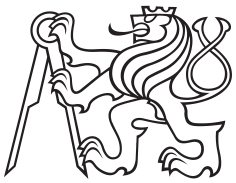
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Using an embedded QP solver for automotive applications

Tomáš Rutrle

Supervisor: doc. Ing. Tomáš Haniš, Ph.D.
May 2022

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Rutrle Tomáš** Personal ID number: **478067**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

Branch of study: **Cybernetics and Robotics**

II. Master’s thesis details

Master’s thesis title in English:

Using an embedded QP solver for automotive applications

Master’s thesis title in Czech:

Využití vestavěného QP solveru pro automotive aplikace

Guidelines:

The goal of the thesis is to develop and implement a Simulink based framework, which will allow the user to solve quadratic problems on embedded hardware using automotive grade bus like CAN and CAN FD. The thesis will address following points:
1. Familiarization with the topic of quadratic programming, automotive grade communication and embedded hardware
2. Analysis of representative automotive grade communication solutions
3. Adaptation of an existing QP solver and its deployment
4. Realization of baseline and improved communication logic and verification of the workings of the embedded QP solver
5. Utilization of the embedded QP solver in the MPC domain, analysis and presentation of acquired results

Bibliography / sources:

[1] J. B. Rawlings, D. Q. Mayne, M. M. Diehl. Model predictive control: theory, computation and design, 2nd ed., Nob Hill Pub, 2019
[2] F. Borrelli, A. Bemporad, and M. Morari, Predictive Control for Linear and Hybrid Systems, 1 edition. Cambridge, New York: Cambridge University Press, 2017
[3] R. De Andrade, K. N. Hodel, J. F. Justo, A. M. Laganá, M. M. Santos and Z. Gu, "Analytical and Experimental Performance Evaluations of CAN-FD Bus," in IEEE Access, vol. 6, pp. 21287-21295, 2018, doi: 10.1109/ACCESS.2018.2826522.
[4] J. Nocedal and S. Wright, Numerical Optimization, 2nd edition. New York: Springer, 2006

Name and workplace of master’s thesis supervisor:

doc. Ing. Tomáš Haniš, Ph.D. Department of Control Engineering FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **28.01.2022** Deadline for master's thesis submission: **20.05.2022**

Assignment valid until:
by the end of summer semester 2022/2023

doc. Ing. Tomáš Haniš, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

I want to express my gratitude towards my family for enabling and supporting me throughout the course of my studies, for which I will always be grateful.

Furthermore, another thank you belongs to my supervisor, friends, and colleagues for the guidance regarding this thesis.

Declaration

I, Tomáš Rutrle, hereby declare that this thesis is a product of my own work and that to the best of my knowledge all information sources have been listed in accordance with the methodical instructions for observing the ethical principles in the preparation of a university thesis.

Prague, May 2022 .....

Abstract

The main focus of this thesis is on the development of a framework for an external quadratic program solver utilizing the CAN and CAN FD communication buses. To begin with, the deployment of two quadratic solvers onto a controller is presented. Further, an embedded application is developed to handle bus communication, memory management, and solver calling. To utilize the external solver, Matlab & Simulink based CAN and CAN FD interfaces are implemented to allow for real-time simulation in the hardware-in-the-loop configuration. Both the embedded application and the interfaces are tested for functionality and performance with a focus on the time it takes the framework to solve quadratic problems of various sizes. Finally, two automotive-related model predictive control demos are presented, employing the external solver interface and showcasing its capabilities.

Keywords: Quadratic programming, QP, Embedded solver, CAN, CAN FD, Optimal control, MPC, External solver, Vehicle control

Supervisor: doc. Ing. Tomáš Haniš, Ph.D.
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Control Engineering

Abstrakt

Hlavním cílem této práce je vývoj rámce pro externí solver kvadratických problémů využívající CAN a CAN FD sběrnice. V první řadě je představen mikrokontroler, na který jsou nasazeny dva různé solvery. Dále je pro kontroler vyvinuta vestavěná aplikace, která obstarává komunikaci přes sběrnici, správu paměti a obsluhu solverů. Na druhé straně jsou navrhnuta a implementována dvě prostředí v programu Matlab & Simulink, jedno pro CAN a druhé pro CAN FD sběrnici. Tato prostředí pak slouží k přístupění externího solveru a umožňují simulaci v reálném čase v konfiguraci "hardware-ve-smyčce". Obě prostředí jsou důkladně otestována spolu s aplikací v kontroleru a je provedena řada měření testující dobu, kterou celý rámec potřebuje k vyřešení kvadratického problému v závislosti na velikosti daného problému. Na závěr jsou předvedena dvě dema využívající modelového prediktivního řízení a vyvinutého externího solveru k optimalizaci formulovaných problémů.

Klíčová slova: Kvadratické programování, QP, Vestavěný solver, CAN, CAN FD, Optimální řízení, Modelové prediktivní řízení, Externí řešič, Řízení vozidla

Překlad názvu: Využití vestavěného QP solveru pro automotive aplikace

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Motivation

Historically, vehicles were considered to be mechanical systems. To improve performance or efficiency, car manufacturers would add cylinders to the combustion engine or make the car's body more aerodynamic. Only in recent decades, with the rise of digitalization, have vehicles transformed into exhaustively complex digital systems, which rely on software no less than on properly tuned suspension. With the current trends of the automotive industry to push for emission reduction and safety improvements, the development cycle of new vehicle models is not only about deploying new hardware components but also about optimizing the software controlling these components. But with the increasing complexity of the system, the need for sophisticated control strategies, which ensure reliable operation of the whole plant, increases as well. For that reason is model based predictive control (MPC) popularized in the automotive industry [1].

It is not uncommon for a modern vehicle to be equipped with a number of systems that would benefit from model based predictive control. These systems can be anything from battery management to cruise control. However, equipping all of them with the hardware and software necessary to run the MPC controller could prove time and money inefficient. Since both linear and non-linear MPCs are eventually reformulated as quadratic optimization problems, centralizing the task of solving these problems may be an elegant solution to this issue. Furthermore, developing a dedicated external quadratic program (QP) solver presents several other advantages. Firstly, it would decrease the computational requirements on the system controllers, as the demanding task of QP optimization would be performed on external hardware. Moreover, deploying and maintaining the embedded QP solver would be simplified as well, since any updates and improvements to the software would directly affect only one controller in the vehicle.

1.2 Thesis goals

The goal of the thesis is to develop and implement a Simulink based framework, which will allow the user to solve quadratic problems on embedded hardware using automotive grade bus like CAN and CAN FD. The thesis will address following points:

- Familiarization with the topic of quadratic programming, automotive grade communication and embedded hardware
- Analysis of representative automotive grade communication solutions
- Adaptation of an existing QP solver and its deployment
- Realization of baseline and improved communication logic and verification of the workings of the embedded QP solver
- Utilization of the embedded QP solver in the MPC domain, analysis and presentation of acquired results

1.3 Thesis outline

This thesis will be divided into five main chapters, excluding the introduction and conclusion. Chapter number two will consist of research on thesis-related topics, such as quadratic programming, linear model based predictive control, and automotive-grade field buses. How does receding horizon control reformulate into a QP, and what communication bus to use to transmit the QP problem to an external controller? This will be followed by a chapter that deals with the presentation of the selected microcontroller and the deployment of two QP solvers onto the said controller. The next chapter will focus on the development of a CAN bus based interface, both on the side of the embedded application and the side of personal computer running Matlab & Simulink. First tests and measurements will be performed, showcasing the workings of the deployed solvers and the newly developed interface. Chapter five will then be similar in its content as the making of a CAN FD based interface will be showcased. This time more focus will be on the interface performance testing and subsequent presentation of acquired results. Last but not least, the final chapter will present MPC demos running in a hardware-in-the-loop (HIL) configuration to display the functionality of the developed *external QP solver* framework.

Chapter 2

Theoretical analysis

2.1 Quadratic programming

Mathematical optimization has always been a subject of great interest for mathematicians. Newton's iterative method for locating roots of real continuous functions, the Lagrange multiplier method used for finding optima of constrained functions, or the simplex method used for solving linear programs, all play an essential role in the current state of mathematical optimization. Quadratic programming as we know it has its root in the 1950' and was popularized f.e. by the Economist and Nobel prize laureate Harry Markowitz who formulated the portfolio optimization problem as a quadratic program [3]. As of today, QP is the most prominent method for solving non-linear optimization problems and is applied in many areas such as computer graphics, signal and image processing, investment optimization, and optimal control theory.

2.1.1 Quadratic program definition

A general linear quadratic program, i.e. an optimization problem with quadratic cost function and linear constraints, can be written as follows.

$$\begin{aligned} \min_{\mathbf{x}} \quad & \frac{1}{2}\mathbf{x}^\intercal \mathbf{H}\mathbf{x} + \mathbf{f}^\intercal \mathbf{x} \\ \text{s.t.} \quad & \mathbf{E}\mathbf{x} = \mathbf{e} \\ & \mathbf{A}\mathbf{x} \leq \mathbf{a} \end{aligned} \tag{2.1}$$

Where $\mathbf{x} \in \mathbb{R}^n$ is the optimization vector and $\mathbf{H} \in \mathbb{R}^{n \times n}$ with $\mathbf{f} \in \mathbb{R}^n$ denote the quadratic and linear parts of the cost function. The constraints are then defined in two categories, firstly the equality constraints, $\mathbf{E} \in \mathbb{R}^{m_1 \times n}$ and $\mathbf{e} \in \mathbb{R}^{m_1}$, secondly the inequality constraints, $\mathbf{A} \in \mathbb{R}^{m_2 \times n}$ and $\mathbf{a} \in \mathbb{R}^{m_2}$.

Furthermore, without a loss of generality it will be assumed that $\mathbf{H}$ is a symmetric matrix [5].

Proof. To prove this it can be shown, that replacing any matrix $\mathbf{H}$ with the symmetric matrix $\frac{1}{2}(\mathbf{H} + \mathbf{H}^\intercal)$ will not affect the value of the cost function.

Since $\mathbf{x}^\intercal\mathbf{H}\mathbf{x} \in \mathbb{R}$:

$$\begin{aligned}
\mathbf{x}^\intercal\mathbf{H}\mathbf{x} &= \mathbf{x}^\intercal\left(\frac{2\mathbf{H}}{2} + \frac{\mathbf{H}^\intercal - \mathbf{H}^\intercal}{2}\right)\mathbf{x} \\
&= \mathbf{x}^\intercal\left(\frac{\mathbf{H}+\mathbf{H}^\intercal}{2} + \frac{\mathbf{H}-\mathbf{H}^\intercal}{2}\right)\mathbf{x} \\
&= \mathbf{x}^\intercal\left(\frac{\mathbf{H}+\mathbf{H}^\intercal}{2}\right)\mathbf{x} + \frac{1}{2}\mathbf{x}^\intercal\mathbf{H}\mathbf{x} - \frac{1}{2}\mathbf{x}^\intercal\mathbf{H}^\intercal\mathbf{x} \\
&= \mathbf{x}^\intercal\left(\frac{\mathbf{H}+\mathbf{H}^\intercal}{2}\right)\mathbf{x} + \frac{1}{2}\mathbf{x}^\intercal\mathbf{H}\mathbf{x} - \left(\frac{1}{2}\mathbf{x}^\intercal\mathbf{H}^\intercal\mathbf{x}\right)^\intercal \\
&= \mathbf{x}^\intercal\left(\frac{\mathbf{H}+\mathbf{H}^\intercal}{2}\right)\mathbf{x}
\end{aligned} \tag{2.2}$$

□

2.1.2 Convex quadratic programming

General set $S \in \mathbb{R}^n$ is said to be *convex* if for any two points $x, y \in S$ the straight line connecting these points lies entirely within S. Formally, $\alpha x + (1-\alpha)y \in S \quad \forall\alpha \in [0,1]$. Similarly, the function $f : A \to B$ is convex if the domain of the function A is a convex set and for any two points $x, y \in A$ the following is true:

$$f(\alpha x + (1-\alpha)y) \leq \alpha f(x) + (1-\alpha)f(y) \quad \forall\alpha \in [0,1] \tag{2.3}$$

For numerical optimization the concept of convexity is extremely useful as a local minimum of a convex function on a convex subspace is also its global minimum. For reference see the topic of convexity (page 7) in [2].

Proof. Let us define $f : A \to B$ as a convex function with a convex domain A. Furthermore, $x^* \in A$ is the local minimum of f. By definition of local minimum, an open neighborhood N exists s such that $f(x^*) \leq f(n) \quad \forall n \in N$. For any arbitrary point $y \in A$ we can take the linear combination $z = (1-\alpha)x^* + \alpha y$ and say that as α approaches 0, z approaches x^* which implies $z \in N$. Therefore, for $\alpha \to 0$:

$$\begin{aligned}
f(x^*) &\leq f(z) \\
f(x^*) &\leq f((1-\alpha)x^* + \alpha y) \\
f(x^*) &\leq (1-\alpha)f(x^*) + \alpha f(y) \\
\alpha f(x^*) &\leq \alpha f(y) \\
f(x^*) &\leq f(y)
\end{aligned} \tag{2.4}$$

This shows that for a convex problem the local minimum is also the global minimum as $f(x^*) \leq f(y) \quad \forall y \in A$. □

Whether the general QP problem as defined in 2.1 is a convex problem or not depends on both the cost function and the linear constraints. The cost function itself generally consists of a sum of two terms - quadratic and linear. As for the linear term, $\mathbf{f}^\intercal\mathbf{x}$ is convex for any $\mathbf{f} \in \mathbb{R}^n$, whereas the quadratic

form $\mathbf{x}^\intercal\mathbf{H}\mathbf{x}$ is convex only for a positive semidefinite matrix $\mathbf{H}$. For such (symmetric) positive semidefinite matrix the following is true:

$$\mathbf{x}^\intercal\mathbf{H}\mathbf{x} \geq 0 \quad \forall \mathbf{x} \in \mathbb{R}^n \iff \forall\, eig(\mathbf{H}) \geq 0 \tag{2.5}$$

In short, the cost function of a general quadratic program is convex on $\mathbb{R}^n$ as long as $\mathbf{H}$ is positive semidefinite, page 8 [2].

As for the QP constraints described by the linear equations and inequations, this is where the topic of convexity analysis becomes more complicated. Firstly, the set of feasible solutions described by these constraints must be non-empty. As an example, if the problem presents two equality constraints $x = 1$ and $x = 2$, the problem cannot be solved. Secondly, no constraints should be redundant. Again, out of these two inequality constraints $x \leq 1$ and $x \leq 2$ is the second one redundant. These two requests imply certain limitations towards the matrices $\mathbf{A}$ and $\mathbf{E}$. For that reason, an expanded, more limiting definition of a QP program is often presented in the literature. The equality constraint matrix $\mathbf{E}$ ought to be of dimension $m_1 \times n$ where $m_1 \leq n$ and $rank(E) = m_1$, i.e. the constraints are linearly independent. Similarly, for the inequality constraint matrix $\mathbf{A}$. This ensures that the set of feasible solutions is an n-dimensional polyhedron which is always convex, see page 451 in [2]. Note that some methods exist to deal with problems with redundant constraints, but the feasibility set must be non-empty in any case, as per page 491 in [2].

To summarize a QP is said to be a convex problem as long as the quadratic cost function matrix $\mathbf{H}$ is positive semidefinite and the problem constraints limit the set of feasible solution to a polyhedron.

2.2 Model predictive control

The roots of model based predictive control can be tracked to the early 1960' when the renowned mathematician Rudolf E. Kalman presented the Linear Quadratic Regulator and subsequently the Linear Quadratic Gaussian (LQG) controller for stochastic systems. Although a powerful tool in theory, LQG presented several drawbacks which prevented it from mass adoption in the control industry. For example, the lack of input/state constraints handling, no way to incorporate process non-linearities, or no robustness guarantees [4]. Developed from the same theory of optimal control, model predictive control is a universal control methodology used to optimize the performance of an arbitrary system under constraints. This section will describe the basic concepts of MPC as well its ties to the concept of quadratic programming discussed in the previous section "Quadratic programming".

2.2.1 Dynamic models

The core idea behind MPC is to use a dynamic model of a system to predict its behavior and, based on that prediction, propose the best control decision. Unlike input/output based regulators, models and the ability to simulate them are key for the performance of model based controllers. A general non-linear system can be described by the following equations:

$$\begin{aligned} \frac{d\mathbf{x}}{dt} &= f(\mathbf{x}, \mathbf{u}, t) \\ \mathbf{y} &= g(\mathbf{x}, \mathbf{u}, t) \\ \mathbf{x}(t_0) &= \mathbf{x_0} \end{aligned} \tag{2.6}$$

Where $\mathbf{x} \in \mathbb{R}^n$ is the state of the system, $\mathbf{u} \in \mathbb{R}^m$ the input and $\mathbf{y} \in \mathbb{R}^p$ the output [6]. Note that it is generally assumed that the input, state and output are functions of time unless specified otherwise.

A specific class of models especially useful in control engineering is the linear state space model of a system:

$$\begin{aligned} \frac{d\mathbf{x}}{dt} &= \mathbf{A}(t)\mathbf{x} + \mathbf{B}(t)\mathbf{u} \\ \mathbf{y} &= \mathbf{C}(t)\mathbf{x} + \mathbf{D}(t)\mathbf{u} \\ \mathbf{x}(t_0) &= \mathbf{x_0} \end{aligned} \tag{2.7}$$

Where $\mathbf{A}(t) \in \mathbb{R}^{n \times n}$ is the state transition matrix, $\mathbf{B}(t) \in \mathbb{R}^{m \times n}$ describes the effect of input on the state dynamics, $\mathbf{C}(t) \in \mathbb{R}^{p \times n}$ is the output matrix and finally $\mathbf{D}(t) \in \mathbb{R}^{p \times m}$ is the direct link between input and output of the system omitting the state dynamics. For state matrices independent on time is such linear model called the *linear time invariant* state space model.

As MPC is used with discrete time models a discrete time LTI model can be defined similarly to its continuous time counterpart:

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k \\ \mathbf{y}_k &= \mathbf{C}\mathbf{x}_k + \mathbf{D}\mathbf{u}_k \\ \mathbf{x}_0 &= \text{given} \end{aligned} \tag{2.8}$$

2.2.2 Discrete model optimal control

To begin with, the problem of optimal control in discrete time system will be defined. Lets assume the goal is to optimally regulate a model described by some discrete time function - i.e., find such sequence of inputs that will guide the state of the system to the origin optimally according to a criterion function $J(\mathbf{x}, \mathbf{u})$.

$$\begin{aligned} \min_{\mathbf{x}_1 \ldots \mathbf{x}_N, \mathbf{u}_0 \ldots \mathbf{u}_{N-1}} \quad & J(\mathbf{x}, \mathbf{u}) = \Phi(\mathbf{x}_N) + \sum_{k=0}^{N-1} L(\mathbf{x}_k, \mathbf{u}_k) \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k) \\ & \mathbf{x}_0 = \text{given} \\ & \mathbf{x}_{lb} \leq \mathbf{x}_k \leq \mathbf{x}_{ub} \\ & \mathbf{u}_{lb} \leq \mathbf{u}_k \leq \mathbf{u}_{ub} \end{aligned} \tag{2.9}$$

When looking at the criterion function $J(\mathbf{x}, \mathbf{u})$ one could propose to incorporate the terminal cost function $\Phi(\mathbf{x}_N)$ into the sum but it is generally a useful practice to to define the terminal cost independently. Another thing to consider is whether the final state $\mathbf{x}_N$ is free and penalized within $J(\mathbf{x}, \mathbf{u})$ (as shown in 2.9) or whether $\mathbf{x}_N$ is fixed and therefore one of the optimization problem constraints [7].

2.2.3 Unconstrained linear MPC

Inspired by 2.9 the optimal regulation of a linear system with quadratic cost function can be defined. Note that this section, together with the following section "Constrained linear MPC" is based on the following materials [5, 7]. For now, the constraints on states and inputs will be omitted and addressed later.

$$\begin{aligned} \min_{\mathbf{x}_1 \ldots \mathbf{x}_N, \mathbf{u}_0 \ldots \mathbf{u}_{N-1}} \quad & J(\mathbf{x}, \mathbf{u}) = \mathbf{x}_N^\intercal \mathbf{P} \mathbf{x}_N + \frac{1}{2} \sum_{k=0}^{N-1} \left(\mathbf{x}_k^\intercal \mathbf{Q} \mathbf{x}_k + \mathbf{u}_k^\intercal \mathbf{R} \mathbf{u}_k \right) \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k \\ & \mathbf{x}_0 = \text{given} \end{aligned} \tag{2.10}$$

In this formulation, the matrices $\mathbf{P}, \mathbf{Q}, \mathbf{R}$ are weights that tune the final controller. Do I allow for slower regulation by increasing the input cost and thus limiting the action amplitude, or do I prefer to regulate the system as fast as possible by penalizing the state error more than the input cost? And how much do I care about the final state? For reasons which will emerge later, it is required that $\mathbf{P}, \mathbf{Q}$ are symmetric real positive semi-definite matrices and $\mathbf{R}$ is symmetric real positive definite.

