{v} = \text{Oracle}_{1/\gamma}(\mathcal{J})$,

$$\frac{1}{2}\nabla\mathcal{J}(\boldsymbol{u})^\top\boldsymbol{v} = -\frac{1}{2}\nabla\mathcal{J}(\boldsymbol{u})^\top(Q + \nu\,\mathrm{I})^{-1}\nabla\mathcal{J}(\boldsymbol{u}) = \min_{\boldsymbol{v}\in\mathbb{R}^{\tau n_u}} m_{\mathcal{J}}(\boldsymbol{v}) + \frac{\nu}{2}\|\boldsymbol{v}\|_2^2 = c_0(0).$$

Therefore, the right-hand part of condition (43) can be given by the value of the cost-to-go function $c_0(0)$. On the other hand, sequences of controllers of the form $\gamma\boldsymbol{v}$ can be defined by modifying the policies output in the backward pass as shown in the following lemma adapted from [32, Theorem 1].

Lemma 9. *Given a sequence of affine policies $(\pi_t)_{t=0}^{\tau-1}$, linear dynamics $(\ell_t)_{t=0}^{\tau-1}$ and an initial state $y_0 = 0$, denote $\boldsymbol{v}^* = \text{Roll}(y_0, (\pi_t)_{t=0}^{\tau-1}, (\ell_t)_{t=0}^{\tau-1})$ and $\pi_t^\gamma : y \to \gamma\pi_t(0) + \nabla\pi_t(0)^\top y$ for $t = 0, \ldots, \tau-1$. We have that*

$$\gamma\boldsymbol{v}^* = \boldsymbol{v}^\gamma, \quad \textit{where} \quad \boldsymbol{v}^\gamma = \text{Roll}(y_0, (\pi_t^\gamma)_{t=0}^{\tau-1}, (\ell_t)_{t=0}^{\tau-1}).$$

Proof. Define $(y_t^\gamma)_{t=0}^{\tau-1}$ as $y_{t+1}^\gamma = \ell_t(y_t^\gamma, \pi_t(y_t^\gamma))$ for $t \in \{0, \ldots, \tau-1\}$ with $y_0^\gamma = 0$. We have that y_1^γ is linear w.r.t. γ. Proceeding by induction, we have that y_t^γ is linear w.r.t. γ using the form of π_t^γ and the fact that ℓ_t is linear. Therefore, $v_t^\gamma = \pi_t^\gamma(y_t^\gamma)$ is linear w.r.t. γ which gives the claim. ◀

Therefore, computing the next sequence of controllers by moving along a descent direction as in (42) according to an Armijo condition (43) amounts to computing, with Algorithm 17,

$$\boldsymbol{u}^{\text{next}} = \text{LineSearch}(\boldsymbol{u}, (h_t)_{t=0}^{\tau}, (f_t)_{t=0}^{\tau-1}, (\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, \text{Pol}),$$

$$\text{where} \quad \text{Pol} : \gamma \to \begin{pmatrix} (\pi_t^\gamma : & y \to \gamma\pi_t(0) + \nabla\pi_t(0)^\top y)_{t=0}^{\tau-1} \\ c_0^\gamma : & y \to \gamma c_0(y) \end{pmatrix}$$

$$(\pi_t)_{t=0}^{\tau-1}, c_0 = \text{Backward}((m_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (m_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, m_{h_\tau}^{x_\tau}, \nu), \text{ for } \nu \geq 0 \text{ s.t. } c_0(0) < 0,$$

where Backward $\in \{\text{Backward}_{\text{GN}}, \text{Backward}_{\text{NE}}\}$ is given in Algorithm 7 or Algorithm 8.

In practice, in our implementation of the backward passes in Algorithm 7, Algorithm 8, the returned initial cost-to-go function is either negative if the step is well-defined or infinite if it is not. To find a regularization that ensures a descent direction, i.e., $c_0(0) < 0$, it suffices thus to find a feasible step. In our implementation, we first try to compute a descent direction without regularization ($\nu = 0$), then try a small regularization $\nu = 10^{-6}$, which we increase by 10 until a finite negative cost-to-go function $c_0(0)$ is returned. See Algorithm 18 for an instance of such implementation.

From the above discussion, it is clear that one iteration of the Iterative Linear Quadratic Regulator algorithm described in Section 2.3 uses a Gauss–Newton oracle without regularization to move along the direction of the oracle by using an Armijo condition. The overall iteration is given in Algorithm 18, where we added a procedure to ensure that the output direction is a descent direction. All other algorithms, with or without regularization can be written in a similar way using a forward, a backward pass, and multiple roll-out phases until the next sequence of controllers is found.

Regularized Step

For regularized steps, the line-search (45) requires computing $m_{\mathcal{J}}^{\boldsymbol{u}}(v^\gamma) + \frac{1}{2\gamma}\|v^\gamma\|_2^2$. This is by definition the minimum of the sub-problem that is computed by dynamic programming. This minimum can therefore be accessed as $m_{\mathcal{J}}^{\boldsymbol{u}}(\boldsymbol{v}^\gamma) + \frac{1}{2\gamma}\|\boldsymbol{v}^\gamma\|_2^2 = c_0(0)$ for c_0 output by the backward pass with a regularization $\nu = 1/\gamma$. Overall, the next sequence of controls is then provided through the line-search procedure given in Algorithm 17 as

$$\begin{aligned}
&\boldsymbol{u}^{\text{next}} = \operatorname{LineSearch}(\boldsymbol{u}, (h_t)_{t=0}^{\tau}, (f_t)_{t=0}^{\tau-1}, (\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, \operatorname{Pol}),\\
&\text{where} \quad \operatorname{Pol} : \gamma \to \operatorname{Backward}((m_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (m_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, m_{h_\tau}^{x_\tau}, 1/\gamma),
\end{aligned}$$

where Backward $\in \{\text{Backward}_{\text{GN}}, \text{Backward}_{\text{NE}}\}$ is given in Algorithm 7 or Algorithm 8.

Line-search for Differential Dynamic Programming Approaches

The line-search for DDP approaches as presented by, e.g., [32, Section 2.2] based on [26], mimics the one done for the classical optimization oracles except that the policies are rolled out on the original dynamics. Namely, the usual line-search consists in applying Algorithm 17 as follows

$$\begin{aligned}
&\boldsymbol{u}^{\text{next}} = \operatorname{LineSearch}(\boldsymbol{u}, (h_t)_{t=0}^{\tau}, (f_t)_{t=0}^{\tau-1}, (\delta_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, \operatorname{Pol})\\
&\text{where} \quad \operatorname{Pol} : \gamma \to \begin{pmatrix} (\pi_t^\gamma : & y \to \gamma\pi_t(0) + \nabla \pi_t(0)^\top y)_{t=0}^{\tau-1}, \\ c_0^\gamma : & y \to \gamma c_0(y) \end{pmatrix}\\
&\qquad (\pi_t)_{t=0}^{\tau-1}, c_0 = \operatorname{Backward}((m_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (m_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, m_{h_\tau}^{x_\tau}, \nu) \text{ for } \nu \geq 0 \text{ s.t. } c_0(0) < 0,
\end{aligned}$$

where Backward $\in \{\text{Backward}_{\text{GN}}, \text{Backward}_{\text{DDP}}\}$ is given by Algorithm 7 or Algorithm 9. As for the classical optimization oracles, a direction is first computed without regularization and if the resulting direction is not a descent direction a small regularization is added to ensure that $c_0(0) < 0$.

We also consider line-searches based on selecting an appropriate regularization. Namely, we consider line-searches of the form

$$\begin{aligned}
&\boldsymbol{u}^{\text{next}} = \operatorname{LineSearch}(\boldsymbol{u}, (h_t)_{t=0}^{\tau}, (f_t)_{t=0}^{\tau-1}, (\delta_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, \operatorname{Pol}),\\
&\text{where} \quad \operatorname{Pol} : \gamma \to \operatorname{Backward}((m_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (m_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, m_{h_\tau}^{x_\tau}, 1/\gamma),
\end{aligned}$$

where Backward $\in \{\text{Backward}_{\text{GN}}, \text{Backward}_{\text{DDP}}\}$ is given by Algorithm 7 or Algorithm 9.

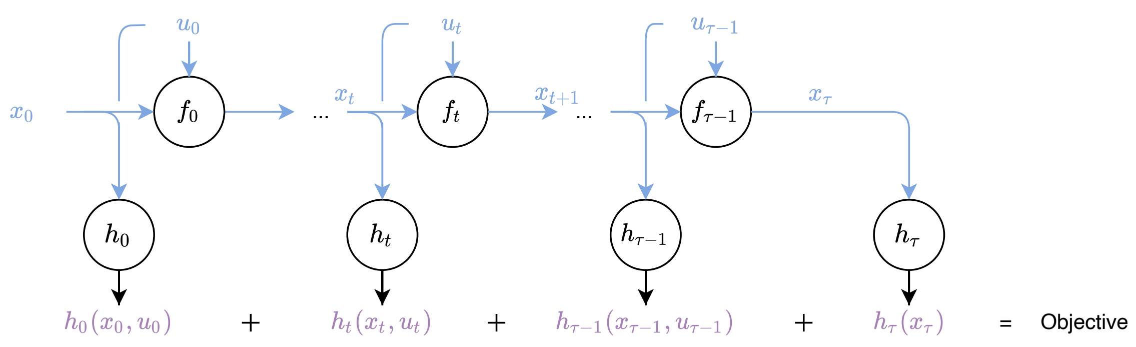
E Detailed Computational Scheme

We detail here the algorithms presented in Figure 1. Recall that our objective is

$$\begin{aligned}
\mathcal{J}(\boldsymbol{u}) = &\sum_{t=0}^{\tau-1} h_t(x_t, u_t) + h_\tau(x_\tau)\\
&\text{s.t.} \quad x_{t+1} = f_t(x_t, u_t) \quad \text{for } t \in \{0, \ldots, \tau-1\}, \quad x_0 = \bar{x}_0,
\end{aligned}$$

that can be summarized as $\mathcal{J}(\boldsymbol{u}) = h(g(\boldsymbol{u}))$, where, for $\boldsymbol{u} = (u_0; \ldots; u_{\tau-1})$, $\boldsymbol{x} = (x_1; \ldots; x_\tau)$,

$$h(\boldsymbol{x}, \boldsymbol{u}) = \sum_{t=0}^{\tau-1} h_t(x_t, u_t) + h_\tau(x_\tau), \; g(\boldsymbol{u}) = (f^{[\tau]}(\bar{x}_0, \boldsymbol{u}), \boldsymbol{u}), \; \begin{aligned} &f^{[\tau]}(x_0, \boldsymbol{u}) = (x_1; \ldots; x_\tau)\\ &\text{s.t.} \quad x_{t+1} = f_t(x_t, u_t) \quad \text{for } t \in \{0, \ldots, \tau-1\}.\end{aligned}$$



u_0
u_t
$u_{\tau-1}$
x_0
f_0
x_t
f_t
x_{t+1}
$f_{\tau-1}$
x_τ
h_0
h_t
$h_{\tau-1}$
h_τ
$h_0(x_0, u_0)$ + $h_t(x_t, u_t)$ + $h_{\tau-1}(x_{\tau-1}, u_{\tau-1})$ + $h_\tau(x_\tau)$ = Objective

Figure 4 Computational scheme of the discrete time control problem (1).

The computational graph of the objective is illustrated in Figure 4.

We present nonlinear control algorithms from a functional viewpoint by introducing finite difference, linear and quadratic expansions of the dynamics and the costs presented in the notations in (2).

For a function $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^p$, with $p = 1$ (for the costs) or $p = n_x$ (for the dynamics), these expansions read for $x, u \in \mathbb{R}^{n_x} \times \mathbb{R}^{n_u}$,

$$\delta_f^{x,u} : y, v \to f(x + y, u + v) - f(x, u), \qquad \ell_f^{x,u} : y, v \to \nabla_x f(x, u)^\top y + \nabla_u f(x, u)^\top v \tag{46}$$
$$q_f^{x,u} : y, v \to \nabla_x f(x, u)^\top y + \nabla_u f(x, u)^\top v + \frac{1}{2}\nabla^2_{xx} f(x, u)[y, y, \cdot\,] + \frac{1}{2}\nabla^2_{uu} f(x, u)[v, v, \cdot\,] + \nabla^2_{xu} f(x, u)[y, v, \cdot\,]$$

For $\lambda \in \mathbb{R}^p$, we denote shortly

$$\frac{1}{2}\nabla^2 f(x, u)[\,\cdot\,, \cdot\,, \lambda] : (y, v) \to \frac{1}{2}\nabla^2_{xx} f(x, u)[y, y, \lambda] + \frac{1}{2}\nabla^2_{uu} f(x, u)[v, v, \lambda] + \nabla^2_{xu} f(x, u)[y, v, \lambda].$$

In the algorithms, we consider storing in memory linear or quadratic functions by storing the associated vectors, matrices or tensors defining the linear or quadratic functions. For example, to store the linear expansion ℓ_f^x or the quadratic expansion q_f^x of a function $f : \mathbb{R}^d \to \mathbb{R}^p$ around a point x, we consider storing $\nabla f(x) \in \mathbb{R}^{d\times p}$ and $\nabla^2 f(x) \in \mathbb{R}^{d\times d\times p}$. In the backward or roll-out passes, we consider that having access to the linear or quadratic functions, means having access to the associated matrices/tensors defining the operations as presented in, e.g., Algorithm 2. The functional viewpoint helps to isolate the main technical operations in the procedures LQBP in Algorithm 2 or LBP in Algorithm 3 and to identify the discrepancies between, e.g., the Newton oracle in Algorithm 14 and a DDP oracle with quadratic approximations presented in Algorithm 16. For a presentation of the algorithms in a purely algebraic viewpoint, we refer the reader to, e.g., [32, 50, 61].

In Algorithms 7, 8, 9, we a priori need to check whether the subproblems defined by the Bellman recursion are strongly convex or not. Namely, in Algorithms 7, 8, 9, we need to check that $q_t(x, \cdot\,) + c_{t+1}(\ell_t(x, \cdot))$ is strongly convex for any x. With the notations of Algorithm 2, this amounts checking that $Q + B^\top J_{t+1} B \succ 0$. This can be done by checking the positivity of the minimum eigenvalue of $Q + B^\top J_{t+1} B$. In our implementation, we simply check that

$$j_t^0 - j_{t+1}^0 = -\frac{1}{2}(q + B^\top j_{t+1})^\top (Q + B^\top J_{t+1} B)^{-1}(q + B^\top j_{t+1}) < 0. \tag{47}$$

If condition (47) is not satisfied then necessarily $Q + B^\top J_{t+1} B \not\succ 0$. We chose to use condition (47) since this quantity is directly available and computing the eigenvalues of $Q + B^\top J_{t+1} B \succ 0$ can slow down the computations. Moreover, if criterion (47) is satisfied for all $t \in \{0, \ldots, \tau - 1\}$, this means that, for the Gauss–Newton and the Newton methods, the resulting direction is a descent direction for the objective. Algorithm 4 details the aforementioned verification step.

Algorithm 1 Dynamic programming procedure
$\big[\mathrm{DynProg} : (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar x_0, \mathrm{BP} \rightarrow (u_0^*; \ldots ; u_{\tau-1}^*)\big]$.

```
1: **Inputs**: Dynamics $(f_t)_{t=0}^{\tau-1}$, costs $(h_t)_{t=0}^{\tau}$, initial state $\bar x_0$, procedure BP
2: Initialize $c_\tau = h_\tau$
3: **for** $t = \tau - 1, \ldots, 0$ **do**
4:     Compute $c_t, \pi_t = \mathrm{BP}(f_t, h_t, c_{t+1})$, store $\pi_t$      ▷ BP = LQBP *(Algorithm 2) for linear quadratic control*
5: **end for**
6: Initialize $x_0^* = \bar x_0$
7: **for** $t = 0, \ldots, \tau - 1$ **do**
8:     Compute $u_t^* = \pi_t(x_t^*)$,   $x_{t+1}^* = f_t(x_t^*, u_t^*)$
9: **end for**
10: **Output:** Optimal command $\boldsymbol{u} = (u_0^*; \ldots ; u_{\tau-1}^*)$ for problem (1)
```

Algorithm 2 Analytic solution of Bellman's equation (6) for linear dynamics, quadratic costs
$\big[\mathrm{LQBP} : \ell_t, q_t, c_{t+1} \rightarrow c_t, \pi_t\big]$

```
1: **Inputs:**
     1. Linear function $\ell_t$ parameterized as$\ell_t(x, u) = A_t x + B_t u$
     2. Quadratic function $q_t$ parameterized as $q_t(x, u) = \frac{1}{2} x^\top P_t x + \frac{1}{2} u^\top Q_t u + x^\top R_t u + p_t^\top x + q_t^\top u$
     3. Quadratic function $c_{t+1}$ parameterized as $c_{t+1}(x) = \frac{1}{2} x^\top J_{t+1} x + j_{t+1}^\top x + j_{t+1}^0$
2: Define the cost-to-go function $c_t : x \rightarrow \frac{1}{2} x^\top J_t x + j_t^\top x + j_t^0$ with
     $J_t = P_t + A_t^\top J_{t+1} A_t - (R_t + A_t^\top J_{t+1} B_t)(Q_t + B_t^\top J_{t+1} B_t)^{-1}(R_t^\top + B_t^\top J_{t+1} A_t)$
     $j_t = p_t + A_t^\top j_{t+1} - (R_t + A_t^\top J_{t+1} B_t)(Q_t + B_t^\top J_{t+1} B_t)^{-1}(q_t + B_t^\top j_{t+1}),$
     $j_t^0 = j_{t+1}^0 - \frac{1}{2}(q_t + B_t^\top j_{t+1})^\top (Q_t + B_t^\top J_{t+1} B_t)^{-1}(q_t + B_t^\top j_{t+1})$
3: Define the policy $\pi_t : x \rightarrow K_t x + k_t$ with
     $K_t = -(Q_t + B_t^\top J_{t+1} B_t)^{-1}(R_t^\top + B_t^\top J_{t+1} A_t), \qquad k_t = -(Q_t + B_t^\top J_{t+1} B_t)^{-1}(q_t + B_t^\top j_{t+1})$
4: **Output:** Cost-to-go $c_t$ and policy $\pi_t$ at time $t$
```

Algorithm 3 Analytic solution of Bellman's equation (17) for linear dynamics, linear regularized costs
$\big[\mathrm{LBP} : \ell_t^f, \ell_t^h, c_{t+1}, \nu \rightarrow c_t, \pi_t\big]$

```
1: **Inputs:**
     1. Linear function $\ell_f$ parameterized as$\ell_t^f(x, u) = A_t x + B_t u$
     2. Linear function $\ell_h$ parameterized as $\ell_t^h(x, u) = p_t^\top x + q_t^\top u$
     3. Affine function $c_{t+1}$ parameterized as $c_{t+1}(x) = j_{t+1}^\top x + j_{t+1}^0$
     4. Regularization $\nu \geq 0$
2: Define $c_t : x \rightarrow j_t^\top x + j_t^0$ with $j_t = p_t + A_t^\top j_{t+1}$,  $j_t^0 = j_{t+1}^0 - \|q_t + B_t^\top j_{t+1}\|_2^2/(2\nu)$.
3: Define $\pi_t : x \rightarrow k_t$ with $k_t = -(q_t + B_t^\top j_{t+1})/\nu$.
4: **Output:** Cost-to-go $c_t$ and policy $\pi_t$ at time $t$
```

Algorithm 4 Check if subproblems given by $q_t(y,\cdot) + c_{t+1}(\ell_t(y,\cdot))$ are valid for solving Bellman's equation (6) [CheckSubProblem : $\ell_t, q_t, c_{t+1} \to$ valid $\in \{\mathrm{True}, \mathrm{False}\}$]

```
1: Option: Check strong convexity of subproblems or check only if the result gives a descent direction
2: Inputs:
    1. Linear function $\ell_t$ parameterized as$\ell_t(x,u) = A_t x + B_t u$,
    2. Quadratic function $q_t$ parameterized as $q_t(x,u) = \frac{1}{2}x^\top P_t x + \frac{1}{2}u^\top Q_t u + x^\top R_t u + p_t^\top x + q_t^\top u$
    3. Quadratic function $c_{t+1}$ parameterized as $c_{t+1}(x) = \frac{1}{2}x^\top J_{t+1} x + j_{t+1}^\top x + j_{t+1}^0$.
3: if check strong convexity then
4:     Compute the eigenvalues $\lambda_1 \leq \ldots \leq \lambda_{n_u}$ of $Q_t + B_t^\top J_{t+1} B_t$
5:     if $\lambda_1 > 0$ then valid = True else valid = False
6: else if check descent direction then
7:     Compute $j_t^0 - j_{t+1}^0 = -\frac{1}{2}(q_t + B_t^\top j_{t+1})^\top (Q_t + B_t^\top J_{t+1} B_t)^{-1}(q_t + B_t^\top j_{t+1})$
8:     if $j_t^0 - j_{t+1}^0 < 0$ then valid = True else valid = False
9: end if
10: Output: valid
```

