Trust Region and Riccati Recursion for NMPC

Stephen Monnet (*327447*)
Supervised by : Shaohui Yang
Professor : Colin Jones

January 16, 2023

Contents

1 Introduction

1.1 Project Context

This project has been conducted at the Laboratoire d'Automatique (LA), which is part of the Ecole Polytechnique Fédérale De Lausanne (EPFL), during fall semester 2022-23. It has been supervised by Shaohui Yang and Prof. Colin Jones and is valued at 10 ECTS credits in the Robotics Master.

1.2 Summary

The main objective of this project is to develop a solver able to handle Non-linear Model Predictive Control (NMPC) problems using trust-region method. The method uses Sequential Quadratic Programming (SQP) but solves the Quadatric Programming (QP) sub-problem using an improved version of Riccati recursion that incorporate a trust-region criterion. After each forward and backward propagation, the solver will verify that the solution is inside the trust radius. If that is not the case, a modification of the eigenvalues will try to enforce the solution to remain into the trust radius.
The idea of using this trust region method is that, for highly non-linear systems (such as switched-time ones), the convergence to an optimal solution might be faster since it chooses both the direction and the amplitude of the Newton step at the same time, while in standard line search methods, these two decisions are generally decoupled. Indeed, since a SQP iteration computes a Newton step based on a second order approximate of the Lagrangian, if the generated step is too big, it may lead to an increase of the cost.
The first part of this report presents the theoretical background required for this project. Based on this latter, it then proposes a formulation to apply the trust-region and Riccati recursion method to an equality constrained Non-linear Optimal Control Problem (NOCP). Then, it provides and explains the implementation of the algorithm which has been done on Matlab. Finally, the algorithm is tested and validated on several SISO and MIMO systems and its solutions are compared to those from the interior-point solver Ipopt, using CasADi. The tested systems include also a switched-time system, which indeed introduces many non-linearities in the problem.
The results obtained show that our solver is able to converge to the same solution as Ipopt within a comparable amount of Newton steps.

2 Theoritical Background

This section presents an overview of the main theoretical aspects required for this project.

2.1 Karush-Kuhn-Tucker Conditions

The Karush-Kuhn-Tucker (KKT) conditions are necessary conditions for an optimal point [1], solution of a possibly constrained problem. To illustrate it, let us consider the following optimization problem :

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & h(x) = 0 \\ & g(x) \leq 0 \end{aligned}$$

With decision variable $x \in \mathbb{R}^{n_x}$, objective function $f : \mathbb{R}^{n_x} \to \mathbb{R}$ and constraints function $h : \mathbb{R}^{n_x} \to \mathbb{R}^{n_h}$ and $g : \mathbb{R}^{n_x} \to \mathbb{R}^{n_g}$. The Lagrangian is defined as :

$$\mathcal{L}(x, \lambda, \nu) := f(x) + \lambda^T g(x) + \nu^T h(x) \tag{1}$$

Where $\lambda \in \mathbb{R}_+^{n_g}$ and $\nu \in \mathbb{R}^{n_h}$ are the dual variables. Then, the KKT conditions state that an optimal point (x^*, λ^*, ν^*) satisfies the following constraints :

$$\nabla_x \mathcal{L}(x^*, \lambda^*, \nu^*) = 0 \tag{2a}$$

$$\begin{aligned} h(x^*) &= 0 \\ g(x^*) &\leq 0 \end{aligned} \tag{2b}$$

$$\lambda^* \geq 0 \tag{2c}$$

$$\lambda^{*T} g(x^*) = 0 \tag{2d}$$

Which are called stationarity (2a), primal feasibility (2b), dual feasibility (2c) and complementary slackness (2d).

2.2 Sequential Quadratic Programming

Sequential Quadratic Programming (SQP) is a method that might be used to iteratively solve a discrete Optimal Control Problem (OCP) by using an approximate of the stationarity (2a) and primal feasibility (2b) conditions around a current guess [2]. Let's consider the following equality constrained OCP with non-linear objective function $f : \mathbb{R}^{n_x} \to \mathbb{R}$, constraint $h : \mathbb{R}^{n_x} \to \mathbb{R}^{n_h}$, and decision variable $x \in \mathbb{R}^{n_x}$.

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & h(x) = 0 \end{aligned} \tag{3}$$

Introducing the dual variable $\lambda \in \mathbb{R}^{n_h}$, the Lagrangian of this problem is :

$$\mathcal{L}(x, \lambda) = f(x) - \lambda^T h(x)$$

For an optimal solution (x^*, λ^*), the stationarity and primal feasibility conditions impose that :

$$\nabla_x \mathcal{L}(x^*, u^*) = \nabla_x f(x^*) - \lambda^{*T} \nabla_x h(x^*) = 0 \tag{4a}$$

$$h(x^*) = 0 \tag{4b}$$

Nevertheless, solving these two equations might be very difficult since they can be arbitrarily non-linear. Fortunately, a solution can be found by applying a line search method on these conditions. Let's denote by (x^k, λ^k) the primal and dual variables guess at the k-th line search iteration. The value (x^{k+1}, λ^{k+1}) at the next iteration can be obtained by linearizing the equations around the current guess. That is :

$$\nabla_x \mathcal{L}(x^{k+1}, u^{k+1}) \approx \nabla_x \mathcal{L}(x^k, \lambda^k) + \nabla_{xx} \mathcal{L}(x^k, \lambda^k) \Delta x^k + \nabla_{x\lambda} \mathcal{L}(x^k, \lambda^k) \Delta \lambda^k = 0$$

$$h(x^{k+1}) \approx h(x^k) + \nabla_x h(x^k)\Delta x^k = 0$$

Where $\Delta x^k = x^{k+1} - x^k$ and $\Delta\lambda^k = \lambda^{k+1} - \lambda^k$ are the Newton's step for primal and dual variables respectively. In matrix form, that is :

$$\begin{bmatrix} \nabla_x f(x^{k+1}) - \lambda^{k+1T}\nabla_x h(x^{k+1}) \\ h(x^{k+1}) \end{bmatrix} \approx \begin{bmatrix} \nabla_x \mathcal{L}(x^k, \lambda^k) \\ h(x^k) \end{bmatrix} + \begin{bmatrix} \nabla^2_{xx}\mathcal{L}(x^k, \lambda^k) & -\nabla_x h(x^k)^T \\ \nabla_x h(x^k) & 0 \end{bmatrix} \begin{bmatrix} \Delta x^k \\ \Delta\lambda^k \end{bmatrix} = 0 \quad (5)$$

It can be used to compute the Newton step $(\Delta x^k, \Delta\lambda^k)$, from which one can compute (x^{k+1}, λ^{k+1}). Repeating this procedure will lead to an optimal solution (x^*, λ^*) which satisfies (for a given tolerance) the primal feasibility and stationarity conditions (4), if the Hessian of the Lagrangian is positive definite (i.e. $\nabla_{xx}\mathcal{L}(x^k, \lambda^k) \succ 0$).
An interesting aspect of this formulation is that it is equivalent to the following equality constrained QP :

$$\begin{aligned} \min_x \quad & \Delta(x^k)^T \nabla^2_{xx}\mathcal{L}(x^k, \lambda^k)\, \Delta x^k + \nabla_x \mathcal{L}(x^k, \lambda^k)^T\, \Delta x^k + c \\ \text{s.t.} \quad & h(x^k) + \nabla_x h(x^k)\, \Delta x^k = 0 \end{aligned} \quad (6)$$

Which consists of minimizing a quadratic approximation of the Lagrangian under a linearized version of $h(x)$ around the current guess x^k. Indeed, deriving the KKT conditions of (6) will lead to :

$$\nabla_x \mathcal{L}(x^k, \lambda^k) + \nabla_{xx}\mathcal{L}(x^k, \lambda^k)\Delta x^k - (\Delta\lambda^k)^T\nabla_x h(x^k) = 0$$
$$h(x^k) + \nabla h(x^k)\Delta x^k = 0$$

2.3 Riccati Recursion

To illustrate the Riccati recursion algorithm, let's take a Linear Quadratic Regulator (LQR) formulation as an example, knowing the initial state $x_0 \in \mathbb{R}^{n_x}$ and assuming $Q, Q_N, R \succeq 0$

$$\begin{aligned} \min_{\mathbf{U}} \quad & \frac{1}{2}\sum_{i=0}^{N-1} \left\{x_i^T Q x_k + u_i^T R u_i\right\} + \frac{1}{2}x_N^T Q_N x_N \\ \text{s.t.} \quad & x_{i+1} = Ax_i + Bu_i, \qquad i = 0, ..., N-1 \end{aligned} \quad (7)$$

With control inputs $\mathbf{U} = \{u_i \in \mathbb{R}^{n_u}\}_{i=0}^{N-1}$, states $\mathbf{X} = \{x_i \in \mathbb{R}^{n_x}\}_{i=0}^{N}$. The idea of Riccati recursion is, at each stage i, to consider x_i as constant and optimize through u_i. The expression of the u_i^* (optimal u_i) with respect to x_i are recursively found with backward iterations $i = N, N-1, ..., 0$. Then, knowing all the expressions $u_i^* = u_i^*(x_i)$ and x_0, the optimal states x_i^* of the system can be computed with a forward propagation from $i = 0, ..., N-1$.

First, assume that the optimal cost at stage $i+1$ is of the form :

$$V_{i+1}(x_{i+1}) = \frac{1}{2}x_{i+1}^T H_{i+1} x_{i+1} = \frac{1}{2}(Ax_i + Bu_i)^T H_{i+1}(Ax_i + Bu_i) \quad (8)$$

At $i = N$, the stage cost is independent of $\mathbf{U}$ if we consider x_N as a constant, that is :

$$V_N^*(x_N) = \frac{1}{2}x_N^T Q_N x_N \rightarrow H_N = Q_N$$

From $i = N-1$ the cost is :

$$V_{N-1}(x_{N-1}) = \frac{1}{2}x_{N-1}^T Q x_{N-1} + \frac{1}{2}u_{N-1}^T R u_{N-1} + V_N(x_N)$$

$$= \frac{1}{2}x_{N-1}^T Q x_{N-1} + \frac{1}{2}u_{N-1}^T R u_{N-1} + \frac{1}{2}(Ax_{N-1} + Bu_{N-1})^T H_N (Ax_{N-1} + Bu_{N-1}) \quad (9)$$

Which can be minimized through u_{N-1} by setting $\partial V_{N-1}/\partial u_{N-1} = 0$ since Q, R and $H_N = Q_N$ are positive semi-definite. It gives an optimal input u_{N-1}^* which depends on x_{N-1} :

$$u_{N-1}^{*T} R + (Ax_{N-1} + Bu_{N-1})^T H_N B = 0$$

$$\rightarrow u_{N-1}^{*T}(R + B^T H_N B) = -x_N^T A^T H_N B$$

With R symmetric it leads to :

$$u_{N-1}^* = -(R + B^T H_N B)^{-1} B^T H_N A x_{N-1} \tag{10}$$

Defining the feedback gain matrix $K_{N-1} \in \mathbb{R}^{n_x \times n_u}$ as :

$$K_{N-1} := -(R + B^T H_N B)^{-1} B^T H_N A \tag{11}$$

Leads to the standard feedback gain command law :

$$u_{N-1}^*(x_{N-1}) = K_{N-1} x_{N-1} \tag{12}$$

Then, replacing this expression in (9) gives an expression which depends only on x_{N-1} :

$$\frac{1}{2} x_{N-1}^T \left[Q + K_{N-1}^T R K_{N-1} + (A + BK_{N-1})^T H_N (A + BK_{N-1}) \right] x_{N-1}$$

Where one can easily identify H_{N-1} :

$$H_{N-1} = Q + K_{N-1}^T R K_{N-1} + (A + BK_{N-1})^T H_N (A + BK_{N-1}) \tag{13}$$

Repeating this backward iteration will allow to compute $H_i \ \forall i = N, ..., 0$ and deduce its associated feedback gain K_i at each stage using the following algorithm :

Backward Pass : Start with $H_N = Q_N$ and iterate from $i = N-1, ..., 0$:

$$\begin{aligned} K_i &= -(R + B^T H_{i+1} B^T)^{-1} B^T H_{i+1} A \\ H_i &= Q + K_i^T R K_i + (A + BK_i)^T H_{i+1} (A + BK_i) \end{aligned} \tag{14}$$

Forward Pass : Start with x_0 and iterate from $i = 0, ..., N-1$:

$$\begin{aligned} u_i^* &= K_i x_i^* \\ x_{i+1}^* &= A x_i^* + B u_i^* \end{aligned} \tag{15}$$

At the end of the backward and forward pass, one get the optimal control inputs $\mathbf{u}^* = [u_0^*, u_1^*, ..., u_{N-1}^*]$ and the optimal states trajectories $\mathbf{x}^* = [x_1^*, x_2^*, ..., x_N^*]$. Note that this algorithm can easily be extended to time-varying system and objective function with first and zero order terms [3].

2.4 Trust Region Method

The idea of using an approximate model during the optimization step is very convenient since it offers a general way of optimizing any multivariate non-linear function. Nevertheless, in the context of SQP, taking the second order approximation of a function around a point may lead to large error in highly non-linear region. In fact, computing a descent direction with an approximated model may lead to an increase of the cost of the true function under certain conditions. Even worse, if the Hessian of the objective function is indefinite, it may lead the optimal solution to $-\infty$, which would provide an unusable Newton step.

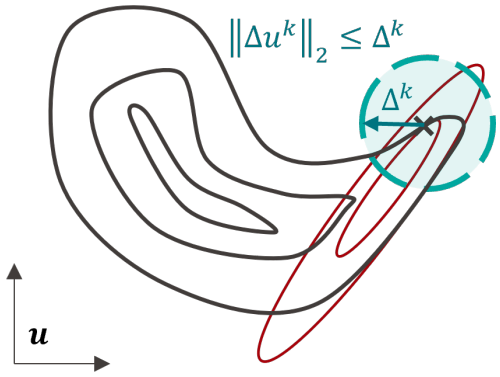

Figure 1: Illustration of a non-linear objective function approximated by a quadratic function around a specific point. Even though the approximation may give good results around this point, a too big step size Δu^k may lead to a decrease of the true objective function.

Therefore, it seems reasonable to limit the region in which we trust our approximation to avoid bad optimization step. Let us consider the following QP with, as objective function, a quadratic approximation $m^k(\mathbf{u})$ of a function $f(\mathbf{u})$ around $\mathbf{u} = \mathbf{u}^k$.

$$\begin{aligned} \min_{\mathbf{\Delta u}} \quad & \frac{1}{2}\mathbf{\Delta u}^T\mathbf{H}\mathbf{\Delta u} + \mathbf{\Delta u}^T\mathbf{g} + \mathbf{c} \\ \text{s.t.} \quad & \|\mathbf{\Delta u}\|_2 \leq \Delta \end{aligned} \tag{16}$$

The quality of the approximation can be evaluated with the *trustworthiness*, defined as the ratio between the actual and predicted reduction [4] :

$$\rho^k = \frac{f(\mathbf{u}^k) - f(\mathbf{u}^k + \mathbf{\Delta u})}{m^k(\mathbf{u}^k) - m^k(\mathbf{u}^k + \mathbf{\Delta u})} \tag{17}$$

$\rho^k \approx 1$ means a very thrustworthy model and if $\rho^k > 0$, the original objective function is decreasing (i.e. $f(\mathbf{u}^k + \mathbf{\Delta u}) < f(\mathbf{u}^k)$). This metric can therefore be used to control the trust region radius Δ. If the approximate model is close to the real one, the radius can be increased. [4] provides a generic trust-region algorithm structure :

Algorithm 1 Trust Region Algorithm

Require: $x_0, \Delta_{max}, \Delta_0, \epsilon > 0, \eta \in [0, \frac{1}{4}]$
  $k \leftarrow 0$
  $\Delta \leftarrow \Delta_0$
  while $\|\nabla f(\mathbf{u}^k)\| > \epsilon$ **do**
    Solve the trust region sub-problem (16) and compute ρ^k using (17)
    if $\rho^k < \frac{1}{4}$ **then**
      $\Delta \leftarrow \frac{1}{4}\Delta$ ▷ *(bad model : reduce radius)*
    else if $\rho^k > \frac{3}{4}$ **then**
      $\Delta \leftarrow \min(2\Delta, \Delta_{max})$ ▷ *(good model : increase radius)*
    end if
    Decide on acceptance of step
    if $\rho^k > \eta$ **then**
      $\mathbf{u}^{k+1} \leftarrow \mathbf{u}_k + \mathbf{\Delta u}$
    else
      $\mathbf{u}^{k+1} \leftarrow \mathbf{u}^k$
    end if
    $k \leftarrow k + 1$
  end while
  $\mathbf{u}^* \leftarrow \mathbf{u}^k$

2.5 Trust Region Method in Riccati Recursion

This method proposes to incorporate the trust-region method along with a Riccati recursion, and apply it to a NOCP [5]. The hope is that, for very non-linear systems, limiting the solution of our sub-problem into a trust radius will improve the solving efficiency.
Let us consider the following time-varying LQR formulation :

$$\begin{aligned} \min_{u_0,u_1,\dots,u_{N-1}} \quad & \sum_{i=0}^{N-1} \frac{1}{2} \begin{bmatrix} u_i \\ x_i \end{bmatrix}^T \begin{bmatrix} R_i & S_i^T \\ S_i & Q_i \end{bmatrix} \begin{bmatrix} u_i \\ x_i \end{bmatrix} + \begin{bmatrix} u_i \\ x_i \end{bmatrix}^T \begin{bmatrix} r_i \\ q_i \end{bmatrix} + \frac{1}{2} x_N^T Q_N x_N + x_N^T q_N \\ \text{s.t.} \quad & x_{i+1} = A_i x_i + B_i u_i + b_i, \qquad i = 0, ..., N-1 \\ & x_0 \text{ given} \end{aligned} \tag{18}$$

With states $x_i \in \mathbb{R}^{n_x}$, inputs $u_i \in \mathbb{R}^{n_u}$, weight matrices $R_i \in \mathbb{R}^{n_u \times n_u}$, $Q_i \in \mathbb{R}^{n_x \times n_x}$, $S_i \in \mathbb{R}^{n_x \times n_u}$ and weight vectors $r_i \in \mathbb{R}^{n_u}$, $q_i \in \mathbb{R}^{n_x}$. The system dynamics is described by matrices $A_i \in \mathbb{R}^{n_x \times n_x}$, $B_i \in \mathbb{R}^{n_x \times n_u}$ and vectors $b_i \in \mathbb{R}^{n_x}$.
By expressing the x_i in terms of the u_i, one can remove these from the objective function and get a new unconstrained optimization problem:

$$\min_{u_0,u_1,\dots,u_{N-1}} \quad \mathbf{u}^T \mathbf{H} \mathbf{u} + \mathbf{u}^T \mathbf{g} + \mathbf{c} \tag{19}$$

In fact, the objective function is exactly the same as equation 16, except that in this latter the trust-region radius constraint is added. The objective of this method is to exploit the structure of (18) knowing that it is equivalent to (19). The proposed algorithm in [5] is the following:

(i) Given $\lambda > -\lambda_1$ with λ_1 to be the smallest eigenvalue of H and $\Delta > 0$.

