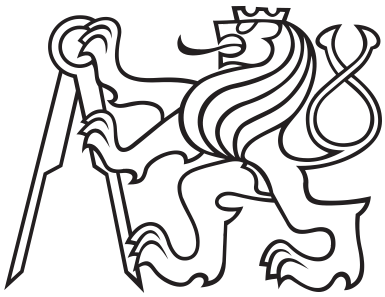
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Control Engineering



MASTER THESIS

A Framework for Nonlinear Model Predictive Control

Prague, January 2016 Ondřej Mikuláš

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Control Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Ondřej Mikuláš**

Study programme: Cybernetics and Robotics
Specialisation: Systems and Control

Title of Diploma Thesis: **A framework for nonlinear model predictive control**

Guidelines:

1. Get familiar with state-of-the-art methods and approaches for Nonlinear Model Predictive Control (NLMPC). Review all important steps in the control design process, such as formulation of NLMPC, model linearization and discretization (i.e. computation of sensitivity matrices), suitable algorithms for solving the resulting optimization problem and final validation of the designed controller in simulation.
2. Prepare a workflow diagram summarizing all important steps needed to design NLMPC controller and implement it in a suitable programming environment (e.g. Matlab or C-language). The framework should be flexible to allow easy reconfiguration of individual components of NLMPC control design process, e.g. specification of the controlled system, or change of the optimization solver.
3. Apply the developed NLMPC framework to a selected system to demonstrate the basic functionality.

Bibliography/Sources:

[1] Rossiter J. A., Model-Based Predictive Control, CRC Press, 2003
[2] Maciejowski J., Predictive control with constraints, Prentice Hall, 2001
[3] Grune, Lars, Pannek, Jurgen, Nonlinear Model Predictive Control, Springer 2011
[4] Relevant journal and conference papers

Diploma Thesis Supervisor: Ing. Ondřej Šantin

Valid until the summer semester 2015/2016

L.S.

prof. Ing. Michael Šebek, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, February 20, 2015

Abstract

Main objective of this thesis is development and implementation of a modular computer framework for nonlinear model predictive control (NMPC). Modularity of the framework is achieved by dividing NMPC algorithm into several logical blocks that can be implemented independently.

The NMPC algorithm is described from the theoretical point of view. Commonly used approaches and individual computation steps are discussed. The control problem formulation is studied and practical guidelines are given to demonstrate how the control objectives can be transformed into the optimization problem cost function and constraints.

The NMPC framework is implemented in Matlab and can be used as a tool for NMPC control development and prototyping. As the framework is modular, different optimization solvers, numerical integration routines and NMPC approaches can be evaluated for any particular application. Functionality of the developed NMPC framework is demonstrated on illustrative control problem examples. The first example is a simplified vehicle steering control and the second one is a diesel engine air-path control.

Abstrakt

Hlavním cílem této práce je návrh a implementace modulárního prostředí pro nelineární prediktivní řízení (NMPC). Modularity je dosaženo rozdělením algoritmu NMPC do samostatných bloků které jsou implementovány nezávisle na sobě.

Algoritmus NMPC je nejprve rozebrán v teoretické rovině. Poté jsou popsány běžně používané postupy a jejich jednotlivé kroky. Dále je vyvětleno jak formulovat požadavky na řízení ve formě kriteriální funkce a omezujících podmínek optimalizační úlohy.

Prostředí pro NMPC je implementováno v jazyce Matlab a může být použito pro návrh a testování nelineárního MPC. Díky tomu že je prostředí modulární, je možné některé bloky nahradit jinými metodami a otestovat tak jejich vliv na celkový výsledek. To platí pro numerickou integraci, optimalizační metody i další části výpočetního procesu. Funkčnost prostředí je předvedena na dvou ukázkových příkladech. Prvním z nich je řízení zatáčení automobilu a druhým řízení vzduchové cesty vznětového spalovacího motoru.

Declaration

I hereby declare that I worked out the thesis individually and that I listed all the literature and software used.

In Prague on 11th January 2016

Ondřej Mikuláš

Acknowledgement

Let me thank to Ondřej Šantin for his kind and helpful supervision of my thesis. Many thanks also come to Honeywell Automotive Software team for providing me with a stimulating environment. Last but not the least, let me thank to my family for their endless support during the course of my studies.

Poděkování

Děkuji Ondřeji Šantinovi za jeho vstřícný a nápomocný přístup při vedení mé diplomové práce. Děkuji také týmu Honeywell Automotive Software, který mi poskytl podnětné prostředí. Nakonec mi dovolte poděkovat mé rodině za jejich bezmeznou podporu po celou dobu mého studia.

Contents

Chapter 1

Introduction

Model predictive control (MPC) is a practical approach that is used to control dynamical constrained systems [1]. MPC uses mathematical model of the system to predict its future behavior on a finite time interval, the *prediction horizon*. This is illustrated in Figure 1.1. The predicted future behavior is optimized using control inputs u with respect to given criteria. The criteria usually describes desired control performance such as setpoint tracking, economical criteria or production quality.

MPC handles systems with multiple inputs and multiple outputs (MIMO) in a systematic way. Furthermore, both under actuated and over actuated systems can be controlled by MPC as well. System constraints are handled in a natural way by including them to the optimization problem. That is a unique feature when compared to other control techniques. The control objectives in MPC are formulated using a cost function. The cost function together with the system constraints creates an optimization problem that needs to be solved at each sampling period.

MPC controller computes optimal control sequence on the prediction horizon, but only the first element of the optimal control sequence is applied to the system. In the next

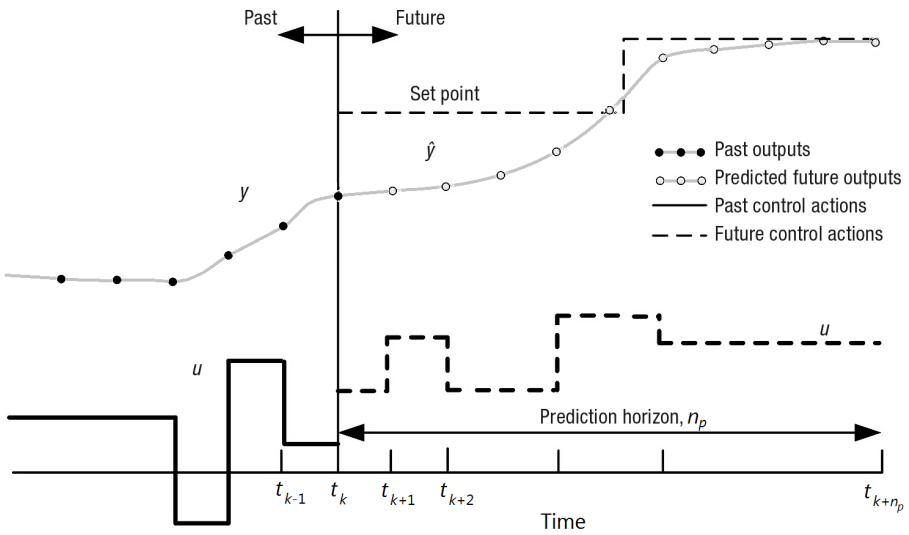


Figure 1.1: Predictive control illustration. The part to the right from time t_k up to time t_{k+n_p} (the prediction horizon) is predicted using the system model.

sampling period, the entire process is repeated with the most recent state estimate or measurement [1]. The concept is known as receding horizon control (RHC) and it incorporates feedback into the MPC control because the control action depends on current state estimate.

Nonlinear model predictive control (NMPC) differs from the general MPC scheme in the fact, that nonlinear process model is used for prediction [2]. Very often, the use of linear models in predictive control is inadequate due to plant-model mismatch. These linear models cannot provide sufficient accuracy of predictions, resulting in poor closed loop performance. As a possible remedy, NMPC can be used to improve the quality of predictions and consequently the closed loop performance.

The objective of this work is to develop a configurable framework for NMPC. The idea is to have a set of tools for fast design, prototyping and validation of NMPC approach applied to user specified control problem. From the user's point of view, the basic usage of the framework should be simple and intuitive so that working prototype of the controller can be obtained rapidly. It will be shown that the NMPC control design is a multistage process that can be divided into several steps, where each step can be implemented separately. This allows for simpler modification and testing the NMPC controllers as well as different NMPC approaches. It also brings an opportunity to compare NMPC performance with other control methods in a rapid way. This is of particular interest because NMPC comes with an additional implementation complexity and computational burden, which must be considered with respect to the possible benefits of application of NMPC.

1.1 Literature Summary

Nonlinear model predictive control has been studied since 1980s. One of pioneering articles on the topic was [3]. General surveys on nonlinear MPC are for example [4] or [5].

As nonlinear model predictive control is quite novel approach, it is not as widely accepted as linear MPC. The authors of a survey paper [6] argue that relatively slow adoption of nonlinear MPC can be due to sevaral factors. The first of them is the difficulty of obtaining reasonable and computationally efficient nonlinear process model. The second is the fact that industrial plants work without serious quality problems under existing control systems (PID, linear MPC) so it is difficult to justify possible benefit of nonlinear MPC. However, a change can be expected when existing control systems are replaced by new ones at their end-of-life.

Nonlinear MPC became first accepted in petrochemical and chemical industries. Here, nonlinear models were available straight from process design which together with slower system dynamics allowed the use of NMPC. A review of approaches used to control chemical processes is provided in [7]. The author mentions not only NMPC approaches used at the time but also other nonlinear control methods together with a number of useful references. Chemical plants are often used as benchmark problems for testing of new control methods. Continuously stirred tank reactor is widely used in the literature [8] because of its relative simplicity combined with significant nonlinear effects.

Nowadays, when the computational power available is no longer prohibitive, many new

applications of NMPC are being reported in the literature. Both academic and practical problems are being approached by NMPC. The range of areas to which NMPC is applied is quite wide.

To name a few we can start with biodiesel production plant control [9]. A first-principle model of the plant is used in NMPC and economical criteria is taken into account when assessing the benefits.

Another interesting area of application is automotive industry. Many researchers dealt with vehicle dynamics and control during limit situations. Stability enhancement, collision avoidance or driver aid under difficult conditions are explored. Due to multivariable and nonlinear nature of such phenomena and constraints present in the system, nonlinear MPC is a method of choice for many authors. See for example [10] or [11], where approximate numerical schemes are used to solve the optimal control problem.

1.2 Existing NMPC Software

There are several NMPC software packages already available. One of them is called YANE and is written by the authors of book [12]. It is implemented in C++ and it can be downloaded from `www.nonlinearmpc.com`. The authors also provide Matlab implementation of nonlinear MPC which was used in their book.

MUSCOD II [13] is another software tool that supports solution of nonlinear optimal control problems. Ordinary differential equation and differential algebraic equation models are supported. The algorithm uses multiple shooting discretization of control problem and tailored sequential quadratic programming method to solve it.

Another package that can be used for nonlinear MPC is called ACADO [14]. It allows the solution of optimal control and parameter estimation problems. It is implemented in C++ and it can be used in connection with Matlab as well via an interface.

There is also a tool available under GNU GPL license called MPC Tools [15]. It is implemented in GNU Octave and contains functions for both linear and nonlinear plant models. The development of the tool seems discontinued.

Another one was developed as a masters project in [16]. It is implemented in Matlab and it also has a simulation graphical user interface. Open-loop predictions can be visually checked during the simulation.

Although there are many tools, a number of which work reasonably well, they are still lacking a user simplicity. Many of the packages require deep understanding of specific optimization methods, ability to formulate the optimal control problem or they are no longer supported. Our aim is to make a tool that would enable the user to have a working implementation of NMPC at hand, while only having to describe the model and to specify the control objectives in an intuitive way. The user should be shielded from most of the technical details regarding the optimization and the computation. The tool should allow the user to run simulations of the designed controller in a rapid way with only a necessary amount of settings. However, more detailed settings should be accessible and changes in default implementation should be possible.

1.3 Thesis Outline

The thesis is divided into six chapters. The second chapter describes NMPC control approach in more detail. It deals with approaches used when formulating the NPMC control problem. Correspondence of the NMPC control problem with a mathematical optimization problem and its solution methods are discussed. The third chapter explains how to formulate a control problem specification as a NMPC problem. Main tools for the formulation are the cost function and the optimization problem constraints. Move blocking is presented as a method of reducing computational complexity of NMPC.

In chapter four, the implemented NMPC framework is introduced. Individual parts of the NMPC design process are described from the implementation point of view and the architecture of implemented framework is described. The last part of the chapter explains how to use the framework and it also serves as a user guide. Illustrative control examples are given in the fifth chapter to demonstrate the functionality of the framework. The last chapter concludes the thesis and summarizes the results.

Mathematical Notation

- $\mathbb{R}$ - set of real numbers
- Lower and upper case italic letters - scalars
- Bold lowercase letters $\mathbf{a}$ to $\mathbf{z}$ - real column vectors
- Bold uppercase letters $\mathbf{A}$ to $\mathbf{Z}$ - real matrices
- $\mathbf{0}$ ($\mathbf{1}$) - vector or matrix of zeros (ones) of corresponding size
- $\mathbf{I}$ - identity matrix of corresponding size
- $\mathbf{a}^{\mathrm{T}}$ ($\mathbf{A}^{\mathrm{T}}$) - vector (matrix) transpose
- $\mathbf{a}_i$ ($\mathbf{A}_i$) - vector (matrix) corresponding to timestep i
- $\star_{(i)}$ - matrix, vector or scalar in the i-th iteration

Chapter 2

Nonlinear MPC

This chapter discusses possible approaches to nonlinear systems using predictive control methodology. In the first part, NMPC is compared to the linear MPC. Architecture of digital control system implementing NMPC is described. Then, an optimal control problem (OCP) is outlined using state-space model. A way to transform the OCP to the form suitable for numerical solution is shown. Sequential quadratic programming (SQP) is described as a method to solve the optimization problem. Finally, three methods of approximating sensitivity of predicted system behavior to the control inputs are shown.

2.1 Linear vs. Nonlinear Predictive Control

In case of linear MPC, predicted state and output trajectory can be directly expressed as a linear function of current state (often viewed as a parameter) and input trajectory (see e.g. [1]). If the cost function is defined as linear or quadratic in states, outputs and inputs the resulting optimization problem is a linear programming (LP) or a quadratic programming (QP) respectively. Both LP and QP can be efficiently solved using high performance solvers. This renders real time application of linear MPC feasible with today's computers and microcontrollers even for very short sampling times [17, 18].

On the other hand, nonlinear system dynamics in NMPC makes it impossible to express exact prediction as a linear function of current state and input trajectory. Therefore, even if the cost function was linear or quadratic (which is frequently the case), the resulting optimization problem would be nonlinear. Nonlinear optimization problems are also known as nonlinear programming (NLP) problems and they are considered very computationally demanding. Hence, completely different approach to the optimization than in case of linear MPC has to be taken.

The cost function should be formulated so that its minimization leads to satisfaction of control objectives such as tracking of reference variables etc. It is control engineer's job to formulate the cost function in this way and this process will be described in more detail in Chapter 3.

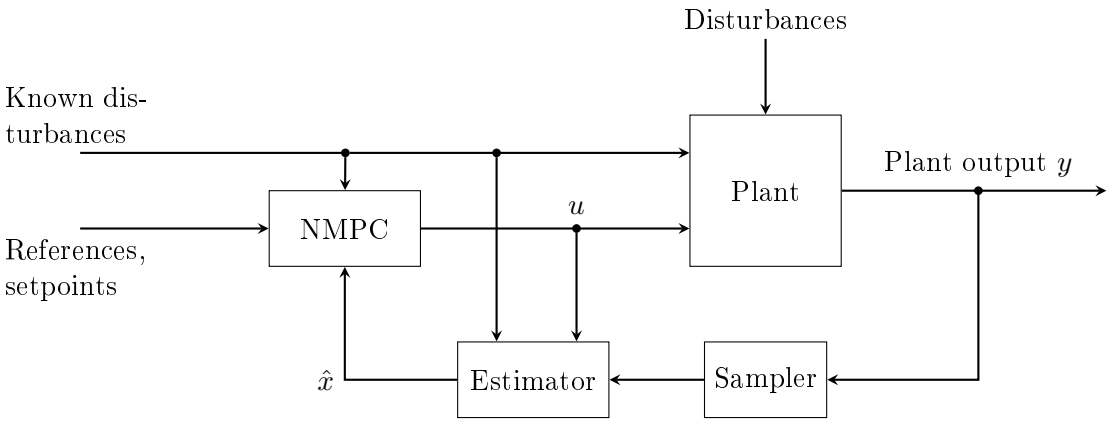


Figure 2.1: Digital control system using nonlinear MPC.

2.2 NMPC Formulation

This section provides formal definition of nonlinear MPC and nomenclature used within the next sections that go into more detail.

In the following, we will assume that the system is to be controlled using digital control system. The situation is depicted in Figure 2.1. The controller marked by the NMPC block takes references, setpoints, known disturbances and current state estimate as its inputs. The state estimates are available only at discrete time instants. At each sampling period, denoted as T_s, the controller solves a finite horizon optimal control problem. The solution of the optimal control problem is an input trajectory over the finite time horizon. The first element of the aforementioned optimal trajectory is applied to the plant using zero-order hold, i.e. the input is held constant between the sampling instants.

This scheme is also described by Algorithm 2.1 from the controller's perspective. Initializing input trajectory $\mathbf{u}_0$ is used to start the algorithm. Current state (or estimate) $\mathbf{x}_k$ an the previous input trajectory $\mathbf{u}_{(k-1)}$ are passed to procedure NMPCstep. The optimal control problem is solved in the procedure NMPCstep once in sampling period. The first element of the optimal input trajectory is applied to the system. Note that references, setpoints and known disturbances also need to be communicated but they are not shown in the Algorithm for the sake of simplicity.

Continuous time model will be used within the controller and so we introduce the following notation to be able to keep both continuous and discrete time notation at the same time. The sampling time instants are denoted t_i and they satisfy $t_{i+1} = t_i + T_s$. System states, inputs and outputs at sampling instants are denoted by

$$\mathbf{u}_i = \mathbf{u}(t_i), \quad \mathbf{x}_i = \mathbf{x}(t_i) \quad \text{and} \quad \mathbf{y}_i = \mathbf{y}(t_i). \tag{2.1}$$

2.2.1 Prediction Model

As was said in the previous parts of this chapter, it is necessary that the controller has some variant of system model. The quality of the model largely determines the success of predictive control application. If the model was not exact, its parameters or structure

Algorithm 2.1 Nonlinear MPC control loop

$k \leftarrow 0$
$\mathbf{u}_{(k-1)} \leftarrow \mathbf{u}_{(0)}$
while TRUE **do**
  get current state $\mathbf{x}_k$
  $\mathbf{u}_{(k)} \leftarrow \text{NMPCSTEP}(\mathbf{x}_k, \mathbf{u}_{(k-1)})$
  apply $\mathbf{u}_{(k),0}$ to the system until the next sampling period
  $k \leftarrow k + 1$
end while

procedure NMPCSTEP($\mathbf{x}_0, \mathbf{u}_{prev}$) ▷ Input trajectory for current prediction window
  minimize predicted cost function $J(\mathbf{x}_0, \mathbf{u}_{prev})$ subj. to constraints
  $\mathbf{u}^* \leftarrow \arg\min J(\mathbf{x}_0, \mathbf{u}_{prev})$
  return $\mathbf{u}^*$
end procedure

were wrong, the model would not describe the actual system behavior correctly. This would cause the predictions to be misleading, and the closed-loop control performance to be deteriorated. Regardless of the optimization methods or prediction algorithms, with inadequate modelling all effort is of very limited use.

System model can be given in many forms. Continuous or discrete time state-space models often originate from physical system description and they can be obtained during the plant design. On the other hand, neural network models, NARMAX models or Wiener-Hammerstein models are usually obtained using model identification techniques based on plant experiment [2].

In this work we restrict ourselves to nonlinear continuous time state-space models. This model class is given by a set of ordinary differential equations (ODEs)

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \tag{2.2}$$

and an algebraic output equation

$$\mathbf{y}(t) = \mathbf{g}(\mathbf{x}(t), \mathbf{u}(t)), \tag{2.3}$$

where $\mathbf{x}(t)$ is system state at time t, $\mathbf{u}(t)$ is control input at time t and $\mathbf{y}(t)$ is system output. The number of state variables is denoted by n, the number of input variables by m and the number of output variables by p respectively. Functions $\mathbf{f} : \mathbb{R}^{n+m} \mapsto \mathbb{R}^n$ and $\mathbf{g} : \mathbb{R}^{n+m} \mapsto \mathbb{R}^p$ are vector valued continuous at least once differentiable functions. Both equations are given in the most general form which allows many real world systems to be described.

Some authors rather use a system of differential algebraic equations (DAE). For the sake of brevity, we stick with ODE and we note that possible algebraic equation ($0 = \mathbf{h}(\mathbf{x}(t), \mathbf{u}(t))$) can be handled as well.

Jacobian Matrix

Jacobian matrix of continuous time state-space model given by equations (2.2) and (2.3) is useful in the formulation of optimization problem that has to be solved online. For the Jacobians of the right hand side function $\mathbf{f}$ of state equation (2.2) we define

$$\nabla_{\mathbf{x}}\mathbf{f} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \quad \text{and} \quad \nabla_{\mathbf{u}}\mathbf{f} = \frac{\partial \mathbf{f}}{\partial \mathbf{u}} = \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \cdots & \frac{\partial f_1}{\partial u_m} \\ \vdots & & \vdots \\ \frac{\partial f_n}{\partial u_1} & \cdots & \frac{\partial f_n}{\partial u_m} \end{bmatrix} \tag{2.4}$$

Analogous definition holds for the right hand side of output equation (2.3) Jacobians $\nabla_{\mathbf{x}}\mathbf{g}$ and $\nabla_{\mathbf{u}}\mathbf{g}$. Partial derivatives of function $\mathbf{g}$ are taken instead.