Algorithm 5 Forward pass
[Forward : $\boldsymbol{u}, (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, o_f, o_h \to \mathcal{J}(\boldsymbol{u}), (m_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (m_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, m_{h_\tau}^{x_\tau}$]

```
1: Inputs: Command $\boldsymbol{u} = (u_0; \ldots; u_{\tau-1})$, dynamics $(f_t)_{t=0}^{\tau-1}$, costs $(h_t)_{t=0}^{\tau}$, initial state $\bar{x}_0$, order of the
   information to collect on the dynamics $o_f \in \{0,1,2\}$ and the costs $o_h \in \{0,1,2\}$
2: Initialize $x_0 = \bar{x}_0$, $\mathcal{J}(\boldsymbol{u}) = 0$
3: for $t = 0, \ldots \tau - 1$ do
4:     Compute $h_t(x_t, u_t)$, update $\mathcal{J}(\boldsymbol{u}) \leftarrow \mathcal{J}(\boldsymbol{u}) + h_t(x_t, u_t)$
5:     if $o_h \geq 1$ then Compute and store $\nabla h_t(x_t, u_t)$ defining $\ell_{h_t}^{x_t,u_t}$ as in (46)
6:     if $o_h = 2$ then Compute and store $\nabla^2 h_t(x_t, u_t)$ defining, with $\nabla h_t(x_t, u_t)$, $q_{h_t}^{x_t,u_t}$ as in (46)
7:     Compute $x_{t+1} = f_t(x_t, u_t)$
8:     if $o_f \geq 1$ then Compute and store $\nabla f_t(x_t, u_t)$ defining $\ell_{f_t}^{x_t,u_t}$ as in (46)
9:     if $o_f = 2$ then Compute and store $\nabla^2 f_t(x_t, u_t)$ defining, with $\nabla f_t(x_t, u_t)$, $q_{f_t}^{x_t,u_t}$ as in (46)
10: end for
11: Compute $h_\tau(x_\tau)$, update $\mathcal{J}(\boldsymbol{u}) \leftarrow \mathcal{J}(\boldsymbol{u}) + h_\tau(x_\tau)$
12: if $o_h \geq 1$ then Compute and store $\nabla h_\tau(x_\tau)$ defining $\ell_{h_\tau}^{x_\tau}$ as in (46)
13: if $o_h = 2$ then Compute and store $\nabla^2 h_\tau(x_\tau)$ defining, with $\nabla h_\tau(x_\tau)$, $q_{h_\tau}^{x_\tau}$ as in (46)
14: Output: Total cost $\mathcal{J}(\boldsymbol{u})$
15: Stored: (if $o_f$ and $o_h$ non-zeros) Approximations $(m_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (m_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, m_{h_\tau}^{x_\tau}$ defined by
```

$$m_{f_t}^{x_t,u_t} = \begin{cases} \ell_{f_t}^{x_t,u_t} & \text{if } o_f = 1 \\ q_{f_t}^{x_t,u_t} & \text{if } o_f = 2 \end{cases}, \quad m_{h_t}^{x_t,u_t} = \begin{cases} \ell_{h_t}^{x_t,u_t} & \text{if } o_h = 1 \\ q_{h_t}^{x_t,u_t} & \text{if } o_h = 2 \end{cases}, \quad m_{h_\tau}^{x_\tau} = \begin{cases} \ell_{h_\tau}^{x_\tau} & \text{if } o_h = 1 \\ q_{h_\tau}^{x_\tau} & \text{if } o_h = 2 \end{cases}$$

Algorithm 6 Backward pass for gradient oracle
[$\mathrm{Backward}_{\mathrm{GD}}$: $(\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (\ell_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, \ell_{h_\tau}^{x_\tau}, \nu) \to (\pi_t)_{t=0}^{\tau-1}, c_0$]

```
1: Inputs: Linear expansions of the dynamics $(\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}$, linear expansions of the costs $(\ell_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, \ell_{h_\tau}^{x_\tau}$,
   regularization $\nu > 0$
2: Initialize $c_\tau = \ell_{h_\tau}^{x_\tau}$
3: for $t = \tau - 1, \ldots 0$ do
4:     Define $\ell_t = \ell_{f_t}^{x_t,u_t}$,    $q_t : y_t, v_t \to \ell_{h_t}^{x_t,u_t}(y_t, v_t) + \frac{\nu}{2}\|v_t\|_2^2$
5:     Compute $c_t, \pi_t = \mathrm{LQBP}(\ell_t, q_t, c_{t+1}) = \mathrm{LBP}(\ell_{f_t}^{x_t,u_t}, \ell_{h_t}^{x_t,u_t}, c_{t+1}, \nu)$ where LBP is given in Algorithm 3
6: end for
7: Outputs: Policies $(\pi_t)_{t=0}^{\tau-1}$, cost-to-go function at initial time $c_0$
```

Algorithm 7 Backward pass for Gauss–Newton oracle
$\left[\mathrm{Backward}_{\mathrm{GN}} : (\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau}, \nu) \to (\pi_t)_{t=0}^{\tau-1}, c_0\right]$

```
1: Inputs: Linear expansions of the dynamics $(\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}$, quadratic expansions of the costs $(q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau}$,
   regularization $\nu \geq 0$
2: Initialize $c_\tau = q_{h_\tau}^{x_\tau}$
3: for $t = \tau - 1, \ldots 0$ do
4:     Define $\ell_t = \ell_{f_t}^{x_t,u_t}$,    $q_t : y_t, v_t \to q_{h_t}^{x_t,u_t}(y_t, v_t) + \frac{\nu}{2}\|v_t\|_2^2$,
5:     if CheckSubProblem($\ell_t, q_t, c_{t+1}$) is True then
6:         Compute $c_t, \pi_t = \mathrm{LQBP}(\ell_t, q_t, c_{t+1})$ with LQBP given in Algorithm 2
7:     else
8:         $\pi_s : x \to 0$ for $s \leq t$, $c_0 : x \to -\infty$, break
9:     end if
10: end for
11: Outputs: Policies $(\pi_t)_{t=0}^{\tau-1}$, cost-to-go function at initial time $c_0$
```

Algorithm 8 Backward pass for Newton oracle
$\left[\mathrm{Backward}_{\mathrm{NE}} : (q_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau}, \nu) \to (\pi_t)_{t=0}^{\tau-1}, c_0\right]$

```
1: Inputs: Quadratic expansions of the dynamics $(q_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}$, quadratic expansions of the costs $(q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau}$,
   regularization $\nu \geq 0$
2: Initialize $c_\tau = q_{h_\tau}^{x_\tau}$, $\lambda_\tau = \nabla h_\tau(x_\tau)$
3: for $t = \tau - 1, \ldots 0$ do
4:     Define $\ell_t = \ell_{f_t}^{x_t,u_t}$,    $q_t : (y_t, v_t) \to q_{h_t}^{x_t,u_t}(y_t, v_t) + \frac{\nu}{2}\|v_t\|_2^2 + \frac{1}{2}\nabla^2 f_t(x_t, u_t)[\cdot, \cdot, \lambda_{t+1}](y_t, v_t)$
5:     Compute $\lambda_t = \nabla_{x_t} h_t(x_t, u_t) + \nabla_{x_t} f_t(x_t, u_t)\lambda_{t+1}$
6:     if CheckSubProblem($\ell_t, q_t, c_{t+1}$) is True then
7:         Compute $c_t, \pi_t = \mathrm{LQBP}(\ell_t, q_t, c_{t+1})$ with LQBP given in Algorithm 2
8:     else
9:         $\pi_s : x \to 0$ for $s \leq t$, $c_0 : x \to -\infty$, break
10:     end if
11: end for
12: Outputs: Policies $(\pi_t)_{t=0}^{\tau-1}$, cost-to-go function at initial time $c_0$
```

Algorithm 9 Backward pass for a DDP approach with quadratic approximations
$\left[\mathrm{Backward}_{\mathrm{DDP}} : (q_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau}, \nu) \to (\pi_t)_{t=0}^{\tau-1}, c_0\right]$

```
1: Inputs: Quadratic expansions on the dynamics $(q_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}$, quadratic expansions on the costs $(q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau}$,
   regularization $\nu \geq 0$
2: Initialize $c_\tau = q_{h_\tau}^{x_\tau}$
3: for $t = \tau - 1, \ldots 0$ do
4:     Define $\ell_t = \ell_{f_t}^{x_t,u_t}$,    $q_t : y_t, v_t \to q_{h_t}^{x_t,u_t}(y_t, v_t) + \frac{\nu}{2}\|y_t\|_2^2 + \frac{1}{2}\nabla^2 f_t(x_t, u_t)[\cdot, \cdot, \nabla c_{t+1}(0)](y_t, v_t)$
5:     if CheckSubProblem($\ell_t, q_t, c_{t+1}$) is True then
6:         Compute $c_t, \pi_t = \mathrm{LQBP}(\ell_t, q_t, c_{t+1})$ with LQBP given in Algorithm 2
7:     else
8:         $\pi_s : x \to 0$ for $s \leq t$, $c_0 : x \to -\infty$, break
9:     end if
10: end for
11: Outputs: Policies $(\pi_t)_{t=0}^{\tau-1}$, cost-to-go function at initial time $c_0$
```

Algorithm 10 Backward pass for Newton oracle with function storage

```
1: Inputs: Stored functions $(f_t)_{t=0}^{\tau-1}$, costs $(h_t)_{t=0}^{\tau}$, inputs $(u_t)_{t=0}^{\tau-1}$ with associated trajectory $(x_t)_{t=0}^{\tau}$
2: Compute the quadratic expansion $q_{h_\tau}^{x_\tau}$ of the final cost and the derivative $\nabla h_\tau(x_\tau)$ of the final cost on $x_\tau$
3: Set $c_\tau = q_{h_\tau}^{x_\tau}$, $\lambda_\tau = \nabla h_\tau(x_\tau)$
4: for $t = \tau - 1, \ldots 0$ do
5:     Compute the linear approximation $\ell_{f_t}^{x_t,u_t}$ of the dynamic around $x_t, u_t$
6:     Compute the quadratic approximation $q_{h_t}^{x_t,u_t}$ of the cost around $x_t, u_t$
7:     Compute the Hessian of $x_t, u_t \to f_t(x_t, u_t)^\top \lambda_{t+1}$ on $x_t, u_t$ which gives $\frac{1}{2}\nabla^2 f_t(x_t, u_t)[\cdot, \cdot, \lambda_{t+1}]$.
8:     Define $\ell_t = \ell_{f_t}^{x_t,u_t}$,   $q_t : (y_t, v_t) \to q_{h_t}^{x_t,u_t}(y_t, v_t) + \frac{\nu}{2}\|v_t\|_2^2 + \frac{1}{2}\nabla^2 f_t(x_t, u_t)[\cdot, \cdot, \lambda_{t+1}](y_t, v_t)$
9:     Compute $\lambda_t = \nabla_{x_t} h_t(x_t, u_t) + \nabla_{x_t} f_t(x_t, u_t)\lambda_{t+1}$
10:    if CheckSubProblem$(\ell_t, q_t, c_{t+1})$ is True then
11:        Compute $c_t, \pi_t = \mathrm{LQBP}(\ell_t, q_t, c_{t+1})$
12:    else
13:        $\pi_s : x \to 0$ for $s \le t$, $c_0 : x \to -\infty$, break
14:    end if
15: end for
16: Outputs: Policies $(\pi_t)_{t=0}^{\tau-1}$, cost-to-go function at initial time $c_0$
```

Algorithm 11 Roll-out on dynamics
$\left[\mathrm{Roll} : y_0, (\pi_t)_{t=1}^{\tau-1}, (\phi_t)_{t=0}^{\tau-1} \to \boldsymbol{v}\right]$

```
1: Inputs: Initial state $y_0$, sequence of policies $(\pi_t)_{t=0}^{\tau-1}$, dynamics to roll-on $(\phi_t)_{t=0}^{\tau-1}$
2: for $t = 0, \ldots, \tau - 1$ do
3:     Compute and store $v_t = \pi_t(y_t)$,  $y_{t+1} = \phi_t(y_t, v_t)$.
4: end for
5: Output: Sequence of controllers $\boldsymbol{v} = (v_0; \ldots; v_{\tau-1})$
```

Algorithm 12 Gradient oracle
$\left[\mathrm{GD} : \boldsymbol{u}, (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, \nu \to \boldsymbol{v}\right]$

```
1: Inputs: Command $\boldsymbol{u} = (u_0; \ldots; u_{\tau-1})$, dynamics $(f_t)_{t=0}^{\tau-1}$, costs $(h_t)_{t=0}^{\tau}$, initial state $\bar{x}_0$, regularization $\nu>0$
2: Compute with Algorithm 5

   $\mathcal{J}(\boldsymbol{u}), (\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (\ell_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, \ell_{h_\tau}^{x_\tau} = \mathrm{Forward}(\boldsymbol{u}, (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, o_f = 1, o_h = 1)$

3: Compute with Algorithm 6

   $(\pi_t)_{t=0}^{\tau-1}, c_0 = \mathrm{Backward}_{\mathrm{GD}}((\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (\ell_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau}, \nu)$

4: Compute with Algorithm 11

   $\boldsymbol{v} = \mathrm{Roll}(0, (\pi_t)_{t=0}^{\tau-1}, (\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1})$

5: Output: Gradient direction $\boldsymbol{v} = \arg\min_{\tilde{\boldsymbol{v}} \in \mathbb{R}^{\tau n_u}} \left\{\ell_{h \circ g}^{\boldsymbol{u}}(\tilde{\boldsymbol{v}}) + \frac{\nu}{2}\|\tilde{\boldsymbol{v}}\|_2^2\right\} = -\nu^{-1}\nabla(h \circ g)(\boldsymbol{u})$
```

Algorithm 13 Gauss–Newton oracle (ILQR)
$\left[\mathrm{GN} : \boldsymbol{u}, (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, \nu \rightarrow \boldsymbol{v}\right]$

```
1: Inputs: Command $\boldsymbol{u}=(u_0; \ldots; u_{\tau-1})$, dynamics $(f_t)_{t=0}^{\tau-1}$, costs $(h_t)_{t=0}^{\tau}$, initial state $\bar{x}_0$, regularization $\nu \geq 0$
2: Compute with Algorithm 5
     $\mathcal{J}(\boldsymbol{u}), (\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau} = \mathrm{Forward}(\boldsymbol{u}, (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, o_f = 1, o_h = 2)$
3: Compute with Algorithm 7
     $(\pi_t)_{t=0}^{\tau-1}, c_0 = \mathrm{Backward}_{\mathrm{GN}}((\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau}, \nu)$
4: Compute with Algorithm 11
     $\boldsymbol{v} = \mathrm{Roll}(0, (\pi_t)_{t=0}^{\tau-1}, (\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1})$
5: Output: If $c_0(0) = +\infty$, returns infeasible, otherwise returns Gauss–Newton direction $\boldsymbol{v} = \arg\min_{\tilde{\boldsymbol{v}} \in \mathbb{R}^{\tau n_u}} \left\{ q_h^{g(\boldsymbol{u})}(\ell_g^{\boldsymbol{u}}(\tilde{\boldsymbol{v}})) + \frac{\nu}{2}\|\tilde{\boldsymbol{v}}\|_2^2 \right\} = -(\nabla g(\boldsymbol{u}) \nabla^2 h(\boldsymbol{x}, \boldsymbol{u}) \nabla g(\boldsymbol{u}) + \nu\,\mathrm{I})^{-1} \nabla (h \circ g)(\boldsymbol{u})$
```

Algorithm 14 Newton oracle
$\left[\mathrm{NE} : \boldsymbol{u}, (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, \nu \rightarrow \boldsymbol{v}\right]$

```
1: Inputs: Command $\boldsymbol{u}=(u_0; \ldots; u_{\tau-1})$, dynamics $(f_t)_{t=0}^{\tau-1}$, costs $(h_t)_{t=0}^{\tau}$, initial state $\bar{x}_0$, regularization $\nu \geq 0$
2: Compute with Algorithm 5
     $\mathcal{J}(\boldsymbol{u}), (q_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau} = \mathrm{Forward}(\boldsymbol{u}, (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, o_f = 2, o_h = 2)$
3: Compute with Algorithm 8
     $(\pi_t)_{t=0}^{\tau-1}, c_0 = \mathrm{Backward}_{\mathrm{NE}}((q_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau}, \nu)$
4: Compute with Algorithm 11
     $\boldsymbol{v} = \mathrm{Roll}(0, (\pi_t)_{t=0}^{\tau-1}, (\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1})$
5: Output: If $c_0(0) = +\infty$, returns infeasible, otherwise returns Newton direction $\boldsymbol{v} = \arg\min_{\tilde{\boldsymbol{v}} \in \mathbb{R}^{\tau n_u}} \left\{ q_{h \circ g}^{\boldsymbol{u}}(\tilde{\boldsymbol{v}}) + \frac{\nu}{2}\|\tilde{\boldsymbol{v}}\|_2^2 \right\} = -(\nabla^2 (h \circ g)(\boldsymbol{u}) + \nu\,\mathrm{I})^{-1} \nabla (h \circ g)(\boldsymbol{u})$
```

Algorithm 15 Differential dynamic programming oracle with linear quadratic approximations (iLQR)
$\left[\mathrm{DDP\text{-}LQ} : \boldsymbol{u}, (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, \nu \rightarrow \boldsymbol{v}\right]$

```
1: Inputs: Command $\boldsymbol{u}=(u_0; \ldots; u_{\tau-1})$, dynamics $(f_t)_{t=0}^{\tau-1}$, costs $(h_t)_{t=0}^{\tau}$, initial state $\bar{x}_0$, regularization $\nu \geq 0$
2: Compute with Algorithm 5
     $\mathcal{J}(\boldsymbol{u}), (\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau} = \mathrm{Forward}(\boldsymbol{u}, (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, o_f = 1, o_h = 2)$
3: Compute with Algorithm 7
     $(\pi_t)_{t=0}^{\tau-1}, c_0 = \mathrm{Backward}_{\mathrm{GN}}((\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau}, \nu)$
4: Compute with Algorithm 11, for $\delta_{f_t}^{x_t,u_t}(y_t, v_t) = f(x_t + y_t, u_t + v_t) - f(x_t, u_t)$,
     $\boldsymbol{v} = \mathrm{Roll}(0, (\pi_t)_{t=0}^{\tau-1}, (\delta_{f_t}^{x_t,u_t})_{t=0}^{\tau-1})$
5: Output: If $c_0(0) = +\infty$, returns infeasible, otherwise returns DDP oracle with linear-quadratic approximations $\boldsymbol{v}$
```

Algorithm 16 Differential dynamic programming oracle with quadratic approximations (DDP)
$\big[\text{DDP-Q} : \boldsymbol{u}, (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, \nu \to \boldsymbol{v}\big]$

```
1: **Inputs:** Command $\boldsymbol{u}=(u_0;\ldots;u_{\tau-1})$, dynamics $(f_t)_{t=0}^{\tau-1}$, costs $(h_t)_{t=0}^{\tau}$, initial state $\bar{x}_0$, regularization $\nu\geq 0$
2: Compute with Algorithm 5
     $\mathcal{J}(\boldsymbol{u}), (q_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau} = \text{Forward}(\boldsymbol{u}, (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, o_f = 2, o_h = 2)$
3: Compute with Algorithm 9
     $(\pi_t)_{t=0}^{\tau-1}, c_0 = \text{Backward}_{\text{DDP}}((q_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau}, \nu)$
4: Compute with Algorithm 11, for $\delta_{f_t}^{x_t,u_t}(y_t, v_t) = f(x_t + y_t, u_t + v_t) - f(x_t, u_t)$,
     $\boldsymbol{v} = \text{Roll}(0, (\pi_t)_{t=0}^{\tau-1}, (\delta_{f_t}^{x_t,u_t})_{t=0}^{\tau-1})$
5: **Output:** If $c_0(0) = +\infty$, returns infeasible, otherwise returns DDP oracle with quadratic approximations $\boldsymbol{v}$
```