(ii) Run the following backward Riccati recursion :

$$\begin{aligned} &\text{start with} P_N = Q_N, p_N = q_N \\ &\text{for } i = N-1, N-2, ..., 1, 0 \text{ do :} \\ &\quad \bar{R}_i = R_i + \lambda I + B_i^T P_{i+1} B_i \\ &\quad \Lambda_i = \text{chol}(\bar{R}_i) \text{ s.t. } \Lambda_i^T \lambda_i = \bar{R}_i \\ &\quad L_i = \Lambda_i^{-T} (S_i + B_i^T P_{i+1} A_i \\ &\quad P_i = Q_i + A_i^T P_{i+1} A_i - L_i^T L_i \\ &\quad l_i = \Lambda_i^{-T} (r_i + B_i^T (P_{i+1} b_i + p_{i+1}) \\ &\quad p_i = q_i + A_i^T (P_{i+1} b_i + p_{i+1}) - L_i^T l_i \\ &\text{end loop} \end{aligned} \tag{20}$$

(iii) Run the forward propagation :

$$\begin{aligned} &\text{start with } x_0 \\ &\text{for } i = 0, 1, ..., N-1 \text{ do} \\ &\quad u_i = -\Lambda_i^{-1} (L_i x_i + l_i) \\ &\quad x_{i+1} = A_i x_i + B_i u_i + b_i \\ &\text{end loop} \end{aligned} \tag{21}$$

(iv) If $\|u\|_2 \leq \Delta$ stop. If not, run the backward update for $q = \{q_i\}_{i=0}^{N-1}$:

$$
\begin{aligned}
&\text{start with } \beta_{N-1} = 0 \\
&\text{for } i = N-1, N-2, ..., 0 \text{ do} \\
&\qquad \alpha_i = u_i + B_i^T \beta_i \\
&\qquad q_i = \Lambda_i^{-T} \alpha_i \\
&\qquad \beta_{i-1} = A_i^T \beta_i - L_i^T \Lambda_i^{-T} \alpha_i \\
&\text{end loop}
\end{aligned}
\tag{22}
$$

Compute the new value of λ :

$$
\lambda = \lambda + \frac{\|u\|_2^2}{\|q\|_2^2} \frac{\|u\|_2 - \Delta}{\Delta} \tag{23}
$$

And go to step (ii).

The objective of this project is to implement this algorithm in *Matlab* and test it on multiple systems, including switched-time systems.

3 Trust Region and Riccati Recursion : Formulation for an Equality Constrained NOCP

This section presents the theoretical development of a Trust Region, Riccati recursion-based solver to an equality constrained NOCP.
As a first step, let us consider the non-linear system $\dot{x} = f(x, u)$ with $x \in \mathbb{R}^{n_x}$, $u \in \mathbb{R}^{n_u}$ and $f : \mathbb{R}^{n_x \times n_u} \to \mathbb{R}^{n_x}$. The objective is to first apply SQP along with trust region method using Riccati recursion to the following optimal control problem :

$$\begin{aligned} \min_{\mathbf{X},\mathbf{U}} \quad & \sum_{i=0}^{N-1} l(x_i, u_i) + V_f(x_N) \\ \text{s.t.} \quad & x_0 - \bar{x} = 0, \\ & x_i + f(x_i, u_i)\delta t - x_{i+1} = 0, \qquad i = 0, ..., N-1 \end{aligned} \tag{24}$$

With $\mathbf{X} = \{x_i\}_{i=0}^{N}$ and $\mathbf{U} = \{u_i\}_{i=0}^{N-1}$ and δt the sampling time. Note that the following developments and notations are heavily based on [6].

3.1 Lagrangian & KKT conditions

Introducing the Lagrange multipliers $\mathbf{\Lambda} = \{\lambda_i \in \mathbb{R}^{n_x}\}_{i=0}^{N}$ with respect to the dynamical constraints, the Lagrangian is therefore :

$$\begin{aligned} \mathcal{L}(\mathbf{X}, \mathbf{U}, \mathbf{\Lambda}) = & \sum_{i=0}^{N-1} l(x_i, u_i) + V_f(x_N) \\ -\lambda_0^T(x_0 - \bar{x}) + & \sum_{i=0}^{N-1} \lambda_{i+1}^T(x_i + f(x_i, u_i)\delta t - x_{i+1}) \end{aligned} \tag{25}$$

Then, the KKT's stationnarity condition impose that $\nabla_{\mathbf{X}}\mathcal{L} = \mathbf{0}$ and $\nabla_{\mathbf{U}}\mathcal{L} = \mathbf{0}$. To ease the derivation let's reorganize the notation :

$$\mathcal{L}(\mathbf{X}, \mathbf{U}, \mathbf{\Lambda}, \mathbf{V}) = \sum_{i=0}^{N-1} \left\{ l(x_i, u_i) + \lambda_{i+1}^T f(x_i, u_i)\delta t \right\} + V_f(x_N) - \lambda_0^T x_0 + \sum_{i=0}^{N-1} \lambda_{i+1}^T x_i - \sum_{i=1}^{N} \lambda_i^T x_i + \lambda_0^T \bar{x}$$

To ease notation, let us define the Hamiltonian $\mathcal{H}(x_i, u_i, \lambda_{i+1}) := l(x_i, u_i) + \lambda_{i+1}^T f(x_i, u_i)\delta t$. The following conditions are obtained :

$$\begin{aligned} r_{x,N} &:= \nabla_x V_f(x_N) - \lambda_N = 0 \\ r_{x,i} &:= \nabla_x \mathcal{H}(x_i, u_i, \lambda_{i+1}) + \lambda_{i+1} - \lambda_i = 0 \qquad i = 0, ..., N-1 \\ r_{u,i} &:= \nabla_u \mathcal{H}(x_i, u_i, \lambda_{i+1}) = 0 \qquad i = 0, ..., N-1 \end{aligned}$$

3.2 Newton Steps

Let's now define the Newton steps of all variables as $\Delta x_0, ..., \Delta x_N \in \mathbb{R}^{n_x}$, $\Delta u_0, ..., \Delta u_{N-1} \in \mathbb{R}^{n_u}$ and $\Delta\lambda_0, ..., \Delta\lambda_N \in \mathbb{R}^{n_x}$. For example, $\Delta x_0 = x_0^{k+1} - x_0^k$ is the Newton step that will bring the state from x_0^k at iteration k to x_0^{k+1} at iteration $k+1$. These variables will naturally appear by taking a first order approximation of the perturbed KKT conditions. This is illustrated with the primal feasibility constraints. Start with $x_0 - \bar{x} = 0$:

$$x_0^{k+1} - \bar{x} \approx x_0^k - \bar{x} + (x_0^{k+1} - x_0^k) = x_0^k - \bar{x} + \Delta x_0 = 0 \tag{26}$$

For the dynamical constraint $x_i + f(x_i, u_i)\delta t - x_{i+1} = 0$, the linearization gives :

$$
\begin{aligned}
x_i^{k+1} + f(x_i^{k+1}, u_i^{k+1})\delta t - x_{i+1}^{k+1} \approx x_i + f(x_i, u_i)\delta t - x_{i+1} \quad &+ \\
\nabla_{x_i}(x_i + f(x_i, u_i)\delta t - x_{i+1})(x_i^{k+1} - x_i) \quad &+ \\
\nabla_u(x_i + f(x_i, u_i)\delta t - x_{i+1})(u_i^{k+1} - u_i) \quad &+ \\
\nabla_{x_{i+1}}(x_i + f(x_i, u_i)\delta t - x_{i+1})(x_{i+1}^{k+1} - x_{i+1}) = \quad &0
\end{aligned}
$$

This latter leads to :

$$
x_i^{k+1} + f(x_i^{k+1}, u_i^{k+1})\delta t - x_{i+1}^{k+1} \approx \bar{x}_i + A_i\Delta x_i + B_i\Delta u_i - \Delta x_{i+1} = 0, \qquad i = 0, ..., N-1 \tag{27}
$$

Where :

$$
\begin{aligned}
\bar{x}_i &= x_i + f(x_i, u_i)\delta t - x_{i+1} \\
A_i &= \mathbf{I} + \nabla_x f(x_i, u_i)\delta t \\
B_i &= \nabla_u f(x_i, u_i)\delta t
\end{aligned}
$$

For the stationnarity conditions, the method remains the same.

$$
\begin{aligned}
r_{x,i}^{k+1} \approx r_{x,i} + \nabla_{xx}\mathcal{H}(x_i, u_i, \lambda_{i+1}) \quad & \Delta x_i \\
+\nabla_{xu}\mathcal{H}(x_i, u_i, \lambda_{i+1}) \quad & \Delta u_i \\
+A_i \quad & \Delta\lambda_{i+1} \\
- \quad & \Delta\lambda_i \\
= \quad & 0
\end{aligned}
$$

$$
r_{x,N}^{k+1} \approx r_{x,N} + \nabla_{xx}V_f(x_N)\Delta x_N - \Delta\lambda_N = 0
$$

$$
\begin{aligned}
r_{u,i}^{k+1} \approx r_{u,i} + \nabla_{ux}\mathcal{H}(x_i, u_i, \lambda_{i+1}) \quad & \Delta x_i \\
+\nabla_{uu}\mathcal{H}(x_i, u_i, \lambda_{i+1}) \quad & \Delta u_i \\
+B_i^T \quad & \Delta\lambda_{i+1} \\
= \quad & 0
\end{aligned}
$$

To make it more compact let's define :

$$
Q_{xx,i} = \nabla_{xx}\mathcal{H}(x_i, u_i, \lambda_{i+1})
$$

$$
Q_{xx,N} := \nabla_{xx}V_f(x_N)
$$

$$
Q_{xu,i} := \nabla_{xu}\mathcal{H}(x_i, u_i, \lambda_{i+1})
$$

$$
Q_{uu,i} := \nabla_{uu}\mathcal{H}(x_i, u_i, \lambda_{i+1})
$$

Which leads to the following approximate of the primal feasibility and stationnarity conditions :

$$
r_{x,i}^{k+1} \approx Q_{xx,i}\Delta x_i + Q_{xu,i}\Delta u_i + A_i\Delta\lambda_{i+1} - \Delta\lambda_i + r_{x,i} = 0, \qquad i = 0, ..., N-1 \tag{28a}
$$

$$
r_{x,N}^{k+1} \approx Q_{xx,N}\Delta x_N - \Delta\lambda_N + r_{x,N} = 0 \tag{28b}
$$

$$
r_{u,i}^{k+1} \approx Q_{uu,i}\Delta u_i + Q_{xu,i}\Delta x_i + B_i^T\Delta\lambda_{i+1} + r_{u,i} = 0, \qquad i = 1, ..., N-1 \tag{29}
$$

3.3 Equivalent QP Formulation

As shown in section 2.2, equations 26, 27, 28 and 29 can be reformulated as an equality constrained QP. Indeed, if $\mathbf{\Delta\Lambda} = \{\Delta\lambda_i\}_{i=0}^{N}$ are seen as Lagrange multipliers of this QP, on can rewrite the problem as :

$$\begin{aligned} \min_{\mathbf{X},\mathbf{U}} \quad & \sum_{i=0}^{N-1} \frac{1}{2} \left\{ \begin{bmatrix} \Delta x_i \\ \Delta u_i \end{bmatrix}^T \begin{bmatrix} Q_{xx,i} & Q_{xu,i} \\ Q_{ux,i} & Q_{uu,i} \end{bmatrix} \begin{bmatrix} \Delta x_i \\ \Delta u_i \end{bmatrix} + \begin{bmatrix} r_{x,i} \\ r_{u,i} \end{bmatrix}^T \begin{bmatrix} \Delta x_i \\ \Delta u_i \end{bmatrix} \right\} + \Delta x_N^T Q_{xx,N} \Delta x_N + r_{x,N}^T \Delta x_N \\ \text{s.t.} \quad & x_0 - \bar{x} + \Delta x_0 = 0 \\ & \bar{x}_i + A_i \Delta x_i + B_i \Delta u_i - \Delta x_{i+1} = 0, \qquad i = 0, ..., N-1 \end{aligned} \tag{30}$$

This very specific structure can be exploited to solve it using a Riccati recursion as done in [6]. Nevertheless, this project aims to solve it using trust-region method to limit the size of the Newton step depending on the confidence that we have in our model.

3.4 Trust Region Method

To apply the trust-region method based on Riccati recursion presented in [5], we must first set the problem in the form of 19. The first step consists of eliminating $\Delta x_0, \Delta x_1, ..., \Delta x_N$ since they are not the real variable that have an action on the system. Let's first rewrite the objective function in dense form :

$$\begin{bmatrix} \Delta x_0 \\ \Delta x_1 \\ \vdots \\ \Delta x_N \end{bmatrix}^T \begin{bmatrix} Q_{xx,0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & Q_{xx,1} & \dots & \mathbf{0} \\ \vdots & & \ddots & \vdots \\ \mathbf{0} & \dots & \dots & Q_{xx,N} \end{bmatrix} \begin{bmatrix} \Delta x_0 \\ \Delta x_1 \\ \vdots \\ \Delta x_N \end{bmatrix} + \begin{bmatrix} \Delta u_0 \\ \Delta u_1 \\ \vdots \\ \Delta u_{N-1} \end{bmatrix}^T \begin{bmatrix} Q_{ux,0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & Q_{ux,1} & \dots & \mathbf{0} \\ \vdots & & \ddots & \vdots \\ \mathbf{0} & \dots & \dots & Q_{ux,N-1} \end{bmatrix} \begin{bmatrix} \Delta x_0 \\ \Delta x_1 \\ \vdots \\ \Delta x_{N-1} \end{bmatrix} +$$
$$\begin{bmatrix} \Delta x_0 \\ \Delta x_1 \\ \vdots \\ \Delta x_{N-1} \end{bmatrix}^T \begin{bmatrix} Q_{xu,0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & Q_{xu,1} & \dots & \mathbf{0} \\ \vdots & & \ddots & \vdots \\ \mathbf{0} & \dots & \dots & Q_{xu,N-1} \end{bmatrix} \begin{bmatrix} \Delta u_0 \\ \Delta u_1 \\ \vdots \\ \Delta u_{N-1} \end{bmatrix} + \begin{bmatrix} \Delta u_0 \\ \Delta u_1 \\ \vdots \\ \Delta u_{N-1} \end{bmatrix}^T \begin{bmatrix} Q_{uu,0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & Q_{uu,1} & \dots & \mathbf{0} \\ \vdots & & \ddots & \vdots \\ \mathbf{0} & \dots & \dots & Q_{uu,N-1} \end{bmatrix} \begin{bmatrix} \Delta u_0 \\ \Delta u_1 \\ \vdots \\ \Delta u_{N-1} \end{bmatrix} +$$
$$\begin{bmatrix} r_{x,0} \\ r_{x,1} \\ \vdots \\ r_{x,N} \end{bmatrix}^T \begin{bmatrix} \Delta x_0 \\ \Delta x_1 \\ \vdots \\ \Delta x_N \end{bmatrix} + \begin{bmatrix} r_{u,0} \\ r_{u,1} \\ \vdots \\ r_{u,N-1} \end{bmatrix}^T \begin{bmatrix} \Delta u_0 \\ \Delta u_1 \\ \vdots \\ \Delta u_{N-1} \end{bmatrix} \tag{31}$$

Which can be compactly written as :

$$\mathbf{\Delta x}^T \mathcal{Q}_{xx} \mathbf{\Delta x} + \mathbf{\Delta u}^T \mathcal{Q}_{ux} \mathbf{\Delta x_{N-1}} + \mathbf{\Delta x_{N-1}}^T \mathcal{Q}_{xu} \mathbf{\Delta u} + \mathbf{\Delta u}^T \mathcal{Q}_{uu} \mathbf{\Delta u} + R_x^T \mathbf{\Delta x} + R_u^T \mathbf{\Delta u}$$

By defining $\mathbf{\Delta x} = [\Delta x_0^T, \Delta x_1^T, ..., \Delta x_N^T]^T$ and $\mathbf{\Delta u} = [\Delta u_0^T, \Delta u_1^T, ..., \Delta u_{N-1}^T]^T$.

