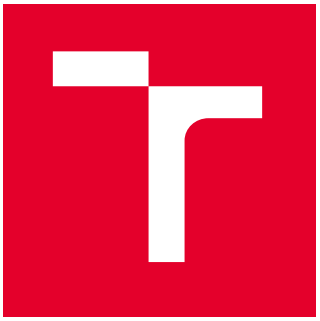

BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF CONTROL AND INSTRUMENTATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

PARALLELIZATION IN NONLINEAR MODEL PREDICTIVE CONTROL OF SYNCHRONOUS MOTOR DRIVES

VYUŽITÍ PARALELIZACE V NELINEÁRNÍM PREDIKTIVNÍM ŘÍZENÍ POHONU SE SYNCHRONNÍM MOTOREM

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. Jan Sedlář

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. Michal Kozubík

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study program **Cybernetics, Control and Measurements**

Department of Control and Instrumentation

Student: Bc. Jan Sedlář ***ID:*** 228521

Year of study: 2 ***Academic year:*** 2024/25

TITLE OF THESIS:

Parallelization in Nonlinear Model Predictive Control of Synchronous Motor Drives

INSTRUCTION:

1. Study the principles of NMPC and optimization algorithms.
2. Design a parallelized optimization algorithm.
3. Use the algorithm in the NMPC controller.
4. Verify the behavior of the motor controlled by the proposed NMPC controller through simulations.
5. Evaluate the use of the parallelized optimization algorithm in NMPC regarding control quality, computation time, and other relevant factors.

RECOMMENDED LITERATURE:

Grüne, Lars, et al. Nonlinear model predictive control. Springer International Publishing, 2017.

Date of project specification: 10.2.2025 ***Deadline for submission:*** 21.5.2025

Supervisor: Ing. Michal Kozubík

doc. Ing. Petr Fiedler, Ph.D.
Chair of study program board

ABSTRACT

This thesis addresses the challenge of utilizing the nonlinear model predictive control (NMPC) for permanent magnet synchronous motors (PMSM) by proposing a novel approach that exploits parallel computing and reduces the complexity of the control problem. The algorithm is optimization-based and utilizes the full nonlinear model of the motor while adhering to the imposed constraints. The controller was implemented on a graphics processing unit (GPU).

The proposed approach initially identifies and eliminates parts of the optimization function that are rendered unnecessary during the offline preparation of the NMPC. The robustness of the control is preserved, but the computational burden is significantly reduced. A dedicated optimization solver was developed to perform the subsequent optimization task in parallel to further exploit the capabilities of the GPU.

The control performance and real-time potential of the solution were demonstrated and further addressed in the thesis. The underlying principles of computational complexity reduction and parallelized optimization were also validated.

KEYWORDS

nonlinear control, model predictive control, parallelism in optimization, optimization solver, problem complexity reduction, PMSM, IPMSM, GPU

ABSTRAKT

Práce se zaměřuje na problém využití nelineárního prediktivního řízení (NMPC) synchronního motoru s permanentními magnety (PMSM). Je představeno inovativní řešení problému pomocí paralelizace řídicího procesu a snížení jeho výpočetní složitosti. Navržený algoritmus je založen na optimalizacích a využívá nezjednodušený nelineární model synchronního motoru s fyzikálními omezeními. Řídicí algoritmus byl implementován na grafické kartě (GPU).

Navržený přístup v offline přípravách regulátoru nalezne a eliminuje části optimalizační funkce, které není třeba počítat. Části jsou redukovány tak, aby bylo zachována přesnost a robustnost řídicího algoritmu, ale zároveň snížena jeho výpočetní náročnost. Vlastní solver byl vyvinut na míru pro tento otimalizační problém. Solver využívá paralelních výpočtů dostupných na GPU pro maximální zrychlení doby výpočtu.

Vlastnosti regulátoru a jeho real-time potenciál byly v práci demonstrovány a diskutovány, stejně jako byly ověřeny další specifické vlastnosti navrženého paralelizovaného algoritmu.

KLÍČOVÁ SLOVA

nelineární řízení, prediktivní řízení, paralelizmus v optimalizacích, optimalizační solver, snížení komplexity, PMSM, IPMSM, GPU

Rozšířený abstrakt

Úvod

Synchronní motory s permanentními magnety (PMSM) jsou díky vysoké účinnosti, výkonové hustotě a širokým možnostem řízení klíčovou součástí současné průmyslové automatizace, vyskytují se v elektromobilovém průmyslu i v oblastech obnovitelných zdrojů. [26] Jejich nelineární dynamika spolu s nutnými omezeními statorových proudů a napětí však klade na regulátor vysoké nároky – zvláště pokud má pracovat s velmi krátkou vzorkovací periodou a v plném rozsahu rychlostí včetně režimu oslabování pole (FW). [53]

Nelineární prediktivní řízení (NMPC) dokáže náročné požadavky na řízení splnit, protože v optimalizační úloze uvažuje úplný (nelineární) model motoru i všechny omezující podmínky. V praxi ale bývá problémem výpočetní náročnost celého algoritmu – klasické solvery nezvládnou během několika desítek mikrosekund najít řešení nelineární nekonvexní optimalizační úlohy, na které je řídicí algoritmus postaven. Cílem diplomové práce je odstranit tuto překážku a umožnit reálné nasazení NMPC pro pohony s rychlými dynamikami pomocí implementace paralelizovaného optimalizačního algoritmu běžícího na GPU a současnou redukcí složitosti úlohy.

Navržené řešení

Ze spojitého stavového popisu PMSM je Eulerovou diskretizací odvozen diskrétní model se vzorkovací dobou $t_s = 200\,\mu$s. Všechny stavy i vstupy jsou předem normalizovány na rozsah $\langle -1, 1\rangle$, což sjednocuje velikosti vah penalizaující jednotlivé veličiny v účelové funkci algoritmu a zlepšuje numerickou stabilitu. Především ale normalizace umožňuje výrazné snížení složitosti problému během offline příprav regulátoru. Pro lepší dosažení nulové ustálené odchylky, je dále model motoru rozšířen o stavy reprezentující kumulativní hodnoty napěťových přírůstků, čímž vzniká inkrementální neboli přírůstkový model.

Účelová funkce je vytvořena jako suma kvadrátů stavů x a vstupů u soustavy při délce predikčního horizontu $N = 4$. Nerovnostní omezení limitující maximální povolené hodnoty napětí a proudů ve statoru jsou převedena na logaritmické bariéry ve tvaru $-\rho \log c_j(x, u)$, v jejichž okolí hodnota účelové funkce strmě roste. K porušení stanovených omezení pak nedochází ani v průběhu optimalizačních iterací.

Rozložením kvadratických forem účelové funkce lze funkci převést na součet velkého množství skalárních členů ve specifickém tvaru. Díky dříve zavedené normalizaci nyní platí, že absolutní příspěvek každého členu je omezen pouze jeho numerickým koeficientem. Koeficienty lze proto porovnat s největším koeficientem

v celé funkci (či její dílčí části) a členy, které nesplní nastavené kritérium, vyřadit. Tato operace probíhá ve specializované aplikaci vyvinuté pro offline přípravu NMPC regulátoru. Tento algoritmus snížení složitosti problému vedl k řádové úspoře počtu operací bez zásadního vlivu na přesnost regulace.

Tímto způsobem zjednodušenou účelovou funkci následně využívá paralelizovaný optimalzační algoritmus založený na BFGS quasi-Newtonově metodě. Zvolený počet agentů je inicializován v rovnoměrně pokrytém prostoru pomocí Haltonovy posloupnosti a následně provádí každý z agentů několik iterací optimalizačního algoritmu. Agenti se pohybují na grafické vzájemně paralelně. Poté co každý z agentů absolvuje předepsaný počet iterací, je z dosažených hodnot splňujících nerovnostní omezení vybrána ta nejlepší. Výsledek nejlepšího z agentů je následně použit jako akční zásah na přivedený na motor. Paralelní zpracování nejen urychluje běh algoritmu, ale také zvyšuje pravděpodobnost nálezu globálního minima u nekonvexních úloh, jakou problém řízení PMSM představuje.

Dosažené výsledky

Dosaženou kvalitu řízení PMSM lze zhodnotit následovně: NMPC bez výrazných překmitů sleduje úhlovou rychlostní referenci, která sestává z náběžných i klesajících ramp, prudkých změn i z oblastí nastolujících odbuzovací režim motoru při kladných i záporných smyslech úhlové rychlosti; typická dosahovaná doba ustálení na krokovou změnu je v řádu desítek milisekund. V porovnání s vektorovým řízením (VC) jsou překmity rychlosti sníženy zhruba na polovinu. Regulátor je schopen eliminovat uměle zavedenou řídicí chybu v podobě zátěžného momentu. V režimu odbuzování je dosaženo rychlosti $\omega = 117\,\mathrm{rad}\cdot\mathrm{s}^{-1}$.

Byla také srovnána energetická spotřeba, která oproti VC pro NMPC klesla o 8,8 % (1,95 J oproti 2,14 J) a odebraná energie ze zdroje o 9 %. Díky méně agresivnímu průběhu proudu jsou navíc způsobeny nižší teoretické ztráty v měniči i menší zahřívání vinutí.

Průměrný čas jedné NMPC iterace na GPU (GTX 1050 Ti) je 56 μs, přičemž 98 % všech iterací bylo spočteno za dobu kratší než 200 μs, což splňuje požadavek na periodu vzorkování $t_s = 200\,\mu$s. Další práce v této oblasti bude však vyžadována pro dosažení real-time požadavku pro všechny vzorkovací okamžiky. Samotné hledání optima účelové funkce trvá bez datových přenosů mezi GPU a CPU přibližně 30 μs.

Logaritmické bariéry zajišťují dodržení omezení napětí i proudů. Oscilace, které se při použití logaritmických bariér objevují v okolí omezení jsou potlačovány vhodným filtrem kvantujícím přírůstky napětí na $0,2\,\mathrm{V}$.

Porovnání paralelní a sekvenční implementace prokázalo téměř identickou kvalitu regulace, rozdíly se objevují pouze při napětí těsně pod limitem, kde hrají roli numerické nepřesnosti plynoucí z hardwarových rozdílů mezi CPU a GPU.

Zhodnocení

Práce prokazuje, že kombinace offline redukce složitosti účelové funkce a paralelního optimalizačního algoritmu implementovaného na GPU umožňuje provoz NMPC pro PMSM v téměř reálném čase. Navržený přístup dramaticky zmenšuje počet vykonávaných operací (a tím zkracuje dobu výpočtu) bez ztráty přesnosti a řeší nekonvexní úlohu v řádu desítek mikrosekund. Ve srovnání s vektorovým řízením dosáhl menší energetické náročnosti. Algoritmus je také snadno aplikovatelný na další úlohy, kde se dá využít rychlá optimalizace nekonvexního problému.

SEDLÁŘ, Jan. *Parallelization in Nonlinear Model Predictive Control of Synchronous Motor Drives*. Master's Thesis. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Control and Instrumentation, 2025. Advised by Ing. Michal Kozubík

Author's Declaration

Author:	Bc. Jan Sedlář
Author's ID:	228521
Paper type:	Master's Thesis
Academic year:	2024/25
Topic:	Parallelization in Nonlinear Model Predictive Control of Synchronous Motor Drives

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno ....

author's signature*

*The author signs only in the printed version.

ACKNOWLEDGEMENT

I wish to convey my heartfelt thanks to my supervisor, Ing. Michal Kozubík, for his expert guidance, invaluable insights, and the freedom he granted me to implement solutions to the explored problem.

The work has been performed in the project A-IQ Ready: Artificial Intelligence using Quantum measured Information for realtime distributed systems at the edge No 101096658/9A22002. The work was co-funded by grants of Ministry of Education, Youth and Sports of the Czech Republic and Chips Joint Undertaking. The work was supported by the infrastructure of RICAIP that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857306 and from Ministry of Education, Youth and Sports under OP RDE grant agreement No CZ.02.1.01/0.0/0.0/17_043/0010085.

Contents

List of Figures

List of Tables

Listings

Introduction

Motor drives, especially those that utilize permanent magnet synchronous motors, play a significant role in modern engineering applications of industrial automation, electromobility, or renewable energetics. These motors are a popular choice because of their high efficiency, high power density, and control capabilities, which are critical for achieving optimal performance. [26] However, the nonlinear dynamics present in these motors combined with the imposed voltage and current limits require sophisticated control approaches to unlock their full potential. [66] Nonlinear model predictive control stands out as a promising method, mainly because of its capability to manage constraints and capture the whole dynamic of the motor. [1] Despite its advantages, the computational demands of the nonlinear model predictive control present a significant obstacle for real-time deployment, especially in systems where the control decision must be made in very short times. [53]

This thesis explores the potential of the nonlinear model predictive control to be utilized for the real-time regulation of synchronous motor drives. Conventional predictive control approaches often fall short of the tight timing requirements imposed due to the high complexity of the underlying problem. This demonstrates the need for innovative strategies to be further explored in order to improve computational speeds while, at the same time, ensuring the robustness of the control performance.

To address these challenges of nonlinear model predictive approach in synchronous motor control, this thesis proposes a novel approach exploiting the capabilities of parallel computations to accelerate the optimization process. The designed algorithm also addresses the problem of computational burden on the front of preceding preparations by reducing the complexity of the underlying problem. By combining these two aspects of the approach, a significant decrease in the time required to obtain solutions of the problem can be achieved.

The thesis is organized in the following structure: first, the theoretical foundation of model predictive control and optimization algorithms is researched, and then the basis for the parallelized control algorithm is designed. The implementation of support applications and the algorithm itself is described in the following chapters, with the parameters chosen for the final implementation of this algorithm being listed later on. The performance of the algorithm in terms of control and also in other relevant areas is tested and validated to conclude the thesis.

1 Model Predictive Control

Model predictive control (MPC) encompasses a wide array of optimization-based control strategies designed to determine optimal control signals for the regulation of dynamic systems nested within a feedback loop, adhering to the specified requirements. Central to all MPC methods is the use of a process model and the computation of the optimal solution of an objective function. This function typically aims to drive the process state toward a desired value while minimizing the control effort. Depending on the process model and the form of the objective function, various optimization techniques are employed to identify the best solution.

A defining characteristic of MPC is its predictive aspect, realized through the *receding horizon* strategy. At each time step, future process states are predicted using the selected model of the system, and an optimal sequence of control signals is derived by minimizing the objective function. However, only the first signal in this sequence is applied to the process at each step, necessitating a recalculation of the solution for the subsequent time instant. Although theoretically any process model can be utilized, the choice significantly impacts the feasibility of the overall solution. The versatility of MPC, in accommodating almost any process model, including those that are the most readily available, has contributed substantially to its widespread adoption. [42]

This adaptability also extends to the predictive control of nonlinear systems, provided an appropriate model is implemented. Different software implementations of MPC distinguish themselves by the specific process models they employ. Consequently, selecting the right process model, defining the objective function appropriately, and solving the optimization problem efficiently are the primary challenges in designing a model predictive control algorithm. [9]

The first industrial implementation of model predictive control emerged in the 1970s, primarily in power plants and refineries. During the following decades, the MPC approach expanded to industries such as food production, chemical processing, and automotive manufacturing. [55] A common trait among these early applications was their focus on dynamic systems with slow response times and long time constants. The limited computational power available at the time restricted the use of MPC in systems with fast dynamics, where real-time computational results were critical.

Initially, MPC was applied exclusively to purely linear control problems. However, most industrial applications required the incorporation of constraints on input, output, or state variables [56]. When such restrictions on variables were introduced, the predictive algorithm of the model became a nonlinear problem, necessitating more advanced mathematical optimization techniques to solve it. If linear con-

straints were applied instead, the problem would take the form of a convex quadratic program, which is generally simpler to solve. Consequently, the *quadratic programming* methods have been widely used in these scenarios.