Algorithm 17 Line-search
$\big[\text{LineSearch} : \boldsymbol{u}, (h_t)_{t=0}^{\tau}, (f_t)_{t=0}^{\tau-1}, (\phi_t)_{t=0}^{\tau-1}, (\text{Pol} : \gamma \to (\pi_t^\gamma)_{t=0}^{\tau-1}, c_0^\gamma) \to \boldsymbol{u}^{\text{next}}\big]$

```
1: **Option:** directional step or regularized step
2: **Inputs:** Current controls $\boldsymbol{u}$, costs $(h_t)_{t=0}^{\tau}$, initial state $\bar{x}_0$, original dynamics $(f_t)_{t=0}^{\tau-1}$, dynamics to roll out on $(\phi_t)_{t=0}^{\tau-1}$, family of policies and corresponding costs given by $\gamma \to (\pi_t^\gamma)_{t=0}^{\tau-1}, c_0^\gamma$, decreasing factor $\rho_{\text{dec}} \in (0,1)$, increasing factor $\rho_{\text{inc}} > 1$, previous stepsize $\gamma_{\text{prev}}$
3: Compute $\mathcal{J}(\boldsymbol{u}) = \text{Forward}(\boldsymbol{u}, (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, o_f = 0, o_h = 0)$
4: **if** directional step **then**
5:     Initialize $\gamma = 1$
6: **else if** regularized step **then**
7:     Compute $\nabla h(\boldsymbol{x}, \boldsymbol{u})$ for $\boldsymbol{x} = f^{[\tau]}(\bar{x}_0, \boldsymbol{u})$
8:     Initialize $\gamma = \rho_{\text{inc}}\gamma_{\text{prev}}/\|\nabla h(\boldsymbol{x}, \boldsymbol{u})\|_2$
9: **end if**
10: Initialize $y_0 = 0$, accept = False, minimal stepsize $\gamma_{\min} = 10^{-12}$
11: **while** not accept **do**
12:     Get $\pi_t^\gamma, c_0^\gamma = \text{Pol}(\gamma)$
13:     Compute $\boldsymbol{v}^\gamma = \text{Roll}(y_0, (\pi_t^\gamma)_{t=1}^{\tau-1}, (\phi_t)_{t=0}^{\tau-1})$
14:     Set $\boldsymbol{u}^{\text{next}} = \boldsymbol{u} + \boldsymbol{v}^\gamma$
15:     Compute $\mathcal{J}(\boldsymbol{u}^{\text{next}}) = \text{Forward}(\boldsymbol{u}^{\text{next}}, (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, o_f = 0, o_h = 0)$
16:     **if** $\mathcal{J}(\boldsymbol{u}^{\text{next}}) - \mathcal{J}(\boldsymbol{u}) \leq c_0^\gamma(0)$ **then** set accept = True **else** set $\gamma \to \rho_{\text{dec}}\gamma$
17:     **if** $\gamma \leq \gamma_{\min}$ **then break**
18: **end while**
19: **if** regularized step **then** $\gamma := \gamma\|\nabla h(\boldsymbol{x}, \boldsymbol{u})\|_2$
20: **Output:** Next sequence of controllers $\boldsymbol{u}^{\text{next}}$, store value of the stepsize selected $\gamma$
```

Algorithm 18 Iterative Linear Quadratic Regulator/Gauss–Newton step with line-search on descent directions

```
1: Inputs: Command $\boldsymbol{u}$, dynamics $(f_t)_{t=0}^{\tau-1}$, costs $(h_t)_{t=0}^{\tau-1}$, initial state $\bar{x}_0$
2: Compute with Algorithm 5
     $\mathcal{J}(\boldsymbol{u}), (\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau} = \mathrm{Forward}(\boldsymbol{u}, (f_t)_{t=0}^{\tau-1}, (h_t)_{t=0}^{\tau}, \bar{x}_0, o_f = 1, o_h = 2)$
3: Compute with Algorithm 7
     $(\pi_t)_{t=0}^{\tau-1}, c_0 = \mathrm{Backward}_{\mathrm{GN}}((\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau}, 0)$
4: Set $\nu = \nu_{\mathrm{init}}$ with, e.g., $\nu_{\mathrm{init}} = 10^{-6}$
5: while $c_0(0) = +\infty$ do
6:     Compute $(\pi_t)_{t=0}^{\tau-1}, c_0 = \mathrm{Backward}_{\mathrm{GN}}((\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, (q_{h_t}^{x_t,u_t})_{t=0}^{\tau-1}, q_{h_\tau}^{x_\tau}, \nu)$
7:     Set $\nu \to \rho_{\mathrm{inc}}\nu$ with, e.g., $\rho_{\mathrm{inc}} = 10$
8: end while
9: Define $\mathrm{Pol} : \gamma \to \begin{pmatrix} (\pi_t^\gamma : & y \to \gamma\pi_t(0) + \nabla\pi_t(0)^\top y)_{t=0}^{\tau-1}, \\ c_0^\gamma : & y \to \gamma c_0(y) \end{pmatrix}$
10: Compute with Algorithm 17
     $\boldsymbol{u}^{\mathrm{next}} = \mathrm{LineSearch}(\boldsymbol{u}, (h_t)_{t=0}^{\tau}, (f_t)_{t=0}^{\tau-1}, (\ell_{f_t}^{x_t,u_t})_{t=0}^{\tau-1}, \mathrm{Pol})$
11: Output: Next sequence of controllers $\boldsymbol{u}^{\mathrm{next}}$
```

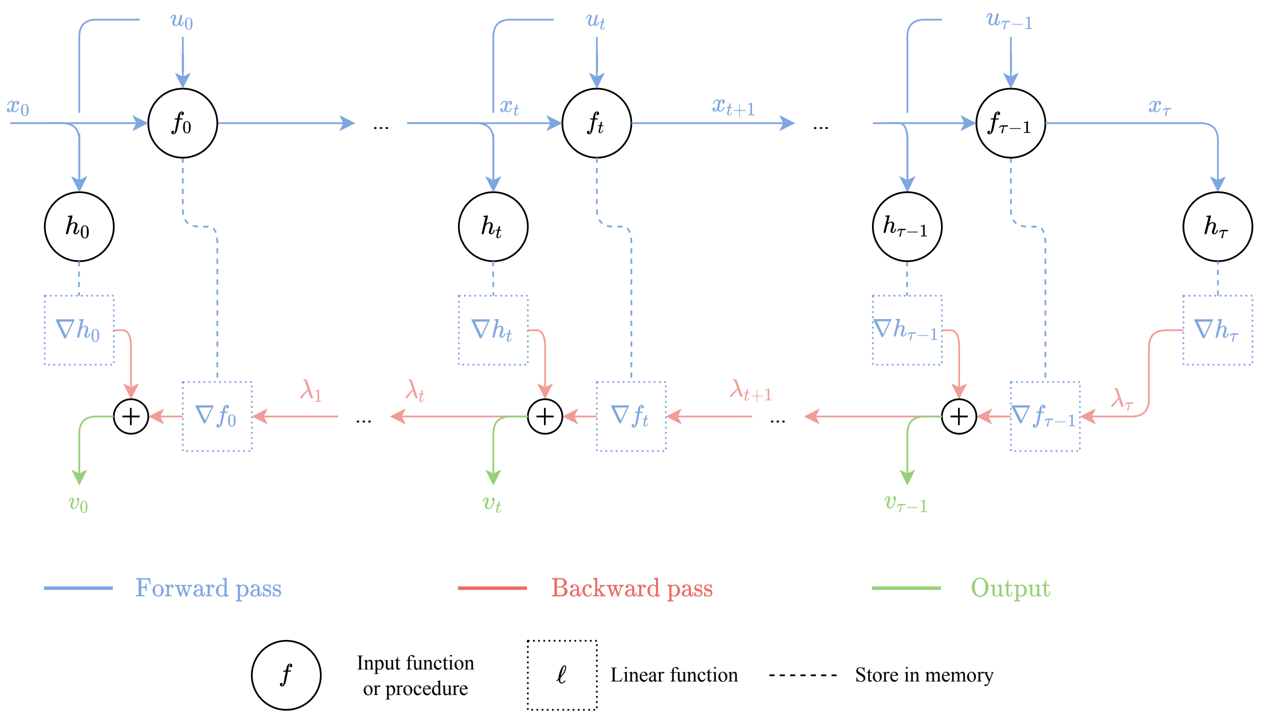


Figure 5 Computational scheme of a gradient oracle.

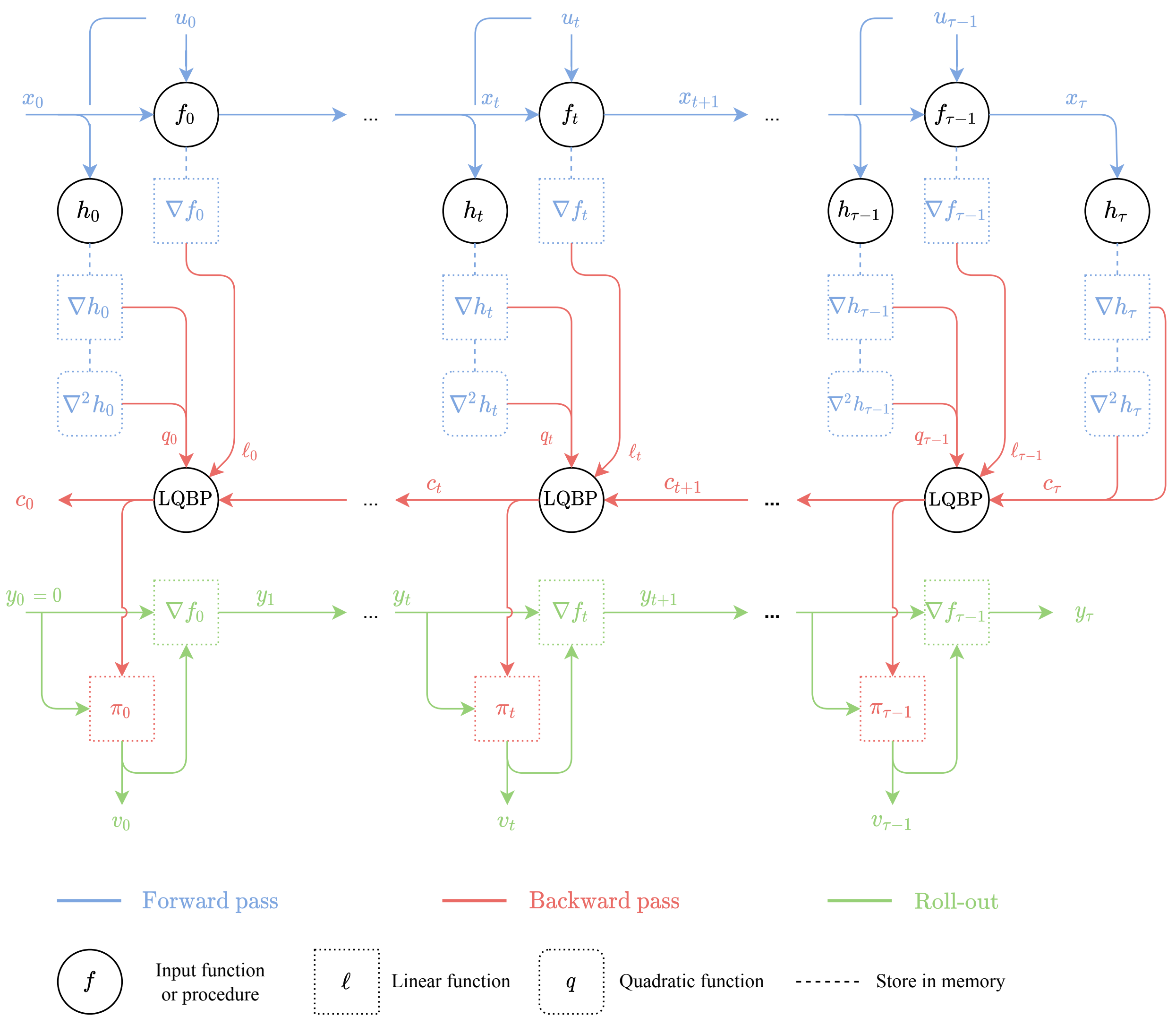


Figure 6 Computational scheme of a Gauss–Newton oracle (ILQR).

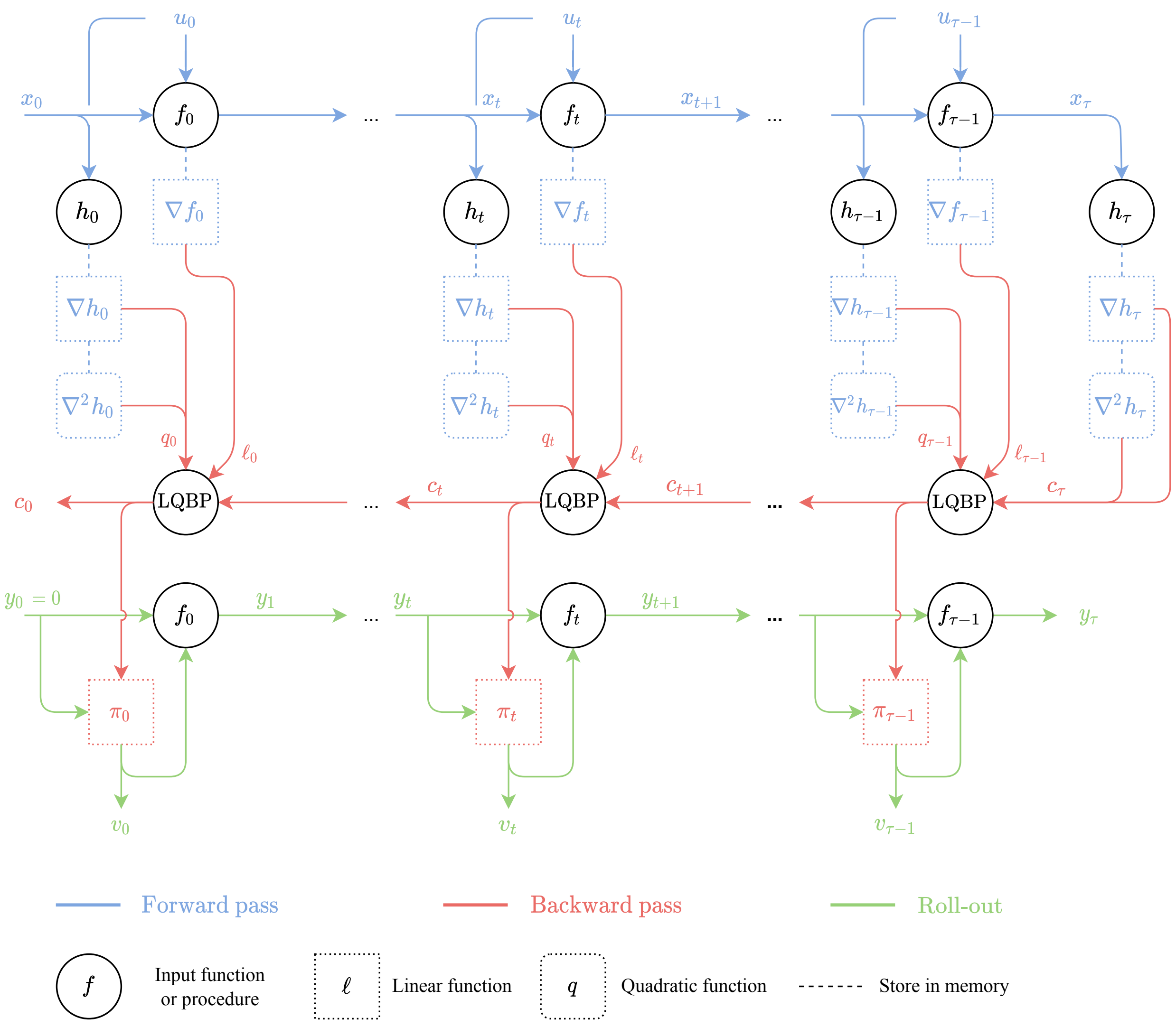


Figure 7 Computational scheme of a DDP oracle with linear quadratic approximations (iLQR).
]

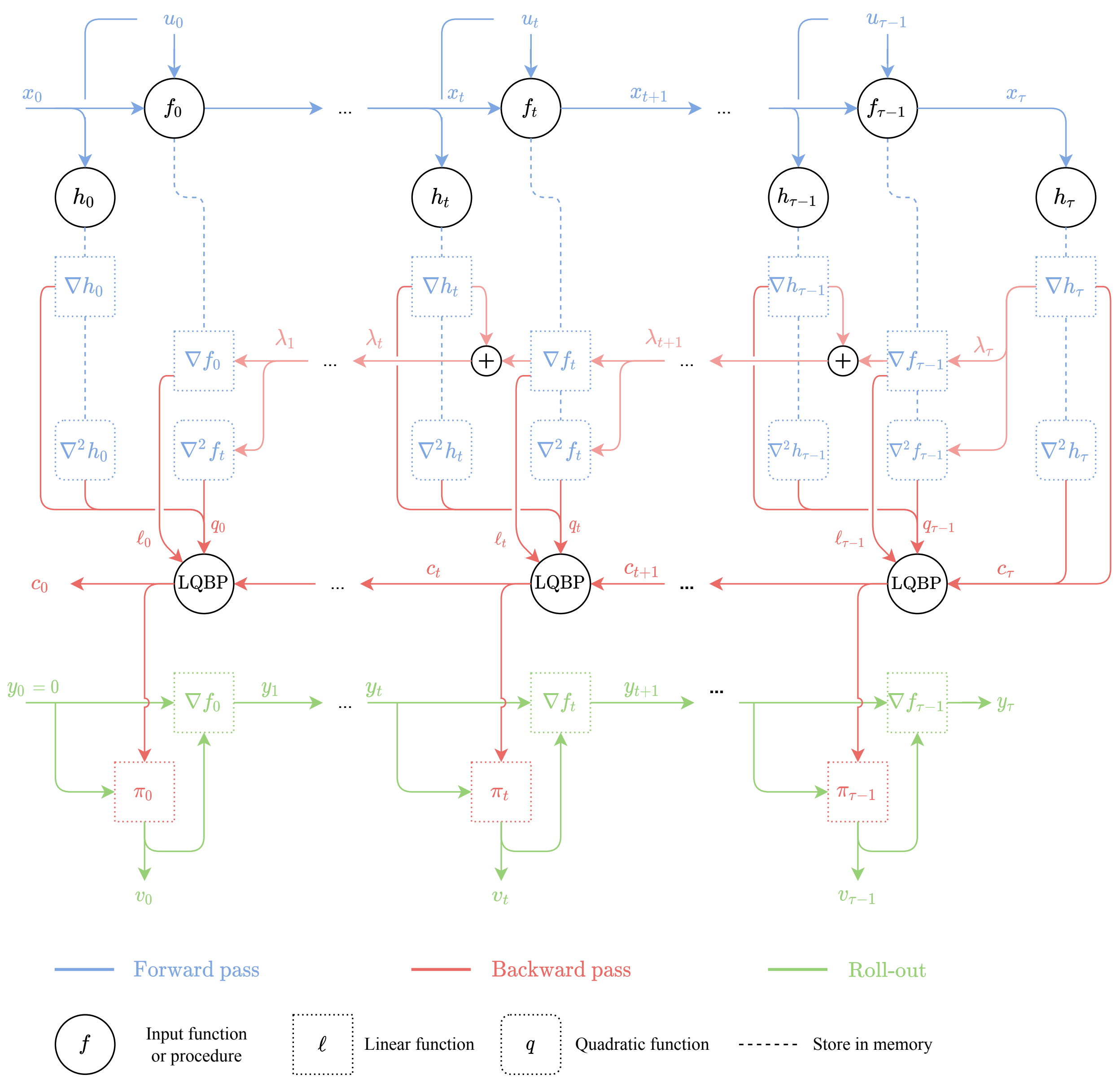


Figure 8 Computational scheme of a Newton oracle.

