Course of Control Problems in Robotics

Control of underactuated robots via input-constrained receding-horizon differential dynamic programming

Simone Orelli 1749732
Antonio Rapuano 2044902
Dhaval Shukla 2047562

DIAG Department
University of Rome "La Sapienza"
[surname].[id number]@studenti.uniroma1.it

October 2024

Abstract

Underactuated robots, characterized by having fewer actuators than degrees of freedom, present unique control challenges and advantages. Here, we investigate the theory behind Differential Dynamic Programming (DDP), exploiting the perks that it offers for optimally controlling nonlinear systems. By reviewing the existing literature, we refine the DDP control algorithm through regularization measures that enhance its reliability and performance, such as the exploitation of a Levenberg-Marquardt parameter and line search. Credibility is improved through the consideration of hard constraints on the control input and the combination of our algorithm with a receding horizon logic reminiscent of Model Predictive Control. We provide pseudocodes that explicate the overall control strategy step-by-step. To validate our approach, we apply the algorithm to the control of the pendubot and the acrobot, two underactuated robots with similar dynamics. Finally, the paper is endowed with a discussion of the results, which demonstrate the effectiveness of our approach in controlling underactuated robots.

Contents

1 Introduction

A fully-actuated (or even redundant) robot is capable of performing operations and accomplishing tasks with extreme accuracy and repeatability. On the other hand, when the number of actuators is less than the number of degrees of freedom, the capabilities of the robot are reduced, and it is said to be underactuated. Underactuation is not only a hurdle, but it brings a whole host of advantages, first and foremost efficiency (but also spectacularity) both in design and operation, although this comes at the cost of severely complicating their control.

A widely-used class of optimal control algorithms that allows good trajectory tracking performance of systems with a high degree of nonlinearity such as underactuated robots is that of Differential Dynamic Programming (DDP) methods. Introduced in 1966 by Mayne [1], they can be exploited to implement a receding-horizon control strategy reminiscent of Model Predictive Control [2]. The optimal control problem to be solved can also embed input constraints, which are common in robotics applications [3].

This document addresses the implementation of a controller for two among the most significative examples of underactuated robots, namely the pendubot and the acrobot. For robustness concerns, DDP is used to find optimal trajectories continuously in real-time according to a receding horizon logic, in the meantime satisfying hard constraints on the control sequence. The performance of the controller is evaluated through simulations, and then the results are discussed.

2 Differential dynamic programming

This section provides a brief overview of the Differential Dynamic Programming (DDP) algorithm. Part of what follows is reprised and recast mostly from [2].

2.1 Optimal control problem

The numerical implementation of the DDP algorithm is constructed on a time-discrete optimal control problem that encodes both the dynamics of the system and the control objective.

One possible way for the mathematical model to be discretized is to use the well-known forward Euler method.
Given a state vector x, a control input u, and the time step T, the discretized dynamics are given by:

$$x^{i+1} = F(x^i, u^i) = x^i + Tf(x^i, u^i), \tag{2.1}$$

where $f(x, u)$ is the continuous-time dynamics of the system.

The cost function to minimize comes from the sum of the running cost $l(x, u)$ and the terminal cost $l_f(x)$. Simple and common choices for these functions are quadratic forms.
Accordingly to 2.1, over a control horizon of duration N, the state sequence $X = \{x^i\}_{i=0}^{N}$ is uniquely identified by the initial state x^0 and the control sequence $U = \{u^i\}_{i=0}^{N-1}$. Therefore, the total cost is given by:

$$J(x^0, U) = \sum_{i=0}^{N-1} l(x^i, u^i) + l_f(x^N) \tag{2.2}$$

In light of the above, the optimal control problem we will be solving is the following.

$$\begin{aligned} \min_{U} \quad & J(x^0, U) = \sum_{i=0}^{N-1} l(x^i, u^i) + l_f(x^N) \\ \text{s.t.} \quad & x^{i+1} = F(x^i, u^i), \qquad i = 0, \dots, N-1 \end{aligned}$$

where the optimal control sequence U^* is the one to feed to the system.
Notice that the DDP algorithm alone does not allow to enforce a constraint on the final state, meaning that the terminal cost is crucial to guide the system towards the desired equilibrium.

2.2 Local dynamic programming

The DDP algorithm plays around the so-called *value* function, defined as the minimum cost-to-go from time i to the end of the horizon:

$$V^i(x^i) = \min_{U^i} \left\{ \sum_{j=i}^{N-1} l(x^j, u^j) + l_f(x^N) \right\},$$

where $U^i = \{u^j\}_{j=i}^{N-1}$ is the control sequence from time i to the end of the horizon. In particular:

$$V^N(x^N) = l_f(x^N). \tag{2.3}$$

According to Bellman's principle of optimality [4] and using 2.1, the value function can be rewritten as[1]:

$$V(x) = \min_{u} \left\{ l(x, u) + V'(F(x, u)) \right\}.$$

We call *Q-function* the minimization argument:

$$Q(x, u) = l(x, u) + V'(F(x, u)), \tag{2.4}$$

leading to:

$$V(x) = \min_{u} Q(x, u). \tag{2.5}$$

[1] For the sake of brevity, we henceforth omit the time index i and denote by V' the value referring to the next instant $i + 1$.