2.2.2 Optimal Control Problem

Finite horizon optimal control problem that has to be solved at each sampling period starting at time t_k in NMPC is given as follows [19]. The prediction horizon length is given as $n_p \cdot T_s$ where n_p is the number of sampling periods. Initial condition for the state trajectory is given by current state $\mathbf{x}_k$.

$$\min_{\mathbf{u}(t)} \quad \int_{t_k}^{t_{k+n_p}} J(\mathbf{u}(t), \mathbf{x}(t))\mathrm{d}t + J_f(\mathbf{u}(t_{k+n_p}), \mathbf{x}(t_{k+n_p})) \tag{2.5a}$$

$$\text{subject to} \quad \dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \tag{2.5b}$$

$$\mathbf{x}(t_k) = \mathbf{x}_k \tag{2.5c}$$

$$\mathbf{y}(t) = \mathbf{g}(\mathbf{x}(t), \mathbf{u}(t)) \tag{2.5d}$$

$$\underline{\mathbf{u}}(t) \leq \mathbf{u}(t) \leq \overline{\mathbf{u}}(t) \tag{2.5e}$$

$$\text{other constraints} \tag{2.5f}$$

The cost function (2.5a) in the problem has to be selected in a way that control objectives are satisfied when the cost function is minimized. Various forms of J and J_f will be described in Section 3.1. The optimal control problem (2.5) is infinite dimensional due to continuous trajectory, or function, $\mathbf{u}(t)$ and it is therefore not suitable for real-time solution.

2.3 Optimal Control Problem as a Numerical Optimization Problem

An important step in practical solution of optimal control problem (2.5) defining the nonlinear MPC is a transformation into a form that is suitable for numerical solution. The goal is to obtain a finite dimensional optimization problem using an appropriate parametrization of the original problem (2.5). Very clear presentation of the topic can be found in [20] (in German).

The prediction horizon is divided into n_p sampling periods giving rise to a shooting grid $t_k < t_{k+1} < \cdots < t_{k+n_p}$. The shooting grid can be defined arbitrarily, however, equidistant

grid corresponding to the sampling is the most frequently used. Then, the input trajectory is parametrized on each subinterval.

Suitable basis functions, a polynomial, or a constant function can be used at each subinterval. In the end, the input trajectory at each subinterval is described by a vector $\mathbf{q}_i$ of parameters. The simplest approach is to use piecewise constant function where

$$\mathbf{u}(t) = \mathbf{q}_i \quad \text{for} \quad t \in \langle t_i, t_{i+1}). \tag{2.6}$$

This parametrization is equivalent to the assumption of digital control system (see Figure 2.1) that implements the control action to the system using zero-order hold. It is also the most widely used approach. In the following subsections, two direct solution methods of the optimal control problem will be presented. They are called single and multiple shooting respectively.

2.3.1 Single Shooting

In the single shooting method, model simulation and optimization are done sequentially. This process is illustrated in Figure 2.2a. First, the nonlinear model is numerically integrated. This way, dynamic constraints (2.5b) and (2.5d) are effectively eliminated from the problem formulation. Simulated trajectories automatically meet the dynamic constraints. Sensitivity of the cost function to the problem parameters $\mathbf{q}_i$ is obtained. Then, the optimization is carried out. The advantage of single shooting is that of implementation simplicity and also the fact that the iterates during the optimization are always feasible [4].

2.3.2 Multiple Shooting

In case of direct multiple shooting approach state trajectory is split into subintervals as well. Optimization problem parameters are not only control trajectory parameters $\mathbf{q}_i$, but also the states at the beginning of each subinterval, denoted by $\mathbf{s}_i$ [19]. This concept is illustrated in Figure 2.2b. The model equations are integrated on the subintervals and the resulting trajectories are optimized separately. The optimization problem has an additional set of equality constraints enforcing continuity of state trajectory between the subintervals.

$$\mathbf{x}(t_{i+1}) = \mathbf{s}_{i+1}$$

Apart from the input trajectory parameters, the initial state of each subinterval is optimized so that the end of resulting state trajectory matches the beginning of the following one.

Multiple shooting approach has an important advantage. It provides better handling of unstable and highly nonlinear systems than single shooting approach. At the same time optimization problem has favorable numerical properties. The optimization problem is structured in a way that is beneficial for the optimization solver. On the other hand, the number of variables is increased by the intermediate state variables $\mathbf{s}_i$.

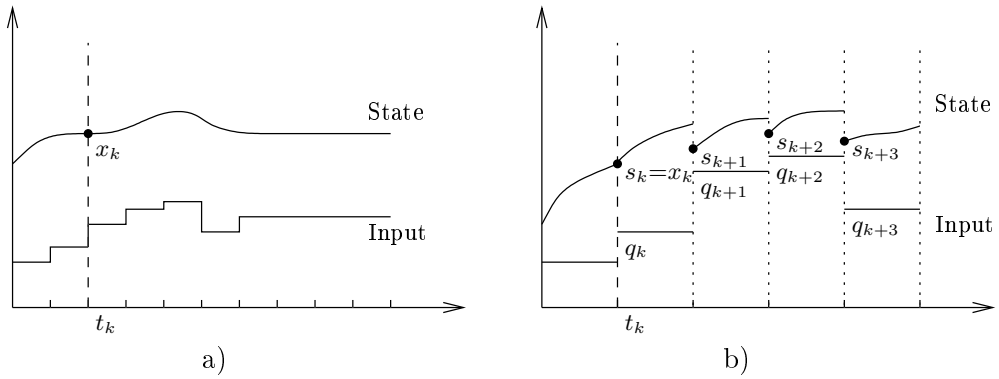


Figure 2.2: a) Single shooting and b) Multiple shooting schematic. In case of multiple shooting, note the discontinuity of state. This captures the intermediate stage of optimization, where the equality constraints are not yet satisfied.

2.3.3 Finite Dimensional Problem

The optimization problem that is obtained can be written as follows.

$$\min_{\mathbf{q}_k,\ldots,\mathbf{q}_{k+n_p-1}} \quad \sum_{i=k}^{k+n_p-1} J_{i,d}(\mathbf{q}_i, \mathbf{x}_i) + J_{f,d}(\mathbf{q}_{k+n_p}, \mathbf{x}_{k+n_p}) \tag{2.7a}$$

$$\text{subject to} \quad \dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \tag{2.7b}$$

$$\mathbf{x}(t_k) = \mathbf{x}_k \tag{2.7c}$$

$$\mathbf{y}(t) = \mathbf{g}(\mathbf{x}(t), \mathbf{u}(t)) \tag{2.7d}$$

$$\underline{\mathbf{u}}(t) \leq \mathbf{u}(t) \leq \overline{\mathbf{u}}(t) \tag{2.7e}$$

$$\mathbf{u}(t) = \mathbf{q}_i \quad \text{for } t \in \langle t_i, t_{i+1}), \;\; i \in k, \ldots, k+n_p-1 \tag{2.7f}$$

$$\text{other constraints} \tag{2.7g}$$

The cost function is now expressed as a sum of functions $J_{i,d}$ and a terminal function $J_{f,d}$. The form of these functions depend on the original cost function in (2.5) and the transformation method used. In case of single or multiple shooting, the functions are integrals over individual shooting subintervals. The integrals are evaluated numerically. This way, the state and input trajectories are considered only at times used by the numerical integrator. It is either a fixed grid or arbitrary time instants given by a solver variable step selection.

There are also other methods to transform the problem (2.5) into a finite dimensional problem. The cost function need not be integrated over time, but only a sum over discrete sampling time instants can be taken. This is often the case because much less computation needs to be done. Basically, the functions $J_{i,d}$ and $J_{f,d}$ are based on discrete samples of state and input trajectories only. It is achieved by simulating the model (2.2) and subsequent linearization and discretization of its dynamics. Similarly, the input constraints (2.7e) are only considered at the sampling time instants $\underline{\mathbf{u}}_i \leq \mathbf{q}_i \leq \overline{\mathbf{u}}_i$. The other constraints (2.7g)

can include e.g. soft constraints on outputs which will be further described in Section 3.2.

In order to solve the optimization problem (2.7), it is important to know the properties of the objective function. In the following, quadratic formulation and the use of transformation methods shown above will be used to describe the cost function.

2.4 Solution of Transformed Optimization Problem

In the previous section, the optimal control problem in NMPC (2.5) was transformed into a finite dimensional nonlinear program (2.7). There are several algorithms that can be used to solve (2.7). Interior point (IP) methods and sequential quadratic programming (SQP) methods.

Interior point methods use additive barrier functions to handle the constraints. The barrier function is added to the original cost function. It grows without bound towards the boundary of the feasible region, which makes such points unattractive [21]. The iterates are always in the interior part of the feasible region given by the constraints. There were only scarce attempts of applying IP methods to NMPC, see e.g. [22].

In the area of NMPC however, SQP is the most frequently used method and so we describe it in more detail.

2.4.1 Sequential Quadratic Programming

SQP is an iterative method for nonlinearly constrained optimization problems. It is guaranteed to find global optimum for convex problems only. If the optimization problem is not convex, it converges to a locally optimal point only. In the following, we will closely follow the presentation in [23]. Suppose we have a nonlinear optimization problem in a form

$$\begin{aligned} \min_{\mathbf{z}} \quad & F(\mathbf{z}) \\ \text{subject to} \quad & \mathbf{c}_E(\mathbf{z}) = \mathbf{0} \\ & \mathbf{c}_I(\mathbf{z}) \geq \mathbf{0} \end{aligned} \tag{2.8}$$

with $\mathbf{z}$ being a vector of optimization variables, F the objective function, $\mathbf{c}_E$ equality constraint function and $\mathbf{c}_I$ inequality constraint function respectively.

SQP iteratively searches for the locally optimal point. The workflow is summarized in Algorithm 2.2. It uses local quadratic approximation of the nonlinear cost function (2.8) to drive the search. Note that the constraints are linearized. In each iteration, it solves local quadratic programming (QP) subproblem and uses the optimizer $\mathbf{p}^*_{(i)}$ of this QP as a search direction. The steplength $\alpha_{(i)}$ is selected according to some strategy and then the step is taken that way to the next iterate. The variable $\mathbf{z}_{(0)}$ denotes the initial (feasible) guess of the solution.

There are two steps in the algorithm that affect the convergence of iteration to the solution. The QP approximation of nonlinear cost function and the strategy of steplength selection. These steps will be discussed in the following subsections. Note that we drop the SQP iteration index i in order to keep the mathematical notation simple.

Algorithm 2.2 Sequential quadratic programming [23]

Require: F, $\mathbf{c}_E$, $\mathbf{c}_I$, feasible $\mathbf{z}_{(0)}$
  $i \leftarrow 0$
  repeat
    find quadratic approximation F_{QP} of F in $\mathbf{z}_{(i)}$
    linearize constraint functions $\mathbf{c}_E$ and $\mathbf{c}_E$ in $\mathbf{z}_{(i)}$
    $\mathbf{p}^*_{(i)} \leftarrow \arg\min F_{QP}$
    select step length $\alpha_{(i)}$
    $\mathbf{z}_{(i+1)} \leftarrow \mathbf{z}_{(i)} + \alpha_{(i)}\mathbf{p}^*_{(i)}$
    $i \leftarrow i + 1$
  until termination condition met
  return $\mathbf{z}_{(i)}$

2.4.2 Local QP Subproblem

As was outlined in the section above, SQP uses quadratic programming approximations to (2.8) recalculated at each iterate. Let us denote current iterate by $\mathbf{z}$ and the new QP variable by $\mathbf{p}$. The local QP approximation of the nonlinear problem can be written as follows:

$$\begin{aligned} \min_{\mathbf{p}} \quad & F_{QP}(\mathbf{p}) \;=\; \frac{1}{2}\mathbf{p}^{\mathrm{T}}\nabla^2 F(\mathbf{z})\mathbf{p} + \nabla F^{\mathrm{T}}(\mathbf{z})\mathbf{p} + F(\mathbf{z}) \\ \text{subject to} \quad & \nabla \mathbf{c}_E(\mathbf{z})^{\mathrm{T}}\mathbf{p} + \mathbf{c}_E(\mathbf{z}) = \mathbf{0} \\ & \nabla \mathbf{c}_I(\mathbf{z})^{\mathrm{T}}\mathbf{p} + \mathbf{c}_I(\mathbf{z}) \geq \mathbf{0} \end{aligned} \tag{2.9}$$

The cost function F_{QP} is given in a standard form of quadratic function using Hessian matrix $\nabla^2 F(\mathbf{z})$ and gradient $\nabla F(\mathbf{z})$ of (2.8) obtained at current iterate $\mathbf{z}$. Note that the constant term $F(\mathbf{z})$ has no influence on the optimizer, but it affects the optimal cost function value. We keep it in the formulation to have consistent information about the cost value between consecutive SQP iterations. The original constraint functions $\mathbf{c}_E$ and $\mathbf{c}_I$ are replaced by their linearizations at current iterate. In the following, we will assume that the solution set defined by the constraints is non-empty. This can be ensured by a proper selection of the constraints in the original NLP.

The local QP (2.9) has a unique minimizer when the Hessian matrix is positive definite [21]. This is crucial for the SQP to work and it can be ensured by appropriate selection of the cost function in (2.7). The local QP is solved using a QP solver which results in the minimizer $\mathbf{p}^*$. The minimizer is used to update the iterate $\mathbf{z}$.

In case of nonlinear model predictive control, the optimization problem (2.8) is quite complex due to the nonlinearity of underlying dynamic model (2.2)-(2.3). To express the Hessian and the gradient of the cost function, it is beneficial to formulate the cost function in a particular way. More specifically, if the cost functions in transformed optimization problem (2.7) is given as a sum of quadratic terms, the local approximation can be constructed using linearized *sensitivity matrices* and input perturbation vector. We will derive sensitivity matrices of the output and state to the manipulated variables in Section 2.5. This way, the state and output trajectories are given as affine functions of input perturbation. Construction of local QP (2.9) in the NMPC will be demonstrated to the end of this

chapter in Section 2.6.

2.4.3 Steplength Selection

The selection of steplength in SQP influences the convergence of the search. There are several strategies of selecting the steplength. One of them is called linesearch. The goal is to find steplength $\alpha < 0$ such that the cost function F restricted to the ray coming from the current iterate $\mathbf{z}$ in a direction of the minimizer $\mathbf{p}^*$ of local QP (2.9) is decreased. Idealy, the cost function should be minimized. This is a nonlinear optimization problem with one decision variable α. However, it is quite complex and therefore it is seldom used in the area of optimal control.

Instead, approximate schemes are used that provide *sufficient* decrease in the cost function along the ray. One of such schemes is called backtracking line search. There are well established conditions that guarantee the decrease. Wolfe or Goldstein conditions [23] both require evaluation of cost function F and its gradient ∇F. During the steplength selection process, the conditions above are checked.

In the case where short computation time is necessary, the steplength is selected according to some heuristic or given sequence [24]. As an example consider so called harmonic steplength sequence

$$\alpha_{(i)} = \frac{\theta}{i} \tag{2.10}$$

with given positive constant θ. Sometimes even a constant steplength may be used. This way the decrease in the cost function is not guaranteed but the computational burden is lower because the evaluation of the cost function F is avoided. It helps keeping the computation time short.

2.5 Sensitivity Computation

Sensitivity is of key importance in the construction of local quadratic subproblems (2.9). It allows one to express the dependence of system states and outputs on the perturbation of input. This section presents two approximation methods and one exact method. The advantage of approximation methods is that in general, they are less computationally demanding. On the other hand, they may be less accurate and it is important to choose the right one for the particular system. It is strongly recommended to test whether the method works reasonably well in a simulation.

We always express sensitivity around given input trajectory $\mathbf{u}_{sim}$. Simulated state and output trajectories $\mathbf{x}_{sim}$ and $\mathbf{y}_{sim}$ are necessary to express the linearized perturbation model. They are obtained using numerical simulation of model (2.2) and (2.3) given the initial condition $\mathbf{x}_k$[1] and input trajectory specified by $\mathbf{u}_{sim}$. Predicted trajectories are then approximated using input perturbation trajectory vector $\delta\mathbf{u} = [\delta\mathbf{u}_k^{\mathrm{T}}, \delta\mathbf{u}_{k+1}^{\mathrm{T}}, \ldots, \delta\mathbf{u}_{k+n_p-1}^{\mathrm{T}}]^{\mathrm{T}}$

[1] Note that the initial condition, the current state, has to be known to the controller. Therefore, either a measurement or an estimate of the state has to be available. A brief discussion of state estimation is provided in Section 3.4.

by the following expressions:

$$\mathbf{u} = \mathbf{u}_{sim} + \delta\mathbf{u}\,, \tag{2.11a}$$
$$\mathbf{x} = \mathbf{x}_{sim} + \mathbf{H}_x\delta\mathbf{u}\,, \tag{2.11b}$$
$$\mathbf{y} = \mathbf{y}_{sim} + \mathbf{H}_y\delta\mathbf{u}\,. \tag{2.11c}$$

In the expressions, $\mathbf{H}_x$ is sensitivity matrix of state $\mathbf{x}$ to input trajectory perturbation $\delta\mathbf{u}$, $\mathbf{H}_y$ is sensitivity matrix of output $\mathbf{y}$ to input trajectory perturbation $\delta\mathbf{u}$. This formulation presumes that the effect of control $\mathbf{u}_{sim}$ is already included in $\mathbf{x}_{sim}$ and $\mathbf{y}_{sim}$ and the approximated effect of perturbation $\delta\mathbf{u}$ can be added.

It is important to note that the equalities in (2.11) represent only internal predicted variables of the controller. Actual system may behave differently under the same control input due to model-plant mismatch. However, our aim is to have the internal variables sufficiently close to the actual ones. Also note that the approximations are valid only for infinitesimal input perturbations as superposition principle does not hold for general nonlinear systems. Therefore, it is necessary to test the actual behavior in simulation.

2.5.1 On-line Model Linearization

One way of treating nonlinear systems using predictive control is to use at each time step the model (2.2) and (2.3) linearized and discretized at the beginning of the prediction horizon, i.e. at single operating point. It is illustrated in part a) of Figure 2.3. The prediction using this method is quite similar to the one used in linear MPC. For this reason, it is called either linear time invariant (LTI) approximation or modified extended linearized predictive control (MELPC). One can find applications of this method e.g. in [25].

First, we express linearized system matrices at current operating point $\mathbf{x}_k$ with control $\mathbf{u}_k$ as Jacobian matrices derived in (2.4).

$$\begin{aligned} \mathbf{A}_c &= \nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x}_k, \mathbf{u}_k)\,, \quad \mathbf{B}_c = \nabla_{\mathbf{u}}\mathbf{f}(\mathbf{x}_k, \mathbf{u}_k)\,, \\ \mathbf{C}_c &= \nabla_{\mathbf{x}}\mathbf{g}(\mathbf{x}_k, \mathbf{u}_k)\,, \quad \mathbf{D}_c = \nabla_{\mathbf{u}}\mathbf{g}(\mathbf{x}_k, \mathbf{u}_k)\,. \end{aligned} \tag{2.12}$$

After that, linearized continuous time perturbation model is given by the following equations:

$$\delta\dot{\mathbf{x}} = \mathbf{A}_c\delta\mathbf{x} + \mathbf{B}_c\delta\mathbf{u}\,, \tag{2.13a}$$
$$\delta\mathbf{y} = \mathbf{C}_c\delta\mathbf{x} + \mathbf{D}_c\delta\mathbf{u}\,. \tag{2.13b}$$

This continuous time model is then discretized with sampling time T_s. This can be achieved either using matrix exponential or using Euler's approximation method [26]. As a result, discrete time perturbation model $\delta\mathbf{x}_{i+1} = \mathbf{A}\delta\mathbf{x}_i + \mathbf{B}\delta\mathbf{u}_i$, $\delta\mathbf{y}_i = \mathbf{C}\delta\mathbf{x}_i + \mathbf{D}\delta\mathbf{u}_i$ is

obtained with

$$\mathbf{A} = e^{\mathbf{A}_c T_s} \approx I + T_s \mathbf{A}_c \,, \tag{2.14}$$

$$\mathbf{B} = \mathbf{A}_c^{-1}(\mathbf{A} - \mathbf{I})\mathbf{B}_c \approx T_s \mathbf{B}_c \,, \tag{2.15}$$

$$\mathbf{C} = \mathbf{C}_c \,, \tag{2.16}$$

$$\mathbf{D} = \mathbf{D}_c \,. \tag{2.17}$$

The prediction is to be done throughout the prediction horizon with this model. The next and the last step necessary is to actually build sensitivity matrices $\mathbf{H}_x$ and $\mathbf{H}_y$. We use the definition of discretized perturbation model to obtain

$$\underbrace{\begin{bmatrix} \delta\mathbf{x}_{k+1} \\ \delta\mathbf{x}_{k+2} \\ \delta\mathbf{x}_{k+3} \\ \vdots \end{bmatrix}}_{\delta\mathbf{x}} = \underbrace{\begin{bmatrix} \mathbf{B} & \mathbf{0} & \mathbf{0} & \cdots \\ \mathbf{AB} & \mathbf{B} & \mathbf{0} & \cdots \\ \mathbf{A^2B} & \mathbf{AB} & \mathbf{B} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}}_{\mathbf{H}_x} \delta\mathbf{u} \,, \quad \underbrace{\begin{bmatrix} \delta\mathbf{y}_k \\ \delta\mathbf{y}_{k+1} \\ \delta\mathbf{y}_{k+2} \\ \vdots \end{bmatrix}}_{\delta\mathbf{y}} = \underbrace{\begin{bmatrix} \mathbf{D} & \mathbf{0} & \mathbf{0} & \cdots \\ \mathbf{CB} & \mathbf{D} & \mathbf{0} & \cdots \\ \mathbf{CAB} & \mathbf{CB} & \mathbf{D} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}}_{\mathbf{H}_y} \delta\mathbf{u} \,. \tag{2.18}$$

This approach may work well for slightly nonlinear systems and for short prediction horizons. For longer prediction horizon, the simulated trajectory can span a large part of state space where linearization from $\mathbf{x}_k$ and $\mathbf{u}_k$ is not valid anymore. However, the computational burden is very low compared to more advanced methods discussed later. This is mainly because the Jacobians (2.12) are only evaluted once per sampling period.