MPC offers several key advantages that have driven its popularity. Unlike conventional PID controllers operating in the frequency domain, MPC, with its receding-horizon strategy, is formulated in the time domain. This characteristic makes it possible to control nonlinear systems predictively. Additionally, MPC can address multiple control objectives simultaneously within a single controller by employing a cost function. The relative importance of these objectives can be customized and fine-tuned for specific applications. Furthermore, MPC provides a systematic and straightforward approach to managing large or highly complex systems, a challenge often faced by traditional industrial control solutions. [17]

As already mentioned, the MPC implementation also presents some challenges that are non-negligible. Mainly, it can be very computationally demanding to solve the underlying optimization problem of the MPC in the given time frame. For systems with very fast dynamics, the implementation of predictive control can, therefore, be especially difficult.

1.1 Basic Principles

Although virtually any process model can be employed in formulating the MPC problem and its associated objective function, this thesis focuses specifically on the discrete-time state-space model of a dynamic process (1.1.1).

$$\boldsymbol{x}(k+1) = \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}\boldsymbol{u}(k) \tag{1.1.1a}$$

$$\boldsymbol{y}(k) = \boldsymbol{C}\boldsymbol{x}(k) \tag{1.1.1b}$$

In the state-space representation, the value of the state $\boldsymbol{x}$ at the next time instant $k+1$ depends on the state value at the current instant k and the current inputs $\boldsymbol{u}$. The output of the model is influenced by the state values. The respective weights of these relations are determined by a set of matrices: the state matrix $\boldsymbol{A}$, the input matrix $\boldsymbol{B}$, and the output matrix $\boldsymbol{C}$. These matrices must be appropriately dimensioned, and in this thesis, all vectors are considered to be in column form.

The desired control objectives are encapsulated within an objective function. This function assigns a scalar value to every possible combination of future model states, outputs, and control inputs. Minimizing this scalar value yields the desired control inputs for the next time instant, ensuring that the control objectives are met. A general definition, as described by [17], is given in (1.1.2).

$$J(\boldsymbol{x}(k), \boldsymbol{U}(k)) = \sum_{(i=k)}^{k+N_p-1} \Lambda(\boldsymbol{x}(i+1), \boldsymbol{u}(i)) \tag{1.1.2}$$

It involves summing N_p elements, where N_p is the length of the prediction horizon. Each element in this summation represents the weighted state and the control input values within the horizon. By adjusting the respective weights, different control objectives, such as the aggression of control actions or reference tracking, can be achieved. Each element in the sum must be nonnegative, and the objective function is typically designed to have a simple and continuous derivative to facilitate minimization. The quadratic function with Euclidean norms (1.1.3) is a widely adopted choice, where γ and λ are weighting parameters for the system states and control inputs, respectively. [22]

$$\Lambda(\boldsymbol{x}(k), \boldsymbol{u}(l)) = \gamma \|\boldsymbol{x}(k)\|^2 + \lambda \|\boldsymbol{u}(l)\|^2 \tag{1.1.3}$$

The objective function (1.1.2) depends on two parameters: the current state vector $\boldsymbol{x}(k)$ and the matrix $\boldsymbol{U}(k)$, which represents the sequence of control inputs over the prediction horizon (1.1.4).

$$\boldsymbol{U}(k) = \begin{bmatrix} \boldsymbol{u}^T(k) & \boldsymbol{u}^T(k+1) & \cdots & \boldsymbol{u}^T(k+N_p-1) \end{bmatrix}^T \tag{1.1.4}$$

Based on these parameters and the state-space model, future states can be predicted for N_p steps. The optimization task is to minimize the objective function with respect to $\boldsymbol{U}(k)$, ensuring that the control objectives are satisfied and feasible control inputs are computed for each future time instant. Only the first set of control inputs, corresponding to the current time instant k, is applied to the system.

In some cases, it is advantageous to divide the objective function into two separate sums of different lengths. One sum accounts for the system states, emphasizing the tracking ability of the system, while the other handles the control inputs. This distinction is particularly useful when tracking a constant reference, as only the first few control inputs in the sequence significantly influence the system behavior. [10] To accommodate this, the prediction horizon N_p and the control horizon N_c are defined, leading to the following refinement (1.1.5) of the objective function.

$$J(\boldsymbol{x}(k), \boldsymbol{U}(k)) = \sum_{(i=k)}^{k+N_p-1} \|\boldsymbol{x}(i+1)\|_{\boldsymbol{Q}}^2 + \sum_{(i=k)}^{k+N_c-1} \|\boldsymbol{u}(i)\|_{\boldsymbol{R}}^2 \tag{1.1.5}$$

Here, N_c represents the length of the control horizon, allowing flexibility to balance control aggressiveness and computational efficiency. The state and input vectors are represented here in the generalized squared norms $\|\boldsymbol{x}\|_Q^2 = \boldsymbol{x}^T \boldsymbol{Q} \boldsymbol{x}$. The

two weighting matrices $\boldsymbol{Q} \succeq 0$ and $\boldsymbol{R} \succ 0$ being positive semi-definite and positive-definite respectively. The Euclidean norm used in (1.1.3) is a special case of the squared norm with both $\boldsymbol{Q}$ and $\boldsymbol{R}$ being the identity matrix.

The algorithm executed at each time instant within a predictive controller can be described as in Algorithm 1.

Algorithm 1 MPC algorithm

while do
  obtain the state $\boldsymbol{x}(k)$ of the system
  minimize $J(\boldsymbol{x}(k), \boldsymbol{U}(k)) = \sum_{(i=k)}^{k+N_p-1} \Lambda(\boldsymbol{x}(i+1), \boldsymbol{u}(i))$
    with respect to $\boldsymbol{u}(\cdot) \in \boldsymbol{U}(k)$
    subject to arbitrary constraints
  denote the obtained optimal control sequence by $\boldsymbol{U}^*(k)$
  set the MPC feedback value to $\boldsymbol{u}^*(k) \in \boldsymbol{U}^*(k)$ and $k = k + 1$
end while

To begin minimizing the objective function, the current states of the system must first be acquired. These states can either be measured directly or, if direct measurement is impractical or infeasible, reconstructed using methods such as *Kalman filter*. Once the optimal control sequence has been determined, only the first control input vector of this sequence, corresponding to the current time instant k, is applied to the system. This process is repeated in subsequent time instants.

Even when the model of the dynamic process is linear, introducing constraints on its states, outputs, or inputs inherently transforms the problem into a nonlinear one. The specific methods for implementing such constraints and addressing the associated challenges will be explored in a subsequent chapter.

1.2 Nonlinear Model Predictive Control

Traditional model predictive control relies on linear models, which perform well for systems that are approximately linear or operate within a small range around an equilibrium point. However, many real-world systems exhibit nonlinear dynamics that cannot be adequately described by linear models. Before delving into the essentials of nonlinear MPC (NMPC), it is important to identify the scenarios where a nonlinear variant is necessary.

Certain real-life applications involve dynamic systems with pronounced nonlinear behavior even when operating near an equilibrium point. Similarly, systems designed to remain in transient modes, such as during startup or transitions between equilibrium points, cannot be effectively controlled with a simple linear MPC. In

such cases, a nonlinear model of the dynamic system becomes essential. Additionally, even when a linearized model might suffice, employing a nonlinear plant model often yields superior performance in terms of tracking accuracy and stability. [7]

A significant challenge in NMPC lies in the nonconvex nature of the global optimization problem. This nonconvexity makes it difficult to identify the global minimum and ensure optimality. However, it is widely recognized that feasibility, rather than optimality, is sufficient for stable NMPC operation. Feasibility in this context refers to the ability of the controller to find any solution (even suboptimal) to the nonconvex problem that satisfies imposed constraints within the time frame dictated by the application. [40]

The following text summarizes popular techniques for solving NMPC problems. A crucial strategy used is to prioritize feasibility over achieving the global minimum of the objective function. This approach, known as *Suboptimal NMPC*, addresses the inherent challenges of solving poorly behaved nonconvex objective functions. The primary focus is on finding a feasible solution that satisfies constraints within a short period rather than expending resources to locate the absolute optimal solution. Stability can still be ensured if the objective function consistently decreases at every iteration performed during a single sampling time, as demonstrated by [59].

One prominent method, called *simultaneous approach* [13] or *multiple shooting* [4], involves optimizing both system inputs and state trajectories concurrently. In this technique, system dynamics are incorporated as new equality constraints (1.2.1), allowing the solver to handle complex interactions between states and inputs effectively. See Fig. 1.1 where δ represents the step of the prediction horizon.

$$\overline{\boldsymbol{s}}_{k+1} = \overline{\boldsymbol{x}}(t_{k+1}, \overline{\boldsymbol{s}}_k, \overline{\boldsymbol{u}}_k) \tag{1.2.1}$$

If the constraints are satisfied, it ensures that all segments of the state trajectory align seamlessly. This approach is particularly advantageous because it generates a specific structure in the resulting quadratic programming problem, which can be exploited by fast convergence algorithms, such as the sequential quadratic programming discussed in Section 2.7. However, a significant limitation is that no conclusions about the system trajectory can be drawn from results computed during ongoing sub-iterations. A valid state trajectory becomes available only after completing the iteration process. If this completion is not achieved within the required time frame, the preliminary solution may be infeasible.

To address computational challenges, the concept of *short prediction horizons* can significantly reduce the computational burden while maintaining control quality. This method takes advantage of a key characteristic of MPC: Only the first control action of the computed sequence is applied and the rest is recalculated at the next time instant. Short-horizon strategies focus on accurately computing the

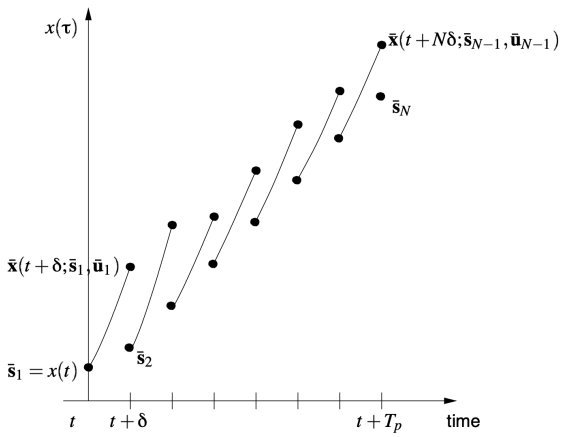

Fig. 1.1: Simultaneous approach [13]

first control input while approximating the remaining sequence. This reduces the number of decision variables to the number of control inputs, effectively focusing computational resources. The subsequent elements of the control sequence are approximated by interpolating between two control laws: one providing an optimal solution to the unconstrained problem and the other offering a large stabilizable set, albeit suboptimal. [71]

Another computationally efficient method is the *decomposition of the control sequence* presented in [8], which draws inspiration from the superposition principle commonly used in linear MPC. Although the superposition principle cannot be directly applied to nonlinear systems, a modified approach can approximate the system response. Two distinct trajectories are computed: the free system response, based on a nonlinear dynamic model, and the forced response, derived from an incremental linearized model of the plant. The combined approximation of the system response achieved using this method is more accurate than relying solely on linearized models. Solving this problem involves iterative adjustment of control increments, enabling the use of standard quadratic programming techniques common to linear MPC.

Feedback linearization, as described by [9], is another notable approach in which a set of transformations (1.2.2) is applied to a nonlinear dynamic system to make it linear (1.2.3), allowing standard linear control methods to be employed.

$$\boldsymbol{z}(t) = h(\boldsymbol{x}(t)) \tag{1.2.2a}$$

$$\boldsymbol{u}(t) = p(\boldsymbol{x}(t), \boldsymbol{v}(t)) \tag{1.2.2b}$$

$$\boldsymbol{z}(t+1) = \boldsymbol{A}\boldsymbol{z}(t) + \boldsymbol{B}\boldsymbol{v}(t) \tag{1.2.3a}$$

$$\boldsymbol{y}(t) = \boldsymbol{C}\boldsymbol{z}(t) \tag{1.2.3b}$$

However, this technique has significant limitations. For many nonlinear systems, finding suitable linearizing transformations is not feasible. Furthermore, any constraints, which are usually linear in the original formulation, become nonlinear after transformation, complicating their satisfaction.

Other methods also discussed by [9] include special Volterra models, which allow the use of fast-converging algorithms, and neural networks, which can model system dynamics or provide control strategies. Additionally, piecewise affine transforms break the nonlinear system into multiple regions of linear behavior, enabling localized control strategies to approximate the overall nonlinear response effectively.

1.3 NMPC for Very Fast Systems

Several recent and important contributions to the field of NMPC control for systems with very fast dynamics are worth noting. Diwan and Deshpande, in [12], implemented an NMPC algorithm to control an inverted pendulum with a sampling period of 20 ms. Their approach incorporated a nonlinear process model within a constrained objective function, optimized using a quadratic programming (QP) method described in more detail in (2.7).

In [38], Ma, Yao, Yang, and Xu explored a fast NMPC method for tracking the trajectory of quadrotor unmanned aerial vehicles (UAV). They used the *continuous/generalized minimum residuals* method to reduce computational complexity and compute acceptable control signals within short time periods. The computational times achieved for different lengths of the prediction horizon were 1.2 ms for $N = 5$, 4.1 ms for $N = 10$ and 6.5 ms for $N = 15$.

Patne et al., in [54], developed an NMPC controller used in advanced driver assistance systems (ADAS). Such systems have complex nonlinear dynamics, safety constraints, and the derived optimization problem has to be solved rapidly. To meet real-time requirements, they introduced suboptimality to reduce computation times, achieving runtimes below one millisecond while performing online linearization, and utilizing linear MPC methods to solve the NMPC problem, significantly improving over industry standard ADAS solutions.

For unmanned ground vehicle (UGV) path planning, Khan and Guivant proposed an NMPC algorithm in [29] that achieved average processing times per iteration of about one millisecond, a ten-fold improvement over Lu et al. [37], who tackled a

similar UGV path planning problem using single-shooting strategies and CasADi's interior-point optimizer. Khan and Guivant used the multiple shooting method, significantly reducing computational complexity and improving time efficiency.

Finally, Kozubík, Veselý, Aufderheide, and Václavek tested the NMPC control for a permanent magnet synchronous motor (PMSM) in [32]. They compared traditional field-oriented control (FOC) and linear MPC with an NMPC implementation that leveraged parallelized optimization based on population methods. Deployed on a programmable logic array, their NMPC approach demonstrated superior performance over FOC and linear MPC.

Based on this research, a design for the NMPC control of PMSM has been formulated. With the main goal of this implementation being its very fast computation, a suboptimal NMPC strategy inspired by [38] and [54] was selected, emphasizing acceptable rather than optimal solutions within reduced computation times. In addition, a custom optimization solver will be developed to take advantage of enhanced parallelization capabilities ([32]) and combine them with a rapid convergence programming method. To ensure robust tracking across the entire operating range of the motor, no linearization will be applied.

2 Mathematical Optimization

Mathematical optimization, also known as mathematical programming, is the process of selecting the most suitable element from a set of feasible options based on predefined criteria. Formally, an optimization problem can be expressed as follows:

$$\min_{\boldsymbol{x} \in X} f(\boldsymbol{x}),$$

$$\text{where} \quad X = \{\boldsymbol{x}, c_i(\boldsymbol{x}) \leq 0, i = [1, m]\}\,.$$

Here, f is the objective function, $f : \mathbb{R}^n \to \mathbb{R}$ on the set X, and $\boldsymbol{x} \in \mathbb{R}^n$ represents the optimization variable. The goal is to find a vector $\boldsymbol{x}^*$ that minimizes f while satisfying all constraint inequalities defined by $c_i(\boldsymbol{x})$. The solution $\boldsymbol{x}^*$ is the vector that yields the smallest possible value of the objective function while adhering to the constraints. Inverse mathematical programming problem can be formulated. In such a situation, the objective function is maximized, that is, the goal is to find $\boldsymbol{x}^*$ that maximizes f. [27]

Algorithms designed to solve optimization problems are called optimization solvers. Although no universally applicable method exists, a wide range of techniques is tailored to specific forms of the objective function or constraints, such as the least squares method. Before examining the optimization methods in more detail, it is crucial to understand some mathematical properties of functions that influence optimization.

A function is considered convex if it satisfies condition (2.0.1) for $\forall x_1, x_2 \in \text{dom}\, f$ and any $t \in [0, 1]$.