It still holds that the goal of this optimization problem is to find $\mathbf{u}^*, \mathbf{x}^*$ that would minimize the cost function and thus optimize the state trajectory of the system. One approach to solve this problem is to reformulate it in a way such that the cost function $J(\mathbf{x}_0, \mathbf{u})$ is only a function of inputs and the initial state. In literature this is referred to as *batch* or *sequential* approach. First, let us rewrite the cost function in matrix form.

$$J(\mathbf{x},\mathbf{u}) = \mathbf{x}_0^\intercal \mathbf{Q}\mathbf{x}_0 + \begin{bmatrix}\mathbf{x}_1\\ \vdots \\ \mathbf{x}_N\end{bmatrix}^\intercal \underbrace{\begin{bmatrix}\mathbf{Q} & \mathbf{0} & \cdots & \cdots & \mathbf{0}\\ \mathbf{0} & \mathbf{Q} & \mathbf{0} & \cdots & \mathbf{0}\\ \vdots & \mathbf{0} & \ddots & & \vdots\\ \vdots & \vdots & & \mathbf{Q} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{P}\end{bmatrix}}_{\bar{\mathbf{Q}}} \begin{bmatrix}\mathbf{x}_1\\ \vdots \\ \mathbf{x}_N\end{bmatrix} + \begin{bmatrix}\mathbf{u}_0\\ \vdots \\ \mathbf{u}_{N-1}\end{bmatrix}^\intercal \underbrace{\begin{bmatrix}\mathbf{R} & & \mathbf{0}\\ & \ddots & \\ \mathbf{0} & & \mathbf{R}\end{bmatrix}}_{\bar{\mathbf{R}}} \underbrace{\begin{bmatrix}\mathbf{u}_0\\ \vdots \\ \mathbf{u}_{N-1}\end{bmatrix}}_{\mathbf{u}} \tag{2.11}$$

And similarly the discrete state space equation.

$$\begin{bmatrix}\mathbf{x}_1\\ \vdots \\ \mathbf{x}_N\end{bmatrix} = \underbrace{\begin{bmatrix}\mathbf{B} & \mathbf{0} & \cdots & \mathbf{0}\\ \mathbf{AB} & \mathbf{B} & \cdots & \mathbf{0}\\ \vdots & \vdots & \ddots & \vdots\\ \mathbf{A^{N-1}B} & \mathbf{A^{N-2}B} & \cdots & \mathbf{B}\end{bmatrix}}_{\bar{\mathbf{S}}} \begin{bmatrix}\mathbf{u}_0\\ \vdots \\ \mathbf{u}_{N-1}\end{bmatrix} + \underbrace{\begin{bmatrix}\mathbf{A}\\ \vdots \\ \mathbf{A}^N\end{bmatrix}}_{\bar{\mathbf{T}}} \mathbf{x}_0 \tag{2.12}$$

By plugging 2.12 into 2.11 the final form of the cost function is obtained.

$$\begin{aligned} J(\mathbf{x}_0,\mathbf{u}) &= \frac{1}{2}\mathbf{u}^\intercal \underbrace{2(\bar{\mathbf{R}} + \bar{\mathbf{S}}^\intercal \bar{\mathbf{Q}}\bar{\mathbf{S}})}_{\mathbf{H}} \mathbf{u} + \mathbf{x}_0^\intercal \underbrace{2\bar{\mathbf{T}}^\intercal \bar{\mathbf{Q}}\bar{\mathbf{S}}}_{\mathbf{F}^\intercal} \mathbf{u} + \underbrace{\mathbf{x}_0^\intercal(\mathbf{Q} + \bar{\mathbf{T}}^\intercal \bar{\mathbf{Q}}\bar{\mathbf{T}})\mathbf{x}_0}_{c\,=\,const.} \\ &= \frac{1}{2}\mathbf{u}^\intercal \mathbf{H}\mathbf{u} + \mathbf{x}_0\mathbf{F}^\intercal \mathbf{u} + c \end{aligned} \tag{2.13}$$

And finally to obtain the optimal control sequence $\mathbf{u}^*$ the gradient of the cost function with respect to $\mathbf{u}$ will be calculated and solved for setting it equal to zero.

$$\nabla_{\mathbf{u}} J(\mathbf{x}_0,\mathbf{u}) = \mathbf{H}\mathbf{u} + \mathbf{F}\mathbf{x}_0 \tag{2.14}$$

$$\mathbf{H}\mathbf{u} + \mathbf{F}\mathbf{x}_0 = 0 \implies \mathbf{u}^* = -\mathbf{H}^{-1}\mathbf{F}\mathbf{x}_0 \tag{2.15}$$

This leaves us with the conclusion that the unconstrained linear MPC with a fixed finite horizon is simply a linear state feedback.

2.2.4 Constrained linear MPC

The main benefit of MPC its capability to handle input and state/output constraints. Given the optimal control problem as defined earlier in 2.10 more

constraints will be added in form of lower and upper bounds of states and inputs.

$$\begin{aligned} \mathbf{u}_{lb} \leq \mathbf{u}_j \leq \mathbf{u}_{ub}, \quad & j = 0...N-1 \\ \mathbf{x}_{lb} \leq \mathbf{x}_i \leq \mathbf{x}_{ub}, \quad & i = 1...N \end{aligned} \tag{2.16}$$

These can be rewritten in matrix form as follow.

$$\begin{aligned} \underbrace{\begin{bmatrix} \mathbf{u}_{lb} \\ \vdots \\ \mathbf{u}_{lb} \end{bmatrix}}_{\mathbf{U}_{lb}} \leq \begin{bmatrix} \mathbf{I} & & \\ & \ddots & \\ & & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{u}_0 \\ \vdots \\ \mathbf{u}_{N-1} \end{bmatrix} \leq \underbrace{\begin{bmatrix} \mathbf{u}_{ub} \\ \vdots \\ \mathbf{u}_{ub} \end{bmatrix}}_{\mathbf{U}_{ub}} \\ \underbrace{\begin{bmatrix} \mathbf{x}_{lb} \\ \vdots \\ \mathbf{x}_{lb} \end{bmatrix}}_{\mathbf{X}_{lb}} \leq \underbrace{\begin{bmatrix} \mathbf{I} & & \\ & \ddots & \\ & & \mathbf{I} \end{bmatrix}}_{\bar{\mathbf{I}}} \underbrace{\begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_N \end{bmatrix}}_{\mathbf{x}} \leq \underbrace{\begin{bmatrix} \mathbf{x}_{ub} \\ \vdots \\ \mathbf{x}_{ub} \end{bmatrix}}_{\mathbf{X}_{ub}} \end{aligned} \tag{2.17}$$

Substituting for $\mathbf{x}$ from 2.12 and rewriting.

$$\begin{aligned} \mathbf{U}_{lb} &\leq \bar{\mathbf{I}}\mathbf{u} \leq \mathbf{U}_{ub} \\ \mathbf{X}_{lb} &\leq \bar{\mathbf{I}}\left(\bar{\mathbf{S}}\mathbf{u} + \bar{\mathbf{T}}\mathbf{x}_0\right) \leq \mathbf{X}_{ub} \end{aligned} \tag{2.18}$$

We can now formulate the final constrained optimization problem using the results obtained in 2.13 and 2.18.

$$\begin{aligned} \min_{\mathbf{u}} \quad & \frac{1}{2}\mathbf{u}^\intercal \mathbf{H}\mathbf{u} + \mathbf{x}_0\mathbf{F}^\intercal\mathbf{u} \\ \text{s.t.} \quad & \begin{bmatrix} \mathbf{U}_{lb} \\ \mathbf{X}_{lb} - \bar{\mathbf{I}}\bar{\mathbf{T}}\mathbf{x}_0 \end{bmatrix} \leq \begin{bmatrix} \bar{\mathbf{I}} \\ \bar{\mathbf{S}} \end{bmatrix} \mathbf{u} \leq \begin{bmatrix} \mathbf{U}_{ub} \\ \mathbf{X}_{ub} - \bar{\mathbf{I}}\bar{\mathbf{T}}\mathbf{x}_0 \end{bmatrix} \\ & \mathbf{x}_0 = \text{given} \end{aligned} \tag{2.19}$$

This quadratic problem finally formulates the task of finding an optimal control sequence $\mathbf{u}^*$ for regulating some linear discrete-time system with constrained inputs and states. It is clearly an instance of a generic QP formulation defined in the previous section (see 2.1). The first row of constraints represents the box constraints of inputs defined directly, and the second row represents the affine constraints of inputs derived from constraints on system states.

Output constraints

It is often beneficial to enforce constraints onto the outputs instead of the states of the system. This can be achieved via small modification to the

previous derivation. As in 2.18 we begin by defining the constraints in the following form.

$$\mathbf{y}_{lb} \leq \mathbf{y}_i \leq \mathbf{y}_{ub}, \quad i = 1...N \tag{2.20}$$

Rewriting as a matrix equation.

$$\underbrace{\begin{bmatrix} \mathbf{y}_{lb} \\ \vdots \\ \mathbf{y}_{lb} \end{bmatrix}}_{\mathbf{Y}_{lb}} \leq \begin{bmatrix} \mathbf{I} & & \\ & \ddots & \\ & & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_N \end{bmatrix} \leq \underbrace{\begin{bmatrix} \mathbf{y}_{ub} \\ \vdots \\ \mathbf{y}_{ub} \end{bmatrix}}_{\mathbf{Y}_{ub}} \tag{2.21}$$

Since $\mathbf{y}_k = \mathbf{C}\mathbf{x}_k$, by using the state space equation 2.12 the vector of outputs can be rewritten using the system matrices and inputs.

$$\begin{bmatrix} \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_N \end{bmatrix} = \underbrace{\begin{bmatrix} \mathbf{CB} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{CAB} & \mathbf{CB} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{CA^{N-1}B} & \mathbf{CA^{N-2}B} & \cdots & \mathbf{CB} \end{bmatrix}}_{\bar{\mathbf{M}}} \begin{bmatrix} \mathbf{u}_0 \\ \vdots \\ \mathbf{u}_{N-1} \end{bmatrix} + \underbrace{\begin{bmatrix} \mathbf{CA} \\ \vdots \\ \mathbf{CA}^N \end{bmatrix}}_{\bar{\mathbf{N}}} \mathbf{x}_0 \tag{2.22}$$

Combining the last two equation 2.21 and 2.22 the output constraint inequality can be formulated as follows.

$$\mathbf{Y}_{lb} \leq \bar{\mathbf{I}} \left(\bar{\mathbf{M}}\mathbf{u} + \bar{\mathbf{N}}\mathbf{x}_0 \right) \leq \mathbf{Y}_{ub} \tag{2.23}$$

Finalizing the formulation, the quadratic program representing input and output constrained optimal control problem is presented.

$$\begin{aligned} \min_{\mathbf{u}} \quad & \frac{1}{2}\mathbf{u}^\intercal \mathbf{H}\mathbf{u} + \mathbf{x}_0 \mathbf{F}^\intercal \mathbf{u} \\ \text{s.t.} \quad & \begin{bmatrix} \mathbf{U}_{lb} \\ \mathbf{Y}_{lb} - \bar{\mathbf{I}}\bar{\mathbf{N}}\mathbf{x}_0 \end{bmatrix} \leq \begin{bmatrix} \bar{\mathbf{I}} \\ \bar{\mathbf{M}} \end{bmatrix} \mathbf{u} \leq \begin{bmatrix} \mathbf{U}_{ub} \\ \mathbf{Y}_{ub} - \bar{\mathbf{I}}\bar{\mathbf{N}}\mathbf{x}_0 \end{bmatrix} \\ & \mathbf{x}_0 = \text{given} \end{aligned} \tag{2.24}$$

Output tracking

Until this point, the obtained results were derived for the system regulation problem, which is guiding the system's states to the origin. More often than not is the actual use case the so-called *output tracking* - the task of controlling the system in such a way that the outputs follow a set trajectory. Excluding a few necessary modifications, the approach to reformulating the tracking problem as a quadratic program is almost identical to the regulation problem. Only the key idea and results will therefore be presented and the reader will be referred to several sources for more detailed derivation.

To begin with, the quadratic cost function of the optimization problem must be reformulated. Given the output $\mathbf{y}_k$ and the reference trajectory $\mathbf{r}_k$ the tracking error $\mathbf{e}_k$ can be defined.

$$\mathbf{e}_k = \mathbf{r}_k - \mathbf{y}_k = \mathbf{r}_k - \mathbf{C}\mathbf{x}_k \tag{2.25}$$

The goal is then to minimize the tracking error as well as the inputs. But in order to formulate the new cost function correctly, the control increment $\Delta\mathbf{u}_k = \mathbf{u}_k - \mathbf{u}_{k-1}$ must be introduced. This is due to the reason that generally, to drive $\mathbf{e}_k$ to zero, the system inputs are not zero. But a cost function minimizing both the error $\mathbf{e}_k$ and the inputs $\mathbf{u}_k$ does not take this into account, and it would therefore fail to provide an input trajectory leading to zero tracking error. By introducing $\Delta\mathbf{u}_k$ as the new input signal, the previous input $\mathbf{u}_{k-1} = \mathbf{x}_k^u$ will be introduced as a new state variable of the system. This leads to the following augmented state-space equations.

$$\begin{aligned} \begin{bmatrix} \mathbf{x}_{k+1} \\ \mathbf{x}_{k+1}^u \end{bmatrix} &= \underbrace{\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}}_{\hat{\mathbf{A}}} \underbrace{\begin{bmatrix} \mathbf{x}_k \\ \mathbf{x}_k^u \end{bmatrix}}_{\hat{\mathbf{x}}_k} + \underbrace{\begin{bmatrix} \mathbf{B} \\ \mathbf{I} \end{bmatrix}}_{\hat{\mathbf{B}}} \Delta\mathbf{u}_k \\ \mathbf{y}_k &= \underbrace{\begin{bmatrix} \mathbf{C} & \mathbf{0} \end{bmatrix}}_{\hat{\mathbf{C}}} \begin{bmatrix} \mathbf{x}_k \\ \mathbf{x}_k^u \end{bmatrix} \end{aligned} \tag{2.26}$$

Using this augmented system the unconstrained optimization problem of linear system tracking can be defined.

$$\begin{aligned} \min_{\hat{\mathbf{x}}_1 \ldots \hat{\mathbf{x}}_N, \Delta\mathbf{u}_0 \ldots \Delta\mathbf{u}_{N-1}} \quad & J(\hat{\mathbf{x}}, \Delta\mathbf{u}) = \left(\mathbf{r}_N - \hat{\mathbf{C}}\hat{\mathbf{x}}_N\right)^\intercal \mathbf{P} \left(\mathbf{r}_N - \hat{\mathbf{C}}\hat{\mathbf{x}}_N\right) \\ & + \frac{1}{2} \sum_{k=0}^{N-1} \left[\left(\mathbf{r}_k - \hat{\mathbf{C}}\hat{\mathbf{x}}_k\right)^\intercal \mathbf{Q} \left(\mathbf{r}_k - \hat{\mathbf{C}}\hat{\mathbf{x}}_k\right) \right] \\ & + \frac{1}{2} \sum_{k=0}^{N-1} \left(\Delta\mathbf{u}_k^\intercal \mathbf{R} \Delta\mathbf{u}_k\right) \\ \text{s.t.} \quad & \hat{\mathbf{x}}_{k+1} = \hat{\mathbf{A}}\hat{\mathbf{x}}_k + \hat{\mathbf{B}}\Delta\mathbf{u}_k \\ & \hat{\mathbf{x}}_0 = \text{given} \end{aligned} \tag{2.27}$$

Utilizing the same sequential approach as in section "Constrained linear MPC" the problem 2.27 can be reformulated into the following form [5, 8, 9].

$$J(\hat{\mathbf{x}}_0, \Delta\mathbf{u}) = \frac{1}{2}\Delta\mathbf{u}^\intercal \hat{\mathbf{H}} \Delta\mathbf{u} + \begin{bmatrix} \hat{\mathbf{x}}_0^\intercal & \mathbf{r}_0^\intercal \end{bmatrix} \hat{\mathbf{F}}^\intercal \Delta\mathbf{u} \tag{2.28}$$

Where $\Delta\mathbf{u} = [\Delta\mathbf{u}_0, ..., \Delta\mathbf{u}_{N-1}]^\intercal$ and $\hat{\mathbf{F}}, \hat{\mathbf{H}}$ are matrices obtained during the derivation. Adding the constraints on inputs, states and outputs is again similar to the approach presented on the mpc regulation problem and results in inequality affine constrained QP.

Transforming affine constraints into box constraints using slack variables and softening

Given an MPC problem formulation with constrained states or outputs, the final quadratic program generally has affine constraints as per the results presented in 2.24 and 2.19. Such a problem is already solvable using today's solvers, but it could be beneficial to transform the affine constraints to obtain a purely box constrained QP as such a problem is generally easier to solve and supported by a greater number of solvers. Furthermore, relaxing the hard state/output constraints increases the odds of finding feasible solutions for the cost of some constraint violations [10].

The approach to this transformation is following. Given a generic box and affine constrained QP problem

$$\begin{aligned} \min_{\mathbf{x}} \quad & \frac{1}{2}\mathbf{x}^\intercal\mathbf{H}\mathbf{x} + \mathbf{f}^\intercal\mathbf{x} \\ \text{s.t.} \quad & \mathbf{x}_{lb} \leq \mathbf{x} \leq \mathbf{x}_{ub} \\ & \mathbf{a}_{lb} \leq \mathbf{A}\mathbf{x} \leq \mathbf{a}_{ub} \end{aligned} \tag{2.29}$$

where $\mathbf{x} \in \mathbb{R}^{n\times 1}$ and $\mathbf{A} \in \mathbb{R}^{m\times n}$, introduce a new vector of slack variables $\zeta \in \mathbb{R}^{m\times 1}$. These variables will then be constrained by the lower and upper bounds $\mathbf{a}_{lb}, \mathbf{a}_{ub}$ and the cost function will be extended by a new element as follows.

$$J(\mathbf{x}, \zeta) = \frac{1}{2}\mathbf{x}^\intercal\mathbf{H}\mathbf{x} + \mathbf{f}^\intercal\mathbf{x} + \frac{1}{2}(\mathbf{A}\mathbf{x} - \zeta)^\intercal\mathbf{W}(\mathbf{A}\mathbf{x} - \zeta) \tag{2.30}$$

The matrix $\mathbf{W} \in \mathbb{R}^{m\times m}$ is a new weight pushing the new problem to respect the original affine constraints. Note that it makes sense to select $\mathbf{W}$ diagonal either with constant diagonal elements or with varying elements depending on the "importance" of individual constraints. Rewriting 2.30 results in.

$$J(\mathbf{x}, \zeta) = \frac{1}{2}\mathbf{x}^\intercal\mathbf{H}\mathbf{x} + \mathbf{f}^\intercal\mathbf{x} + \frac{1}{2}\left(\mathbf{x}^\intercal\mathbf{A}^\intercal\mathbf{W}\mathbf{A}\mathbf{x} - \zeta^\intercal\mathbf{W}\mathbf{A}\mathbf{x} - \mathbf{x}^\intercal\mathbf{A}^\intercal\mathbf{W}\zeta + \zeta^\intercal\mathbf{W}\zeta\right) \tag{2.31}$$

And after rewriting the cost function in vector form the transformation of the QP problem can be finalized.

$$\begin{aligned} \min_{\mathbf{x},\zeta} \quad & \begin{bmatrix}\mathbf{x}^\intercal & \zeta^\intercal\end{bmatrix}\begin{bmatrix}\mathbf{H} + \mathbf{A}^\intercal\mathbf{W}\mathbf{A} & -\mathbf{W}\mathbf{A} \\ -\mathbf{A}^\intercal\mathbf{W} & \mathbf{W}\end{bmatrix}\begin{bmatrix}\mathbf{x} \\ \zeta\end{bmatrix} + \begin{bmatrix}\mathbf{F}^\intercal & \mathbf{0}^\intercal\end{bmatrix}\begin{bmatrix}\mathbf{x} \\ \zeta\end{bmatrix} \\ \text{s.t.} \quad & \begin{bmatrix}\mathbf{x}_{lb} \\ \mathbf{a}_{lb}\end{bmatrix} \leq \begin{bmatrix}\mathbf{x} \\ \zeta\end{bmatrix} \leq \begin{bmatrix}\mathbf{x}_{ub} \\ \mathbf{a}_{ub}\end{bmatrix} \end{aligned} \tag{2.32}$$

This QP problem finally represents the transformation of the original affine constrained problem 2.29 into a box and soft constrained problem. Getting rid of these affine restrictions is at the cost of larger number of optimization variables ($n + m$ instead of n) and potential violation of the original constraints.