2.3 Quadratic approximation

If we consider a nominal state-control pair $(\overline{x}, \overline{u})$, the Q-function can be approximated by a second-order Taylor expansion around these:

$$Q(\overline{x}+\delta x, \overline{u}+\delta u) \approx Q(\overline{x}, \overline{u}) + Q_x^\top(\overline{x}, \overline{u})\delta x + \\ + Q_u^\top(\overline{x}, \overline{u})\delta u + \frac{1}{2}\begin{bmatrix}\delta x \\ \delta u\end{bmatrix}^\top \begin{bmatrix}Q_{xx} & Q_{xu} \\ Q_{ux} & Q_{uu}\end{bmatrix}(\overline{x}, \overline{u})\begin{bmatrix}\delta x \\ \delta u\end{bmatrix}. \tag{2.6}$$

The values of the gradient and Hessian of the Q-function can be computed by expanding also 2.4 and equating the coefficients of the same order. Therefore, we get[2]:

$$Q_x = l_x + V_x'^\top F_x, \tag{2.7}$$
$$Q_u = l_u + V_x'^\top F_u, \tag{2.8}$$
$$Q_{xx} = l_{xx} + F_x^\top V_{xx}' F_x + V_x' F_{xx}, \tag{2.9}$$
$$Q_{uu} = l_{uu} + F_u^\top V_{xx}' F_u + V_x' F_{uu}, \tag{2.10}$$
$$Q_{xu} = l_{xu} + F_x^\top V_{xx}' F_u + V_x' F_{xu}. \tag{2.11}$$

The optimal control adjustment in response to a state deviation δx is obtained by minimizing the Q-function with respect to δu, yielding the local state-affine policy:

$$\delta u^\star = k + K\delta x, \tag{2.12}$$

with coefficients given by:

$$k = -Q_{uu}^{-1}Q_u, \tag{2.13}$$
$$K = -Q_{uu}^{-1}Q_{ux}. \tag{2.14}$$

Similarly, the we can also expand the value function:

$$V(\overline{x}+\delta x) \approx V(\overline{x}) + V_x^\top(\overline{x})\delta x + \frac{1}{2}\delta x^\top V_{xx}(\overline{x})\delta x.$$

Using 2.5 with $x = \overline{x}+\delta x$ and $u = \overline{u}+\delta u$, and plugging 2.12 into 2.3, we can extract the gradient and Hessian of the value function, whose expressions, once simplified according to [5], are:

$$V_x = Q_x - K^\top Q_{uu} k, \tag{2.15}$$
$$V_{xx} = Q_{xx} - K^\top Q_{uu} K. \tag{2.16}$$

2.4 Backward and forward sweeps

Given an initial control sequence U_{init}, the first real step of the algorithm is to compute the corresponding state evolution X_{init} by recursively applying 2.1.
We can consider these as the nominal trajectories, making it possible to compute $V^N(\overline{x}^N)$ according to 2.3, which unlocks the ability to calculate the coefficients of the expansion of the Q-function around the pair $(\overline{x}^{N-1}, \overline{u}^{N-1})$ using 2.7 - 2.11. These form the gains k in 2.13 and K in 2.14, which in turn allow us to find the coefficients of the expansion of the value function $V(\cdot)$ around $(\overline{x}^{N-1}, \overline{u}^{N-1})$ with 2.15 and 2.16, and so on until the initial state x^0 is reached. This phase is called the *backward sweep*.
Subsequently, the *forward sweep* consists of recursively applying the local state-affine policy 2.12 to the state deviation $\delta x^i = x^i - \overline{x}^i$, exploiting once again 2.1 to obtain new control and state sequences from the previous (nominal) ones using:

$$x^0 = \overline{x}^0, \tag{2.17}$$
$$u^i = \overline{u}^i + k^i + K^i(x^i - \overline{x}^i). \tag{2.18}$$

The alternation of backward and forward sweeps goes on until a satisfactory solution is found.

2.5 Regularization

In this part, we focus on common weaknesses of DDP and how they can be addressed in the implementation phase.

2.5.1 Levenberg-Marquardt parameter

The Hessian of the Q-function Q_{uu} is required to be positive definite in order to guarantee the existence of a minimum. To ensure this, a regularization term is added, and the resulting modified Hessian $\tilde{Q}_{uu}$ is then used to compute the gains k and K. The regularization term is proportional to the identity matrix and is weighted by a scalar μ, that plays the role of a Levenberg-Marquardt parameter:

$$\tilde{Q}_{uu} = Q_{uu} + \mu I. \tag{2.19}$$

This corresponds to adding a quadratic term around the current control sequence, which makes the steps more conservative.
Further details can be found in [5].