For the equality constraints, one can notice that if the whole sequence of Newton steps $\Delta u_0, \Delta u_1, ..., \Delta u_{N-1}$ is known, then the Newton steps $\Delta x_0, \Delta x_1, ..., \Delta x_N$ can be found recursively, starting from Δx_0 :

$$\begin{aligned} \Delta x_0 &= \bar{x} - x_0 \\ \Delta x_1 &= A_0(\bar{x} - x_0) + B_0 \Delta u_0 + \bar{x}_0 \\ \Delta x_2 &= A_1 A_0(\bar{x} - x_0) + A_1 B_0 \Delta u_0 + A_1 \bar{x}_0 + B_1 \Delta u_1 + \bar{x}_1 \\ \Delta x_3 &= A_2 A_1 A_0(\bar{x} - x_0) + A_2 A_1 B_0 \Delta u_0 + A_2 B_1 \Delta u_1 + B_2 \Delta u_2 + A_2 A_1 \bar{x}_0 + A_2 \bar{x}_1 + \bar{x}_2 \\ &\vdots \end{aligned}$$

Which, in matrix form gives :

$$\begin{bmatrix} \Delta x_0 \\ \Delta x_1 \\ \Delta x_2 \\ \vdots \\ \Delta x_{N-1} \\ \Delta x_N \end{bmatrix} = \begin{bmatrix} 0 & \dots & \dots & & \dots & 0 \\ B_0 & 0 & & & & \\ A_1B_0 & B_1 & 0 & & & \vdots \\ A_2A_1B_0 & A_2B_1 & B_2 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ (\prod_{i=1}^{N-2} A_i)B_0 & \ddots & \ddots & \ddots & \ddots & 0 \\ (\prod_{i=1}^{N-1} A_i)B_0 & (\prod_{i=2}^{N-1} A_i)B_1 & \dots & \dots & A_{N-1}B_{N-2} & B_{N-1} \end{bmatrix} \begin{bmatrix} \Delta u_0 \\ \Delta u_1 \\ \Delta u_2 \\ \vdots \\ \Delta u_{N-1} \end{bmatrix} + \begin{bmatrix} \bar{x} - x_0 \\ \bar{x}_0 + A_0(\bar{x} - x_0) \\ \bar{x}_1 + A_1\bar{x}_0 + A_1A_0(\bar{x} - x_0) \\ \bar{x}_2 + A_2\bar{x}_1 + A_2A_1\bar{x}_0 + A_2A_1A_0(\bar{x} - x_0) \\ \vdots \end{bmatrix} = \mathbf{M}\boldsymbol{\Delta}\mathbf{u} + \mathbf{h} \tag{32}$$

By replacing $\boldsymbol{\Delta}\mathbf{x}$ with this expression in 31, we get an objective function which depends only on $\boldsymbol{\Delta}\mathbf{u}$, same as in section 2.4.

$$\begin{aligned} & (\boldsymbol{\Delta}\mathbf{u}^T\mathbf{M}^T + \mathbf{h}^T)\mathcal{Q}_{xx}(\mathbf{M}\boldsymbol{\Delta}\mathbf{u} + \mathbf{h}) + \boldsymbol{\Delta}\mathbf{u}^T\mathcal{Q}_{ux}(\mathbf{M_{N-1}}\boldsymbol{\Delta}\mathbf{u} + \mathbf{h_{N-1}}) \\ + \quad & (\boldsymbol{\Delta}\mathbf{u}^T{\mathbf{M_{N-1}}}^T + {\mathbf{h_{N-1}}}^T)\mathcal{Q}_{xu}\boldsymbol{\Delta}\mathbf{u} + \boldsymbol{\Delta}\mathbf{u}^T\mathcal{Q}_{uu}\boldsymbol{\Delta}\mathbf{u} \\ + \quad & R_x^T(\mathbf{M}\boldsymbol{\Delta}\mathbf{u} + \mathbf{h}) + R_u^T\boldsymbol{\Delta}\mathbf{u} \\ = \quad & \boldsymbol{\Delta}\mathbf{u}^T \left[\mathbf{M}^T\mathcal{Q}_{xx}\mathbf{M} + \mathcal{Q}_{ux}\mathbf{M_{N-1}} + {\mathbf{M_{N-1}}}^T\mathcal{Q}_{xu} + \mathcal{Q}_{uu}\right] \boldsymbol{\Delta}\mathbf{u} \\ + \quad & \left[\mathbf{h}^T\mathcal{Q}_{xx}\mathbf{M} + \mathbf{h}^T\mathcal{Q}_{xx}^T\mathbf{M} + {\mathbf{h_{N-1}}}^T\mathcal{Q}_{ux}^T + {\mathbf{h_{N-1}}}^T\mathcal{Q}_{xu} + R_x^T\mathbf{M} + R_u^T\right] \boldsymbol{\Delta}\mathbf{u} \\ + \quad & \mathbf{h}^T\mathcal{Q}_{xx}\mathbf{h} + R_x^T\mathbf{h} \\ & = \boldsymbol{\Delta}\mathbf{u}^T\mathbf{H}\boldsymbol{\Delta}\mathbf{u} + \mathbf{g}^T\boldsymbol{\Delta}\mathbf{u} + \mathbf{c} \end{aligned} \tag{33}$$

The new Trust-Region sub-problem is now simply :

$$\begin{aligned} \min \quad & \boldsymbol{\Delta}\mathbf{u}^T\mathbf{H}\boldsymbol{\Delta}\mathbf{u} + \mathbf{g}^T\boldsymbol{\Delta}\mathbf{u} + \mathbf{c} \\ \text{s.t.} \quad & \|\boldsymbol{\Delta}\mathbf{u}\|_2 \leq \Delta \end{aligned} \tag{34}$$

Where Δ is indeed the trust region radius. The solution of this QP can be found for example using an interior point solver such as *Ipopt* [7]. Nevertheless, the purpose of this project is to use a combination of Riccati recursion and trust-region method to solve this sub-problem, as proposed in [5].
Not that all the developments presented in this section apply also for time-variant systems, as long as the switching-time are not part of the optimization. Therefore, the algorithm will also be tested for a switched-time system with fixed switching times.

4 Matlab Implementation

The whole code has been implemented in Matlab, using two classes. The first one (*riccati_TR*) handles the Riccati recursion with trust region constrained. It is accessible by the second one (*NOCP*), which handles the matrices construction, as well as the main solver's loop which update the trust radius and decide when to stop. A code example that demonstrate how to use these classes is provided in the appendix A.3. This latter also computes automatically the different gradients and Hessian for each sub-systems (in the case of switched systems).

4.1 NOCP Class

The objective of this class is to handle the main, outer optimization loop. For the inner loop which solve the trust region sub-problem, it can use either *Ipopt* (*CasADi*) or the Riccati recursion algorithm presented in section 2.5. Its constructor has the following prototype :

```
NOCP(primal, dual, dynamic, cost, params)
```

Where `primal` and `dual` are structures that contain respectively the initial values of the primal and dual variables, `dynamic` is a structure which contains function handle that describe the dynamic of the system, `cost` contains function handle of the stage cost and total cost and `params` is a structure containing parameters such as the tolerance to stop the optimization or the max number of iterations.
The class has also the following methods :

- `[] = build_SQP()`
- `[x, u, lambda] = updateSol()`
- `[x, u] = solve()`
- `[] = plotIter(i)`
- `[L] = evalLagrangian()`
- `[DELTA] = updateRadius()`
- `[LAMBDA] = update_dlambda()`

When the *solve* method is called, this class first constructs the approximation of the Lagrangian to have the same form as equation 30, using the initial primal and dual variables $\{x_i\}_{i=0}^{N}$, $\{u_i\}_{i=0}^{N-1}$ and $\{\lambda_i\}_{i=0}^{N}$ stored in the input `primal`. From there, it construct the matrix $\mathbf{H}$ by expressing $\mathbf{\Delta x} = \mathbf{\Delta x}(\mathbf{\Delta u})$, such as shown by equation 32. If the Riccati recursion is used, it then finds the smallest eigenvalue λ_1 of $\mathbf{H}$ and creates an object from the *riccati_TR* class which will manage to solve the trust-region sub-problem.
Then, it starts the main optimization loop where it iteratively (i) solves the trust region sub-problem by calling the *solve* method from the *riccati_TR* object (ii) update the values of the primal and dual variables (iii) evaluates the Lagrangian and the cost function at the current guess (iv) update the trust-region radius (v) check if the cost is stable enough to stop the optimization (vi) if no, constructs again the matrices at the current guess and go back to (i).
The evolution of the solution after each Newton step can be viewed by setting `params.showIter = true`. The whole code for this class is available in the appendix A.1.

4.2 riccati_TR Class

This class handles the Riccati recursion adapted for trust-region. Its objective is to compute the Newton steps $\mathbf{\Delta u}$ such that $\|\mathbf{\Delta u}\|_2 \leq \Delta$. Its constructor has the following prototype :

```
riccati_TR(N, LAMBDA, DELTA, A, B, Q, R, S, q, r, b)
```

Where `N` is the horizon, `LAMBDA` is the initial value of λ (see equation 23), `DELTA` is the trust radius (constant for one Newton step iteration), `A` and `B` contain the matrices $\{A_i\}_{i=0}^{N-1}$, $\{B_i\}_{i=0}^{N-1}$, `Q`, `R` and `S` contain the

matrices $\{Q_{xx,i}\}_{i=0}^{N}$, $\{Q_{uu,i}\}_{i=0}^{N-1}$ and $\{Q_{xu,i}\}_{i=0}^{N-1}$ and `q`, `r` and `b` contain the vectors $\{r_{x,i}\}_{i=0}^{N}$, $\{r_{u,i}\}_{i=0}^{N-1}$ and $\{\bar{x}_i\}_{i=0}^{N-1}$ (this naming convention matches the notation from 30 to 18).
This class has the following methods :

- `[] = backward()`
- `[dx, du] = forward()`
- `[LAMBDA] = updateLambda()`
- `[dx, du] = solve()`

When the `solve` method is called, it iteratively (i) runs the backward recursion (equation 20) (ii) runs the forward recursion (equation 21) (iii) check if $\|\mathbf{u}\|_2 < \Delta$, if yes stop, if no (iv) update the value of λ (equation 23) and go back to (i).
The whole code for this class is available in the appendix A.2.

4.3 Full Algorithm

This section present the full algorithm implemented using the `riccati_TR` and `NOCP` classes. Note that a state vector at stage i and at Newton step k is written as $\mathbf{x}_i^k$ and its j-th component is written as x_{ij}^k. The same conventions apply also for the inputs and dual variables and their trajectory at Newton step k are written $\mathbf{u}^k = [u_0^k, u_1^k, ..., u_{N-1}^k]$ and $\lambda^k = [\lambda_0^k, \lambda_1^k, ..., \lambda_{N-1}^k]$ respectively. Finally, the *Objective* parameter refers to the stage objective function to optimize and *Dynamic* refers to the expression $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$.

Algorithm 2 SQP using trust region and Riccati recursion

Require: Objective, Dynamic, $\mathbf{u}^0, \lambda^0, x_0, \Delta_{max}, \Delta_0, \epsilon > 0$

$k \leftarrow 0$ ▷ Initialization
$\Delta \leftarrow \Delta_0$

while $\|\nabla\mathcal{L}(\mathbf{u}^k, \mathbf{x}^k, \lambda^k)\| > \epsilon$ **do** ▷ Main optimization loop
  Build matrix $\mathbf{H}$, find its smallest eigenvalue λ_1 and set $\lambda > -\lambda_1$

  Run backward recursion (20) and forward propagation (21)
  while $\|\mathbf{\Delta u}^k\|_2 > \Delta$ **do** ▷ Riccati Recursion within trust radius
    Update λ using (23)
    Run backward recursion (20) and forward propagation (21)
  end while

  Compute $\mathbf{\Delta x}^k$ and $\mathbf{\Delta}\lambda^k$ using (32) and (28) ▷ Newton step
  Update current guess : $\mathbf{u}^k \leftarrow \mathbf{u}^{k-1} + \mathbf{\Delta u}^k \quad \mathbf{x}^k \leftarrow \mathbf{x}^{k-1} + \mathbf{\Delta x}^k \quad \lambda^k \leftarrow \lambda^{k-1} + \mathbf{\Delta}\lambda^k$

  Compute ρ^k using (17) ▷ Update trust region radius
  if $\rho^k < \frac{1}{4}$ **then**
    $\Delta \leftarrow \frac{1}{4}\Delta$
  else if $\rho^k > \frac{3}{4}$ **then**
    $\Delta \leftarrow \min(2\Delta, \Delta_{max})$
  end if

  $k \leftarrow k + 1$
end while

$\mathbf{u}^* \leftarrow \mathbf{u}^k \quad \mathbf{x}^* \leftarrow \mathbf{x}^k \quad \lambda^* \leftarrow \lambda^k$ ▷ Get optimal values

5 Test of the Algorithm

The algorithm presented in the previous chapter can be used for non-linear systems with any number of states and inputs. This chapter presents the results obtained for some example of non-linear systems, mainly inspired from those presented in [6].

5.1 1 State & 1 Input NL system

The first system studied is a 1-dimensional, single input non-linear system defined by :

$$\dot{x} = f(x, u) = x \cdot u + u^2$$

With the state $x_i \in \mathbb{R} \ \forall i = 0, ..., N$ and input $u_i \in \mathbb{R} \ \forall i = 0, ..., N-1$. The stage cost is the same as an LQR, with $Q \in \mathbb{R}_+$ and $R \in \mathbb{R}_+$.

$$l(x_i, u_i) = \frac{1}{2} x_i Q x_i + \frac{1}{2} u_i R u_i$$

And the terminal cost is, with $Q_N \in \mathbb{R}$:

$$V_N(x_N) = \frac{1}{2} x_N Q_N x_N$$

Note that, if nothing is specified, the weight are all taken equal to 1.

5.1.1 Computation of the Hamiltonian

A very important parameter to construct our trust region objective function is the Hamiltonian and its gradients with respect to the primal variables. With $\lambda_i \in \mathbb{R} \ \forall i = 0, ..., N$, the Hamiltonian is therefore :

$$\mathcal{H}(x_i, u_i, \lambda_{i+1}) = \frac{1}{2} x_i Q x_i + \frac{1}{2} u_i R u_i + \lambda_{i+1}(x \cdot u + u^2)\delta t$$

And its first and second order derivatives are :

$$\frac{\partial}{\partial x}\mathcal{H}(x_i, u_i, \lambda_{i+1}) = Q x_i + \lambda_{i+1} \cdot u_i \cdot \delta t$$

$$\frac{\partial}{\partial u}\mathcal{H}(x_i, u_i, \lambda_{i+1}) = R u_i + \lambda_{i+1} \cdot (x_i + 2u_i) \cdot \delta t$$

$$\frac{\partial^2}{\partial x^2}\mathcal{H}(x_i, u_i, \lambda_{i+1}) = Q$$

$$\frac{\partial^2}{\partial u^2}\mathcal{H}(x_i, u_i, \lambda_{i+1}) = R + 2 \cdot \lambda_{i+1} \cdot \delta t$$

$$\frac{\partial^2}{\partial x \partial u}\mathcal{H}(x_i, u_i, \lambda_{i+1}) = \lambda_{i+1} \cdot \delta t$$

$$\frac{\partial^2}{\partial u \partial x}\mathcal{H}(x_i, u_i, \lambda_{i+1}) = \lambda_{i+1} \cdot \delta t$$

5.1.2 Comparison With CasADi Results

The first test simply consisted of verifying that the solution from our Riccati recursion, trust-region method operates correctly. To do so, the solutions from our solver have been compared with those from *CasADi* (*Ipopt* solver). Note that this latter is only used to solve the trust region sub-problem (34) which provides the Newton steps $\Delta u_0, \Delta u_1, ..., \Delta u_{N-1}$.
The solvers have been compared with several initial states $x_0 \in [-5; 5]$, a maximal trust-region radius $\Delta_{max} = 10$ (which is also the initial value Δ_0) and an horizon $N = 50$. All the tests with this system led to the same optimal trajectories for the two solvers. Figures 2 and 3 show the results obtained using these two methods. In this case, our solver takes only 6 Newton steps while *CasADi* needs 9. Note also that the input trajectory remains sometime noisy with *CasADi*.

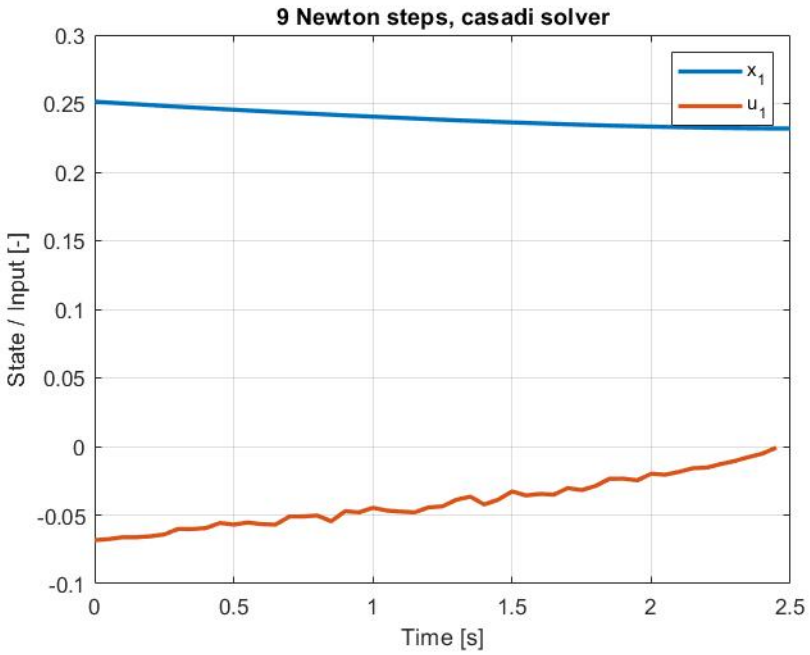

Figure 2: State and input trajectory using CasADi to solve the trust-region sub-problem.

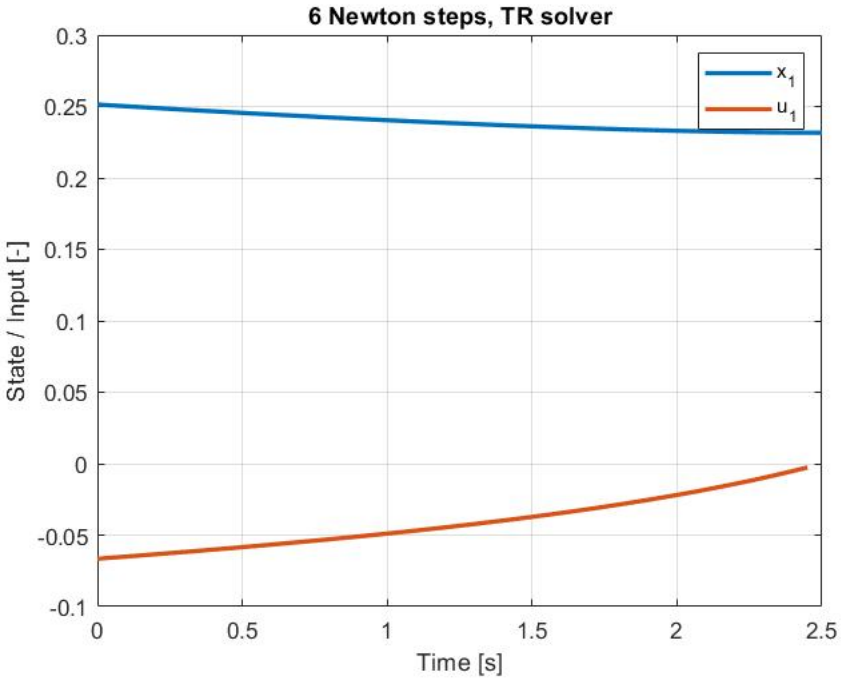

Figure 3: State and input trajectory using Riccati recursion to solve the trust-region sub-problem.

5.1.3 Effect of the Horizon on the Solving Duration

The duration to solve the problem has been measured for different initial values $x_0 \in [-2.5; 2.5]$, a constant maximum trust-region radius $\Delta_{max} = 10$ and a horizon from 5 to 100 time-steps. For each horizon, the problem has been solved 20 times to get an average solving time.