2.5.2 Linearization Along Simulated Trajectory

This method is quite similar to the LTI approximation described above. The difference lies it the fact that instead of single linearized model from the initial state, successive linearization along simulated trajectory is used. This is illustrated in part b) of Figure 2.3. This formulation basically approximates nonlinear system by a linear time varying (LTV) system that is obtained by online linearization along simulated trajectory. There are many successful applications of this method reported in literature, see e.g. [11].

At each timestep of the simulated trajectory $\mathbf{x}_{sim}$ and $\mathbf{u}_{sim}$, for the nonlinear model (2.2) and (2.3) linearization and discretization is carried out. Linearized (continuous time) perturbation model time dependent matrices are given by the Jacobians evaluated in corresponding timesteps $i \in \{k, \ldots, k + n_p - 1\}$ of the simulated trajectory:

$$\begin{aligned} \mathbf{A}_{c,i} &= \nabla_{\mathbf{x}} \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) \,, \quad \mathbf{B}_{c,i} = \nabla_{\mathbf{u}} \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) \,, \\ \mathbf{C}_{c,i} &= \nabla_{\mathbf{x}} \mathbf{g}(\mathbf{x}_i, \mathbf{u}_i) \,, \quad \mathbf{D}_{c,i} = \nabla_{\mathbf{u}} \mathbf{g}(\mathbf{x}_i, \mathbf{u}_i) \,. \end{aligned} \tag{2.19}$$

The subscript i indicates Jacobian obtained at timestep i. Discrete time perturbation model is then given using discretization formulas in (2.14)-(2.17) (for each timestep i) by

$$\delta\mathbf{x}_{i+1} = \mathbf{A}_i \delta\mathbf{x}_i + \mathbf{B}_i \delta\mathbf{u}_i \tag{2.20a}$$

$$\delta\mathbf{y}_i = \mathbf{C}_i \delta\mathbf{x}_i + \mathbf{D}_i \delta\mathbf{u}_i. \tag{2.20b}$$

The prediction is to be done throughout the prediction horizon with this model. The next and the last step necessary is to actually build sensitivity matrices $\mathbf{H}_x$ and $\mathbf{H}_y$. We use the definition of linear time dependent discretized perturbation model (2.20a) to obtain

$$\underbrace{\begin{bmatrix} \delta\mathbf{x}_{k+1} \\ \delta\mathbf{x}_{k+2} \\ \delta\mathbf{x}_{k+3} \\ \vdots \end{bmatrix}}_{\delta\mathbf{x}} = \underbrace{\begin{bmatrix} \mathbf{B}_k & \mathbf{0} & \mathbf{0} & \cdots \\ \mathbf{A}_{k+1}\mathbf{B}_k & \mathbf{B}_{k+1} & \mathbf{0} & \cdots \\ \mathbf{A}_{k+2}\mathbf{A}_{k+1}\mathbf{B}_k & \mathbf{A}_{k+2}\mathbf{B}_{k+1} & \mathbf{B}_{k+2} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}}_{\mathbf{H}_x} \delta\mathbf{u}\,, \tag{2.21a}$$

$$\underbrace{\begin{bmatrix} \delta\mathbf{y}_k \\ \delta\mathbf{y}_{k+1} \\ \delta\mathbf{y}_{k+2} \\ \vdots \end{bmatrix}}_{\delta\mathbf{y}} = \underbrace{\begin{bmatrix} \mathbf{D}_k & \mathbf{0} & \mathbf{0} & \cdots \\ \mathbf{C}_{k+1}\mathbf{B}_k & \mathbf{D}_{k+1} & \mathbf{0} & \cdots \\ \mathbf{C}_{k+2}\mathbf{A}_{k+1}\mathbf{B}_k & \mathbf{C}_{k+2}\mathbf{B}_{k+1} & \mathbf{D}_{k+2} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}}_{\mathbf{H}_y} \delta\mathbf{u}\,. \tag{2.21b}$$

These sensitivity matrices are then used to formulate the optimization problem. The method is more precise than the LTI approximation described in Section 2.5.1 because model dynamics is considered along the sampled simulated trajectory and so the model nonlinearity is captured in a wider operating range. However, this comes at a price of increased computational burden. The model Jacobians has to be evaluated many more times compared to the LTI approximation. More specifically, the LTI approximation evaluates the Jacobians only once, at the beginning of the prediction horizon, while for the LTV approximation the Jacobians have to be evaluated n_p times.

2.5.3 Analytical Calculation of Sensitivity

The last method we mention here is based on the fact that sensitivity of differential equation solution to input and to initial state can be obtained as a solution of another differential equation [27].

The sensitivity of system state $\mathbf{x}(t)$ to the unit step change in input vector at time $t_0 \leq t$ is defined by $\Gamma(t, t_0) = \frac{\partial \mathbf{x}(t)}{\partial \mathbf{u}(t_0)}$. The dynamics of this matrix is given by the following differential equation:

$$\dot{\Gamma}(t, t_0) = \nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \cdot \Gamma(t, t_0) + \nabla_{\mathbf{u}}\mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \cdot \mathbf{1}(t - t_0) \tag{2.22}$$

together with the initial condition

$$\Gamma(t_0, t_0) = \mathbf{0}\,. \tag{2.23}$$

The sensitivity of system output $\mathbf{y}(t)$ to the step change in input vector $\mathbf{u}$ at time $t_0 \leq t$ is given by $\Pi(t, t_0) = \frac{\partial \mathbf{y}(t)}{\partial \mathbf{u}(t_0)}$. It is described by the following expression in terms of

the state sensitivity matrix Γ and the output function Jacobians:

$$\Pi(t, t_0) = \nabla_{\mathbf{x}}\mathbf{g}(\mathbf{x}(t), \mathbf{u}(t)) \cdot \Gamma(t, t_0) + \nabla_{\mathbf{u}}\mathbf{g}(\mathbf{x}(t), \mathbf{u}(t)) \cdot \mathbf{1}(t - t_0)\,. \tag{2.24}$$

The state sensitivity ODEs are simulated together with the model equations over the prediction horizon for the input step changes at $t_k, \dots t_{k+n_p-1}$. Output sensitivity matrices are calculated using the state sensitivity matrices and output equation Jacobians. Once the matrices are evaluated at timesteps $t_k, \dots, t_{k+n_p}$, they are organized into the matrices $\mathbf{H_x}$ and $\mathbf{H_y}$ respectively.

$$\underbrace{\begin{bmatrix} \delta\mathbf{x}_{k+1} \\ \delta\mathbf{x}_{k+2} \\ \delta\mathbf{x}_{k+3} \\ \vdots \end{bmatrix}}_{\delta\mathbf{x}} = \underbrace{\begin{bmatrix} \Gamma(t_{k+1}, t_k) & \Gamma(t_{k+1}, t_{k+1}) = \mathbf{0} & \Gamma(t_{k+1}, t_{k+2}) = \mathbf{0} & \cdots \\ \Gamma(t_{k+2}, t_k) & \Gamma(t_{k+2}, t_{k+1}) & \Gamma(t_{k+2}, t_{k+2}) = \mathbf{0} & \cdots \\ \Gamma(t_{k+3}, t_k) & \Gamma(t_{k+3}, t_{k+1}) & \Gamma(t_{k+3}, t_{k+2}) & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}}_{\mathbf{H}_x} \delta\mathbf{u}\,, \tag{2.25a}$$

$$\underbrace{\begin{bmatrix} \delta\mathbf{y}_k \\ \delta\mathbf{y}_{k+1} \\ \delta\mathbf{y}_{k+2} \\ \vdots \end{bmatrix}}_{\delta\mathbf{y}} = \underbrace{\begin{bmatrix} \Pi(t_k, t_k) & \Pi(t_k, t_{k+1}) = \mathbf{0} & \Pi(t_k, t_{k+2}) = \mathbf{0} & \cdots \\ \Pi(t_{k+1}, t_k) & \Pi(t_{k+1}, t_{k+1}) & \Pi(t_{k+1}, t_{k+2}) = \mathbf{0} & \cdots \\ \Pi(t_{k+2}, t_k) & \Pi(t_{k+2}, t_{k+1}) & \Pi(t_{k+2}, t_{k+2}) & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}}_{\mathbf{H}_y} \delta\mathbf{u}\,. \tag{2.25b}$$

This method increases the number of differential equations to be integrated at each time step. Due to the state sensitivity ODEs, there are $n \cdot m \cdot n_p$ new differential equations to solve. On the other hand, this method is more precise because the model dynamics is considered all along the prediction horizon as depicted in part c) of Figure 2.3. This is in contrast with the two methods above (compare in Figure 2.3) where the model dynamics is accounted for only at selected sampled time steps. This property is beneficial in certain cases. Mostly when the model nonlinearity is significant and should not be approximated. However, this precision is paid for by the increased computational burden caused by the need to numerically solve the new differential equations. Therefore, one must carefully consider if the precision is worth the additional computational burden.

There are numerical software packages that come with ODE sensitivity analysis capabilities. One of them is e.g. SUNDIALS with CVODES solver [28] that supports both analytical sensitivity solution as well as numerical approximation schemes.

Optimal control sequence is obtained from this approximated (through the use of sensitivity) problem. Then the simulation step is repeated with the new control sequence until a termination condition is met.

2.6 Construction of Local QP

It is usual in the area of optimal control to use quadratic cost function. This, together with the sensitivity matrices, allows relatively straightforward construction of local quadratic

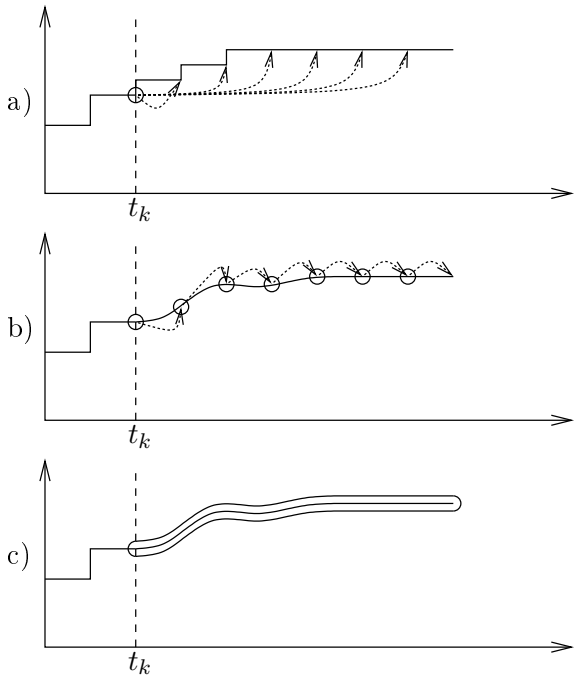


Figure 2.3: Sensitivity calculation methods. a) Linearized model obtained at current time step. b) Linearization along simulated trajectory. c) Sensitivity ODE based approach - sensitivity calculated along trajectory solving augmented ODE system.

optimization problem (2.9). As an example, we use the following quadratic cost function penalizing state:

$$J_x = \frac{1}{2} \sum_{i=k}^{k+n_p-1} \mathbf{x}_i^{\mathrm{T}} \mathbf{Q}_i \mathbf{x}_i = \frac{1}{2} \mathbf{x}^{\mathrm{T}} \mathbf{Q} \mathbf{x} \,, \tag{2.26}$$

with $\mathbf{x}$ being stacked state vectors $\mathbf{x}_i$ and $\mathbf{Q} = \mathbf{Q}^{\mathrm{T}} \succeq 0$ being symmetric positive semidefinite block diagonal penalty matrix with blocks $\mathbf{Q}_i$. The cost function can be further modified to penalize output tracking error, or control increments. This is usually done using additive terms with a specific purpose in mind. More cost function terms are discussed in Section 3.1.. The point is however, to always express the cost as a function of optimized variables, the input perturbation trajectory vector $\delta\mathbf{u}$.

To achieve this, sensitivity matrices derived in Section 2.5 are crucial. Hence, predicted state trajectory $\mathbf{x}$ can be expressed in terms of input perturbation variables using (2.11b). Substituting this expression into the cost function gives us

$$J_x = \frac{1}{2} (\mathbf{x}_{sim} + \mathbf{H}_x \delta\mathbf{u})^{\mathrm{T}} \mathbf{Q} (\mathbf{x}_{sim} + \mathbf{H}_x \delta\mathbf{u}) \,. \tag{2.27}$$

Factoring out the input perturbation trajectory $\delta\mathbf{u}$ yields after a simple manipulation and reordering a standard form of quadratic function.

$$J_x = \frac{1}{2} \delta\mathbf{u}^{\mathrm{T}} \underbrace{(\mathbf{H}_x^{\mathrm{T}} \mathbf{Q} \mathbf{H}_x)}_{\mathbf{G}_x} \delta\mathbf{u} + \underbrace{(\mathbf{x}_{sim}^{\mathrm{T}} \mathbf{Q} \mathbf{H}_x)}_{\mathbf{f}_x^{\mathrm{T}}} \delta\mathbf{u} + \underbrace{\frac{1}{2} \mathbf{x}_{sim}^{\mathrm{T}} \mathbf{Q} \mathbf{x}_{sim}}_{c_x} \tag{2.28}$$

Hessian $\mathbf{G}_x \in \mathbb{R}^{m \cdot n_p \times m \cdot n_p}$, gradient $\mathbf{f}_x \in \mathbb{R}^{m \cdot n_p \times 1}$ and constant $c_x \in \mathbb{R}$ can be directly passed to a QP solver. This way, we formulated a local QP approximation (2.9) using single shooting and the perturbation model (2.11) and sensitivity matrices from Section 2.5. The local QP is solved as a single step of SQP Algorithm 2.2. By this, we are ready to iteratively solve the finite dimensional problem (2.7), obtained in Section 2.3.3, which is an essential ingredient of the NMPC Algorithm 2.1.

Chapter 3

Practical Considerations

As the name of the chapter suggests, topics covered here are more of a practical importance. This chapter describes additional topics and technical details that are useful during the implementation of NMPC algorithm. First, several quadratic cost function terms are described to broaden the scope provided in the previous chapter. Then constraint handling strategies are outlined. Hard constraints are described and two approaches to soft constraints implementation are shown. Move blocking technique that is used to reduce overall computational complexity is shown. Finally, techniques for state estimation of nonlinear systems are briefly introduced.

3.1 Cost Function Terms

The following subsections list several commonly used cost function terms. Their purpose in NMPC and their form will be explained. We will use quadratic terms only, because they allow straightforward formulation of QP. The overall cost function of the local QP approximation (2.9) can be written as a sum of additive terms denoted by $J_\star \in \{J_x, J_u, J_{y,r}, J_{u,r}, J_{\Delta u}, \dots\}$ presented in the following subsections:

$$J = \sum J_\star = \sum \frac{1}{2}\delta\mathbf{u}^{\mathrm{T}}\mathbf{G}_\star\delta\mathbf{u} + \mathbf{f}_\star^{\mathrm{T}}\delta\mathbf{u} + c_\star\,. \tag{3.1}$$

Note that in the end, all the terms have to be transformed to the standard form shown on the right hand side of (3.1) with $\delta\mathbf{u}$ factored out. This transformation can be done in a similar way to the one outlined in Section 2.6. Linearized prediction of state and output trajectories from (2.11) have to be used. However, detailed derivation of this transformation will not be provided for all the terms.

Whenever the cost function consists of multiple terms, the resulting control is always a compromise between several different goals. The individual terms often dictate opposing actions to be taken. Even in very simple cases with as many as two cost function terms. The controller then has to make a decision based on the relative penalty imposed by the cost function terms. Therefore, one has to be careful when formulating the cost function.

3.1.1 Reference Trajectory Tracking

Output reference trajectory is described by the vector $\mathbf{y}_r$ on the prediction horizon. To achieve output trajectory tracking, a cost function term that penalizes tracking error $\mathbf{e}_i = \mathbf{y}_{r,i} - \mathbf{y}_i$ at timestep t_i is used. The controller then attempts to minimize the tracking error over the prediction horizon. Corresponding cost function term can be expressed as follows:

$$J_{y,r} = \frac{1}{2} \sum_{i=k}^{k+n_p-1} (\mathbf{y}_{r,i} - \mathbf{y}_i)^{\mathrm{T}} \mathbf{Q}_{r,i} (\mathbf{y}_{r,i} - \mathbf{y}_i) = \frac{1}{2} (\mathbf{y}_r - \mathbf{y})^{\mathrm{T}} \mathbf{Q}_r (\mathbf{y}_r - \mathbf{y}) \,. \tag{3.2}$$

Block diagonal square matrix $\mathbf{Q}_r = \mathbf{Q}_r^{\mathrm{T}} \succeq 0$ is positive semidefinite and consists of blocks $\mathbf{Q}_{r,i}$. This requirement is necessary for the resulting quadratic optimization problem to be convex. The diagonal blocks $\mathbf{Q}_{r,i}$ correspond to individual time steps of the prediction horizon.

3.1.2 Actuator Position Penalization

Actuator position is penalized to balance the use of actuators. By increasing the weighting for selected actuators, our preferences can be expressed. This is achieved by the following quadratic function:

$$J_u = \frac{1}{2} \sum_{i=k}^{k+n_p-1} \mathbf{u}_i^{\mathrm{T}} \mathbf{R}_i \mathbf{u}_i = \frac{1}{2} \mathbf{u}^{\mathrm{T}} \mathbf{R} \mathbf{u} \,. \tag{3.3}$$

Block diagonal penalty matrix $\mathbf{R} = \mathbf{R}^{\mathrm{T}}$ consisting of blocks $\mathbf{R}_i$ has to be positive definite. Hypothetically, if there was no penalization mechanism for the actuator use, nothing except control limits would stop the controller from applying as much control as possible. This could possibly result in bang bang like control actions, which is generally not considered good practice.

3.1.3 Actuator Movement Penalization

Often, it is a good idea to penalize actuator increments rather than actuator position. This is important because of offset in reference tracking [1]. Any time the reference value requires nonzero control in steady state, there would be a penalty for actuator position. As a result, there would be a compromise between reference tracking performance and actuator position used. Consequently, neither of the conflicting goals could be met exactly.

A simple remedy for this is to penalize control increment $\Delta\mathbf{u}_i = \mathbf{u}_i - \mathbf{u}_{i-1}$. If the tracked output reaches steady state, control should be steady as well. Therefore, forcing $\Delta\mathbf{u}$ and tracking error $\mathbf{y}_r - \mathbf{y}$ to zero produces desired behavior.

One has to express control increments in terms of control trajectory $\mathbf{u}$ and previous

control $\mathbf{u}_{k-1}$.

$$\Delta\mathbf{u} = \underbrace{\begin{bmatrix} I & 0 & 0 & \cdots & \\ -I & I & 0 & \cdots & \\ 0 & \ddots & \ddots & & \\ 0 & 0 & \cdots & -I & I \end{bmatrix}}_{\mathbf{K}} \mathbf{u} + \underbrace{\begin{bmatrix} -I \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{\mathbf{L}} \mathbf{u}_{k-1} \tag{3.4}$$

Having control increments vector expressed, it is quite simple to use it in the cost function. Penalization is then done using quadratic term shown below:

$$\begin{aligned} J_{\Delta u} &= \frac{1}{2} \sum_{i=k}^{k+n_p-1} \Delta\mathbf{u}_i^{\mathrm{T}} \mathbf{R}_{\Delta u,i} \Delta\mathbf{u}_i = \Delta\mathbf{u}^{\mathrm{T}} \mathbf{R}_{\Delta u} \Delta\mathbf{u} \\ &= (\mathbf{K}\mathbf{u} + \mathbf{L}\mathbf{u}_{k-1})^{\mathrm{T}} \mathbf{R}_{\Delta u} (\mathbf{K}\mathbf{u} + \mathbf{L}\mathbf{u}_{k-1}) \,. \end{aligned} \tag{3.5}$$

The penalty matrix $\mathbf{R}_{\Delta u} = \mathbf{R}_{\Delta u}^{\mathrm{T}}$ has to be positive definite and it consists of the blocks $\mathbf{R}_{\Delta u,i}$. The effect of actuator movement penalization is that the resulting optimal trajectory is more likely to be smooth, without large bumps or glitches.

3.1.4 Actuator Reference Tracking

Actuator reference tracking is advantageous in several cases. For example, precomputed optimal (in some sense) actuator positions can be known beforehand. Or when the system is over-actuated. There are more degrees of freedom in the optimization than there is setpoints. In this case, the optimization problem would be ill-posed. Actuator tracking then helps by keeping the inputs near the desired positions and fixing the extra degrees of freedom.

For the actuator reference trajectory given by vector $\mathbf{u}_r$ on the prediction horizon the cost function term is given as follows:

$$J_{u,r} = \frac{1}{2} \sum_{i=k}^{k+n_p-1} (\mathbf{u}_{r,i} - \mathbf{u}_i)^{\mathrm{T}} \mathbf{R}_{r,i} (\mathbf{u}_{r,i} - \mathbf{u}_i) = \frac{1}{2} (\mathbf{u}_r - \mathbf{u})^{\mathrm{T}} \mathbf{R}_r (\mathbf{u}_r - \mathbf{u}) \,, \tag{3.6}$$

where the penalty matrix $\mathbf{R}_r$ is positive semidefinite and consists of diagonal blocks $\mathbf{R}_{r,i}$. Minimizing this term leads to the actuators following the prescribed reference trajectory $\mathbf{u}_r$ if possible.