2.5.2 Line search

At each forward sweep, an issue may arise if the newly found trajectory significantly diverges from the previous one, resulting in a substantial error in the quadratic approximation. To prevent this, a line search is performed to find the optimal step size α that minimizes the cost along the direction of the control adjustment. This is done by iteratively reducing (usually halving) the step size until the cost of the proposed trajectory decreases sufficiently (i.e., it becomes smaller than the cost of the nominal one), and then performing the step. Logically, if α becomes too small it means that the optimal trajectory has been found.

[2] Once again for brevity, we neglect the dependence on the nominal state-control pair $(\overline{x}, \overline{u})$.

The implementation of the line search is done by integrating α in the forward sweep, so that the control sequence is updated as:

$$u^i = \overline{u}^i + \alpha k^i + K^i(x^i - \overline{x}^i), \tag{2.20}$$

indeed replacing 2.18. Notice that the feedback term is not affected by the line search.

3 Optimal controller

The aim of this section is to design an optimal control law that exploits the DDP algorithm in such a way that a nonlinear system (such as a robot) is led to an objective optimally with respect to some cost.

3.1 Hard input constraints

Within the robotics field, actuators are often subject to saturation limits. In the case of revolute joints, the torque that can be applied is limited by the maximum torque that the motor can supply. The presence of these limitations translates into the birth of hard constraints on the input of the form of 4.2, referred to as *box constraints*.
We propose three alternatives derived from [3] to handle these constraints.

3.1.1 Naive clamping

A simple way to handle the constraints is to clamp the control input in the forward sweep to the limits of the box constraints. The operation performed elementwise on the control input vector u is:

$$\mathcal{NC}(u) = \min\{\max\{u, u_{\min}\}, u_{\max}\}.$$

As a consequence, 2.20 becomes:

$$u^i = \mathcal{NC}(\overline{u}^i + \alpha k^i + K^i(x^i - \overline{x}^i)). \tag{3.1}$$

The drawback of such a trivial intervention is that the clamped search direction may not be feasible anymore, compromising convergence.

3.1.2 Squashing functions

Another way to enforce the constraints is to use component-wise sigmoidal squashing functions, such as the following.

$$\mathcal{S}(u) = \frac{u_{\max} - u_{\min}}{2} \tanh \frac{2\,u}{u_{\max} - u_{\min}} + \frac{u_{\max} + u_{\min}}{2},$$

where the hyperbolic tangent is used to squash the input to the interval $[u_{\min}, u_{\max}]$:

$$\lim_{u \to -\infty} \mathcal{S}(u) = u_{\min}, \quad \lim_{u \to +\infty} \mathcal{S}(u) = u_{\max}.$$

Correspondigly, the update rule for the control input 2.20 becomes:

$$u^i = \mathcal{S}(\overline{u}^i + \alpha k^i + K^i(x^i - \overline{x}^i)). \tag{3.2}$$

In the cost function, the unmodified control inputs should be used in order to prevent their explosion. Altough the squashing function is differentiable, its nonlinearity is not well captured by the quadratic approximation of the dynamics that is being performed in the backward pass. This may lead to suboptimal solutions or even convergence issues.

3.1.3 Constrained quadratic programming

The best-performing method to take into account the control limitations is to condition, through the constraints, the minimization of the quadratic model of the Q-function.
At the moment, the solution 2.12 is valid in the unconstrained case. To include the constraints, the idea is to solve the following constrained quadratic program:

$$\begin{aligned} \min_{\delta u} \quad & Q(\overline{x} + \delta x, \overline{u} + \delta u) \\ \text{s.t.} \quad & u_{\min} \leq \overline{u} + \delta u \leq u_{\max}. \end{aligned}$$

With reference to 2.3, we know that the minimization argument depends both on δx and δu. Being δu the minimization variable, since δx is unknown during the backward sweep, a direct solution can be found only for the feedforward term:

$$\begin{aligned} k = \arg\min_{\delta u} \left\{ Q_u^\top \delta u + \frac{1}{2} \delta u^\top Q_{uu} \delta u \right\} \\ \text{s.t.} \quad u_{\min} \leq \overline{u} + \delta u \leq u_{\max}, \end{aligned} \tag{3.3}$$

which replaces 2.13.
In order to find the feedback term K, the Hessian Q_{uu} must be decomposed as $Q_{uu} = Q_{uu}^f + Q_{uu}^c$, where Q_{uu}^f is the submatrix corresponding to the free dimensions of the control input, and Q_{uu}^c is the submatrix corresponding to the clamped dimensions. Then, instead of 2.14, the feedback term is computed as:

$$K = -Q_{uu}^f Q_{ux}, \tag{3.4}$$

implying that the rows of K corresponding to clamped controls are identically zero.
In addition, we have found that applying the naive clamping on top of this in the forward sweep ensures the feasibility of the input even in the event that, despite the control dimension falling within the free section of Q_{uu}, the action of the feedback term on the state deviation causes the control to exceed its boundaries.

3.2 Receding-horizon control

Following the proposed scheme, the DDP algorithm can certainly identify an optimal control sequence that, combined with the nominal dynamics of the system, returns a state evolution that satisfies a high-level objective. In order to design a control law inducing the system to perform its intended task with robustness guarantees with respect to model uncertainty and possible presence of disturbances, we propose to enrich the algorithm with a receding horizon control logic, in the spirit of Model Predictive Control (MPC).