2.2.5 Receding horizon control

With the obtained result, the general approach of MPC regulation can now be described. At each time step, the current state of the system is obtained. This current state can be denoted as an initial state at time $t_0 = 0$ as an LTI system is assumed. Given $\mathbf{x}_0$, the QP problem is formulated, solved, and the optimal input trajectory $\mathbf{u}^*$ is obtained. As a next step, the first input $\mathbf{u}_0$ is applied to the system, and in the next time step, the whole process is repeated. This strategy is often referred to as *receding horizon control*. See figure 2.1 for an illustration of a single MPC regulation step.

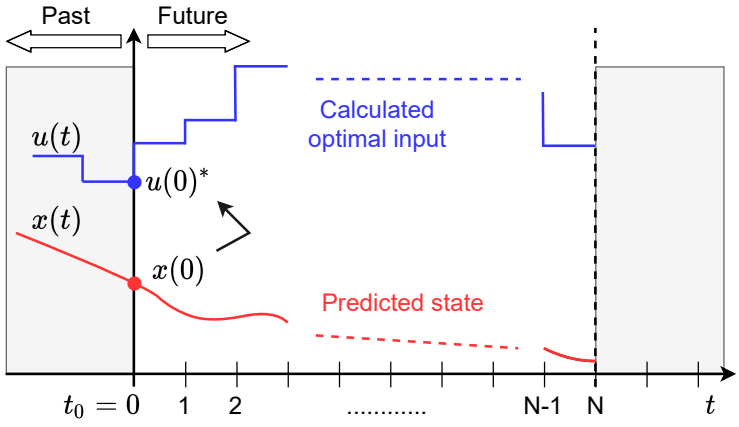


Figure 2.1: Basic MPC horizon preview

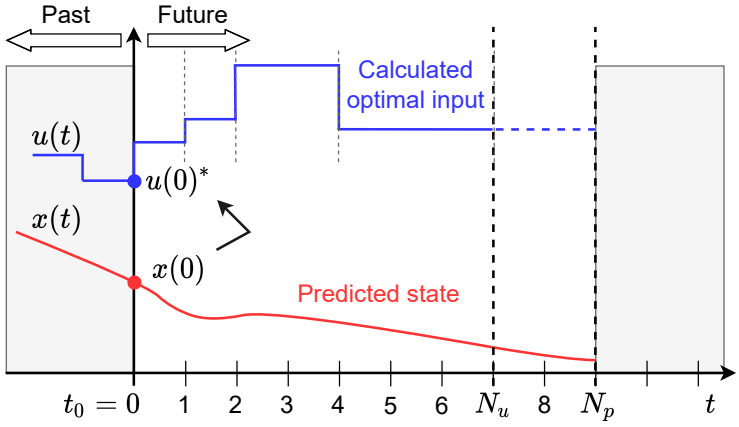


Figure 2.2: MPC with control horizon and input blocking

Control horizon and input blocking

The most computationally and memory demanding step of the receding horizon control is solving a new quadratic problem at each time step. Given the prediction horizon N and the number of inputs m, the QP optimizes over

$N \times m$ variables, and it is up to the designer to select N large enough for acceptable controller performance but small enough to keep the dimensionality of the QP within reasonable bounds.

In order to reduce said dimensionality, one strategy is to define so called *prediction horizon* N_p and *control horizon* N_u such that $N_u \leq N_p$. The problem is then reformulated in such way that the input is assumed constant after N_u steps which reduces the number of optimization variables. Another approach is to introduce *input blocking*. Not only is a constant input assumed after the control horizon but further blocks are defined within the control horizon in which is the input again deemed constant. Best explained with a simple example (see 2.2), let's assume $N_u = 7$, $N_p = 9$ and input is blocked via the following strategy $u_{blocks} = [1, 1, 2, 3]$, where u_{blocks} denotes the length of blocks (sample vise) over which $\Delta u = 0$. Note that $\sum u_{blocks} = N_u$.

2.3 Automotive grade communication buses

With the first two sections of this chapter describing the concept of quadratic programming and its use in model predictive control, this section will focus on another key topic of this thesis, automotive grade bus communication. As the number of electronic control units in a vehicle grew, the need to replace their point-to-point connections with more space and cost-efficient solutions increased. This is where the concept of a fieldbus comes in. A fieldbus is a distributed computer network that allows multiple nodes (ECUs) to be connected to said bus and exchange messages with each other, thus removing the need for designated connections between individual pairs of nodes [11]. Typically, multiple communication buses operate in a vehicle simultaneously, interconnecting vehicle subsystems based on their needs and importance. The following figure 2.3 shows an example distribution of vehicle subsystems with regards to their communication speed and safety requirements.

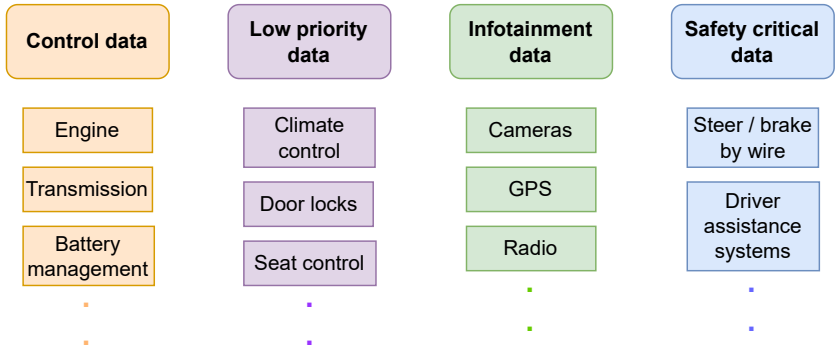


Figure 2.3: Distribution of vehicle subsystems based on their requirements, adopted from [12]

Throughout the years, a large number of automotive field buses have been developed. For example, the Media Oriented System Transport (MOST)

network is used for infotainment data transmission due to its high bandwidth. On the other hand, it only supports the ring network topology and is therefore not suitable for more safety critical data [12]. Another popular solution is the FlexRay communication bus. Suitable for sensitive data due to its inherent redundant and deterministic properties, its main disadvantage is the need for all nodes to be made completely aware of the whole network topology. This makes designing a network using FlexRey more complicated than using some alternatives [12]. For the usecase of this project, three other automotive buses will be described in more detail.

2.3.1 Local interconnected network

Local interconnected network (LIN) typically operates over the *Low priority data* shown in figure 2.3. Some other usecases include temperature, light and rain sensors reading, headlights control and a number of other functions. An individual LIN bus message (frame) is shown in the following picture 2.4. It consists of the following fields:

- **Break:** Start of frame notification for all bus nodes
- **Sync:** Predefined bit sequence used for bit rate synchronization
- **ID:** Identifier, some messages are meant only for some nodes
- **Payload:** Actual data content of the frame
- **Checksum:** Used to validate the payload contents (sometimes including the ID as well)

Figure 2.4: Local interconnected network frame, adopted from [13]

The main advantages of the LIN bus are [13]:

- Low-cost and simple solution
- Single wire with vehicle chassis acting as ground
- Master slave configuration - option to add and remove slave nodes without affecting other slave nodes, up to sixteen slave nodes
- Checksum error detection
- Collision avoidance using *time divided media access*

On the other hand, LIN bus also has several disadvantages. Mainly its low bit-rate of maximum 20 kbit/s and small payload size in a single frame which is at most 8 bytes [12]. Remember that the scope of this project includes using a communication bus to send a full box constrained QP problem as defined in 2.32 and subsequently receive its solution. The slow communication speed of the LIN bus thus poses a serious problem. To illustrate this, let's do some simple calculations.

QP sending time using the LIN bus

Given the number of optimization variables n_{var}, the total number of numeric values n_{tot} within a box constrained QP can be determined as follows.

$$n_{tot} = n_H + n_F + n_{lb} + n_{ub} + n_{x_0} = n_{var}^2 + 4n_{var} \tag{2.33}$$

Where $n_H, n_F, n_{lb}, n_{ub}, n_{x_0}$ denote the number of numeric values corresponding to individual parts of the box constrained QP problem. To reduce n_{tot}, two assumptions will be made. Firstly, $\mathbf{H}$ is symmetric and only its upper triangular part needs to be sent, therefore $n_H = n_{var}(n_{var}+1)/2$. Secondly, the initial condition $\mathbf{x}_0$ is not known and therefore not transmitted as well. These assumptions result in:

$$n_{tot} = \frac{n_{var}(n_{var}+1)}{2} + 3n_{var} \tag{2.34}$$

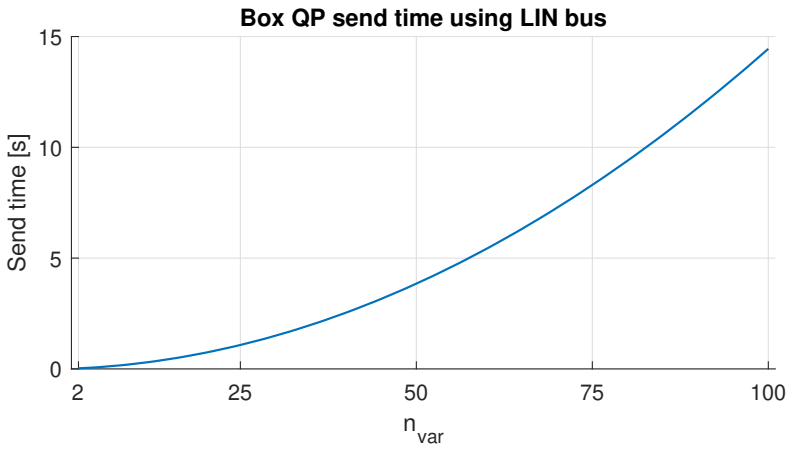


Figure 2.5: QP size and send time dependency using the LIN bus with bit-rate of 20 kbit/s

Additionally, these numeric values will be represented using the 32 bit long single precision floating point format, and only these values will need to be transmitted via the bus. Given the payload size of 64 bits and the total size of the frame, which is 108 bits (as per 2.4), it can be reasoned that to transmit two elements of the QP problem takes precisely 108 bits of information. Finally, taking into account the maximum bit rate of the LIN bus, the dependency between the QP size and the sending time is plotted in

the following figure 2.5. Note that this figure represents an upper estimate as an ideal scenario is assumed where the bus works at 100% all the time and no overhead information is transmitted.

By looking at this dependency it is obvious that using LIN bus is basically infeasible for the purposes of this project as the communication speed would simply be too slow for any reasonable controller.

2.3.2 Controller Area Network

Controller Area Network (CAN) is another example of a frequently used automotive grade communication bus. Since its initial release by Robert Bosch GmbH.™ in the late 1980s, several versions of the CAN specification were developed, and it went on to become the most widely used communication network in the automotive industry [14]. To refer to the in-vehicle systems distribution discussed previously in 2.3, CAN is typically used for the control and safety critical data applications due to its robustness and higher bit rate of up to 1 Mbit/s. It has not been popularized only in the automotive industry but in other industries as well, such as aviation, ship industry and/or robotics.

In this thesis, the CAN 2.0 specification will be discussed, sometimes referred to as high speed CAN. Referring to the seven-layer Open Systems Interconnection (OSI) model, the CAN 2.0 specification is defined for the two "lowest layers" - the physical layer and data link layer, both defined in their respective ISO standards [16]. In a typical embedded application, the implementation of CAN bus bypasses the connection between application and data link layers to maximize the performance and minimize the resource overhead. Alternatively, including the rest of the OSI layers within the application requires a higher layer protocol such as CANopen or SAE J1939 [17]. An overview is shown in the following figure 2.6.

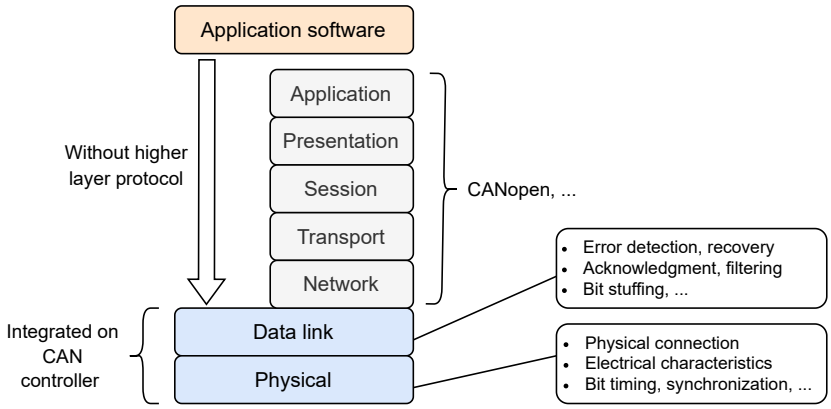


Figure 2.6: Controller Area Network arbitration phase state diagram, adopted from [16, 17]

CAN bus arbitration

Unlike the LIN bus, CAN is not a master-slave standard and is instead a multi-master bus, meaning that any node can initiate data transfer on its own initiative. Typically, networks operating in the multi-master configuration have to implement some form of data collision detection and recovery. The CAN standard solves this issue of data collision by defining a so-called arbitration phase at the beginning of each message, see 2.9. In case two or more nodes start transmitting simultaneously, the network automatically gives exclusive access to the node transmitting the highest priority message, which is detected by the remaining nodes.

The main idea behind the arbitration process could be described as follows. The bus finds itself either in a dominant or recessive state at all times. During the frame transmission, a zero bit sets the bus to the dominant state a one bit sets it to the recessive state. Furthermore dominant state always overrides the recessive one in case of simultaneous transmitting. Practically, the state of the bus is determined by the logical AND operation where all nodes' inputs give the final state of the bus, note that nodes in listening mode can be viewed as outputting recessive bits [15].

Given this property of the CAN bus, the logic behind giving access to the most important node/frame is straightforward. If multiple nodes begin transmitting a frame at the same time, the frame transmitting the message with the highest priority (lowest ID) gets access. This is best explained with the following state diagram 2.7.

Figure 2.8 shows an example of the arbitration. Two nodes begin transmitting simultaneously, meaning their start of frame bits overlap. Following that, both begin transmitting the IDs of their messages, beginning with the most significant bit. The first two bits match, so no collision is detected. On the other hand, during the third bit window, the first node notices that it transmitted a recessive bit, but the bus is in a dominant state, thus meaning another node is transmitting a higher priority message. The first node enters listening mode and tries to transmit its frame from the beginning once the second node is finished with its transmission. As long as each message is labeled with a unique ID, lower ID messages are always transmitted first.

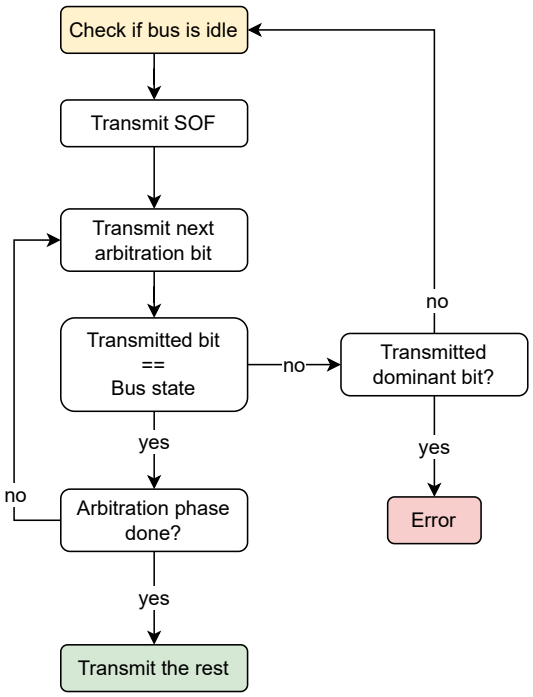


Figure 2.7: Controller Area Network arbitration phase state diagram, adopted from [15]

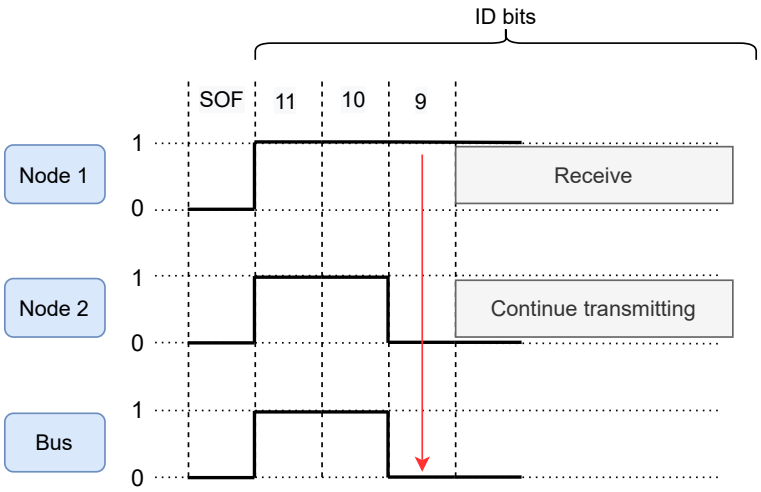


Figure 2.8: An example of arbitration process between two nodes on the controller network area bus

CAN bus data frame

The previous subsection dealt with the issue of CAN bus arbitration which utilizes a dedicated part of the data frame. The CAN frame as a whole is shown in the figure 2.9 and it consists of the following slots:

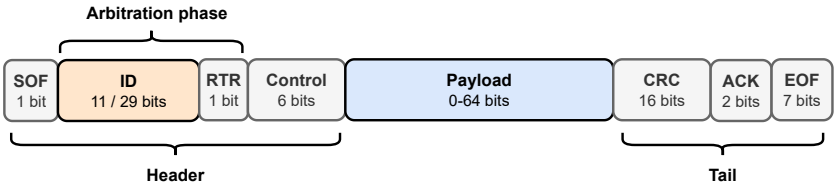


Figure 2.9: Controller Area Network data frame, adopted from [15, 16, 18]

- **SOF:** Dominant bit indicating the start of frame
- **ID:** Message identifier used for arbitration and message filtering, two CAN 2.0 specifications are used, defining either 11 or 29 bit long ID
- **RTR:** Remote transmission request, set to dominant for data request frame and recessive for data frame
- **Control:** Control section contains three parts
  - Identifier extension bit: Dominant for 11 bit ID, recessive for 29 bit ID
  - Data length code (DLC), 4 bit code declaring the size of the payload (0-8 bytes)
  - 1 reserved bit, defined as a dominant
- **Payload:** Actual data contents
- **CRC:** 15 bit long cyclic redundancy check plus recessive bit delimiter
- **ACK:** 1 bit long acknowledgment slot plus recessive bit delimiter
  - The acknowledgment during the frame transmission works as follows. First the sender sets the ACK bit as recessive. Once a listening node detects this, it overwrites the bus to the dominant state. This is again detected by the sending node to which it indicates successful acknowledgment [18].
- **EOF:** 7 bit long recessive slot indicating the end of frame

CAN bus robustness

As aforementioned, in the automotive setting is the CAN bus heavily utilized in safety-critical applications. This is due to its robustness both on physical and protocol levels. Physically the bus is realized as a balanced differential medium using a twisted pair with 120Ω characteristic impedance, which offers high noise immunity [18]. The protocol itself includes several error detection mechanisms.

Using the cyclic redundancy check, the listening nodes simply do not acknowledge proper frame reception during the acknowledgment slot. Furthermore, as described in the previous subsection "CAN bus data frame", the frame has several predefined slots with recessive bits, which also aids with error detection. On a lower level, the transmitting nodes also check for the state of the bus after a bit has been transmitted. If the bus state does not correspond to said bit an error is raised. Note that this mechanic is disabled during the arbitration and acknowledgment slots for obvious reasons. Finally, as the CAN protocol does not use a return to zero bit representation, bit stuffing was introduced. In case five same level consecutive bits are transmitted, a bit of opposing logic level is "stuffed" into the frame. This ensures that if the transmitting node gets stuck transmitting a constant logic level the receiving nodes detect this and respond with an error frame. Bit stuffing is disabled during the seven bit long EOF phase [18, 14].

QP sending time using the CAN bus

Given the popularity, robustness, and typical use cases of the CAN bus, it seems like a good candidate for the purposes of this project. Similarly, as in section 2.3.1 let's create a simple plot demonstrating the relationship between the size of a box constrained QP n_{var} and the ideal time it would take the CAN bus to transmit such QP. Again, the number n_{tot} of single-type float values needed to be sent is given by this equation 2.34. As shown in the structure of the CAN frame, see 2.9, one frame can hold two of these values, each at 32 bits, and the total length of the frame is 108 bits using the standard 11 bit ID frame. In reality, the total number of frame bits can be higher due to the bit stuffing mechanic.

Excluding the end of the frame identifier, the length of the frame becomes 101 bits. Given this, the maximum number possible of stuffed bits is 25, which increases the total size of the frame to 133 bits. Assuming a 500 kbit/s bit rate, the following figure shows the ideal box QP sending times for both the 108 and 133 bit long CAN bus frames. Although these time values are highly optimistic due to a number of reasons, it seems that the higher bit rate of the CAN bus could allow for the usage of this protocol for the purposes of this project.