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2) \tag{2.0.1}$$

Geometrically, this means that for any two points on the graph of a convex function, the line segment connecting these points lies above or on the graph. Furthermore, if this inequality holds strictly (i.e., replace $\leq$ with $<$), the function is strictly convex. A strictly convex function will always take a unique minimum. Such functions are particularly valuable in optimization because they have a unique property: a local minimum is also a global minimum. Thus, solving an optimization problem involving a strictly convex function guarantees the discovery of the global optimum. On the contrary, nonconvex functions may have multiple local minima, not all of which are global minima, complicating the optimization process. If the inverse optimization problem is formulated, the same applies to concave and strictly concave functions. [72]

Differentiability is another critical property of functions in optimization. A function is differentiable if its derivative exists at every point within its domain. For multivariate functions, differentiability requires the existence of all partial derivatives across the domain. Mathematically, a function $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable in x if there exists a linear function L such that (2.0.2) holds. Here, L approximates the differential in a given direction, differing from the exact change of the function by an infinitesimally small correction term $\alpha(h)$.

$$f(x+h) = f(x) + h \cdot L(x) + \alpha(h), \quad \text{where } \lim_{h \to 0} \frac{\alpha(h)}{\|h\|} = 0. \tag{2.0.2}$$

It is also important to distinguish between constrained and unconstrained optimization problems. Unconstrained are such, where no bounds are imposed on the variables, whereas the constrained ones require the variables to follow some given conditions. These constraints can be simple inequalities such as $0 < x < 10$, more general linear constraints, or even nonlinear functions.

The topic of optimization is very broad and, therefore, only the concepts that have a significance in relation to this thesis are further explored.

2.1 Optimization Algorithms

Optimization algorithms can be categorized on the basis of various criteria, some of which are described below. One key distinction is based on the type of values that the optimization variables can take. If the variables are restricted to integer values, the problem is termed integer programming, often associated with a finite set of possible solutions. In contrast, continuous optimization deals with variables that can take any real value within a specified range. [46]

Another classification arises from the nature of the objective function. Optimization algorithms are tailored to specific types of objective functions, reflecting the diversity of practical applications. Linear programming addresses optimization problems with linear objective functions and constraints, typically simpler to solve than nonlinear cases. A notable special case of nonlinear programming is quadratic programming, where the objective function comprises quadratic terms. Nonlinear programming can further be divided into convex and nonconvex categories. Convex problems are easier to solve because they guarantee a global minimum, whereas nonconvex problems present significant challenges due to the existence of multiple local minima. [5]

In nonconvex optimization, a critical trade-off often arises: whether to settle for a local minimum, which is faster and less resource-intensive to find, or to pursue

the global minimum, which may better meet application-specific requirements but demands significantly more computational effort.

Constrained optimization introduces additional complexity compared to unconstrained optimization. However, constrained problems can often be reformulated as unconstrained problems by incorporating penalty terms into the objective function. These terms increase the functional value when the variables approach or violate the imposed constraints, effectively steering the solution back into the feasible region. [46]

The differentiability of the objective function is another crucial factor. Differentiable functions enable the use of gradient-based methods, which rely on the function's derivatives to iteratively identify the optimal solution. For non-differentiable functions, alternative approaches are required, often necessitating specialized algorithms.

Optimization methods can also be distinguished by their strategy for navigating the solution space. Deterministic methods use precise analytical techniques to determine the optimal direction for each iteration, ensuring predictable results. In contrast, stochastic methods incorporate randomness to explore the solution space, which can be advantageous in escaping local minima in complex landscapes.

Regardless of the specific approach, all optimization algorithms are inherently iterative. They incrementally update the optimization variables at each step, refining the solution until an acceptable level of accuracy is achieved. This iterative nature underscores the incremental progress characteristic of optimization processes.

2.2 Line Search

Line search methods form a versatile class of optimization algorithms that iteratively update the optimization variable by determining both the direction of the step and its length (2.2.1). Although the selection of the step direction will be explored in subsequent chapters, the determination of the step length, a common element across all line search methods, will be reviewed here.

$$\phi(\alpha) = f(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k), \quad \text{where } \alpha > 0. \tag{2.2.1}$$

In (2.2.1) x_k represents the optimization variable, p_k the selected step direction and α the length of the step.

Choosing the step size often involves a trade-off between precision and computational efficiency. The optimal step length, denoted as α^*, minimizes the value of the objective function along the search direction. However, identifying this precise value is computationally expensive. In practice, it is more common to perform an

inexact line search, with the aim of finding a step length that provides a sufficient decrease in the objective function without excessive computational effort. This approach focuses on finding an acceptable α that meets the predefined terminating conditions, balancing efficiency with functional improvement. [45]

One widely used set of terminating conditions for the inexact line search is the Wolfe conditions, first published by Wolfe in [67]. Two key criteria are enforced. The condition (2.2.2) ensures that the step produces a substantial reduction in the objective function and is known as *Armijo rule* [2]. This condition alone is usually used for the backtracking line search, where the step size α is iteratively multiplied by a positive coefficient smaller than one until the condition is satisfied or the maximum number of iterations is reached.

$$f(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k) \leq f(\boldsymbol{x}_k) + c_1 \alpha \nabla f_k^T \boldsymbol{p}_k \tag{2.2.2}$$

To avoid excessively short steps, the second condition (2.2.3) can be used to ensure a sufficient change in the gradient along the search direction.

$$\nabla f(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k)^T \boldsymbol{p}_k \geq c_2 \nabla f_k^t \boldsymbol{p}_k \tag{2.2.3}$$

The pair of constants c_1 and c_2 in (2.2.2) and (2.2.3) appear in the Nocedal and Wright formulations of the Wolfe conditions [46]. The authors suggest values of $c_1 = 10^{-4}, c_2 = 0.9$. When (2.2.3) is reformulated to strengthen the constraints, as seen in (2.2.4) and in pair with the first condition, these criteria are known as strong Wolfe conditions, which yield a more reliable solution to the line search.

$$|\nabla f(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k)^T \boldsymbol{p}_k| \leq c_2 |\nabla f_k^t \boldsymbol{p}_k| \tag{2.2.4}$$

An alternative to Wolfe's conditions are Goldstein's conditions (2.2.5), that also aim to identify an acceptable α for the line search. Goldstein's approach requires only one constant c, selected within the interval $0 < c < 0.5$. These conditions provide a simpler yet effective means of balancing sufficient decrease with computational feasibility. [20]

$$f(\boldsymbol{x}_k) + (1-c)\alpha_k \nabla f_k^t \boldsymbol{p}_k \leq \nabla f(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k) \leq f(\boldsymbol{x}_k) + c\alpha_k \nabla f_k^t \boldsymbol{p}_k \tag{2.2.5}$$

2.3 Gradient Descent Methods

Gradient-based optimization methods, such as those based on the computation of the gradient of the objective function, are foundational approaches in optimization.

For such methods, the direction in which the search for the optimal solution is carried out is determined by the negative gradient of the objective function. [65]

The optimal solution is reached when the calculated gradient is zero. In practice, however, a terminating condition is usually set to be that $\|\nabla f\|^2$ is smaller than some chosen error threshold. This, of course, is not true when optimizing a nonconvex function, where a zero gradient can be reached even at local minima, maxima, or saddle points of the function.

Algorithm 2 Gradient descent method

Require: a starting points $x \in \operatorname{dom} f$

while Terminating condition is not reached **do**

$\Delta \boldsymbol{x} := -\nabla f(\boldsymbol{x})$

Line search. Select a suitable candidate for α via exact or inexact line search

Update $\boldsymbol{x} := \boldsymbol{x} + t\Delta \boldsymbol{x}$

end while

As stated by Tran-Dinh and van Dijk in [65], the approach described by Algorithm 2 usually has linear or sublinear convergence when working with a convex function. However, the speed of convergence can be quite slow. The main advantage of gradient descent methods is their simplicity.

2.4 Steepest Descent Methods

The search direction in this approach is derived from the first-order Taylor approximation of the objective function (2.4.1), as described in [5].

$$f(\boldsymbol{x} + \boldsymbol{\delta}_{\boldsymbol{x}}) \approx \widehat{f}(\boldsymbol{x} + \boldsymbol{\delta}_{\boldsymbol{x}}) = f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T \boldsymbol{\delta}_{\boldsymbol{x}} \tag{2.4.1}$$

The term on the right-hand side of (2.4.1) is referred to as the directional derivative and approximates the change in the value of the objective function in the direction $\boldsymbol{v}$. The goal is to find such $\boldsymbol{v}$ that the directional derivative becomes the most negative, representing the steepest descent. The step of the steepest descent approach can be mathematically expressed as in (2.4.2)

$$\Delta \boldsymbol{x}_{\text{nsd}} = \operatorname{argmin} \left\{ \nabla f(\boldsymbol{x})^T \boldsymbol{\delta}_{\boldsymbol{x}} \mid \|\boldsymbol{\delta}_{\boldsymbol{x}}\| \leq 1 \right\}, \tag{2.4.2}$$

where $\boldsymbol{x}_{\text{nsd}}$ stands for normalized steepest descent. By scaling the normalized steepest descent step, we obtain equation (2.4.3)

$$\Delta \boldsymbol{x}_{\text{sd}} = \|\nabla f(\boldsymbol{x})\|_* \Delta \boldsymbol{x}_{\text{nsd}} \ . \tag{2.4.3}$$

In (2.4.3) the $\|\cdot\|_*$ denotes the dual norm to the one used for the size of $\boldsymbol{v}$. Depending on the selection of these norms, different formulations of the steepest descent are obtained. For the Euclidean norm, a straightforward equation holds: the steepest descent direction is simply the negative gradient, identical to that in *gradient descent*. For a quadratic norm, the steepest descent step is given by (2.4.4)

$$\Delta \boldsymbol{x}_{\text{sd}} = -\boldsymbol{P}^{-1}\nabla f(\boldsymbol{x}), \tag{2.4.4}$$

where $\boldsymbol{P}$ is a positive definite matrix used in the quadratic norm. If the ℓ_1 norm is chosen for the size of the descent direction as in (2.4.5a), the steepest descent step is defined by (2.4.5b).

$$\Delta \boldsymbol{x}_{\text{nsd}} = \text{argmin}\left\{\nabla f(\boldsymbol{x})^T \boldsymbol{\delta}_{\boldsymbol{x}} \mid \|\boldsymbol{\delta}_{\boldsymbol{x}}\|_1 \leq 1\right\} \tag{2.4.5a}$$

$$\Delta \boldsymbol{x}_{\text{sd}} = \Delta \boldsymbol{x}_{\text{nsd}} \|\nabla f(\boldsymbol{x})\|_\infty = -\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}_i} e_i \tag{2.4.5b}$$

With e_i being the i-th standard basis vector and i being any index satisfying condition (2.4.6).

$$|\nabla f_i| = \max_j |\nabla f_j| \tag{2.4.6}$$

The selection of the norms is crucial to achieve an acceptable convergence rate. In practice, the quadratic norm is often preferred due to the flexibility to manipulate the results through the $\boldsymbol{P}$ matrix. However, finding an appropriate $\boldsymbol{P}$ matrix for the optimization problem can be challenging. As summarized in [5], steepest descent methods perform well when a suitable $\boldsymbol{P}$ matrix is identified for the problem at hand.

2.5 Newton and quasi-Newton Methods

One approach to determining a suitable $\boldsymbol{P}$ matrix for the quadratic norm in the steepest descent method is to set it as the Hessian of the objective function. This choice of the $\boldsymbol{P}$ matrix defines a class of optimization techniques known as Newton methods. [15]

The fundamental equation for these algorithms is (2.5.1)

$$\Delta \boldsymbol{x}_{\text{nt}} = -\boldsymbol{G}^{-1}\nabla f(\boldsymbol{x}), \tag{2.5.1}$$

where $\boldsymbol{G} = \nabla^2 f(\boldsymbol{x})$ represents the Hessian matrix of the objective function $f(\boldsymbol{x})$. The equation for the Newton step is derived from the second-order Taylor series approximation (2.5.2). Such a formulation is one of several interpretations

of Newton methods, as the Newton step equation can be derived and justified in multiple ways.

$$f(\boldsymbol{x}+\boldsymbol{\delta}_{\boldsymbol{x}}) \approx \widehat{f}(\boldsymbol{x}+\boldsymbol{\delta}_{\boldsymbol{x}}) = f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T \boldsymbol{\delta}_{\boldsymbol{x}} + \frac{1}{2}\boldsymbol{\delta}_{\boldsymbol{x}}^T \boldsymbol{G} \boldsymbol{\delta}_{\boldsymbol{x}} \tag{2.5.2}$$

The optimization algorithm itself proceeds iteratively as illustrated in Algorithm 3. This version is often referred to as *damped Newton method*, as it incorporates a line search for α to adjust the step size. In contrast, the regular Newton method sets $\alpha = 1$ directly. The Newton method achieves very rapid convergence to the optimal solution, often quadratic, and performs well even with a large number of variables. Additionally, it is affine invariant and its performance is relatively insensitive to parameter selection, such as the $\boldsymbol{P}$ matrix in steepest descent methods. However, the standard Newton method fails when $\boldsymbol{G}$ is not positive definite, which can occur when $\boldsymbol{x}$ is far from the solution $\boldsymbol{x}^*$. Several modifications to the original method address this issue of global convergence, with the approach of Goldfeld et al. chosen as an example [19]. In this implementation, a small multiple of the identity matrix is added to $\boldsymbol{G}$. This gives the Hessian matrix a slight bias toward the vector of the steepest descent. The convergence of the method, in a case where the Hessian is positive definite, is ensured when performing a line search for step size determination.

Algorithm 3 Newton's method

Require: a starting points $x \in \operatorname{dom} f$

Ensure: tolerance $\epsilon > 0$

while $\lambda^2/2 > \epsilon$ **do**

$\Delta \boldsymbol{x}_{\text{nt}} := -\boldsymbol{G}^{-1}\nabla f(\boldsymbol{x})$

$\lambda^2 := \|\nabla f(\boldsymbol{x})\|^2_{\boldsymbol{G}^{-1}}$

Line search. Select a suitable candidate for α via exact or inexact line search

Update $\boldsymbol{x} := \boldsymbol{x} + t\Delta \boldsymbol{x}_{\text{nt}}$

end while

From the Newton step equation (2.5.1), it is evident that the objective function must be twice continuously differentiable for this method to be applicable. The primary drawback of the Newton method is the computational cost associated with calculating the Hessian matrix and its inversion. This issue is addressed by a family of derived methods known as quasi-Newton methods. In these methods, the Newton step equation is modified by replacing the Hessian matrix with its symmetric positive-definite approximation $\boldsymbol{B}$, which is iteratively updated at each step. The new Newton step is then computed in accordance with (2.5.3).

$$\Delta \boldsymbol{x}_{\text{nt}} = -\boldsymbol{B}_k^{-1}\nabla f_k \tag{2.5.3}$$

The approach of quasi-Newton methods provides several benefits over the standard Newton algorithm. First, the quasi-Newton method does not require the second-order derivatives of the optimized function to be computed, as the approximation $\boldsymbol{B}$ is updated with only the derivatives of the first order. Since the matrix $\boldsymbol{B}$ is positive definite, as ensured by its update algorithm, the issues associated with global convergence do not have to be addressed. Finally, fewer computations are required in each iteration. While the classis Newton approach requires $\mathcal{O}(n^3)$ multiplications, in the quasi-Newton approach, only $\mathcal{O}(n^2)$ are performed. [15]

It has been demonstrated that the fast convergence of the Newton method is retained in quasi-Newton methods, even if $\boldsymbol{B}$ does not converge to the true Hessian. It is sufficient for the approximation to become increasingly accurate with each iteration of the optimization process. [46]

2.6 Trust-Region Methods

The Goldfeld et al. [19] research on the modifications of the Newton method referenced in Section 2.5 also introduced another important technique of the trust region. Trust-region methods, like some previously discussed approaches, are based on the Taylor series expansion of the objective function. Specifically, a local quadratic model in the form of (2.6.1) is defined, where $\boldsymbol{f}_k = f(\boldsymbol{x}_k)$. This model can be interpreted as a locally scaled objective function.

$$\boldsymbol{\phi}_k(\boldsymbol{p}) = \boldsymbol{f}_k + \nabla f(\boldsymbol{x}_k)^T \boldsymbol{p} + \frac{1}{2}\boldsymbol{p}^T \boldsymbol{G}\boldsymbol{p} \tag{2.6.1}$$

The algorithm defines a trust region, a bounded area around the current variable value, within which this approximation (2.6.1) is considered sufficiently accurate. Then a step $\boldsymbol{p}$ is determined by minimizing the approximation within this trust region. [63]

If the step determined by minimizing the approximation does not yield a better result, that is, if it does not bring the solution closer to the optimum, it suggests that the trust region was set too large and the approximation is not accurate enough in that region. Consequently, the trust region is reduced. Conversely, if the step results in a satisfactory improvement, the trust region can be expanded.