Briefly, the idea is to apply the DDP algorithm at each time step, considering the current state as the initial state, and the current control sequence as the initial guess. The first control input of the optimal sequence is then applied to the system, and the process is repeated at the next time step. This way, the control law is updated at each time step, and the system is driven to the desired state in a receding horizon fashion.

To this end, crucial is the computation of the very first control sequence, which should be carried out employing enough resources to identify an optimal (or close to the optimal) trajectory. The following iterations, on the other hand, can be performed with less computational effort, since the algorithm is already warm-started with a good initial guess, therefore allowing for real-time applications.

3.3 Implementation

We investigate the implementation of the control algorithm in the following pseudocode. The algorithm is composed of four main functions, respectively implementing the backward sweep, the forward sweep, one single iteration of the DDP algorithm and the whole receding-horizon controller.

Algorithm 1: Backward sweep (*bwd*)

Data: x^0, $U = \{u^i\}_{i=0}^{N-1}$
Result: $G = \{(k^i, K^i)\}_{i=0}^{N-1}$

Compute $X = \{x^i\}_{i=0}^{N}$ by applying recursively 2.1 to x^0 with U;
Compute $V'_x = l_{f,x}(x^N)$ and $V'_{xx} = l_{f,xx}(x^N)$ according to 2.3;
for $i = N-1, \dots, 0$ **do**
    Compute Q_u, Q_x, Q_{xx}, Q_{uu} and Q_{xu} using 2.7 - 2.11;
    If necessary, regularize Q_{uu} using 2.19;
    Compute k^i and K^i using 3.3 and 3.4;
    Compute V_x and V_{xx} using 2.15 and 2.16;
    Set $V'_x = V_x$ and $V'_{xx} = V_{xx}$;

Starting from an initial state x^0 and a control sequence U, Algorithm 1 realizes the function *bwd*, which computes the gain sequence $G = \{(k^i, K^i)\}_{i=0}^{N-1}$ by iteratively applying the constrained quadratic programming.

Given a nominal trajectory and a sequence of gains, Algorithm 2 implements a function named *fwd*, which iteratively computes the modified trajectory by applying the constrained local state-affine policy $\delta u^\star$ to the state deviation, using α as a step-size for the feedforward term.

Algorithm 2: Forward sweep (*fwd*)

Data: x^0, $\overline{U} = \{\overline{u}^i\}_{i=0}^{N-1}$, $G = \{(k^i, K^i)\}_{i=0}^{N-1}$, α
Result: $U = \{u^i\}_{i=0}^{N-1}$

Compute $\overline{X} = \{\overline{x}^i\}_{i=0}^{N}$ by applying recursively 2.1 to $\overline{x}^0 = x^0$ with $\overline{U}$;
for $i = 0, \dots, N-1$ **do**
    Compute u^i using 2.20;
    Constrain u^i using 3.1 or 3.2;
    Compute x^{i+1} using 2.1;

Algorithm 3 realizes the application of an iteration of DDP, alternating between backward and forward sweeps until the cost converges to a minimum. Moreover, this function executes the line search to find the step size α that leads to a decrease in the cost. If the cost does not decrease, being stuck in the loop signifies that we are already at the minimum, therefore the algorithm stops.

Algorithm 3: DDP iteration (*ddp*)

Data: x^0, $U_{\text{init}} = \{u^i_{\text{init}}\}_{i=0}^{N-1}$
Result: $U^\star = \{u^{\star,i}\}_{i=0}^{N-1}$

Initialize $U = U_{\text{init}}$;
repeat
    Compute the cost J using 2.2;
    Perform a backward sweep $G = bwd(x^0, U)$;
    Reset α;
    repeat
        Perform a forward sweep $U' = fwd(x^0, U, G, \alpha)$;
        Compute the cost J' using 2.2 with U';
        Halve α;
    until $\alpha \approx 0 \vee J' < J$;
    Set $U = U'$;
until $\alpha \approx 0$;
Set $U^\star = U$;

Finally, Algorithm 4 implements the previously-

discussed receding-horizon control logic. Its purpose is to iteratively apply the DDP algorithm to the system, identifying the best the control input at each time step in function of on-line measurements of the state.

Algorithm 4: Receding-horizon control

```
Initialize U_init = {0}_{i=0}^{N-1};
repeat
    Measure the current state x;
    Perform a DDP iteration
      U* = ddp(x, U_init);
    Apply the first control input u^{*,0} to the
      system;
    Set U_init = {u^{*,i}}_{i=1}^{N-1};
    Append {u^{*,N}} to U_init;
until termination condition;
```

4 Control of underactuated robots

The present section delves into the application of the strategy illustrated so far to the control of two examples of underactuated robots - the pendubot and the acrobot.