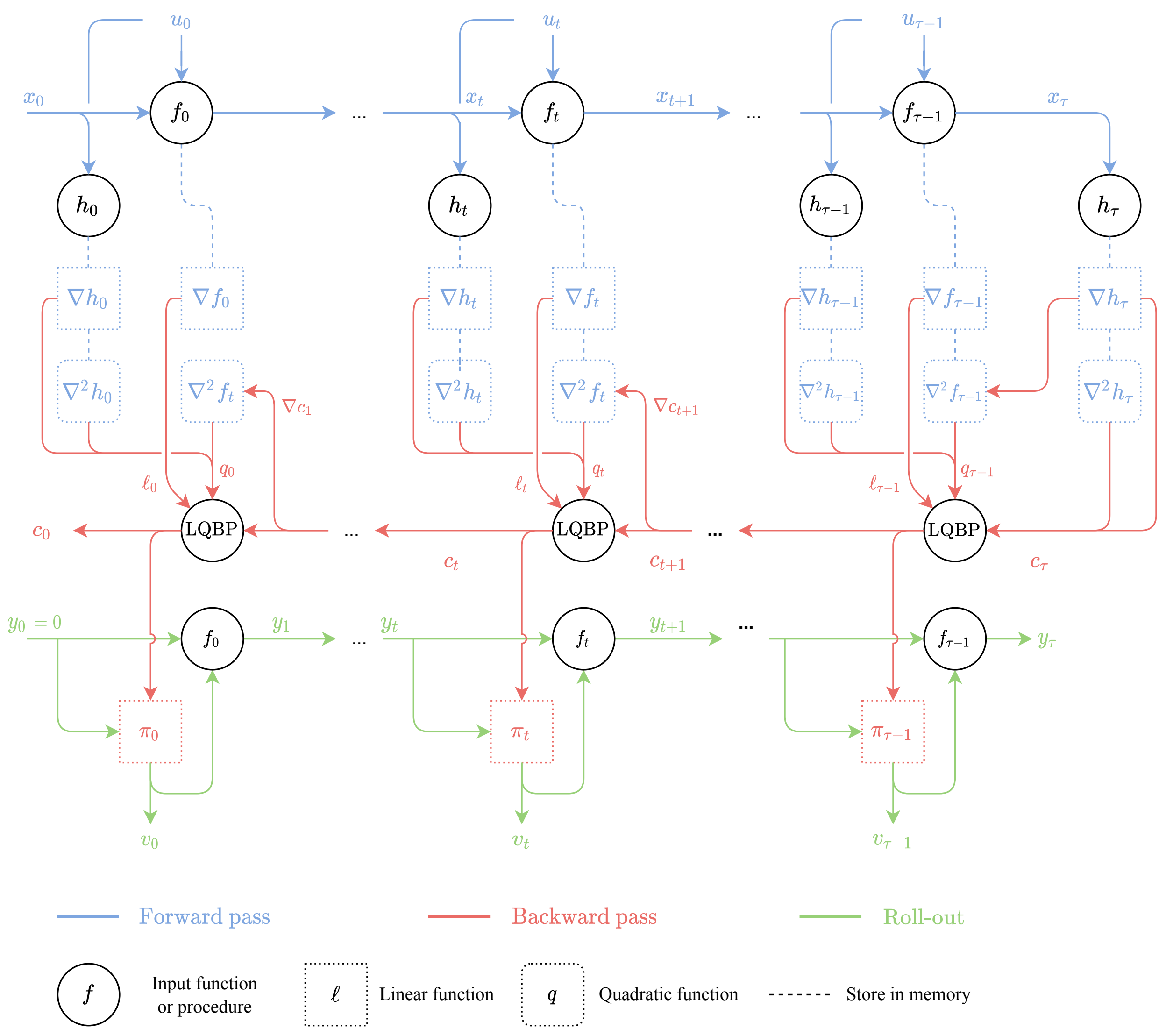


Figure 9 Computational scheme of a DDP oracle with quadratic approximations. (DDP)

Table 2 Space and time complexities of the oracles when storing functions as in, e.g., Algorithm 10.

	Time complexities of the forward pass
All cases	$\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{f_t}+\underbrace{n_x+n_u}_{h_t}\Big) = O(\tau({n_x}^2+n_xn_u))$
	Space complexities of the forward pass
Function eval.	0
All other cases	$\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{f_t}+\underbrace{n_x+n_u}_{h_t}\Big) = O(\tau({n_x}^2+n_xn_u))$
	Time complexities of the backward passes
GD	$\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{\nabla f_t}+\underbrace{n_x+n_u}_{\nabla h_t}+\underbrace{{n_x}^2+n_xn_u}_{\text{Roll}}+\underbrace{{n_x}^2+n_xn_u}_{\text{LBP}}\Big) = O(\tau({n_x}^2+n_xn_u))$
GN/DDP-LQ	$\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{\nabla f_t}+\underbrace{n_x+n_u}_{\nabla h_t}+\underbrace{{n_x}^2+{n_u}^2+n_xn_u}_{\nabla^2 h_t}\Big)$
	$+\,\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{\text{Roll}}+\underbrace{{n_x}^3+{n_u}^3+{n_u}^2n_x}_{\text{LQBP}}\Big) = O(\tau(n_x+n_u)^3)$
NE/DDP-Q	$\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{\nabla f_t}+\underbrace{n_x+n_u}_{\nabla h_t}+\underbrace{{n_x}^2+{n_u}^2+n_xn_u}_{\nabla^2 h_t}+\underbrace{{n_x}^2+{n_u}^2+n_xn_u}_{\nabla^2(f_t^\top\lambda)}\Big)$
	$+\,\tau\Big(\underbrace{{n_x}^2+n_xn_u}_{\text{Roll}}+\underbrace{{n_x}^3+{n_u}^3+{n_u}^2n_x}_{\text{LQBP}}\Big) = O(\tau(n_x+n_u)^3)$

F Computational Complexity in a Differentiable Programming Framework

We detail here how to alleviate intermediate storing of second order information to lower the computational cost of oracles based on quadratic approximations.

The time complexities of the forward pass presented in Section 5, corresponding to the computations of the gradients of the dynamics or the costs and Hessians of the costs, are then incurred during the backward pass. A major difference lies in the computation of the quadratic information of the dynamic required in quadratic oracles such as a Newton oracle or a DDP oracle with quadratic approximations. Indeed, a closer look at Algorithm 8 and Algorithm 9 show that only the Hessians of scalar functions of the form $x, u \to f(x,u)^\top\lambda$ need to be computed, which comes at a cost $(n_x+n_u)^2$. In comparison, the cost of computing the second order information of f is $O((n_x+n_u)^2 n_x)$. As an example, Algorithm 10 presents an implementation of a Newton step using stored functions and inputs.

The computational complexities of the oracles when the dynamics and the costs functions are stored in memory are presented in Table 2. We consider for simplicity that the memory cost of storing the information necessary to evaluate a function $f:\mathbb{R}^d\to\mathbb{R}^n$ is nd as it is the case for a linear function f.

G Alternative Resolution of Linear-Quadratic Control Problem

We presented the implementation of all algorithms in a unified viewpoint with dynamic programming as the core subroutine. For classical optimization steps such as Gauss–Newton or Newton, once the problem has been instantiated, as done in Lemma 13, the resulting quadratic optimization subproblem can be solved in several other ways. We present such alternatives for completeness.

G.1 Block Band Diagonal Underlying Structure

The subproblems we are interested to solve are linear quadratic control problems of the form

$$\min_{\substack{x_0,\ldots,x_\tau\in\mathbb{R}^{n_x}\\ u_0,\ldots,u_{\tau-1}\in\mathbb{R}^{n_u}}} \sum_{t=0}^{\tau-1}\left(\frac{1}{2}x_t^\top P_t x_t + \frac{1}{2}u_t^\top Q_t u_t + x_t^\top R_t u_t + p_t^\top x_t + q_t^\top u_t\right) + \frac{1}{2}x_\tau^\top P_\tau x_\tau + p_\tau^\top x_\tau$$
$$\text{subject to}\quad x_{t+1} = A_t x_t + B_t u_t, \quad \text{for } t\in\{0,\ldots,\tau-1\}, \quad x_0 = \bar{x}_0.$$

By introducing Lagrange multipliers $(\lambda_t)_{t=0}^{\tau}$ for the constraints, the problem can be stated as follows.

$$
\begin{aligned}
&\min_{\substack{x_0,\ldots,x_\tau\in\mathbb{R}^{n_x}\\ u_0,\ldots,u_{\tau-1}\in\mathbb{R}^{n_u}}} \quad \sup_{\lambda_0,\ldots,\lambda_\tau\in\mathbb{R}^{n_x}} \mathcal{L}((x_t)_{t=0}^{\tau}, (u_t)_{t=0}^{\tau-1}, (\lambda_t)_{t=0}^{\tau})\\
&\text{for } \mathcal{L}((x_t)_{t=0}^{\tau}, (u_t)_{t=0}^{\tau-1}, (\lambda_t)_{t=0}^{\tau})\\
&\quad = \sum_{t=0}^{\tau-1}\left(\frac{1}{2}x_t^\top P_t x_t + \frac{1}{2}u_t^\top Q_t u_t + x_t^\top R_t u_t + p_t^\top x_t + q_t^\top u_t + \lambda_{t+1}^\top(x_{t+1} - A_t x_t - B_t u_t)\right)\\
&\quad\quad + \lambda_0^\top(x_0 - \bar{x}_0) + \frac{1}{2}x_\tau^\top P_\tau x_\tau + p_\tau^\top x_\tau.
\end{aligned}
$$

The optimality conditions, a.k.a. KKT conditions, are

$$
\begin{aligned}
x_0 - \bar{x}_0 &= 0 && (\partial_{\lambda_0}\mathcal{L} = 0)\\
P_t x_t + R_t u_t + p_t - A_t^\top \lambda_{t+1} + \lambda_t &= 0 && t \in \{0, \ldots, \tau - 1\}\ (\partial_{x_t}\mathcal{L} = 0)\\
Q_t u_t + R_t^\top x_t + q_t - B_t^\top \lambda_{t+1} &= 0 && t \in \{0, \ldots, \tau - 1\}\ (\partial_{u_t}\mathcal{L} = 0)\\
x_{t+1} - A_t x_t - B_t u_t &= 0 && t \in \{0, \ldots, \tau - 1\}\ (\partial_{\lambda_{t+1}}\mathcal{L} = 0)\\
P_\tau x_\tau + p_\tau + \lambda_\tau &= 0 && (\partial_{x_\tau}\mathcal{L} = 0).
\end{aligned}
$$

As noted by [62], these equations can be ordered as

$$
\begin{aligned}
x_0 &= \bar{x}_0 && (\partial_{\lambda_0}\mathcal{L} = 0)\\
\lambda_0 + P_0 x_0 + R_0 u_0 - A_0^\top \lambda_1 &= -p_0 && (\partial_{x_0}\mathcal{L} = 0)\\
R_0^\top x_0 + Q_0 u_0 - B_0^\top \lambda_1 &= -q_0 && (\partial_{u_0}\mathcal{L} = 0)\\
-A_0 x_0 - B_0 u_0 + x_1 &= 0 && (\partial_{\lambda_1}\mathcal{L} = 0)\\
\lambda_1 + P_1 x_1 + R_1 u_1 - A_1^\top \lambda_2 &= -p_1 && (\partial_{x_1}\mathcal{L} = 0)\\
R_1^\top x_1 + Q_1 u_1 - B_1^\top \lambda_2 &= -q_1 && (\partial_{u_1}\mathcal{L} = 0)\\
&\vdots\\
\lambda_\tau + P_\tau x_\tau &= -p_\tau && (\partial_{x_\tau}\mathcal{L} = 0).
\end{aligned}
$$

Written in matrix form the system to be solved is

$$
\begin{pmatrix}
0 & I & & & & & & & & \\
I & P_0 & R_0 & -A_0 & & & & & & \\
 & R_0^\top & Q_0 & -B_0^\top & & & & & & \\
 & -A_0 & -B_0 & 0 & I & & & & & \\
 & & & I & P_1 & R_1 & -A_1 & & & \\
 & & & & R_1^\top & Q_1 & -B_1^\top & & & \\
 & & & & -A_1 & -B_1 & 0 & \ddots & & \\
 & & & & & & \ddots & \ddots & & \\
 & & & & & & & & \ddots & I \\
 & & & & & & & & I & P_\tau
\end{pmatrix}
\begin{pmatrix}
\lambda_0\\ x_0\\ u_0\\ \lambda_1\\ x_1\\ u_1\\ \lambda_2\\ \vdots\\ \lambda_\tau\\ x_\tau
\end{pmatrix}
=
\begin{pmatrix}
-s_0\\ -p_0\\ -q_0\\ -s_1\\ -p_1\\ -q_1\\ -s_2\\ \vdots\\ -s_\tau\\ -p_\tau
\end{pmatrix},
$$

where $s_0 = -\bar{x}_0$ and $s_t = 0$ are simply introduced for readability.

The system above is band block diagonal, which hints why it can be solved efficiently by various methods. If all blocks were of size 1, that is, $n_x = n_u = 1$, the system would amount to a band diagonal matrix M with bandwidth $\sup\{|i-j| : M_{ij} > 0\} = 2$. Gaussian eliminations of band-diagonal $n \times n$ matrices of bandwidth k are well-known to have a complexity of the order $O(nk^2)$. In our case, since the blocks are not of size one, implementations of Gaussian elimination-like algorithms would incur an $O(\dim_x^3)$ or $O(\dim_u^3)$ to inverse each block.

G.2 Riccati-Based Implementation

Implementation

The system of equations presented above suggest some elimination strategies ([44, 62]). For example, the control variables u_t can be eliminated from the system of equations as we have

$$u_t = -Q_t^{-1} R_t^\top x_t - Q_t^{-1} q_t + Q_t^{-1} B_t^\top \lambda_{t+1}.$$

After eliminating the control variables, the optimality conditions read

$$\begin{aligned}
x_0 &= \bar{x}_0 && (\partial_{\lambda_0}\mathcal{L} = 0)\\
C_t x_t - D_t^\top \lambda_{t+1} + \lambda_t &= c_t && t \in \{0, \ldots, \tau-1\}\ (\partial_{x_t}\mathcal{L} = 0)\\
x_{t+1} - D_t x_t - E_{t+1}\lambda_{t+1} &= e_{t+1} && t \in \{0, \ldots, \tau-1\}\ (\partial_{\lambda_{t+1}}\mathcal{L} = 0)\\
P_\tau x_\tau + \lambda_\tau &= -p_\tau && (\partial_{x_\tau}\mathcal{L} = 0).
\end{aligned}$$

for

$$\begin{aligned}
C_t &= P_t - R_t Q_t^{-1} R_t^\top,\\
D_t &= A_t + B_t Q_{t-1} R_t^\top,\\
E_{t+1} &= B_t Q_t^{-1} B_t^\top,\\
c_t &= -p_t + R_t Q_t^{-1} q_t,\\
e_{t+1} &= -B_t Q_t^{-1} q_t.
\end{aligned}$$

The corresponding system of equations to solve is then band diagonal of the following form, denoting $e_0 = \bar{x}_0, c_\tau = -p_\tau$,

$$\begin{pmatrix}
0 & I & & & & & \\
I & C_0 & -D_0^\top & & & & \\
 & -D_0 & -E_1 & I & & & \\
 & & I & C_1 & \ddots & & \\
 & & & \ddots & \ddots & & \\
 & & & & & -E_\tau & I\\
 & & & & & I & P_\tau
\end{pmatrix}
\begin{pmatrix}\lambda_0\\ x_0\\ \lambda_1\\ x_1\\ \vdots\\ \lambda_\tau\\ x_\tau\end{pmatrix}
=
\begin{pmatrix}e_0\\ c_0\\ e_1\\ c_1\\ \vdots\\ e_\tau\\ c_\tau\end{pmatrix}.$$

We can show by induction that the Lagrange multipliers necessarily satisfy

$$\lambda_t = F_t x_t + f_t \text{ for all } t \in \{1, \ldots, \tau-1\},$$

for some matrices F_t and vectors f_t. For $t = \tau$, we already know that $\lambda_\tau = -P_\tau x_\tau - p_\tau$. Assume the property is true at time $t+1$, then

$$\lambda_{t+1} = F_{t+1}(A_t x_t + B_t u_t) + f_{t+1} = F_{t+1}(A_t x_t - B_t Q_t^{-1} R_t^\top x_t - B_t Q_t^{-1} q_t + B_t Q_t^{-1} B_t^\top \lambda_{t+1}) + f_{t+1}.$$

Rearranging the terms, we get that

$$\lambda_{t+1} = (I - F_{t+1}E_{t+1})^{-1}(F_{t+1}P_t x_t + F_{t+1}e_{t+1} + f_{t+1}).$$

Injecting this expression in the optimality conditions associated to x_t (that is the line $\partial_{x_t}\mathcal{L} = 0$), we get

$$\lambda_t = (D_t^\top (I - F_{t+1}E_{t+1})^{-1} F_{t+1} P_t - C_t) x_t + c_t + D_t^\top (I - F_{t+1}E_{t+1})^{-1}(F_{t+1}e_{t+1} + f_{t+1}).$$

Hence, we can express $\lambda_t = F_t x_t + f_t$ with

$$F_t = (D_t^\top (I - F_{t+1}E_{t+1})^{-1} F_{t+1} P_t - C_t), \qquad f_t = c_t + D_t^\top (I - F_{t+1}E_{t+1})^{-1}(F_{t+1}e_{t+1} + f_{t+1}). \tag{48}$$

Similarly, given F_{t+1}, f_{t+1} such that $\lambda_{t+1} = F_{t+1}x_{t+1} + f_{t+1}$, we can compute an expression of the optimal x_{t+1} in terms of x_t from the optimality condition on λ_{t+1}. Namely, we have

$$x_{t+1} - D_t x_t - E_{t+1}F_{t+1}x_{t+1} + E_{t+1}f_{t+1} = e_{t+1},$$

and so

$$x_{t+1} = (I - E_{t+1}F_{t+1})^{-1}(D_t x_t - E_{t+1} f_{t+1} + e_{t+1}). \tag{49}$$

The whole resolution consists then in

1. Computing F_t, f_t from $t = \tau$ to 0 using (48) starting from $F_\tau = P_\tau$, $f_\tau = p_\tau$.
2. Computing the optimal $x_0, \ldots, x_\tau$ starting from $x_0 = \bar{x}_0$ and using (49) from $t = 0, \ldots, \tau - 1$.

Computational Complexity

Compared to the implementation by dynamic programming, we retrieve a linear complexity with respect to the horizon τ (only two passes on the dynamics), and cubic in the control and state dimensions. One finds that the computational complexity of the method presented above, taking into account the symmetry of some matrices, ([62]) is of the order

$$\tau\left(7{n_x}^3 + 4{n_x}^2 n_u + 4n_x{n_u}^2 + \frac{1}{3}{n_u}^3\right).$$

In comparison, the computational complexity of a dynamic programming-based approach is ([62])

$$\tau\left(3{n_x}^3 + 5{n_x}^2 n_u + 3n_x{n_u}^3 + \frac{1}{3}{n_u}^3\right) + O(\tau({n_x}^2 + {n_u}^2)).$$

While the method presented in this section may be slightly more computationally expansive than a dynamic programming approach, it may be easier to use in a parallel context as recalled below.

G.3 Parallel Implementation

Rather than eliminating the set of control variables, one can consider eliminating blocks of variables to enable parallel implementations of such methods as presented by [62]. Briefly, the approach consists in considering a system reduced to the variables at $L+1$ time steps, i.e., $(\lambda_{t_i}, x_{t_i}, u_{t_i})_{i=0}^L$ for $t_0 = 0$ and $t_L = \tau$. Intermediate variables between time-steps, that is $(\lambda_{t_i+j}, x_{t_i+j}, u_{t_i+j})_{j=1}^{t_{i+1}-1}$ are eliminated by appropriate computations to reduce the system as a set of $3(P+1)-1$ equations, akin to the original system,

$$\begin{pmatrix}
0 & I & & & & & & & & \\
I & \widetilde{P}_0 & \widetilde{R}_0 & -\widetilde{A}_0 & & & & & & \\
 & \widetilde{R}_0^\top & \widetilde{Q}_0 & -\widetilde{B}_0^\top & & & & & & \\
 & -\widetilde{A}_0 & -\widetilde{B}_0 & 0 & I & & & & & \\
 & & & I & \widetilde{P}_1 & \widetilde{R}_1 & -\widetilde{A}_1 & & & \\
 & & & & \widetilde{R}_1^\top & \widetilde{Q}_1 & -\widetilde{B}_1^\top & & & \\
 & & & & -\widetilde{A}_1 & -\widetilde{B}_1 & 0 & \ddots & & \\
 & & & & & & \ddots & \ddots & & \\
 & & & & & & & \ddots & I & \\
 & & & & & & & & I & \widetilde{P}_L
\end{pmatrix}
\begin{pmatrix}
\lambda_{t_0} \\ x_{t_0} \\ u_{t_0} \\ \lambda_{t_1} \\ x_{t_1} \\ u_{t_1} \\ \lambda_{t_2} \\ \vdots \\ \lambda_{t_L} \\ x_{t_L}
\end{pmatrix}
=
\begin{pmatrix}
-\widetilde{s}_{t_0} \\ -\widetilde{p}_{t_0} \\ -\widetilde{q}_{t_0} \\ -\widetilde{s}_{t_1} \\ -\widetilde{p}_{t_1} \\ -\widetilde{q}_{t_1} \\ -\widetilde{s}_{t_2} \\ \vdots \\ -\widetilde{s}_{t_L} \\ -\widetilde{p}_{t_L}
\end{pmatrix},$$

The matrices $\widetilde{M}_j$ for $M \in \{A, B, P, Q, R\}$ can be computed as functions of the intermediate results at that stage, that is a function of $M_{t_j+1}, \ldots, M_{t_j-1}$ for $M \in \{A, B, P, Q, R\}$, see [62] for detailed expressions. Solving the reduced system above is naturally less computationally expensive than computing the original system, while the computations of the reduced system, that is, the computations of $\widetilde{M}_j$ for $M \in \{A, B, P, Q, R\}$ can be done in parallel.