The size of the trust region is dynamically adjusted based on the performance of the algorithm in previous iterations. This adaptability helps the algorithm balance exploration and exploitation, ensuring efficient progress toward the solution.

As with Newton methods, the true Hessian can be replaced by an approximation that is iteratively updated at each step, reducing computational costs. Trust-region

methods often exhibit convergence speeds comparable to Newton methods, typically achieving superlinear convergence.

2.7 Constrained Optimization

If the optimization variable is restricted by equality or inequality constraints, whether linear or nonlinear, the problem is referred to as constrained optimization. The general formulation of such a problem is

$$\min_{x\in\mathbb{R}^n} f(\boldsymbol{x}), \quad \text{subject to} \begin{cases} c_i(\boldsymbol{x}) = 0, & i \in \mathcal{E}, \\ c_i(\boldsymbol{x}) \geq 0, & i \in \mathcal{I}, \end{cases}$$

where $f(\boldsymbol{x})$ is the objective function and $c_i(\boldsymbol{x}), i \in \mathcal{E}$ represents equality constraints and $c_i(\boldsymbol{x}), i \in \mathcal{I}$ inequality constraints.

Certain algorithms inherently handle specific types of constraints, such as linear constraints in quadratic programming. Other methods require modifications or incorporate constraints directly into the objective function to solve the problem. A widely used tool for constrained optimization is the Lagrangian function (2.7.1).

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) - \sum_{i\in\mathcal{E}\cup\mathcal{I}} \lambda_i c_i(\boldsymbol{x}) \tag{2.7.1}$$

The condition (2.7.2) must hold true at each solution $\boldsymbol{x}^*$. The scalar λ^* is called the *Lagrange multiplier* to the c_i constraint and provides insight into how tightly a particular constraint is binding at the solution. [46]

$$\nabla_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}^*, \lambda_i^*) = 0 \tag{2.7.2}$$

A set of conditions (2.7.3), known as *Karush-Kuhn-Tucker* or simply KKT conditions, which are necessary to meet in local solutions to the optimization problem, are the foundation of many constrained optimization algorithms. [33]

$$\nabla_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}^*, \boldsymbol{\lambda}^*) = 0 \tag{2.7.3a}$$

$$c_i(\boldsymbol{x}^*) = 0, \qquad \forall i \in \mathcal{E} \tag{2.7.3b}$$

$$c_i(\boldsymbol{x}^*) \geq 0, \qquad \forall i \in \mathcal{I} \tag{2.7.3c}$$

$$\lambda_i^* \leq 0, \qquad \forall i \in \mathcal{I} \tag{2.7.3d}$$

Since the KKT conditions involve first-order derivatives, they can also hold for solutions that maximize the objective function instead of minimizing it. To differentiate between minima and maxima, second-order conditions are sometimes applied.

A significant class of constrained optimization methods transforms the problem into an unconstrained one by embedding the constraints into the objective function. The way in which this is handled determines the set of methods that are used to solve the optimization problem. One option is to rewrite the original inequality-constrained optimization problem into a form (2.7.4) where the constraints are represented as a sum of functions in the minimization objective. Those barrier functions posses the property of growing increasingly, as the variable approaches the constraint $c_i(\boldsymbol{x})$ and therefore increasing the overall value of the objective function. These functions can be logarithmic, as these terms grow as the constraints are violated. [46]

$$\text{minimize } f(\boldsymbol{x}) + \sum_{i \in \cup \mathcal{I}} -\rho \log(-c_i(\boldsymbol{x})) \tag{2.7.4}$$

This so-called barrier function can also be of other natures, for example, a quadratic one or an absolute value, even though that makes the objective function impossible to differentiate continuously.

Another approach is to solve a sequence of penalty-modified unconstrained problems (2.7.5) with gradually increasing ρ or to use a constant value, which is then called the exact penalty function method. The solution of the modified problem should ideally converge to the solution of the original problem. In practice, however, this poses a problem, as the problem should be solved for $\rho \to \infty$. [45]

$$\text{minimize } f(\boldsymbol{x}) + \rho \sum_{i \in \mathcal{E}} c_i^2(\boldsymbol{x}) \tag{2.7.5}$$

The Augmented Lagrangian method combines penalty functions with Lagrangian terms, expressed as (2.7.6). The minimized function f can be subjected to equality constraints $c_i(\boldsymbol{x}) = 0,\ i \in \mathcal{E}$. In this method, the estimates for *Lagrange multipliers* $\boldsymbol{\lambda}$ and the constant ρ are iteratively updated as the minimization process progresses until the desired accuracy is reached. The method was first described by Hestenes in [25].

$$\mathcal{L}_A(\boldsymbol{x}, \boldsymbol{\lambda}) = f(\boldsymbol{x}) - \sum_{i \in \mathcal{E}} \lambda_i c_i(\boldsymbol{x}) + \rho \sum_{i \in \mathcal{E}} c_i^2(\boldsymbol{x}) \tag{2.7.6}$$

The method first solves the unconstrained problem of minimizing (2.7.6) with an initial value of ρ, which is larger in each iteration. The value of *Lagrange multiplier* estimate λ_i is updated according to the formula (2.7.7), with $\boldsymbol{x}^*(k)$ denoting the optimum obtained in step $k = 1$. The *Lagrange multiplier* estimate is more accurate with each iteration and the method proves to provide better results then the penalty function method.

$$\lambda_i(k+1) = \lambda_i(k) + 2\rho c_i(\boldsymbol{x}^*(k)) \tag{2.7.7}$$

Another popular category of constrained optimization algorithms is quadratic programming, which focuses on problems where the objective function contains quadratic terms. In its basic form, it exploits the fundamental characteristics of objective function formed out of these terms and is considered one of the easiest to solve in nonlinear programming. The formulation (2.7.8) introduces a symmetric matrix $\boldsymbol{P}$, and its definiteness influences the difficulty of the problem. A positive semi-definite $\boldsymbol{P}$ ensures convexity, making the problem easier to solve. [5]

$$\min_{x\in\mathbb{R}^n} g(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{P}\boldsymbol{x} + \boldsymbol{x}^T\boldsymbol{c} \quad \text{subject to} \begin{cases} \boldsymbol{a}_i^T\boldsymbol{x} = b_i, & i \in \mathcal{E} \\ \boldsymbol{a}_i^T\boldsymbol{x} \geq b_i, & i \in \mathcal{I} \end{cases} \tag{2.7.8}$$

Sequential Quadratic Programming (SQP) extends this approach by iteratively approximating a nonlinear problem as a quadratic one, solving these subproblems using quadratic programming techniques. This model is created using the second-order Taylor approximation, similar to (2.5.2). The ideal value for the matrix $\boldsymbol{G}$ is the Hessian of the Lagrangian function (2.7.1) of the original constrained problem. The optimization variable is updated according to (2.2.1). The SQP problem can therefore be expressed as in (2.7.9) with $\boldsymbol{G} = \nabla^2_{\boldsymbol{xx}}\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda})$. For the same reasons as for quasi-Newton methods, $\boldsymbol{G}$ can be replaced with the Hessian approximation. [68]

$$\text{minimize } \nabla f(\boldsymbol{x})^T\boldsymbol{p}_k + \frac{1}{2}\boldsymbol{p}_k^T\boldsymbol{G}\boldsymbol{p}_k, \text{ for } \boldsymbol{p}_k \tag{2.7.9a}$$

$$\text{subjected to: } c_i(\boldsymbol{x}) + \nabla c_i(\boldsymbol{x})^T\boldsymbol{p}_k = 0, \ i \in \mathcal{E} \tag{2.7.9b}$$

$$c_i(\boldsymbol{x}) + \nabla c_i(\boldsymbol{x})^T\boldsymbol{p}_k \leq 0, \ i \in \mathcal{I} \tag{2.7.9c}$$

3 Regulated System

Before describing the control algorithm, it is crucial to understand the plant it is supposed to regulate. The nonlinear dynamic system in this context is an electric rotational synchronous motor with permanent magnets embedded in the rotor, commonly referred to as a Permanent Magnet Synchronous Motor (PMSM). The continuous model of the PMSM is described by a set of state equations that represent its electrical (3.0.1a), (3.0.1b) and mechanical dynamics (3.0.1c). [44]

$$\frac{\partial i_d}{\partial t} = \frac{u_d}{L_d} + \frac{L_q}{L_d} P_p \omega_m i_q - \frac{R_s}{L_d} i_d \tag{3.0.1a}$$

$$\frac{\partial i_q}{\partial t} = \frac{u_q}{L_q} - \frac{L_d}{L_q} P_p \omega_m i_d - \frac{R_s}{L_q} i_q - \frac{\Psi_{PM}}{L_q} P_p \omega_m \tag{3.0.1b}$$

$$\frac{\partial \omega_m}{\partial t} = \frac{1}{J}(T_{el} - T_l) = \frac{1}{J}\left(\frac{3}{2} P_p \left[\Psi_{PM} i_q + (L_d - L_q) i_d i_q\right] - T_l\right) \tag{3.0.1c}$$

The quantities and constants in equations (3.0.1) have the following meaning:

u_d, u_q	stator voltage components in the rotating frame,
i_d, i_q	stator current components in the rotating frame,
R_s	stator winding resistance,
L_d, L_q	stator inductance components,
Ψ_{PM}	permanent magnet flux linkage,
P_p	number of pole-pairs,
ω_m	rotor mechanical angular speed,
J	rotational moment of inertia,
T_l	load torque.

The voltages applied to the motor stator and the resulting currents are represented in the model using d and q coordinates, which are widely used in similar applications. However, these coordinates cannot be applied to the motor directly and must be transformed into standard a, b, and c coordinates first. During the design and implementation of the control algorithm, no physical motor was used; instead, the predictive control of the model was tested in a simulation environment.

The system imposes practical constraints on applied voltages and generated currents to ensure safe and effective operation. The magnitude of the voltage is limited by the power source, and current levels must be kept within acceptable limits to prevent damage to the motor. These constraints are mathematically expressed as (3.0.2)

$$\|\boldsymbol{u}\| \leq U_{max} \quad \sim \quad u_d^2 + u_q^2 \leq U_{max}^2 \ , \tag{3.0.2a}$$

$$\|\boldsymbol{i}\| \leq I_{max} \quad \sim \quad i_d^2 + i_q^2 \leq I_{max}^2 \ , \tag{3.0.2b}$$

where U_{max} and I_{max} are the maximum allowable voltage and current magnitudes, respectively. The vector sum of the components d and q, which are always orthogonal to each other, must satisfy these limits.

When the controlled quantity is the speed of the motor, it can operate in different regions, two of them being significant for this work. When the stator voltage is below the imposed limit and the optimal operating point of the motor is determined by the currents efficiency, we speak of maximum torque per Ampere (MTPA) operation. Therefore, the desired currents in this region are determined by the MTPA curve, which connects the optimal dq pairs of currents. However, this approach is admissible only if the requested motor speed is less than a certain base speed. If the speed reference grows past this point, the motor needs to transition to field weakening (FW) operations. Field weakening (also called flux weakening) is used to reach higher speeds at the expense of a lower torque produced. During this operation, the direct part of the current, which is otherwise held around zero as given by the MTPA curve, grows into negative values causing the magnetic field to weaken. This drop in i_d is caused by a simultaneous drop in u_d. As the voltage is already on its imposed limit before field-weakening operations begin, the quadrature part of the voltage needs to be reduced to accommodate the higher negative magnitude of the direct part. [35]

3.1 Discrete Model of the System

The continuous system equations cannot be used directly with discrete control logic and must first be discretized. *Euler method* was used to achieve this, resulting in a set of discrete state equations (3.1.1). These equations describe the dynamics of the system over discrete time intervals with a sampling period t_s. The physical meaning of each constant remains consistent with the continuous model of the plant.

$$i_d(k+1) = \left(1 - t_s\frac{R_s}{L_d}\right) i_d(k) + t_s P_p \frac{L_q}{L_d}\omega_m(k) i_q(k) + \frac{t_s}{L_d} u_d(k) \tag{3.1.1a}$$

$$i_q(k+1) = \left(1 - t_s\frac{R_s}{L_q}\right) i_q(k) - t_s P_p \frac{L_d}{L_q}\omega_m(k) i_d(k) + \frac{t_s}{L_q} u_q(k) - t_s P_p \frac{\Psi_{PM}}{L_q}\omega_m(k) \tag{3.1.1b}$$

$$\omega_m(k+1) = \omega_m(k) + \frac{t_s}{J}\left(\frac{3}{2}P_p\left[\Psi_{PM} i_q(k) + (L_d - L_q) i_d(k) i_q(k)\right] - T_l\right) \tag{3.1.1c}$$

To account for one of the unique features of the designed nonlinear MPC algorithm, model normalization is introduced. In this approach, all model variables (inputs, states, and outputs) are scaled so that their absolute values are within the range of $[-1, 1]$. These normalized variables are denoted by a subscript n, and the normalization transformation is expressed in (3.1.2), where x_j is the original variable and x_j^* is its maximum allowable value.

$$x_{jn} = \left(\frac{x_j}{x_j^*} \right) \tag{3.1.2}$$

The equations for the normalized system (3.1.3) are derived accordingly. In these equations, the arguments (k) of the system states from the right-hand sides were omitted for better readability, and the upperscript $(\cdot)^*$ stands for the maximum value of the variable. To use an MPC designed with the normalized model on a denormalized (real) plant, a two-part transformation is required: normalize MPC inputs by dividing them by their respective maximum values, and denormalize MPC outputs by multiplying them by their respective maximum values.

$$i_{dn}(k+1) = \left(1 - t_s \frac{R_s}{L_d}\right) i_{dn} + t_s P_p \frac{L_q}{L_d} \frac{i_q^* \omega_m^*}{i_d^*} \omega_{mn} i_{qn} + \frac{t_s}{L_d} \frac{u_d^*}{i_d^*} u_{dn} \tag{3.1.3a}$$

$$\begin{aligned} i_{qn}(k+1) = &\left(1 - t_s \frac{R_s}{L_q}\right) i_{qn} - t_s P_p \frac{L_d}{L_q} \frac{i_d^* \omega_m^*}{i_q^*} \omega_{mn} i_{dn} + \frac{t_s}{L_q} \frac{u_d^*}{i_q^*} u_{qn} - \\ &- t_s P_p \frac{\Psi_{PM}}{L_q} \frac{\omega_m^*}{i_q^*} \omega_{mn} \end{aligned} \tag{3.1.3b}$$

$$\omega_{mn}(k+1) = \omega_{mn} + \frac{t_s}{J} \left(\frac{3}{2} P_p \left[\Psi_{PM} \frac{i_q^*}{\omega_m^*} i_{qn} + (L_d - L_q) \frac{i_q^* i_d^*}{\omega_m^*} i_{dn} i_{qn} \right] - \frac{T_l}{\omega_m^*} \right) \tag{3.1.3c}$$

Although the quantities normalization approach will later serve as a necessary requirement for the designed optimization algorithm, it is also important to consider some practical aspects of system control. In practical control applications, achieving zero control error is highly desirable. In traditional control methods, such as a PID controller, this is often done by incorporating an integrator into the control structure. For model predictive control, this is accomplished using an incremental plant model, where the control inputs at each step are replaced by their increments. The total magnitude of the controls is then computed by summing the increment value and the total control input value from the previous sampling step, which is stored in the model in the form of an additional state.