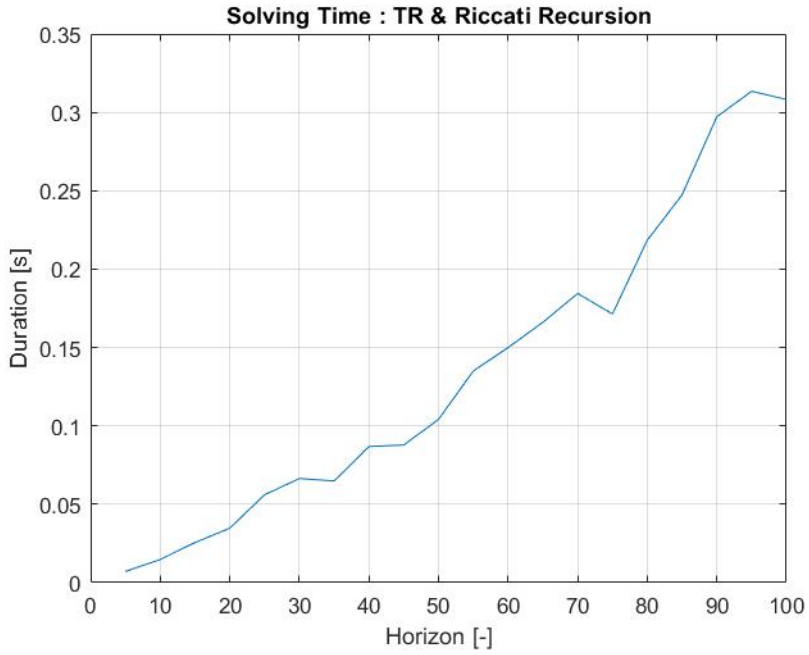

Figure 4: Evolution of the solving procedure using trust-region method combined with Riccati recursion. $x_i \in \mathbb{R}\ \forall i = 0, ..., N$ and $u_i \in \mathbb{R}\ \forall i = 0, ..., N-1$.

5.2 2 States & Single Input NL System

The second system studied is a multi-variable system with $u_i \in \mathbb{R}\ \forall i = 0, ..., N-1$ and $\mathbf{x}_i \in \mathbb{R}^2\ \forall i = 0, ..., N$ defined with :

$$\dot{x} = f(\mathbf{x}, u) = \begin{bmatrix} x_1 + u \cdot \sin(x_1) \\ -x_2 - u \cdot \cos(x_2) \end{bmatrix}$$

The stage cost is the same as an LQR, with $\mathbf{Q} \in \mathbb{S}^2_+$ and $R \in \mathbb{R}_+$.

$$l(\mathbf{x}_i, u_i) = \frac{1}{2}\mathbf{x}_i^T \mathbf{Q} \mathbf{x}_i + \frac{1}{2} u_i^T R u_i$$

And the terminal cost is, with $\mathbf{Q}_N \in \mathbb{R}^2$:

$$V_N(\mathbf{x}_N) = \frac{1}{2}\mathbf{x_N}^T \mathbf{Q}_N \mathbf{x_N}$$

Note that, if nothing is specified, the weight matrices are all taken as unitary matrices $\mathbf{I}$.

5.2.1 Computation of the Hamiltonian

With $\lambda_i \in \mathbb{R}^2\ \forall i = 0, ..., N$, the Hamiltonian is therefore :

$$\mathcal{H}(\mathbf{x}_i, u_i, \lambda_{i+1}) = \frac{1}{2}\mathbf{x}_i^T \mathbf{Q} \mathbf{x}_i + \frac{1}{2} u_i^T R u_i + \lambda_{i+1}^T f(\mathbf{x}_i, u_i)\delta t \qquad \forall i = 0, ..., N-1$$

The following gradients are then obtained :

$$\nabla_x \mathcal{H}(\mathbf{x}_i, u_i, \lambda_{i+1}) = \mathbf{Q}\mathbf{x}_i + \nabla_x f(\mathbf{x}_i, u_i)^T \lambda_{i+1} \delta t = \mathbf{Q}\mathbf{x}_i + \begin{bmatrix} 1 + u \cdot \cos(x_{i1}) & 0 \\ 0 & -1 + u \cdot \sin(x_{i2}) \end{bmatrix} \lambda_{i+1} \delta t$$

$$\nabla_u \mathcal{H}(\mathbf{x}_i, u_i, \lambda_{i+1}) = Ru_i + \nabla_u f(\mathbf{x}_i, u_i)^T \lambda_{i+1} \delta t = Ru_i + \begin{bmatrix} \sin(x_{i1}) \\ -\cos(x_{i2}) \end{bmatrix}^T \lambda_{i+1} \delta t$$

And thus :

$$\nabla_{xx} \mathcal{H}(\mathbf{x}_i, u_i, \lambda_{i+1}) = \mathbf{Q} + \begin{bmatrix} -u \cdot \sin(x_{i1}) & 0 \\ 0 & u \cdot \cos(x_{i2}) \end{bmatrix} \lambda_{i+1} \delta t$$

$$\nabla_{xu} \mathcal{H}(\mathbf{x}_i, u_i, \lambda_{i+1}) = \begin{bmatrix} \cos(x_{i1}) & 0 \\ 0 & \sin(x_{i2}) \end{bmatrix} \lambda_{i+1} \delta t$$

$$\nabla_{ux} \mathcal{H}(\mathbf{x}_i, u_i, \lambda_{i+1}) = \begin{bmatrix} \cos(x_{i1}) & 0 \\ 0 & \sin(x_{i2}) \end{bmatrix} \lambda_{i+1} \delta t$$

$$\nabla_{uu} \mathcal{H}(\mathbf{x}_i, u_i, \lambda_{i+1}) = R$$

5.2.2 Results

The code has been implemented with the same framework as in chapter 5.1.2, simply replacing the system. The first test consisted again of verifying that our trust-region Ricatti recursion worked correctly by comparing its results with those obtained with *CasADi*.
The solvers have been compared with several initial states $\mathbf{x}_0 \in [-5; 5] \times [-5; 5]$, a maximal trust-region radius $\Delta_{max} = 5$ (which is also the initial value Δ_0) and an horizon $N = 50$. All the tests with this system led to the same optimal trajectories for the two solvers. Figures 5 and 6 show the result obtained using these two methods. The two algorithms have converged within the same number of Newton steps, to almost the same solution. Nevertheless, for both solution the input trajectory looks very noisy.

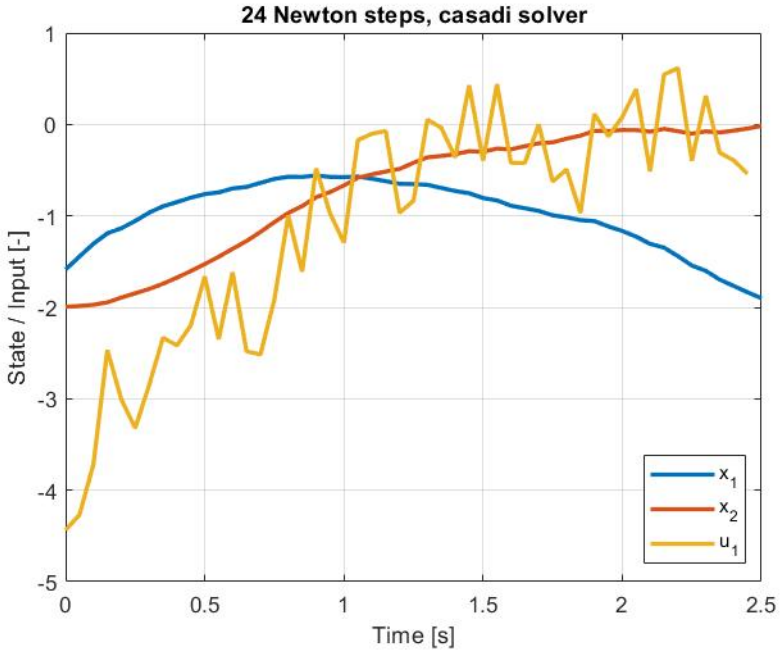

Figure 5: States and input trajectory using *CasADi* to solve the trust-region sub-problem.

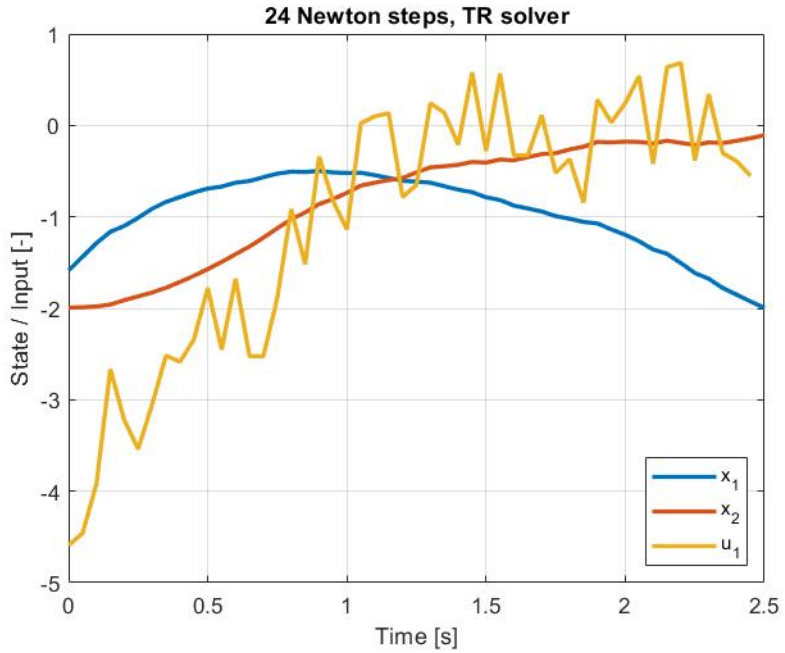

Figure 6: States and input trajectory using Riccati recursion to solve the trust-region sub-problem.

To improve the input signal the weight matrices have been modified to $\mathbf{Q} = \texttt{diag([10, 1])}$ and $R = 40$. The results are shown in figure 7 and 8.

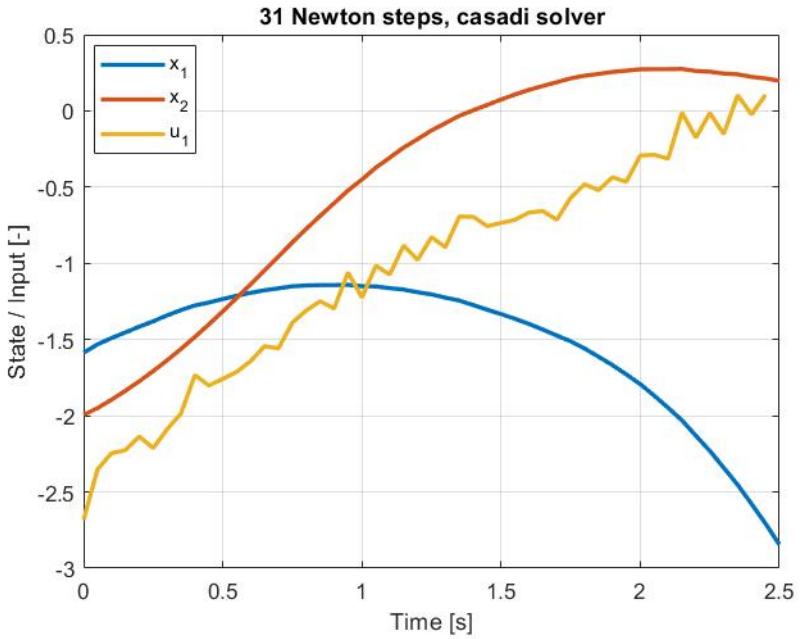

Figure 7: States and input trajectory using *CasADi* to solve the trust-region sub-problem with an increased weight on the input cost u and state x_1.

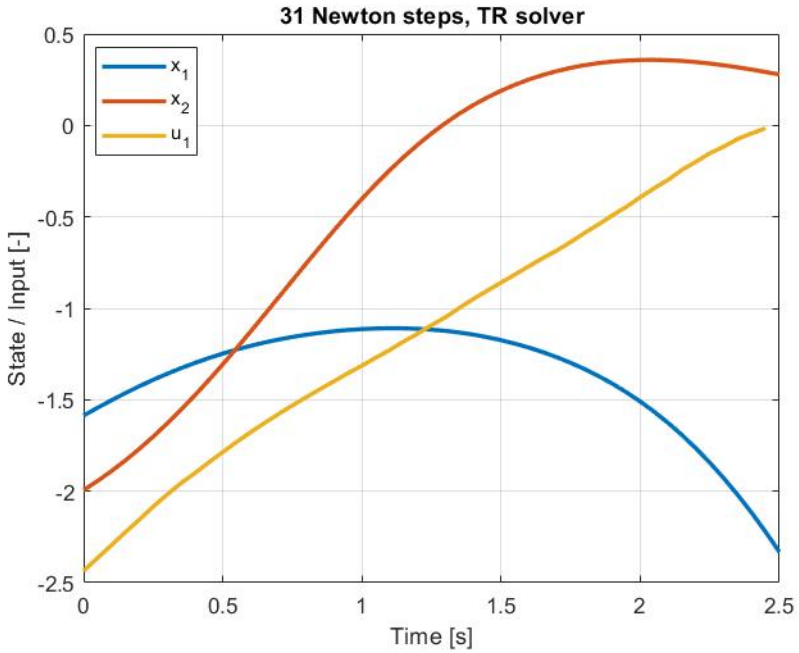

Figure 8: States and input trajectory using Riccati recursion to solve the trust-region sub-problem with an increased weight on the cost u and state x_1.

It clearly shows the effect of increasing the weight on the input with respect to the weight on the states. Not only the input signal is smoother, but also the optimization care less about state x_2 compared to before. The total cost (objective of (30)) after each Newton step, shown in figure 9 clearly shows that the two algorithms converge to the same optimal value.

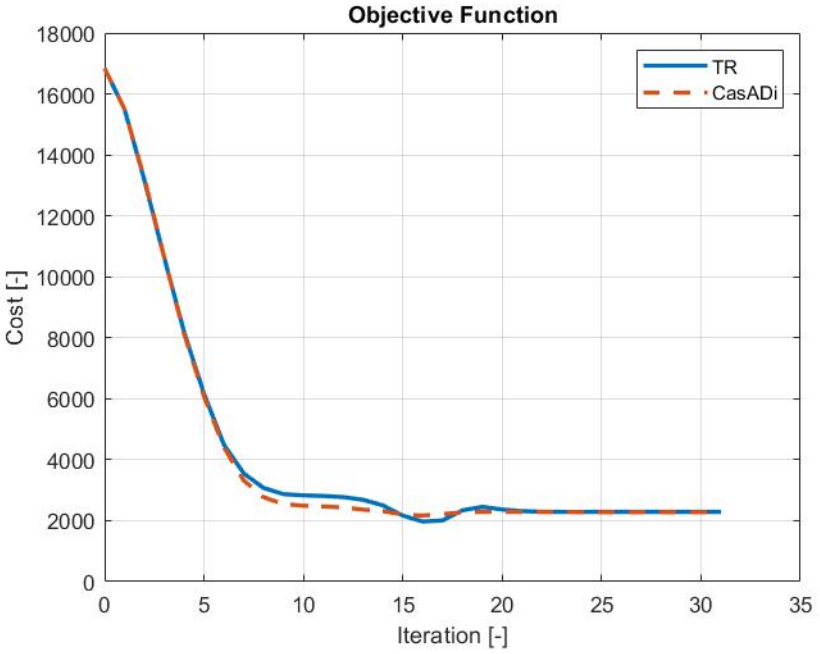

Figure 9: Evolution of the cost at each Newton step for trust-region Riccati recursion and *CasADi*. $\mathbf{x}_i \in \mathbb{R}^2 \ \forall i = 0, ..., N$ and $u_i \in \mathbb{R} \ \forall i = 0, ..., N-1$

5.2.3 Effect of the Horizon on the Solving Duration

Figure 10 shows the evolution of the solving duration depending on the horizon of the OCP. It has been obtained by taking the average duration over 20 optimizations for each horizon.

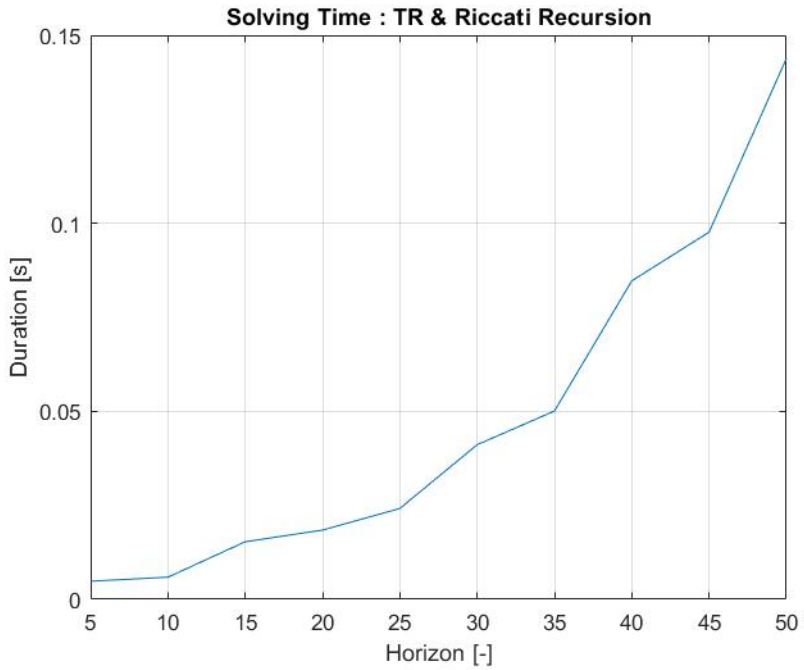

Figure 10: Evolution of the solving duration using trust-region method combined with Riccati recursion. $\mathbf{x}_i \in \mathbb{R}^2 \ \forall i = 0, ..., N$ and $u_i \in \mathbb{R} \ \forall i = 0, ..., N-1$.