3.2 Constraints

Constraints or limits are a natural part of model based predictive control algorithm. They enable systematic handling of real system limitations. Constraints are propagated through optimization problem to the control law calculation. The controller does its best to achieve prescribed objectives while obeying the limitations. This is in contrast with other control methods that do not have such feature and constraints has to be implemented ad hoc.

3.2.1 Hard Constraints

Hard constraints are constraints that are applied to the optimization problem. The solver then tries to find minimizer satisfying all the constraints.

Input Constraints

Probably the most obvious and most often used type of hard constraints are constraints on control variables. These constraints are useful for limiting the control action. It is of much practical importance because constraints are often present in actuators. As an example, take e.g. valve opening from 0 to 100 percent. Another example might be heating system that is only capable of delivering positive heat to the system. Input outside of the range does not make any sense and cannot be even implemented.

Input constraints prevent the controller unintended or impossible use of actuators. They are added to the optimization problem in a form of inequalities. Note that in this formulation, they have form of lower bound $\underline{\mathbf{u}}$ and upper bound $\overline{\mathbf{u}}$ respectively.

$$\underline{\mathbf{u}} \leq \mathbf{u} \leq \overline{\mathbf{u}}\,, \tag{3.7}$$

where $\mathbf{u} = \mathbf{u}_{sim} + \delta\mathbf{u}$. The input trajectory $\mathbf{u}_{sim}$ is constant in the optimization problem. Hence, the constraint can be expressed in terms of input perturbation vector as follows:

$$\underline{\mathbf{u}} - \mathbf{u}_{sim} \leq \delta\mathbf{u} \leq \overline{\mathbf{u}} - \mathbf{u}_{sim}\,. \tag{3.8}$$

Other Constraints

The use of hard constraints for system states or outputs is not recommended. This is due to possible optimization problem infeasibility. In case of disturbance entering the system or in case of too strict control requirements, it might happen that no trajectory satisfies the hard constraints. The optimization procedure would return an error due to infeasibility. Then, some kind of ad hoc control policy would have to be applied to recover from the error.

3.2.2 Soft Constraints

As opposed to hard constraints, soft constraints can be violated. Possible violation of these constraints is penalized using a term in the cost function. Comparison of soft and hard constraint is illustrated in Figure 3.1.

Sometimes it is not required to track references exactly but rather keep the outputs inside some predefined ranges or alternatively above or under a limit. Inside the admissible range no penalty is imposed. The penalty is then applied only outside the prescribed range or beyond the limits.

To achieve this kind of behavior, new variable or variables has to be introduced to the optimization problem. We will demonstrate it on soft output maximum limit. Assume that we want to penalize outputs $\mathbf{y}_i$ exceeding $\bar{\mathbf{y}}_i$ at times t_i over the prediction horizon. Basically, there are two implementation options.

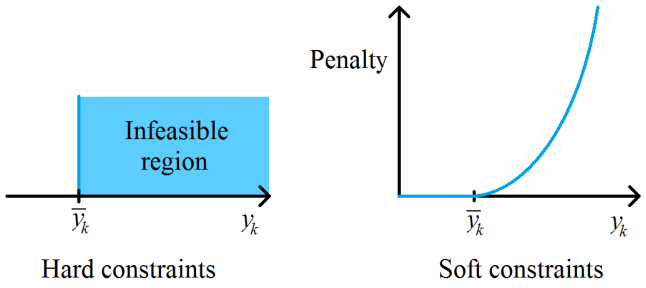


Figure 3.1: Comparison of hard versus soft maximum limits.

Option 1

In this way, there is a new vector of variables $\epsilon \in \mathbb{R}^{p \cdot n_p}$ which quantifies soft limit violation at each timestep of prediction horizon.

$$\begin{aligned} \text{minimize} \quad & J(\mathbf{x}_0, \mathbf{u}_{0,k}) + \frac{1}{2}\epsilon^{\mathrm{T}}\mathbf{G}\epsilon \\ \text{subject to} \quad & \mathbf{y} \leq \bar{\mathbf{y}} + \epsilon \end{aligned} \tag{3.9}$$

with $\mathbf{G} \in \mathbb{R}^{p \cdot n_p \times p \cdot n_p}$ symmetric positive semidefinite. If the soft limit is not violated, i.e. $\mathbf{y} \leq \bar{\mathbf{y}}$, all variables in ϵ are forced to zero by the minimization of $\epsilon^{\mathrm{T}}\mathbf{G}\epsilon$ term. On the other hand, if it is more advantageous (in terms of cost function value) to violate the limit at some point t_i, ϵ_i becomes greater than zero allowing $\mathbf{y}_i > \bar{\mathbf{y}}_i$.

The new constraints in (3.9) contain predicted output trajectory $\mathbf{y}$. After substituting it from (2.11c) into the inequality, the inequality can be rearranged into a standard form with decision variables $\delta\mathbf{u}$ and ϵ

$$\begin{bmatrix} \mathbf{H}_y & -I \end{bmatrix} \begin{bmatrix} \delta\mathbf{u} \\ \epsilon \end{bmatrix} \leq (\bar{\mathbf{y}} - \mathbf{y}_{sim}) \,. \tag{3.10}$$

Option 2

The other way differs from the previous one in the place where the new vector ϵ appears in the optimization problem.

$$\begin{aligned} \text{minimize} \quad & J(\mathbf{x}_0, \mathbf{u}_{0,k}) + \frac{1}{2}(\mathbf{y} - \epsilon)^{\mathrm{T}}\mathbf{G}(\mathbf{y} - \epsilon) \\ \text{subject to} \quad & \epsilon \leq \bar{\mathbf{y}} \end{aligned} \tag{3.11}$$

As can be seen in (3.11), the predicted output trajectory $\mathbf{y}$ appears only in the cost function and not in the constraints. This way, the new constraints have form of simple bounds on variables which is not the case in (3.10). Some QP solvers require exclusive use of box constraints in the problem [29] so only the later variant (3.11) can be used for them.

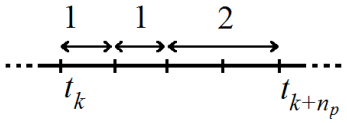


Figure 3.2: Example of move blocking strategy. The block sizes are shown above the timeline.

Complexity Reduction

Soft constrains add more degrees of freedom to the optimization problem by the introduction of the slack variables ϵ in (3.9) and (3.11). In case of real time implementation of NMPC, it is therefore advisable to take measures to avoid unnecessary growth of complexity. This can be done by using only a scalar variable instead of a vector over the prediction horizon. This results in penalizing the maximum of soft constraint violation.

Conceptually, we only replace vector ϵ by $\mathbf{1}\hat{\epsilon}$, with vector of ones with $p \cdot n_p$ entries. This way, only one variable $\hat{\epsilon}$ appears in the optimization problem for each constrained output. It corresponds to the maximum of soft limit violation taken over the prediction horizon.

When such violation occurs, it naturally relaxes the constraint even for the remainder of prediction horizon timesteps. This does not happen when there is a degree of freedom for each timestep and output. On the other hand, it does not increase the complexity of the optimization problem by adding that many new variables.

The other way is the reduction of number of time steps in the prediction horizon where the constraint violation is checked.

3.3 Move Blocking

Move blocking is a way to decrease computational burden during the solution of optimization problem. The point is to effectively reduce the number of decision variables in the problem. It is achieved by fixing the input signal for a period longer than one sampling period. This way, so called *blocks* of different sizes are created. For each block, the input is assumed constant and so only m decision variables are required, equal to the number of inputs.

An example of move blocking strategy is shown in Figure 3.2. The prediction horizon is set to $n_p = 4$ sampling periods. There are only $n_c = 3$ input blocks present of sizes 1, 1 and 2 sampling periods respectively. Therefore, the number of decision variables is decreased from $4 \cdot m$ to $3 \cdot m$. Often, only $n_c - 1$ blocks of unit size are used at the beginning of the prediction horizon and the remainder is replaced by a single block.

The effect of input blocking strategies has been studied in [30]. In general, shorter blocks are useful at the beginning of the prediction horizon. Closer to the end of the prediction horizon, the blocks can be longer, thus saving the number of variables. This is due to the nature of prediction horizon *moving* forward in time. The predicted behavior

does not match actual closed loop behavior because of unknown disturbances and imperfect modelling. The input trajectory far in the future thus rarely gets applied as predicted and so less effort at the end of prediction horizon is often sufficient.

Move blocking can be implemented by fixing corresponding input perturbation vector $\delta\mathbf{u}$ entries to the same values. Assuming the trajectory $\mathbf{u}_{sim}$ respects the block sizes, only the perturbation vector has to be modified. For the example above, the input perturbation vector $\delta\mathbf{u}$ can be represented using a shorter vector $\delta\mathbf{u}_{MB}$ and a move blocking matrix $\mathbf{M}$:

$$\underbrace{\begin{bmatrix} \delta\mathbf{u}_k \\ \delta\mathbf{u}_{k+1} \\ \delta\mathbf{u}_{k+2} \\ \delta\mathbf{u}_{k+3} \end{bmatrix}}_{\delta\mathbf{u}} = \underbrace{\begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & I \\ 0 & 0 & I \end{bmatrix}}_{\mathbf{M}} \underbrace{\begin{bmatrix} \delta\mathbf{u}_k \\ \delta\mathbf{u}_{k+1} \\ \delta\mathbf{u}_{k+2} \end{bmatrix}}_{\delta\mathbf{u}_{MB}}. \tag{3.12}$$

The dimension of move blocking matrix $\mathbf{M}$ is $n_p \cdot m \times n_c \cdot m$ where n_c is the number of input blocks. Instead of full input perturbation vector, only the shorter version $\delta\mathbf{u}_{MB}$ comes out in the equations. The move blocking matrix $\mathbf{M}$ is attached to the other vectors and matrices.

3.4 State Estimation

As was described earlier in Chapter 2, it is necessary to have current state at hand because it is key to efficient prediction of future system behavior. For linear MPC, situation is simpler in the sense that only Kalman filter or even linear estimator is sufficient for state estimation. Linear Kalman filter with additive disturbance estimation can be used for nonlinear systems as well. In general, more complicated methods must be used to cope with nonlinearity. As the nonlinear estimation methods are out of the scope of this work, only two of them are briefly mentioned.

3.4.1 Extended Kalman Filter

De facto standard method for nonlinear estimation is extended Kalman filter (EKF) [31]. Linearized model is used to predict state from previous time step. Linearized model is also used during the data update step. Therefore, Jacobian matrices of the system must be known or approximated during the estimation process. This is major weakness if the system is *strongly* nonlinear.

3.4.2 Unscented Kalman Filter

Other way is unscented Kalman filter (UKF). It uses Unscented transform to approximate prior and posterior probability density functions. Actual probability density function is replaced by a deterministic set of points that is then projected through nonlinear model equations. The resulting posterior mean and covariance are then calculated from these projected points. While it performs well for nonlinear models, it is not increasing computational burden significantly. More detailed explanation can be found in [32].

Chapter 4

NMPC Framework

This chapter describes the framework for nonlinear predictive control. Model based control design process is shortly described. The computational algorithm implemented in the framework is presented together with a workflow diagram. The framework is implemented in Matlab programming environment and code organization into functions and modules is described. In the end, framework usage is explained in detail using a simple example.

4.1 Control Design Process

As with any other model based control strategy, the design of NMPC controller is an iterative process that consists of several steps. It is shown in the diagram in Figure 4.1. Firstly, a suitable plant model has to be defined. An appropriate model form and structure has to be set, model parameters identified and the whole model validated. Secondly, control problem specification has to be found. This includes selection of manipulated variables and controlled variables. Decision on control objectives, e.g. setpoints, constraints or soft limits. These requirements are then translated into the cost function terms and other tuning parameters. The next step is tuning of the controller parameters. Finally, the control performance should be evaluated. It can be done either via numerical simulation or using model in the loop (MIL), hardware in the loop (HIL) or a rapid prototyping system. If the performance provided by designed controller meets expectations, the controller can be implemented and deployed to the actual plant.

4.2 Implemented Algorithm and Features

The framework implements single shooting algorithm with SQP. State space models defined by differential equations and output mappings in the general form of equations (2.2) and (2.3) are supported. Jacobians of model right hand side functions have to be specified. It can be done either by explicitly specifying the formulas or by using numerical approximation. For the second case, there is a helper function to approximate the Jacobian using finite difference scheme. Methods of approximating sensitivity matrices described in Section 2.5 are implemented.

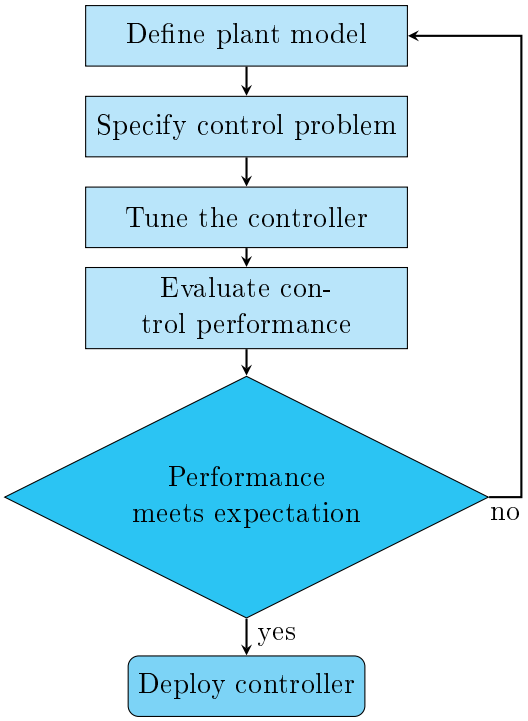


Figure 4.1: Flowchart of model based control design process.

Model inputs are divided into two groups with different meaning. Manipulated variables are those inputs used as free variables in the optimization. Disturbance variables are known and fixed along the prediction horizon. They can be used in place of known disturbances or external input variables supplied by another control system. Future preview information can be used for disturbance variables if available.

The cost function of the optimization problem consists of individual quadratic terms serving specific purpose. The terms selected by the user are then added to make the final form of the cost function. Hard constraints on model inputs can be added. Soft constraints handling is implemented as well to be able to limit selected model outputs. Reduced complexity approach described in Section 3.2.2 is used.

Simulink models are currently not supported for the control design (they cannot be used for prediction). However, there is an interface allowing the use of designed controller in Simulink models to validate the controller. This is achieved by an Simulink S-function[1] block calling framework routines.

4.2.1 Workflow Diagram

Workflow diagram of the computation process implemented in the framework is shown in Figure 4.2. The whole process is implemented as a method of `nmpc` class. Subsequently, it calls methods corresponding to the steps in Figure 4.2 that are implemented separately. The loop in the diagram corresponds to the iteration of SQP optimization method. The individual steps inside the loop prepare the local quadratic approximation of the nonlinear

[1] Simulink S-function is a block type that allows calling user specified Matlab functions from Simulink environment. More specifically, Level-2 Matlab S-function is used in the framework.

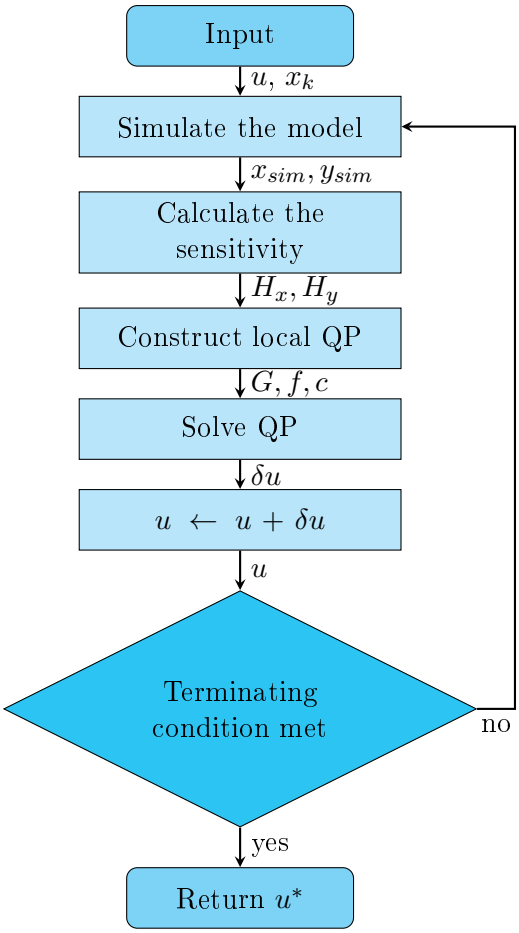


Figure 4.2: Flowchart of NMPC algorithm as it is used in the framework.

optimization problem.

4.3 Programming Environment

The framework is implemented in Mathworks Matlab programming environment. Matlab allows for rapid prototyping and testing. Framework can be used for testing of nonlinear model based predictive control strategy on various systems. Modification to NMPC algorithm can be done in a simple way using Matlab programming language.

As a part of framework, we supply a Simulink S-function block that calls the main framework routines. This way, one can test the NMPC controller designed with the framework even within Simulink models.

4.4 Code Organization

NMPC framework is self contained Matlab package called `nmpc`. Content of the package is listed in Figure 4.3. There are three classes in the package, `model`, `mpc_setup` and `nmpc`. Complete list of class properties and methods is included for reference in Appendix A.

Each class has a specific purpose and it corresponds to a step in the control design process shown in Figure 4.1. The `model` class is used to define the system model. It is the

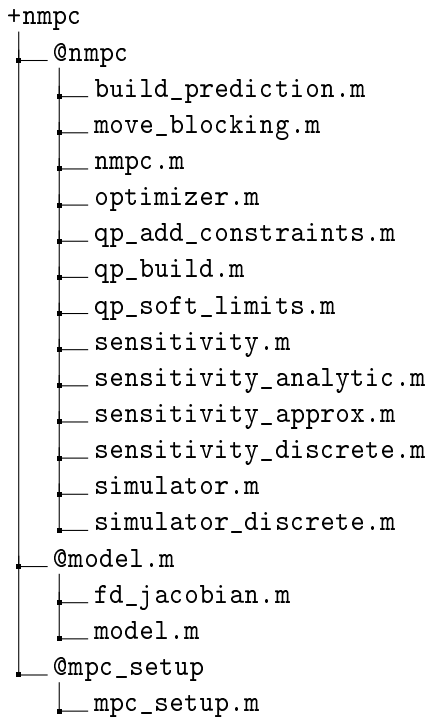


Figure 4.3: Contents of the Matlab package.

first part of NMPC control design process. Model dimensions, equations, their Jacobians and discretization sampling period is specified.

The `mpc_setup` class holds information about the control problem definition and controller tuning. Cost function terms, constraints, inputs and outputs that are intended for reference tracking are selected using this class. NMPC algorithm settings such as SQP stopping condition or sensitivity approximation method are also specified here.

The `nmpc` class takes care of all the computational parts in the framework. It has several methods making the computation process modular. This way, parts of the process can be modified to fit the needs. Different handling of soft constraints can be achieved by modifying the `qp_soft_limits` function. More QP solvers can be used by adding them to the definition in `optimizer` function.

A step by step description of control design process using the implemented framework follows. It is explained using a simple example.

4.5 Guiding Example

The example used in this user guide is control of lateral dynamics of planar vehicle using steering and velocity as manipulated variables. It can be found in the Demos section of framework help within Matlab. The source codes can be copied elsewhere and modified as a starting point for your control problem design.

The dynamic model of the vehicle is given by the following set of differential equa-

Table 4.1: Model variables and Matlab vector order.

	Name	Matlab code
States	x	`x(1)`
	y	`x(2)`
	φ	`x(3)`
Inputs	ω	`u(1)`
	v	`u(2)`
Outputs	x	`y(1)`
	y	`y(2)`

tions [33]:

$$\dot{x} = v\cos\varphi\,, \tag{4.1a}$$

$$\dot{y} = v\sin\varphi\,, \tag{4.1b}$$

$$\dot{\varphi} = \omega\,. \tag{4.1c}$$

Variables x and y are the coordinates of the vehicle in plane, v is forward velocity and ω is angular velocity of the vehicle given by steering command. Variable φ is heading angle of the vehicle. Both v and ω are manipulated variables.

The control objective is to follow prescribed x-y trajectory with limited velocity and steering. Possible application of this problem is e.g. a lane change maneuver or obstacle avoidance. For the vehicle to stay on the road, we set up a soft constraints for y.

In the Matlab code, the notation is slightly different. States and inputs are ordered in the vector and the model variable names are no longer used. The order of variables in the model is summarized in Table 4.1 because it is important in specifying control goals and cost function terms.

4.6 Model Definition

Current implementation of the framework only supports models in state space representation (see Section 2.2.1). Model definition is done using `model` class. It has obligatory properties with predefined default values that will guide user through the process. They are listed for reference in Section A.1. To define a model, create an instance of `model` class using class constructor with the following syntax:

```
new_model = nmpc.model(n,m,p);
```

where `n`, `m` and `p` define the number of states, inputs and outputs respectively.

4.6.1 Manipulated Variables and External Inputs

The framework distinguishes two types of inputs. Manipulated variables, i.e. those inputs that it can directly control, and external inputs (or measured disturbances) that it cannot affect. This has to be defined using `mv` and `dv` property of `model` class. These are vectors containing a list of indices. Note that the sets must cover all `m` inputs and the sets must

have an empty intersection. For example, if the model had 2 inputs, both of them being manipulated variables, the definition would look like

```
new_model.mv = [1;2];
new_model.dv = [];
```

4.6.2 State Equation

State and output equations are defined using a function handle. Basically, there are two ways of getting a function handle. One of them is the use of *anonymous* function and the second one is using a general function defined elsewhere.