To accommodate for this, an additional set of state equations (3.1.4) storing the total value of control inputs has to be introduced. Together with (3.1.3), they now form a suitable discrete plant model, with Δu_{dn} and Δu_{qn} serving as input to the model.

$$u_{dn}(k+1) = u_{dn}(k) + \frac{\Delta u_d^*}{u_d^*} \Delta u_{dn}(k) \tag{3.1.4a}$$

$$u_{dn}(k+1) = u_{qn}(k) + \frac{\Delta u_q^*}{u_q^*} \Delta u_{qn}(k) \tag{3.1.4b}$$

4 Parallelized Optimization Algorithm

The algorithm was designed primarily with the objective of controlling the PMSM motor within the nonlinear MPC structure. However, it can be used very easily in other similar applications or even in fields beyond predictive control. The features and potential limitations are outlined in this chapter.

One of the key ideas of the proposed algorithm is to handle the problems commonly associated with nonconvex optimization by initializing multiple search agents that work in parallel. In theory, each agent should iteratively converge to one local minimum, sometimes even to the same minimum as other agents, depending on the count of initialized agents. The results obtained are subsequently compared against each other, and the best is selected based on a defined decision criterion. Ideally, a global minimum is found in this way, and the challenges posed by the nonconvex objective function are henceforth addressed. This, of course, depends on the nature of the objective function and the number of search agents, as with a function that has many local minima and is not sufficiently explored using an adequate number of agents, the global minimum may not be reached.

As the number of agents is limited in practice, it was also important to design an optimization algorithm that accelerates the convergence of each individual agent to the minimum of the function. This process consisted of two distinct parts: modifying the objective function to contain as few terms as possible and selecting a suitable optimization method to find a minimum of the simplified function. The latter is now discussed.

4.1 Optimization Method

Mainly due to its fast convergence described in Section 2.5, a quasi-Newton method was selected as a suitable candidate for the optimization method used for each search agent. A more detailed overview of the chosen algorithm is presented in the following text, with a general overview available in Section 2.5. The *Broyden-Fletcher-Goldfarb-Shanno* (BFGS) algorithm was selected as it is currently considered the most effective of all quasi-Newton methods. It was initially presented independently by its creators Broyden [6], Fletcher [14], Goldfarb [18] and Shanno [62] and belongs to a subclass of quasi-Newton methods that do not directly approximate the Hessian itself but instead approximate its inverse. This means that the algorithm does not require second-order derivatives of the objective function at each iteration and avoids inverting a large matrix. Although the final convergence is not as rapid as the pure Newton method, this trade-off is well balanced by its significantly lower computational demands and enhanced stability and robustness.

$$\boldsymbol{H}_{k+1} = \boldsymbol{H}_k + \frac{\left(\boldsymbol{s}_k^T \boldsymbol{y}_k + \boldsymbol{y}_k^T \boldsymbol{H}_k \boldsymbol{y}_k\right)\left(\boldsymbol{s}_k \boldsymbol{s}_k^T\right)}{\left(\boldsymbol{s}_k^T \boldsymbol{y}_k\right)^2} - \frac{\boldsymbol{H}_k \boldsymbol{y}_k \boldsymbol{s}_k^T + \boldsymbol{s}_k \boldsymbol{y}_k^T \boldsymbol{H}_k}{\boldsymbol{s}_k^T \boldsymbol{y}_k} \tag{4.1.1}$$

The update formula for the Hessian inverse approximation $\boldsymbol{H}$ in the BFGS algorithm (4.1.1) is designed to ensure positive definitness of the matrix by performing a rank-two update at each iteration. The whole BFGS algorithm is described in Algorithm 4.

Algorithm 4 BFGS method

Require: a starting point $\boldsymbol{x} \in \operatorname{dom} f$, inverse Hessian approximation initial value $\boldsymbol{H}_0$

Ensure: tolerance $\epsilon > 0$

$k \leftarrow 0$

∇f_k stands for $\nabla f(\boldsymbol{x}_k)$

while $\|\nabla f_k\| > \epsilon$ **do**

Compute search direction $\boldsymbol{p}_k = -\boldsymbol{H}_k \nabla f_k$

Line search. Select a suitable candidate for α_k via inexact line search satisfying the Wolfe conditions

Update $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$

Define $\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$ and $\boldsymbol{y}_k = \nabla f_{k+1} - \nabla f_k$

Update $\boldsymbol{H}_{k+1}$ by means of (4.1.1)

$k \leftarrow k + 1$

end while

After choosing an initial guess for the inverse Hessian matrix, denoted $\boldsymbol{H}_0$, the first search direction, denoted $\boldsymbol{p}_k$, is computed. A line search—typically an inexact line search—is then performed to find a suitable value for the parameter α, ensuring that it satisfies the Wolfe conditions as described in Section 2.2. The optimization variable is updated using $\boldsymbol{s}_k = \alpha_k \boldsymbol{p}_k$, and $\boldsymbol{y}_k$ is set to the difference in gradients. The Hessian inverse approximation is subsequently updated using the (4.1.1) formula.

For the actual implementation of the algorithm, [46] provides practical suggestions that were incorporated. Determining an appropriate initial guess $\boldsymbol{H}_0$ for the matrix can be challenging. In practice, a scaled multiple of the identity matrix is often used, such as equation (4.1.2) which is used to calculate the value of the multiple with I representing the identity matrix of the appropriate dimension. This step is performed during the first iteration of the Algorithm 4 after the vectors $\boldsymbol{y}_k$ and $\boldsymbol{s}_k$ have been calculated but before $\boldsymbol{H}_k$ is updated.

$$\boldsymbol{H}_0 = \frac{\boldsymbol{y}_k^T \boldsymbol{s}_k}{\boldsymbol{y}_k^T \boldsymbol{y}_k} I \tag{4.1.2}$$

The line search used to determine the parameter α is terminated once Wolfe conditions are met. As noted in [46], the algorithm may not perform as effectively if alternative termination criteria are employed. The parameters for Wolfe conditions are recommended to be set to $c_1 = 10^{-4}$ and $c_2 = 0.9$, with their respective significance illustrated in (2.2.2) and (2.2.3). It is also worth mentioning that the algorithm first attempts to check whether $\alpha_k = 1$ satisfies these conditions, only proceeding to additional steps of the inexact line search if this initial value does not meet the requirements.

During the implementation process, a few changes were made to the algorithm outlined in this section. This was done to accommodate specific demands of the NMPC and is discussed in more detail in Chapter 6 and Chapter 8.

4.2 Objective Function

It is quite common for the objective functions of MPC to take the quadratic form of (1.1.3). This approach was also chosen in this implementation, with quadratic norms used for the terms incorporated in the objective function. The final function, which does not yet include constraints, can be seen in (4.2.1). Here, N represents the length of the prediction horizon, while $\boldsymbol{P}$, $\boldsymbol{Q}$ and $\boldsymbol{R}$ are matrices used to weigh the respective terms of $\boldsymbol{x}$ (the states) and $\boldsymbol{u}$ (the inputs) of the discrete, normalized incremental model. The notation $(\cdot)\,(k+1|k)$ stands for the quantity predicted in the $k+1$-th step based on information from the k-th step.

$$J(\boldsymbol{x}(k), \boldsymbol{u}(k)) = \|\boldsymbol{x}(N|k)\|^2_{\boldsymbol{P}} + \sum_{i=k}^{k+N-1} \left[\|\boldsymbol{x}(i+1|k)\|^2_{\boldsymbol{Q}} + \|\boldsymbol{u}(i)\|^2_{\boldsymbol{R}} \right] \tag{4.2.1}$$

Inequality constraints on system states were integrated into the algorithm as additional terms in the objective function. This is shown in equation (4.2.2), where a second summation term is introduced. A logarithmic barrier function was used to penalize those variable values that approach their maximum limits. The steepness of these barrier functions can be tuned using the parameter ρ, which weighs the logarithmic functions, as illustrated in Fig. 4.1. [5]

$$J_C(\boldsymbol{x}(k), \boldsymbol{u}(k)) = J(\boldsymbol{x}(k), \boldsymbol{u}(k)) + \sum_{i=k+1}^{k+N} \sum_{j \in \mathcal{I}} -\rho \log c_j(\boldsymbol{x}(i), \boldsymbol{u}(k)) \tag{4.2.2}$$

As shown in Fig. 4.1, the logarithmic functions asymptotically approach infinity at $x = 0$ and take smaller values for negative values of x. Thus, the constraints must be rewritten as $c_j(\boldsymbol{x}(k), \boldsymbol{u}(k)) \leq 0$ inequalities. This is done by subtracting

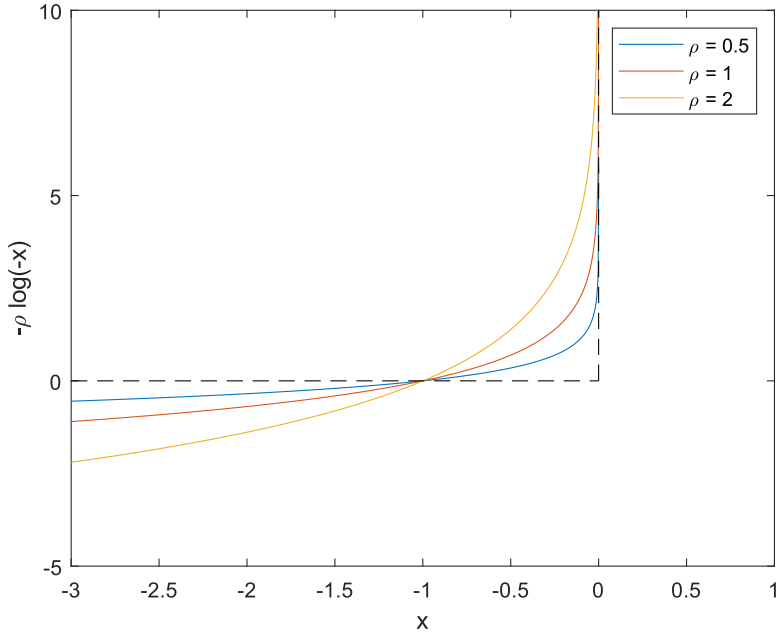

Fig. 4.1: Ideal barrier approximated by logarithmic function

the right-hand side term of the original inequality from both sides. For positive values of x, the logarithmic functions are not defined, which must be considered during the implementation of the algorithm.

4.3 Coefficients Pruning

As mentioned previously, additional modifications are applied to the objective function before its optimization. Specifically, parts of the function that have minimal impact on the final value of the function are omitted to reduce the computational demands. This is performed off-line, that is, during the design phase of the deployment of the NMPC. Considering the objective function of (4.2.2), it is evident that it cannot be computed in a matrix form in standard programming languages. Instead, the function must be expressed explicitly as the sum of individual scalar terms resulting from the matrix multiplication of the quadratic norm. Furthermore, the function includes the states and inputs over the prediction horizon of the NMPC. While the equations for the state variables in the first step are obtained directly from equations (3.1.3), the equation for an arbitrary step of the horizon (larger than one) is derived by recursively substituting the state equation from the previous step into the equation for the next. If the equations of states (3.1.3) and inputs (3.1.4) are rewritten in the form of (4.3.1), where each coefficient multiplying a state is expressed with a letter a–l (letter i has been omitted due to the similarity to a name

of the state), those equations can be substituted into the state equation for the next step. The resulting equation (4.3.2) shows $i_{dn}(3)$ rewritten just in terms of the state values in step $k = 1$, which is necessary to obtain the derivatives of the cost function with respect to the optimization variable. This process can be performed recursively for the whole objective function; however, the inputs must be distinguished not only by their d and q coordinates but also by the subscript corresponding to the prediction horizon step they pertain to, as inputs are the actual optimization variable, the value of which is not known a priori.

$$i_{dn}(3) = ai_{dn}(2) + b\omega_{mn}(2)i_{qn}(2) + cu_{dn}(2) \tag{4.3.1a}$$
$$i_{qn}(3) = di_{qn}(2) - e\omega_{mn}(2)i_{dn}(2) + fu_{qn}(2) - g\omega_{mn}(2) \tag{4.3.1b}$$
$$\omega_{mn}(3) = \omega_{mn}(2) + hi_{qn}(2) + ji_{dn}(2)i_{qn}(2) - k \tag{4.3.1c}$$
$$u_{dn}(3) = u_{qn}(2) + l\Delta u_{qn}(2) \tag{4.3.1d}$$

$$\begin{aligned} i_{dn}(3) = {} & a\left[ai_{dn}(1) + b\omega_{mn}(1)i_{qn}(1) + cu_{dn}(1)\right] + \\ & + b\left[\omega_{mn}(1) + hi_{qn}(1) + ji_{dn}(1)i_{qn}(1) - k\right] \cdot \\ & \cdot \left[di_{qn}(1) - e\omega_{mn}(1)i_{dn}(1) + fu_{qn}(1) - g\omega_{mn}(1)\right] + \\ & + c\left[u_{qn}(1) + l\Delta u_{qn}(1)\right] \end{aligned} \tag{4.3.2}$$

In this way a set of terms is obtained that contains only elements of the original matrices. If all such terms are summed, they form the original cost function, as illustrated by (4.3.3). The term $\boldsymbol{T}_i$ is then in the form shown in (4.3.4).

$$J_C(\boldsymbol{x}(k), \boldsymbol{u}(k)) = \sum_i \boldsymbol{T}_i \tag{4.3.3}$$

In (4.3.4), x_i represents the states of the model with $i \in [0, n_x]$ and n_x being the total number of states and similarly u_j represents the input of the model with $j \in [0, n_u]$ and n_u being the total number of inputs, k is the step of the prediction horizon with length N. $M \in \mathbb{R}$ is the quantification of the elements of the respective weighting matrices, σ_i and τ_{jk} are indicators that denote whether a state or input is present in the respective term $\sigma_i \in \{0,1\}^{n_x \times 1}, \tau_{jk} \in \{0,1\}^{n_u \times N}$. Note that such a term can contain only values of x_i from the first step of the prediction horizon (the currently available values), but inputs throughout the length of the horizon.

$$M \prod_{i=1}^{n_x} \sigma_i x_i(0) \cdot \prod_{j=1}^{n_u} \prod_{k=0}^{N-1} \tau_{jk} u_j(k) \tag{4.3.4}$$

Upon reviewing Section 3.1, it becomes evident that the variables present in each term (x_i and u_j) are restricted to the range $[-1, 1]$. Consequently, the absolute value of each term cannot exceed the coefficient M associated with it. This observation

implies that coefficients $\boldsymbol{M}$ play a critical role in determining the influence of each term on the overall value of the objective function.

This conclusion allows for the omission of terms with a negligible degree of influence from the optimization. Identifying such terms requires substituting numerical values for all constants in the plant model and parameter values of the objective function. The implementation of this filter is elaborated in Section 5.1, but it is important to note that the coefficients $\boldsymbol{M}$ are compared relative to each other. First, the highest coefficient value is identified and used as a reference. A predefined threshold δ then serves as the criterion to omit terms from the objective function. If inequality (4.3.5) holds for a specific term, it is removed before the optimization process begins. In this equation, M^* stands for the coefficient with maximum value.

$$M_i < \frac{M^*}{\delta}, \quad \forall M_i \in \boldsymbol{M} \tag{4.3.5}$$

4.4 Parallelism

Once the objective function is prepared, multiple search agents are initialized, each assigned different starting coordinates. These coordinates can be determined either stochastically, by randomly generating initial optimization variable values, or deterministically to ideally cover the entire domain of the function. It is evident that if a sufficient number of agents is deployed, the global minimum of a nonconvex function with an arbitrarily large domain will almost certainly be found. However, this is not always feasible in practice.

All agents are initialized at locations in a predetermined multi-dimensional region, which is reasonably restricted. However, it was not verified a priori whether the starting location adhered to the imposed constraints. It is essential to note that the domain of a function is not always identical to the domain of its derivative. This distinction applies to the logarithmic function used in the cost function of this algorithm. Although the logarithmic barrier strictly prevents a variable from crossing its constraint, this only holds if the variable is already within the set where the constraint is active. If the variable is placed on the opposite side of the barrier, the optimization method might still produce results, as the derivative of the negative logarithmic function is defined even for positive numbers. However, the result itself is not feasible, as the overlaying logarithmic function is not defined for such input.

To ensure feasibility, it is crucial to check whether an agent resides within the feasible set of possible solutions. This can be achieved by evaluating the objective function with the value of the optimization variable of an agent as its argument. If the result is not a real number, the results achieved by that agent are invalid.