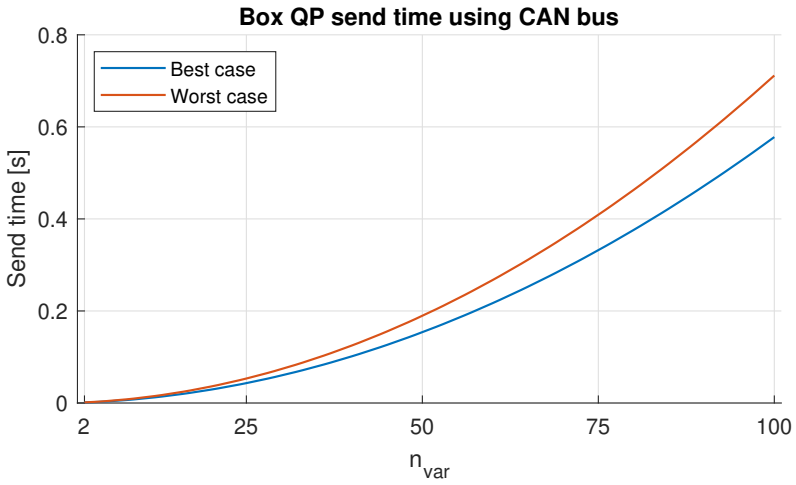


Figure 2.10: QP size and send time dependency using the CAN bus with bit-rate of 500 kbit/s

2.3.3 Controller Area Network Flexible Data-Rate

Despite its popularity, the classical CAN protocol began to be insufficient for the needs of the fast-evolving automotive industry. With the limited bit rate of 1 Mbit/s and the payload size of a single frame being eight bytes, car manufacturers would have to look elsewhere for a faster and more efficient solution. When increasing the communication speed of the CAN protocol, there are two issues. Firstly, the payload size of a single CAN frame could be increased and could potentially transfer more than two bytes of data. But with an unchanged bit rate, longer frames would block the bus for a longer time, and safety-critical data could be therefore delayed [19].

The second issue arises by trying to increase the bit rate itself. But even here is the classical CAN bus limited due to its arbitration and message acknowledgment specifications. The key problem is that the network must be set up in such a way that during a single bit time window is the "information wave" able to propagate between the two physically furthest nodes back and forth. Otherwise, the frame collision during the arbitration phase or message acknowledgment could be missed. Given the limits imposed by the finite speed of light, for 1 Mbit/s bit rate is the maximum length of the CAN bus wiring forty meters, and even less is used in practice [19]. Both of these issues are addressed in the newer CAN FD protocol specification, presented by Bosch in 2012 and ISO standardized in 2015 [21].

CAN FD bit rate switching

The main idea behind the CAN FD protocol was to introduce two bit rates and switch between them during a single frame transmission. As explained earlier, the bit rate is only really limited during the arbitration and acknowledgment phases of the CAN frame. So why not increase the communication speed, at

least during the transmission of the payload? This is exactly what CAN FD specifies. The protocol divides its frame into three main sections, see 2.11. The header, payload, and tail where the header and tail are transmitted using the so-called *arbitration bit rate* - the same factors limit this bit rate as in the classical CAN protocol, maximum 1 Mbit/s. The payload is then transmitted using the faster *payload bit rate*, utilizing the fact that during the payload phase, the information flows purely in a single direction. The payload bit rate is at most 5 Mbit/s, although even higher values might be feasible [19].

CAN FD bus data frame

The CAN FD frame is similar to the one of the standard CAN but several new fields were added due to the increased complexity of the protocol. A single frame is shown in the following figure 2.11. Note that this frame structure takes into account the standard 11 bit identifier length, for a 29 bit identifier is the header of the frame slightly modified.

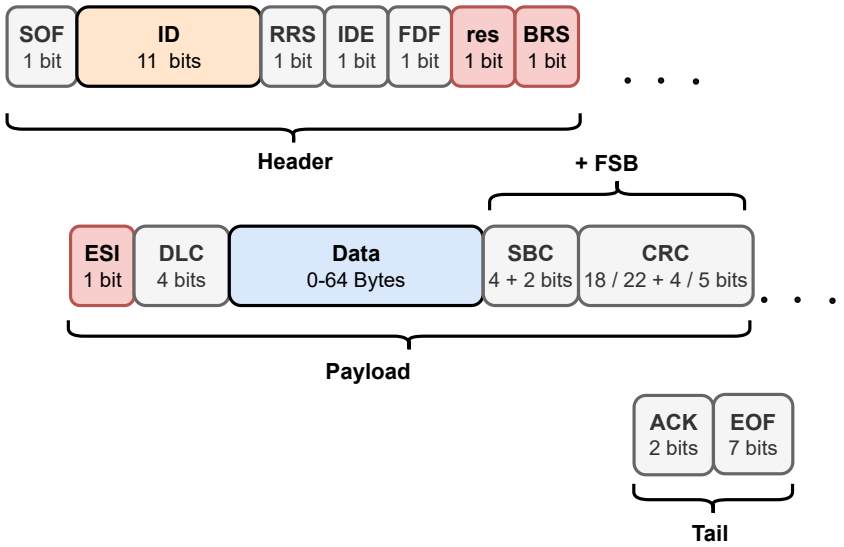


Figure 2.11: Controller Area Network Flexible Data-Rate frame for standard 11 bit ID length, adopted from [19, 21]

The meaning of some of the individual fields within the data frame has already been explained in the "CAN bus data frame" subsection, only the new fields will therefore be further described [19, 20]:

- **RRS:** Remote request substitution, replaces the RRT bit from the CAN protocol, defined as always dominant
- **IDE:** Identifier extension bit, dominant for standard ID, recessive for extended ID
- **FDF:** Defined as recessive, in CAN frame this bit is dominant (called the *reserve* bit)

- **res:** Reserved bit, defined as dominant
- **BRS:** Bit rate switch, dominant denotes no bit rate switching, recessive signalizes the use of different *payload bit rate*
- **ESI:** Error state indicator, transmitted dominant by *error active* nodes, recessive by *error passive* nodes
  - Error state is a new robustness mechanic introduced by the CAN FD protocol. Each node within the CAN FD network is initialized as *error active*. Each node has two inner counters, the transmit and receive error counter. These counters are increased/decreased according to a set of rules but once a certain value is surpassed the node is set to *error passive*. When the counters increase even further, the node becomes silent [22].
- **DLC:** Data length code, encoded by four bits, therefore the data lengths corresponding to the code are: $\{0, .., 8, 12, 16, 20, 24, 32, 48, 64\}$
- **SBC:** Stuffed bit counter, counts the number of stuffed bits before the CRC field for improved error detection
- **CRC:** Cyclic redundancy check plus CRC delimiter bit, CRC length is either 17 bits for data of lengths up to 16 bytes, or 21 bits for longer data
- **FSB:** Fixed stuff bits, to further improve error detection the frame specifies several slots with stuffed bits, the value of these bits is the opposite of the previous bit

QP sending time using the CAN FD bus

As in the previous cases of CAN and LIN protocols, given the knowledge about the structure of the data frame and bit rates of the CAN FD bus, some upper estimation about the sending times of a box constrained QP can be made. The setup will be similar as previously in "QP sending time using the CAN bus. The 11 bit identifier and 64 byte data field will be assumed which leads to 577 bit long frames (for simplicity, only fixed stuffing will be assumed), each having the capacity to transmit up to 16 single type float values.

For the CAN FD protocol, it is needed to take the dual bit rates into account. Each frame has a certain number of bits to be transmitted using the arbitration speed and another number of bits to be transmitted using the payload speed. To calculate the total time, it would take the bus to transmit this frame, an "average" bit rate can be obtained as follows:

$$\text{Average bit rate} = \frac{\text{Total number of bits}}{\frac{\text{Payload bits}}{\text{Payload bit rate}} + \frac{\text{Arbitration bits}}{\text{Arbitration bit rate}}} \tag{2.35}$$

The following figure 2.12 then shows the ideal sending times of variously sized box constrained QP problems with 500 kbit/s arbitration bit rate and 2 Mbit/s payload bit rate. The figure was plotted for CAN FD frames with different data field sizes to visualize how the sending time decreases with the growing length of the data field.

As a side note, it is interesting to compare the sending times using the standard CAN bus (see figures 2.10 and 2.12) and using the CAN FD bus with 8 bytes long data field. Although the bit overhead of "non-data" bits is larger in the case of CAN FD, and one could argue that in this sense is the standard CAN more efficient, the higher payload bit rate of CAN FD still ensures that the sending times are approximately twice as fast. For larger data fields (12 bytes and more) is the CAN FD not only faster but also more efficient than the standard CAN, as shown in the table 2.1.

Data bytes [-]	8	12	16	20	24	32	48	64
CAN frame eff. [%]	0.59	-	-	-	-	-	-	-
CAN FD frame eff. [%]	0.52	0.61	0.68	0.71	0.75	0.80	0.85	0.88

Table 2.1: CAN and CAN FD frame efficiencies defined as a number of "useful" data bits divided by the number of total frame bits, note that only fixed bit stuffing is assumed

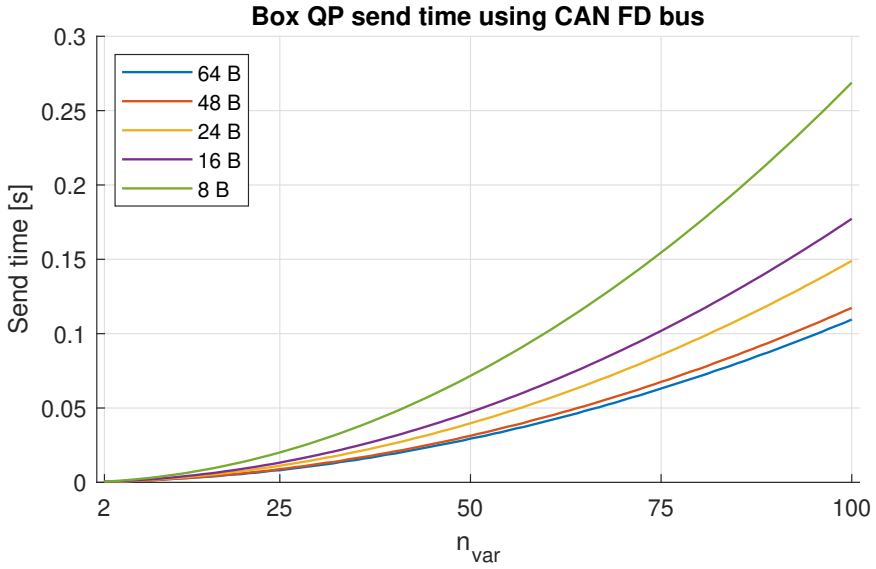


Figure 2.12: QP size and send time dependency using the CAN FD bus with various data filed sizes and bit rates of 500/2000 kbit/s

Chapter 3

Embedded solver ECU setup

With the previous chapter discussing the theoretical background of the three key topics of this thesis, this chapter will focus on the setup of the embedded QP solver. The main idea of the project is quite simple. Take some two ECUs, the first one will generate arbitrary QP problems and send them to the second ECU. The second board will then solve these problems and transmit their solution to the original sender. For the purposes of this thesis, the ECU sending the problems will be a desktop computer running Matlab & Simulink, whereas the system hosting the QP solver will be an automotive grade embedded controller described further in an upcoming section. Furthermore, the communication will be realized using the CAN and CAN FD bus. A basic overview of this setup is visualized in the following figure 3.1.

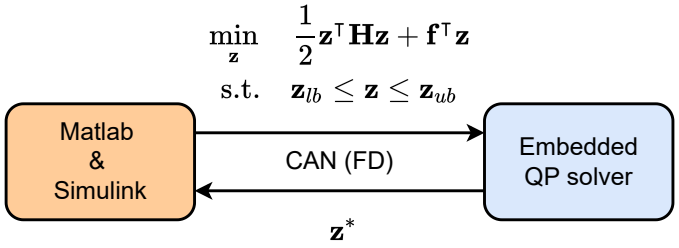


Figure 3.1: Basic overview of the embedded QP solver setup

3.1 ECU hardware

The controller used for this project is the *TC387* 32 bit single-chip microcontroller developed by Infineon Technologies™. It offers a high-performance architecture along with advanced features for connectivity, security, and functional safety. It is therefore suited for a wide range of automotive applications, including the control domain and data fusion applications. Some of the specific systems utilizing this controller are air-bag system, braking system, active suspension control, and others [23]. The main features of the controller are listed below [23]:

- Four TriCore™ cores running at 300 MHz
  - Both fixed and floating point representation supported on all cores
- 10 MB Flash memory (Error-Correcting Code protected)
- 1.5 MB static RAM memory (Error-Correcting Code protected)
- Peripherals: Ethernet, CAN (FD), FlexRay, LIN, I^2C, ...
- Up to ASIL-D/SIL3 safety requirements supported
- AUTOSAR 4.2 supported

To use the microcontroller for development purposes, a custom printed circuit board was ordered by Garrett Motion Inc. and manufactured by ADWITECH systems s.r.o. The final PCB shown in picture 3.3 hosts not only the controller itself but also a power management unit, peripheral transceivers plus connectors, and other supporting circuitry. The basic overview of this PCB is shown in the presented block diagram 3.2.

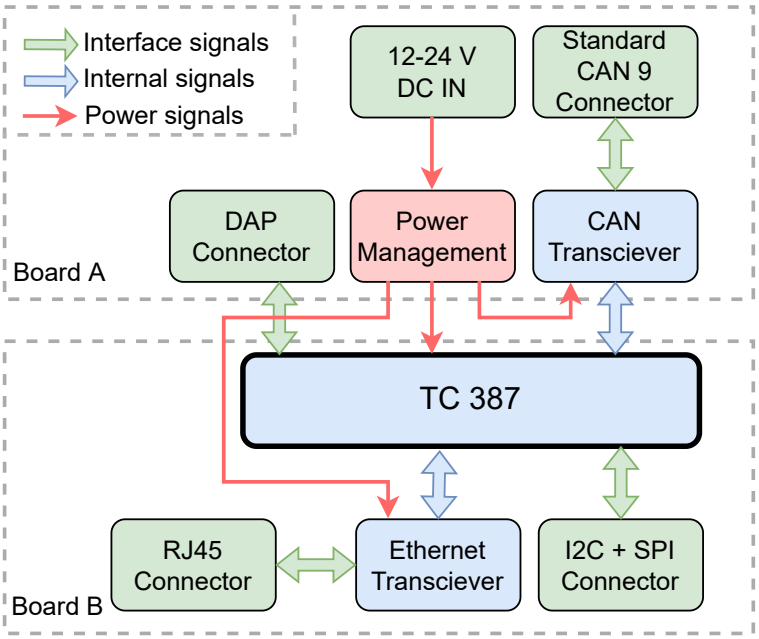
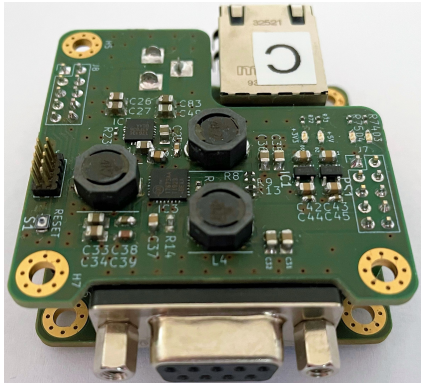
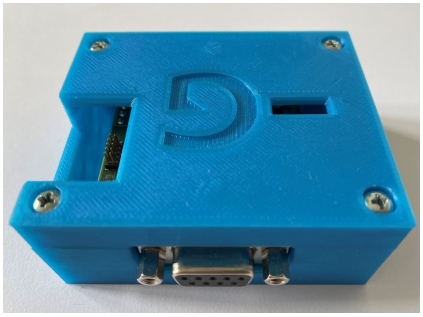


Figure 3.2: Basic overview of the printed circuit board hosting the *TC387* microcontroller, designed and developed by Garrett Motion Inc. and ADWITECH systems s.r.o.



(a) : Top view of the PCB



(b) : Custom case for the PCB

Figure 3.3: Photos of the custom made printed circuit board used for the purposes of this project

3.2 OSQP solver deployment

The first QP solver to be deployed onto the embedded system is the OSQP (Operator Splitting solver for Quadratic Programs) solver developed at the University of Oxford and published as an open-source project in 2020 under the *Apache 2.0* license. It is a general-purpose solver for convex QPs based on the alternating direction method of multipliers (ADMM). The main advantage of this solver is its robustness, where the solver algorithm places no requirements on the problem formulation. Such requirements could be the definiteness of the quadratic cost function term or the linear independence of the problem constraints as discussed in the introductory section "Convex quadratic programming" [24]. Another aspect important for the use case of this thesis is the support of the deployment of the source code onto embedded hardware. OSQP offers an embedded code generation software package that is able to generate an "embedded ready" source code from a specific QP problem formulation based on the OSQP algorithm. The generated code supports updating of the problem parameters during runtime and offers a number of configuration options to fine-tune the performance of the generated solver [25].

The inner workings of the OSQP solver will not be described in this thesis as the background of numerical optimization is quite extensive and outside the scope of this project. To see a detailed description and derivation of the OSQP algorithm, including examples and benchmarking, please refer to the original paper [24]. The solver solves a convex QP problem in the already familiar form:

$$\begin{aligned} \min_{\mathbf{z}} \quad & \frac{1}{2}\mathbf{z}^\top \mathbf{H}\mathbf{z} + \mathbf{f}^\top \mathbf{z} \\ \text{s.t.} \quad & \mathbf{a}_{lb} \leq \mathbf{A}\mathbf{z} \leq \mathbf{a}_{ub} \end{aligned} \tag{3.1}$$

Where $\mathbf{H} \in \mathbb{R}^{n \times n}$ is positive semi-definite and $\mathbf{A} \in \mathbb{R}^{m \times n}$ defines the potentially affine linear constraints. Furthermore, individual elements of $\mathbf{a}_{lb}, \mathbf{a}_{ub} \in \mathbb{R}^m$ can be defined as $\pm\infty$ [26].

3.2.1 Code generation

The source of the OSQP solver is written in C, but a number of official interfaces are supported, including Python, Julia, and Matlab. As the aim is to generate a parameterized and embedded ready C code, an interface supporting code generation will be selected. For that reason, the solver will be deployed using the Matlab interface.

Following the user documentation [26], the setup of the Matlab interface for OSQP is straightforward. Firstly, clone the official OSQP Matlab repository (available at [27]) and secondly, build the interface by calling the `make_osqp.m` function. This generates an `osqp_mex.mexw64` binary file callable from Matlab, which acts as a wrapper for the OSQP source functions. Note that both CMake™ and Matlab supported C compiler are needed for the build step. At this point is the OSQP solver fully accessible within the Matlab environment via the `osqp.m` object handle, which in itself calls the `mex` binary wrapper. As an example, a solution to a given QP problem can be obtained as follows.

```
% Init the osqp object handle
osqp_obj = osqp;
% Setup the QP problem with default solver settings
osqp_obj.setup(H, f, A, lb, ub);
% Obtain the solution
result = osqp_obj.solve();
```

Listing 3.1: Using the Matlab environment to solve a QP problem using the OSQP solver

Now the code generation feature of the Matlab OSQP package will be used to obtain a parametrized source code ready to be deployed onto the embedded controller. The final code will be library free, without dynamic memory allocation, and possibly division-free [27]. The downside of no dynamic memory is that the deployed solver will only be able to solve QP problems of predefined size, and the problem will be pre-allocated. To solve a differently sized QP, the embedded source code has to be regenerated and redeployed.

A second issue is that the solver operates over sparse $\mathbf{H}$ matrix representation, meaning that the memory pre-allocated for the hessian is also dependent on the number of zero elements within the matrix. The consequence of this is that even if another QP is of the dimension the source code was generated for unless the new hessian has an identical sparsity pattern, the deployed solver will not be able to solve such a problem. Again leading to the need for regeneration and redeployment of the solver. From a practical standpoint this

does not pose a critical issue at the moment as many applications utilizing QPs, such as linear model predictive control, operate with a fixed QP problem dimension and constant cost matrix **H**.

As for the code generation itself the Matlab OSQP package offers several options to configure the final code, the key options are:

- `'parameters'`: Specify either `'vectors'` or `'matrices'` to select whether only the vectors or both vectors and matrices of the QP problem will be parametrized during runtime
- `'FLOAT'`: Set to true to use single-precision types, false to use double-precision types
- `'LONG'`: Set to true to use `long long` integers, false to use standard integers
- `'mexname'`: Name of the executable `.mexw64` file used for calling the generated source code

Finally, to followup on the example 3.1, generating and testing the code is done in the following fashion.

```
% Generate the source code and build the mex binary
osqp_obj.codegen(target_directory, 'parameters', 'matrices', ...
    'FLOAT', true, 'LONG', false, 'mexname', 'my_osqp_mex')
% Test the code by calling the solve function
[x, y, status, iter, runtime] = my_osqp_mex('solve');
```

Listing 3.2: Using the Matlab environment to generate an embedded ready OSQP source code

This generates an embeddable library into the target directory where the problem specified during the `osqp` setup is hard-coded into the source files to ensure no dynamic memory allocation is needed.