5.3 2 States & 2 Inputs NL System

The third system studied is a multi-variable system with $\mathbf{u}_i \in \mathbb{R}^2 \ \forall i = 0, ..., N-1$ and $\mathbf{x}_i \in \mathbb{R}^2 \ \forall i = 0, ..., N$ defined with :

$$\dot{x} = f(\mathbf{x}, \mathbf{u}) = \begin{bmatrix} x_1 + u_1 \cdot \sin(x_1) \\ -x_2 - u_2 \cdot \cos(x_2) \end{bmatrix}$$

It is basically the same system as the previous one except it has 2 inputs, one for each of its states. The main objective to use such a system is only to confirm that the algorithm is working well for multiple-inputs systems. The stage cost is the same as an LQR, with $\mathbf{Q} \in \mathbb{S}^2_+$ and $\mathbf{R} \in \mathbb{S}^2_+$.

$$l(\mathbf{x}_i, \mathbf{u}_i) = \frac{1}{2}\mathbf{x}_i^T \mathbf{Q} \mathbf{x}_i + \frac{1}{2}\mathbf{u}_i^T \mathbf{R} \mathbf{u}_i$$

And the terminal cost is, with $\mathbf{Q}_N \in \mathbb{R}^2$:

$$V_N(\mathbf{x}_N) = \frac{1}{2}\mathbf{x_N}^T \mathbf{Q}_N \mathbf{x_N}$$

Note that, if nothing is specified, the weight matrices are all taken as unitary matrices $\mathbf{I}$.

5.3.1 Computation of the Hamiltonian

With $\lambda_i \in \mathbb{R}^2 \ \forall i = 0, ..., N$, the Hamiltonian is therefore :

$$\mathcal{H}(\mathbf{x}_i, \mathbf{u}_i, \lambda_{i+1}) = \frac{1}{2}\mathbf{x}_i^T \mathbf{Q} \mathbf{x}_i + \frac{1}{2}\mathbf{u}_i^T \mathbf{R} \mathbf{u}_i + \lambda_{i+1}^T f(\mathbf{x}_i, \mathbf{u}_i)\delta t \qquad \forall i = 0, ..., N-1$$

The following gradients are then obtained :

$$\nabla_x \mathcal{H}(\mathbf{x}_i, \mathbf{u}_i, \lambda_{i+1}) = \mathbf{Q}\mathbf{x}_i + \nabla_x f(\mathbf{x}_i, u_i)^T \lambda_{i+1} \delta t = \mathbf{Q}\mathbf{x}_i + \begin{bmatrix} 1 + u_{i1} \cdot \cos(x_{i1}) & 0 \\ 0 & -1 + u_{i2} \cdot \sin(x_{i2}) \end{bmatrix} \lambda_{i+1} \delta t$$

$$\nabla_u \mathcal{H}(\mathbf{x}_i, \mathbf{u}_i, \lambda_{i+1}) = Ru_i + \nabla_u f(\mathbf{x}_i, \mathbf{u}_i)^T \lambda_{i+1} \delta t = \mathbf{R}\mathbf{u}_i + \begin{bmatrix} \sin(x_{i1}) & 0 \\ 0 & -\cos(x_{i2}) \end{bmatrix} \lambda_{i+1} \delta t$$

And thus :

$$\nabla_{xx} \mathcal{H}(\mathbf{x}_i, \mathbf{u}_i, \lambda_{i+1}) = \mathbf{Q} + \begin{bmatrix} -u_{i1} \cdot \sin(x_{i1}) & 0 \\ 0 & u_{i2} \cdot \cos(x_{i2}) \end{bmatrix} diag(\lambda_{i+1}) \delta t$$

$$\nabla_{xu} \mathcal{H}(\mathbf{x}_i, \mathbf{u}_i, \lambda_{i+1}) = \begin{bmatrix} \cos(x_{i1}) & 0 \\ 0 & \sin(x_{i2}) \end{bmatrix} diag(\lambda_{i+1}) \delta t$$

$$\nabla_{ux} \mathcal{H}(\mathbf{x}_i, \mathbf{u}_i, \lambda_{i+1}) = \begin{bmatrix} \cos(x_{i1}) & 0 \\ 0 & \sin(x_{i2}) \end{bmatrix} diag(\lambda_{i+1}) \delta t$$

$$\nabla_{uu} \mathcal{H}(\mathbf{x}_i, \mathbf{u}_i, \lambda_{i+1}) = \mathbf{R}$$

5.3.2 Results

Figures 11 and 12 show the results for a maximal trust region $\Delta_{max} = 5$ and a horizon $N = 50$ time-steps. The two optimal solutions are almost identical which again tends to confirm that our algorithm operates correctly. Figure 13 also show that the two methods converge to the same optimal point $(\mathbf{x}^*, \mathbf{u}^*)$ and follow approximately the same path.

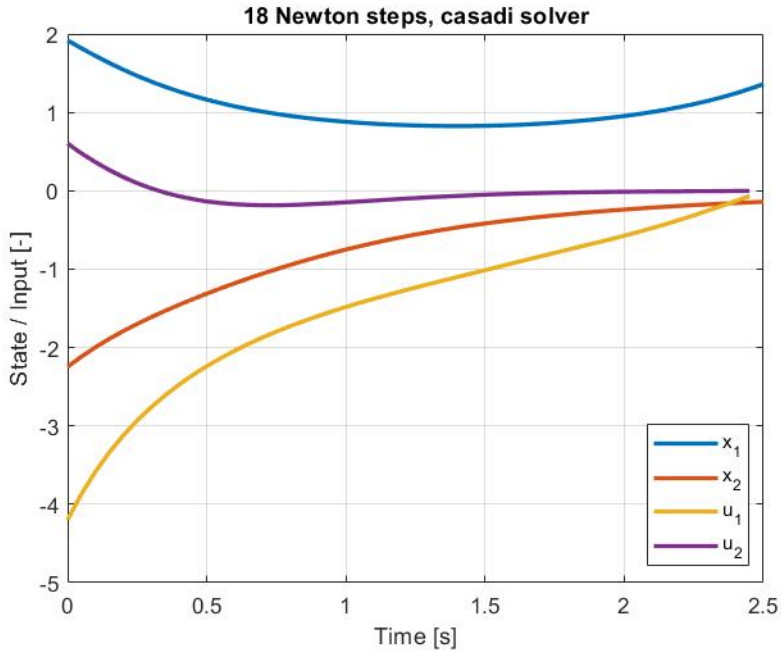

Figure 11: States and input trajectory using *CasADi* to solve the trust-region sub-problem with 2 states and 2 inputs.

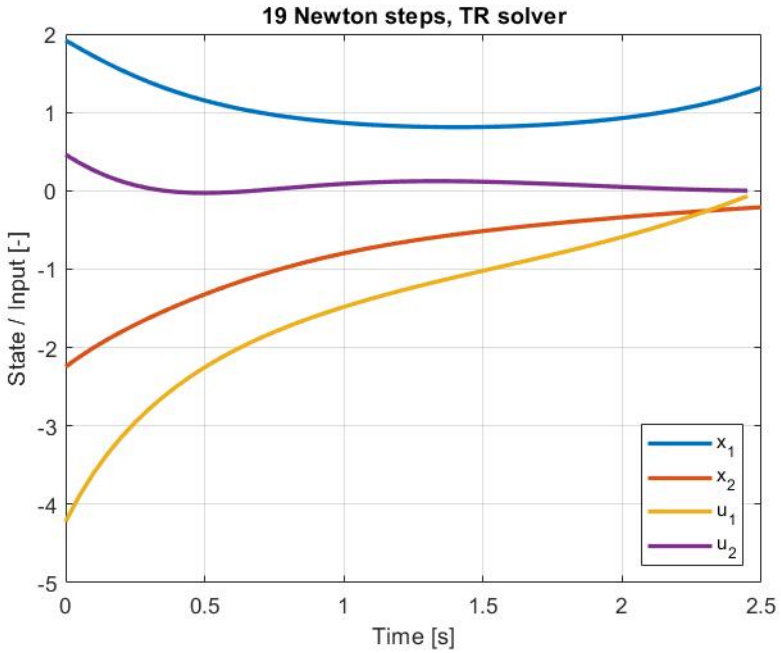

Figure 12: States and input trajectory using Riccati recursion to solve the trust-region sub-problem with 2 states and 2 inputs.

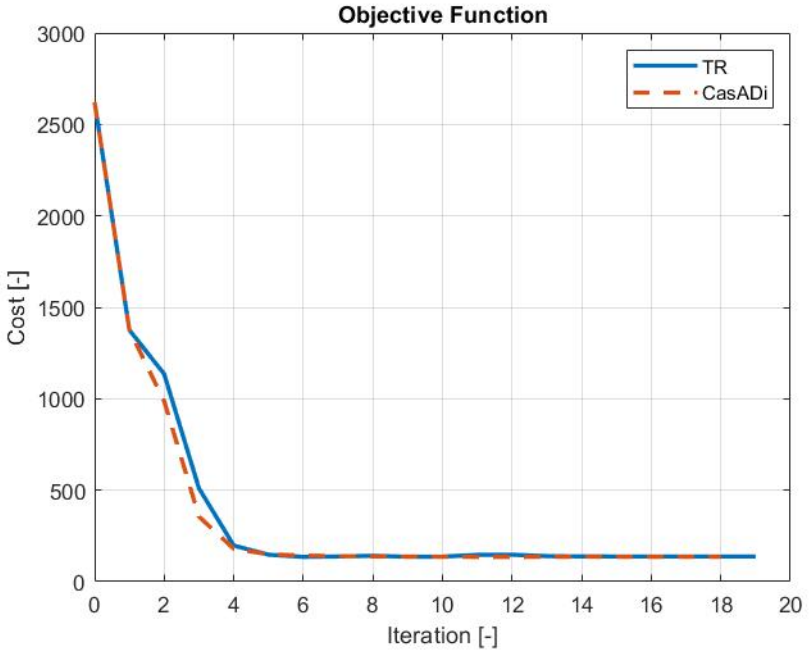

Figure 13: Evolution of the objective function at each Newton step using either trust-region Riccati recursion or *CasADi*. $\mathbf{x}_i \in \mathbb{R}^2 \ \forall i = 0, ..., N$ and $\mathbf{u}_i \in \mathbb{R}^2 \ \forall i = 0, ..., N-1$.

5.3.3 Effect of the Horizon on the Solving Duration

Figure 14 shows the evolution of the solving duration depending on the horizon of the OCP. It has been obtained by taking the average duration over 20 optimizations for each horizon.

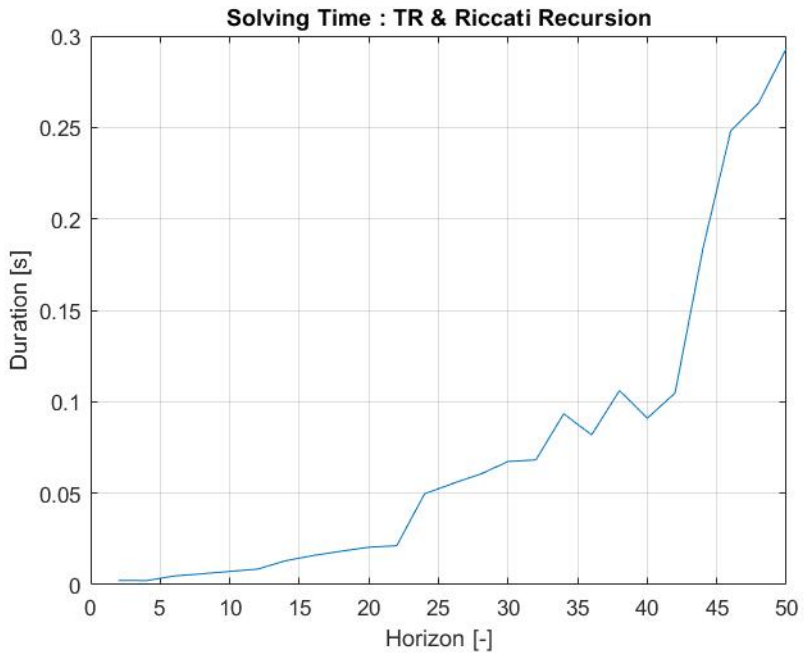

Figure 14: Evolution of the solving duration using trust-region method combined with Riccati recursion. $\mathbf{x}_i \in \mathbb{R}^2 \ \forall i = 0, ..., N$ and $\mathbf{u}_i \in \mathbb{R}^2 \ \forall i = 0, ..., N-1$.

5.4 Switched-Time NL MIMO System

The last system studied is a multi-variable switched-time system with $\mathbf{u}_i \in \mathbb{R}^2 \ \forall i = 0, ..., N-1$ and $\mathbf{x}_i \in \mathbb{R}^3 \ \forall i = 0, ..., N$ defined with :

$$\dot{x} = f_1(\mathbf{x}, \mathbf{u}) = \begin{bmatrix} x_1 + u_1 \cdot \sin(x_1) \\ -x_2 - u_2 \cdot \cos(x_2) \\ x_2 \cdot x_3 \end{bmatrix} \qquad \forall t \in [t_0; t_1]$$

$$\dot{x} = f_2(\mathbf{x}, \mathbf{u}) = \begin{bmatrix} x_2 + u_2 \cdot \sin(x_2) \\ -x_1 - u_1 \cdot \cos(x_1) \\ x_1 \cdot x_3 \end{bmatrix} \qquad \forall t \in [t_1; t_2]$$

$$\dot{x} = f_3(\mathbf{x}, \mathbf{u}) = \begin{bmatrix} -x_1 - u_1 \cdot \sin(x_1) \\ -x_2 + u_2 \cdot \cos(x_2) \\ x_1 \cdot x_2 \end{bmatrix} \qquad \forall t \in [t_2; t_f]$$

Note that the switching times $0 < t_1 < t_2 < N \cdot \delta t$ are manually fixed by the user. The stage cost is the same as an LQR, with $\mathbf{Q} \in \mathbb{S}^3_+$ and $\mathbf{R} \in \mathbb{S}^2_+$.

$$l(\mathbf{x}_i, \mathbf{u}_i) = \frac{1}{2}\mathbf{x}_i^T \mathbf{Q} \mathbf{x}_i + \frac{1}{2}\mathbf{u}_i^T \mathbf{R} \mathbf{u}_i$$

And the terminal cost is, with $\mathbf{Q}_N \in \mathbb{R}^3$:

$$V_N(\mathbf{x}_N) = \frac{1}{2}\mathbf{x_N}^T \mathbf{Q}_N \mathbf{x_N}$$

Note that, if nothing is specified, the weight matrices are all taken as unitary matrices $\mathbf{I}$.

5.4.1 Computation of the Hamiltonian

In the case of switched-time system, the Hamiltonian must be computed according to the current active system f_i at step i. With $\lambda_i \in \mathbb{R}^3 \ \forall i = 0, ..., N$, the Hamiltonian for each sub-system are therefore :

$$\mathcal{H}_1(\mathbf{x}_i, \mathbf{u}_i, \lambda_{i+1}) = \frac{1}{2}\mathbf{x}_i^T\mathbf{Q}\mathbf{x}_i + \frac{1}{2}\mathbf{u}_i^T\mathbf{R}\mathbf{u}_i + \lambda_{i+1}^T f_1(\mathbf{x}_i, \mathbf{u}_i)\delta t \qquad \forall i = 0, ..., N-1$$

$$\mathcal{H}_2(\mathbf{x}_i, \mathbf{u}_i, \lambda_{i+1}) = \frac{1}{2}\mathbf{x}_i^T\mathbf{Q}\mathbf{x}_i + \frac{1}{2}\mathbf{u}_i^T\mathbf{R}\mathbf{u}_i + \lambda_{i+1}^T f_2(\mathbf{x}_i, \mathbf{u}_i)\delta t \qquad \forall i = 0, ..., N-1$$

$$\mathcal{H}_3(\mathbf{x}_i, \mathbf{u}_i, \lambda_{i+1}) = \frac{1}{2}\mathbf{x}_i^T\mathbf{Q}\mathbf{x}_i + \frac{1}{2}\mathbf{u}_i^T\mathbf{R}\mathbf{u}_i + \lambda_{i+1}^T f_3(\mathbf{x}_i, \mathbf{u}_i)\delta t \qquad \forall i = 0, ..., N-1$$

The gradients and Hessian of the three Hamiltonian can be obtained in the same way as before or using Matlab as in our case, using the symbolic toolbox. The code in appendix A.3 compute automatically these vectors and matrices for any switched-time system with an arbitrary number of functions f_i.

5.4.2 Results

Figures 15 and 16 show the results for a maximal trust region $\Delta_{max} = 5$ and a horizon $N = 30$ time-steps. The two optimal solutions are almost identical which tends to confirm that our algorithm operates correctly, even for switched-time systems. Figure 17 also show that the two methods converge to the same optimal point $(\mathbf{x}^*, \mathbf{u}^*)$ and follow approximately the same path.

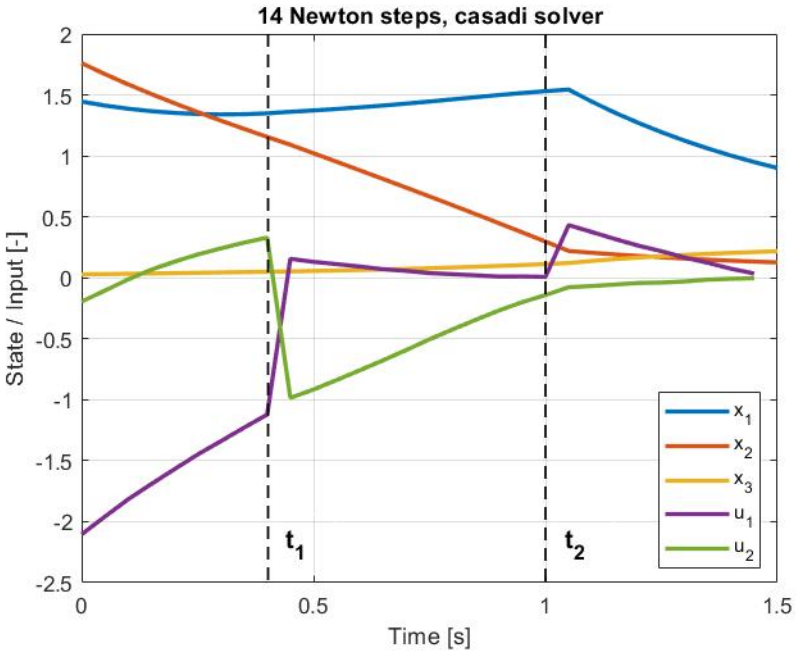

Figure 15: States and input trajectory using *CasADi* to solve the trust-region sub-problem with 3 states and 2 inputs. The switching times are fixed manually at time-steps $i_1 = 10$ and $i_2 = 22$.