Upon completion, all valid agents must be compared, to determine which result should be selected as the best value of the optimization variable. Although it might be possible to perform this comparison using properties of the Hessian matrix approximation, which is already calculated, and this would thus require no additional computation, the actual value of the objective function is inherently available, also. This value is computed during the inexact line search for α_k and the verification of the first Wolfe condition (2.2.2). Therefore, even without explicit validity checks, the calculated objective function value can be utilized to compare the results of the agents.

4.5 Overview of the Parallelized Algorithm

The schematical overview of the proposed parallel optimization algorithm is illustrated in Fig. 4.2. It consists of the following steps: offline preparation of the objective function, initialization of the search agents operating in parallel at the beginning of the online optimization, iterative identification of the minima of the objective function, and selection of the most suitable candidate for the optimal solution.

The process begins with the preparation of the objective function, which is composed of multiple quadratic norms representing the states and inputs of the controlled system, as well as logarithmic barrier functions which enforce constraints. These components are broken down into individual terms and those with negligible influence on the overall value of the cost function are removed to reduce the computational complexity.

Once the objective function is prepared, a selected number of search agents is initialized. Each agent is assigned a unique starting location within the determined starting space. Agents explore the domain using the BFGS algorithm, iterating until they reach a stationary point or other terminating criteria are met. During the optimization process, the agents identify stationary points that are potential local minima of the nonconvex objective function. The optimization process ensures that the results are valid and adhere to the constraints of the problem.

In the final stage, the results obtained by the agents are compared, and the solution most likely representing the global minimum of the nonconvex objective function is selected as the optimal candidate. This comparison takes advantage of the values computed for the objective function during the optimization process.

It is important to emphasize that the inputs to the algorithm must be normalized before optimization begins, and the outputs must be denormalized before being applied to the motor to ensure compatibility with the real system.

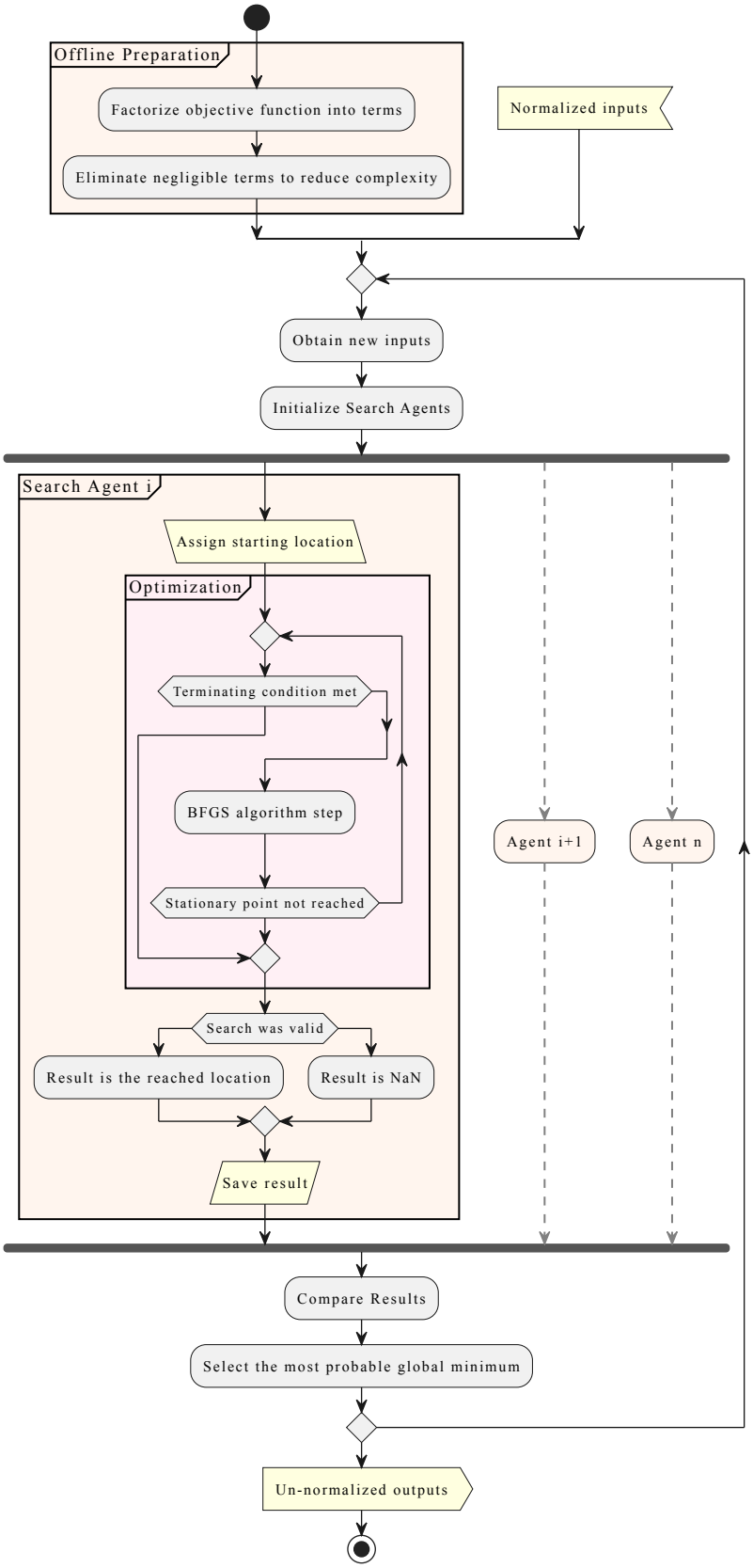

Fig. 4.2: Overview of the Parallelized Algorithm

5 Preparing the Objective Function

Preparing the objective function is a critical step in the implementation of the proposed optimization algorithm for a nonlinear MPC with a very short sampling period. One of the key methods for reducing the computational complexity of the algorithm, as described in Section 4.3, involves minimizing the number of terms in the objective function. Given that the plant model remains constant during the operation of the NMPC, this filtering process can be carried out offline, prior to the commencement of the control process. As such, the overall algorithm visualized in Fig. 4.2 is divided into two parts: offline preparation of the NMPC and online execution of the control process. This chapter focuses on the offline phase, preparing the objective function for subsequent online use.

The preparation process described in Section 4.2 and Section 4.3 includes several steps. First, the state equations for all steps of the prediction horizon are rewritten in terms of the current state values, as illustrated in (4.3.2), and substituted into the objective function (4.2.2). Next, matrix multiplication is performed to produce the factorized objective function, which comprises a large number of terms in the format (4.3.4). Finally, coefficient filtering is applied to reduce the number of terms, ensuring that the function remains computationally manageable.

As the length of the objective function increases exponentially with a growing length of the prediction horizon N, manual preparation of the function is impractical. To address this challenge, a specialized utility was developed to automate the preparation process for NMPC, streamlining the preparation of the cost function and enabling efficient offline computation.

5.1 NMPC Preparation Utility

The NMPC preparation utility was developed using the MATLAB App Designer, chosen mainly for its robust capabilities in matrix and symbolic computations.[1] The user interface, shown in Fig. 5.1, includes a variety of controls and tuning options. Parts of the user interface with significant meaning are marked with numbers in this figure, as those parts are referenced in the text.

The general workflow begins with defining the plant model by creating input (**1**) and state variables (**2**), where the inputs serve as optimization variables. The dynamics of the model are specified using discrete-time state equations (**3**), which may include constants listed separately for practicality (**4**). An objective function in the form of a quadratic form is then defined (**5**), with weight matrices also enumerated

[1] `https://www.mathworks.com/help/symbolic/symbolic-computations-in-matlab.html`

separately (**6**). An option to define those matrices dynamically exists and is triggered by the "Tunable" switch. Then, all matrix weights are defined in a standalone header and can be tuned later on without the need to reconstruct the BFGS-related function again. However, this option has an impact on the pruning of the cost function. The resulting function will contain a larger number of terms which cannot be fully enumerated due to the presence of variables, than its non-tunable counterpart. If there is a requirement that only some of the weights of the matrices should be tunable, this can be achieved using the variable input array (**7**).

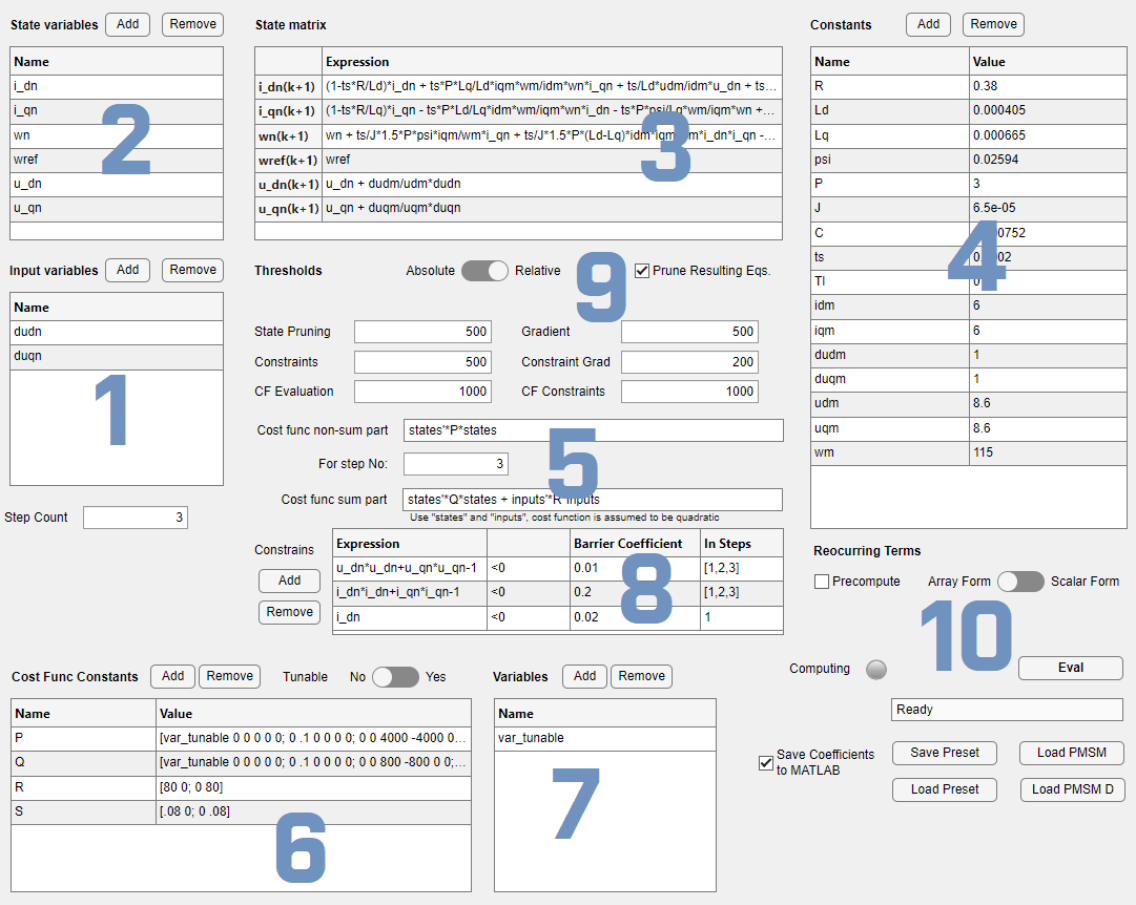

Fig. 5.1: NMPC Preparation Utility User Interface

Inequality constraints are established in the form of $c_j(\boldsymbol{x}) \leq 0$ inequalities in a dedicated table (**8**). The appropriate logarithmic barrier coefficient described in Section 4.2 is selected. These equations are computed for each step of the prediction horizon by default but can be set to be applicable to just a limited number of steps. The pruning of the cost function can be adjusted using a set of six pruning thresholds $\boldsymbol{\delta}$ in the middle of the user interface (**9**). Those thresholds can either be absolute, where all coefficients with absolute values below the threshold δ are erased, or relative, where coefficients at least $\delta\times$ less than the maximum coefficient present

in the pruned function are omitted. The state pruning threshold δ_S is used to reduce the complexity of state equations obtained by the recursive substitution discussed in Section 4.3. The constraining functions represented by logarithmic barrier functions are pruned using the constraints threshold δ_{SC}. If the option of "Prune Resulting Eqs." is not selected, other thresholds are not used, and the cost function obtained from the state- and constraints equations is not pruned further. However, if further pruning is required, a set of four additional thresholds can be enabled. The cost function evaluation threshold δ_{CF} can be used to prune the standard part of the cost function, disregarding the constraints, whose complexity can be reduced by the "CF Constraints" threshold δ_{CFC}. Gradient equations that are obtained as derivatives of the cost function with respect to the optimization variable can be pruned to a degree determined by the gradient threshold δ_{∇}. The pruning of the partial derivatives of the logarithms representing the constraints is set by the "Constraint Grad" threshold $\delta_{\nabla C}$. After all parameters have been defined, the objective function is evaluated. The results are saved as C-style header files, which can be directly integrated into the NMPC implementation code. The structure of these files and their appropriate use will be discussed in Section 5.2. There is, however, an option to find and define terms in the cost function which are utilized repeatedly. Those terms can then be precomputed and stored in a dedicated variable, which is then substituted into the function instead of the original more complex term. These variables representing the recurring terms can be saved as scalar variables with unique names or in an array. In practice, a distinction is made between recurring-term variables that contain input variables and those that do not. As the input variables are the optimization variable of the BFGS, they need to be evaluated after each iteration of the optimization, whereas those that do not contain inputs can be computed only once per iteration when the state values are obtained. The utility allows users to save and load presets, making it easy to reuse configurations. In addition, a dedicated button is provided to load a preset specifically tailored to the incremental PMSM model.

5.2 Preparation Utility Backend

The actual preparation process of the objective function will now be illustrated using examples from the MATLAB code that utilizes symbolic computations. In the first step, shown in Listing 5.1, the state equations of the provided model are expressed using symbolic variables that represent the states and inputs of the model. This involves replacing each occurrence of the state or input name in the state equations, initially represented as strings, with their corresponding symbolic variable. A similar procedure is applied to substitute constants with their numerical values. Once

all substitutions are complete, the output string is evaluated, transformed into a symbolic function, and pruned using the `filterCoefficients` function.

Listing 5.1: State equations preparation, k = 1

```
565 states = sym('state',[stepCount stateCount], 'real');
566 inputs = sym('input',[stepCount inputCount], 'real');
567 variables = sym('variables', [variablesCount 1], 'real');
672     ⋮
673 for i = 1:stateCount
674     eqn = app.AMatrix.Data(i);
675     for c = 1:height(app.constantsTable.Data)
676         eqn = strrep(eqn, app.constantsTable.Data(c,1), '('
              +app.constantsTable.Data(c,2)+')');
677     end
678     for j = 1:stateCount
679         eqn = strrep(eqn, app.stateVarTable.Data(j), '(
              states(1,'+num2str(j)+'))');
680     end
681     for j = 1:inputCount
682         eqn = strrep(eqn, app.inputTable.Data(j), '(inputs
              (1,' + num2str(j) + '))');
683     end
684     for j = 1:variablesCount
685         eqn = strrep(eqn, app.variablesTable.Data(j,1), '(
              variables(' + string(j) + ',1))');
686     end
687     statesEq(1,i) = eval(eqn); %#ok<*AGROW>
688     statesEqPruned(1,i) = filterCoefficients(app, statesEq
          (1,i), inputs, states, P, Q, R, S, app.threshold.
          Value, variables); %#ok<*AGROW>
689 end
```

The `filterCoefficients` function displayed in Listing 5.2 takes a large amount of variables as its argument, including the function to prune `symFunc`, input and state symbolic variables, as well as the weight function of the cost function. The provided `symFunc` function is first rewritten to the form described in Section 4.3 by calling the MATLAB `coeffs` function. If the weight matrices are set to be tunable, their nonzero elements are introduced as additional variables potentially present in each term. The `variable` argument represents the variables defined in section **7**. The provided threshold is then applied to each of the resulting terms in either relative or absolute form. The resulting function is then obtained by summing the remaining terms that have not been erased.