3.2.2 Generated code deployment

In the previous subsection "Code generation" the process of obtaining an embedded friendly source code was described. Deploying the solver onto the hardware is then simply a matter of copying the generated library into the controller source project and using the well-documented C application programming interface (API) [28]. All the solver information, including solver settings, problem data, and optimization results, is stored within a `"OSQPWorkspace workspace"` C structure. This structure is generated in Matlab during the code generation phase and is defined as global. The interface also supplies all necessary functions to update and solve individual QP problems. An example of how to work with the deployed OSQP solver within the embedded application follows. Note that for the use cases of this project, neither the matrix **H** nor **A** will need to be updated for the OSQP solver during runtime.

```c
/* Include the necessary header files */
#include "osqp.h"
#include "workspace.h"

void OSQP_Update_and_Solve(){
    /* Update the individual vectors of the QP problem */
    osqp_update_lin_cost(&workspace, &f_new);
    osqp_update_lower_bound(&workspace, &lb_new);
    osqp_update_upper_bound(&workspace, &ub_new);

    /* Result available at (&workspace)->solution->x */
    osqp_solve(&workspace);
}
```

Listing 3.3: Calling the generated OSQP solver from the embedded application

To bridge the gap between the main application and the OSQP API functions, a storage structure `OSQP_storage` is created. The QP problem will be first fully defined within this storage, and only if all data is received will the OSQP workspace be updated, see figure 3.4. This is done to utilize the API update functions, which can only update the individual QP vectors all at once but, on the other hand, run a number of feasibility checks before updating the actual problem in the OSQP workspace.

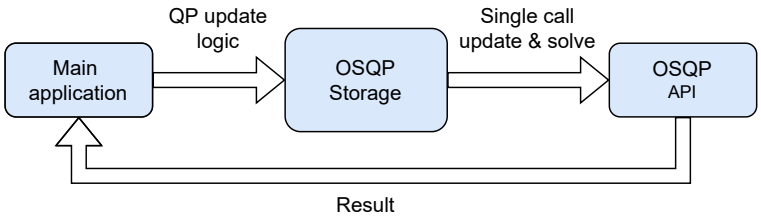


Figure 3.4: QP problem data propagation in the embedded application when using the OSQP solver

3.3 CGNP solver deployment

The second solver to be deployed onto the controller is the Combined Gradient and Newton Projection (CGNP) QP solver. Originally presented as a part of a dissertation thesis at Czech Technical University in Prague [29], its current form along with its source code are now intellectual property of Garrett Motion Inc. For that reason will this solver be treated purely as a black box function for the purposes of this project. The CGNP solver solves QPs in the box constrained form:

$$\begin{aligned} \min_{\mathbf{z}} \quad & \frac{1}{2}\mathbf{z}^\top \mathbf{H}\mathbf{z} + \mathbf{f}^\top \mathbf{z} \\ \text{s.t.} \quad & \mathbf{z}_{lb} \leq \mathbf{z} \leq \mathbf{z}_{ub} \end{aligned} \tag{3.2}$$

It was developed to be an efficient embedded solver for the use cases of model predictive control. Deploying it onto the controller is, therefore, even simpler than in the case of the OSQP solver, as the source code is by design library and dynamic memory free. The CGNP solver offers two important advantages over the OSQP solver.

Firstly, one of the input arguments of CGNP is the QP problem size. This means that as long as enough memory is pre-allocated for the solver, by simply feeding the solver information about the current problem dimensions, QP problems of varying sizes can be solved without any need for code regeneration and redeployment. Secondly, CGNP does not operate with the cost function matrix $\mathbf{H}$ in its sparse representation, and it is therefore much simpler to update not only the individual vectors of the QP problem but also the quadratic cost function term itself. Note that the second point is discussed as advantageous in this case, but the sparse representation used by OSQP is well-founded when it comes to memory efficiency.

Similarly, as in the case of OSQP, to properly implement the CGNP solver into the main embedded application, three structures will be defined. The `CGNP_U` structure will hold all the input data of the QP problem, `CGNP_Y` the output data of the QP solver, and `CGNP_tmp_mem` allocates temporary memory used by the solver during runtime, for the content of these structures see figure 3.5.

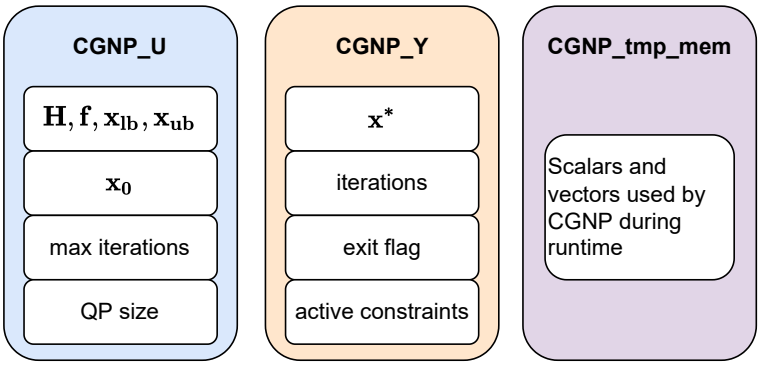


Figure 3.5: Data structures used by the CGNP solver

The following code snippet 3.4 shows a (simplified) example of how the CGNP solver will be called from the main application, taking the input structure with the problem definition and updating the output structure with the obtained solution.

```
/* Necessary includes */
#include "QPSolverLib.h"

void CGNP_solve_call(){
    /* Zeroing-out memory, resetting flags, ... */
    CGNP_initialize(&CGPN_Y, &CGNP_tmp_mem);

    /* Main solver function call */
    CGNP_Y.iterations = QPSolve(&CGPN_U, &CGPN_Y,
                                &CGNP_tmp_mem);
}
```

Listing 3.4: An example of using the CGNP solver in the embedded aplication

Chapter 4

CAN bus based interface

The previous chapter dealt with the deployment of two QP solvers onto the embedded system. The next step will be to implement a CAN bus based communication channel between a desktop computer running Matlab & Simulink and the controller. This channel will then be used to test the individual solvers and to verify their functionality. Furthermore, a baseline communication scheme will be presented, which will act as a simple proof of concept solution showing that the controller can indeed be used as an external QP solver. The following sections will deal with the definition of custom CAN messages, implementation of the CAN communication logic into the controller, and finally the development of both Matlab and Simulink interfaces used for connecting to the controller.

4.1 CAN messages definition

To begin with, the individual messages to be used for the purpose of CAN communication will be described. The workings of the CAN protocol and the complete structure of the CAN bus data frame were described previously in section "Controller Area Network". Four messages will be defined in a CAN database, each serving a different purpose in the communication scheme. The main properties of these four messages are then shown in the following table 4.1.

Name	ID format	ID decimal	DLC	Transmit node
TxSetup	11 bit	1	8	Desktop PC
TxData	11 bit	2	8	Desktop PC
ResultData	11 bit	4	8	Controller
SolverDebug	11 bit	5	8	Controller

Table 4.1: Characteristics of the defined CAN messages

Solver setup message *"TxSetup"*

The first message will be used to update tunable parameters on the side of the embedded solvers. Note, that the setup message does not need to be

transmitted in order for the system to work properly, as default settings are hard-coded within the embedded application. The contents of the message, shown in figure 4.1, are defined over the available eight payload bytes.

Byte #	1 - 4	5	6	7	8
Name	Reserve	WarmStart	MaxIter	SolverSelect	PrblmSize
Datatype	float32	uint8	uint8	uint8	uint8

Figure 4.1: Solver setup CAN message definition

Where the individual data fields have the following meaning.

- *Reserve*: Four bytes are reserved for possible future use
- *WarmStart*: Flag indicating whether the solver uses the optimal solution of the previous problem as a starting point for the new problem
- *MaxIter*: Maximum number of allowed QP solver iterations
- *SolverSelect*: Flag which selects the solver to be used for the next QP problem, 0 - CGNP, 1 - OSQP
- *PrblmSize*: Size of the next QP problem, used only by CGNP as OSQP does not support varying QP problem sizes

QP problem data message *"TxData"*

The second message is defined to transmit the actual problem data - cost function and bounds. In this case is the limitation of eight byte long payload really obvious. As an individual single-precision value occupies four bytes, the remaining four bytes will hold some additional information about the transmitted value. Therefore, by using the CAN protocol, it is possible to transmit only one numerical value per a complete CAN message. With this in mind, the structure of the message is presented in figure 4.2.

Byte #	1 - 4	5	6	7	8
Name	Data	HozlPosn	VertPosn	PrblmVar	TaskID
Datatype	float32	uint8	uint8	uint8	uint8

Figure 4.2: Problem data CAN message definition

With the following meaning of the individual parts of the payload:

- *Data*: Single-precision numerical value, part of the QP problem definition

- *HozlPosn*: Denotes the horizontal position of the numerical value within the vector or matrix defining the QP problem, for a column vector this value is always one
- *VertPosn*: Denotes the vertical position of the numerical value within the vector or matrix defining the QP problem
- *PrblmVar*: Numerical value denoting the part of the QP problem to which the numerical value corresponds, $\mathbf{H} \implies 0$, $\mathbf{f} \implies 1$, $\mathbf{x}_{lb} \implies 2$, $\mathbf{x}_{ub} \implies 3$
- *TaskID*: Value used for transmitting commands to the controller, it practically marks the final data message and signalizes the controller to calculate and transmit the result
  - 1: Update the stored QP problem with the newly received value
  - 2: Update, solve and transmit the result as soon as possible
  - 3: Update, solve, transmit the result and finally transmit the *SolverDebug* message discussed later ("Solver information message *"SolverDebug"*")

QP problem result message *"ResultData"*

The next message will be configured to transmit the result of the optimization from the controller back to the sender. It is mostly identical to the previous *TxData* message and its contents, shown in figure 4.3, will not be therefore explained in detail.

Byte #	1 - 4	5	6	7	8
Name	Data	HozlPosn	VertPosn	isLastMsg	Reserve
Datatype	float32	uint8	uint8	uint8	uint8

Figure 4.3: Result data CAN message definition

Solver information message *"SolverDebug"*

Finally the last message that needs defining will be used to obtain the solver information from the controller after a problem has been solved. Similarly as the setup message, requesting the debug solver information is optional and can be disregarded to minimize data flow over the CAN bus. The individual fields of the payload can be seen in the upcoming figure 4.4.

Byte #	1	2	3 - 6	7	8
Name	SolverStatus	Iterations	SolvTime	Reserve1	Reserve2
Datatype	uint8	uint8	float32	uint8	uint8

Figure 4.4: Solver debug information CAN message definition

Where:

- *SolverStatus*: Unsigned integer representing the status returned by either the CGNP or OSQP solver
  - CGNP: 1 - optimal solution found, 0 - suboptimal solution found
  - OSQP: 1 - optimal solution found, for the meaning of other constants please refer to one of OSQP's headers `constants.h`, note that the transmitted value is offset by 10 to transmit negative flag values via the `uint8` datatype
- *Iterations*: Number of performed solver iterations
- *SolvTime*: Time (in seconds) it took the optimization algorithm to reach its termination

4.2 Embedded application CAN interface

With the previous section describing the structure of the CAN messages that will be used for communication between the controller and Matlab, the implementation of the controller CAN interface follows. Firstly, to interact with the CAN interface, the MCMCAN software module supplied by Infineon will be used [30]. It supports both classical CAN and CAN FD and offers complete node configuration, including operation bit rate, message ID acceptance filter, and message received interrupt priority.

Decoding CAN messages

When a message with the correct ID is received, an interrupt is raised, calling a callback function. Within this function, the payload of the message is read and stored into dedicated storage consisting of two `uint32` variables. To translate this data into the individual values defined by the specific CAN message, a C union will be used. As an example, take the message described in the previous subsection "QP problem data message *"TxData"*". The individual variables of this messages will be defined within a standard structure. Furthermore, a union will be initiated where an instance of this structure will share its memory with the two `uint32` values read by the CAN node, see 4.1. Finally, by storing the received data into `canRxMsgData.uintData` the real values of the message can be accessed in `canRxMsgData.canMsgData`. Note that to transmit a CAN message, the same approach is used.

```c
/* Typedef for "TxData" CAN message*/
typedef struct canMsgDataStruct {
        float32 fData;
        uint8 HozlPosn;
        uint8 VertPosn;
        uint8 PrblmVar;
        uint8 TaskID;
} canMsgDataStructT;

/* Union to transform between raw data and specific values */
typedef union canMsgDataUni {
        canMsgDataStructT canMsgData;
        uint32 uintData[2];
} canMsgDataUniT;

/* Create an instance of the union */
canMsgDataUniT canRxMsgData;
```

Listing 4.1: Decoding a CAN message in the embedded application using C union

Application logic overview

In this stage is the embedded application able to receive, decode plus transmit CAN messages, and call the QP solvers to solve a defined problem. The logic of the final application utilizing the CAN interface is shown in the following block diagram 4.5.

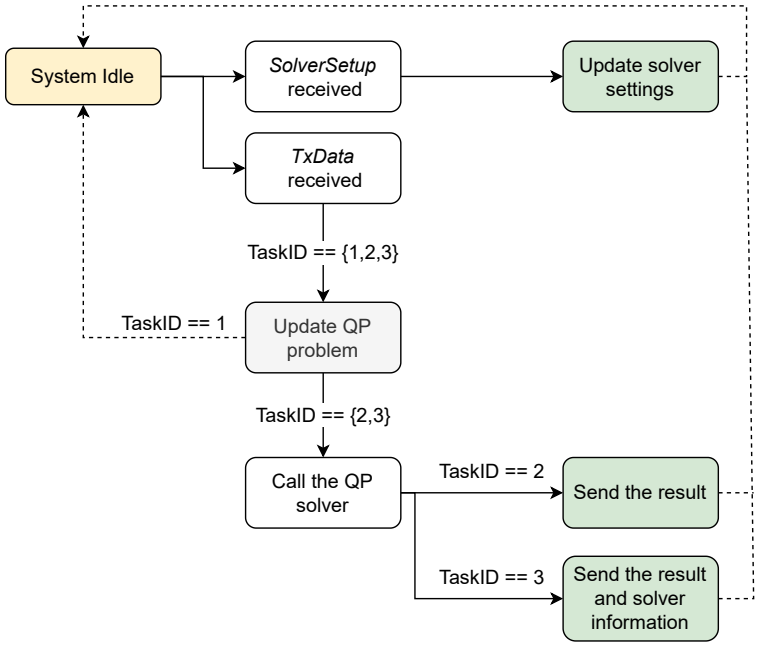


Figure 4.5: Control logic of the embedded application utilizing the CAN communication interface

4.3 Matlab & Simulink CAN interface

With the controller side of things finished, the next step is to create a Matlab & Simulink interface, which will be able to communicate via the CAN bus properly and utilize the controller as an external QP solver. To interconnect the controller with a PC, a USB to CAN converter will be used, namely *Kvaser Leaf Pro HS v2* [31]. An advantage of this device is that it is directly supported by Mathworks' Vehicle Network Toolbox, leading to straightforward utilization of Matlab's build in CAN and CAN FD functionality.

4.3.1 Matlab based CAN interface

The first step of the development process of the final interface will be to verify the workings of the embedded system by creating a simple demo purely in Matlab. This is to create an easy to debug code that will test the functionality of the controller within the Matlab environment. A generic example of how to use the developed API is shown in the following code listing.

```
% Initialize the interface object handle
mb_can = MB_CAN_sys();
% Start the CAN channel, bitrate in bit/s
mb_can.start_can_channel(bitrate);
% Optional: Update the external solver settings
mb_can.send_solver_setup(PrblmSize, SolverSelect, MaxIter, ...
    WarmStart);
% Finally, solve a QP problem
solution = mb_cab.solve_qp(PrblmSize, GetSolverDebug, H, f, lb, ...
    ub, |z_opt_ref|);
```

Listing 4.2: Calling the external QP solver via custom Matlab interface, using the CAN bus

See that the final argument `z_opt_ref` of the solve function is optional and is used to supply a reference solution to obtain a relative error between the reference solution and the solution received from the external solver. The output structure `solution` consists of the following fields, of which the meaning should be self explanatory: `'z_opt_mb'`, `'rel_err'`, `'Iterations'`, `'SolvTime'` and `'SolverStatus'`. Note that in case `GetSolverDebug` is set to zero, the solver debug message is not received from the controller, and only the solution itself is returned.

Embedded system functionality verification

To test the system, randomly generated QP problems will be used for now. When generating such a problem, two conditions must be met. Firstly, ensuring that the cost function matrix $\mathbf{H}$ is positive semi definite is necessary, and secondly, the lower and upper bounds of the box constrained QP problem ought to be feasible. A reference solution will be obtained using Matlab's

`quadprog` solver and the relative error between the reference solution $\mathbf{z}^*_{ref}$ and the received solution $\mathbf{z}^*_{mb}$ will be calculated as follows.

$$\text{Relalitve error} = \frac{\|\mathbf{z}^*_{ref} - \mathbf{z}^*_{mb}\|_2}{\|\mathbf{z}^*_{ref}\|_2} \tag{4.1}$$

To begin with, the deployed CGNP solver will be tested with no warm starting and fifty allowed iterations. One hundred random QPs of thirty four variables were sent to the controller, and the returned results are shown in the following figure 4.6. From the obtained results, it seems that the CAN interface is indeed working well, both on the side of the controller and Matlab. All problems were solved optimally as per the `SolverStatus` flag, and the relative error of the received solutions is well within tolerable bounds.

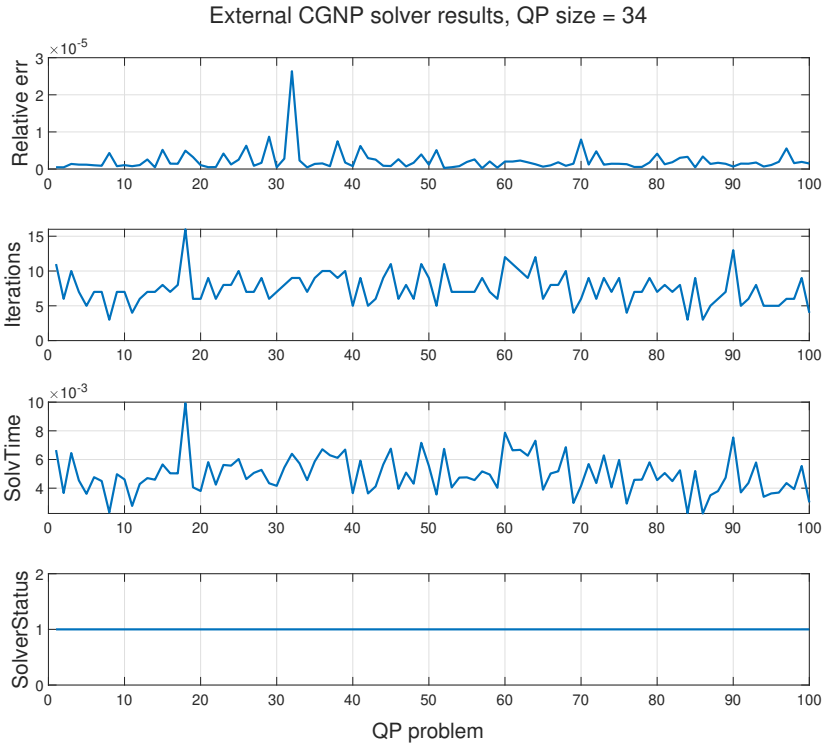


Figure 4.6: Results calculated by the CGNP solver for one hundred random QP problems of size thirty four, obtained via the Matlab CAN interface

A similar experiment will be repeated for the OSQP solver with the exception that the random problems will have constant cost matrix $\mathbf{H}$ due to reasons explained in section "OSQP solver deployment". The number of allowed iterations for the OSQP solver will also be increased to one hundred. See figure 4.7 for the results. Again, positive results were obtained as the relative error of the solutions is minimal, and the `SolverStatus` reports either "QP problem solved (12)" or "maximum iterations reached (8)".