G.4 Matrix-free Solver

Finally, rather than considering computing Newton or Gauss–Newton steps by exploiting the structure of the problem, one can directly use the access to hessian-vector products in a differentiable programming framework.

Implementation

Consider the case of a Newton step, which requires computing

$$\nabla^2 \mathcal{J}(\boldsymbol{u})^{-1} \nabla \mathcal{J}(\boldsymbol{u}),$$

for $\mathcal{J}$ the objective defined in Section 3. Rather than computing the Hessian, and inverting it, this oracle can be computed by solving for $\boldsymbol{v}$ such that

$$\nabla^2 \mathcal{J}(\boldsymbol{u}) \boldsymbol{v} = \nabla \mathcal{J}(\boldsymbol{u}),$$

which can be done approximately by an iterative method such as a conjugate gradient method or a generalized minimal residual method ([41]), provided that we have access only to the linear operator $\boldsymbol{v} \mapsto \nabla^2 \mathcal{J}(\boldsymbol{u})\boldsymbol{v}$. This can be done efficiently in a differentiable programming framework as recalled below.

Automatic differentiation naturally gives access to the gradient $\nabla \mathcal{J}(\boldsymbol{u})$ of the objective at some inputs $\boldsymbol{u}$ by means of the reverse mode of automatic differentiation. For a function $g : \mathbb{R}^n \to \mathbb{R}^m$, its directional derivative at $\boldsymbol{u}$ along a direction $\boldsymbol{v}$, that is the derivative of $t \mapsto g(\boldsymbol{u} + t\boldsymbol{v})$ at $t = 0$, denoted $\partial g(\boldsymbol{u})[\boldsymbol{v}]$, can be computed by forward mode automatic differentiation. The linear operator $\boldsymbol{v} \mapsto \nabla^2 \mathcal{J}(\boldsymbol{u})\boldsymbol{v}$ amounts to the directional derivative of the gradient, that is, $\nabla^2 \mathcal{J}(\boldsymbol{u})\boldsymbol{v} = \partial(\nabla \mathcal{J})(\boldsymbol{u})[\boldsymbol{v}]$. It can then be computed by forward mode automatic differentiation on top of reverse mode automatic differentiation at approximately twice the computational cost of the gradient ([24]).

Computational Complexity

The overall computational cost of computing the Newton oracle depends then on the condition number of the Hessian $\kappa = \sigma_{\max}(\nabla^2 \mathcal{J}(\boldsymbol{u}))/\sigma_{\min}(\nabla^2 \mathcal{J}(\boldsymbol{u}))$ as

$$O\left(\min\left\{\tau n_u, \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\log(\varepsilon^{-1})\right\} \mathcal{T}(\nabla^2 \mathcal{J}(\boldsymbol{u}))\right) = O\left(\min\left\{\tau n_u, \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\log(\varepsilon^{-1})\right\} \tau({n_x}^2 + {n_u}^2)\right)$$

operations, where $\mathcal{T}(\nabla^2 \mathcal{J}(\boldsymbol{u})))$ denotes the cost of computing the Hessian-vector product $\boldsymbol{v} \mapsto \nabla^2 \mathcal{J}(\boldsymbol{u}))\boldsymbol{v}$ in a differentiable programming framework and can be approximated roughly as $\mathcal{T}(\nabla^2 \mathcal{J}(\boldsymbol{u}))) = O(\tau({n_x}^2 + {n_u}^2))$. Overall, such "matrix-free" methods, which circumvent the need to compute actually the Hessian, have a priori a quadratic complexity and not linear complexity w.r.t. the horizon if the matrix is ill-conditioned. On the other hand, the complexity of such methods remains quadratic in the state dimension.

A similar approach can be used to compute Gauss–Newton steps by using the Jacobian vector product of the function that at controls associate the trajectory. For problems with a single final cost, Gauss–Newton methods can also benefit from considering their dual formulation as shown by [47].

We present in Appendix I numerical comparisons of such matrix-free implementations to the implementation by dynamic programming. Note that by using matrix-free solvers in a differentiable programming framework we can cast any nonlinear control as a generic numerical optimization problem amenable to solutions with off-the-shelf programs such as IPOPT ([59]).

H Experimental Detail

We describe in detail the continuous time systems studied in the experiments. The code is available at `https://github.com/vroulet/ilqc`. Numerical constants are detailed at the end for reference. All algorithms are run with double precision.

H.1 Discretization

In the following, we denote by $z(t)$ the state of a system at time t. Given a control $u(t)$ at time t, we consider time-invariant dynamical systems governed by a differential equation of the form

$$\dot{z}(t) = \mathrm{f}(z(t), u(t)), \quad \text{for } t \in [0, T],$$

where f models the physics of the movement and is described below for each model.

Given a continuous time dynamic, the discrete time dynamics are given by a discretization method such that the states follow dynamics of the form

$$z_{t+1} = f(z_t, u_t) \quad \text{for } t \in \{0, \ldots \tau - 1\},$$

for a sequence of controls $u_0, \ldots, u_{\tau-1}$. One discretization method is the Euler method, which, for a time-step $\Delta = T/\tau$, is

$$f(z_t, u_t) = z_t + \Delta \mathrm{f}(z_t, u_t).$$

Alternatively, we can consider a Runge–Kutta method of order 4 that defines the discrete-time dynamics as

$$\begin{aligned} & f(z_t, u_t) = z_t + \frac{\Delta}{6}(k_1 + k_2 + k_3 + k_4) \\ \text{where} \quad & k_1 = \mathrm{f}(z_t, u_t) \qquad\qquad k_2 = \mathrm{f}(z_t + \Delta k_1/2, u_t) \\ & k_3 = \mathrm{f}(z_t + \Delta k_2/2, u_t) \qquad k_4 = \mathrm{f}(z_t + \Delta k_3, u_t), \end{aligned}$$

where we consider the controls to be piecewise constant, i.e., constant on time intervals of size Δ. We can also consider a Runge–Kutta method with varying control inputs such that, for $u_t = (v_t, v_{t+1/3}, v_{t+2/3})$,

$$\begin{aligned} & f(z_t, u_t) = z_t + \frac{\Delta}{6}(k_1 + k_2 + k_3 + k_4) \\ \text{where} \quad & k_1 = \mathrm{f}(z_t, v_t) \qquad\qquad\qquad k_2 = \mathrm{f}(z_t + \Delta k_1/2, v_{t+1/3}) \\ & k_3 = \mathrm{f}(z_t + \Delta k_2/2, v_{t+1/3}) \qquad k_4 = \mathrm{f}(z_t + \Delta k_3, v_{t+2/3}). \end{aligned}$$

H.2 Swinging up a Pendulum

H.2.1 Fixed Pendulum

We consider the problem of controlling a fixed pendulum such that it swings up as illustrated in Figure 10. Namely, the dynamics of a pendulum are given as

$$ml^2\ddot{\theta}(t) = -mlg \sin \theta(t) - \mu\dot{\theta}(t) + u(t),$$

with θ the angle of the rod, m the mass of the blob, l the length of the blob, μ a friction coefficient, g the gravitational constant, and u a torque applied to the pendulum (which defines the control we have on the system). Denoting the angle speed $\omega = \dot{\theta}$ and the state of the system $x = (\theta; \omega)$, the continuous time dynamics are

$$\mathrm{f} : (x = (\theta; \omega), u) \rightarrow \begin{pmatrix} \omega \\ -\frac{g}{l} \sin \theta - \frac{\mu}{ml^2}\omega + \frac{1}{ml^2}u \end{pmatrix},$$

such that the continuous time system is defined by $\dot{x}(t) = \mathrm{f}(x(t), u(t))$. After discretization by an Euler method, we get discrete time dynamics $f_t(x_t, u_t) = f(x_t, u_t)$ of the form, for $x_t = (\theta_t; \omega_t)$ and Δ the discretization step,

$$f(x_t, u_t) = x_t + \Delta \mathrm{f}(x_t, u_t) = \begin{pmatrix} \theta_t + \Delta\omega_t \\ \omega_t + \Delta \left(-\frac{g}{l} \sin \theta_t - \frac{\mu}{ml^2}\omega_t + \frac{1}{ml^2}u_t\right) \end{pmatrix}.$$

A classical task is to enforce the pendulum to swing up and stop without using too much torque at each time step, i.e., for $\bar{x}_0 = (0; 0)$, the costs we consider are, for some non-negative parameters $\lambda \geq 0, \rho \geq 0$,

$$h_t(x_t, u_t) = \lambda \|u_t\|_2^2 \quad \text{for } t \in \{0, \ldots, \tau - 1\}, \quad h_\tau(x_\tau) = (\pi - \theta_\tau)^2 + \rho\|\omega_\tau\|_2^2.$$

H.2.2 Pendulum on a Cart

We consider here controlling a pendulum on a cart as illustrated in Figure 11. This system is described by the angle θ of the pendulum with the vertical and the position z_x of the cart on the horizontal axis. Contrary to the previous example, here we do not control directly the angle of the pendulum we only control the system

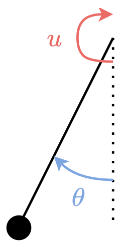


Figure 10 Fixed pendulum.

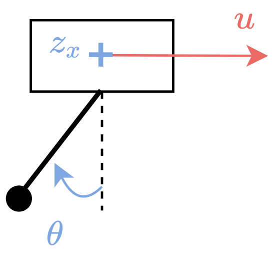


Figure 11 Pendulum on a cart.

with a force u that drives the acceleration of the cart. The dynamics of the system satisfy (see [35] for detailed derivations)

$$\begin{aligned}(M+m)\ddot{z}_x + ml\cos\theta\ddot{\theta} &= -b\dot{z}_x + ml\dot{\theta}^2\sin\theta + u\\ ml\cos\theta\ddot{z}_x + (I+ml^2)\ddot{\theta} &= -mgl\sin\theta,\end{aligned} \tag{50}$$

where M is the mass of the cart, m is the mass of the pendulum rod, I is the pendulum rod moment of inertia, l is the length of the rod, and b is the viscous friction coefficient of the cart. The system of equations can be written in matrix form and solved to express the angle and position accelerations as

$$\begin{aligned}\begin{pmatrix}\ddot{z}_x\\ \ddot{\theta}\end{pmatrix} &= \begin{pmatrix}M+m & ml\cos\theta\\ ml\cos\theta & I+ml^2\end{pmatrix}^{-1}\begin{pmatrix}-b\dot{z}_x + ml\dot{\theta}^2\sin\theta + u\\ -mgl\sin\theta\end{pmatrix}\\ &= \frac{1}{I(M+m)+ml^2M+m^2l^2\sin^2\theta}\begin{pmatrix}I+ml^2 & -ml\cos\theta\\ -ml\cos\theta & M+m\end{pmatrix}\begin{pmatrix}-b\dot{z}_x + ml\dot{\theta}^2\sin\theta + u\\ -mgl\sin\theta\end{pmatrix}.\end{aligned}$$

The discrete dynamical system follows using an Euler discretization scheme or a Runge Kutta method. We consider the task of swinging up the pendulum and keeping it vertical for a few time steps while constraining the movement of the cart on the horizontal line. Formally, we consider the following cost, defined for $x_t = (z_x, \theta, \zeta_x, \omega)$, where ζ_x, ω represent the discretizations of $\dot{z}_x$ and $\dot{\theta}$ respectively,

$$h(x_t, u_t) = \begin{cases}\rho_2(\max((z_x - \bar{z}_x^+)^3, 0) + \max((z_x + \bar{z}_x^-)^3, 0)) + \lambda u_t^2 + (\theta+\pi)^2 + \rho_1\omega^2 & \text{if } t \geq \bar{t}\\ \rho_2(\max((z_x - \bar{z}_x^+)^3, 0) + \max((z_x + \bar{z}_x^-)^3, 0)) + \lambda u_t^2 & \text{if } t < \bar{t},\end{cases}$$

where ρ_1, ρ_2, λ are some non-negative parameters, $\bar{t}$ is a time step after which the pendulum needs to stay vertically inverted and $\bar{z}_x^+, \bar{z}_x^-$ are bounds that restrain the movement of the cart along the whole horizontal line.

H.3 Autonomous Car Racing

We consider the control of a car on a track through two different dynamical models: a simple one where the orientation of the car is directly controlled by the steering angle, and a more realistic one that takes into account the tire forces to control the orientation of the car. In the following, we present the dynamics, a simple tracking cost, and a contouring cost enforcing the car to race the track at a reference speed or as fast as possible.

H.3.1 Dynamic

H.3.1.1 Simple Model

A simple model of the car is described in Figure 12. The state of the car is decomposed as $z(t) = (x(t), y(t), \theta(t), v(t))$, where (dropping the dependency w.r.t. time for simplicity)
1. x, y denote the position of the car on the plane,
2. θ denotes the angle between the orientation of the car and the horizontal axis, a.k.a. the yaw,
3. v denotes the longitudinal speed.

The car is controlled through $u(t) = (a(t), \delta(t))$, where
1. a is the longitudinal acceleration of the car,
2. δ is the steering angle.

For a car of length L, the continuous time dynamics are then

$$\dot{x} = v\cos\theta \qquad \dot{y} = v\sin\theta \qquad \dot{\theta} = v\tan(\delta)/L \qquad \dot{v} = a. \tag{51}$$

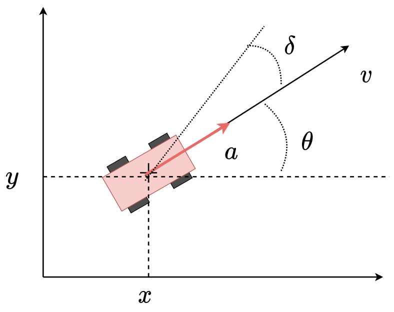


Figure 12 Simple model of a car.

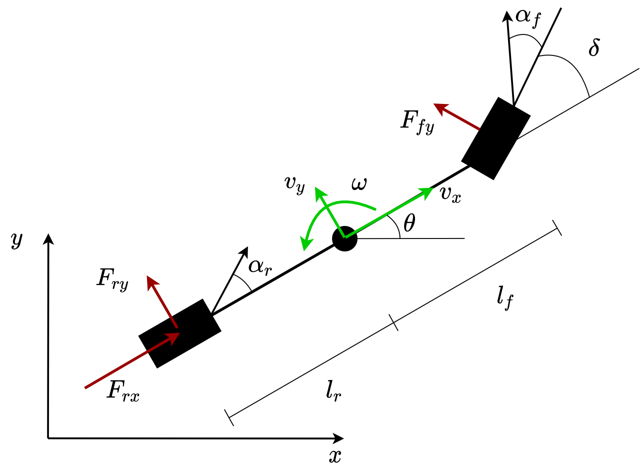


Figure 13 Bicycle model of a car.

H.3.1.2 Bicycle Model

We consider the model presented by [33] recalled below and illustrated in Figure 13. In this model, the state of the car at time t is decomposed as $z(t) = (x(t), y(t), \theta(t), v_x(t), v_y(t), \omega(t))$ where

1. x, y denote the position of the car on the plane,
2. θ denotes the angle between the orientation of the car and the horizontal axis, a.k.a. the yaw,
3. v_x denotes the longitudinal speed,
4. v_y denotes the lateral speed,
5. ω denotes the derivative of the orientation of the car, a.k.a. the yaw rate.

The control variables are analogous to the simple model, i.e., $u(t) = (a(t), \delta(t))$, where

1. a is the PWM duty cycle of the car, this duty cycle can be negative to take into account braking,
2. δ is the steering angle.

These controls act on the state through the following forces.

1. A longitudinal force on the rear wheels, denoted $F_{r,x}$ modeled using a motor model for the DC electric motor as well as a friction model for the rolling resistance and the drag

   $$F_{r,x} = (C_{m1} - C_{m2}v_x)a - C_{r0} - C_{rd}v_x^2,$$

   where $C_{m1}, C_{m2}, C_{r0}, C_{rd}$ are constants estimated from experiments, see Appendix H.
2. Lateral forces on the front and rear wheels, denoted $F_{f,y}, F_{y,r}$ respectively, modeled using a simplified Pacejka tire model

   $$F_{f,y} = D_f \sin(C_f \arctan(B_f \alpha_f)) \quad \text{where } \alpha_f = \delta - \operatorname{arctan2}\left(\frac{\omega l_f + v_y}{v_x}\right)$$
   $$F_{r,y} = D_r \sin(C_r \arctan(B_r \alpha_r)) \quad \text{where } \alpha_r = \operatorname{arctan2}\left(\frac{\omega l_r - v_y}{v_x}\right)$$

   where α_f, α_r are the slip angles on the front and rear wheels respectively, l_f, l_r are the distance from the center of gravity to the front and the rear wheel respectively and the constants $B_r, C_r, D_r, B_f, C_f, D_f$ define the exact shape of the semi-empirical curve, presented in Figure 14.

   The continuous time dynamics are then

$$\begin{aligned}
\dot{x} &= v_x \cos\theta - v_y \sin\theta & \dot{v}_x &= \frac{1}{m}(F_{r,x} - F_{f,y}\sin\delta) + v_y\omega \\
\dot{y} &= v_x \sin\theta + v_y \cos\theta & \dot{v}_y &= \frac{1}{m}(F_{r,y} + F_{f,y}\cos\delta) - v_x\omega \\
\dot{\theta} &= \omega & \dot{\omega} &= \frac{1}{I_z}(F_{f,y} l_f \cos\delta - F_{r,y} l_r),
\end{aligned} \tag{52}$$

where m is the mass of the car and I_z is the inertia.

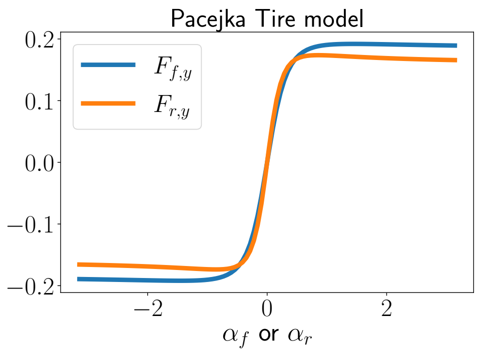


Figure 14 Pacejka model of the friction on the tires as a function of the slip angles

H.3.2 Cost

H.3.2.1 Track

We consider tracks that are given as a continuous curve, namely a cubic spline approximating a set of points. As a result, for any time t, we have access to the corresponding point $\widehat{x}(t), \widehat{y}(t)$ on the curve. The track we consider is a simple track illustrated in Figure 15.

H.3.2.2 Tracking Cost

A simple cost on the states is

$$c_t(z_t) = \|x_t - \widehat{x}(\Delta v^{\mathrm{ref}} t)\|_2^2 + \|y_t - \widehat{y}(\Delta v^{\mathrm{ref}} t)\|_2^2 \quad \text{for } t = 1, \ldots, \tau, \tag{53}$$

for $z_t = (x_t, y_t)$, where Δ is some discretization step and v^{ref} is some reference speed. The cost above is the one we choose for the simple model of a car. The disadvantage of such a cost is that it enforces the car to follow the track at a constant speed which may not be physically possible. We consider in the following a contouring cost as done by [33].

H.3.2.3 Ideal Cost

Given a track parameterized in continuous time, an ideal cost is to enforce the car to be as close as possible to the track, while moving along the track as fast as possible. Formally, define the distance from the car at position (x, y) to the track defined by the curve $\widehat{x}(t), \widehat{y}(t)$ as

$$d(x, y) = \min_{t \in \mathbb{R}} \quad \sqrt{((x - \widehat{x}(t))^2 + (y - \widehat{y}(t))^2}.$$

Denoting $t^* = t(x, y) = \arg\min_{t \in \mathbb{R}} \quad (x - \widehat{x}(t))^2 + (y - \widehat{y}(t))^2$, the reference time on the track for a car at position (x, y), the distance $d(x, y)$ can be expressed as

$$d(x, y) = \sin(\theta(t^*))\,(x - \widehat{x}(t^*)) - \cos(\theta(t^*))\,(y - \widehat{y}(t^*))\,,$$

where $\theta(t) = \frac{\partial \hat{y}(t)}{\partial \hat{x}(t)}$ is the angle of the track with the x-axis. The distance $d(x, y)$ is illustrated in Figure 16. An ideal cost for the problem is then defined as $h(z) = h(x, y) = d(x, y)^2 - t(x, y)$, which enforces the car to be close to the track by minimizing $d(x, y)^2$, and also encourages the car to go as far as possible by adding the term $-t(x, y)$.