4.1 Mathematical model

The pendubot and the acrobot share the same dynamics of a 2R planar robot moving in the vertical plane. With reference to [6], which in turn quotes [7], to describe these dynamics we introduce the system of differential equations:

$$M(q)\ddot{q} + B\dot{q} + c(q,\dot{q}) + e(q) = u, \tag{4.1}$$

where $q = \begin{bmatrix} q_1 & q_2 \end{bmatrix}^\top$ is the vector of joint angles, $\dot{q}$ and $\ddot{q}$ are the joint velocity and acceleration, respectively, and $u = \begin{bmatrix} u_1 & u_2 \end{bmatrix}^\top$ is the control input.
Moreover, $M(q)$ is the inertia matrix:

$$M(q) = \begin{bmatrix} a_1 + 2a_2c_2 & a_3 + a_2c_2 \\ a_3 + a_2c_2 & a_3 \end{bmatrix},$$

B is the (positive definite) dissipation matrix:

$$B = \begin{bmatrix} b_1 & 0 \\ 0 & b_2 \end{bmatrix},$$

$c(q,\dot{q})$ is the Coriolis and centrifugal vector:

$$c(q,\dot{q}) = \begin{bmatrix} a_2s_2\dot{q}_2(\dot{q}_2 + 2\dot{q}_1) \\ a_2s_2\dot{q}_1^2 \end{bmatrix},$$

and $e(q)$ is the gravity vector:

$$e(q) = \begin{bmatrix} a_4s_1 + a_5s_{12} \\ a_5s_{12} \end{bmatrix}.$$

Here, we use the compact notation:

$$\begin{aligned} s_1 &= \sin(q_1), & c_1 &= \cos(q_1), \\ s_2 &= \sin(q_2), & c_2 &= \cos(q_2), \\ s_{12} &= \sin(q_1 + q_2), & c_{12} &= \cos(q_1 + q_2), \end{aligned}$$

and the (positive) constants:

$$\begin{aligned} a_1 &= I_{1zz} + m_1d_1^2 + I_{2zz} + m_2(l_1^2 + d_2^2), \\ a_2 &= m_2l_1d_2, \\ a_3 &= I_{2zz} + m_2d_2^2, \\ a_4 &= g(m_1d_1 + m_2l_1), \\ a_5 &= gm_2d_2. \end{aligned}$$

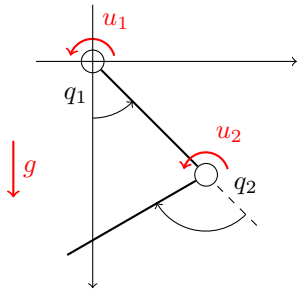

Figure 4.1: Graphical representation of a 2R planar robot in the vertical plane.

The only difference between the two robots lies in the actuation, since the only actuated joint of the pendubot is the "shoulder" (implying that $u_2 = 0$), while that of the acrobot is the "elbow"(therefore $u_1 = 0$).

4.2 Control objective

The equilibria of the robot are found by setting all the derivatives of the generalized coordinates to zero. Among these, the unforced ones are identified by solving the equation:

$$e(q) = 0,$$

which corresponds to the condition of balance of the gravity forces, yielding the four solutions:

$$q_{uu} = \begin{bmatrix} \pi \\ 0 \end{bmatrix}, \quad q_{ud} = \begin{bmatrix} \pi \\ \pi \end{bmatrix}, \quad q_{du} = \begin{bmatrix} 0 \\ \pi \end{bmatrix}, \quad q_{dd} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

The control objective is to stabilize the robot in the upright position q_{uu} with zero final velocity, starting from the initial conditions $q_0 = q_{dd}$ and $\dot{q}_0 = 0$. This is known as the *swing-up problem*, and it is a challenging task due to the underactuation of the system, since the control input is constrained to the actuated joint, and the other joint is passive.

In addition, the control input is subject to the (component-wise) inequality constraint:

$$u_{\min} \leq u \leq u_{\max}, \tag{4.2}$$

to be intended as a saturation limit on the torque that can be applied to the actuated joint. Obviously, the input is also subject to the physical constraint $u_2 = 0$ for the pendubot and $u_1 = 0$ for the acrobot, and therefore $u_{\min}$ must be non-positive and $u_{\max}$ non-negative.

4.3 Control strategy tailoring

The control strategy illustrated so far can be adapted to the specific case of the pendubot and the acrobot by considering the following.

4.3.1 Dynamics

First of all, we have to translate our nonlinear dynamics 4.1 in terms of 2.1. To this end, we define the state vector:

$$x = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix}^\top = \begin{bmatrix} q^\top & \dot{q}^\top \end{bmatrix}^\top.$$

As far as the control input is concerned, we use:

$$u = \begin{bmatrix} u_1 & 0 \end{bmatrix}^\top$$

for the pendubot and:

$$u = \begin{bmatrix} 0 & u_2 \end{bmatrix}^\top$$

for the acrobot. Therefore, the dynamics of the robot, rewritten in terms of 2.1, are given by defining the 4×1 state transition function:

$$f(x,u) = \begin{bmatrix} x_3 \\ x_4 \\ M^{-1}(x)\left(-B\begin{bmatrix} x_3 \\ x_4 \end{bmatrix} - c(x) - e(x) + u\right) \end{bmatrix}.$$

Finally, according to this new representation, the swing-up problem undergoes the following modifications:

- the initial state is $x_0 = \begin{bmatrix} q_{dd}^\top & 0 & 0 \end{bmatrix}^\top$;
- the (final) reference state is $x^* = \begin{bmatrix} q_{uu}^\top & 0 & 0 \end{bmatrix}^\top$.