Listing 5.2: Term pruning function

```
138 function results = filterCoefficients(app, symFunc, inputs,
        states, P, Q, R, S, threshold, variables)
139     results = 0;
140     if app.TunableSwitch.Value == 'No'
141         [c,t] = coeffs(symFunc, [reshape(inputs,1,[])
                reshape(states,1,[]) variables]);
142     else
143         [c,t] = coeffs(symFunc, [reshape(inputs,1,[])
                reshape(states,1,[]), reshape(P(P~=0),1,[]),
                reshape(Q(Q~=0),1,[]), reshape(R(R~=0),1,[]),
                reshape(S(S~=0),1,[]), variables]);
144     end
145     c = double(c);
146     to = t;
147     cm = max(abs(c));
148
149     if (app.TypeSwitch.Value == 'Relative')
150         keptCoeffs = (cm ./ abs(c)) <= threshold;
151     else
152         keptCoeffs = abs(c) >= threshold;
153     end
154     results = results+c(keptCoeffs)*to(keptCoeffs)';
155 end
```

Following this, the state equations for the rest of the prediction horizon $k = [2, N]$ are derived by iteratively substituting the state equations from the previous step into the equations for the subsequent steps. This process generates a set of equations representing $\boldsymbol{x}(k)$ for $k = [1, N]$ which is stored in a variable called `statesEqPruned`. The appropriate code can be seen in Listing 5.3. Similarly, a set of constraints is prepared for each step of the prediction horizon and stored in `constraintsEqPruned`.

Listing 5.3: State equations preparation, k = 2..N

```
for s = 2:stepCount
    for i = 1:stateCount
        for c = 1:height(app.constantsTable.Data)
            eqn = strrep(eqn, app.constantsTable.Data(c,1),
                '('+app.constantsTable.Data(c,2)+')');
        end
        for j = 1:stateCount
            eqn = strrep(eqn, app.stateVarTable.Data(j), '(
               statesEqPruned('+num2str(s-1)+','+num2str(j)
               +'))');
        end
        for j = 1:inputCount
            eqn = strrep(eqn, app.inputTable.Data(j), '(
               inputs('+num2str(s)+','+num2str(j)+'))');
        end
        for j = 1:variablesCount
            eqn = strrep(eqn, app.variablesTable.Data(j,1),
                '(variables('+string(j)+',1))');
        end
        statesEq(s,i) = eval(eqn);
        statesEqPruned(s,i) = filterCoefficients(app,
           statesEq(s,i), inputs, states, P, Q, R, S, app.
           threshold.Value, variables);
    end
end
```

Listing 5.4: Weight matrices preparation

```
sz = width(statesWeightMatrix)*height(statesWeightMatrix);
Q = sym('q_mat', size(statesWeightMatrix));
weightMatrixDefines = '// Q Matrix ';
weightMatrixDefinesTmp = strings(sz, 1);
for i = 1:height(statesWeightMatrix)
    for j = 1:width(statesWeightMatrix)
        if (statesWeightMatrix(i,j) == 0)
            Q(i,j) = sym(0);
            continue
        end
        weightMatrixDefinesTmp((i-1)*height(
           statesWeightMatrix)+j) = '#define q_mat'+num2str
           (i)+'_' + num2str(j) + " " + string(
           statesWeightMatrix(i,j));
    end
end
weightMatrixDefines = [weightMatrixDefines;
   weightMatrixDefinesTmp];
```

The next step involves identifying and transforming the weight matrices of the objective function from their string representation in the user interface into MATLAB matrices. If the option for tunable matrices is selected, each nonzero element in these matrices is replaced with a symbolic variable. This allows the matrices to be tuned during the NMPC algorithm implementation, as they are represented in the objective function by variables rather than fixed numerical values. This process is illustrated by Listing 5.4 for the matrix $\boldsymbol{Q}$.

Finally, the objective function is obtained by factorizing the expressions for each step of the prediction horizon as illustrated by Listing 5.5. The part of the function that represents the constraints is prepared separately in Listing 5.6.

Listing 5.5: Cost function preparation, standard part

```
761 nonSumStep = app.ForstepNoEditField.Value;
762 cfPruned = statesEqPruned(nonSumStep,:)*P*statesEqPruned(
    nonSumStep,:)';
763 cfPruned = cfPruned + inputs(nonSumStep,:)*S*inputs(
    nonSumStep,:)';
764 for i = 1:stepCount
765     if i == nonSumStep
766         continue
767     end
768     cfPruned = cfPruned + statesEqPruned(i,:)*Q*
        statesEqPruned(i,:)';
769     cfPruned = cfPruned + inputs(i,:)*R*inputs(i,:)';
770 end
```

Listing 5.6: Cost function preparation, constraints part

```
778 cfConstraintsPruned = 0;
779 for i = 1:stepCount
780     for j = 1:constraintsCount
781         forWhichSteps = eval(app.constraintTable.Data(j,4))
            ;
782         if (ismember(i, forWhichSteps))
783             constraintCoeff = double(app.constraintTable.
                Data(j,3));
784             cfConstraintsPruned = cfConstraintsPruned-
                constraintCoeff*log(-constraintsEqPruned(i,j
                ));
785         end
786     end
787 end
```

The next step involves computing the gradient of the objective function by differentiating it with respect to the optimization variable. This is done using the

expression `diff(cfPruned, inputs(j,i))`, where `cfPruned` represents the prepared objective function, `j` denotes the step count, and `i` specifies the number of the optimization variable. Once the gradient is calculated, the coefficient filter is applied again if the setting to prune the resulting function is turned on. A similar filtering process is applied to the part of the objective function that represents the constraints. This function is composed of multiple logarithmic terms, which, when differentiated according to (5.2.1), result in a rational function. The coefficient filtering is then applied separately to the numerator and the denominator, eliminating those with minimal influence on the final value of the rational function.

$$\frac{\partial \log\left(f(x,y)\right)}{\partial x} = \frac{\partial f(x,y)}{\partial x} \cdot \frac{1}{f(x,y)} \tag{5.2.1}$$

Before exporting the derived symbolic expressions as callable C-style functions, it is essential to replace the terms containing exponentiation in the MATLAB `x^y` notation with the C-style `x*x*···*x`. This conversion is accomplished using the `rewritePowers` function that takes advantage of MATLAB's string pattern capabilities, as illustrated in Listing 5.7. First, the exponents are extracted using a suitable pattern and stored in `strPW`. Similarly, the bases are captured and stored in `strB`. Finally, `strW` contains the complete terms to be located in the function and replaced. The resulting callable functions compute the gradient of the objective function with the value of the optimization variable provided as an argument, and their final form is shown in Listing 5.12. The numerical coefficients are defined before hand with `#define`. Inside the function, the terms of the standard part of the objective function are listed first and the terms representing the derivative of a logarithm are listed last.

Listing 5.7: Rewriting the terms containing exponentiation

```
182 function [alreadyDefinedPwrs, pwrsDefinition, results,
       alreadyDefinedPwrsInputs, pwrsDefinitionInputs] =
       rewritePowers(app, inputTextArray, alreadyDefinedPwrs,
       alreadyDefinedPwrsInputs)
183     results = inputTextArray;
184     pwrsDefinition = '';
185     pwrsDefinitionInputs = '';
186     for k = 1:length(results)
187         strP = results(k);
188         pat = '\(([^)]+)\)\^(\d+)';
189
190         strP = regexp(strP, pat, 'match');
191         for i = 1:length(strP)
192             strPi = extractBetween(strP(i), '(', ')',
                   Boundaries='inclusive');
193             strE = str2double(extractAfter(strP(i), '^'));
194             strRep = strPi;
195             for j = 1:strE-1
196                 strRep = strRep + '*' + strPi;
197             end
198             results(k) = replace(results(k), strP(i), '(' +
                   strRep + ')');
199         end
200         ⋮
```

The `rewritePowers` function is also responsible for the potential identification of recurring terms and their subsequent substitution. The code that performs this substitution for recurring exponential terms is listed in Listing 5.8. The name for this potential term is constructed in a predetermined manner on line 216. Its occurrence is then counted in `alreadyDefinedPwrs` and `alreadyDefinedPwrsInputs` arrays which were provided to the function as its argument. The first represents the set of recurring terms that do not contain any input variables, while the latter is comprised of terms that do. Then in lines 231-237 it is determined which category the term belongs to. If it does contain any input variables and is also not yet defined, its definition is created and stored to `pwrsDefinitionInputs` array (lines 238-241). Likewise, if it contains input, it is stored in `pwrsDefinition` (lines 242-246). The term is then replaced by the newly created variable in the original equation (lines 247-251). If the term has already been defined and is found in the `alreadyDefinedPwrs` or `alreadyDefinedPwrsInputs` arrays, it is just replaced in the original equation without its redefinition.

Listing 5.8: Rewriting the recurring terms

```
215 if (app.PrecomputeCheckBox.Value)
216     varName = 'pwr_' + strTR(l) + '_' + strPW(l);
217     cnt = 0;
218     cntI = 0;
219     for m = 1:length(alreadyDefinedPwrs)
220         if (alreadyDefinedPwrs(m) == varName)
221             cnt = cnt + 1;
222             break;
223         end
224     end
225     for mI = 1:length(alreadyDefinedPwrsInputs)
226         if (alreadyDefinedPwrsInputs(mI) == varName)
227             cntI = cntI + 1;
228             break;
229         end
230     end
231     containsInputs = false;
232     for n = 1:height(app.inputTable.Data)
233         if (contains(rep(l), app.inputTable.Data(n,1)))
234             containsInputs = true;
235             break;
236         end
237     end
238     if (containsInputs && cntI==0)
239         mI = mI + 1;
240         pwrsDefinitionInputs = [pwrsDefinitionInputs; '
              floatVarsInputs['+string(mI-2)+']='+rep(l) + ';
              //' + varName];
241         alreadyDefinedPwrsInputs = [
              alreadyDefinedPwrsInputs; varName];
242     elseif(~containsInputs && cnt==0)
243         m = m + 1;
244         pwrsDefinition = [pwrsDefinition; 'floatVars['+
              string(m-2)+']='+rep(l)+'; //'+varName];
245         alreadyDefinedPwrs = [alreadyDefinedPwrs; varName];
246     end
247     if (containsInputs)
248         rep(l) = 'floatVarsInputs[' + string(mI-2) + ']';
249     else
250         rep(l) = 'floatVars[' + string(m-2) + ']';
251     end
252 end
```

The body of the future C function used to compute the gradient values is constructed as demonstrated by Listing 5.9. This function will be complemented

with a heading in the form of: `void gradVal (float gradOut[n], variables...)`. The arrays of numeric coefficients and their corresponding variable terms are transformed to the equation they represent by the `writeEqnFromArrays` function which defines all coefficient as float variables with the `.f` suffix. It also performs other code-readability enhancing operations.

Listing 5.9: Gradient evaluation function body

```
940 varName = 'gradOut['+string((i-1)*stepCount+j-1)+']';
941 tmp = writeEqnFromArrays(app, c, to);
942 funcBody = [varName+'='+tmp+';'; ''; strings(numOfDenParts,
        1)];
943 for k = 1:numOfDenParts
944     tmp = writeEqnFromArrays(app, denPartCoeffs{k},
            denPartTerms{k});
945     tmpStr = varName+'+=('+numPart(k)+')/('+tmp+');';
946     if (contains(tmpStr, 'Inf'))
947         tmpStr = '';
948         coeffCountNotComp = coeffCountNotComp+2;
949     end
950     funcBody (k + 2) = tmpStr;
951 end
952 funcBodyGrad = [funcBodyGrad; '/* ----- */'; funcBody];
```

In a fashion similar to the gradient function preparation, the cost function evaluation function is also prepared. In addition to the standard part of the function and the logarithm functions, the constraining equations are also prepared to be evaluated by the code in Listing 5.10. The produced C code is by default set to compute just the logarithms as a representation of the constraints that correspond to the gradients computed, and the direct constraint evaluation is disabled. The predicted state values that need to be computed in order to evaluate the constraining equation at each prediction step are recursively identified on lines 1201-1226. The state equation is then defined on lines 1232-1249. On line 1251 this definition is added to the body of the function and finally the code that is used to check whether the constraint has been violated is constructed.

Listing 5.10: Cost function constraints evaluation

```
constraintStates = false(stateCount, 1);
for k = 1:constraintsCount
    eqn = app.constraintTable.Data(k,1);
    for l = 1:stateCount
        constraintStates(l) = constraintStates(l) + count(eqn, app.
            stateVarTable.Data(l,1));
    end
end

while (1)
    constraintStatesPrev = constraintStates;
    for k = 1:stateCount
        if (constraintStates(k) > 0)
            [ctmp, ttmp] = coeffs(statesEqPruned(1, k));
            eqn = writeEqnFromArrays(app, double(ctmp), string(ttmp));
            for m = 1:stateCount
                eqn = strrep(eqn, 'state1_'+num2str(m), app.stateVarTable.
                    Data(m));
            end
            for l = 1:stateCount
                constraintStates(l) = constraintStates(l)+count(eqn, app.
                    stateVarTable.Data(l,1));
            end
        end
    end
    if (constraintStatesPrev == constraintStates)
        break
    end
end

varNames = '';
varEqs = '';
for k = 1:stateCount
    if (constraintStates(k) > 0)
        for l = 1:stepCount
            ⋮
            varEqs = [varEqs; app.stateVarTable.Data(k,1)+string(l)+'='+
                eqTmp+';'];
        end
    end
end
funcBodyEvalConstraints = [funcBodyEvalConstraints; 'float '+extractBefore
    (varNames, strlength(varNames)-1)+';'; varEqs(2:end)];
for k = 1:constraintsCount
    for l = 1:stepCount
        forWhichSteps = eval(app.constraintTable.Data(k,4));
        if (ismember(l, forWhichSteps))
            varEqs = app.constraintTable.Data(k,1);
            for m = 1:stateCount
                varEqs = replace(varEqs, app.stateVarTable.Data(m,1), app.
                    stateVarTable.Data(m,1)+string(l));
            end
            funcBodyEvalConstraints = [funcBodyEvalConstraints; 'if ('+
                varEqs+' > 0) return 1e7;'];
        end
    end
end
```

In addition, several functions required by the BFGS algorithm are generated. These include `obtainP` for computing the BFGS step, `obtainH` for updating the inverse Hessian approximation and `obtainpktGf` for calculating $\boldsymbol{p}_k^T \nabla f(\boldsymbol{x_k})$ when verifying Wolfe (2.2.3) or Goldstein (2.2.5) conditions. An example of this process is given in Listing 5.11 demonstrating the process of constructing the `obtainP` function. $\boldsymbol{H}_k$ and $\nabla f(\boldsymbol{x})$ are defined as symbolic variables on lines 969 and 970. Then a string representation of the equation $\boldsymbol{s}_k = -\boldsymbol{H}_k \nabla f(\boldsymbol{x})$ is obtained. As the previously defined symbolic arrays are one-based indexed, the regular expression operation on line 976 transforms these into zero-based indexed C arrays. An example of C code generated by this preparation program can be seen in Listing 5.12 with various parts omitted for illustration purposes. If tunable matrices were selected during setup, the definitions for these matrices are stored in a separate file, allowing them to be adjusted without the need to regenerate all files repeatedly.