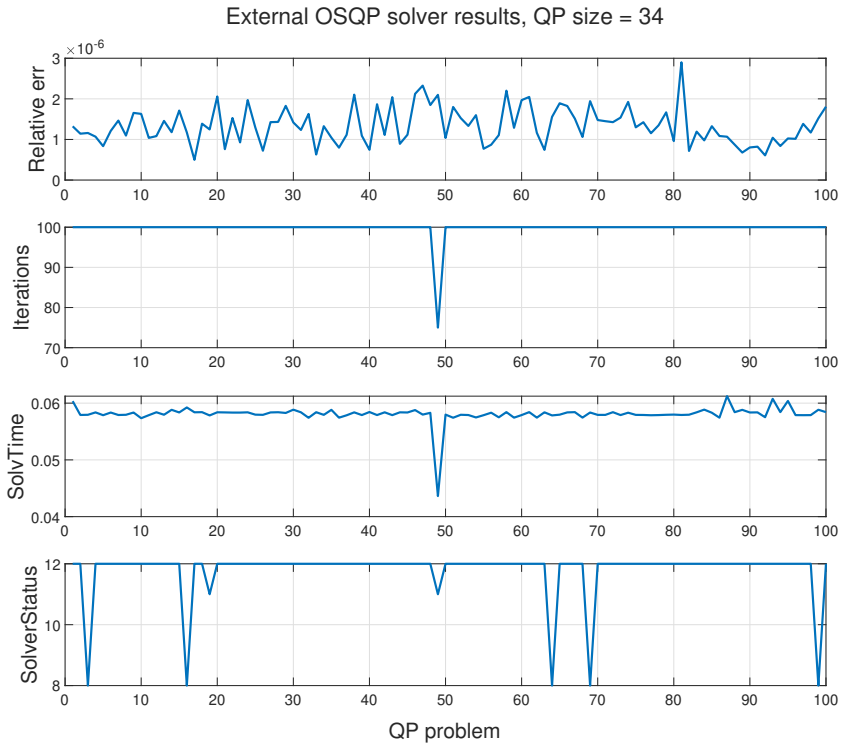


Figure 4.7: Results calculated by the OSQP solver for one hundred random QP problems of size thirty four, obtained via the Matlab CAN interface

4.3.2 Simulink based CAN interface

With the previous section verifying the functionality of the controller and the deployed QP solvers, the next step is to develop a functional plug-and-play Simulink interface that would allow the user to utilize the controller as an external QP solver. The Simulink Desktop Real-Time™ toolbox will be used for this purpose. It provides a real-time kernel for simulating Simulink models on a Windows desktop computer. Furthermore, it includes special library blocks for I/O devices allowing for CAN and CAN FD communication integration and precise timing of the message transmit/receive logic. The main idea is to develop a masked subsystem model with several inputs and outputs that would be configurable and would handle all of the communication logic. This concept is shown in 4.8.

Interface model design

The "brain" controlling this model will be a System Object™ called `MB_CAN_logic`. Simulated in discrete steps and driven by a configurable clock, this object will determine the status of the system, controlling the sending and receiving of messages, updating data indices, etc. To configure the clock period of the system, the following reasoning was used. Given a

CAN bus bit rate of 500 kbit/s and the worst-case size of the CAN data frame being 133 bits (discussed in section "QP sending time using the CAN bus"), the bus should be able to transmit a whole frame every $\sim 2.7 \times 10^{-4}$ seconds. Setting the clock period to a higher value (for example, 3×10^{-4} seconds) should therefore leave enough time for any overhead during the data transmission.

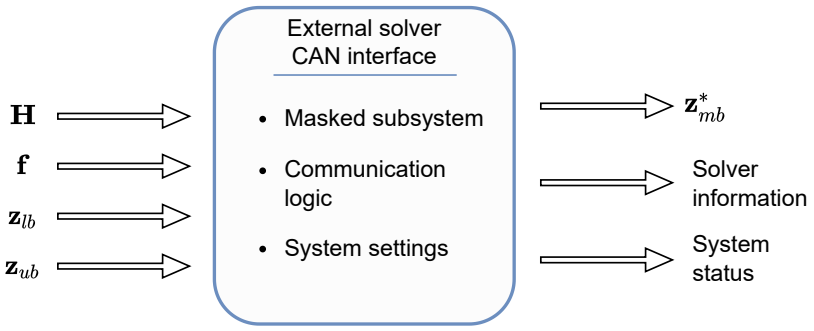
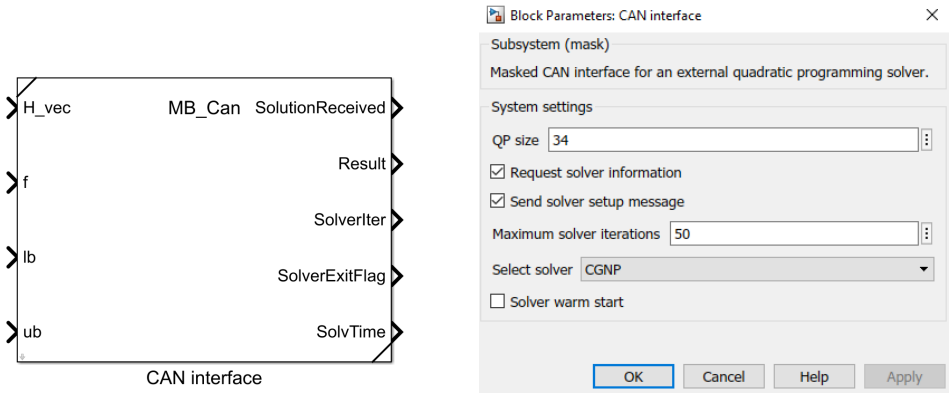


Figure 4.8: Concept of the Simulink CAN interface for the external QP solver

The functionality of the `MB_CAN_logic` system is best visualized by examining its outputs. An example can be seen in the figure 4.10 where a QP problem of 34 variables is to be solved. The most important output property is the system status (shown in the first subplot) which controls when and what part of the QP problem is currently being sent, when to expect a result and solver debug messages, or when to send the solver setup message. Furthermore, the system outputs the horizontal and vertical indices used for selecting individual numerical values from the problem's vectors and matrices. Finally, TaskID corresponds to the property of the same name described in "QP problem data message *"TxData"*".



(a) : Imported CAN interface block as seen in Simulink

(b) : Configurable block parameters

Figure 4.9: The final Simulink CAN interface for the external QP solver

See figure 4.9 for the final form of the Simulink CAN bus interface. Utilizing the aforedescribed `MB_CAN_logic` system the inner structure of the interface

is visualized in the block diagram 4.11.

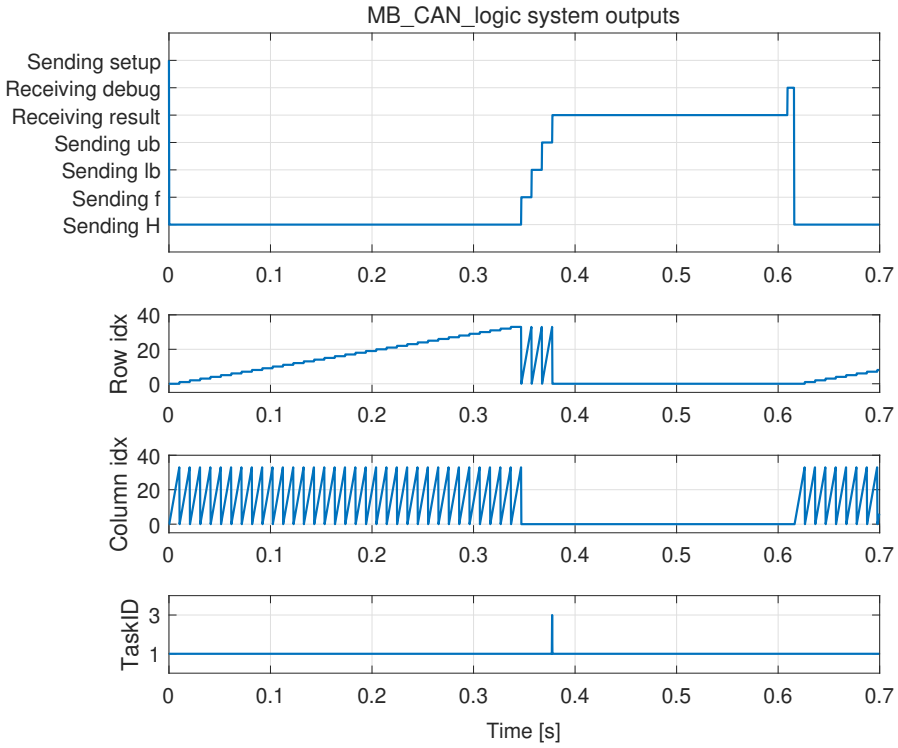


Figure 4.10: An example of outputs of the `MB_CAN_logic` Matlab System™ used as the main control block of the final CAN Simulink interface

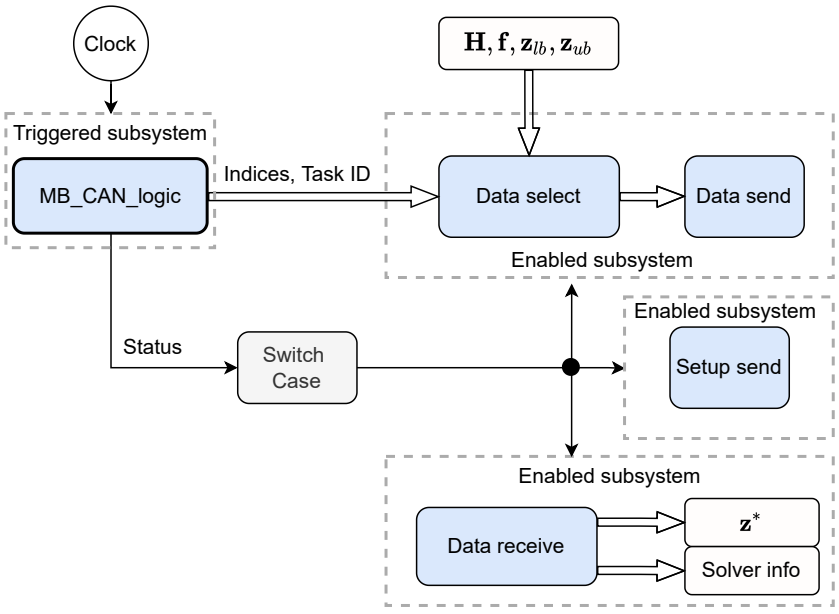


Figure 4.11: Simulink CAN interface inner logic

CAN interface testing

With the Simulink CAN interface finalized, the next step will be to test the performance of the HIL system. Similarly, as in previous subsections, one hundred random problems will be solved via the Simulink interface. The simulation is set up in such a fashion that a new problem is sent right after the solution of the previous one is received, and the problems should therefore be solved as fast as possible. The results confirming the proper workings of the system can be seen in the following figure 4.12.

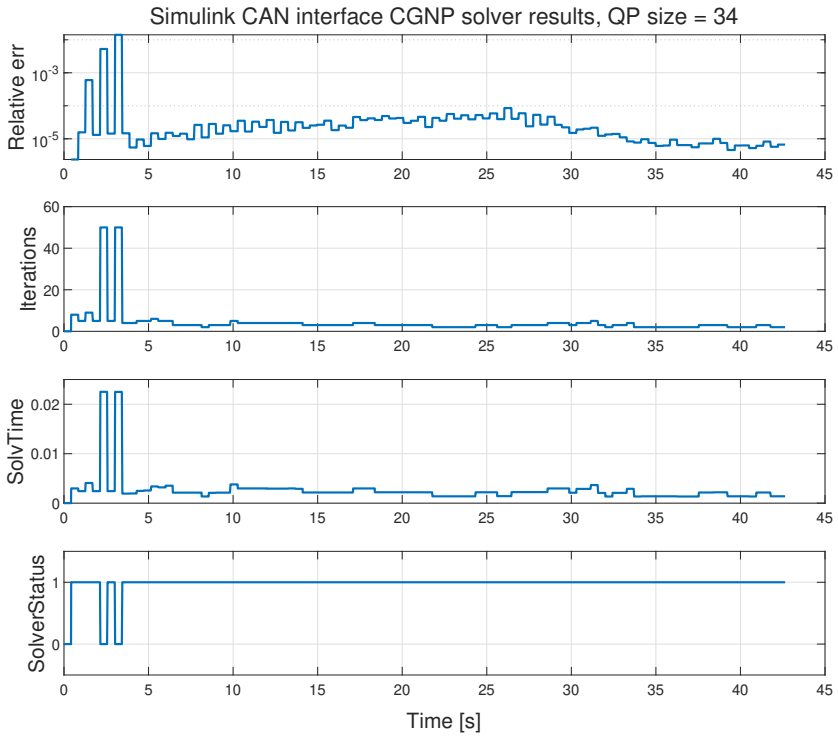


Figure 4.12: Results calculated by the CGNP solver for one hundred random QP problems of size thirty four, obtained via the Simulink CAN interface

Besides verifying the functionality of the developed interface, an experiment will be conducted to measure the time between two consecutive problems being solved - Time to Next Solution (TNS). In other words, how will the theoretical computation of the communication speed fare against the measured empirical results? To begin with, figure 4.13 shows TNS for the previous simulation 4.12. Note that to disregard the performance of the QP solvers, the time spent solving the actual problem was subtracted from these values. The average TNS measured over the one hundred problems is very close to the expected time, which was calculated using the knowledge about the number of total messages and the frequency of message transmissions.

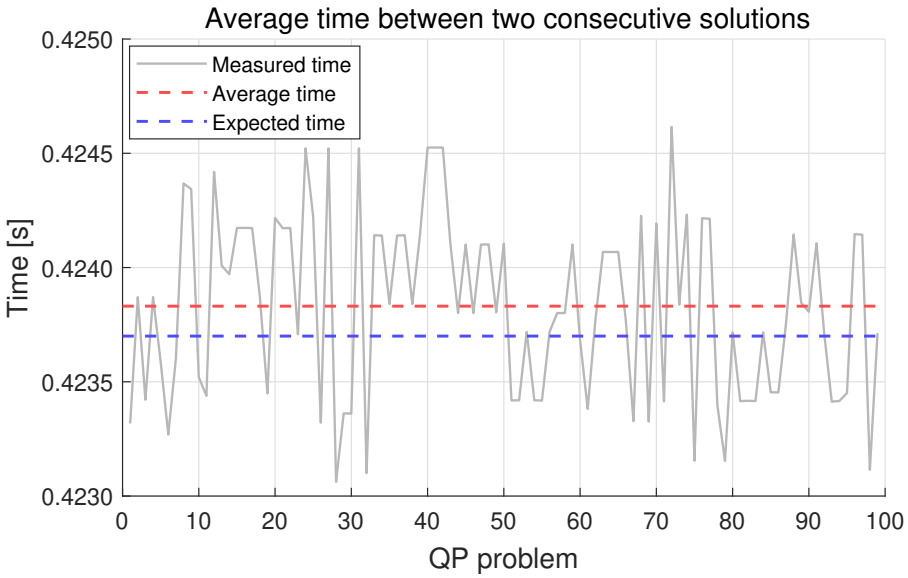


Figure 4.13: TNS for one hundred QP problems using the CAN Simulink interface, disregarding the QP solver time consumption

Similarly, the second figure 4.14 visualizes the dependency of the TNS on the size of the QP problem. With the increasing number of optimization variables, the size of data to be transmitted grows exponentially, and the usefulness of the interface decreases. For information sake, the QP solver computation time was plotted as well to see the ratio between the optimization time and the overall time it took the system to present a solution. To improve upon these results, a second interface utilizing the CAN FD communication bus will be presented in the following chapter.

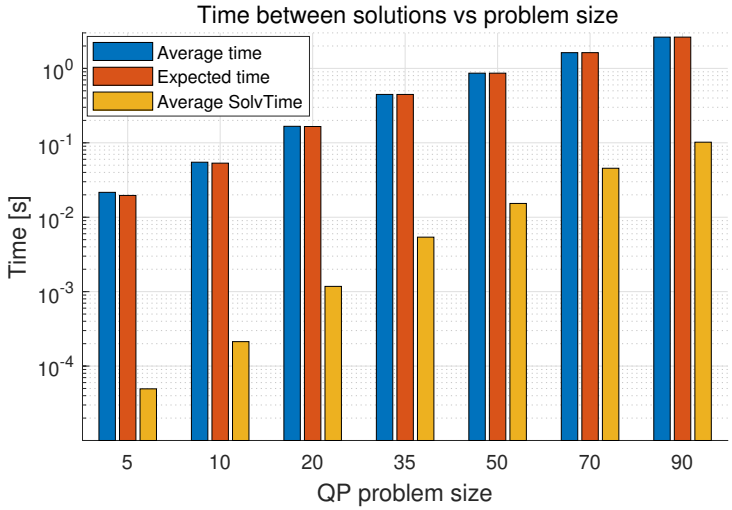


Figure 4.14: Dependency of the average TNS on the QP problem size

Chapter 5

CAN FD bus based interface

As shown in the last section,"CAN interface testing" utilizing the CAN interface for the external solver may only be feasible for small problems and/or relatively slow use cases. For fifty optimization variables, the setup took almost a second on average to return a correct solution, which may be too slow for many applications. The goal of this chapter will be to follow up on the systems developed in the previous chapter by utilizing the faster and more efficient CAN FD field bus. Furthermore, new features will be presented to speed up the system further.

5.1 Developing the CAN FD interface

As the previous chapter thoroughly presented the implementation process of the CAN interface, the development of the new CAN FD interface will be described in less detail as the approach is very similar.

CAN FD messages

Firstly, several CAN FD messages will be defined to transmit data between the controller and the desktop computer, see table 5.1.

Name	ID format	ID decimal	DLC	Transmit node
SolverSetup	29 bit	13	8	Desktop PC
SolverDebug	29 bit	21	8	Controller
FullHess	29 bit	10	15	Desktop PC
TriuHess	29 bit	12	15	Desktop PC
FullVec	29 bit	11	15	Desktop PC
Result	29 bit	20	15	Controller

Table 5.1: Characteristics of the defined CAN FD messages

The first two messages *SolverSetup* and *SolverDebug* will play the same role as in the case of the CAN communication interface. That is to update the settings of the external QP solver such as iterations and warm starting and to transmit the information provided by the QP solver after the optimization, f.e. exit flag or the number of performed iterations. The following messages

FullHess and *TriuHess* are defined for transmission of the hessian data **H**. To maximize the frame efficiency, the size of the payload is set to sixty four bytes which allows each message to transmit fourteen numerical values. The *FullHess* message is used for sending the whole matrix, i.e. all of its $n \times n$ values where n is the number of optimization variables, whereas the *TriuHess* message is configured for the transmission of only the upper triangular part of **H**. Finally, messages *FullVec* and *Result* are for transmitting the vector parts of the QP problem definition to the controller and the found solution back to the sender. Both of these messages can transmit up to fifteen individual single type values.

Matlab interface

After defining the structure of the individual messages, the next step will be the development of a Matlab interface to test the basic functionality. Similar to the CAN API, the usage of the new interface is shown in the following code snippet 5.1.

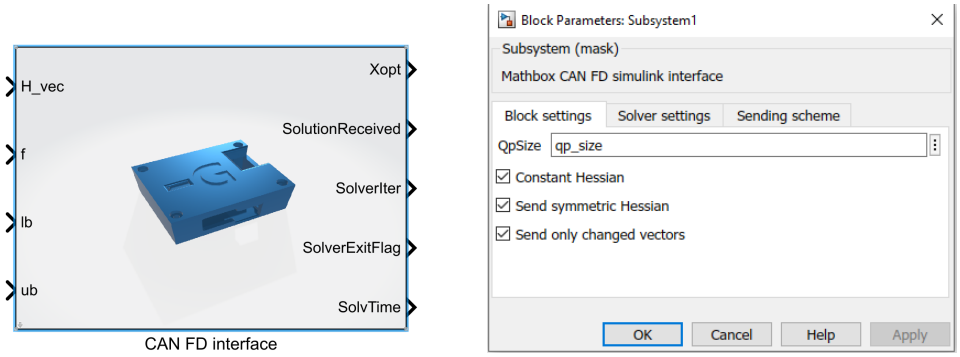
```
% Initialize the interface object handle
mb_canFD = MB_CAN_FD_sys();
% Start the CAN FD channel, bitrate (br) in bit/s
mb_canFD.start_can_channel(arbitration_br, payload_br);
% Optional: Update the external solver settings
mb_canFD.send_solver_setup(PrblmSize, SolverSelect, MaxIter, ...
    WarmStart);
% Finally, solve a QP problem
solution = mb_cabFD.solve_qp(PrblmSize, GetSolverDebug, H, f, ...
    lb, ub, |z_opt_ref|, |SendTriuHess|);
```

Listing 5.1: Calling the external QP solver via custom Matlab interface, using the CAN FD bus

See that as flexible data rate is used, specifying both the arbitration and payload bit rates is necessary for setting up the channel. Furthermore a new optional argument `SendTriuHess` was added to the solve function. It is a flag determining whether the whole matrix **H** should be sent or whether to send only its upper triangular part, which is the default option.

Simulink interface

Finally, the main focus of this chapter will be a Simulink based CAN FD interface for the external QP solver. Following the same approach as presented in the previous chapter, the final model will be a configurable block with inputs and outputs as presented in figure 4.9. Controlled by a Matlab System™ at a given frequency, this system will present a number of new features, and improvements in comparison to the baseline CAN interface, which will be discussed in following the following section.