4.3.2 Cost function

Another important object to tailor is the cost function 2.2, and, for this purpose, a common and simple choice is to use quadratic functons. In particular, we select:

$$\begin{aligned} l(x,u) &= (x - x^*)^\top P (x - x^*) + u^\top R u, \\ l_f(x) &= (x - x^*)^\top P^N (x - x^*), \end{aligned}$$

where P, R and P^N are positive definite matrices. The choice of the cost function is fundamental, since it determines the behavior of the control law. In this case, in the spirit of our control objective, we choose P and P^N to penalize the deviation from the equilibrium position, as well as R to weight the control effort.
In conclusion, the total cost function is modified accordingly:

$$\begin{aligned} J(x^0, U) = \sum_{i=0}^{N-1} (x^i - x^*)^\top P^i (x^i - x^*) + u^{i\top} R u^i + \\ + (x^N - x^*)^\top P^N (x^N - x^*). \end{aligned}$$

Notice that, since J is evaluated over the current control horizon, the weight matrix P^i should change with each iteration to accomodate the slide of the final cost P^N.

4.4 Simulations

The current section is intended to present the results that we obtained.
The two closed loops, each consisting of the complete controller and one of the two robots (whose parameters are summarized in 4.1), have been implemented and simulated in *MATLAB R2024a*.
The experiments consist of running the code for a total time of 5 seconds, with time steps of $T = 0.01$ seconds. The (receding) control horizon of the controller is set to $N = 300$, which corresponds to 3 seconds.
We have also tested, for each robot, the three different methods for constraining the input, aiming to highlight their differences.

Parameter	Symbol	Value
Gravity	g	9.81 $[\mathrm{m\,s^{-2}}]$
Link length	l	0.5 [m]
Link mass	m	2 [kg]
Link inertia	I	$\frac{1}{12}ml^2$ $[\mathrm{kg\,m^2}]$
Link COM	d	$\frac{l}{2}$ [m]

Table 4.1: Robot parameters

4.4.1 Pendubot

Recall that the model of the pendubot is obtained by not actuating the second joint of the 2R vertical planar robot in Figure 4.1, i.e., by setting u_2 identically equal to zero. Meanwhile, the limits imposed on the actuated joint are $u_{\min} = -5\,\mathrm{N\,m}$ and $u_{\max} = 5\,\mathrm{N\,m}$.

Figure 4.2 shows the trajectories of the pendubot under the three different constraining strategies: notice that, other than the case with no constraints, only

the constrained quadratic programming method is able to stabilize the robot in the upright position (as illustrated also in Figure 4.3). With the naive clamping approach, it is evident (especially at the beginning) that the control sequence is a sharply cut version of the unclamped sequence, which is not adequate to reach the goal; using the squashing function, the smoothing effect on the control curve is evident, the nonlinearity of which, however, puts the controller in a difficult position to find a useful control trajectory.