H.3.2.4 Contouring and Lagging Cost

The computation of t^* involves solving an optimization problem and is not practical. As [33], we rather augment the states with a flexible reference time. Namely, we augment the state of the car by adding a variable s whose objective is to approximate the reference time t^*. The cost is then decomposed into the *contouring cost* and the *lagging cost* illustrated in Figure 17 and defined as

$$\begin{aligned}
e_c(x, y, s) &= \sin(\theta(s))\,(x - \widehat{x}(s)) - \cos(\theta(s))\,(y - \widehat{y}(s)) \\
e_l(x, y, s) &= -\cos(\theta(s))\,(x - \widehat{x}(s)) - \sin(\theta(s))\,(y - \widehat{y}(s))\,.
\end{aligned}$$

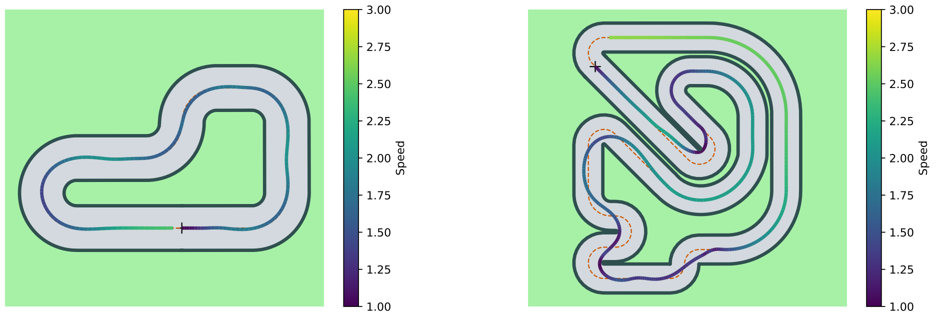


Figure 15 Simple and complex tracks used with a trajectory computed on the bicycle model (52).

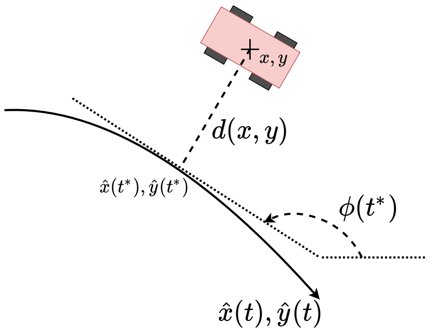


Figure 16 Distance to the track.

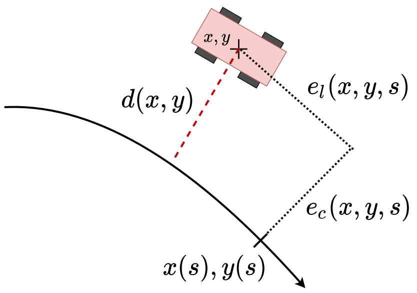


Figure 17 Approx. by contouring and lagging costs.

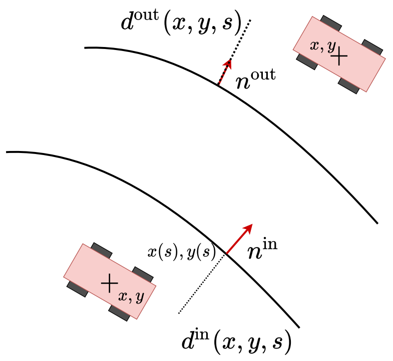


Figure 18 Border costs.

Rather than encouraging the car to make the most progress on the track, we enforce them to keep a reference speed. Namely, we consider an additional penalty of the form $\|\dot{s} - v^{\text{ref}}\|_2^2$ where v^{ref} is a parameter chosen in advance. For the reference time s not to go backward in time, we add a log-barrier term $-\varepsilon \log(\dot{s})$ for $\varepsilon = 10^{-6}$.

Finally, we let the system control the reference time through its second order derivative $\ddot{s}$. Overall this means that we augment the state variable by adding the variables s and $\nu := v_s$ and that we augment the control variable by adding the variable $\alpha := a_s$ such that the discretized problem is written for, e.g., the bicycle model, as

$$\begin{aligned}
\min_{(a_0,\delta_0,\alpha_0),\ldots,(a_{\tau-1},\delta_{\tau-1},\alpha_{\tau-1})} \quad & \sum_{t=0}^{\tau-1} \rho_c e_c(x_t, y_t, s_t)^2 + \rho_l e_l(x_t, y_t, s_t)^2 + \rho_v \|v_{s,t} - v^{\text{ref}}\|_2^2 - \varepsilon \log \nu_t \\
\text{s.t.} \quad & x_{t+1}, y_{t+1}, \theta_{t+1}, v_{x,t+1}, v_{y,t+1}, \omega_{t+1} = f(x_t, y_t, \theta_, v_{x,t}, v_{y,t}, \omega_t, \delta_t, a_t) \\
& s_{t+1} = s_t + \Delta \nu_t, \quad \nu_{t+1} = \nu_t + \Delta \alpha_t \\
& z_0 = \widehat{z}_0 \quad s_0 = 0 \quad \nu_0 = v^{\text{ref}},
\end{aligned}$$

where f is a discretization of the continuous time dynamics, Δ is a discretization step and $\widehat{z}_0$ is a given initial state where z_0 regroups all state variables at time 0 (i.e. all variables except a_0, δ_0).

This cost is defined by the parameters $\rho_c, \rho_l, \rho_v, v^{\text{ref}}$ which are fixed in advance. The larger the parameter ρ_c, the closer the car to the track. The larger the parameter ρ_l, the closer the car to its reference time s. In practice, we want the reference time to be a good approximation of the ideal projection of the car on the track so ρ_l should be chosen large enough. On the other hand, varying ρ_c allows having a car that is either conservative and potentially slow or a car that is fast but inaccurate, i.e., far from the track. The most important aspect of the trajectory is to ensure that the car remains inside the borders of the track defined in advance.

H.3.2.5 Border Cost

To enforce the car to remain inside the track defined by some borders, we penalize the approximated distance of the car to the border when it goes outside the border as $e_b(x, y, s) = e_b^{\mathrm{in}}(x, y, s) + e_b^{\mathrm{out}}(x, y, s)$ with

$$\begin{aligned} e_b^{\mathrm{in}}(x, y, s) &= \max((w + d^{\mathrm{in}}(x, y, s))^3, 0) & d^{\mathrm{in}}(x, y, s) &= -(z - z^{\mathrm{in}}(s))^\top n^{\mathrm{in}}(s) \\ e_b^{\mathrm{out}}(x, y, s) &= \max((w + d^{\mathrm{out}}(x, y, s))^3, 0) & d^{\mathrm{out}}(x, y, s) &= (z - z^{\mathrm{out}}(s))^\top n^{\mathrm{out}}(s) \end{aligned} \tag{54}$$

for $z = (x, y)$, where $n^{\mathrm{in}}(s)$ and $n^{\mathrm{out}}(s)$ denote the normal at the borders at time s and w is the width of the car. In practice, we use a smooth approximation of the max function in (54). The normals $n^{\mathrm{in}}(s)$ and $n^{\mathrm{out}}(s)$ can easily be computed by differentiating the curves defining the inner and outer borders. These costs are illustrated in Figure 18.

H.3.2.6 Constrained Control

We constrain the steering angle to be between $[-\pi/3, \pi/3]$ by parameterizing the steering angle as

$$\delta(\widetilde{\delta}) = \frac{2}{3}\arctan(\widetilde{\delta}) \quad \text{for } \widetilde{\delta} \in \mathbb{R}.$$

Similarly, we constrain the acceleration a to be between $[c, d]$ (with $c = -0.1, d = 1.$), by parameterizing it as

$$a(\widetilde{a}) = (d - c)\,\mathrm{sig}(4\widetilde{a}/(d - c)) + c$$

with sig : $x \to 1/(1 + e^{-x})$ the sigmoid function. The final set of control variables is then $\widetilde{a}, \widetilde{\delta}, \alpha$.

H.3.2.7 Control Cost

For both trajectory costs, we add a square regularization on the control variables of the system, i.e., the cost on the control variables is $\lambda\|u_t\|_2^2$ for some $\lambda \geq 0$ where u_t are the control variables at time t.

H.3.2.8 Overall Contouring Cost

The whole problem with contouring cost is then

$$\begin{aligned} \min_{(\tilde{a}_0, \tilde{\delta}_0, \alpha_0), \ldots, (\tilde{a}_{\tau-1}, \tilde{\delta}_{\tau-1}, \tilde{\alpha}_{\tau-1})} \quad & \sum_{t=0}^{\tau-1} \Big[\rho_c e_c(x_t, y_t, s_t)^2 + \rho_l e_l(x_t, y_t, s_t)^2 + \rho_v \|v_{s,t} - v^{\mathrm{ref}}\|_2^2 - \varepsilon \log(\nu_t) \\ & \qquad + \rho_b e_b(x_t, y_t, s_t)^2 + \lambda(\widetilde{a}_t^2 + \widetilde{\delta}_t^2 + \alpha_t^2)\Big] \\ \text{s.t.} \quad & x_{t+1}, y_{t+1}, \theta_{t+1}, v_{x,t+1}, v_{y,t+1}, \omega_{t+1} = f(x_t, y_t, \theta_t, v_{x,t}, v_{y,t}, \omega_t, \delta_t(\widetilde{\delta}_t), a_t(\widetilde{a}_t)) \\ & s_{t+1} = s_t + \Delta\nu_t, \quad \nu_{t+1} = \nu_t + \Delta\alpha_t \\ & z_0 = \widehat{z}_0 \quad s_0 = 0 \quad \nu_0 = v^{\mathrm{ref}}, \end{aligned} \tag{55}$$

with parameters $\rho_c, \rho_l, \rho_v, v^{\mathrm{ref}}, \rho_b, \lambda$ and f given in (52).

H.4 Numerical Constant

The code is available at `https://github.com/vroulet/ilqc`. We add for ease of reference, the hyperparameters used for each setting.

Pendulum

1. mass $m = 1$,
2. gravitational constant $g = 10$,
3. length of the blob $l = 1$,
4. friction coefficient $\mu = 0.01$,
5. speed regularization $\lambda = 0.1$,
6. control regularization $\rho = 10^{-6}$,
7. total time of the movement $T = 2$, discretization step $\Delta = T/\tau$ for varying τ
8. Euler discretization scheme.

Pendulum on a cart

1. mass of the rod $m = 0.2$,
2. mass of the cart $M = 0.5$,
3. viscous coefficient $b = 0.1$,
4. moment of inertia $I = 0.006$,
5. length of the rod 0.3,
6. speed regularization $\lambda_1 = 0.1$,
7. barrier parameter $\rho_2 = 10^{-6}$.,
8. control regularization $\rho = 10^{-6}$,
9. total time of the movement $T = 2.5$, discretization step $\Delta = T/\tau$ for varying τ,
10. stay put time $\bar{t} = \tau - \lfloor 0.6/\Delta \rfloor$,
11. barriers $\bar{z}^{+} = 2$, $\bar{z}^{-} = -2$,
12. Euler discretization scheme.

Simple car with tracking cost

1. length of the car $L = 1$,
2. reference speed $v^{ref} = 3$,
3. initial speed $v^{init} = 1$,
4. control regularization $\lambda = 10^{-6}$,
5. total time of the movement $T = 2$,
6. simple track,
7. Euler discretization scheme.

Bicycle model of a car with a contouring objective

1. $C_{m1} = 0.287$, $C_{m2} = 0.0545$,
2. $C_{r0} = 0.0518$, $C_{rd} = 0.00035$,
3. $B_r = 3.3852$, $C_r = 1.2691$, $D_r = 0.1737$, $l_r = 0.033$
4. $B_f = 2.579$, $C_f = 1.2$, $D_f = 0.192$, $l_f = 0.029$
5. $m = 0.041$, $I_z = 27.8 \cdot 10^{-6}$
6. contouring error penalty $\rho_c = 0.1$,
7. lagging error penalty $\rho_l = 10$,
8. reference speed penalty $\rho_v = 0.1$,
9. barrier error penalty $\rho_b = 0.$,
10. reference speed $v^{\mathrm{ref}} = 3$,
11. initial speed $v^{init} = 1$,
12. control regularization $\lambda = 10^{-6}$,
13. total time of the movement $T = 1$,
14. simple track,
15. Runge–Kutta discretization scheme.

I Additional Experiment

I.1 Time Comparison

Figures 19 and 20 present the convergence of the algorithms presented in 2 and 3 in time rather than in iterations.

I.2 Stepsize Selection

In Figure 23, we plot the stepsizes taken by algorithms using linear-quadratic approximations for the pendulum and the simple model of a car.

1. On the pendulum example, the stepsizes used by directional steps quickly tend to 1 which means that the algorithms (GN or DDP-LQ) are then taking the largest possible stepsize for this strategy and may exhibit quadratic convergence. On the other hand, for the regularized steps, on the pendulum example, the

regularization (i.e. the reciprocal of the stepsizes) quickly converges to 0, which means that, as the number of iterations increases, the regularized and directional steps coincide.

2. For the car example, the step sizes for the directional steps slowly increase to one. We note yet that while stepsizes taken by DDP-LQ and GN are similar, DDP-LQ displays a faster convergence in terms of gradient norm. For the regularized steps, the regularizations (i.e. the reciprocal of the stepsizes) tend to remain low and stable. As for the directional steps, the regularizations taken by regularized steps are similar between DDP-LQ and GN, yet DDP-LQ displays faster convergence in terms of gradient norm.

In Figure 24, we compare the stepsizes taken by the methods using quadratic approximations.

1. In terms of directional steps, DDP-Q appears to take relatively large steps while its NE counterpart displays more variations on, e.g., the pendulum example. For both algorithms, the stepsizes tend to oscillate for the car example, never steadily taking full steps (stepsize of 1) closer to convergence.
2. In terms of regularized steps, DDP-Q tends to take larger steps (smaller regularizations) than its Newton counterpart, in particular on the car example.

I.3 Comparison of Inner Solver

As presented in Appendix G, we may consider using directly Hessian-vector products to solve the linear quadratic controls arising from the computation of Gauss–Newton and Newton steps. In Figure 27, we plotted the ratio of time between an implementation using dynamic programming and an implementation using matrix-free solvers for varying dimensions of the state, the control and various horizons on synthetic linear quadratic control problems. Namely, for each triplet (n_x, n_u, τ), we generated five linear quadratic control problems, solved each problem as if those were nonlinear dynamics for which we are computing a Gauss–Newton step, by using each of the aforementioned methods. We then averaged the time needed for each method over the five instances and computed the ratio of time between an implementation by dynamic programming and an implementation by matrix-free solvers. These values are recorded in a heatmap in 27, where blue cells correspond to instances where dynamic programming is faster than the matrix-free program and red cells correspond to instances where dynamic programming is slower than its counterpart.

The matrix-free solver approach can readily be implemented in any differentiable programming framework such as CasADI ([2]) which takes advantage of the differentiable dynamic programming framework to cast nonlinear control problems as numerical optimization problems fed into off-the-shelf solvers like IPOPT ([59]).

The results presented in 27 show that for small state dimensions, the dynamic programming approach is generally faster. As soon as the state dimension exceeds a few dozen dimensions, the matrix-free approach is generally faster. This observation matches the computation complexities delineated in Appendix G and Section 5 as the matrix-free approach a priori scales quadratically in terms of the state dimension while the dynamic programming approach scales cubically.

Beyond the time comparisons, each approach has different advantages. The matrix-free approach enables simple introduction of constraints in control variables by casting the whole problem as an optimization problem solved by interior-point methods. On the other-hand, the dynamic programming approach can be adapted to differential dynamic programming procedures as explained in this manuscript.

We already presented in Section 6 numerical comparisons in *iterations* of classical optimization methods (Gauss–Newton or Newton) against their differential dynamic programming counterparts (iLQR or DDP). By using matrix-free solvers instead of dynamic programming procedures to implement Gauss–Newton or Newton steps, these behaviors in iterations would not change. The comparisons in time presented in Figure 19 and Figure 20 can change by using matrix-free solvers as suggested by the heatmaps presented in Figure 27. However, the qualitative conclusions presented in this manuscript remain the same and suggest that differential dynamic programming methods may offer overall gains over classical optimization algorithms.

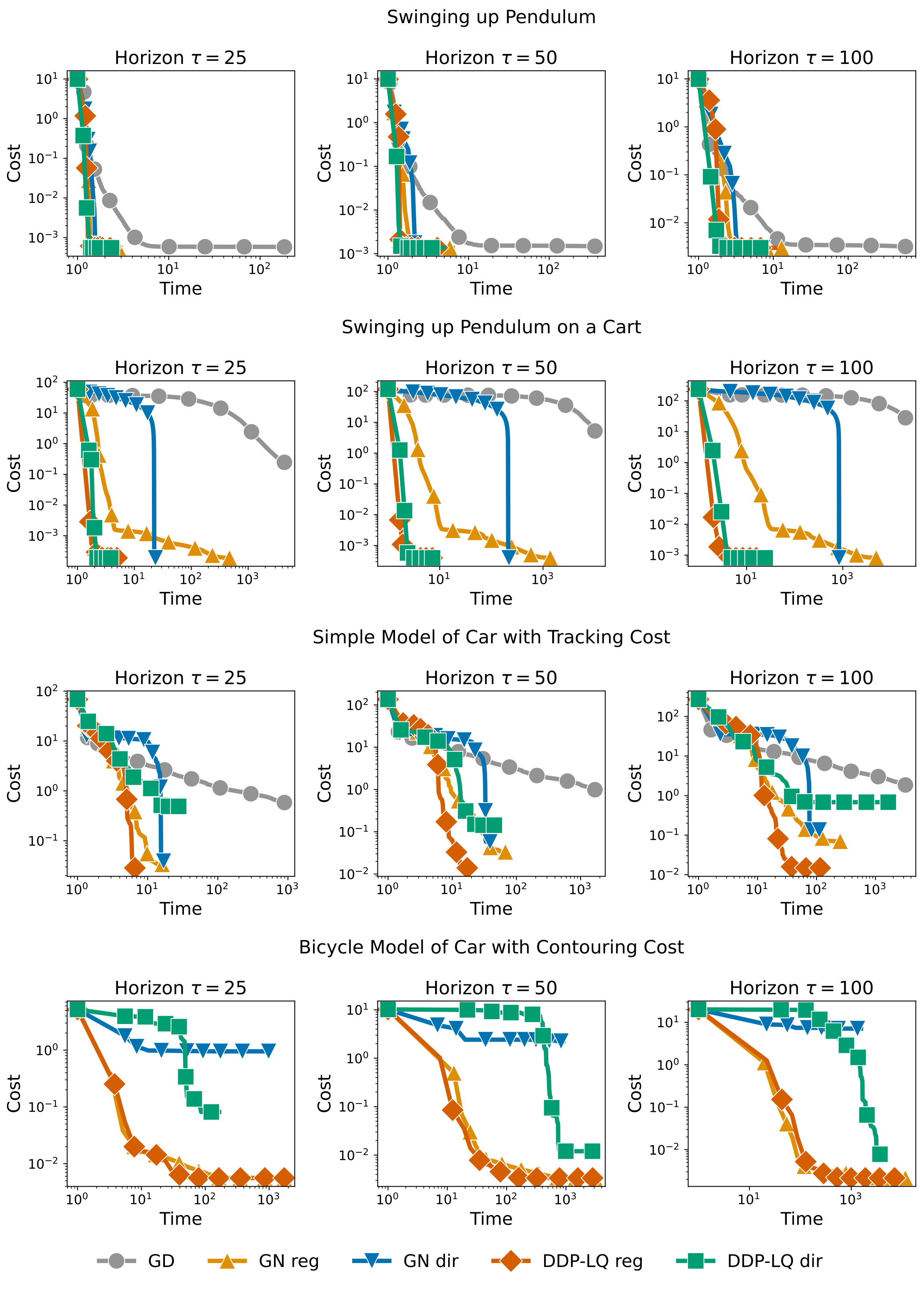


Figure 19 Cost along computational time on various control problems (see Appendix H) with algorithms using linear (GD) or linear-quadratic approximations (GN, DDP-LQ, see Figure 1 for taxonomy details) and directional (dir (43)) or regularized (reg (45)) steps.