(a) : Imported CAN FD interface block as seen in Simulink

(b) : Configurable block parameters

Figure 5.1: The final Simulink CAN FD interface for the external QP solver

For the developed model see figure 5.1. With identical inputs and outputs as the CAN interface the only difference visible to the user will be the block configuration options. Divided into three tabs these options have the following meaning.

Block settings

- `'Constant Hessian'`: When selected the system assumes a constant cost matrix **H** throughout the simulation and transmits the hessian data only once to reduce communication overhead.
- `'Send symmetric Hessian'`: When selected the interface only sends the upper triangular part of the cost matrix
- `'Send only changed vectors'`: When selected, each time a new QP problem ought to be transmitted to the external solver, the model automatically sends only the parts of the problem (matrix / vector vise) that have changed since the previous QP problem.

Solver settings

- `'Request solver debug information'`: When selected the *SolverDebug* message containing the "post optimization" information is requested by the interface.
- `'Send solver setup message'`: When selected the *SolverSetup* message is transmitted configuring the external QP solver, otherwise default settings are used to solve the QP problem.
- Furthermore, this tab allows to select the preferred QP solver, iterations and warm starting.

Sending scheme

- `'Busy send'`: The model operates in "busy" mode, meaning that each time a problem solution is received, the next problem begins to be transmitted right away. Note that this is not the preferred use case when deploying the interface into some larger model, since the user loses control over the timing of the solver interface. On the other hand, if the model is built around the interface, this option guarantees the fastest possible simulation.
- `'At input change'`: The transmission of a new problem is triggered by a change of one of the input vectors.

5.2 Testing the CAN FD interface

With the interface developed, this section will focus on the testing of its functionality and performance. To begin with, a simple sanity check will be performed. Similarly, as before, one hundred random problems of size thirty four will be solved, but instead of just comparing the solutions to their reference, a cross-check will be done to compare the controller outputs for the `'Send symmetric Hessian'` option. The expectations are that the external solver outputs will be identical regardless of these settings since only the upper triangular part of a symmetric hessian is necessary to solve the problem. Results of these experiments are shown in figure 5.2 and confirm the proper workings of both the system itself by returning a solution with minimal error and the system settings by returning identical outputs for both simulations.

However, what is not shown in the figure is the average time it takes the interface to present a solution to the current QP problem disregarding the QP solver computation time, i.e. TNS. By transmitting only $n(n+1)/2$ values of $\mathbf{H}$ instead of all n^2 values, sending the hessian should take up to 50% less time as $n \to \infty$. How this projects into the TNS reduction is shown in the figure 5.3, where the average TNS over one hundred random problems was measured. Firstly, see that for the QP problem of size five, there is no visible difference in the average times. That is due to the reason that a single CAN FD frame transmitting hessian data can carry at most fourteen values, as mentioned in "CAN FD messages". But for $n = 5$ that means transmitting two frames regardless of the `'Send symmetric Hessian'` option. On the other hand, for $n \geq 10$, the average TNS difference becomes more and more pronounced as the quantity of data contained within $\mathbf{H}$ scales quadratically with the problem size and sending the hessian data eventually takes up the majority of the communication time. To conclude, the worth of sending the upper triangular part of the cost matrix $\mathbf{H}$ only increases with the growing number of optimization variables.

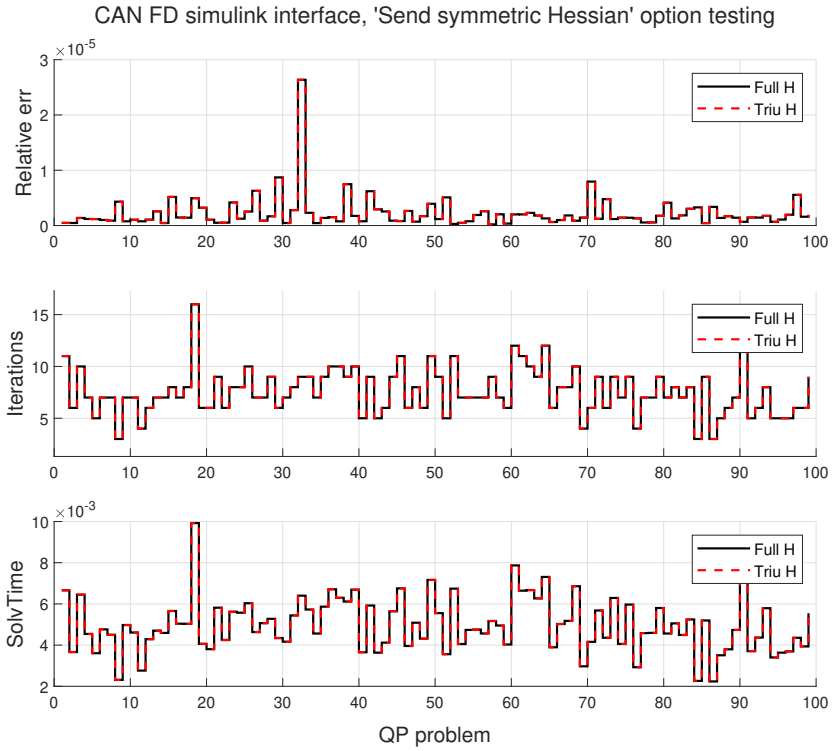


Figure 5.2: Outputs using the Simulink CAN FD interface, comparison between sending the full matrix $\mathbf{H}$ vs. sending its upper triangular part

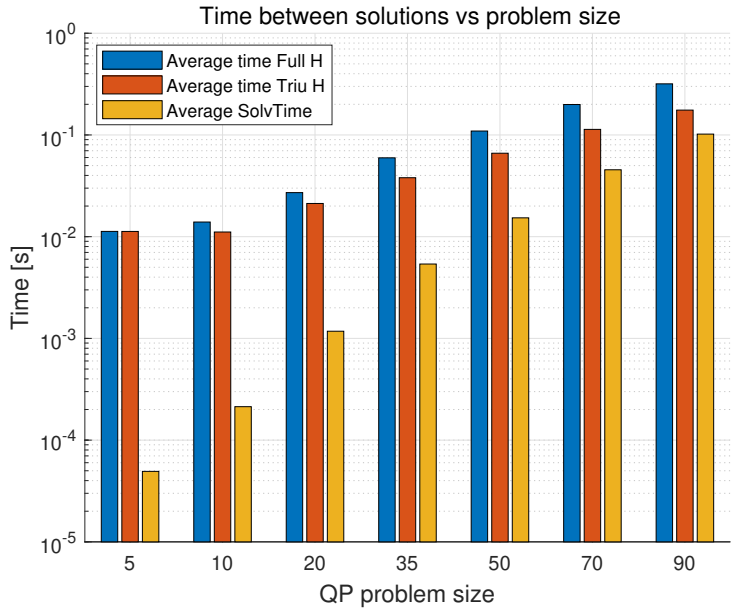


Figure 5.3: Dependency of the average TNS on the QP problem size and the `'Send symmetric Hessian'` option for the CAN FD interface, 500 kbit/s arbitration and 2 Mbit/s payload bit rate

After showcasing the benefit of not sending duplicate data contained within a symmetric matrix, what about not sending the hessian at all? This is the use-case typical for linear MPC as the hessian is constant throughout the runtime of the regulation. Whether constant **H** ought to be assumed is by the interface controllable by the `'Constant Hessian'` parameter. When selected, the interface only sends the matrix **H** data once with the first QP problem. For the following problems, only the vector parts are updated, which should significantly speed up TNS. The time reduction achieved for **H** = const. is shown in the following figure 5.5. Similarly, as before, no significant difference between the times for sizes five and ten is visible. This is again caused by the definition of the CAN FD messages and how much data each frame can carry. For these two problem dimensions, the amount of transmitted bytes are identical, which is a small disadvantage of defining the data frame to be of maximum size.

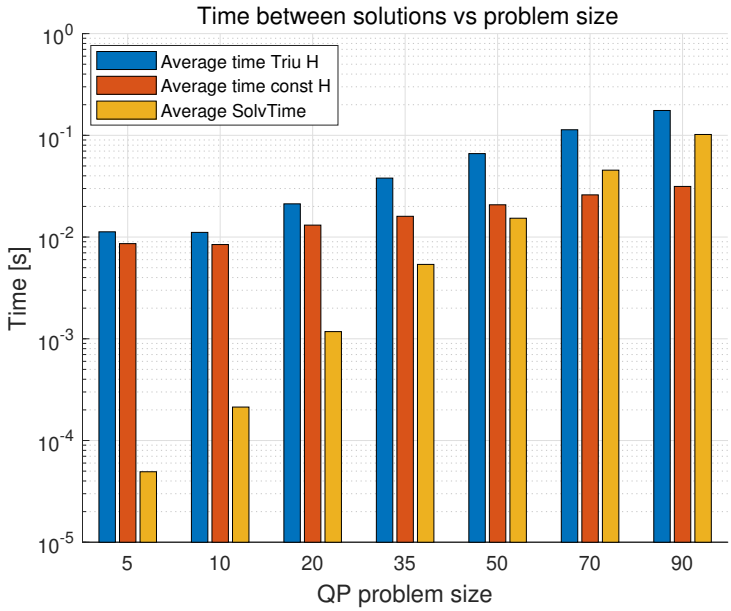


Figure 5.4: Dependency of the average TNS on the QP problem size and the `'Constant Hessian'` option for the CAN FD interface, 500 kbit/s arbitration and 2 Mbit/s payload bit rate

Finally, to further visualize how the TNS differs for the three options (full, upper triangular, and constant **H**), one more figure will be presented, see 5.5. The first subplot shows the absolute average time in seconds between two consecutive solutions plotted against the number of optimization variables. The second subplot than scales this data to show the differences between the three options relative to the slowest measurement. As an interesting result, by not having to send the hessian data, the TNS is reduced by up to an order of magnitude when compared to the baseline performance of sending the full hessian.

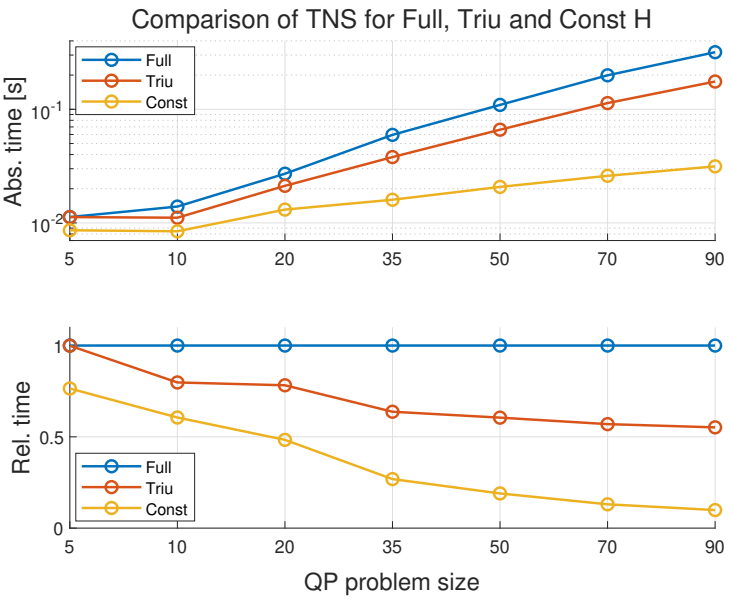


Figure 5.5: Dependency of the average TNS on the QP problem size and the `'Constant Hessian'` option for the CAN FD interface

Chapter 6

HIL MPC showcase

After finalizing the development and testing of the whole *external QP solver* framework, the final part of this thesis will focus on using the framework in an MPC control loop utilizing the Hardware-in-the-Loop (HIL) testing methodology. The approach will be to simulate the plant and construct the QP problems in the real-time Simulink model and use the CAN (FD) interface to solve these problems with the embedded solver. The obtained result will then be used as the system input in the next time step, as is typical for the "Receding horizon control". This is visualized in the following figure 6.1.

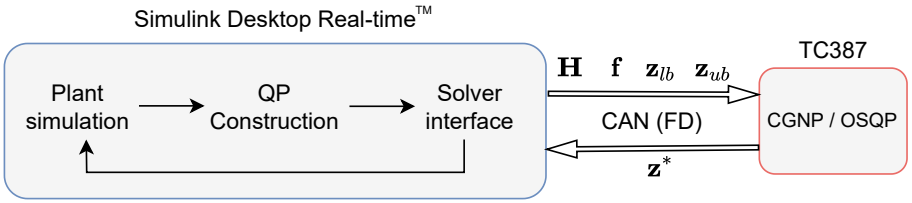


Figure 6.1: The HIL configuration used for the MPC demos

Using this configuration, two automotive related demos will be presented. The first one is based on a linear lateral dynamics vehicle model, and the second one is based on a longitudinal cruise control model.

6.1 Lateral dynamics control

To begin with, the model describing the lateral dynamics of a single track vehicle will be presented. The model is adopted from [32], for detailed derivation of the upcoming equations please see the listed source.

Consisting of two states $\mathbf{x} = [v_y, r_z]^\intercal$, where v_y is the lateral velocity and r_z the yaw rate of the vehicle, and a single input $\mathbf{u} = \delta$, where δ is the steering angle of the front wheel, the dynamics are described by the following state space model corresponding to the form presented in 2.7.

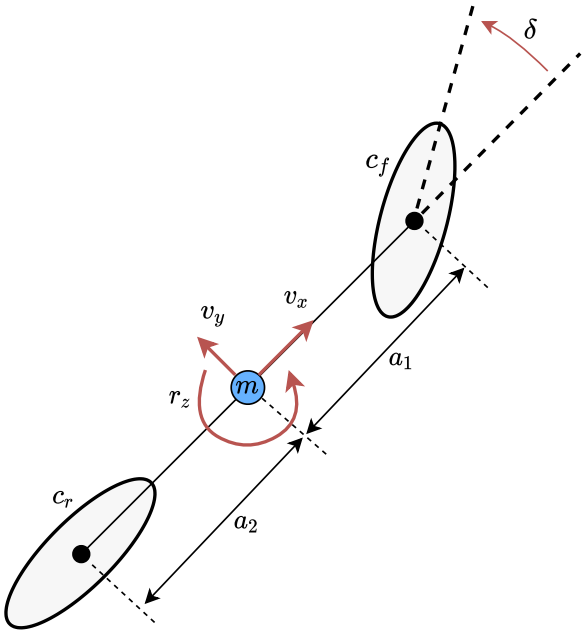


Figure 6.2: Simple single track vehicle model, adopted from [32]

$$\begin{bmatrix} \dot{v}_y(t) \\ \dot{r}_z(t) \end{bmatrix} = - \begin{bmatrix} \frac{c_r+c_f}{mv_x} & \frac{c_f a_1 - c_r a_2}{mv_x} + v_x \\ \frac{c_f a_1 - c_r a_2}{I_z v_x} & \frac{c_f a_1^2 + c_r a_2^2}{I_z v_x} \end{bmatrix} \begin{bmatrix} v_y(t) \\ r_z(t) \end{bmatrix} + \begin{bmatrix} \frac{c_f}{m} \\ \frac{c_f a_1}{I_z} \end{bmatrix} \delta(t) \tag{6.1}$$

The parameters of the plant, visualized in 6.2, have the following meaning:

- m - Vehicle mass [kg]
- v_x - Longitudinal velocity [$\mathrm{m} \cdot \mathrm{s}^{-1}$]
- I_z - Moment of inertia about the z axis [$\mathrm{kg} \cdot \mathrm{m}^2$]
- a_1, a_2 - Distances between the center of gravity and wheels [m]
- c_r, c_f - Cornering stiffness of front and rear tire [$\mathrm{N} \cdot \mathrm{rad}^{-1}$]

For the needs of this thesis is the selection of the numerical values of the parameters done arbitrarily as long as the system is stable. To see the values used for the following simulations, refer to table 6.1. The goal will be to track the lateral velocity of the vehicle by controlling the steering angle under some (hard) input constraints. The process of reformulating the generic constrained optimal control task into a box constrained QP problem was described in detail in the chapter "Theoretical analysis", specifically the section "Model predictive control". The derivation of the controller will, therefore, not be described here. Note that as the system 6.1 is a continuous time state space model, it will need to be discretized before designing the MPC controller with a sample time T_s which will be equal to the real-time HIL simulation period.

m [kg]	v_x [m · s^{-1}]	I_z [kg · m^2]	a_1, a_2 [m]	c_r, c_f [N · rad^{-1}]
1400	8	500	1.5	2000

Table 6.1: Lateral dynamics single track model parameters

The model dynamics, along with the QP construction logic, will be implemented in a `SingleTrack_sys` Matlab System™. This system will have two inputs $\Delta\delta$ and $v_{y,ref}$, several outputs, including the QP problem matrices or model outputs, and finally, several tunable parameters, including the MPC weight matrices, length of prediction horizon or initial conditions of the model. As the system only has a single input and only input bounds are assumed, the resulting size of the QP problem will be equal to the prediction horizon. For the upcoming simulations, the following parameters will be used. Model sample time $T_s = 0.1$s, prediction horizon set to $N_p = 20$, running error weight $Q = 20$, terminal error weight $P = 5$ and finally the input increment weight $R = 1000$.

The first test can be viewed as a sort of obstacle avoidance maneuver, which will force the vehicle to drift sideways. To see the reference trajectory of the lateral velocity and the performance of the MPC controller running in a real-time HIL simulation utilizing the Simulink CAN FD interface, refer to figure 6.3. As shown, the controller performs well by tracking the velocity reference and preemptively reacting to the reference changes, all whilst respecting the input bounds. The red dots marking several samples throughout the simulation will be addressed later.

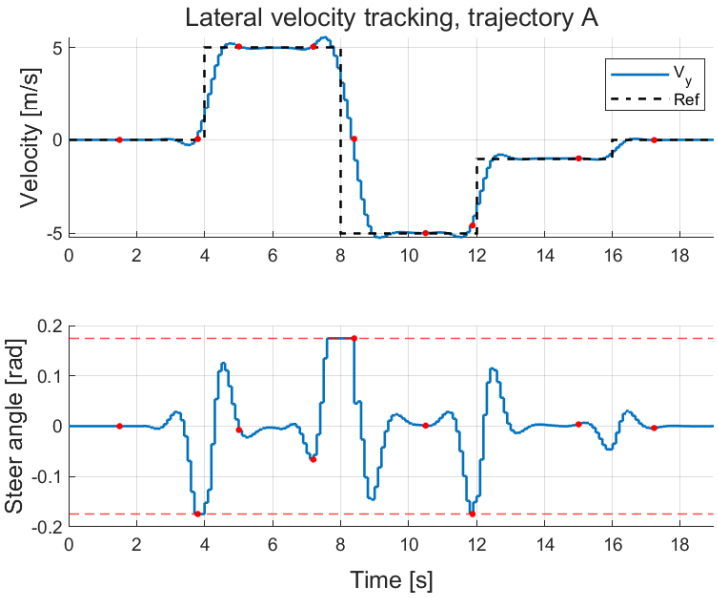


Figure 6.3: Tracking of the lateral velocity with bounded input and prediction horizon of twenty samples for reference trajectory A

See the following figure 6.4 for the information about iterations and opti-

mization times of the embedded solver during the HIL simulation. During the test, the QP solver successfully solved all of the presented QP problems and returned their solution in time for the MPC controller to apply the optimal input at each sample.

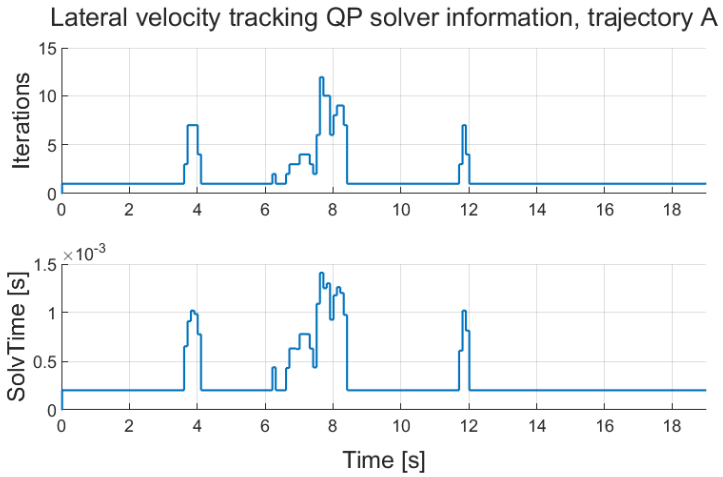


Figure 6.4: External QP solver information for the lateral velocity tracking for reference trajectory A

To further visualize the simulated maneuver, a simple figure 6.5 will be plotted, showing the trajectory of the vehicle relative to the origin, together with the heading of the vehicle. See the afore-described figure 6.3 for the times at which the vehicles were plotted. As requested by the reference trajectory, the vehicle is drifting sideways in the positive y direction and then in the negative.