Multivariable functions have to be defined in a matrix form with correct number of rows and columns. In the example above, there are three states and two inputs. Function `f` must therefore have input arguments `x` of dimension 3 by 1, and `u` of dimension 2 by 1. The return value is a 3 by 1 (i.e. column) vector of individual state derivatives. Function handle is created using an anonymous function as follows:

```
new_model.f = @(x,u)[u(2)*cos(x(3));
                     u(2)*sin(x(3));
                     u(1)];
```

User also has to supply Jacobians of the functions. This is due to flexibility and to ensure that the derivatives are correct. In our example model, Jacobian with respect to state vector returned by `dfdx` is a 3 by 3 matrix. Jacobian with respect to input vector is returned by `dfdu` is 3 by 2 matrix. The number of rows correspond to the number of states.

```
new_model.dfdx = @(x,u)[0, 0, -u(2)*sin(x(3));
                        0, 0, u(2)*cos(x(3));
                        0, 0, 0];
new_model.dfdu = @(x,u)[0, cos(x(3));
                        0, sin(x(3));
                        1, 0];
```

4.6.3 Output Equation

Output equation is to be defined in an analogous way. The property for output function handle is `g` and Jacobians are defined using `dgdx` and `dgdu`. In our example, the output equation syntax would be as follows:

```
new_model.g = @(x,u)[x(1);
                     x(2)];
```

The Jacobians of output equations are defined using

```
new_model.dgdx = @(x,u)[1, 0;
                        0, 1];
new_model.dgdu = @(x,u)[0, 0;
                        0, 0];
```

4.6.4 Jacobian Approximation

Jacobian matrices should be expressed analytically using the system equations if possible. The differentiation should be done by the user. Under certain circumstances, this can be complicated or undesirable.

The user can approximate a Jacobian matrix using finite differences scheme and use it in place of `dfdx`, `dfdu`, `dgdx` or `dgdu` functions. This can be done using `fd_jacobian` method of `model` class. It implements the forward (one sided) finite difference scheme to approximate the derivatives. Schematically, for a scalar function f of scalar arguments x and u the one sided approximation of derivative is:

$$\frac{\partial f}{\partial x}(x,u) = \frac{f(x+h,u) - f(x,u)}{h}\,, \qquad \text{where } h = px + a\,.$$

The step length h is selected as a p multiple of the evaluation point plus an additive constant a. Both constants are defined in the function file and can be changed if necessary. Usage of the function is as follows. If we want to approximate Jacobian of state function `f` with respect to input vector `u`, we use

```
new_model.dfdu = @(x,u) model.fd_jacobian('f','u',x,u);
```

The two string arguments specify the function and the variable of the Jacobian approximation. The first string argument specifies the function for which the Jacobian approximation is computed. For the state function `'f'` is used and for the output function `'g'` is used. The second argument defines with respect to which variable is the approximation computed. The options are `'x'` for state and `'u'` for input. The last two arguments (`x` and `u`) are the state and input values at which the approximated Jacobian is evaluated.

4.6.5 Sampling Period

The last part of the model definition is sampling period. This is the period used in control discretization for the optimization. It is defined in `Ts` property in seconds.

```
new_model.Ts = 0.02;
```

Note that it should be verified by the user that sampling with this period is appropriate beforehand. If the sampling period is unnecessarily short, the computations can take much longer. If on the other hand the sampling period is too long, sampled system behavior may not capture actual behavior correctly as some fast dynamics may be hidden.

4.6.6 Model Formal Verification

Formal correctness of model can be verified using its `verify` method. It checks the dimensions of functions' return parameters, manipulated variables and disturbance. The function returns 1 if the instance is formally valid and 0 otherwise. If there were any problems, a report and warnings are printed to the Command Window. The method is called by issuing a command `model.verify()`.

4.7 NMPC Problem Definition

NMPC setup is done using `mpc_setup` class. It summarizes control objectives using cost function terms and constraints. Prediction and control horizon lengths are specified here as well. Properties and methods are also provided in Section A.2 for reference.

4.7.1 Receding Horizon Settings

An important part of MPC strategy is the length of prediction and control horizons. They are set using `np` property for prediction horizon and `nc` for control horizon. Please note that both of them are given as a number of sampling periods `model.Ts`. In the example bellow prediction horizon is set to 30 steps and control horizon to 15 steps.

```
mpc_setup.np = 30;
mpc_setup.nc = 15;
```

If no other parameter is specified, the control will remain constant after the `nc` steps. This is so called move blocking. For more details see Section 3.3.

However, if a different move blocking strategy is required, the distribution of control steps among the prediction horizon can be customized. This can be achieved using `blocks` property. It is a column vector containing number of steps between individual control changes.

```
mpc_setup.blocks = [1;1;2;2;3;4;5];
mpc_setup.nc = numel(mpc_setup.blocks);
```

The sum of block lengths should be less than or equal to `np`. If the sum is less than `np`, the last block is automatically prolonged to span the rest of the prediction horizon. The number of elements in blocks must equal the control horizon `nc`. It can be ensured by setting `nc` equal to this number as is demonstrated in the example above.

Note that with the move blocking strategies, one should be very careful, because they may lead to numerical instability or divergence of SQP optimization routine.

4.7.2 Cost Function Terms

This subsection describes the definition of cost function terms for the control problem. There are several cost function terms available. For more detailed description of the cost function please see Section 3.1. The terms are defined using symmetric, positive definite and positive semidefinite matrices. For the optimization problem to have unique solution, at least one of the matrices `R` and `R_du` has to be positive definite. The other nonempty matrices have to be positive semidefinte.

If a term is not present in the cost function, empty matrix has to be used instead. Note that all the cost function weighting matrix properties have empty matrix as a default value. However, we encourage the user to explicitly set empty matrix `[]` for unused terms for clarity.

All of the terms must be specified for the whole prediction horizon. Usually, block diagonal matrices are the most suitable but not mandatory. There should be a block of

correct dimension for each time step in the prediction horizon. For this purpose, Kronecker product of matrices can be quite advantageous.

Control Magnitude

Control magnitude penalization weighting is defined using positive definite `R` matrix. The term corresponds to quadratic cost function in Equation (3.3). It has to be of correct dimension for the whole prediction horizon, that is `np*m_mv` by `np*m_mv`, where `m_mv` is the number of manipulated variables defined in `mv` property (see Section 4.6.1).

Control Increments

Control increment weighting can be set using positive definite `R_du` matrix. The term is defined in Equation (3.5). Control increments are defined using difference between consecutive input steps. The same dimensions hold as for control magnitude weighting matrix `R`. In the example, the weighting is set equally for both inputs and for all the steps in the prediction horizon:

```
mpc_setup.R_du = kron(eye(mpc_setup.np),diag([1,1]));
```

With Kronecker product, block diagonal matrix is created. Each block corresponds to a time step in the prediction horizon. The blocks have a dimension 2 by 2, fitting the number of manipulated variables.

Output Reference Tracking

Output reference tracking is achieved by minimization of weighted tracking error. Tracking error is given by the difference between predicted output $\mathbf{y}$ and output reference signal $\mathbf{y}_{ref}$. First of all, the outputs that are to be tracked have to be selected by column index vector in `y_tr`.

Weighting for the quadratic cost function term (see (3.2)) is specified using positive semidefinite matrix `Q_r`. Correct dimension is `np*p_tr` by `np*p_tr` where `p_tr` is the number of tracked outputs. In our example, the outputs number 1 and 2 are intended for tracking.

```
mpc_setup.y_tr = [1;2];
mpc_setup.Q_r = kron(eye(mpc_setup.np),diag([10,100]));
```

Equal tracking error weighting is set through the prediction horizon. Kronecker product with identity matrix makes block diagonal matrix with `diag([10,100])` blocks. The first tracked output (output number 1) has weight 10 assigned while the second one (output number 2) has 100 assigned.

The output reference trajectories are passed as an argument to `nmpc.control` function. It will be further described in Section 4.8.

Input Reference Tracking

Input reference tracking setup is similar to the output reference tracking. Input reference tracking is used to define preferred actuator position. First, the inputs for which the reference is set are selected using column index vector `u_tr`. Note that indices refer to the system input indexing and not to the manipulated variables (as specified in `model.mv`, see Section 4.6.1).

However, only manipulated variables can be used in input reference tracking. Disturbances, or external inputs, cannot be influenced by the controller and hence it makes no sense to have references for them.

Input reference tracking error $u - u_{ref}$ is penalized by the quadratic cost function (3.6) with a positive semidefinite weighting matrix `R_r`. Correct dimension for this matrix is `np*m_tr` by `np*m_tr` where `m_tr` is the number of tracked inputs.

In the following example we set reference tracking for input number 2, the vehicle velocity (which has been defined to be a manipulated variable in Section 4.6.1). The weight is set to 10 over the prediction horizon.

```
mpc_setup.u_tr = [2];
mpc_setup.R_r = kron(eye(mpc_setup.np),[10]);
```

The input reference trajectories are passed as an argument to `nmpc.control` function and it will be described in Section 4.8.

4.7.3 Control Limits

Input constraints are defined using `u_lb` property for lower bound and `u_ub` property for upper bound. They are to be defined only for manipulated variables, not for disturbance variables. Lower and upper bounds remain constant over the prediction horizon. They must be specified using column vector with correct number of rows. For our example, we can set them up as follows:

```
mpc_setup.u_lb = [-pi/2; 10];
mpc_setup.u_ub = [pi/2; 25];
```

For the first manipulated variable, the limits are given by $-\frac{\pi}{2} \leq u_i \leq \frac{\pi}{2}$. For the second manipulated variable, the limits are $10 \leq u_i \leq 25$.

In case we decided to have no limit on a manipulated variable, we enter ∞ or $-\infty$ for the corresponding upper or lower bound respectively. In Matlab syntax, `inf` and `-inf` can be used.

If there were disturbance variables, limits are always defined only for the manipulated variables. The number of entries in `u_lb` and `u_ub` should be the same as the number of manipulated variables.

4.7.4 Output Soft Limits

Output soft limits can be entered to the problem using indexing to select which outputs to constrain. The indices are to be given in `y_min` and `y_max` column vectors. The limit

values itself are supplied in column vectors `y_min_lim` and `y_max_lim` for selected outputs only. The weighting of soft limits in the cost function is given in `G_min` and `G_max` matrices. The size of these vectors (`y_min_lim`, `y_max_lim`), and matrices (`G_min`, `G_max`) must be compatible with the number of outputs selected for output reference tracking.

In the following example, we set soft minimum and maximum limit on output 2.

```
mpc_setup.y_min = [2];
mpc_setup.y_min_lim = [-1];
mpc_setup.G_min = 1000;

mpc_setup.y_max = [2];
mpc_setup.y_max_lim = [6];
mpc_setup.G_max = 1000;
```

The weighting is set to one thousand for both limits. Output 1 is desired to be kept within -1 and 6. Note that soft limits might be violated if there are other conflicting goals such as output or input reference tracking. Soft constraints are discussed in greater detail in Section 3.2.2.

4.7.5 NMPC Algorithm Setup

Current implementation of the framework provides several options regarding the NMPC algorithm itself. The optimization problem is solved using sequential quadratic programming. The SQP method uses local quadratic programming approximation of the nonlinear cost function. User can specify which approximation method to use.

Sensitivity Calculation Method One can select the way how the local approximations of the cost function are calculated. This is done using `sens_type` property. Possible options are given as a string value:

- `'lti_mpc'` linearization at the beginning of prediction horizon (see Section 2.5.1). Perturbation model is obtained at current operating point $\mathbf{x}_k$ and $\mathbf{u}_k$. Very fast but suitable only for systems with insignificant nonlinearity or short prediction horizons. When speed is a priority and precision can be sacrificed.

- `'ltv_mpc'` linearization along simulated trajectory (see Section 2.5.2). Perturbation model is obtained at each timestep of the simulated trajectory. Fast and suitable for general type of nonlinear systems.

- `'analytic'` analytical sensitivity calculation using augmented ODE system (see Section 2.5.3). Extra differential equations have to integrated. The slowest one, suitable when precision is a top priority or the nonlinearity is strong.

For discrete time models, there are two options. They differ from their continuous time counterparts only in leaving out the discretization of model dynamics.

- `'lti_discrete'` linearization at the beginning of prediction horizon. Perturbation model is obtained at current operating point $\mathbf{x}_k$ and $\mathbf{u}_k$. Very fast but suitable for systems with negligible nonlinearity or when speed is more important than precision.
- `'ltv_discrete'` linearization along simulated trajectory. Perturbation model is obtained at each timestep of the simulated trajectory. Fast and suitable for a wide range of nonlinear systems.

Any other value is invalid an results in error. In our example with continuous time model, we choose linear time varying approximation. The nonlinearity in the model not negligible so `lti_mpc` method is not very suitable. On the other hand, `analytical` method gives almost identical results with significantly longer computation times. Hence, the `ltv_mpc` method is reasonable compromise between precision and speed:

```
mpc_setup.sens_type = 'ltv_mpc';
```

SQP Stopping Condition There are two settings regarding SQP optimization method. Maximum number of SQP iterations may be specified using `sqp_max_iter` property. The default value is 5 iterations which proved to be sufficient most of the time. Valid options are integer values greater than zero. Lets try to decrease the maximum allowed number of iterations to 3.

```
mpc_setup.sqp_max_iter = 3;
```

Another option for SQP algorithm is stopping condition for update of control trajectory. If the difference of control trajectories in consecutive SQP iterations $\mathbf{u}_{(k)}$ and $\mathbf{u}_{(k-1)}$ falls bellow prescribed value, the SQP is stopped and the last iterate $\mathbf{u}_{(k)}$ is returned. The update is measured by 2-norm of the difference divided by the 2-norm of the control in previous iteration:

$$u_{update} = \frac{||\mathbf{u}_{(k)} - \mathbf{u}_{(k-1)}||_2}{||\mathbf{u}_{(k-1)}||_2}, \quad \infty \quad \text{if} \quad ||\mathbf{u}_{(k-1)}||_2 = 0 \tag{4.2}$$

Valid options for the update stopping condition are positive scalars and zero. Default value is 0, so that the SQP will run all `sqp_max_iter` iterations. The SQP is terminated as soon as the update defined in (4.2) is less than or equal the predefined threshold value.

The stopping value is defined using `sqp_u_update` property. As an example, we define the threshold value to be 0.5 percent update between consecutive iterations.

```
mpc_setup.sqp_u_update = 0.005;
```

QP Solver Currently, there are two QP solver interfaces supported in the framework. The first of them is `quadprog` from Matlab's Optimization toolbox and the other is `qpOASES` [34].

Neither of the solvers is included in the package due to licensing policy. Therefore, the solver must be installed on the system and present at Matlab path.

The solver is selected using `qp_solver` property. Valid options are `'quadprog'` and `'qpoases'`. In our example, we use `quadprog`:

```
mpc_setup.qp_solver = 'quadprog';
```

4.7.6 Formal Verification of MPC Settings

There is a function `verify` that checks the formal correctness of MPC settings. It works by comparing the dimensions of the matrices, consistency of bounds etc. For the method to work, a model instance has to be passed as an input argument. This is due the fact that e.g. matrix dimensions depend on the model settings.

Call the function by `mpc_setup.verify(model)`. It returns 1 if the settings are correct and 0 otherwise. If there were any flaws, a short report with warnings is printed to the Command Window so that one can fix them.

4.8 Controller Simulation

Actual computation of control law takes place in the `nmpc` class. Complete list of properties and methods is present in Section A.3 for reference. To start using the NMPC computation, one has to create an instance of `nmpc` class. The class constructor takes valid and compatible instances of `model` and `mpc_setup` classes that were created above:

```
nmpc_ctrl = nmpc.nmpc(model,mpc);
```

Controller simulation is not part of the framework as the requirements for the simulation are usually very problem-specific. For this reason, we only include two fully configured simulation examples so that one can use them as a starting point for his/her own code. To run the simulation, one has to program a loop that calls controller code and passes references, current state and previous input vector. System simulation over the sampling period is carried out externally as well as disturbance trajectory generation. In the course of simulation, the data should be stored and a suitable presentation e.g. in the form of graphs should be tailored to fit the problem.

One of possible simulation schemes is illustrated in Figure 4.4. The portion implemented by NMPC Framework is highlighted using dark background color. Detailed description of `control` function follows in the next section.

4.8.1 Control Function

Control function is the one used for the calculation of control trajectory for given initial state. It takes current state, control trajectory from the previous step and input and output references over the prediction horizon as input arguments. It returns optimal control trajectory and predicted state and output trajectories over the prediction horizon. It is a method of `nmpc` class and has the following syntax:

```
[u_opt,x_pred,y_pred,fval] = obj.control(x0,u_last,y_ref,u_ref);
```

The `obj` is a valid instance of `nmpc` class for which the control action is to be computed.

Input arguments of the `control` function are:

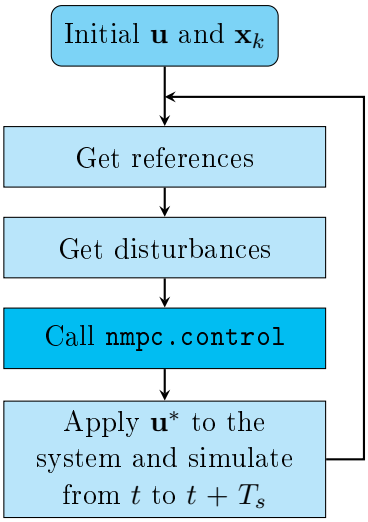


Figure 4.4: Possible simulation scheme. Part implemented by the framework is highlighted.

- `x0` Current state value. Column vector with `n` entries. It is used as an initial condition for the model simulation inside the controller.
- `u_last` Previous input trajectory matrix `m` by `np`. It contains previous input trajectory given by column vectors for each sampling period organized side by side in the matrix. It is used as an initial guess during the first iteration of SQP. When calling the `control` method for the first time, use a meaningful and feasible value. During the simulation, you may use the previous result (`u_opt`) or start over.
- `y_ref` Output reference trajectory over the prediction horizon. It is a `p_tr` by `np` matrix consisting of output references at each sampling period of the prediction horizon. If reference preview is turned off, single column vector can be used instead (`p_tr` by 1). The trajectory is then assumed constant over the prediction horizon.
- `u_ref` Input reference trajectory over the prediction horizon. A `m_tr` by `np` matrix with manipulated variables trajectory at each sampling period of the prediction horizon. If reference preview is turned off, single column vector can be used instead (`m_tr` by 1). The trajectory is then assumed constant over the prediction horizon.

The function return values are:

- `u_opt` Optimal input trajectory, disturbance variables included.
- `x_pred` Predicted state trajectory with control given by `u_opt`.
- `y_pred` Predicted output trajectory with control given by `u_opt`.
- `fval` Optimal predicted cost function value given the predicted trajectories.

All the trajectory signals are organized in matrices. The value of signal at each time step relative to the current time is a column in the matrix. This is shown on the example of `u_opt` below:

$$\mathbf{u}_{opt} = \underbrace{\begin{bmatrix} \mathbf{u}_{opt,k} & \mathbf{u}_{opt,k+1}, & \cdots & \mathbf{u}_{opt,k+np-1} \end{bmatrix}}_{np} \Big\} m$$

Note that np is the number of steps of prediction horizon and m is the number of model inputs. For other trajectories, corresponding dimensions apply.

4.9 Framework Installation

The framework package is distributed as a compressed archive. To obtain the latest version available, please contact the author [2] for details.

The framework package can be installed by copying all the files and folders from a compressed archive to a folder that is accessible by Matlab. Then, the folder containing the framework package has to be added to Matlab path. This way, the toolbox is automatically recognized at the next Matlab start-up and the above mentioned commands become available. The help files are also prepared and so either a Documentation Viewer window or standard command line tools can be used to browse the help as described in the following section.

4.10 Built-in Help and Documentation in Matlab

Toolboxes provided with Matlab contain extensive help and examples as part of the package. Matlab users are familiar with tools to get help during the work with Matlab. Therefore, it is natural to include code documentation and demo examples as well. The help for Matlab help command is automatically extracted from specifically formatted comments in the source code. Function declarations have a unified format to allow simple use and modification of the code.

Standard Matlab help and documentation commands can be used to get more information about individual functions or classes. To list the properties or methods of a class (e.g. `model`), run one of the following commands:

```
properties nmpc.model
methods nmpc.model
```

The command output contains only property and method names respectively. To see the entire class description, use `doc` command instead:

```
doc nmpc.model
```

The command opens Matlab Documentation Viewer and you can browse the class hierarchy. To get help on individual class methods, either browse to them using Documentation Viewer or use the `doc` or `help` commands.

The help contains short method summary, syntax description and list and meaning of input and output arguments. As an example, help for `control` method of `nmpc` class is obtained using any of the following commands:

```
doc nmpc.nmpc.control
help nmpc.nmpc.control
```

[2] Ondřej Mikuláš, e-mail address `mikulon2@fel.cvut.cz`

Matlab toolbox help is prepared using several definition files that describe the structure and linking of individual documentation parts [35]. Illustrative extract from the toolbox documentation can be found in Appendix B. Apart from the toolbox description, two fully configured demo examples are present in the Matlab documentation. These examples can serve as a starting point and guidance for new users.

Chapter 5

Application Examples

5.1 Vehicle Steering Control

The first application example is a planar vehicle steering control. The purpose is to demonstrate the capabilities of the NMPC Framework with a very simple model before proceeding to a more complex one. Even though the model used is extremely simple, it cannot be handled with linear MPC due to nonlinearity given by geometry of the problem.