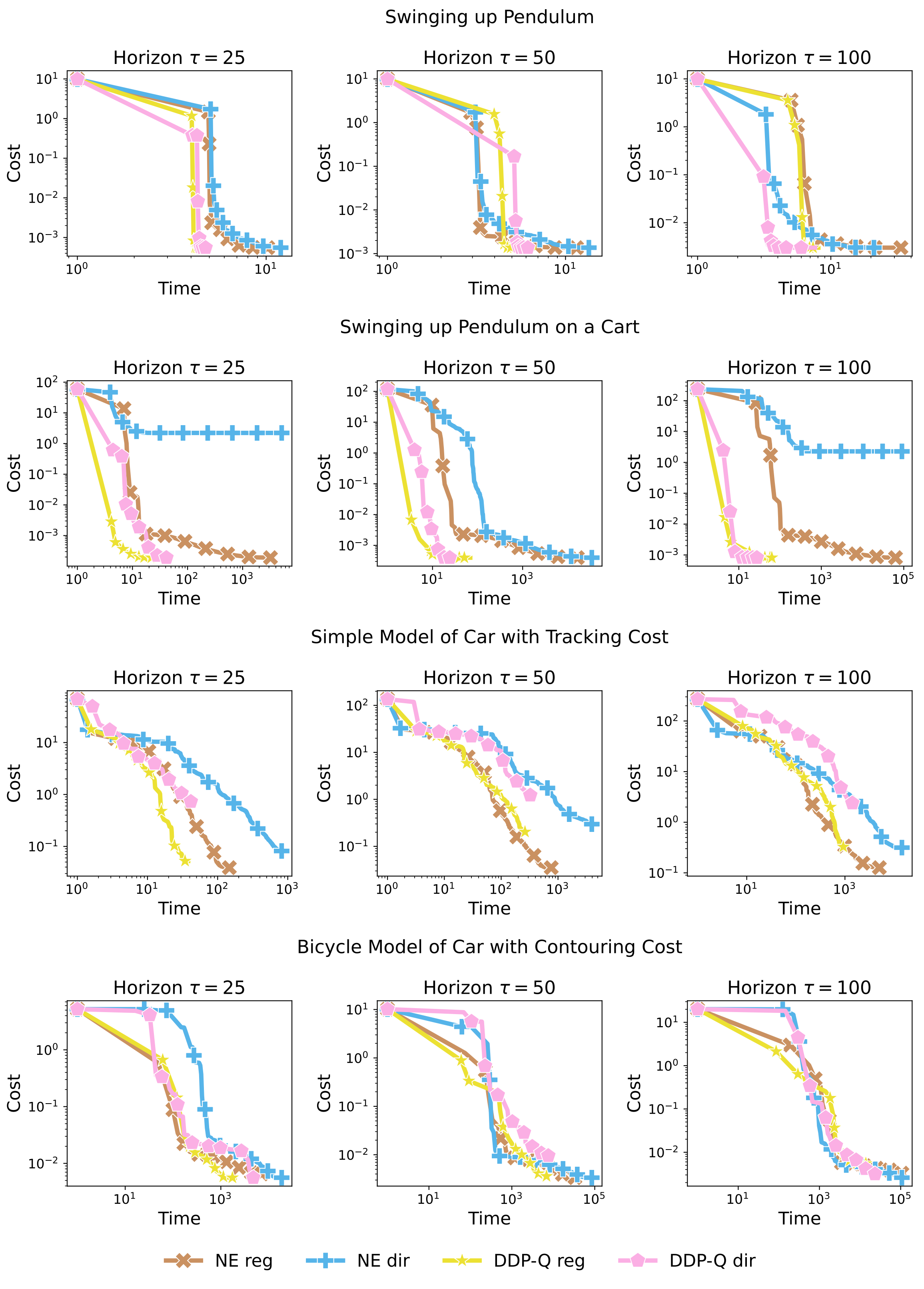


Figure 20 Cost along computational time on various control problems (see Appendix H) with algorithms using quadratic approximations (NE, DDP-Q, see Figure 1 for taxonomy details) and directional (dir (43)) or regularized (reg (45)) steps.

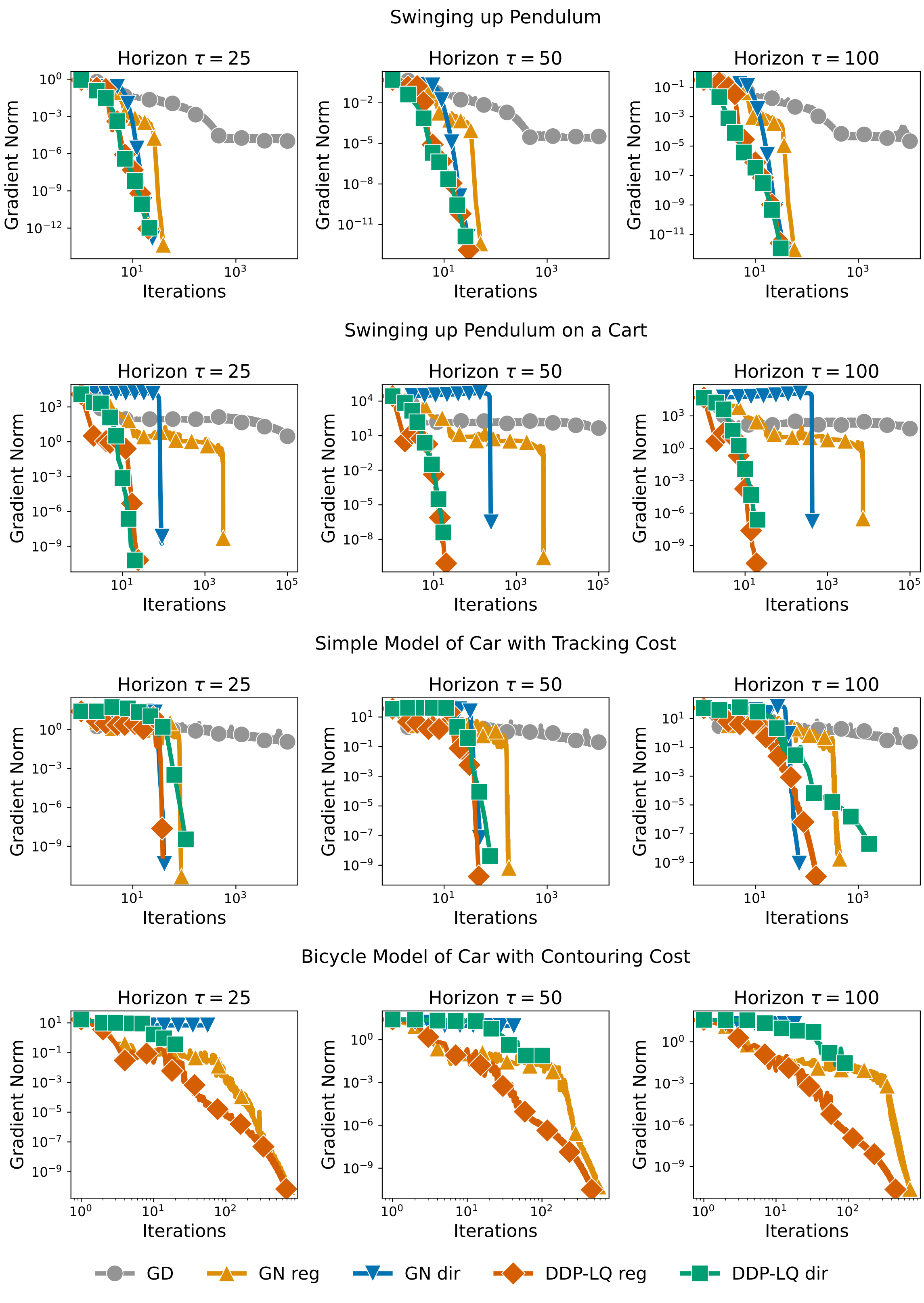


Figure 21 Gradient norm along iterations on various control problems (see Appendix H) with algorithms using linear (GD) or linear-quadratic approximations (GN, DDP-LQ, see Figure 1 for taxonomy details) and directional (dir (43)) or regularized (reg (45)) steps.

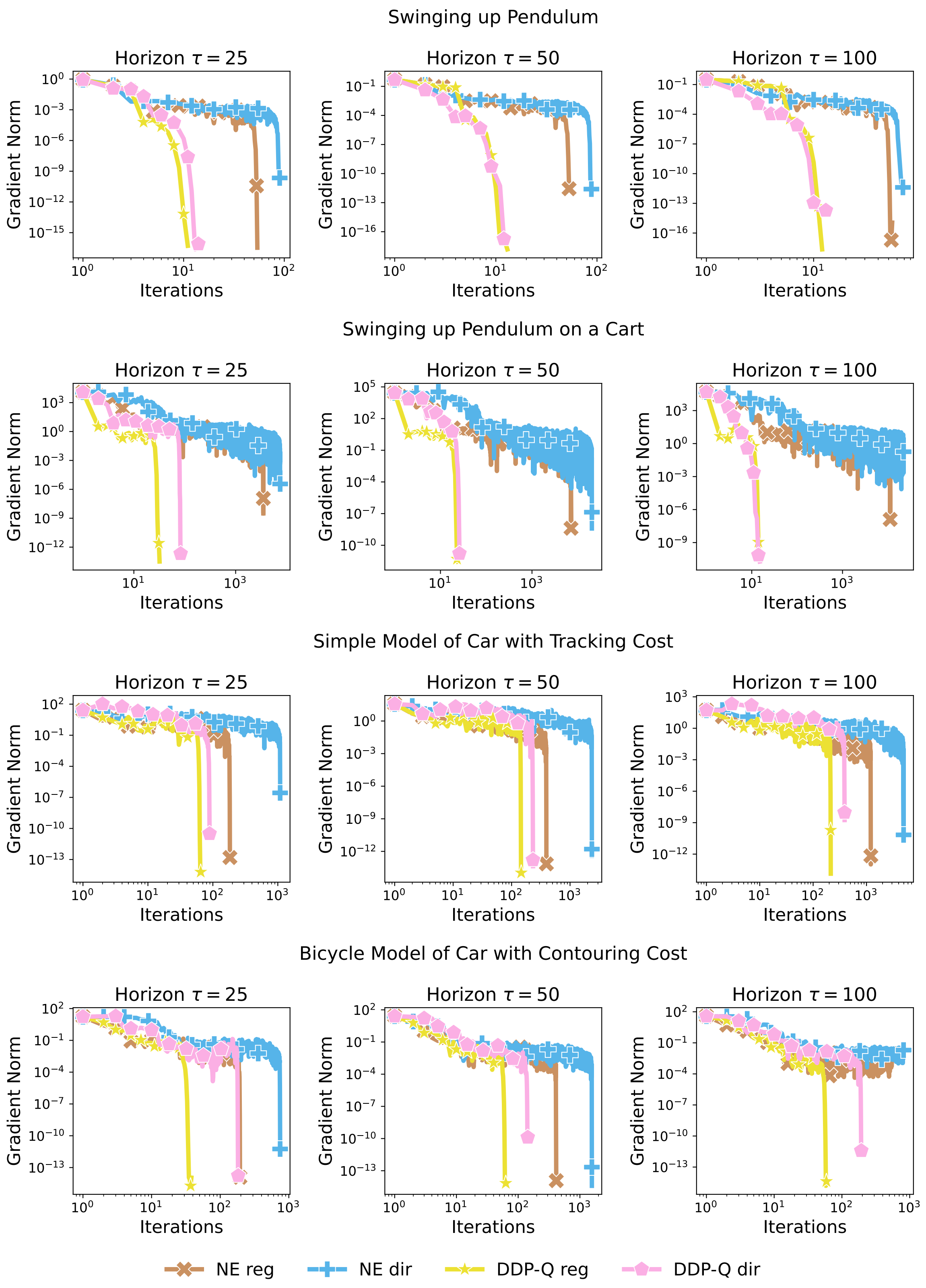


Figure 22 Gradient norm along iterations on various control problems (see Appendix H) with algorithms using quadratic approximations (NE, DDP-Q, see Figure 1 for taxonomy details) and directional (dir (43)) or regularized (reg (45)) steps.

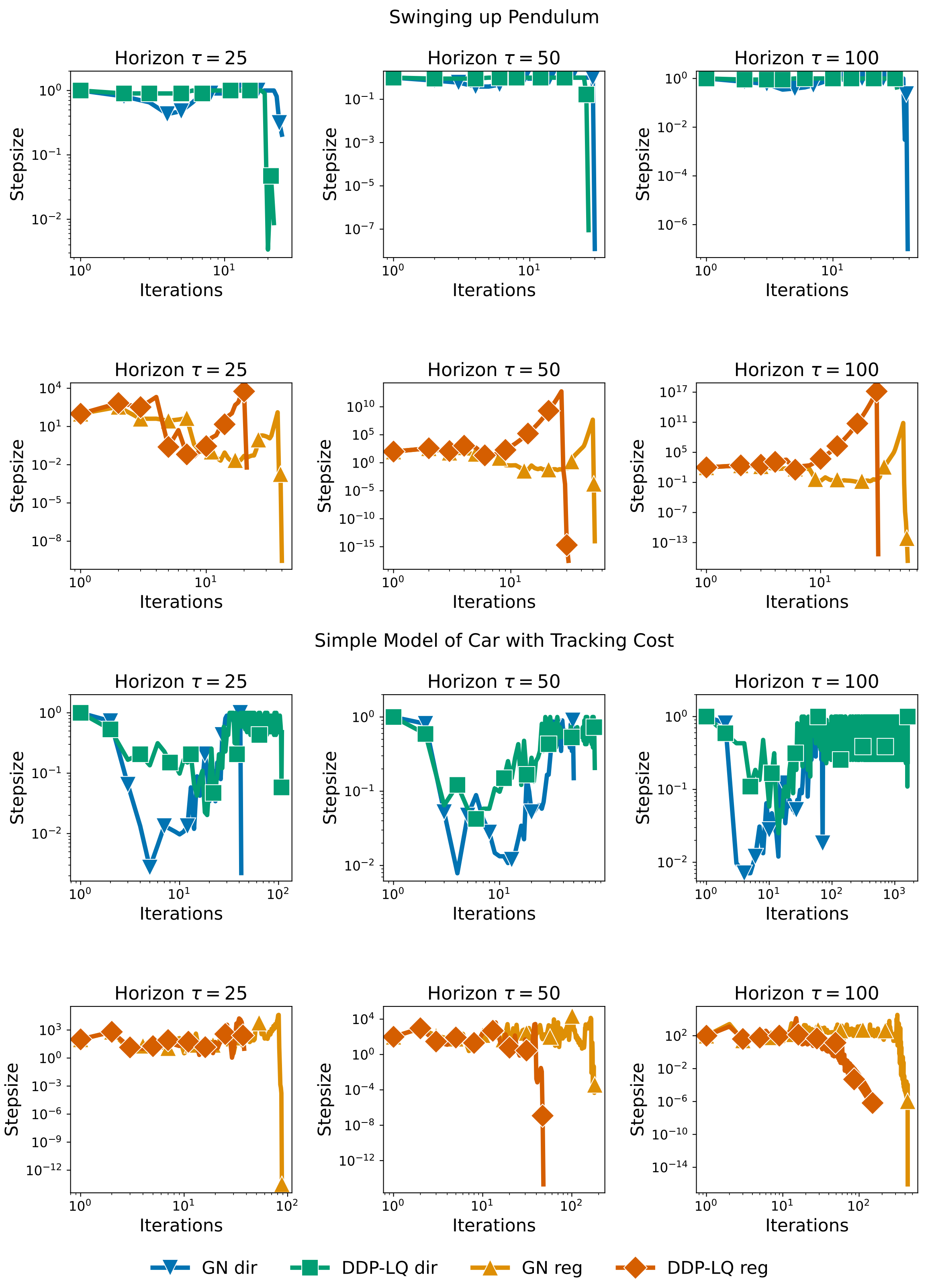


Figure 23 Stepsizes taken along the iterations on various control problems (see Appendix H) with algorithms using linear (GD) or linear-quadratic approximations (GN, DDP-LQ, see Figure 1 for taxonomy details) and directional (dir (43)) or regularized (reg (45)) steps.

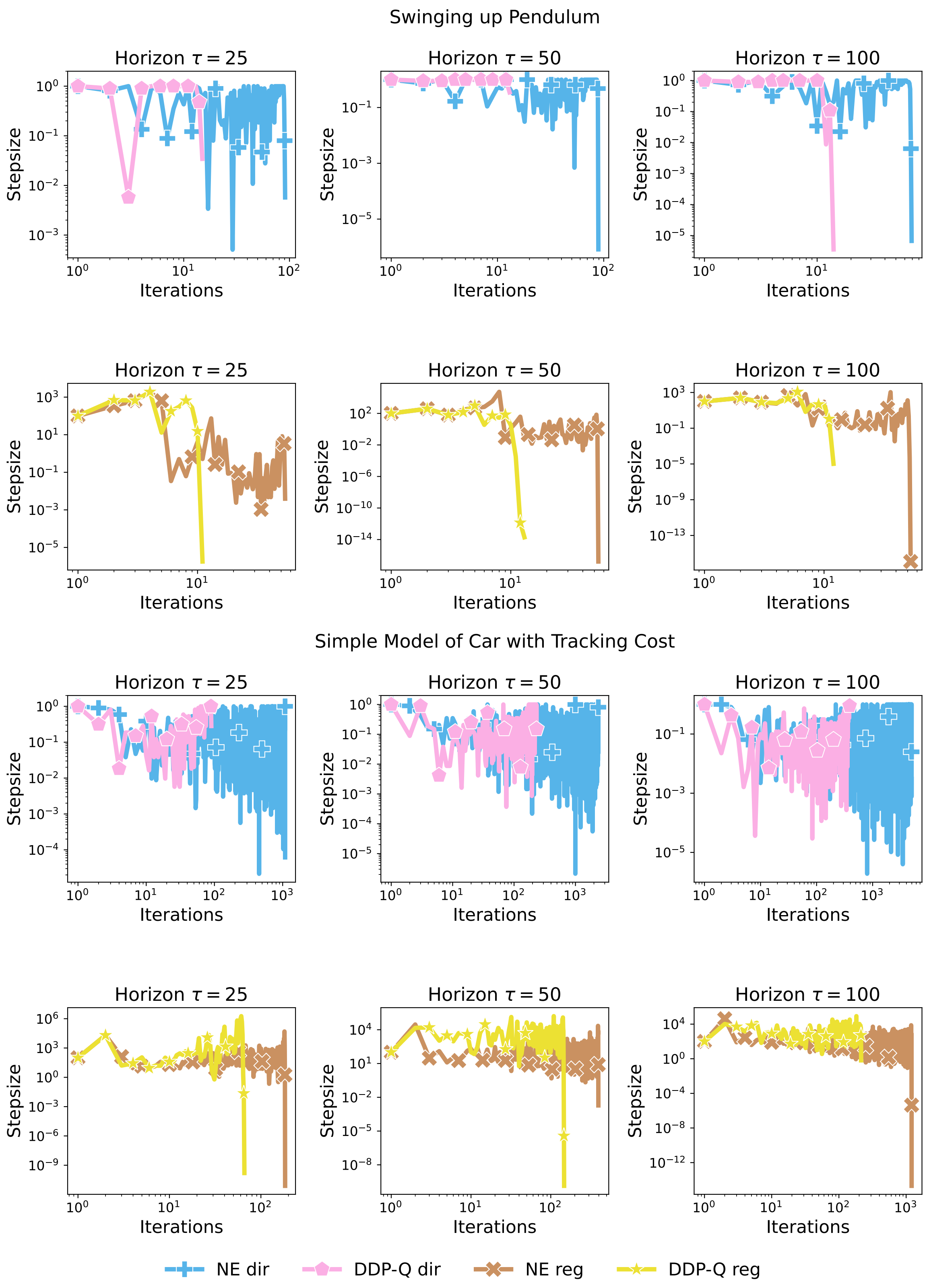


Figure 24 Stepsizes taken along the iterations on various control problems (see Appendix H) with algorithms using quadratic approximations (NE, DDP-Q, see Figure 1 for taxonomy details) and directional (dir (43)) or regularized (reg (45)) steps.