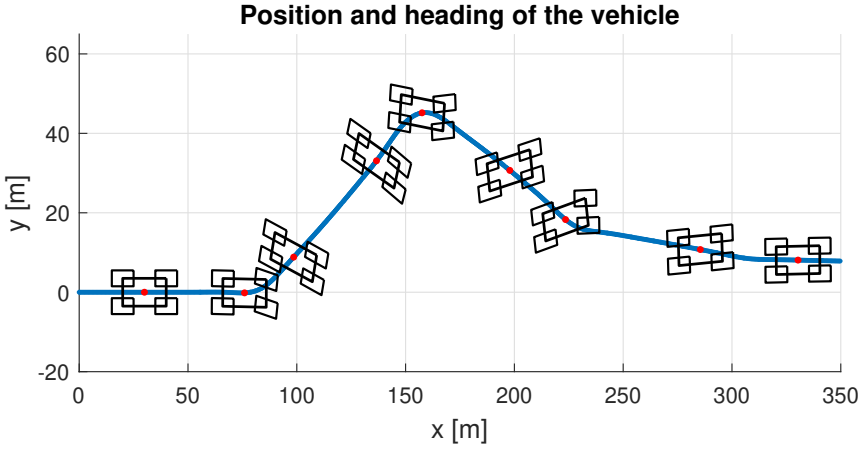


Figure 6.5: Position and heading of a single track vehicle model tracking the lateral velocity reference

Similarly, for information sake, a second simulation will be run with different reference trajectory and control input bounds. See figures 6.6 and 6.7 for the

results. Furthermore, the comparison was plotted for enabling and disabling the warm starting of the external solver. As expected, this only affects the iterations and optimization times but the input and output trajectories of the simulated model are identical in either case.

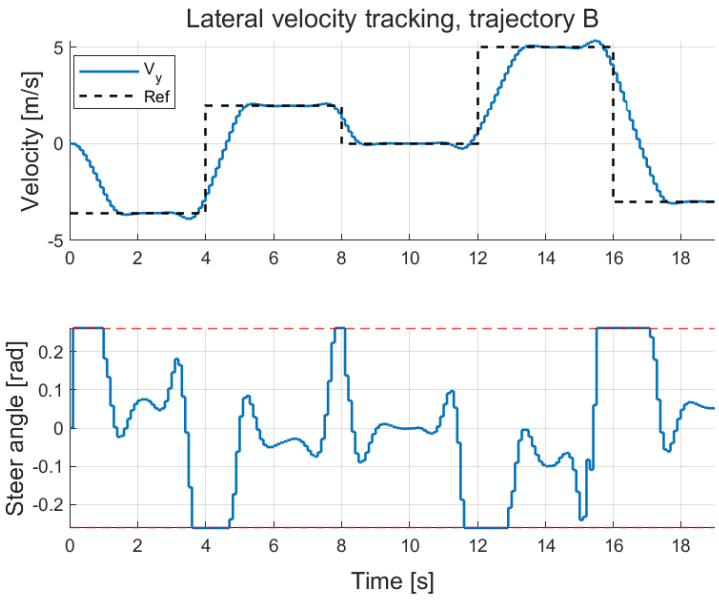


Figure 6.6: Tracking of the lateral velocity with bounded input and prediction horizon of twenty samples for reference trajectory B

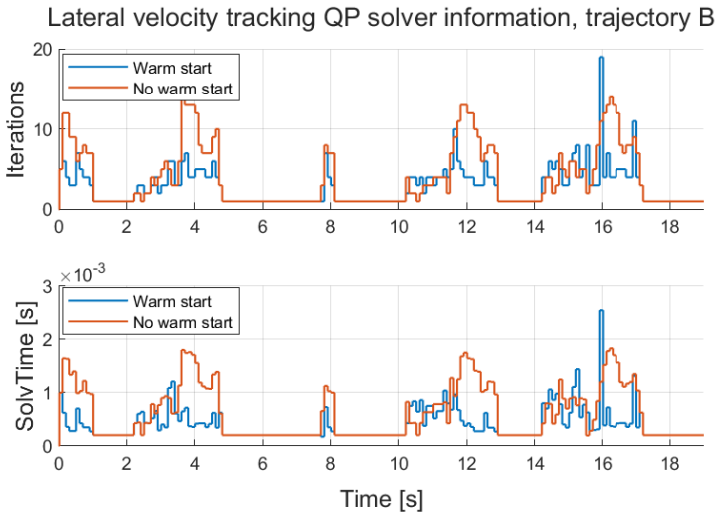


Figure 6.7: External QP solver information for the lateral velocity tracking for reference trajectory B and different settings of the QP solver warm starting option

6.2 Predictive cruise control

The second linear MPC demo will showcase the use case of a predictive cruise controller with additional unmanipulated input in the form of road grade. This time the model and QP problem construction implementation will not be within the scope of this thesis as the demo falls under Garrett Motion's intellectual property. The HIL testing approach (presented in 6.1) will still be identical as for the previous demo, only the steps of *plant simulation* and *QP construction* will be replaced with the new system. The cruise controller is based on a simplified longitudinal vehicle model adopted from [33]. For the logic behind obtaining the following equation, please refer to the original paper. For a small angle ϕ can the longitudinal vehicle dynamics be described as follows.

$$\frac{dv(T, v, \phi)}{dt} = \beta_1 T + \beta_2 v^2 + \beta_3 \phi + \beta_4 \tag{6.2}$$

Where T is the torque applied directly to the driven wheels, v the longitudinal velocity of the vehicle, ϕ the road grade and the parameters β_i are estimated to represent a given vehicle. To obtain the discrete time state spaced model, the equation 6.2 will firstly be linearized in the operating point $v_{op} = 15\text{ms}^{-1}$, $\phi_{op} = 0\%$ and T_{op} given for steady state, and secondly discretized for $T_s = 0.2\text{s}$.

As for the MPC setup itself, to formulate the QP problems during the simulation, Garrett Motion's framework called `QP Builder` will be used. By specifying the MPC weights, reference trajectories, and individual input/output limits, the framework will build the box constrained QP problems based on these specifications. Like previously for the "Lateral dynamics control", bounds will be introduced to limit the maximal and minimal input. Furthermore, the tracked velocity will be (soft) constrained as well to define some region of tolerance for the controller. The framework also allows to define input blocking, described in section "Control horizon and input blocking" as well as output error evaluation points (output error is evaluated only at specific samples in the prediction horizon) to reduce the final QP problem dimensionality.

Finally, two MPC simulation results running in HIL mode will be presented. In both cases was the length of the prediction horizon set to $N_p = 50$, and the problem was defined such that the QP problem had $n = 53$ optimization variables. See figure 6.8 for the first controller performance. In this case were the weights configured such that the tracking error was of the highest importance. The actual vehicle velocity matched the reference whenever possible and has remained within the constrained bounds even with the reduced torque limit around the 20 seconds mark. Subsequently, figure 6.9 showcases the same maneuver with slightly different MPC weight settings. In this case was the focus more on minimizing the torque input which comes at the cost of imperfect velocity tracking. On the other hand, the actual

velocity still stays within the user defined bounds except for the 25 second mark, where the lower bound is slightly violated. This can happen however, as the optimization is formulated as a box constrained QP, and the output limits are therefore softened.

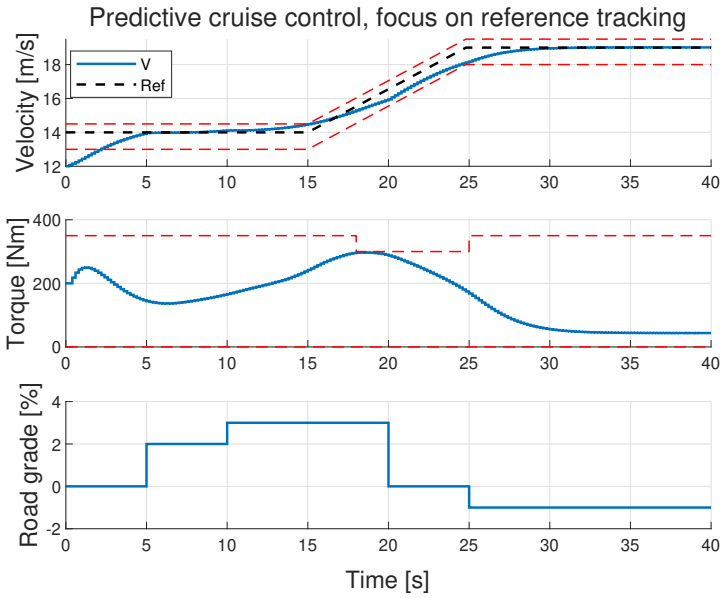


Figure 6.8: Predictive cruise control HIL simulation with a prediction horizon of length fifty and focus on minimizing the tracking error

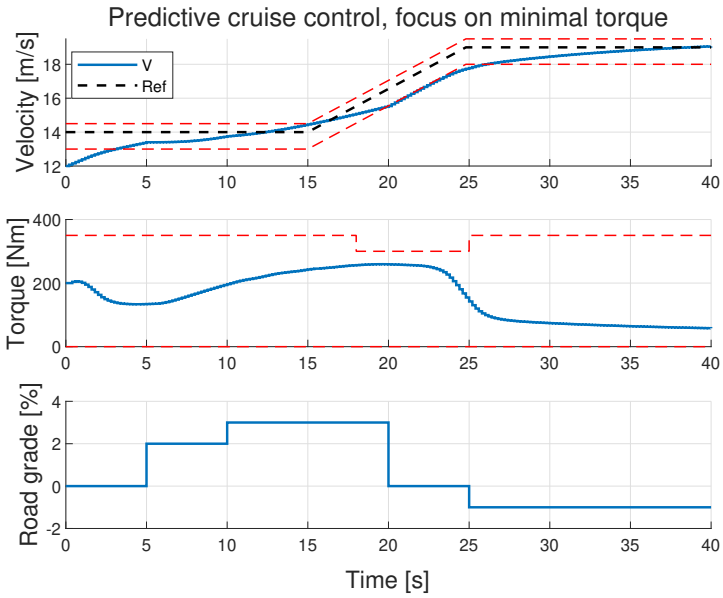


Figure 6.9: Predictive cruise control HIL simulation with a prediction horizon of length fifty and focus on minimizing the torque input

Chapter 7

Conclusion

With its current heading, the automotive industry is predicted to undergo large-scale changes regarding its software and electronic/electrical (E/E) architectures. Driven by several trends, such as e-mobility, autonomous driving, and connected vehicle(s), the E/E architecture is evolving from a distributed system into a more domain centralized and eventually vehicle centralized architecture [34]. In this spirit, this thesis presented the development of an *external QP solver* framework, which can be utilized by model based predictive controllers in the automotive setting. The main goals of this thesis were to deploy an embeddable QP solver onto a controller and develop a Matlab & Simulink based interface for this solver by utilizing an automotive grade communication bus.

To begin with, the topic of quadratic programming and its use in linear MPC was presented. This was followed by a discussion on topic of automotive field buses, specifically the LIN, CAN and CAN FD buses. Their basic concepts were presented along with their advantages and disadvantages for the use case of transmitting QP definitions. The outcome of this section was the decision to utilize CAN and CAN FD communication for this project.

After analyzing the problem from a theoretical standpoint, the next step was to deploy two QP solvers onto the selected AURIX™ controller. Firstly, the open-source OSQP solver, which posed the issue of not being able to solve varying-sized QP problems without redeployment, and secondly, the CGNP solver. Furthermore, the rest of the embedded application was developed to handle CAN(FD) communication, memory management, and solver handling. With the embedded side of things finalized, the next step was the development of the Matlab & Simulink interfaces. Initially, the CAN based interface was presented, but due to the inherited communication speed limitations, its main goal was to verify the functionality of the deployed solvers and the embedded application. Further, a second CAN FD based interface was developed. Several measurements were performed to showcase the dependency of expected communication times on the QP problem size and a number of user tunable options. These results were positive and showed that with the correct configuration, this framework could solve QP problems at a relatively high frequency. This CAN FD external solver interface built for Simulink

Desktop Real-Time™ was the main product of this part of the project.

Finally, to display the framework's capabilities, two linear MPC demos in the HIL configuration were presented. The first demo, based on a simple single track vehicle model, utilized MPC to track the lateral velocity of the vehicle with the steering angle as a bounded input. The formulated QP problems, while of smaller dimension, were solved without any issues and therefore the controller performed as expected. The second demo, predictive cruise control, introduced another MPC related features, such as input blocking and output limits. Again, the external solver framework worked well, and the MPC controller managed to track the reference trajectories.

To conclude, I believe that the goal of developing and implementing a Simulink based framework, which will allow the user to solve quadratic problems on embedded hardware using an automotive grade communication bus, was achieved.

Future work

As for the future steps, which would follow up on this thesis, three main goals come to mind. Firstly, modify the embedded application such that QP problem reception from multiple nodes is supported, i.e. solve QP problems for several controllers. Another task would be to implement automotive Ethernet based communication to increase the communication bandwidth further. Finally, while the topic of this thesis was to develop an *external QP solver* framework, the controller could support any other number of control related functions, such as filtration or state estimation.

Bibliography

[1] GlobeNewswire by notified (2021, June). Garrett Motion Launches Predictive Control Software with Hyundai Motor Company. Garrett Motion Inc. Retrieved May 8, 2022, from https://www.globenewswire.com/news-release/2021/06/17/2248951/0/en/Garrett-Motion-Launches-Predictive-Control-Software-with-Hyundai-Motor-Company.html

[2] J. Nocedal and S. Wright (2006). Numerical Optimization, 2nd edition. New York: Springer

[3] Cottle, R.W., Infanger, G. (2010). Harry Markowitz and the Early History of Quadratic Programming. In: Guerard, J.B. (eds) Handbook of Portfolio Construction. Springer, Boston, MA. Retrieved May 2, 2022, from https://doi.org/10.1007/978-0-387-77439-8_8

[4] Qin, Joe & Badgwell, Thomas. (1997). An Overview Of Industrial Model Predictive Control Technology. AIChE Symposium Series. 93

[5] Bemporad, A. (2022, March). Linear MPC. Model predictive control. Retrieved May 2, 2022, from http://cse.lab.imtlucca.it/~bemporad/mpc_course.html

[6] J. B. Rawlings, D. Q. Mayne, M. M. Diehl (2019). Model predictive control: theory, computation and design, 2nd ed., Nob Hill Pub

[7] Z. Hurák (2021, March). Discrete-time optimal control - direct approach. Lecture 3 on Optimal and Robust Control at CTU in Prague. Retreived May 2, 2022, from https://moodle.fel.cvut.cz/course/view.php?id=5716

[8] Z. Hurák (2017, March). Introduction to Model Predictive Control (MPC) - reference tracking. Lecture 3 on Optimal and Robust Control at CTU in Prague. Retreived May 2, 2022, from https://www.youtube.com/watch?v=GnFaLl7qwco&t=2s&ab_channel=aa4cc

[9] Alberto Bemporad, Manfred Morari, Vivek Dua, Efstratios N. Pistikopoulos (2002). The explicit linear quadratic regulator for constrained systems.Automatica, Volume 38, Pages 3-20. Retreived May 2, 2022, from

http://cse.lab.imtlucca.it/~bemporad/publications/papers/automatica-mpqp.pdf

[10] Dale E. Seborg, Thomas F. Edgar, Duncan A. Mellichamp, Francis J. Doyle III (2016, September). Process Dynamics and Control. Chapter 20, Model Predictive Control. Section 20.4.2, MPC with Inequality Constraints. Retreived May 2, 2022, from https://folk.ntnu.no/skoge/vgprosessregulering/papers-pensum/seborg-c20ModelPredictiveControl.pdf

[11] Nolte, Thomas & Hansson, Hans & Lo Bello, Lucia. (2005). Automotive communications-past, current and future. Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on. 10.1109/ETFA.2005.1612631.

[12] Correa Colt (2020, September). Automotive Networking Technologies. Intrepid Control Systems presentation. Retreived May 2, 2022, from https://cdn.intrepidcs.net/events/Webinars/Automotive_Networking_Technologies_20200910.pdf

[13] CSS Electronics (2022). LIN Bus Explained - A Simple Intro [2022]. Retreived May 2, 2022, from https://www.csselectronics.com/pages/lin-bus-protocol-intro-basics

[14] Rohit Mathur, Ritesh Saraswat, Gunjan Mathur (2014). An Analytical Study of Communication Protocols Used in Automotive Industry, IJERT ETRASCT – 2014 (Volume 2 – Issue 03)

[15] W. Voss (2018, November). Controller Area Network (CAN Bus) - Bus Arbitration. Copperhill technologies. Retreived May 2, 2022, from https://copperhilltech.com/blog/controller-area-network-can-bus-bus-arbitration/

[16] CSS Electronics (2022). CAN Bus Explained - A Simple Intro [2022]. Retreived May 2, 2022, from https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial#top-4-benefits-can

[17] W. Voss (2020, April). The Benefits Of Controller Area Network - CAN Bus - For Embedded Application. Copperhill technologies. Retreived May 2, 2022, from https://copperhilltech.com/blog/the-benefits-of-controller-area-network-can-bus-for-embedded-applications/

[18] S. Corrigan (2002, August). Introduction to the Controller Area Network (CAN). Texas Instruments Application report. Retreived May 2, 2022, from https://www.ti.com/lit/an/sloa101b/sloa101b.pdf?ts=1647764164724&ref_url=https%253A%252F%252Fwww.google.com%252F

[19] CSS Electronics (2022). CAN FD Bus Explained - A Simple Intro [2022]. Retreived May 2, 2022, from https://www.csselectronics.com/pages/can-fd-flexible-data-rate-intro

[20] Bosch (2012, April). CAN with Flexible Data-Rate. Specification version 1.0

[21] Mach Systems (2015, December). CAN FD - nová verze CAN protokolu. Retreived May 2, 2022, from https://www.machsystems.cz/novinky/2015/can-fd-nova-verze-can-protokolu

[22] Kvaser (2018, April). Kvaser CAN Protocol Course: CAN Error Handling (Part 8). Retreived May 2, 2022, from https://www.kvaser.com/lesson/can-error-handling/

[23] Infineon Technologies. AURIX™ Family – TC38xQP. Retreived May 2, 2022, from https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc3xx/aurix-family-tc38xqp

[24] Stellato, B. and Banjac, G. and Goulart, P. and Bemporad, A. and Boyd, S., "OSQP: an operator splitting solver for quadratic programs", Mathematical Programming Computation, volume 12, 2020

[25] Banjac, G. and Stellato, B. and Moehle, N. and Goulart, P. and Bemporad, A. and Boyd, S., "Embedded code generation using the OSQP solver", IEEE Conference on Decision and Control (CDC), 2017

[26] Bartolomeo Stellato, Goran Banjac (2021). OSQP User documentation - The solver. University of Oxford. Retreived May 2, 2022, from https://osqp.org/docs/solver/index.html

[27] Bartolomeo Stellato, Goran Banjac (2021). OSQP User documentation - Code generation. University of Oxford. Retreived May 2, 2022, from https://osqp.org/docs/codegen/index.html

[28] Bartolomeo Stellato, Goran Banjac (2021). OSQP User documentation - Interfaces - Official - C - Main solver API. University of Oxford. Retreived May 2, 2022, from https://osqp.org/docs/interfaces/C.html#sublevel-api

[29] O. Šantin (2016, August). Numerical Algorithms of Quadratic Programming for Model Predictive Control. CTU in Prague. Retreived May 2, 2022, https://dspace.cvut.cz/handle/10467/65525

[30] Infineon Technologies. MCMCAN_1 for KIT_AURIX_TC397_TFT MCMCAN data transmission. AURIX™ TC3xx Microcontroller Training, presentation. Retreived May 2, 2022 https://www.infineon.com/dgdl/Infineon-AURIX_TC3xx_MCMCAN_1_KIT_TC397_TFT-Training-v01_00-EN.pdf?fileId=5546d46274cf54d50174da20a6242215

[31] Kvaser. Kvaser Leaf Pro HS v2. Retreived May 2, 2022, https://www.kvaser.com/product/kvaser-leaf-pro-hs-v2/

[32] The F1 Clan (2020, December). Vehicle dynamics: The dynamic vehicle model. Retreived May 2, 2022, https://thef1clan.com/2020/12/23/vehicle-dynamics-the-dynamic-bicycle-model/

[33] Santin, O., Pekar, J., Beran, J., D'Amato, A. et al. (2016). Cruise Controller with Fuel Optimization Based on Adaptive Nonlinear Predictive Control. SAE Int. J. Passeng. Cars – Electron. Electr. Syst. 9(2):262-274

[34] Burkacky, O., Deichmann, J., Stein, J.P., (2019). Automotive software and electronics 2030 - Mapping the sector's future landscape. McKinsey & Company. Retreived May 8, 2022, https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/mapping-the-automotive-software-and-electronics-landscape-through-2030