5.1.1 System Description and Model Derivation

Suppose a planar vehicle idealized by a point mass vehicle model [33]. The state variables of the model and their geometric relations are shown in Figure 5.1. State variable x is a longitudinal coordinate, y is a lateral coordinate and ϕ is heading angle with respect to positive x semiaxis. Input variables are vehicle forward velocity v in the heading direction and ω is the angular velocity. The model dynamics is given by the following differential equations:

$$\dot{x} = v\cos\varphi, \tag{5.1a}$$

$$\dot{y} = v\sin\varphi, \tag{5.1b}$$

$$\dot{\varphi} = \omega. \tag{5.1c}$$

Model state variables can be compactly written in state vector $\mathbf{x}(t) = [x(t), y(t), \phi(t)]^{\mathrm{T}}$ and model inputs can be written in the input vector $\mathbf{u}(t) = [v(t), \omega(t)]^{\mathrm{T}}$. Model outputs are identical to state variables, thus $\mathbf{y}(t) = \mathbf{x}(t)$.

5.1.2 Control Objectives

The control objective is tracking of predefined x-y reference trajectory. In practice, the trajectory is provided by higher level planning algorithm that can use a world map or a local measurements (e.g. a radar or a camera vision system) to plan feasible trajectory. NMPC controller then ensures that this trajectory is tracked using available system inputs.

Double lane change maneuver [36] will be used to demonstrate the capabilities of the NMPC Framework. The maneuver is used to test the stability of vehicles during emergency

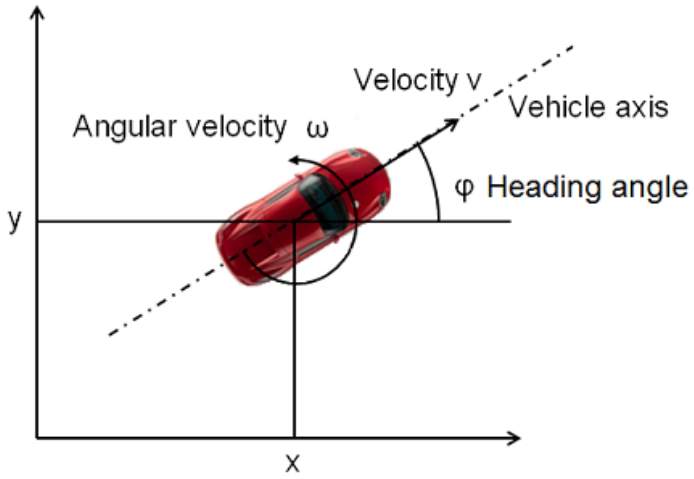


Figure 5.1: Point mass vehicle model.

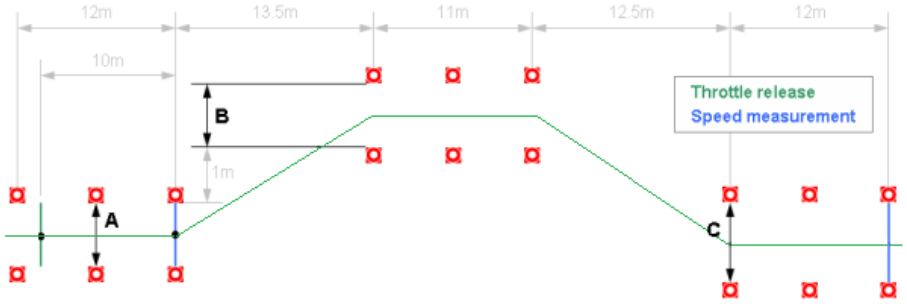


Figure 5.2: Marked track used during double lane change maneuver testing (Adapted from [37]).

obstacle avoidance situations. The car has to go to the left lane and return back to the original lane shortly after that. The track is usually marked using traffic cones and the goal is to go through the track without hitting any cone and without the vehicle rolling over. Exact dimensions of the test track are shown in Figure 5.2. The lane widths A, B and C depend on the vehicle width. A margin of half the vehicle width around the cones was set. The example planned trajectory of vehicle center is shown in green color. Note that the predefined x-y trajectory also determines velocity at each time step.

5.1.3 Framework Configuration

The framework is configured in the same way as described in sections starting from Section 4.5. The same problem has been used as a demonstration example. For further details about controller tuning and corresponding framework settings please refer to the above mentioned sections.

5.1.4 Simulation Results

Planar x-y trajectory resulting form the controller simulation is shown in Figure 5.3. End of maneuver is at 61 metres and 3.08 s, it is highlighted by black dashed vertical line. It can

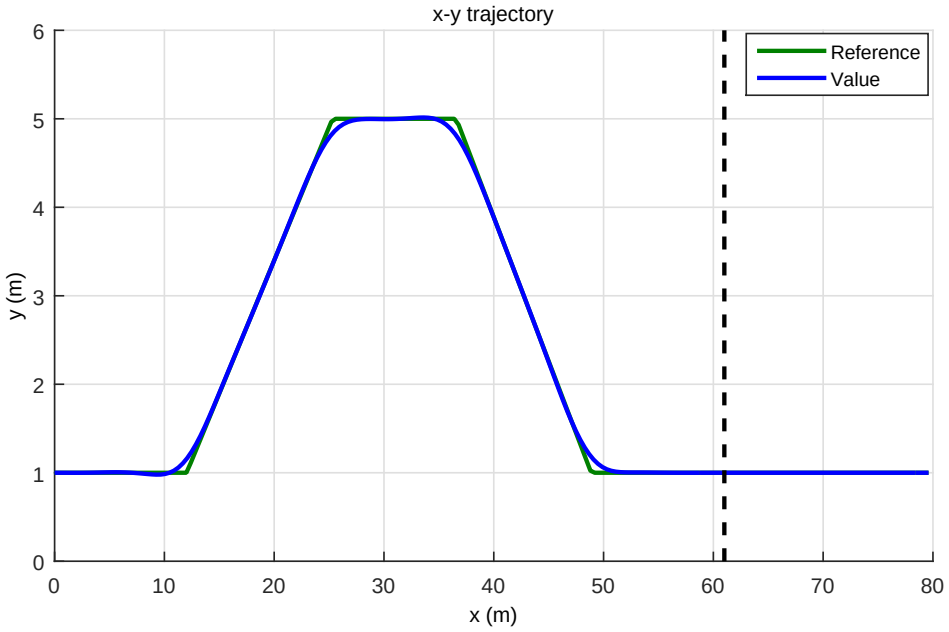


Figure 5.3: x-y trajectory for double lane change maneuver.

be seen that the vehicle is able to track the predefined reference trajectory. Discrepancies can be seen in the sharp corners of the trajectory at 12 m, 25.5 m, 36.5 m and 59 m of longitudinal distance. This is due to limited angular velocity ω. With given speed, this also restricts vehicle turning radius.

The individual components of the trajectory, i.e. x and y coordinates are plotted separately in Figure 5.4. It can be seen that the y trajectory violates soft limits. This caused by the conflicting goals of tracking the reference signal and not violating soft limits. Due to the weighting used, the controller rather slightly violates the soft limit at 1.8 s and tracks the reference signal more tightly between 1 and 2 s. Otherwise, it would have to start turning earlier, departing from the reference even further at the time. This behavior can be influenced by controller tuning. More emphasis could be put on the soft limits if desired using penalty matrices `G_min` and `G_max` respectively.

Heading angle and input signals are plotted in Figure 5.5. Input constraints are shown in red coloring. Note that these constraints are implemented as hard constraints and hence they cannot be violated at all (compare with soft limits on y coordinate).

5.2 Combustion Engine Air Path Control

The second example shows control of diesel engine air path. Realistic configuration of control problem similar to [38] is used to show the capabilities of the framework. The referenced paper mainly deals with control of intake manifold pressure surge during heavy transient operation. This is simulated by a tip-in maneuver. In this example, similar problem will be handled. However, different type of engine is used and so the issue solved by the original paper is not very distinct.

Internal combustion engine control became a challenging control problem. Since the

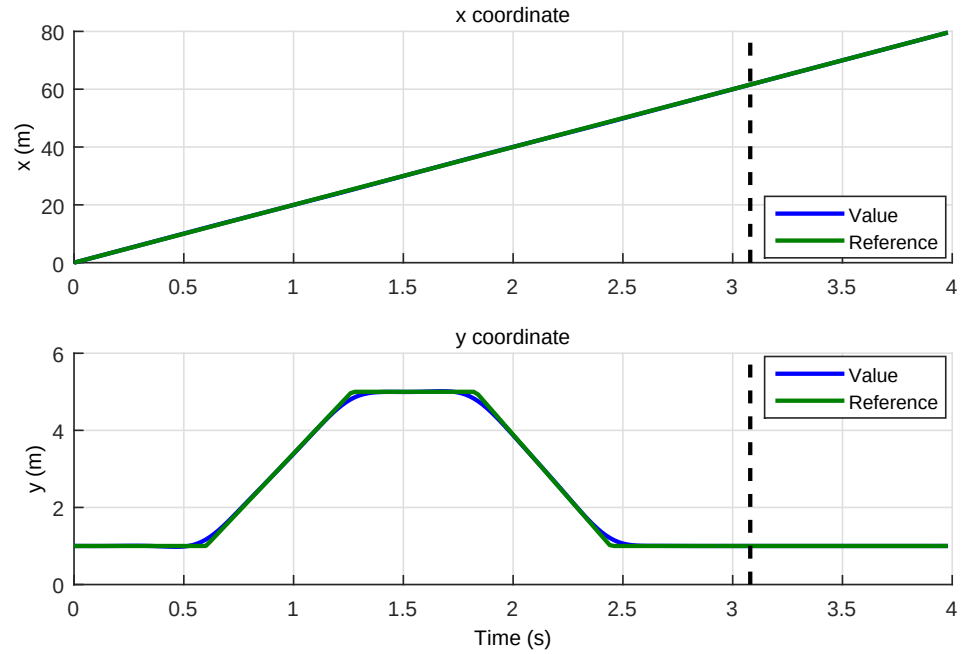


Figure 5.4: x and y trajectories for double lane change maneuver.

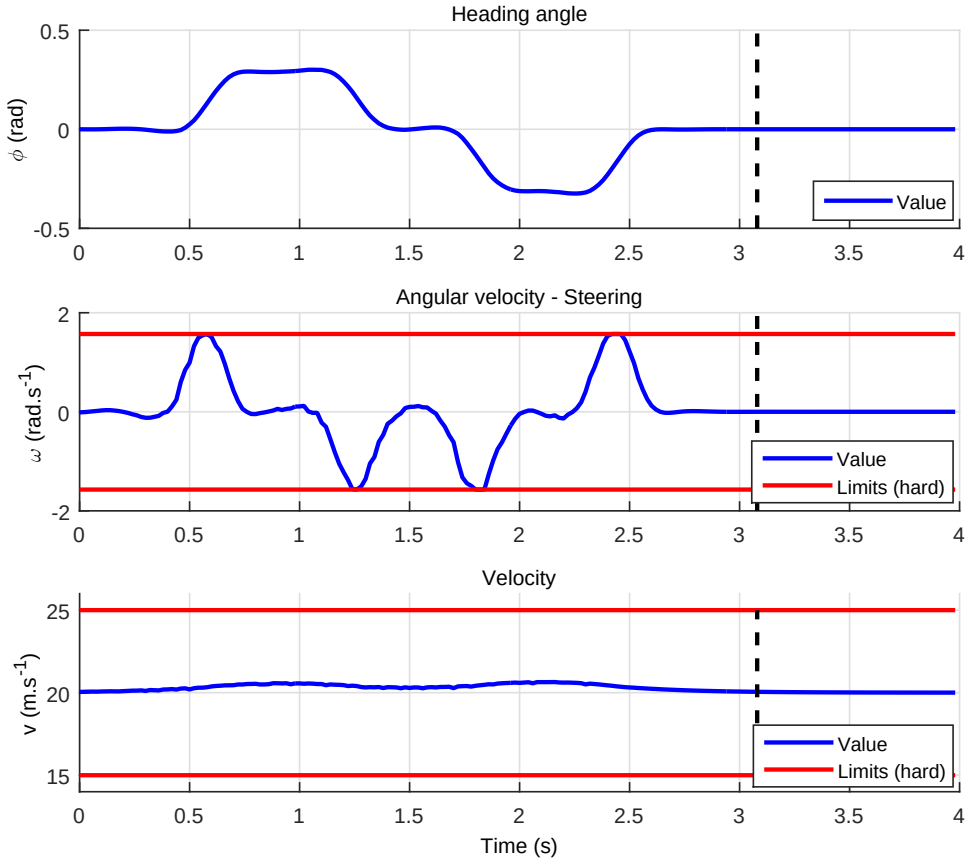


Figure 5.5: Heading angle ϕ and acuators during double lane change maneuver.

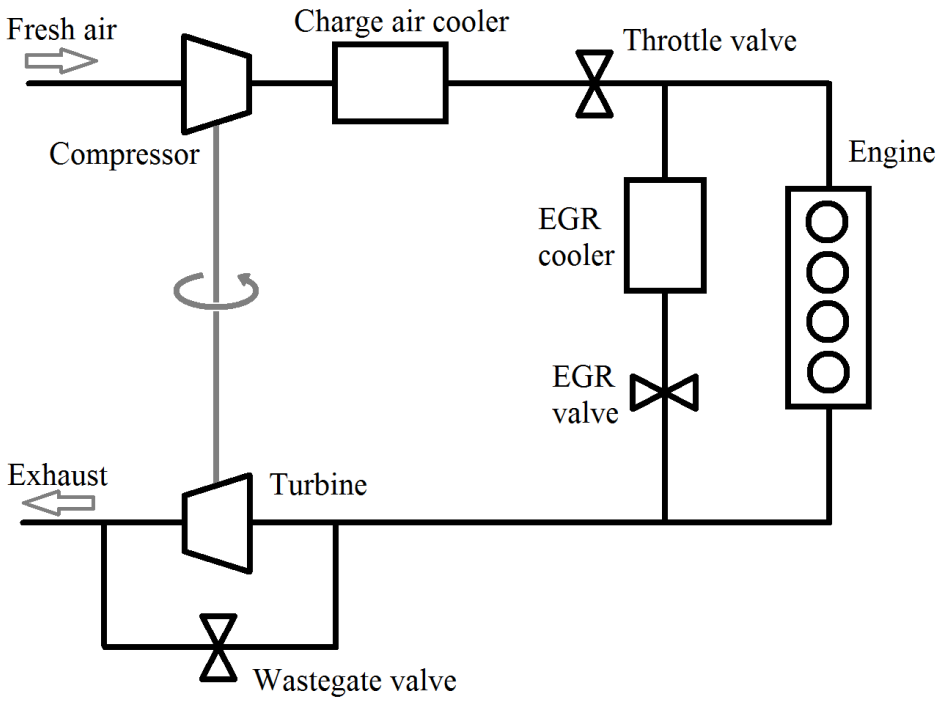


Figure 5.6: Schematic drawing of engine layout.

requirements on fuel economy and emissions are tighter these days, engines become more and more complex. There are various actuators that can be used and there is also a number of controlled quantities. Complicated multivariable interactions are present. The phenomena taking place in the engine are nonlinear and so the use of a nonlinear MPC based control strategy is very attractive.

It this section, we describe control of compression-ignition (diesel) engine air-path with realistic control objectives. Detailed discussion of internal combustion engine technology can be found e.g. in [39]. First, air-path system of a heavy duty diesel engine is described, then a simplified model based on first principle modelling is shown. Finally, control problem is specified and a simulation is carried out.

5.2.1 System Description and Model Derivation

Consider an internal combustion engine depicted in Figure 5.6. The layout given there is quite typical these day for gasoline and diesel engines.

There is a turbocharger and exhaust gas recirculation system (EGR). Downstream the turbo compressor, there is a charge air cooler. Its purpose is to cool down the air that got warm when compressed.

After the charge air cooler, there is a throttle valve. It is used to lower the intake manifold pressure when necessary, for example when the throttle pedal is suddenly released. However, the throttle valve remains wide open most of the time since its closing leads to unwanted pumping losses.

On the EGR path, there is another cooler. The EGR cooler is meant to cool the exhaust gas that flows back to the intake manifold. The purpose of EGR is to increase the heat

capacity of intake air by mixing it with cooled exhaust gas. The heat capacity is important because it helps reduce overall temperature in the cylinders after the fuel mixture is burnt. The temperature decrease helps reduce the amount of nitrogen oxides (NOx) emissions.

On the exhaust piping, there is so called wastegate. Wastegate bypasses the turbine and there is a valve on it which can be used to regulate the flow through the turbine and hence the turbo speed. When open, it can therefore prevent turbo overspeed and help heat-up exhaust gas aftertreatment system.

5.2.2 Control Oriented Model

The physics based models are reasonably precise, they allow parameter identification from measurement data but they are often difficult to simulate. This is due to the fact that the models may be stiff. In some parts of the state space, the model exhibits very fast dynamics compared to the rest of the state space. During the numerical simulation, this forces the ODE solver to reduce the step significantly which slows the simulation down.

In the example, a simplified approximate model will be used. The idea is to replace the fast dynamic parts of the physics based model (pressure and temperature equations) by algebraic equations. These equations can be solved on a grid and then a multivariable polynomial function can be fit to the solution. The same type of model was used for Kalman filter design in [40]. As a result of simplification, the turbo speed n_T is the only differential state of the model and the remaining output variables $\mathbf{y}$ are expressed as a function of n_T and inputs $\mathbf{u}$ only:

$$\begin{aligned} \dot{n}_T &= P(n_T, \mathbf{u}) \,, \\ \mathbf{y} &= Q(n_T, \mathbf{u}) \,. \end{aligned} \tag{5.2}$$

The advantage is that the right hand side of the state equation is a polynomial function P. The output function Q is polynomial as well. They can be evaluated in a very fast manner which is important for online simulation. Furthermore, the Jacobian of these functions is a lower degree polynomial as well and it can be computed analytically. Both of these properties are very favorable to nonlinear predictive control.

Actuators

Actuators that are used to control the system are as follows.

- Wastegate valve - Wastegate valve is located at the bypass of turbine. Its purpose is to control the turbo speed. It can release the pressure from the exhaust manifold and hence make the turbo slow down. (% open)
- EGR valve - EGR valve opens and closes the recirculation channel. This can be used to regulate the amount of exhaust gas entering the intake and to control the pressure difference between the intake and exhaust manifolds. (% open)
- Throttle valve - Throttle valve can restrict the intake of fresh air from the compressor to the engine. If it is closed down, the amount of fresh air entering the intake drops

and so does the intake manifold pressure. If the engine load abruptly drops, the back-pressure drops as well. The compressor pushes more air into the engine while the exhaust gas flow through EGR may stop due to the lack of back-pressure. As a result, EGR ratio would drop. (% open)

External Inputs

The inputs affecting the system are also engine speed given and injection quantity. These are regarded as external inputs because they are not affected by the actuators.

- Engine speed – Number of engine revolutions per minute. (RPM)
- Injection quantity – The amount of diesel fuel injected into the cylinders. It is the variable that has the most significant effect on engine torque and it is used for load control. (mg/stroke)

Controlled Variables

There are basically three distinct requirements in the control of air-path.

- Intake manifold pressure - Sufficient back-pressure enables the EGR to work. If there was no back-pressure, there would be no flow of exhaust gas through EGR system to the intake manifold. On the other hand, high back-pressure increases pumping losses, reducing mechanical efficiency of the engine. (kPa)
- EGR ratio - It is important to keep EGR ratio at certain level, because it helps keep the NOx emission at low levels. If the ratio of exhaust gas to the fresh air was out of prescribed range, the NOx formation during high temperature is much higher. (%)
- Air-fuel equivalence ratio λ - λ is of great importance to the quality of combustion. It describes the richness of fuel-air mixture and it is expressed using stoichiometric value of fuel. If lambda falls below 1, the mixture is too rich and the fuel combustion is imperfect. As a result, a part of fuel would leave the engine unburnt, negatively influencing fuel efficiency.

  In diesel engines, the mixture is generally lean. Unlike gasoline engines, the lambda need not be held close to stoichiometric value of the fuel. Instead, diesel engine work in a wide range of lambda values. Anyway, a minimum limit on lambda is usually imposed to prevent smoke formation for fuel mixture too rich. (unitless)
- Turbo speed - Turbocharger is highly stressed component of modern engine and it has to be protected against overspeed. Mechanical damage and even destruction of turbocharger can occur if the turbine spins too fast. It is therefore common practice to restrict the angular velocity of turbine rotation by means of engine control strategy. Either a wastegate valve or variable geometry turbine is used to control the speed of rotation. (kRPM)

5.2.3 Experiment

The control strategy is evaluated using two maneuvers simulating sudden change of engine load from almost idle to full power and back from full power to idle.

- Tip-in maneuver - With engaged gear, the vehicle accelerates from engine speed of 600 to 2 000 RPM during 10 seconds. Injection quantity is very high due to high torque demand during acceleration.

- Deceleration maneuver - Engine running under full load with 2 000 RPM slows down to 1 000 RPM during 15 seconds after the engine load suddenly drops and the injection quantity is decreased.

In both cases, fuel injection quantity and engine speed are treated as external inputs. They are not known to the controller over the whole prediction horizon. Instead, they are approximated by constant trajectory during the prediction horizon.

Output reference trajectories for intake manifold pressure and EGR ratio are assumed constant over the prediction horizon as well. They are obtained using interpolation in steady state data tables for the specific engine speed and injection quantity trajectory. The goal is to track these reference trajectories.

There are also the above mentioned limits. They can be treated as soft output constraints in NMPC Framework. Soft minimum limit on λ is imposed to the value of 1.2. This is to prevent excessive smoke with too much fuel. Turbo speed maximum limit is imposed to prevent turbo overspeeding. The limit value is set to 102 000 RPM.

As there are three actuators in the control problem and only two output reference tracking variables, we add input reference tracking for the actuators. The references are set to preferred actuator positions. The throttle should be kept wide open and the wastegate valve be closed. This makes the problem well posed by removing extra actuator degrees of freedom. Note that weighting of the wastegate reference tracking is very small as it may compromise offset-free tracking of output references.