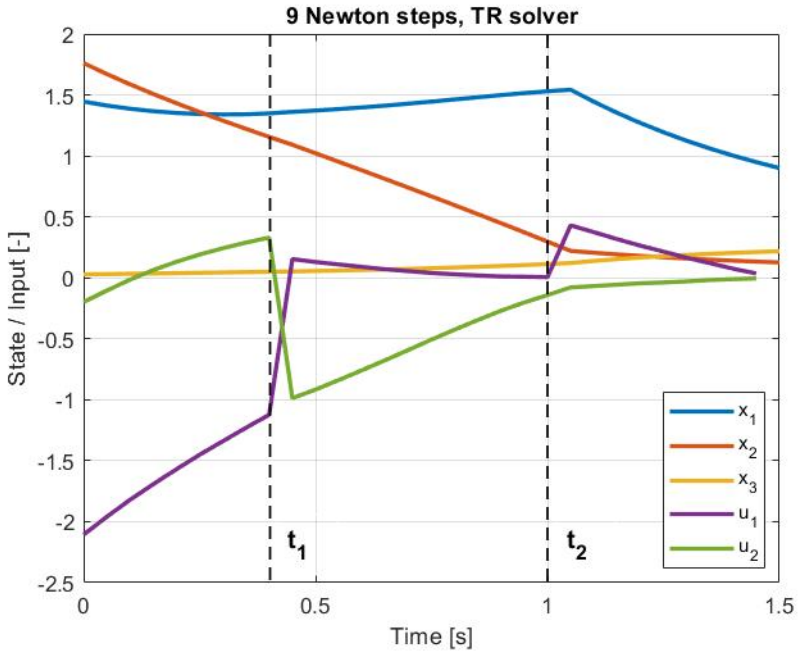

Figure 16: States and input trajectory using Riccati recursion to solve the trust-region sub-problem with 3 states and 2 inputs. The switching times are fixed manually at time-steps $i_1 = 10$ and $i_2 = 22$.

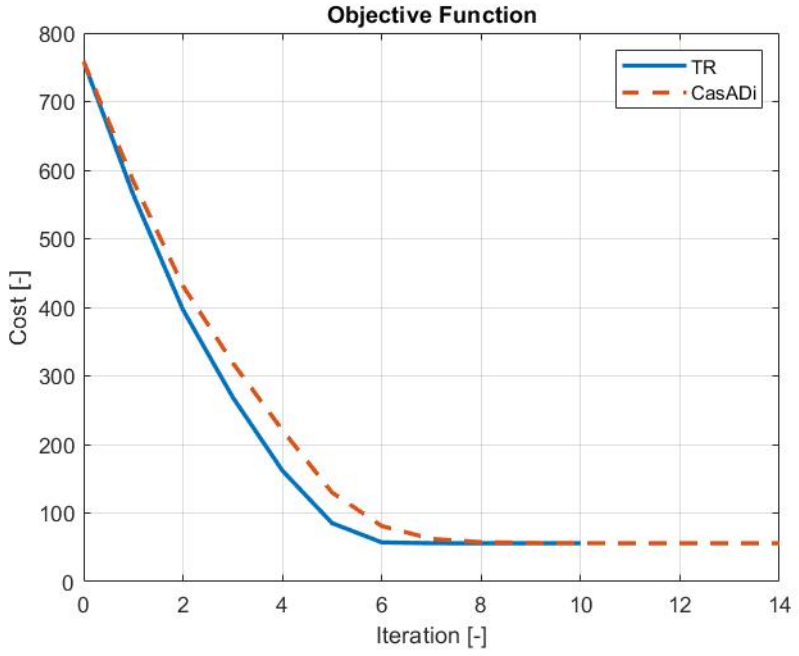

Figure 17: Evolution of the objective function at each Newton step using either trust-region Riccati recursion or *CasADi*. $\mathbf{x}_i \in \mathbb{R}^3 \ \forall i = 0, ..., N$ and $\mathbf{u}_i \in \mathbb{R}^2 \ \forall i = 0, ..., N-1$. The switching times are fixed manually at time-steps $i_1 = 10$ and $i_2 = 22$.

5.4.3 Effect of the Horizon on the Solving Duration

Figure 14 shows the evolution of the solving duration depending on the horizon of the OCP. It has been obtained by taking the average duration over 20 optimizations for each horizon.

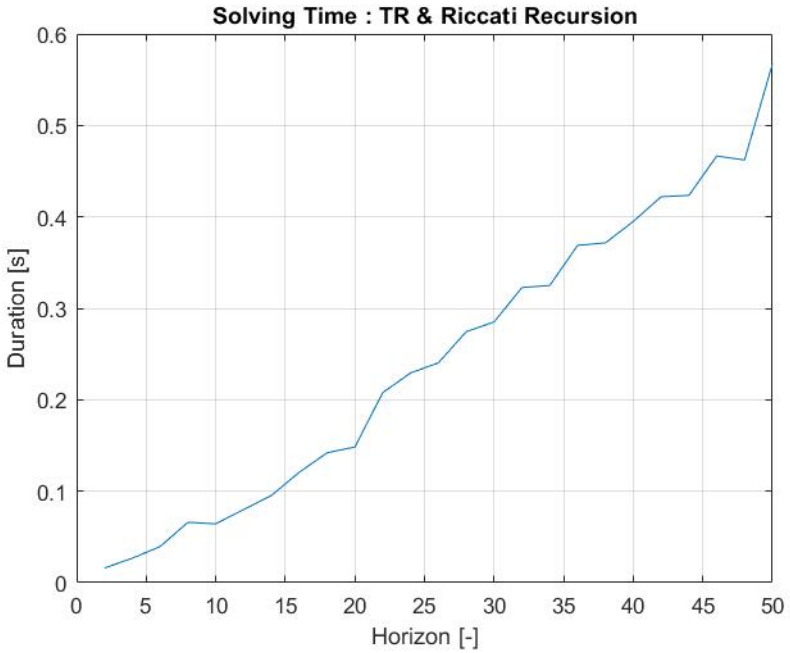

Figure 18: Evolution of the solving duration using trust-region method combined with Riccati recursion. $\mathbf{x}_i \in \mathbb{R}^3 \ \forall i = 0, ..., N$ and $\mathbf{u}_i \in \mathbb{R}^2 \ \forall i = 0, ..., N-1$. The switching times are fixed manually at time-steps $i_1 = 10$ and $i_2 = 22$.

5.5 Further Tests

The tests proposed in this section focused on non-linear MIMO systems with equality constraints. The next step could be to find a way to include inequality constraints to the framework. It would allow to conduct test with more complicated problem such as Switching Time Optimization (STO) which can be applied to legged robots [8][9][10]. Appendix B provides some of the theoretical developments to apply SQP to an equality and inequality constrained OCP with interior-point method. These are heavily based on [6] and might be useful to generalize the current algorithm to any type of constraint.

6 Conclusion

This project has presented and tested an algorithm that combines Riccati recursion with trust-region to solve equality constrained NOCP, including switching-time systems. The theoretical development took an important part of this project but led to an easy-to-implement formulation. The implementation has been done on *Matlab* and allowed to test the algorithm with several systems and compare its performances with *Ipopt* through the *CasADi* interface. The results have shown that our solver converges to the same solutions as *Ipopt* and with a comparable number of Newton steps.
The algorithm is available in appendix A with code examples. One of the code example handles the computation of the gradients and Hessians of each sub-systems (for switched-time system) and therefore makes its use very convenient. The only thing the user must provide is the expression of each sub-system, the dimensions n_x and n_u of the states and inputs, and the switching times.
A future improvement could be to generalize further the solver to be able to handle NOCP with inequality constraints. Such improvement would allow to apply the algorithm to solve STO problems, which have many applications such as legged robots.

References

[1] Lieven Vandenberghe Stephen Boyd. *Convex Optimization*. 2004.

[2] Stephen J. Wright Jorge Nocedal. *Numerical Optimization*. 2006.

[3] Shaohui Yang. *A Trust-Region Method for Multiple Shooting Optimal Control*. 2022.

[4] Moritz Diehl. *Lecture Notes on Numerical Optimization*. 2016.

[5] Shaohui Yang. *Trust region method in Riccati recursion*. 2022.

[6] Sotaro Katayama and Toshiyuki Ohtsuka. *Structure-Exploiting Newton-Type Method for Optimal Control of Switched Systems*. 2021. DOI: 10.48550/ARXIV.2112.07232. URL: https://arxiv.org/abs/2112.07232.

[7] *Interior Point Optimizer*. https://coin-or.github.io/Ipopt/. Accessed: 2023-14-01.

[8] Sotaro Katayama and Toshiyuki Ohtsuka. *Whole-body model predictive control with rigid contacts via online switching time optimization*. 2022. DOI: 10.48550/ARXIV.2203.00997. URL: https://arxiv.org/abs/2203.00997.

[9] Sotaro Katayama et al. "Nonlinear Model Predictive Control for Systems with State-Dependent Switches and State Jumps Using a Penalty Function Method". In: *2018 IEEE Conference on Control Technology and Applications (CCTA)*. 2018, pp. 312–317. DOI: 10.1109/CCTA.2018.8511448.

[10] Sotaro Katayama, Masahiro Doi, and Toshiyuki Ohtsuka. "A moving switching sequence approach for nonlinear model predictive control of switched systems with state-dependent switches and state jumps". In: *International Journal of Robust and Nonlinear Control* 30.2 (2020), pp. 719–740. DOI: https://doi.org/10.1002/rnc.4804. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rnc.4804. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/rnc.4804.

A Matlab Code

A.1 NOCP

```
classdef NOCP < handle
    properties
        % primal & dual variables
        primal, dual
        % dynamic, Hamiltonian & cost
        dynamic, Hamilt, cost
        % solver
        riccati_solver
        % Miscellaneous
        dt, N, nx, nu, DELTA, DELTA_MAX, maxIter, tol, costCurve, stopCriterion
        Q_xx, Q_uu, Q_xu, r_x, A
        % Figures
        fig_xu, fig_cost, fig_xu_axis, showIter
        % Lagrangian
        Lagrangian, LagrangianCurve
        % solver = choice of the solver, casadi_opti = optimizer
        solver, casadi_opti
        % Matrices
        H, g, c, M, h
    end

    methods

        %
        % %%%%%%%%%%%%%%%%%%%%%%% CONSTRUCTOR %%%%%%%%%%%%%%%%%%%%%%%
        %

        function obj = NOCP(primal, dual, dynamic, cost, params)
            obj.primal = primal;
            obj.dual = dual;
            obj.dynamic = dynamic;
            obj.cost = cost;
            obj.dt = params.dt;
            obj.DELTA = params.DELTA_0;
            obj.DELTA_MAX = params.DELTA_MAX;
            obj.showIter = params.showIter;
            obj.maxIter = params.maxIter;
            obj.tol = params.tol;
            obj.solver = params.solver;
            obj.N = size(primal.x, 2) - 1;
            obj.nx = size(primal.x, 1);
            obj.nu = size(primal.u, 1);

            obj.Hamilt = dynamic.Hamilt;
        end

        %
        % %%%%%%%%%%%%%%%% SETUP MATRICES & TR-RICCATI SOLVER %%%%%%%%%%%%%%%%
        %

        function [] = build_SQP(obj)
            % Initialization of the cells
            obj.Q_xx = cell(obj.N+1, 1);
            obj.Q_xu = cell(obj.N, 1);
            Q_ux = cell(obj.N, 1);
            obj.Q_uu = cell(obj.N, 1);

            obj.r_x = cell(obj.N+1, 1);
            r_u = cell(obj.N, 1);

            x_bar = cell(obj.N, 1);

            obj.A = cell(obj.N, 1);
```

```
64              B = cell(obj.N, 1);
65
66              %
67              % %%%%%%%%%% CONSTRUCT MATRICES QXX, QUU, QXU, QUX %%%%%%%%5
68              %
69
70              % iteration 0, ..., N-1
71              for i=1:1:obj.N
72                  % Get state, input, dual variables at current stage i & i+1
73                  xi = obj.primal.x(:, i);
74                  xip1 = obj.primal.x(:, i+1);
75                  ui = obj.primal.u(:, i);
76                  li = obj.dual.lambda(:, i); % lambda_i+1
77                  lip1 = obj.dual.lambda(:, i+1); % lambda_i+1
78
79                  % Compute matrices Q_xx, Q_xu, Q_ux, Q_uu, ... at stage i
80                  obj.Q_xx{i} = obj.Hamilt.dhdxx{i}(xi, ui, lip1);
81                  obj.Q_xu{i} = obj.Hamilt.dhdxu{i}(xi, ui, lip1);
82                  obj.Q_uu{i} = obj.Hamilt.dhduu{i}(xi, ui, lip1);
83                  Q_ux{i} = obj.Hamilt.dhdux{i}(xi, ui, lip1);
84
85                  obj.r_x{i} = obj.Hamilt.dhdx{i}(xi, ui, lip1) + lip1 - li;
86                  r_u{i} = obj.Hamilt.dhdu{i}(xi, ui, lip1);
87
88                  obj.A{i} = eye(obj.nx) + obj.dynamic.dfdx{i}(xi, ui) * obj.dt;
89                  B{i} = obj.dynamic.dfdu{i}(xi, ui) * obj.dt;
90
91                  x_bar{i} = xi + obj.dynamic.f{i}(xi, ui) * obj.dt - xip1;
92              end
93              % Iteration N
94              xN = obj.primal.x(:, i+1);
95              lN = obj.dual.lambda(:, i+1);
96              obj.Q_xx{i+1} = obj.cost.dVfdxx(xN);
97              obj.r_x{i+1} = obj.cost.dVfdx(xN) - lN;
98
99
100             %
101             % %%%%%%%%% Construct matrix \Delta x = M * \Delta u + h %%%%%%
102             %
103
104             obj.M = zeros((obj.N+1)*obj.nx, obj.N*obj.nu);
105             obj.h = zeros((obj.N+1) * obj.nx, 1);
106
107             % Go through the matrix M and fill it with appropriate matrices
108             for x = 1:1:obj.N
109                 for y = x+1:1:obj.N+1
110                     if(y == x+1)
111                         obj.M((y-1)*obj.nx+1:y*obj.nx, (x-1)*obj.nu + 1:x*obj.nu) = B{x};
112                     else
113                     obj.M((y-1)*obj.nx+1:y*obj.nx, (x-1)*obj.nu + 1:x*obj.nu) = obj.A{y-1} * obj.M
                            ((y-2)*obj.nx+1: (y-1)*obj.nx, (x-1)*obj.nu+1:x*obj.nu);
114                     end
115                 end
116             end
117
118             % Go through vector h and fill it with appropriate values
119             obj.h(1:obj.nx) = obj.primal.xbar - obj.primal.x(:, 1);
120             for y = 1:1:obj.N
121                 obj.h(y*obj.nx+1:y*obj.nx+obj.nx) = x_bar{y} + obj.A{y} * obj.h((y-1)*obj.nx+1:(y
                        -1)*obj.nx+obj.nx);
122             end
123
124             %
125             % %%%%%%%%%%%% Setup matrices QXX, QUX, QXU, QUU %%%%%%%%%%%%%%%%
126             %
127
128             QXX = zeros(obj.nx*(obj.N+1));
129             QUX = zeros(obj.nu*obj.N, obj.nx*obj.N);
```

```
            QXU = zeros(obj.nx*obj.N, obj.nu*obj.N);
            QUU = zeros(obj.nu*obj.N, obj.nu*obj.N);
            RX = zeros(obj.nx*(obj.N+1), 1);
            RU = zeros(obj.nu*obj.N, 1);

            for i = 0 : 1 : obj.N
                QXX(i*obj.nx + 1 : (i+1)*obj.nx, i*obj.nx + 1 : (i+1)*obj.nx) = obj.Q_xx{i+1};
                RX(i*obj.nx+1:i*obj.nx+obj.nx) = obj.r_x{i+1};

                if i < obj.N
                    QUX(i*obj.nu + 1 : (i+1)*obj.nu, i*obj.nx + 1 : (i+1)*obj.nx) = Q_ux{i+1};
                    QXU(i*obj.nx + 1 : (i+1)*obj.nx, i*obj.nu + 1 : (i+1)*obj.nu) = obj.Q_xu{i+1};
                    QUU(i*obj.nu + 1 : (i+1)*obj.nu, i*obj.nu + 1 : (i+1)*obj.nu) = obj.Q_uu{i+1};
                    RU(i*obj.nu + 1 : (i+1)*obj.nu) = r_u{i+1};
                end
            end

            %
            % %%%%%%%%%%%%%%%%%% Create matrix H %%%%%%%%%%%%%%%%%%
            %

            obj.H = obj.M' * QXX * obj.M + QUX * obj.M(1:end-obj.nx, :) + obj.M(1:end-obj.nx, :)'
                * QXU + QUU;

            switch obj.solver
                case "TR"

                    %
                    % %%%%%%%%%%%%%%%%%%% Check eigenvalues of H %%%%%%%%%%%%%%%%%%%
                    %

                    E = eig(obj.H);
                    LAMBDA1 = min(E);
                    % Initialize LAMBDA as bigger than the smallest
                    % eigevalue of H
                    LAMBDA = (-LAMBDA1 + 8e-1 * abs(LAMBDA1)) * eye(obj.nu);

                    % Create an object that handles Riccati recursion with
                    % trust region
                    obj.riccati_solver = riccati_TR(obj.N, LAMBDA, obj.DELTA, obj.A, B, obj.Q_xx,
                        obj.Q_uu, Q_ux, obj.r_x, r_u, x_bar);
                case "casadi"
                    % If casasadi is used, g and c must be provided since
                    % the objective depends on it
                    obj.g = (obj.h' * QXX * obj.M + obj.h' * QXX' * obj.M + obj.h(1:end-obj.nx, :)
                        ' * QUX' + obj.h(1:end-obj.nx, :)' * QXU + RX' * obj.M + RU')';
                    obj.c = obj.h' * QXX * obj.h + RX' * obj.h;
                otherwise
                    error("Invalid Solver name != ['TR', 'casadi']");
            end
        end

        %
        % %%%%%%%%%%%%%%%%%%% UPDATE PRIMAL AND DUAL %%%%%%%%%%%%%%%%%%%
        %

        function [x, u, lambda] = updateSol(obj)
            % du and dx are directly updated
            obj.primal.u = obj.primal.u + obj.primal.du;
            obj.primal.x = obj.primal.x + obj.primal.dx;
            % Update dlambda
            obj.update_dlambda();
            obj.dual.lambda = obj.dual.lambda + obj.dual.dlambda;
            x = obj.primal.x;
            u = obj.primal.u;
            lambda = obj.dual.lambda;
        end