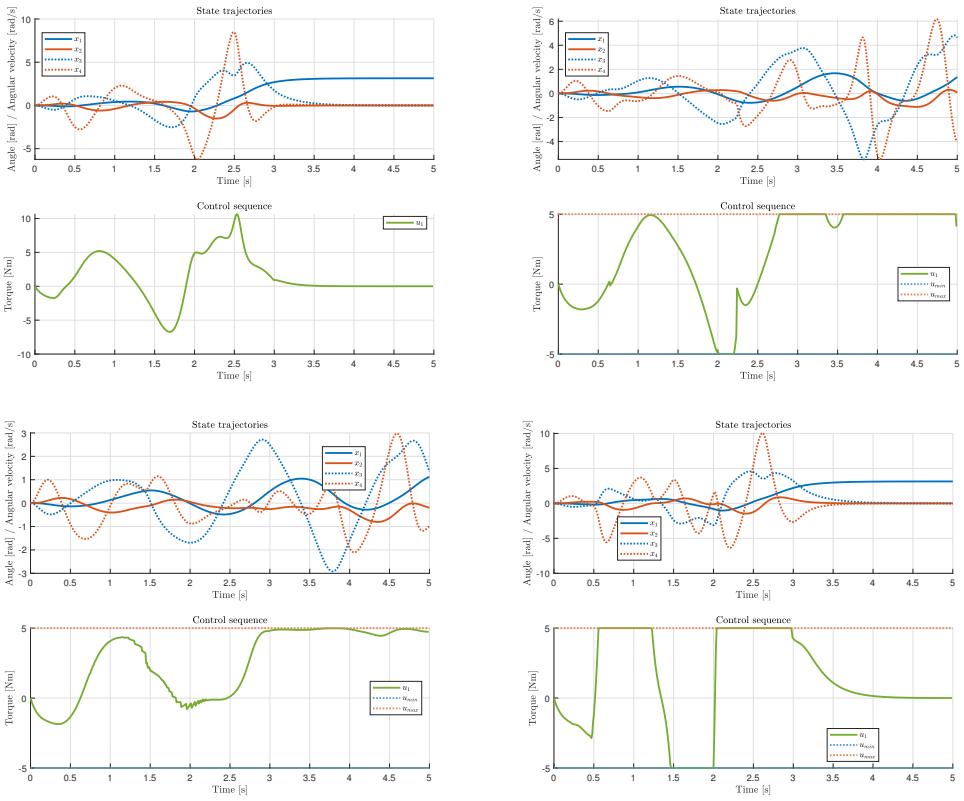

Figure 4.2: Trajectories of the pendubot under: no constraints (top left), naive clamping (top right), squashing function (bottom left), constrained quadratic programming (bottom right).

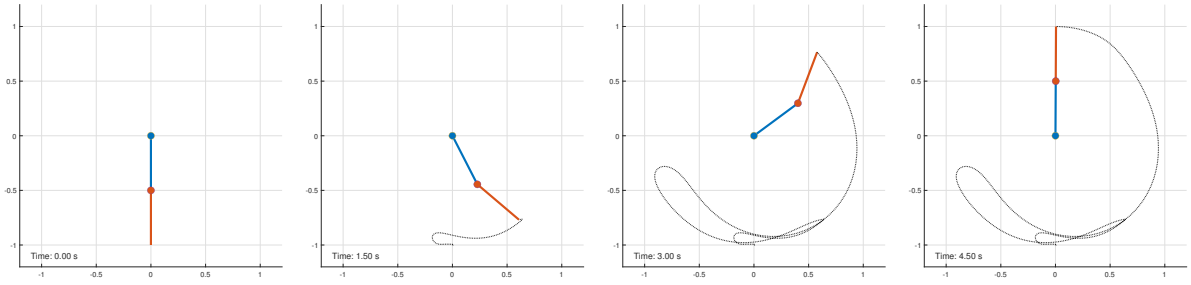

Figure 4.3: Animation frames of the pendubot swing-up motion under constrained quadratic programming.

4.4.2 Acrobot

Conversely, the model of the acrobot is retreived by not actuating the first joint of the robot in Figure 4.1, i.e., by setting u_1 identically equal to zero. This time, the limits imposed on the actuated joint are $u_{\min} = -3.5\,\mathrm{N\,m}$ and $u_{\max} = 3.5\,\mathrm{N\,m}$, which are less restrictive than the previous case due to the different physical properties of the robot.

Just as it was done for the pendubot, the trajectories of the acrobot under the three different con-

straining strategies are depicted in Figure 4.4, and the animation of the swing-up motion under constrained quadratic programming is shown in Figure 4.5. The results are consistent with the previous expreriments, as the constrained quadratic programming method is the only one able to perform the swing-up motion, besides the case with no constraints. In addition to this, note the fact that, when employing naive clamping or the squashing function, the control sequences are similar to each other, are not too close to the extremes of the admissibility interval, and yet are unable to achieve the intended goal: evidently, the constraining methods have penalized the search for the optimal solution during the controller iterations, reflecting the disadvantages that were pointed out in the related section.