5.2.4 Framework Configuration

Model variables are defined in the order given in Table 5.1. Manipulated variables and known disturbances (external inputs) are distinguished using index vectors with their respective order. Tracked or constrained variables are selected using the order of variables as well. Due to numerical properties of the model, some variables are represented in a modified way. For example, instead of λ, scaled reciprocal $100/\lambda$ is used in the model. Similarly, EGR ratio is scaled to its value given as a percentage, 100·EGR ratio. Turbo speed is represented in a thousands of RPM order. On the other hand, the three manipulated variables are scaled to 0–1 range.

The NMPC controller is designed in the following way. Prediction and control horizon are set to 2 seconds. This is reasonably long considering the dynamics of the engine. Controller discretization period is set to 0.1 s. This corresponds to 20 samples. The Jacobians of model equations are given analytically using partial derivatives of the multivariable

Table 5.1: Model variables and Matlab vector order.

	Name	Matlab code
States	0.001·Turbo speed	`x(1)`
Inputs	EGR valve	`u(1)`
	Wastegate	`u(2)`
	Throttle	`u(3)`
	Start of injection	`u(4)`
	RPM	`u(5)`
	Injection quantity	`u(6)`
Outputs	0.001·Turbo speed	`y(1)`
	$100/\lambda$	`y(2)`
	Intake manifold pressure	`y(3)`
	100·EGR ratio	`y(4)`

polynomial functions P and Q. Manipulated variables and known disturbances are defined using the order of the variables in the model by `mv = [1;2;3]` and `dv = [4;5;6]`.

Cost function is selected so that the control objectives specified above are met. Control increment penalization (3.5) is used for the manipulated variables. Note that this penalty matrix should be positive definite so that the optimization problem has unique minimizer.

```
R_du = kron(eye(np),diag([1,1,10]));
```

Output reference tracking is active for intake manifold pressure and EGR ratio. Therefore, model outputs number 3 and 4 are selected. Corresponding cost function term is (3.2) with equal weighting for both outputs. Actual reference signals are passed to the framework just during the simulation.

```
y_tr = [3;4];
Q_r = kron(eye(np),diag([10,10]));
```

Input reference tracking (3.6) is turned on for wastegate and throttle inputs with index 2 and 3 respectively. There is higher penalization for throttle leaving wide open position. The reference positions are set to 1 % for wastegate and 99 % for throttle and they are passed to the framework during the simulation.

```
u_tr = [2;3];
R_r = kron(eye(np),diag([1,100]));
```

Upper and lower limits for all manipulated variables are set to `u_lb = [0.01;0.01;0.01]` and `u_ub = [0.99;0.99;0.99]`. Note that the manipulated variables are scaled to 0–1 range in the model representation. Zero and 100 % values are avoided to prevent possible numerical issues.

Soft output constraints are defined for λ and turbo speed. Maximum limit is defined for turbo speed to 102 kRPM. Minimum limit for λ at 1.2 translates to maximum limit for $100/\lambda$ at a value of $100/1.2$. Penalty weighting matrix for the soft limits are set with more emphasis on $100/\lambda$.

```
y_max = [1;2];
y_max_lim = [102;100/1.2];
G_max = diag([10,1000]);
```

The sensitivity matrix computation method is set to LTV approximation (see Section 2.5.2), i.e. the model is linearized and discretized at each time step of the prediction horizon. The SQP is set up to run five iterations everytime and no premature stopping based on update of the input trajectory is done. By inspection of the SQP iterates and the final quality of control, it was concluded that these setting are sufficient.

5.2.5 Simulation Results

The results of closed loop simulation with NMPC controller designed above are shown in Figures 5.7–5.10. There are several interesting moments present in the simulation. The observations are described below.

Tip-in Maneuver

Simulation results for tip-in maneuver are plotted in Figures 5.7 and 5.8. The former one shows controlled variables. Green line designates reference trajectories. Red line shows maximum and minimum limits.

EGR actuator works well in adjusting the EGR ratio to the desired value. The only major flaw in tracking is between 5 and 10 second. This is due to the tuning of the controller. The intake manifold pressure has higher priority and so the EGR valve is closed in order to spin the turbo faster. If on the other hand the EGR valve was open more, there would be less exhaust gas flowing to the turbine, the intake manifold pressure would take longer to reach the desired value.

Wastegate is fully closed between 5 and 11 seconds. This is to maximize the amount of energy spinning the turbo. It is then slightly open from 11 to 14 seconds. This stops the intake manifold pressure from rising above the reference value.

The use of throttle to control intake manifold pressure is almost avoided. This is mainly due to the high penalty for leaving fully open position. If the penalty was lower, the controller would use both wastegate opening and throttle closing to relieve the intake manifold pressure after 11 seconds.

λ violates the soft minimum limit between 6 and 8 seconds. This cannot be avoided with current control configuration. As fuel injection quantity is fixed, the only way to increase λ is via increased intake air flow. For given values, no more airflow can be achieved as all the actuators are already on the limits providing maximum airflow possible. This may result in excessive smoke or even misfires and so actual engines have a device to limit injection quantity during such transient conditions based on lambda value. The injection quantity is eventually cut down so that λ is kept above the limit even though the drivability suffers.

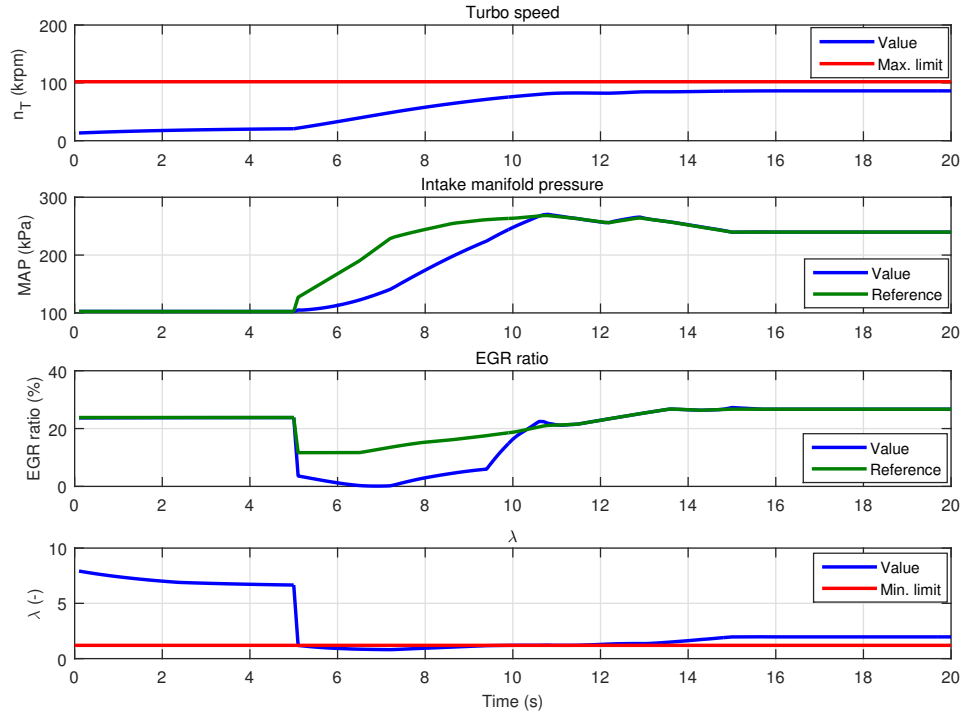


Figure 5.7: Controlled variable trajectories during the tip-in experiment.

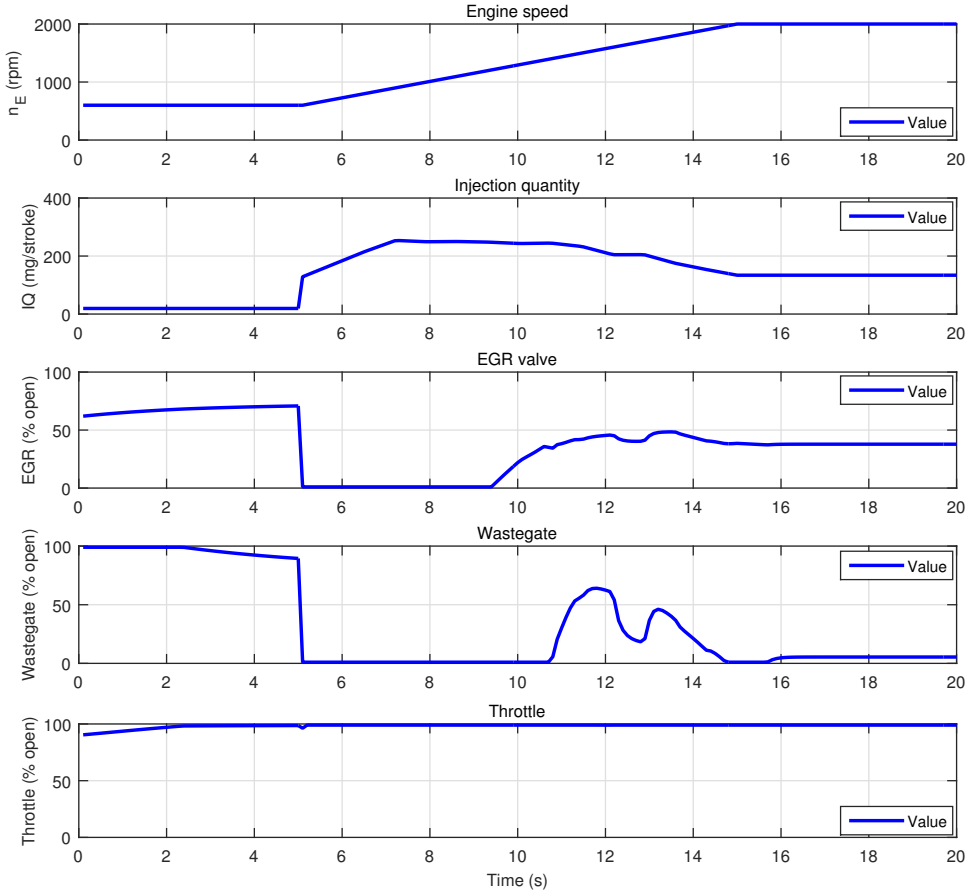


Figure 5.8: Input variable trajectories during the tip-in experiment.

Deceleration Maneuver

During the deceleration maneuver, the injection quantity is cut down at 5 seconds. The engine power consequently drops and the vehicle starts to coast-down during the next 15 seconds. Simulation results are plotted in Figures 5.9 and 5.10.

Intake manifold pressure tracks the decreasing reference signal almost perfectly. As the transition is relatively slow, actuators are almost steady until 14 seconds.

EGR ratio is controlled mainly by EGR valve opening. Around 14 seconds, wastegate is partially opened a throttle is slightly closed. This helps keep the pressure difference between intake and exhaust manifold high enough to provide required EGR flow.

The minimum limit for λ is not violated during the simulation. Up to 5 seconds it is almost constant due to fixed injection quantity and steady intake manifold pressure. After the 5 seconds it suddenly rises with the decrease in injection quantity. Afterwards it steadily decreases with decreasing intake manifold pressure and constant injection quantity of 50 mg/stroke.

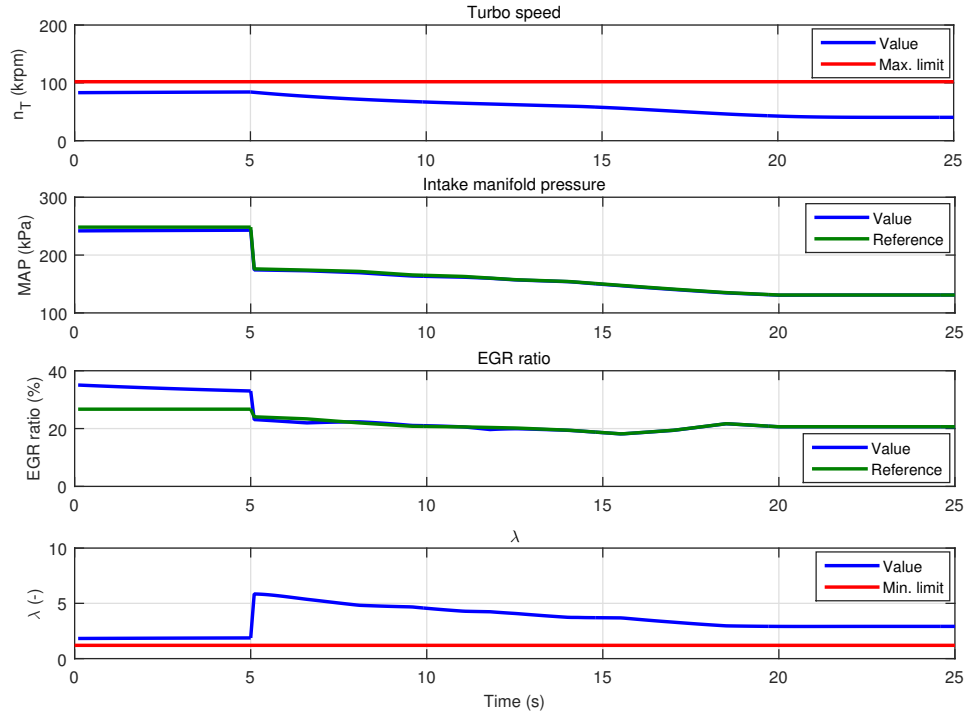


Figure 5.9: Controlled variable trajectories during the deceleration experiment.

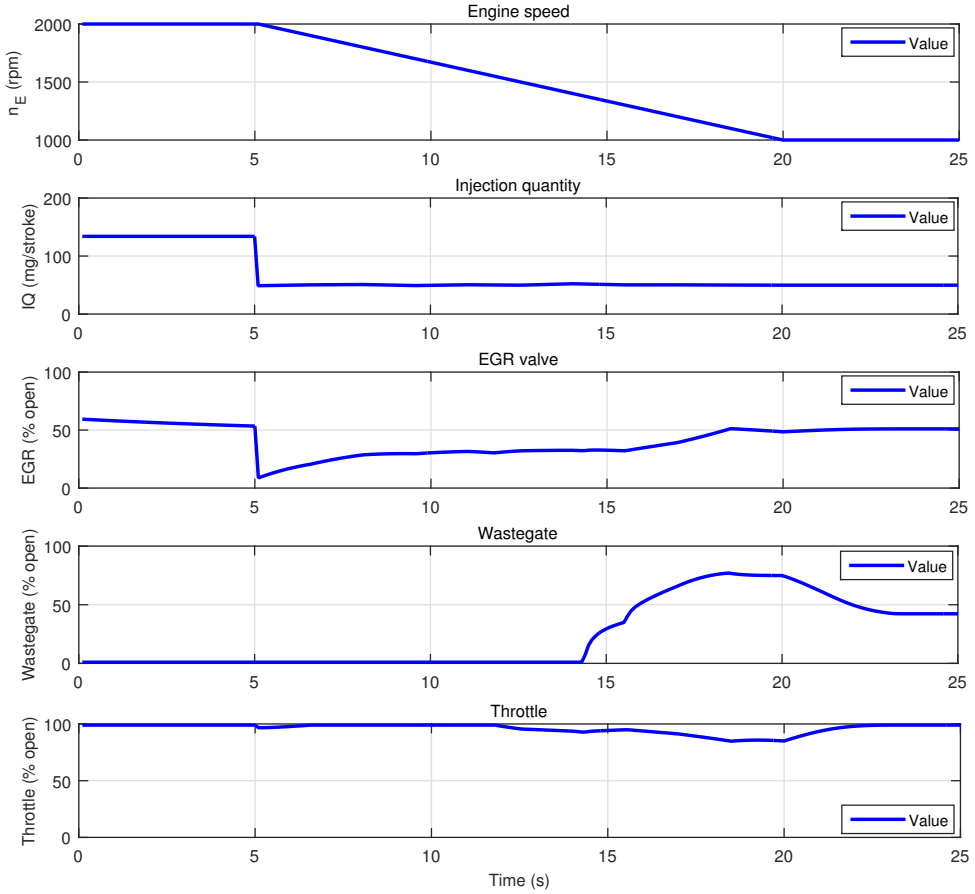


Figure 5.10: Input variable trajectories during the deceleration experiment.

Chapter 6

Conclusion

We presented an implementation of modular computer framework for the design and testing of nonlinear model based predictive controllers. In the first three chapters of the thesis a theoretical overview of existing approaches to nonlinear MPC was provided. Then, single shooting discretization method together with sequential quadratic programming was implemented as a Matlab package. The code is separated into logical blocks and the workflow respects the process that is commonly used in controller design. The framework alongside with extensive documentation fits the intended purpose very well.

This was demonstrated in two application examples. The first example demonstrated the use of nonlinear MPC for planar vehicle steering control. More specifically, tracking of a predefined trajectory given by Cartesian coordinates was outlined.

The second example is an application of nonlinear MPC to the problem of diesel engine air path control. Typical control problem setup was selected and the goals described. Then, a NMPC controller was designed using the framework. It was shown, that if a nonlinear model of the process is available, the design and implementation of nonlinear MPC using the framework is a matter of sensible control problem specification.

Apart from the presented examples, several projects in the area of automotive controls were successfully solved. Simulation based evaluation of NMPC and comparison with different approaches was carried out. The projects include for example thermal management of a combustion engine or an advanced cruise control. The framework was also used to generate QP instances for a performance comparison of several QP solvers. This proved the framework to be working and meaningful.

6.1 Future Work

During the use of the implemented framework, several improvements and extensions were proposed. Different handling of soft constraints, time varying limits or more flexible settings of cost function are on schedule. Support of a proprietary QP solver or implementation of selected program parts in C language are also considered. Thanks to the modularity of the framework, all the above mentioned changes can be implemented easily.

A major extension, the use of Simulink models was suggested as well. Simulink provides a mean of evaluating the continuous time model differential equation as well as the

output function. Using this interface, Simulink models could be used in the implemented framework directly in the controller design. This feature would significantly broaden the scope of problems that could be solved using the framework. Advanced nonlinear models are often prepared in Simulink environment or other domain specific tools. Having them supported would be a great benefit.

Bibliography

[1] J. A. Rossiter, *Model-based predicitve control: a practical approach.* CRC Press LLC, 2003.

[2] F. Allgöwer and A. Zheng, *Nonlinear Model Predictive Control.* Birkhäuser, first ed., 2000.

[3] D. Mayne and H. Michalska, "Receding horizon control of nonlinear systems," *IEEE Transactions on Automatic Control*, vol. 35, no. 7, pp. 814 – 824, 1990.

[4] F. Allgöwer, R. Findeisen, and Z. K. Nagy, "Nonlinear model predictive control: From theory to application," *Journal of the Chinese Institute of Chemical Engineers*, vol. 35, no. 3, pp. 299–315, 2004.

[5] S. J. Qin and T. Badgwell, "An overview of nonlinear model predictive control applications," *Nonlinear model predictive control*, vol. 26, pp. 369–392, 2000.

[6] M. Kano and M. Ogawa, "The state of the art in advanced chemical process control in Japan," *IFAC Proceedings Volumes (IFAC-PapersOnline)*, vol. 7, no. PART 1, pp. 10–25, 2009.

[7] W. B. Bequette, "Nonlinear control of chemical processes: A review," *Industrial & Engineering Chemistry Research*, vol. 30, pp. 1391–1413, July 1991.

[8] H. Chen, A. Kremling, and F. Allgöwer, "Nonlinear Predictive Control of a Benchmark CSTR.," in *Proc. 3rd European Control Conference ECC'95*, (Rome, Italy), pp. 3247–3252, January 1995.

[9] A. S. Brásio, A. Romanenko, J. Leal, L. O. Santos, and N. C. Fernandes, "Nonlinear model predictive control of biodiesel production via transesterification of used vegetable oils," *Journal of Process Control*, vol. 23, pp. 1471–1479, nov 2013.

[10] S. M. Erlien, J. Funke, and J. C. Gerdes, "Incorporating non-linear tire dynamics into a convex approach to shared steering control," *Proceedings of the American Control Conference*, pp. 3468–3473, 2014.

[11] P. Falcone, M. Tufo, F. Borrelli, J. Asgari, and H. E. Tseng, "A Linear Time Varying Model Predictive Control Approach to the Integrated Vehicle Dynamics Control Problem in Autonomous Systems," in *Proceedings of the 46th IEEE Conference on Decision and Control*, (New Orleans), pp. 2980–2985, 2007.

[12] L. Grüne and J. Pannek, *Nonlinear Model Predictive Control: Theory and Algorithms.* Communications and Control Engineering, Springer, 2011.

[13] C. Hoffmann, C. Kirches, A. Potschka, S. Sager, L. Wirsching, M. Diehl, D. B. Leineweber, and A. A. S. Schäfer, "MUSCOD II User Manual," 2011. Release 6.0.

[14] B. Houska, H. J. Ferreau, M. Vukov, and R. Quirynen.

[15] R. Amrit and J. B. Rawlings, "Nonlinear Model Predictive Control Tools (NMPC Tools)," pp. 1–12, 2008.

[16] E. Harati, *Nonlinear Model Predictive Controller Toolbox.* Master thesis, Chalmers University of Technology, 2011.

[17] H. J. Ferreau, P. Ortner, P. Langthaler, L. D. Re, and M. Diehl, "Predictive control of a real-world Diesel engine using an extended online active set strategy," *Annual Reviews in Control*, vol. 31, pp. 293–301, Jan. 2007.

[18] Y. Wang and S. Boyd, "Fast Model Predictive Control Using Online Optimization," *IEEE Transactions on Control Systems Technology*, vol. 18, pp. 267–278, Mar. 2010.

[19] M. Diehl, *Real-Time Optimization for Large Scale Nonlinear Processes.* PhD thesis, Ruprecht-Karls-Universität Heidelberg, 2001.