```

```
195         %
196         % %%%%%%%%%%%%%%%%%%%%%%%% SOLVE THE NOCP %%%%%%%%%%%%%%%%%%%%%%%%%
197         %
198
199         function [x, u] = solve(obj)
200             if obj.showIter
201                 obj.fig_xu = figure;
202             end
203
204             obj.costCurve = [];
205             obj.LagrangianCurve = [];
206
207             % Initialize with first cost at initialization
208             obj.costCurve = obj.cost.COST(obj.primal.x', obj.primal.u');
209             obj.evalLagrangian(); % update lagrangian value with initial guess
210             obj.LagrangianCurve = obj.Lagrangian;
211             obj.stopCriterion = false;
212
213             i = 1;
214             while(i <= obj.maxIter && ~obj.stopCriterion)
215                 obj.build_SQP();
216                 switch obj.solver
217                     case "TR"
218                         [obj.primal.dx, obj.primal.du] = obj.riccati_solver.solve();
219                     case "casadi"
220                         obj.casadi_opti = casadi.Opti();
221                         p = struct("expand",true, "verbose", 0);
222                         s = struct("max_iter",obj.maxIter);
223                         obj.casadi_opti.solver('ipopt', p, s);
224                         obj.primal.du_casadi = obj.casadi_opti.variable(obj.nu * obj.N, 1);
225                         objective = obj.primal.du_casadi' * obj.H * obj.primal.du_casadi + obj.g'
                                * obj.primal.du_casadi + obj.c;
226                         obj.casadi_opti.subject_to( obj.primal.du_casadi' * obj.primal.du_casadi
                                <= obj.DELTA^2 );
227                         obj.casadi_opti.minimize(objective);
228                         sol = obj.casadi_opti.solve();
229
230                         % Get du from casADi
231                         du = sol.value(obj.primal.du_casadi);
232                         % Get dx
233                         dx = obj.M * du + obj.h;
234                         % Reshape dx to be consistent with du
235                         for k = 0:1:obj.N-1
236                             obj.primal.dx(:, k+1) = dx(k*obj.nx+1:(k+1)*obj.nx);
237                             obj.primal.du(:, k+1) = du(k*obj.nu+1:(k+1)*obj.nu);
238                         end
239                         obj.primal.dx(:, obj.N+1) = dx(obj.N*obj.nx+1:(obj.N+1)*obj.nx);
240                     otherwise
241                         error("Invalid Solver name != ['TR', 'casadi']");
242                 end
243
244                 obj.updateSol(); % Update x = x + dx, u = u + du, lambda = lambda + dlambda
245                 obj.evalLagrangian(); % Evaluate Langrangian at current guess
246                 obj.LagrangianCurve = [obj.LagrangianCurve obj.Lagrangian];
247                 obj.costCurve = [obj.costCurve obj.cost.COST(obj.primal.x', obj.primal.u')]; %
                        Update cost
248                 obj.updateRadius(); % Update the trust region radius (DELTA)
249
250                 % Check if the cost decrease is below the tolerance
251                 obj.stopCriterion = abs((obj.costCurve(end) - obj.costCurve(end-1))) < obj.tol;
252
253                 if obj.showIter % Plot current solution
254                     plotIter(obj, i);
255                 end
256
257                 i = i + 1;
258             end
259
```

```
260             % Returns the optimal solution
261             x = obj.primal.x;
262             u = obj.primal.u;
263         end
264 
265         %
266         % %%%%%%%%%%%%%% PLOT EACH NEWTON'S ITERATION RESULT %%%%%%%%%%%%%%
267         %
268 
269         function [] = plotIter(obj, i)
270             % Plot results
271             figure(obj.fig_xu);
272             legend_str = [];
273 
274             % Plot the states trajectory
275             for k = 1:1:obj.nx
276                 plot(obj.primal.x(k, :));
277                 legend_str = [legend_str strcat("x_", num2str(k))];
278                 hold on;
279             end
280 
281             % Plot the inputs trajectory
282             for k = 1:1:obj.nu
283                 plot(obj.primal.u(k, :));
284                 legend_str = [legend_str strcat("u_", num2str(k))];
285                 hold on;
286             end
287 
288             title(strcat("Newton Step ", num2str(i)));
289             legend(legend_str);
290             grid on;
291             hold off;
292 
293             if i == 1
294                 obj.fig_xu_axis = axis;
295             end
296 
297             axis(obj.fig_xu_axis);
298             if i == obj.maxIter || obj.stopCriterion
299 
300                 figure(obj.fig_xu);
301                 for k = 1:1:obj.nx
302                     plot(obj.primal.time, obj.primal.x(k, :), 'LineWidth', 2);
303                     hold on;
304                 end
305                 for k = 1:1:obj.nu
306                     plot(obj.primal.time(1:end-1), obj.primal.u(k, :), 'LineWidth', 2);
307                     hold on;
308                 end
309 
310                 title(strcat(num2str(i), " Newton steps, ", obj.solver, " solver"));
311                 xlabel("Time [s]");
312                 ylabel("State / Input [-]");
313                 legend(legend_str);
314                 grid on;
315                 hold off;
316 
317                 obj.fig_cost = figure;
318                 plot([0:1:i], obj.costCurve, 'LineWidth', 2);
319                 xlabel("Iteration [-]");
320                 ylabel("Cost [-]");
321                 title(strcat("Objective function, ", obj.solver, " solver"));
322                 grid on;
323             end
324         end
325 
326         %
327         % %%%%%%%%%%%%%%%%%%%%%% EVALUATE THE CURRENT LAGRANGIAN %%%%%%%%%%%%%%%%%%%%%%
```

```
        %

        function [L] = evalLagrangian(obj)
            obj.Lagrangian = -obj.dual.lambda(:, 1)' * (obj.primal.x(:, 1) - obj.primal.xbar);

            for i = 1 : 1: obj.N
                xi = obj.primal.x(:, i); % x_i
                xip1 = obj.primal.x(:, i+1); % x_i+1
                ui = obj.primal.u(:, i); % ui
                lip1 = obj.dual.lambda(:, i+1); % lambda_i+1

                % Compute the sum
                obj.Lagrangian = obj.Lagrangian + ...
                        obj.cost.l(xi, ui) + ...
                        lip1' * (xi + obj.dynamic.f{i}(xi, ui) * obj.dt - xip1);
            end
            % Add sum at iteration N
            xN = obj.primal.x(:, i+1);
            obj.Lagrangian = obj.Lagrangian + obj.cost.Vf(xN);
            L = obj.Lagrangian;
        end

        %
        % %%%%%%%%%%%%%%%%%%%%%% UPDATE THE CURRENT TRUST RADIUS %%%%%%%%%%%%%%%%%%%%%%
        %

        function [DELTA] = updateRadius(obj)
            % Compute a measure of confidence that we have in our
            % approximation
            rho = (obj.LagrangianCurve(end-1) - obj.LagrangianCurve(end))/(obj.costCurve(end-1) -
                obj.costCurve(end));

            % Update the trust radius depending on the confidence we have
            if rho < 0.25
                obj.DELTA = 0.25 * obj.DELTA;
            elseif rho > 0.75
                obj.DELTA = min(2 * obj.DELTA, obj.DELTA_MAX);
            end
            DELTA = obj.DELTA;
        end

        %
        % %%%%%%%%%%%%%%%%%%%%%% UPDATE THE NEWTON STEP OF LAMBDA %%%%%%%%%%%%%%%%%%%%%%
        %

        function [dlambda] = update_dlambda(obj)
            obj.dual.dlambda = zeros(obj.nx, obj.N+1);
            obj.dual.dlambda(:, obj.N+1) = obj.r_x{obj.N+1} + obj.Q_xx{obj.N+1} * obj.primal.dx(:,
                 obj.N+1);

            % Iteratively find the Newton step dlambda to find lambda +=
            % dlambda
            for i = obj.N : -1 : 1
                obj.dual.dlambda(:, i) = obj.r_x{i} + obj.A{i} * obj.dual.dlambda(:, i+1) + obj.
                    Q_xx{i} * obj.primal.dx(:, i) + obj.Q_xu{i} * obj.primal.du(:, i);
            end
            dlambda = obj.dual.dlambda;
        end
    end
end
```

A.2 riccati_TR

```
classdef riccati_TR < handle
    properties
        % Objective variables
        Q, R, S, q, r, b
        % Linearized dynamic
        A, B
        % Inner variables
        R_bar, P, Lambda, L, l, p
        % Horizon
        N
        % u-dimensions, x-dimensions
        nx, nu
        % Newton steps
        du, dx
        % Lambda and TR radius
        LAMBDA, DELTA
    end
    methods

        %
        % %%%%%%%%%%%%%%%%%% CONSTRUCTOR %%%%%%%%%%%%%%%%%%%
        %

        function obj = riccati_TR(N, LAMBDA, DELTA, A, B, Q, R, S, q, r, b)
            obj.N = N;
            obj.LAMBDA = LAMBDA;
            obj.DELTA = DELTA;
            obj.A = A;
            obj.B = B;
            obj.nx = size(A{1}, 2);
            obj.nu = size(B{1}, 2);
            obj.Q = Q;
            obj.R = R;
            obj.S = S;
            obj.q = q;
            obj.r = r;
            obj.b = b;
            obj.R_bar = cell(N, 1);
            obj.P = cell(N+1, 1);
            obj.Lambda = cell(N, 1);
            obj.L = cell(N, 1);
            obj.l = cell(N, 1);
            obj.p = cell(N+1, 1);
        end

        %
        % %%%%%%%%%%%%%%%%%% BACKWARD RICCATI RECURSION %%%%%%%%%%%%%%%%%%%
        %

        function [] = backward(obj)
            obj.P{obj.N+1} = obj.Q{obj.N+1};
            obj.p{obj.N+1} = obj.q{obj.N+1};

            % Backward recursion
            for k = obj.N : -1 : 1
                obj.R_bar{k} = obj.R{k} + obj.LAMBDA + obj.B{k}' * obj.P{k+1} * obj.B{k};
                obj.Lambda{k} = chol(obj.R_bar{k});
                obj.L{k} = obj.Lambda{k}' \ (obj.S{k} + obj.B{k}' * obj.P{k+1} * obj.A{k});
                obj.P{k} = obj.Q{k} + obj.A{k}' * obj.P{k+1} * obj.A{k} - obj.L{k}' * obj.L{k};
                obj.l{k} = obj.Lambda{k}' \ (obj.r{k} + obj.B{k}' * (obj.P{k+1} * obj.b{k} + obj.p
                    {k+1}));
                obj.p{k} = obj.q{k} + obj.A{k}' * (obj.P{k+1} * obj.b{k} + obj.p{k+1}) - obj.L{k}'
                     * obj.l{k};
            end
        end

```

```
        %
        % %%%%%%%%%%%%%%%%%% FORWARD RICCATI RECURSION %%%%%%%%%%%%%%%%%%%%
        %

        function [dx, du] = forward(obj)
            du = zeros(obj.nu, obj.N);
            dx = zeros(obj.nx, obj.N+1);

            % Forward propagation
            for k = 1 : 1 : obj.N
                du(:, k) = - obj.Lambda{k} \ (obj.L{k} * dx(:, k) + obj.l{k});
                dx(:, k+1) = obj.A{k} * dx(:, k) + obj.B{k} * du(:, k) + obj.b{k};
            end

            obj.du = du;
            obj.dx = dx;
        end

        %
        % %%%%%%%%%%%%%%%%%%% Q-UPDATE & LAMBDA UPDATE %%%%%%%%%%%%%%%%%%%%
        %

        function [LAMBDA] = updateLambda(obj)
            % Initialize beta and q
            beta = zeros(obj.nx, 1);
            q_norm_squared = 0;
            q_temp = cell(obj.N, 1);

            % q-update -> Used to update LAMBDA
            for k = obj.N : -1 : 1
                alpha = obj.du(:, k) + obj.B{k}' * beta;
                q_temp{k} = obj.Lambda{k}' \ alpha;
                beta = obj.A{k}' * beta - obj.L{k}' * (obj.Lambda{k}' \ alpha);
                q_norm_squared = q_norm_squared + q_temp{k}' * q_temp{k};
            end

            % LAMBDA-update
            obj.LAMBDA = obj.LAMBDA + eye(obj.nu) .* norm(obj.du)^2 * (1 * norm(obj.du) - obj.
                DELTA) / (q_norm_squared * obj.DELTA);
            LAMBDA = obj.LAMBDA;
        end

        %
        % %%%%%%%%%%%%%%%%%%%%%%%% SOLVE TR-RICCATI %%%%%%%%%%%%%%%%%%%%%%%%
        %

        function [dx, du] = solve(obj)
            obj.backward();
            obj.forward();

            % While norm > trust radius -> update LAMBDA & solve
            while norm(obj.du, 2) > obj.DELTA
                obj.updateLambda();
                obj.backward();
                obj.forward();
            end
            % At this point Newton step du is inside the trust region
            du = obj.du;
            dx = obj.dx;
        end
    end
end
```

A.3 Test Code

```
% ************************ TestAlgorithm_riccati_TR.m ************************
%
% Stephen Monnnet
% Laboratoire d'Automatique, 2023
%
% This code allows to test our trust-region Riccati recursion solver to any
% non-linear system. The user must provide the expression \Dot{x}= f(x, u)
% for each sub-system as well as the switching-times.
%
% Two tests are possible :
%       - Comparison with casADi
%         -> Uncomment lines 178-197
%       - Measure evolution of the solving duration w.r.t. the horizon N
%         -> Uncomment lines 203-247

%% Initialisation
close all;
clear;
clc;

% DON'T FORGET TO CORRECT THIS PATH ACCORDING TO YOUR INSTALLATION
addpath('C:/casadi-matlabR2016a-v3.5.5');
import casadi.*;

%% Miscellaneous variables
N = 50; % Horizon
dt = 0.05; % Sampling time
t0 = 0; % Initial time
tf = t0 + N * dt; % Final time

DELTA_0 = 5; % Initial trust radius
maxIter = 100; % Max number of Newton steps
showIter = true; % If true, plot intermediate solution of x, u
solver = 'TR'; % 'casadi' to use casADi to solve trust region sub-problem

%% Declare System
% Dimensions : Change it according to your system
nx = 3; nu = 2;

% Symbolic variables
x = sym('x', [nx, 1], 'real');
u = sym('u', [nu, 1], 'real');
lambda = sym('lambda', [nx, 1], 'real');

% System dynamic : Change nx and nu according to your system
f1 = [x(1) + u(1) * sin(x(1)); -x(2) - u(2) * cos(x(2)); x(2) * x(3)];
f2 = [x(2) + u(2) * sin(x(2)); -x(1) - u(1) * cos(x(1)); x(1) * x(3)];
f3 = [-x(1) - u(1) * sin(x(1)); -x(2) + u(2) * cos(x(2)); x(1) * x(2)]; % nx = 2, nu = 2

% Switching Times -> for n systems, n-1 switching times s.t. 0 < t_i < N
switchTime = [10, 22];

% Construct cell array of dynamical models
f = {f1, f2, f3};
Nsys = length(f);

% Cost function
Q = eye(nx);
R = eye(nu);
QN = eye(nx);
l = 0.5 * x' * Q * x + 0.5 * u' * R * u;
Vf = 0.5 * x' * QN * x;

% Hamiltonian
h = cell(Nsys, 1);