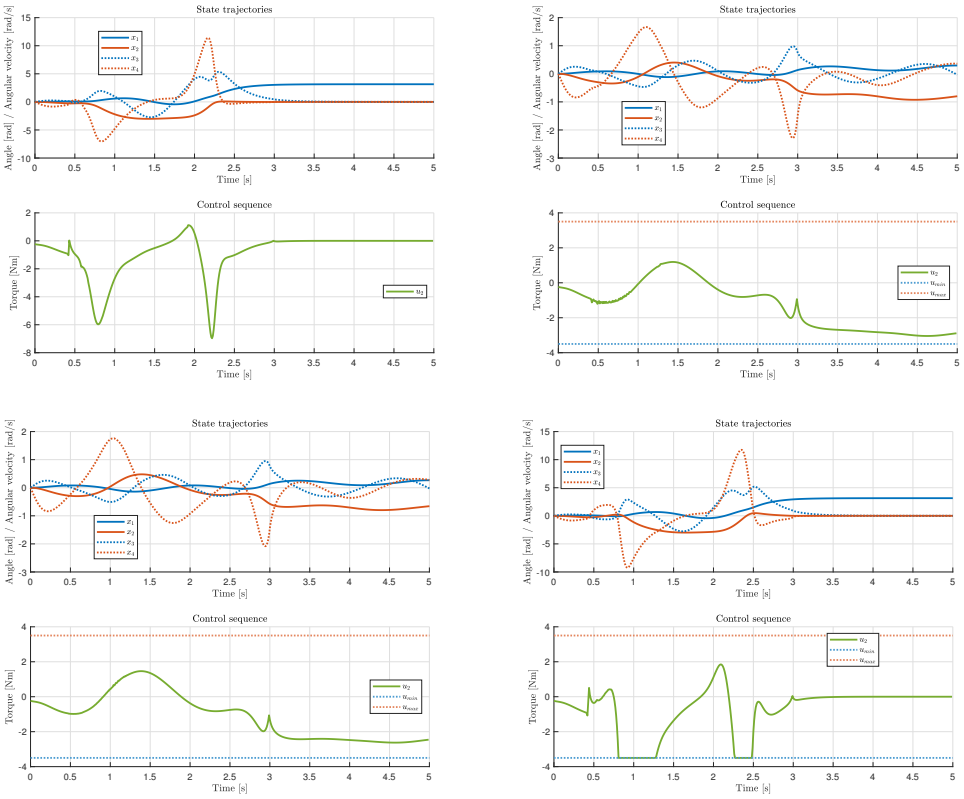

Figure 4.4: Trajectories of the acrobot under: no constraints (top left), naive clamping (top right), squashing function (bottom left), constrained quadratic programming (bottom right).

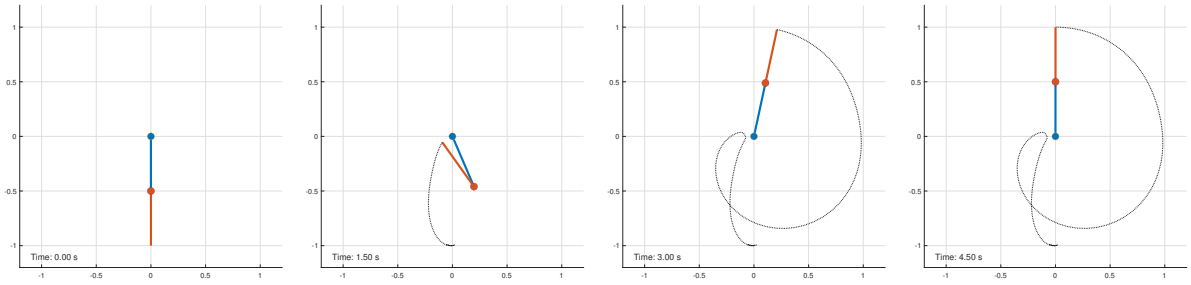

Figure 4.5: Animation frames of the acrobot swing-up motion under constrained quadratic programming.

5 Conclusion

This paper presents a comprehensive approach to the control of underactuated robots using input-constrained receding-horizon Differential Dynamic Programming. The study focuses on the pendubot and the acrobot, which are exemplary underactuated systems that highlight the challenges and advantages of such control strategies.

The DDP algorithm, renowned for its efficiency in handling nonlinear systems, is adapted to accommodate hard input constraints through three primary methods: naive clamping, squashing functions, and constrained quadratic programming. Among these, the constrained quadratic programming approach proves superior, effectively managing the swing-up problem of both robots while adhering to input constraints. Moreover, the implementation of a receding-horizon strategy, akin to Model Predictive Control, further enhances the real-time applicability of the DDP algorithm.

Simulation results validated the effectiveness of the proposed method. Whereas naive clamping and squashing functions failed to achieve the desired swing-up motions (because the first method causes a blockage of some search directions, while the second one introduces unrequired and harmful nonlinearities), the constrained quadratic programming method managed to attain the objective, demonstrating both accuracy and efficiency. These simulations not only confirmed the theoretical advantages but also underscored the practical feasibility of the approach.

In summary, the research illustrates that input-constrained receding-horizon DDP is a robust and efficient method for controlling underactuated robots. It effectively balances the need for constraint satisfaction with the performance demands of nonlinear control tasks. Future work could extend this approach to more complex systems and explore integration with other control methodologies to further improve robustness and adaptability.

References

[1] David Q. Mayne. "A Second-order Gradient Method for Determining Optimal Trajectories of Non-linear Discrete-time Systems". In: *International Journal of Control* 3.1 (1966), pp. 85–95. DOI: 10.1080/00207176608921369.

[2] Yuval Tassa, Tom Erez, and William Smart. "Receding Horizon Differential Dynamic Programming". In: *Advances in Neural Information Processing Systems*. Ed. by J. Platt et al. Vol. 20. Curran Associates, Inc., 2007. URL: https://proceedings.neurips.cc/paper_files/paper/2007/file/c6bff625bdb0393992c9d4db0c6bbe45-Paper.pdf.

[3] Yuval Tassa, Nicolas Mansard, and Emo Todorov. "Control-limited differential dynamic programming". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014, pp. 1168–1175. DOI: 10.1109/ICRA.2014.6907001.

[4] Richard Bellman. *Dynamic Programming*. Dover Publications, 1957. ISBN: 9780486428093.

[5] 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*. 2012, pp. 4906–4913. DOI: 10.1109/IROS.2012.6386025.

[6] Leonardo Lanari and Giuseppe Oriolo. *The Pendubot*. Course handouts. Handouts for the course "Control Problems in Robotics". 2024.

[7] Bruno Siciliano et al. *Robotics: Modelling, Planning and Control*. 1st. Springer Publishing Company, Incorporated, 2008. ISBN: 1846286417.