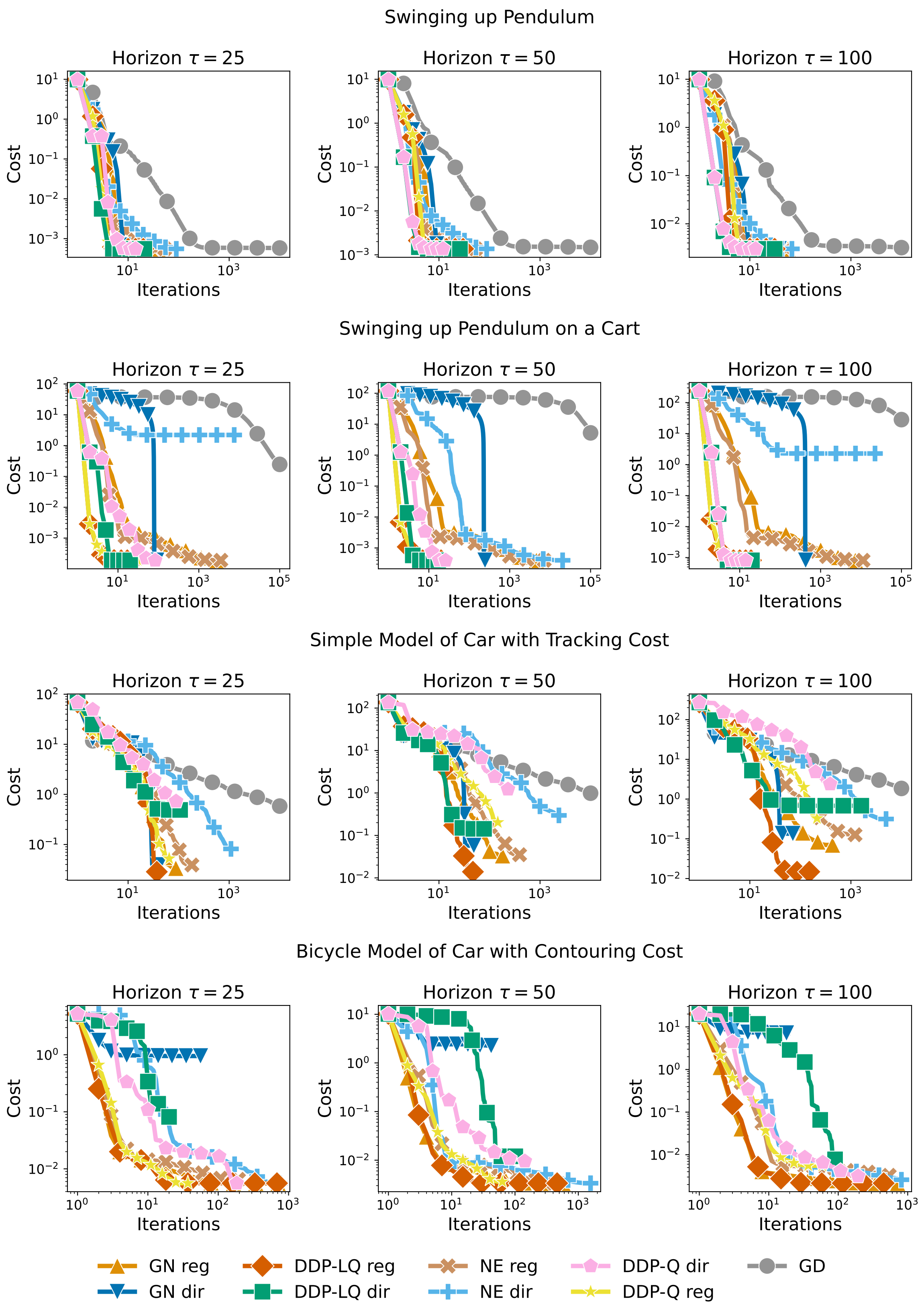


Figure 25 Cost along iterations on various control problems (see Appendix H) with the algorithms presented in Figure 1 and directional (dir (43)) or regularized (reg (45)) steps.

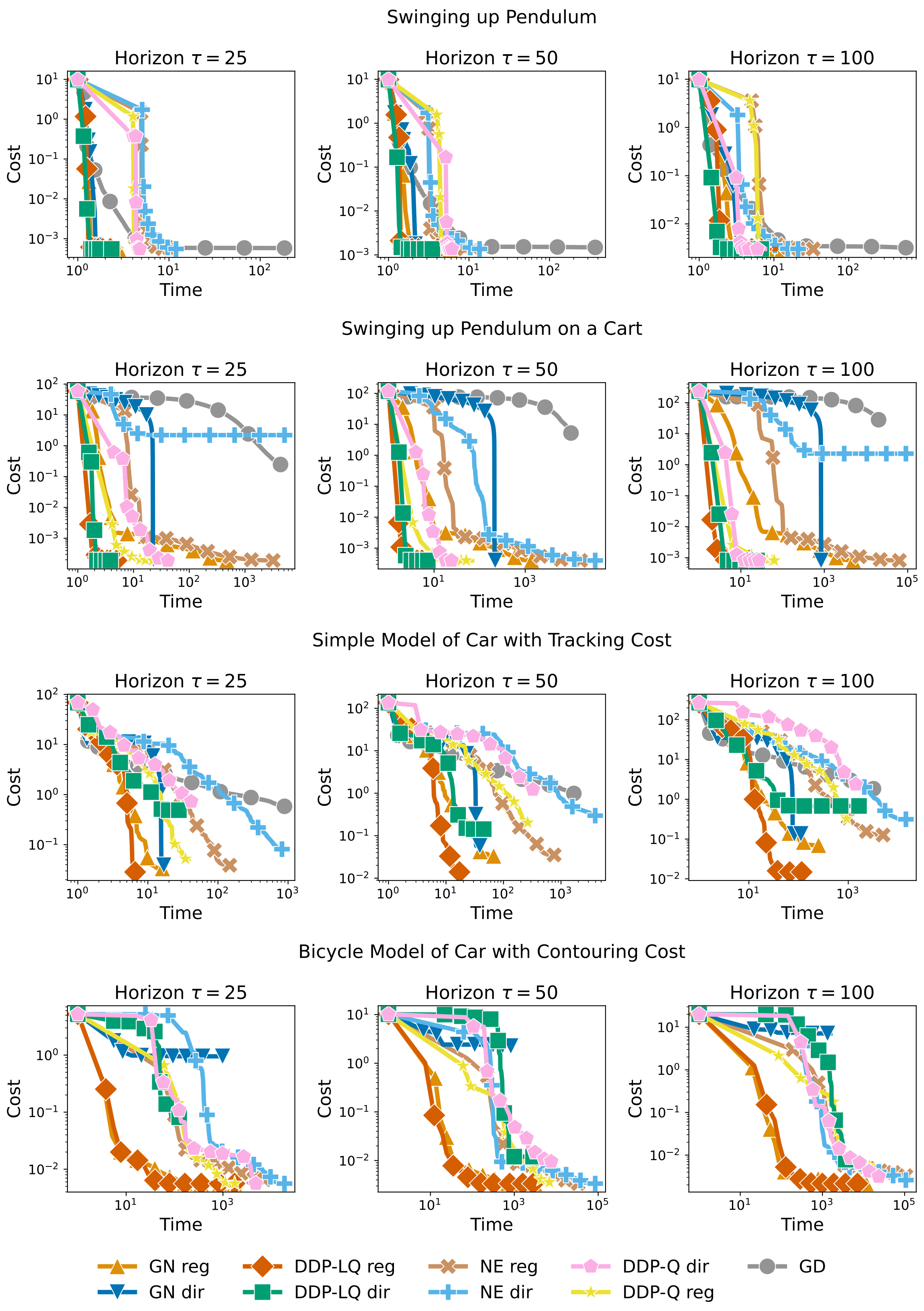


Figure 26 Cost along computational time on various control problems (see Appendix H) with the algorithms presented in Figure 1 and directional (dir (43)) or regularized (reg (45)) steps.

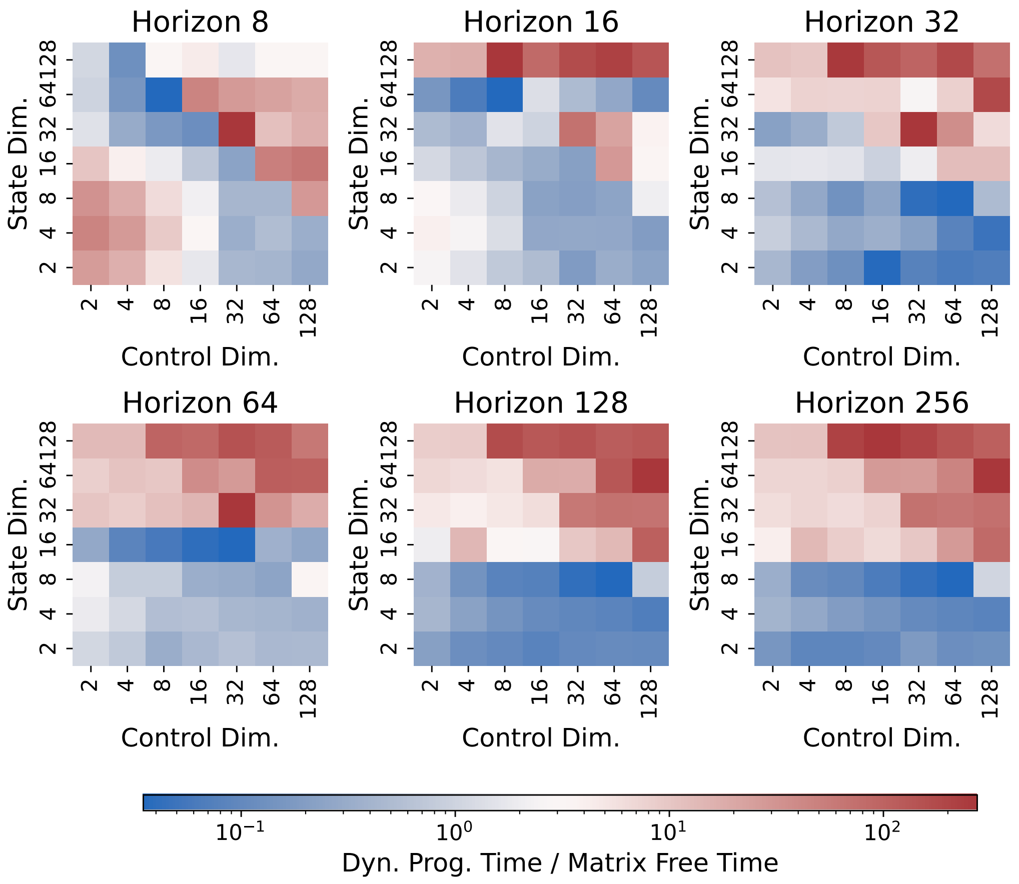


Figure 27 Comparison of time needed to solve synthetic linear quadratic control problems with either a matrix-free implementation or a dynamic programming implementation as presented in this manuscript. Blue cells indicate that dynamic programming is faster than matrix-free procedures, while red cells indicate the opposite.

References

1 Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. `https://tensorflow.org/`.

2 Joel A. E. Andersson, Joris Gillis, Greg Horn, James Rawlings, and Moritz Diehl. CasADi – A software framework for nonlinear optimization and optimal control. *Math. Program. Comput.*, 11:1–36, 2019.

3 Antoine Bambade, Sarah El-Kazdadi, Adrien Taylor, and Justin Carpentier. Prox-QP: Yet another quadratic programming solver for robotics and beyond. RSS 2022-Robotics: Science and Systems, 2022.

4 Walter Baur and Volker Strassen. The complexity of partial derivatives. *Theor. Comput. Sci.*, 22(3):317–330, 1983.

5 Atilim Gunes Baydin, Barak Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.*, 18: article no. 153 (43 pages), 2018.

6 Richard Bellman. *Introduction to the mathematical theory of control processes. Vol. II: Nonlinear processes*, volume 40-II of *Mathematics in Science and Engineering*. Academic Press Inc., 1971.

7 John Betts. *Practical methods for optimal control and estimation using nonlinear programming*. Society for Industrial and Applied Mathematics, 2010.

8 Hans Georg Bock and Karl-Josef Plitt. A multiple shooting algorithm for direct solution of optimal control problems. *IFAC Proceedings Volumes*, 17(2):1603–1608, 1984.

9 Jerome Bolte and Edouard Pauwels. A mathematical model for automatic differentiation in machine learning. In *Proceedings of the 34rd International Conference on Neural Information Processing Systems*. Curran Associates, Inc., 2020.

10 Stephen Boyd and Lieven Vandenberghe. Semidefinite programming relaxations of non-convex problems in control and combinatorial optimization. In *Communications, Computation, Control, and Signal Processing*, pages 279–287. Springer, 1997.

11 Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge University Press, 2004.

12 James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. version 0.3.13, `https://github.com/google/jax`.

13 Michael L. Bynum, Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson, and David L. Woodruff. *Pyomo–optimization modeling in python*, volume 67 of *Springer Optimization and Its Applications*. Springer, third edition, 2021.

14 Moritz Diehl, Hans Georg Bock, Holger Diedam, and P.-B. Wieber. Fast direct multiple shooting algorithms for optimal robot control. In *Fast motions in biomechanics and robotics: optimization and feedback control*, volume 340 of *Lecture Notes in Control and Information Sciences*, pages 65–93. Springer, 2006.

15 Moritz Diehl, Hans Joachim Ferreau, and Niels Haverbeke. Efficient numerical methods for nonlinear MPC and moving horizon estimation. In *Nonlinear model predictive control: towards new challenging applications*, volume 384 of *Lecture Notes in Control and Information Sciences*, pages 391–417. Springer, 2009.

16 Joseph Dunn and Dimitri Bertsekas. Efficient dynamic programming implementations of Newton's method for unconstrained optimal control problems. *J. Optim. Theory Appl.*, 63(1):23–38, 1989.

17 Iain Dunning, Joey Huchette, and Miles Lubin. JuMP: A modeling language for mathematical optimization. *SIAM Rev.*, 59(2):295–320, 2017.

18 Farbod Farshidian, Michael Neunert, Alexander W. Winkler, Gonzalo Rey, and Jonas Buchli. An efficient optimal planning and control framework for quadrupedal locomotion. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 93–100. IEEE, 2017.

19 Janick V. Frasch, Sebastian Sager, and Moritz Diehl. A parallel quadratic programming method for dynamic optimization problems. *Math. Program. Comput.*, 7(3):289–329, 2015.

20 Markus Giftthaler, Michael Neunert, Markus Stäuble, Jonas Buchli, and Moritz Diehl. A family of iterative Gauss-Newton shooting methods for nonlinear optimal control. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–9. IEEE, 2018.

21 Jean Charles Gilbert. Automatic differentiation and iterative processes. *Optim. Methods Softw.*, 1(1):13–21, 1992.

22 Philip E. Gill, Walter Murray, and Michael A. Saunders. Snopt: An SQP algorithm for large-scale constrained optimization. *SIAM Rev.*, 47(1):99–131, 2005.

23 Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, 2016.

24 Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Society for Industrial and Applied Mathematics, 2008.

25 Boris Houska and Moritz Diehl. A quadratically convergent inexact SQP method for optimal control of differential algebraic equations. *Optim. Control Appl. Methods*, 34(4):396–414, 2013.

26 David Jacobson and David Mayne. *Differential Dynamic Programming.* Elsevier, 1970.

27 Wilson Jallet, Antoine Bambade, Etienne Arlaud, Sarah El-Kazdadi, Nicolas Mansard, and Justin Carpentier. Proxddp: Proximal constrained trajectory optimization. 2023.

28 Sham Kakade, Akshay Krishnamurthy, Kendall Lowrey, Motoya Ohnishi, and Wen Sun. Information theoretic regret bounds for online nonlinear control. *Adv. Neural Inf. Process. Syst.*, 33:15312–15325, 2020.

29 Yann LeCun. A theoretical framework for back-propagation. In *1988 Connectionist Models Summer School, CMU, Pittsburg, PA*, 1988.

30 Weiwei Li and Emanuel Todorov. Iterative linearization methods for approximately optimal control and estimation of non-linear stochastic system. *Int. J. Control*, 80(9):1439–1453, 2007.

31 Li-Zhi Liao and Christine Shoemaker. Convergence in unconstrained discrete-time differential dynamic programming. *IEEE Trans. Autom. Control*, 36(6):692–706, 1991.

32 Li-Zhi Liao and Christine Shoemaker. Advantages of differential dynamic programming over Newton's method for discrete-time optimal control problems. Technical report, Cornell University, 1992.

33 Alexander Liniger, Alexander Domahidi, and Manfred Morari. Optimization-based autonomous racing of 1: 43 scale RC cars. *Optim. Control Appl. Methods*, 36(5):628–647, 2015.

34 Pierre-Louis Lions. *Generalized Solutions of Hamilton-Jacobi Equations.* Pitman, 1982.

35 Mohamed Magdy, Abdallah El Marhomy, and Mahmoud A. Attia. Modeling of inverted pendulum system with gravitational search algorithm optimized controller. *Ain Shams Engineering Journal*, 10(1):129–149, 2019.

36 David Mayne and Elijah Polak. First-order strong variation algorithms for optimal control. *J. Optim. Theory Appl.*, 16(3):277–301, 1975.

37 Florian Messerer, Katrin Baumgärtner, and Moritz Diehl. Survey of sequential convex programming and generalized Gauss–Newton methods. *ESAIM, Proc. Surv.*, 71:64–88, 2021.

38 Donald Murray and Sidney Yakowitz. Differential dynamic programming and Newton's method for discrete optimal control problems. *J. Optim. Theory Appl.*, 43(3):395–414, 1984.

39 Yurii Nesterov. *Lectures on convex optimization.* Springer, 2018.

40 John Nganga and Patrick Wensing. Accelerating Second-Order Differential Dynamic Programming for Rigid-Body Systems. *IEEE Robotics and Automation Letters*, 6(4):7659–7666, 2021.

41 Jorge Nocedal and Stephen Wright. *Numerical optimization.* Springer, 2006.

42 J. Pantoja. Differential dynamic programming and Newton's method. *Int. J. Control*, 47(5):1539–1553, 1988.

43 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pages 8026–8037. Curran Associates, Inc., 2019.

44 Elijah Polak. *Computational methods in optimization: a unified approach*, volume 77 of *Mathematics in Science and Engineering.* Academic Press Inc., 1971.

45 Christopher Rao, Stephen Wright, and James Rawlings. Application of interior-point methods to model predictive control. *J. Optim. Theory Appl.*, 99(3):723–757, 1998.

46 Benjamin Recht. A tour of reinforcement learning: The view from continuous control. *Annual Review of Control, Robotics, and Autonomous Systems*, 2:253–279, 2019.

47 Vincent Roulet, Siddhartha Srinivasa, Dmitriy Drusvyatskiy, and Zaid Harchaoui. Iterative linearized control: stable algorithms and complexity guarantees. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 5518–5527. JMLR, 2019.

48 David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

49 Jürgen Schmidhuber. Making the world differentiable: on using self supervised fully recurrent neural networks for dynamic reinforcement learning and planning in non-stationary environments. Technical report, Inst. für Informatik, 1990.

50 Athanasios Sideris and James Bobrow. An efficient sequential linear quadratic algorithm for solving nonlinear optimal control problems. In *Proceedings of the 2005 American Control Conference*, pages 2275–2280. IEEE, 2005.

51 Akshay Srinivasan and Emanuel Todorov. Graphical Newton. `https://arxiv.org/abs/1508.00952`, 2015.

52 Yuval Tassa, Tom Erez, and William Smart. Receding horizon differential dynamic programming. *Adv. Neural Inf. Process. Syst.*, 20, 2007.

53 Yuval Tassa, Tom Erez, and Emanuel Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4906–4913. IEEE, 2012.

54 Yuval Tassa, Nicolas Mansard, and Emanuel Todorov. Control-limited differential dynamic programming. In *2014*

IEEE International Conference on Robotics and Automation (ICRA), pages 1168–1175. IEEE, 2014.

55 Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 5026–5033. IEEE, 2012.

56 Robin Verschueren, Gianluca Frison, Dimitris Kouzoupis, Jonathan Frey, Niels van Duijkeren, Andrea Zanelli, Branimir Novoselnik, Thivaharan Albin, Rien Quirynen, and Moritz Diehl. acados — a modular open-source framework for fast embedded optimal control. *Math. Program. Comput.*, 14:147–183, 2022.

57 Robin Verschueren, Niels van Duijkeren, Rien Quirynen, and Moritz Diehl. Exploiting convexity in direct optimal control: a sequential convex quadratic programming method. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 1099–1104. IEEE, 2016.

58 Oskar Von Stryk. *Numerical solution of optimal control problems by direct collocation.* Springer, 1993.

59 Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program.*, 106:25–57, 2006.

60 Paul Werbos. *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting.* Wiley-Interscience, 1994.

61 Stephen Wright. Solution of discrete-time optimal control problems on parallel computers. *Parallel Comput.*, 16(2-3):221–237, 1990.

62 Stephen Wright. Partitioned dynamic programming for optimal control. *SIAM J. Optim.*, 1(4):620–642, 1991.

63 Stephen Wright. Structured interior point methods for optimal control. In *Proceedings of the 30th IEEE Conference on Decision and Control*, pages 1711–1716. IEEE, 1991.

64 Stephen Wright. Interior point methods for optimal control of discrete time systems. *J. Optim. Theory Appl.*, 77(1):161–187, 1993.

65 Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning.* Cambridge University Press, 2023.