[20] H. G. Bock, M. Diehl, C. Kirches, K. Mombaur, and S. Sager, "Optimierung bei gewöhnlichen Differentialgleichungen," 2014.

[21] S. Boyd and L. Vandenberghe, *Convex Optimization.* Cambridge University Press, seventh ed., 2004.

[22] A. G. Wills and W. P. Heath, "Interior point algorithms for nonlinear model predictive control," in *Assessment and Future Directions of Nonlinear Model Predictive Control* (R. Findeisen, F. Allgöwer, and L. T. Biegler, eds.), (New York), pp. 207–216, Springer, 2007.

[23] J. Nocedal and S. J. Wright, *Numerical Optimization.* New York: Springer, second ed., 2006.

[24] D. P. Bertsekas, *Convex Optimization Algorithms.* Nashua: Athena Scientific, first ed., 2015.

[25] M. Y. El Ghoumari, H. J. Tantau, and J. Serrano, "Non-linear constrained MPC: Real-time implementation of greenhouse air temperature control," *Computers and Electronics in Agriculture*, vol. 49, pp. 345–356, 2005.

[26] K. Ogata, *Modern control engineering.* New Jersey: Prentice-Hall, fourth ed., 2002.

[27] J. Pekař, *Robust Model Predictive Control.* PhD thesis, Czech Technical University, 2005.

[28] R. Serban and A. C. Hindmarsh, "CVODES: the Sensitivity-Enabled ODE Solver in SUNDIALS," *ACM Transactions on Mathematical Software*, vol. 5, no. September, pp. 1–18, 2003.

[29] J. J. Moré and G. Toraldo, "Algorithms for bound constrained quadratic programming problems," *Numer. Math.*, vol. 55, pp. 377–400, July 1989.

[30] R. Cagienard, P. Grieder, E. Kerrigan, and M. Morari, "Move blocking strategies in receding horizon control," *2004 43rd IEEE Conference on Decision and Control (CDC) (IEEE Cat. No.04CH37601)*, vol. 0, no. x, pp. 2023–2028 Vol.2, 2004.

[31] V. Havlena and J. Štecha, *Moderní teorie řízení*. 1999.

[32] E. A. Wan and R. Van Der Merwe, "The Unscented Kalman Filter for Nonlinear Estimation," in *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*, pp. 153–158, 2002.

[33] U. Ozguner, T. Acarman, and K. Redmill, *Autonomous Ground Vehicles*. Boston: Artech House, 2011.

[34] H. Ferreau, C. Kirches, A. Potschka, H. Bock, and M. Diehl, "qpOASES: A parametric active-set algorithm for quadratic programming," *Mathematical Programming Computation*, vol. 6, no. 4, pp. 327–363, 2014.

[35] Mathworks, *Display Custom Documentation*. The Mathworks, Inc., Natick, Massachusetts USA, 12 2015. Retrieved on 2015-11-29.

[36] ISO, "Passenger cars—Test track for a severe lane-change manoeuvre—Part 1: Double lane-change," ISO 3888-1:1999, International Organization for Standardization, Geneva, Switzerland, 1999.

[37] VEHICO, *ISO Double Lane Change Test*. VEHICO, GmbH., Braunschweig, Germany, 10 2015. Retrieved on 2015-10-26.

[38] D. Cieslar, P. Dickinson, K. Glover, and N. Collings, "Closed-loop control of WAVE models for assessing boost assist options," 2014.

[39] J. B. Heywood, *Internal Combustion Engine Fundementals*. McGraw-Hill series in mechanical engineering, McGraw-Hill, 1988.

[40] D. Pachner and J. Beran, "Comparison of Sensor Sets for Real-Time EGR Flow Estimation," in *Paper to be presented at SAE 2016 World Congress & Exhibition. Society of Automotive Engineers.*, (Detroit, Michigan, USA), April 2016.

Appendix A

NMPC Framework Classes

A.1 Model Class

Properties

Name	Explanation	Type
`n`	Number of model states	integer
`m`	Number of model inputs	integer
`p`	Number of model outputs	integer
`Ts`	Sampling time used during model discretization (in seconds)	float
`mv`	Index vector of model inputs treated as manipulated variables	column vector
`dv`	Index vector of model inputs treated as external inputs or known disturbance	column vector
`f`	Model state ODE right hand side (2.2)	function
`dfdx`	Jacobian of `f` w.r.t. state vector	function
`dfdu`	Jacobian of `f` w.r.t. input vector	function
`g`	Model output equation right hand side (2.3)	function
`dgdx`	Jacobian of output function `g` w.r.t. state vector	function
`dgdu`	Jacobian of output function `g` w.r.t input vector	function

Methods

Name	Explanation
`model`	Constructor
`verify`	Formal verification of `model` consistency
`fd_jacobian`	Helper function to calculate Jacobian matrix of `f` and/or `g` using forward finite difference scheme

A.2 MPC Setup Class

Properties

Name	Explanation	Type
`np`	Prediction horizon as a number of sampling times	integer
`nc`	Control horizon as a number of blocks	integer
`blocks`	Specification of control moves distribution. Each entry specifies the length of a control block. The number of entries must equal `nc`. Default: [1,1,1,...,`np-nc`+1]	column vector
`u_lb`	Manipulated variables lower bounds	column vector
`u_ub`	Manipulated variables upper bounds	column vector
`R`	Control magnitude penalization matrix (3.3)	matrix
`R_du`	Control increment penalization matrix (3.5)	matrix
`y_tr`	Index vector of outputs intended for reference tracking. Output reference trajectory is passed as an input argument of `nmpc.control` function.	column vector
`Q_r`	Output reference tracking penalization matrix (3.2)	matrix
`u_tr`	Index vector of inputs intended for reference tracking. Only inputs defined as manipulated variables (`model.mv`) can be used. Input reference trajectory is passed as an input argument of `nmpc.control` function.	column vector
`R_r`	Input reference tracking penalization matrix (3.6)	matrix
`preview`	Use of reference preview information. 1 (default): ON, 0: OFF	boolean
`y_min`	Index vector of output soft minimum limits	column vector
`y_min_lim`	Output soft minimium limit values	column vector
`G_min`	Output soft minimum limit penalization matrix	matrix
`y_max`	Index vector of output soft maximum limits	column vector
`y_max_lim`	Output soft maximum limit values	column vector
`G_max`	Output soft maximum limit penalization matrix	matrix
`sens_type`	Sensitivity calculation method. Continuous time models: `lti_mpc` (Section 2.5.1), `ltv_mpc` (Section 2.5.2), `analytic` (Section 2.5.3); Discrete time models: `lti_discrete`, `ltv_discrete`	string
`sqp_max_iter`	Maximum number of SQP iterations. (default: 5) (Section 4.7.5)	integer
`sqp_u_update`	Threshold on u update to stop SQP. (default: 0) (see (4.2) in Section 4.7.5)	float
`qp_solver`	QP solver selection: `quadprog`, `qpoases`	string
`verbose`	Amount of information printed to command window during SQP iterations. 2 (default): Detailed iteration info, 1: Short summary, 0: Quiet	integer

Methods

Name	Explanation
`mpc_setup`	Class constructor
`verify`	Formal verification of `mpc_setup` consistency

A.3 NMPC Class

Properties

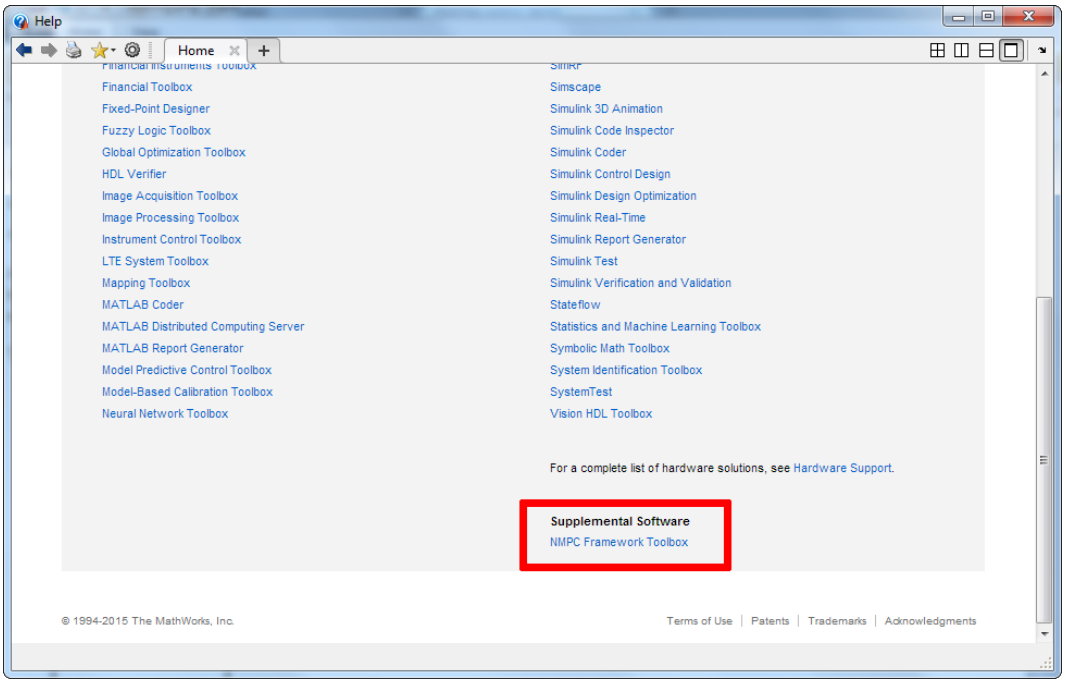
Name	Explanation	Type
`model`	Model data - `model` class instance	`model`
`mpc`	MPC algorithm settings - `mpc_setup` class instance	`mpc_setup`
`M`	Move blocking matrix	matrix

Methods

Name	Explanation
`build_prediction`	Creates prediction matrices for the case of LTI or LTV sensitivity approximation (Sections 2.4, 2.5)
`control`	Main computation method
`move_blocking`	Move blocking matrix construction
`nmpc`	Class constructor
`optimizer`	Wrapper function for QP solver interfacing
`qp_add_constraints`	Adds hard constraints to the problem (Section 3.2.1)
`qp_build`	Builds quadratic cost function based on `mpc_setup` (Section 3.1)
`qp_soft_limits`	Adds soft constraints to the problem (Section 3.2.2)
`sensitivity`	Wrapper function for sensitivity matrix calculation methods (Section 2.5)
`sensitivity_analytic`	Analytical sensitivity computation function
`sensitivity_approx`	Approximate sensitivity computation function (LTI and LTV)
`sensitivity_discrete`	Sensitivity computation function for discrete time models (LTI and LTV)
`simulator`	Wrapper function for continuous time model simulator routines
`simulator_discrete`	Wrapper function for discrete time model simulator

Appendix B

Matlab Documentation Screenshots



x

Figure B.1: Main screen of Matlab help browser showing integration of NMPC Framework documentation (highlighted red).

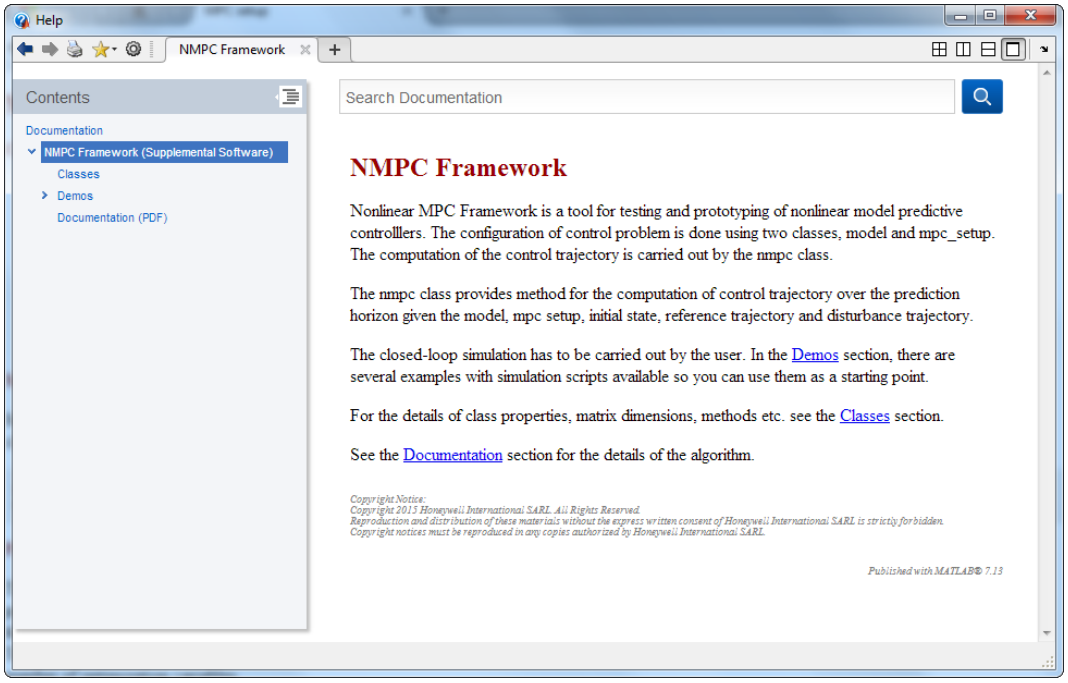


Figure B.2: NMPC Framework documentation - Start screen.

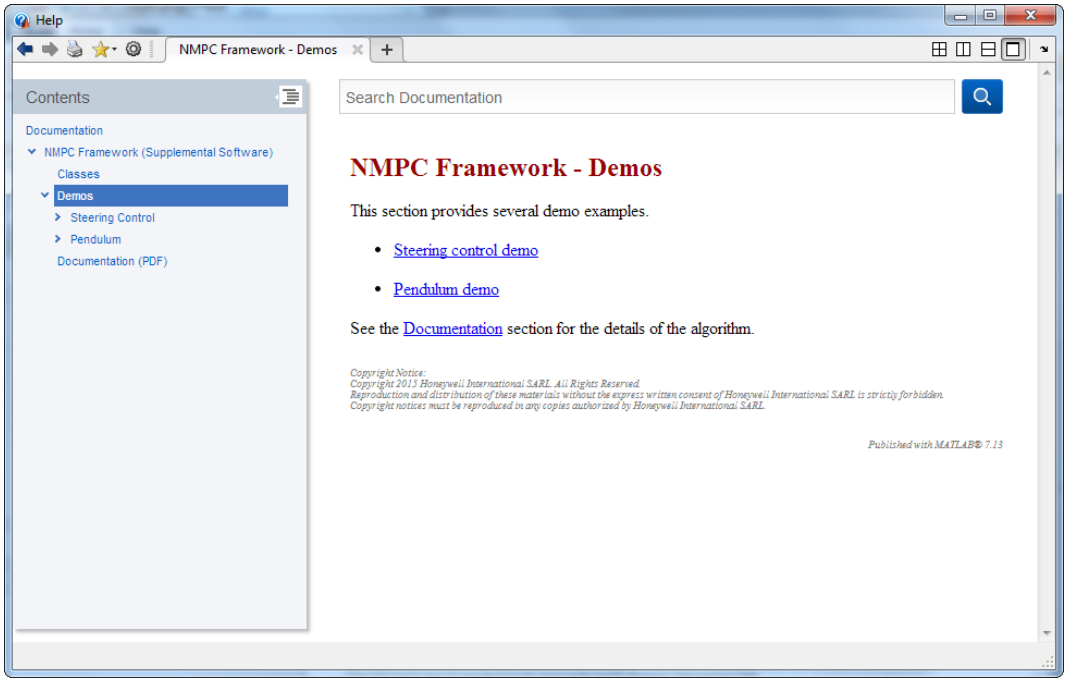


Figure B.3: NMPC Framework documentation - Demos section.

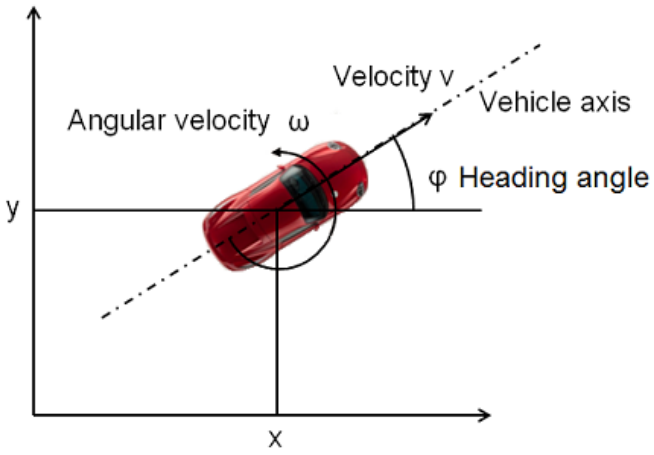
Steering control demo

This demo example shows the configuration of planar vehicle lateral dynamics control. Very simple model of the form

```
dx/dt = v*cos(phi)
dy/dt = v*sin(phi)
dphi/dt = omega
```

is used. `x` and `y` are plane coordinates of vehicle center. `phi` is a heading angle with respect to x axis. Variables `omega` and `v` are manipulated variables. The objective is to track `x` and `y` references and stay in prescribed `y` range.

The model variables are ilustrated in the following figure:



- Model definition
- MPC settings
- Simulation script

Published with MATLAB® 7.13

Figure B.4: Planar vehicle steering control demo example - Start screen.

Model definition script

Contents

Description

This script contains definition of dynamic model. It is in the state space form given by the following equations

```
dx/dt = f(x,u);
y = g(x,u);
```

f is a vector valued function of x and u R^(n+m) -> R^n g is an output mapping from x and u to outputs R^(n+m) -> R^p

It is also neccesary to specify derivatives of f and g

Model instance

```
model = nmpc.model(3,2,3); % Model dimensions (no. of states, inputs, outputs)
```

State equation

State equation right hand side and its Jacobians with respect to state x and input u.

```
model.f = @(x,u) [  u(2)*cos(x(3)); % x dot
                    u(2)*sin(x(3)); % y dot
                    u(1)];          % phi dot
model.dfdx = @(x,u) [   0, 0, -u(2)*sin(x(3));
                        0, 0, u(2)*cos(x(3));
                        0, 0, 0];
model.dfdu = @(x,u) [   0, cos(x(3));
                        0, sin(x(3));
                        1, 0];
```

Output function

Output function right hand side and its Jacobians with respect to state x and input u.

```
model.g = @(x,u) [  x(1);   % x
                    x(2);   % y
                    x(3)];  % phi
model.dgdx = @(x,u) eye(3);
model.dgdu = @(x,u) zeros(3,2);
```

Manipulated variables and disturbance inputs

This section specifies the indices of inputs that are treated as manipulated variables (mv) and disturbance variables (dv).

Figure B.5: Planar vehicle steering control demo example - Model settings.

MPC setup

Contents

MPC configuration and description

MPC definition is passed to the controller using the mpc_setup class instance. This script demonstrates the structure of the control problem settings and the corresponding properties of mpc_setup class that are necessary to correctly define the problem.

We start by running steering_model script and creating an empty instance of mpc_setup class.

```
steering_model
mpc = nmpc.mpc_setup();
```

Prediction and control horizon settings

The following options set up the number of sampling periods (see the nmpc.model.Ts property) used in the controller. The prediction horizon defines the length of the window that the controller 'sees' in the future. The control horizon is the number of control moves. Together with the number of manipulated variables (nmpc.model.mv), it determines the number of optimization variables.

The control horizon has to be less than or equal to the prediction horizon. Manipulated variables are kept constant beyond the control horizon. If you want to specify different 'blocking' strategy, use the nmpc.mpc_setup.blocks property.

```
mpc.np = 30; % Prediction horizon - number of steps
mpc.blocks = [ones(1,6),4*ones(1,6)]; % MV block lengths
mpc.nc = numel(mpc.blocks); % Control horizon - number of blocks
```

Control constraints

Manipulated variable upper and lower bound. The bounds are constant over the prediction horizon. There is one value for each manipulated variable defined in (nmpc.model.mv) property. The values are organized in a column vector.

In this example, the bounds correspond to max and min rate fo change of heading angle and forward velocity respectively.

Figure B.6: Planar vehicle steering control demo example - MPC settings.

Simulation script

Contents

NMPC + model setup

This script demonstrates the use of the nmpc.nmpc class to simulate the closed loop behavior with the NMPC controller. Note that the simulation interface is the `nmpc.control` method. This method calculates one step of predictive control given the initial state, reference trajectories and possibly disturbances.

We start by running the initialization scripts for `model` and `mpc_setup` classes and creating an instance of nmpc class.

```
steering_model
steering_mpc_setup

my_nmpc = nmpc.nmpc(model,mpc);
```

Reference trajectory

Reference x,y trajectory is prepared for the whole simulation time. Velocity reference trajectory is prepared as well. For each simulation step (sampling period), one value is provided in the column vector.

```
x = [linspace(0,2*6+2*13.5+11+50,500)];
y = [1*ones(1,30),linspace(1,6,70),6*ones(1,50),linspace(6,1,70),1*ones(1,250+
t = linspace(0,5,500);

t_ref = 0:model.Ts:max(t);
x_ref = interp1(t,x,t_ref);
y_ref = interp1(t,y,t_ref);

YREF = [x_ref;y_ref]; % Output reference trajectory

UREF = 25*ones(1,numel(t_ref)); % Input reference trajectory
```

Simulation setup

The lines of code in this section declare auxiliary simulation variables. These are used to store the simulation results, define the initial condition, stop the simulation etc.

```
x0 = [0;1;0];   % Initial state (x,y,phi)
U = [zeros(1,mpc.np);20*ones(1,mpc.np,1)];

plot_pred = 1;
plot_final = 1;

T = 130; % Simulation end sample
YSIM = zeros(3,T);
USIM = zeros(2,T);
```

Figure B.7: Planar vehicle steering control demo example - Simulation script.