```

```
%% %%%%%%%%%%%%%%%%%%%%%%%% BUILD DERIVATIVES AND FUNCTION HANDLES %%%%%%%%%%%%%%%%%%%%%%%%
dfdx = cell(N, 1);
dfdu = cell(N, 1);
dhdx = cell(N, 1);
dhdu = cell(N, 1);
dhdxx = cell(N, 1);
dhduu = cell(N, 1);
dhdxu = cell(N, 1);
dhdux = cell(N, 1);

dynamic.f = cell(N, 1);
dynamic.dfdx = cell(N, 1);
dynamic.dfdu = cell(N, 1);

dynamic.Hamilt.h = cell(N, 1);
dynamic.Hamilt.dhdx = cell(N, 1);
dynamic.Hamilt.dhdu = cell(N, 1);
dynamic.Hamilt.dhdxx = cell(N, 1);
dynamic.Hamilt.dhduu = cell(N, 1);
dynamic.Hamilt.dhdxu = cell(N, 1);
dynamic.Hamilt.dhdux = cell(N, 1);

switchTime = [0 switchTime N];

for n = 1:1:N
    for i = 1:1:Nsys
        if n >= switchTime(i) && n <= switchTime(i+1)
            dfdx{n} = jacobian(f{i}, x);
            dfdu{n} = jacobian(f{i}, u);
            h{n} = l + lambda' * f{i} * dt;

            dhdx{n} = jacobian(h{n}, x)';
            dhdu{n} = jacobian(h{n}, u)';
            dhdxx{n} = hessian(h{n}, x);
            dhduu{n} = hessian(h{n}, u);
            dhdxu{n} = jacobian(dhdx{n}, u);
            dhdux{n} = jacobian(dhdu{n}, x);

            dynamic.f{n} = matlabFunction(f{i}, 'vars', [{x}, {u}]);
            dynamic.dfdx{n} = matlabFunction(dfdx{n}, 'vars', [{x}, {u}]);
            dynamic.dfdu{n} = matlabFunction(dfdu{n}, 'vars', [{x}, {u}]);

            dynamic.Hamilt.h{n} = matlabFunction(h{n}, 'vars', [{x}, {u}, {lambda}]);
            dynamic.Hamilt.dhdx{n} = matlabFunction(dhdx{n}, 'vars', [{x}, {u}, {lambda}]);
            dynamic.Hamilt.dhdu{n} = matlabFunction(dhdu{n}, 'vars', [{x}, {u}, {lambda}]);
            dynamic.Hamilt.dhdxx{n} = matlabFunction(dhdxx{n}, 'vars', [{x}, {u}, {lambda}]);
            dynamic.Hamilt.dhduu{n} = matlabFunction(dhduu{n}, 'vars', [{x}, {u}, {lambda}]);
            dynamic.Hamilt.dhdxu{n} = matlabFunction(dhdxu{n}, 'vars', [{x}, {u}, {lambda}]);
            dynamic.Hamilt.dhdux{n} = matlabFunction(dhdux{n}, 'vars', [{x}, {u}, {lambda}]);
        end
    end

end

dldx = jacobian(l, x)';
dldu = jacobian(l, u)';
dldxx = jacobian(dldx, x);
dlduu = jacobian(dldu, u);
dldxu = jacobian(dldx, u);
dldux = jacobian(dldu, x);
dVfdx = jacobian(Vf, x)';
dVfdxx = jacobian(dVfdx, x);

cost.l = matlabFunction(l, 'vars', [{x}, {u}]);
cost.dldx = matlabFunction(dldx, 'vars', [{x}, {u}]);
cost.dldu = matlabFunction(dldu, 'vars', [{x}, {u}]);
cost.dldxx = matlabFunction(dldxx, 'vars', [{x}, {u}]);
cost.dlduu = matlabFunction(dlduu, 'vars', [{x}, {u}]);
```

```
cost.dldux = matlabFunction(dldux, 'vars', [{x}, {u}]);
cost.dldxu = matlabFunction(dldxu, 'vars', [{x}, {u}]);
cost.Vf = matlabFunction(Vf, 'vars', {x});
cost.dVfdx = matlabFunction(dVfdx, 'vars', {x});
cost.dVfdxx = matlabFunction(dVfdxx, 'vars', {x});
cost.COST = @(X, U) totalCost(cost, X, U);

%% Initialize primal and dual
primal.time = [0:dt:N*dt];
xbar = 5 * rand([nx, 1]) - 2.5;
primal.xbar = xbar;
primal.x = zeros(nx, N+1);
primal.u = 2 * (rand([nu, N])-0.5); % Initial guess between -1 and +1
dual.lambda = 2e-3*(rand(nx, N+1)); % Initial guess between 2e-3 and 0

primal.dx = zeros(nx, N+1);
primal.du = zeros(nu, N);
dual.dlambda = zeros(nx, N+1);

%% Initialize Parameters
params.DELTA_0 = DELTA_0;
params.DELTA_MAX = DELTA_0;
params.dt = dt;
params.tol = N * 0.1e-3;
params.maxIter = maxIter;
params.showIter = showIter;
params.solver = solver; % "TR" : Trust region / "casadi"

%% Forward Propagation to Initalize x as feasible
primal.x(:, 1) = xbar;
for i=1:1:N
    xi = primal.x(:, i);
    ui = primal.u(:, i);

    primal.x(:, i+1) = xi + dynamic.f{i}(xi, ui) * dt;
end

%% OPTIMIZATION : UNCOMMENT THE TEST THAT YOU WANT TO RUN

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%%%%%%%%%%%%%%% COMPARISON WITH CASADI %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% OCP_TR = NOCP(primal, dual, dynamic, cost, params);
% tic
% [x, u] = OCP_TR.solve();
% toc
%
% params.solver = 'casadi';
% OCP_casadi = NOCP(primal, dual, dynamic, cost, params);
% tic
% [x_, u_] = OCP_casadi.solve();
% toc
%
% figure;
% plot([0:1:length(OCP_TR.costCurve)-1], OCP_TR.costCurve, 'Linewidth', 2);
% hold on;
% plot([0:1:length(OCP_casadi.costCurve)-1], OCP_casadi.costCurve, '--', 'Linewidth', 2);
% grid on;
% xlabel("Iteration [-]");
% ylabel("Cost [-]");
% title("Objective Function");
% legend("TR", "CasADi");

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%% EVOLUTION OF THE SOLVING DURATION %%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```
% t = 0;
% tvec = [];
% Nvec = [2:2:N]; % CHOOSE THE HORIZON VALUES TO TEST
% primal.xbar = 5 * rand([nx, 1]) - 2.5;
% params.showIter = false;
%
% for N = Nvec
%     params.tol = N * 0.1e-3;
%     N
%
%     % Variables
%     primal.time = [0:dt:N*dt];
%     primal.x = zeros(nx, N+1);
%     primal.u = 2 * (rand([nu, N])-0.5); % Initial guess between -1 and +1
%     dual.lambda = 2e-3*(rand(nx, N+1)-0.5); % Initial guess between 1e-3 and -2e-3
%
%     primal.dx = zeros(nx, N+1);
%     primal.du = zeros(nu, N);
%     dual.dlambda = zeros(nx, N+1);
%     cost.COST = @(X, U) 0.5 * trace(X(1:N, :) * Q * X(1:N, :)') + 0.5 * trace(U * R * U') + cost
    .Vf(X(N+1, :)');
%
%     primal.x(:, 1) = xbar;
%     for i=1:1:N
%         xi = primal.x(:, i);
%         ui = primal.u(:, i);
%
%         primal.x(:, i+1) = xi + dynamic.f{i}(xi, ui) * dt;
%     end
%
%     t = 0;
%     for i = 1:1:10
%         OCP_TR = NOCP(primal, dual, dynamic, cost, params);
%         tic;
%         [x, u] = OCP_TR.solve();
%         t = t + toc;
%     end
%     tvec = [tvec t/10];
% end
%
% figure;
% plot(Nvec, tvec);
% title("Solving Time : TR & Riccati Recursion");
% xlabel("Horizon [-]");
% ylabel("Duration [s]");
% grid on;

%% Function to compute the total cost (Used to construct a function handle)
function [COST] = totalCost(cost,X, U)
    COST = 0;
    for i = 1:1:length(U)
        COST = COST + cost.l(X(i, :)', U(i, :)');
    end
    COST = COST + cost.Vf(X(end, :)');
end
```

B Riccati Recursion for an Inequality Constrained NOCP, Theoretical Development

B.1 Problem Statement

As a first step, let us consider the non-linear system $\dot{x} = f(x,u)$ with $x \in \mathbb{R}$, $u \in \mathbb{R}$ and $f : \mathbb{R}^2 \to \mathbb{R}$. The objective is to first apply SQP along with trust region method using Riccati recursion to the following optimal control problem :

$$\begin{aligned} \min_{\mathbf{X},\mathbf{U}} \quad & \sum_{i=0}^{N-1} l(x_i,u_i) + V_f(x_N) \\ \text{s.t.} \quad & x_0 - \bar{x} = 0, \\ & x_i + f(x_i,u_i)\delta t - x_{i+1} = 0, \qquad i = 0,...,N-1 \\ & g(x_i,u_i) \leq 0, \qquad i = 0,...,N-1 \end{aligned} \tag{35}$$

Note that the following development is heavily based on [6].

B.2 Lagrangian & KKT conditions

With $\mathbf{X} = \{x_i\}_{i=0}^{N}$ and $\mathbf{U} = \{u_i\}_{i=0}^{N-1}$, δt the sampling time and $g : \mathbb{R}^2 \to \mathbb{R}^2$ represents the constraints on states and inputs. The related inequality constraint is treated with the primal-dural interior point method. Introducing the slack variables $z_0, z_1, ..., z_{N-1} \in \mathbb{R}^2$, we define :

$$r_{g,i} := g(x_i,u_i) + z_i = 0, \qquad i = 0,...,N-1 \tag{36}$$

The barrier function $-\epsilon \sum_{i=0}^{N-1} ln(z_i)$ is also added to the cost function, where $\epsilon \in \mathbb{R}_+$ is the barrier parameter. Introducing the Lagrange multipliers $\mathbf{\Lambda} = \{\lambda_i\}_{i=0}^{N} \in \mathbb{R}^N$ with respect to the dynamical constraints and the Lagrange multipliers $\mathbf{V} = \{\nu_i\}_{i=0}^{N-1} \in \mathbb{R}^{2\times N-1}$ with respect to equation 36, the Lagrangian is therefore :

$$\begin{aligned} \mathcal{L}(\mathbf{X},\mathbf{U},\mathbf{\Lambda},\mathbf{V}) = & \sum_{i=0}^{N-1} l(x_i,u_i) + V_f(x_N) - \epsilon \sum_{i=0}^{N-1} ln(z_i) \\ & -\lambda_0(x_0 - \bar{x}) + \sum_{i=0}^{N-1} \lambda_{i+1}(x_i + f(x_i,u_i)\delta t - x_{i+1}) \\ & + \sum_{i=0}^{N-1} \nu_i(g(x_i,u_i) + z_i) \end{aligned} \tag{37}$$

Then, the KKT's stationnarity condition impose that $\nabla_{\mathbf{X}}\mathcal{L} = \mathbf{0}$ and $\nabla_{\mathbf{U}}\mathcal{L} = \mathbf{0}$. To ease the derivation let's reorganize the notation with on the first line, term depending on primal variables.

$$\begin{aligned} \mathcal{L}(\mathbf{X},\mathbf{U},\mathbf{\Lambda},\mathbf{V}) = & \sum_{i=0}^{N-1} \{\, l(x_i,u_i) + \nu_i g(x_i,u_i) + \lambda_{i+1} f(x_i,u_i)\delta t \,\} + V_f(x_N) - \lambda_0 x_0 + \sum_{i=0}^{N-1} \lambda_{i+1} x_i - \sum_{i=1}^{N} \lambda_i x_i \\ & + \sum_{i=0}^{N-1} \nu_i z_i - \epsilon \sum_{i=0}^{N-1} ln(z_i) + \lambda_0 \bar{x} \end{aligned}$$

To ease notation, let us define the Hamiltonian $\mathcal{H}(x_i,u_i,\lambda_{i+1}) := l(x_i,u_i) + \lambda_{i+1} f(x_i,u_i)\delta t$. The following conditions are obtained :

$$r_{x,N} := \nabla_x V_f(x_N) - \lambda_N = 0$$

$$r_{x,i} := \nabla_x \mathcal{H}(x_i,u_i,\lambda_{i+1}) + \nu_i \nabla_x g(x_i,u_i) + \lambda_{i+1} - \lambda_i = 0, \qquad i = 0,...,N-1$$

$$r_{u,i} := \nabla_u \mathcal{H}(x_i, u_i, \lambda_{i+1}) + \nu_i \nabla_u g(x_i, u_i) = 0, \qquad i = 0, ..., N-1$$

$$\nu_i = \epsilon \nabla_z ln(z_i) \quad \leftrightarrow \quad r_{z,i} := diag(z_i)\nu_i - \epsilon \mathbf{1} = 0 \qquad i = 0, ..., N-1$$

B.3 Newton Step

Let's now define the Newton steps of all variables as $\Delta x_0, ..., \Delta x_N \in \mathbb{R}$, $\Delta u_0, ..., \Delta u_{N-1} \in \mathbb{R}$, $\Delta \lambda_0, ..., \Delta \lambda_N \in \mathbb{R}$, $\Delta z_0, ..., \Delta z_{N-1}$ and $\Delta \nu_0, ..., \Delta \nu_{N-1} \in \mathbb{R}^2$. For example, $\Delta x_0 = x_0^{k+1} - x_0^k$ is the Newton step that will bring the state from x_0^k at iteration k to x_0^{k+1} at iteration $k+1$. These variables will naturally appear by taking a first order approximation of the perturbed KKT conditions. This is illustrated with the primal feasibility constraints. Start with $x_0 - \bar{x} = 0$:

$$x_0^{k+1} - \bar{x} \approx x_0^k - \bar{x} + (x_0^{k+1} - x_0^k) = x_0^k - \bar{x} + \Delta x_0 = 0 \tag{38}$$

For the dynamical constraint $x_i + f(x_i, u_i)\delta t - x_{i+1} = 0$, the linearization gives :

$$\begin{aligned} x_i^{k+1} + f(x_i^{k+1}, u_i^{k+1})\delta t - x_{i+1}^{k+1} \approx x_i^k + f(x_i^k, u_i^k)\delta t - x_{i+1}^k \quad & + \\ \nabla_{x_i}(x_i^k + f(x_i^k, u_i^k)\delta t - x_{i+1}^k)(x_i^{k+1} - x_i^k) \quad & + \\ \nabla_{u_i}(x_i^k + f(x_i^k, u_i^k)\delta t - x_{i+1}^k)(u_i^{k+1} - u_i^k) \quad & + \\ \nabla_{x_{i+1}}(x_i^k + f(x_i^k, u_i^k)\delta t - x_{i+1}^k)(x_{i+1}^{k+1} - x_{i+1}^k) = \quad & 0 \end{aligned}$$

This latter leads to :

$$x_i^{k+1} + f(x_i^{k+1}, u_i^{k+1})\delta t - x_{i+1}^{k+1} \approx \bar{x}_i + A_i \Delta x_i + B_i \Delta u_i - \Delta x_{i+1} = 0, \qquad i = 0, ..., N-1 \tag{39}$$

$$\begin{aligned} \bar{x}_i &= x_i^k + f(x_i^k, u_i^k)\delta t - x_{i+1}^k \\ A_i &= \mathbf{I} + \nabla_{x_i} f(x_i^k, u_i^k)\delta t \\ B_i &= \nabla_{u_i} f(x_i^k, u_i^k)\delta t \end{aligned}$$

Finally, for the state and input constraints :

$$r_{g,i}^{k+1} = g(x_i^{k+1}, u_i^{k+1}) + z_i^{k+1} \approx r_{g,i} + \nabla_x g(x_i, u_i)\Delta x_i + \nabla_u g(x_i, u_i)\Delta u_i + \Delta z_i = 0$$

For the stationarity conditions, the method remains the same.

$$\begin{aligned} r_{x,i}^{k+1} \approx r_{x,i} + [\nabla_{xx}\mathcal{H}(x_i, u_i, \lambda_{i+1}) + \nu_i \nabla_{xx} g(x_i, u_i)] \quad & \Delta x_i \\ + [\nabla_{xu}\mathcal{H}(x_i, u_i, \lambda_{i+1}) + \nu_i \nabla_{xu} g(x_i, u_i)] \quad & \Delta u_i \\ + \nabla_x g(x_i, u_i) \quad & \Delta \nu_i \\ + A_i \quad & \Delta \lambda_{i+1} \\ - \quad & \Delta \lambda_i \\ = \quad & 0 \end{aligned}$$

$$r_{x,N}^{k+1} \approx r_{x,N} + \nabla_{xx} V_f(x_N)\Delta x_N - \Delta \lambda_N = 0$$

$$\begin{aligned} r_{u,i}^{k+1} \approx r_{u,i} + [\nabla_{ux}\mathcal{H}(x_i, u_i, \lambda_{i+1}) + \nu_i \nabla_{ux} g(x_i, u_i)] \quad & \Delta x_i \\ + [\nabla_{uu}\mathcal{H}(x_i, u_i, \lambda_{i+1}) + \nu_i \nabla_{uu} g(x_i, u_i)] \quad & \Delta u_i \\ + \nabla_u g(x_i, u_i) \quad & \Delta \nu_i \\ + \nabla_x f(x_i, u_i) \quad & \Delta \lambda_{i+1} \\ = \quad & 0 \end{aligned}$$

$$r_{z,i}^{k+1} = diag(z_i^{k+1})\nu_i^{k+1} - \epsilon \mathbf{1} \approx r_{z,i} + diag(\nu_i)\Delta z_i + diag(z_i)\Delta \nu_i = 0$$

Nevertheless, since the KKT system is treated as the standard primal-dual interior point method, the Newton direction with respect to the inequality constraints ($\Delta z_0, \Delta z_1, ..., \Delta z_{N-1}$ and $\Delta \nu_0, \Delta \nu_1, ..., \Delta \nu_{N-1}$) must be eliminated.

$$\Delta z_i = -r_{g,i} - \nabla_x g(x_i, u_i)\Delta x_i - \nabla_u g(x_i, u_i)\Delta u_i$$

$$\Delta \nu_i = -diag(z_i)^{-1} r_{z,i} - diag(z_i)^{-1} diag(\nu_i)\Delta z_i$$

$$= -diag(z_i)^{-1} r_{z,i} - diag(z_i)^{-1} diag(\nu_i)\left[-r_{g,i} - \nabla_x g(x_i, u_i)\Delta x_i - \nabla_u g(x_i, u_i)\Delta u_i\right]$$

$$\begin{aligned} &= diag(z_i)^{-1}(diag(\nu_i) r_{g,i} - r_{z,i}) \\ &+ diag(z_i)^{-1} diag(\nu_i)\nabla_x g(x_i, u_i) \quad \Delta x_i \\ &+ diag(z_i)^{-1} diag(\nu_i)\nabla_u g(x_i, u_i) \quad \Delta u_i \end{aligned}$$

Now that Δz_i and $\Delta \nu_i$ are expressed depending on Δx_i, Δu_i, they can be replaced in the KKT expressions. To regroup terms that multiply Δx_i, Δu_i and constants, let's define :

$$Q_{xx,i} = \nabla_{xx}\mathcal{H}(x_i, u_i, \lambda_{i+1}) + \nu_i \nabla_{xx} g(x_i, u_i) + \nabla_x g(x_i, u_i) diag(z_i)^{-1} diag(\nu_i)\nabla_x g(x_i, u_i)$$

$$Q_{xx,N} := \nabla_{xx} V_f(x_N)$$

$$Q_{xu,i} = \nabla_{xu}\mathcal{H}(x_i, u_i, \lambda_{i+1}) + \nu_i \nabla_{xu} g(x_i, u_i) + \nabla_x g(x_i, u_i) diag(z_i)^{-1} diag(\nu_i)\nabla_u g(x_i, u_i)$$

$$Q_{uu,i} = \nabla_{uu}\mathcal{H}(x_i, u_i, \lambda_{i+1}) + \nu_i \nabla_{ux} g(x_i, u_i) + \nabla_u g(x_i, u_i) diag(z_i)^{-1} diag(\nu_i)\nabla_u g(x_i, u_i)$$

$$\bar{l}_{x,i} := r_{x,i} + \nabla_x g(x_i, u_i) diag(z_i)^{-1}(diag(\nu_i) r_{g,i} - r_{z,i})$$

$$\bar{l}_{x,N} := r_{x,N}$$

$$\bar{l}_{u,i} := r_{u,i} + \nabla_u g(x_i, u_i) diag(z_i)^{-1}(diag(\nu_i) r_{g,i} - r_{z,i})$$

$$r_{x,i}^{k+1} \approx Q_{xx,i}\Delta x_i + Q_{xu,i}\Delta u_i + A_i \Delta \lambda_{i+1} - \Delta \lambda_i + \bar{l}_{x,i} = 0, \qquad i = 0, ..., N-1 \tag{40a}$$

$$r_{x,N}^{k+1} \approx Q_{xx,N}\Delta x_N - \Delta \lambda_N + \bar{l}_{x,N} = 0 \tag{40b}$$

$$r_{u,i}^{k+1} \approx Q_{uu,i}\Delta u_i + Q_{xu,i}\Delta x_i + B_i \Delta \lambda_{i+1} + \bar{l}_{u,i} = 0, \qquad i = 1, ..., N-1 \tag{41}$$