Listing 5.11: BFGS step direction function

```
969 Hk = sym('H', [1 inputCount^2*stepCount^2]);
970 gradF = sym('gradF', [inputCount*stepCount 1]);
971 Hk = reshape(Hk, [inputCount*stepCount, inputCount*
        stepCount]);
972 pk = -Hk*gradF;
973 pk = string(pk);
974
975 for k = 1:length(pk)
976     pk(k) = regexprep(pk(k), '(?<=[A-Za-z])(\d+)', '${
            sprintf(''[%d]'', str2double($1) - 1)}');
977 end
978 for i = 1:height(pk)
979     pk(i) = 'p['+string(i-1)+']='+pk(i)+';';
980 end
981
982 pk = ['void obtainP (float* p, float* H, float* gradF) {';
        pk; '}'];
```

Listing 5.12: Example of resulting C code

```
#include "math.h"

// Created with 500 Relative threshold

static float floatVars[38];
static float floatVarsInputs[48];

void updateFloats (float i_dn,float i_qn,...) {
    floatVars[0] = u_dn*variables1; //reg_u_dn_variables1
    floatVars[1] = i_dn*variables1; //reg_i_dn_variables1
        ...
void updateFloatsInputs (float i_dn,float i_qn,...,float dudn1,float dudn2
    ,...) {
    floatVarsInputs[0] = dudn3*variables1; //reg_dudn3_variables1
    floatVarsInputs[1] = duqn3*floatVars[9]; //reg_duqn3_i_dn_wn
        ...
static float var_tunable = 1;

void gradVal (float gradOut[6], float i_dn,float i_qn,...,float dudn1,
    float dudn2,...) {

    /* ----- */
    gradOut[0] =  + 160.0081f*dudn1 + 0.54874f*floatVars[1] + 1.2113f*
        floatVars[0];
    gradOut[0] += ( + dudn1 + dudn2 + 8.6000f*u_dn)/( - 50.0f*
        floatVarsInputs[16] ...);
        ...
void obtainP (float* p, float* H, float* gradF) {
    p[0] = - H[0]*gradF[0] - H[6]*gradF[1] - H[12]*gradF[2] - ...;
        ...
void obtainHk (float* Hk, float* H, float* y, float* s) {
    float var_sy = s[0]*y[0] + s[1]*y[1] + s[2]*y[2] + s[3]*y[3] + ...;
    float var_sy_sq_inv = 1/(var_sy*var_sy);
    float var_sy_inv = 1/var_sy;
    float var_Hy[6] = { 0 };
        ...
float obtainpktGf (float* pk, float* gradF) {
    return gradF[0]*pk[0] + gradF[1]*pk[1] + gradF[2]*pk[2] + ...;
}
float costFunctionValue (float i_dn,float i_qn,float wn,float wref,float
    u_dn,float u_qn,float dudn1,float dudn2,float dudn3,float duqn1,float
    duqn2,float duqn3) {
    float ret =  + 11.4158f*floatVars[3]*var_tunable ...;

    ret += (-0.0100)*logf( - 0.013521f*floatVarsInputs[16] + ...);
    /*
    float i_dn1, i_dn2, i_dn3, ...;
    i_dn1 =  + 0.082305f*dudn1 + 0.1133f*floatVars[13] + ...;
    i_dn2 =  + 0.082305f*dudn2 + 0.1133f*i_qn1*wn1 + ...;
        ...
    if (i_dn1*i_dn1+i_qn1*i_qn1-1 > 0) return 1e7;
    if (i_dn2*i_dn2+i_qn2*i_qn2-1 > 0) return 1e7;
    if (i_dn3*i_dn3+i_qn3*i_qn3-1 > 0) return 1e7;
    */
return ret; }
```

6 NMPC Implementation

Although earlier chapters referenced the implementation of the proposed algorithm to provide context, the descriptions were presented in an overall general manner to ensure that the approach could be applied to a variety of scenarios. However, in this section, the focus is on detailing the specific implementation of the proposed NMPC algorithm.

The simulation structure developed in [31], which combines the robust capabilities of the Simulink environment with C-style functions, was selected to implement, test, and tune the nonlinear MPC. This structure, illustrated in Fig. 6.1, comprises two distinct components: one implementing the continuous PMSM model (see (3.0.1)) and the other generating control inputs using the NMPC. The transformation of the quantities computed in the rotating dq reference frame to the three-phase reference frame of the motor was performed by a set of Clarke-Park transformation equations (6.0.1).

$$x_a = x_d \cos(\theta) - x_q \sin(\theta) \tag{6.0.1a}$$

$$x_b = x_d \cos\left(\theta - \frac{2\pi}{3}\right) - x_q \sin\left(\theta - \frac{2\pi}{3}\right) \tag{6.0.1b}$$

$$x_c = x_d \cos\left(\theta + \frac{2\pi}{3}\right) - x_q \sin\left(\theta + \frac{2\pi}{3}\right) \tag{6.0.1c}$$

Although the discrete plant model used in the NMPC was discussed earlier in Section 3.1, the final equations expanded by a new system state ω_{rn}, which is the normalized speed reference ω_{ref}, are provided here (6.0.2) again for clarity and completeness with (k) argument parts on the right sides of these equations being omitted for better readability. Assuming the objective function (4.2.1), (4.2.2), by introducing the reference signal as an additional state ω_{rn}, minimizing the control error becomes part of the optimization objective. This is ensured by modifying the quadratic-form weight matrices (namely $\boldsymbol{P}$ and $\boldsymbol{Q}$), so that after factorization of the objective function, a penalization term $(\omega_{mn} - \omega_{rn})^2$ is produced. Note that the modified matrices are no longer diagonal. The states of the system are therefore $\boldsymbol{x} = \{i_{dn}, i_{qn}, \omega_{mn}, \omega_{rn}, u_{dn}, u_{qn}\}$, and the inputs are $\boldsymbol{u} = \{\Delta u_{dn}, \Delta u_{qn}\}$. At this stage, it is also essential to assign numerical values to the constants in these equations (for both continuous and discrete models). These values, listed in Table 6.1, were chosen to represent the same motor as described in [32]. If the stator inductance components L_d and L_q differ, as in this case, such a motor is called an internal permanent magnet synchronous motor (IPMSM).

Table 6.1: IPMSM constants values

Contant	Value
R_S	$0.38\,\Omega$
L_d	$0.405\,\mu\mathrm{H}$
L_q	$0.665\,\mu\mathrm{H}$
Ψ_{PM}	$25.94\,\mathrm{mWb}$
J	$6.5 \cdot 10^{-6}\,\mathrm{kg} \cdot \mathrm{m}^2$
U_{MAX}	$8.6\,\mathrm{V}$
I_{MAX}	$6\,\mathrm{A}$
t_s	$200\,\mu\mathrm{s}$

$$i_{dn}(k+1) = \left(1 - t_s \frac{R_s}{L_d}\right) i_{dn} + t_s P_p \frac{L_q}{L_d} \frac{i_q^* \omega_m^*}{i_d^*} \omega_{mn} i_{qn} + \frac{t_s}{L_d} \frac{u_d^*}{i_d^*} u_{dn} \tag{6.0.2a}$$

$$\begin{aligned} i_{qn}(k+1) = &\left(1 - t_s \frac{R_s}{L_q}\right) i_{qn} - t_s P_p \frac{L_d}{L_q} \frac{i_d^* \omega_m^*}{i_q^*} \omega_{mn} i_{dn} + \frac{t_s}{L_q} \frac{u_d^*}{i_q^*} u_{qn} - \\ &- t_s P_p \frac{\Psi_{PM}}{L_q} \frac{\omega_m^*}{i_q^*} \omega_{mn} \end{aligned} \tag{6.0.2b}$$

$$\omega_{mn}(k+1) = \omega_{mn} + \frac{t_s}{J} \left(\frac{3}{2} P_p \left[\Psi_{PM} \frac{i_q^*}{\omega_m^*} i_{qn} + (L_d - L_q) \frac{i_q^* i_d^*}{\omega_m^*} i_{dn} i_{qn} \right] - \frac{T_l}{\omega_m^*} \right) \tag{6.0.2c}$$

$$\omega_{rn}(k+1) = \omega_{rn} \tag{6.0.2d}$$

$$u_{dn}(k+1) = u_{dn} + \frac{\Delta u_d^*}{u_d^*} \Delta u_{dn} \tag{6.0.2e}$$

$$u_{qn}(k+1) = u_{qn} + \frac{\Delta u_q^*}{u_q^*} \Delta u_{qn} \tag{6.0.2f}$$

To integrate the C implementation of the NMPC into the Simulink environment, a level-2 C-MEX S-Function is employed. This MATLAB API allows code written in C, C++, or Fortran to seamlessly interact with Simulink via custom-generated simulation blocks with multiple inputs and outputs. These blocks can process signals generated in Simulink and enable run-time debugging. The basic structure of the C code, shown in Listing 6.1, is based on multiple callback methods to establish the interface between Simulink and the C language. In the `mdlInitializeSizes` method, the simulation block is configured with its number of inputs and outputs, along with their dimensions (scalar or vector). Additionally, S-Function states, which store values throughout the simulation, are initialized here. The sampling period is specified in the `mdlInitializeSampleTimes` method. During simulation, the `mdlOutputs` method is invoked at each step, while `mdlStart` and `mdlTerminate` are called at the start and end of the simulation, respectively. Consequently, the core implementation of the NMPC algorithm resides in the `mdlOutputs` method, with its initialization distributed across the `mdlInitializeSizes`, `mdlInitializeSampleTimes`, and `mdlStart` methods.

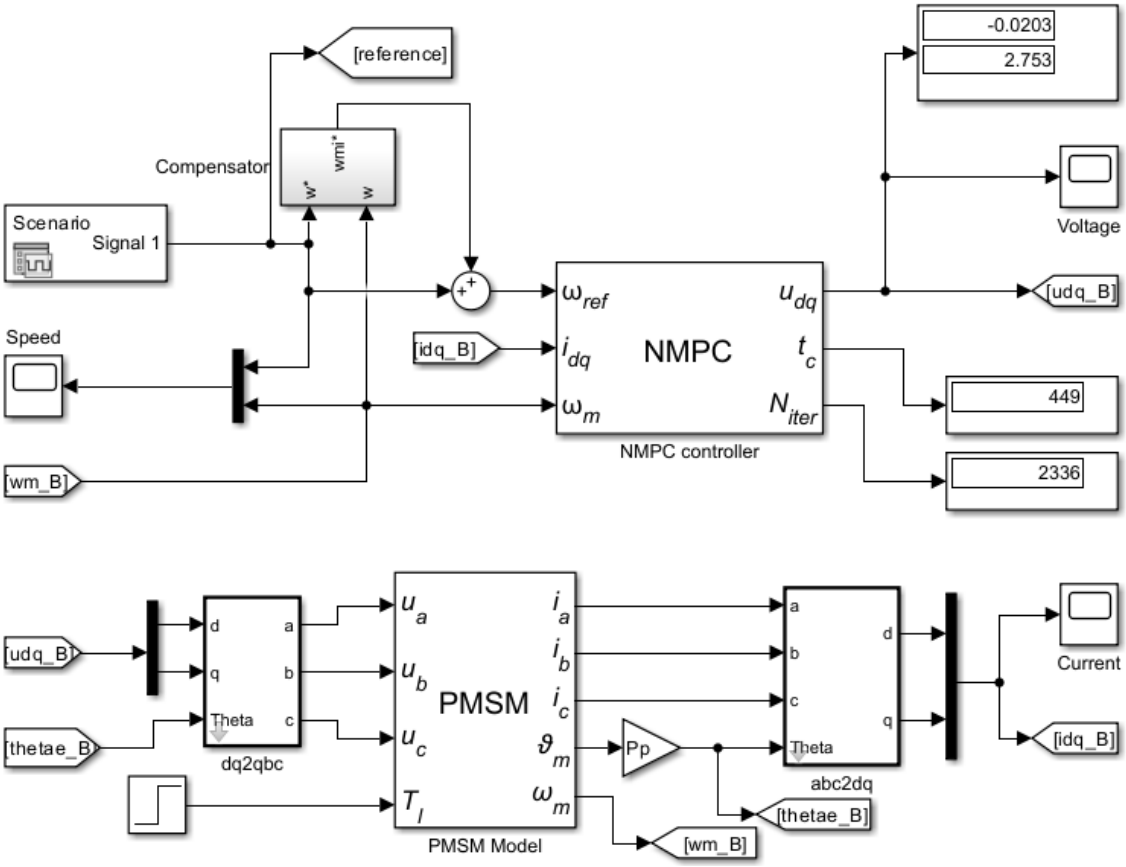

Fig. 6.1: Simulation structure

Listing 6.1: C-MEX S-Function example

```
#define S_FUNCTION_NAME  /* sfunction_name */
#define S_FUNCTION_LEVEL 2
#include 'simstruc.h'

static void mdlInitializeSizes(SimStruct *S) {}
static void mdlInitializeSampleTimes(SimStruct *S) {}
static void mdlStart(SimStruct *S) {}

/* <additional S-function routines/code> */

static void mdlOutputs(SimStruct *S) {}
static void mdlTerminate(SimStruct *S) {}

#ifdef MATLAB_MEX_FILE    /* Compiled as a MEX-file? */
#include 'simulink.c'     /* MEX-file interface mechanism
   */
#else
#include 'cg_sfun.h'      /* Code generation registration
   */
#endif
```

The inputs and outputs of the NMPC block were configured as follows. The block features three primary inputs that supply the current values of i_d, i_q and ω_m, and a reference signal ω_{ref}. In addition to the control signals u_d and u_q as output, two additional outputs were included: t_c, representing the total computation time for each sampling period, and N_{iter}, indicating the total number of iterations. To calculate the average number of iterations per agent, N_{iter} should be divided by the total number of agents.

Crucial parameters are predefined using the `#define` macro in Listing 6.2. However, a clarification is needed regarding the nomenclature: the inputs to the NMPC block differ from the inputs to the plant model used within the NMPC, and the same distinction applies to outputs. While the values of i_d, i_q, ω_{ref} and ω_m are obtained during each function call, the cumulative values of u_d and u_q must be stored in the block throughout the simulation. To achieve this, two discrete states were defined. The key parameters of the BFGS algorithm are also defined in advance. These include `NMPC_ErrorThreshold`, which sets the termination threshold ϵ (see Algorithm 4); `NMPC_oneStepMaxIterations`, the maximum number of iterations per search agent; and `NMPC_startingPointsMaxCount`, the total number of search agents. In addition, the maximum values for the quantities used in normalization and denormalization are specified.

Listing 6.2: BFGS in C: parameter definition

```
#define NMPC_Horizon 4
#define NMPC_numOfStates 2
#define NMPC_numOfInputs 4
#define NMPC_numOfOutputs 2

#define NMPC_ErrorThreshold 1e-3
#define NMPC_oneStepMaxIterations 1e2
#define NMPC_startingPointsMaxCount 100
#define NMPC_oneStepMaxVoltageChange 1

#define ID_MAX 6
#define IQ_MAX 6
#define UD_MAX 8.6
#define UQ_MAX 8.6
#define DUD_MAX 1
#define DUQ_MAX 1
#define W_MAX 150
```

6.1 Sequential implementation

In this section, a detailed description of the sequential implementation of the NMPC algorithm is presented. The code provided here is part of the `mdlOutputs` method described previously. Note that this method is repeatedly called at each sampling period of the simulation.

At the start of each method call (see Listing 6.3), the input and state values are retrieved and assigned to variables for subsequent use. For instance, the `initialValues` array represents the PMSM model states, while `in_idq`, `in_w`, and `in_ref` reference the NMPC block inputs. Similarly, `o[0]` and `o[1]` reference the S-Function's discrete states used to store voltage values. The variables for the BFGS algorithm are defined, including `gdVarVal`, which stores the value of the optimization variable (with size determined by the number of inputs and the length of the prediction horizon). Other variables in BFGS include `gdGradVal` for gradient values, `H` and `Hk` for inverse Hessian approximation, and `s` and `y` as supporting variables. Their respective roles are detailed in Section 4.1. Furthermore, `gdVarValRes` stores all the results achieved by the search agents, and `cfRes` records the value of the objective function for each agent. The variables are defined as single-precision `float` data types.

Listing 6.3: BFGS in C: obtaining state, input and other variable values

```
199 initialValues[IDPOS] = (float)(*in_idq[0]) / ID_MAX;
200 initialValues[IQPOS] = (float)(*in_idq[1]) / IQ_MAX;
201 initialValues[WPOS] = (float)(**in_w) / W_MAX;
202 initialValues[WREFPOS] = (float)(**in_ref) / W_MAX;
203 initialValues[4] = o[0] / UD_MAX;
204 initialValues[5] = o[1] / UQ_MAX;
205
206 float gdVarVal[NMPC_numOfOutputs * NMPC_Horizon];
207 float gdGradVal[NMPC_numOfOutputs * NMPC_Horizon];
208 float H[NMPC_numOfOutputs*NMPC_numOfOutputs * NMPC_Horizon*
      NMPC_Horizon];
209 float Hk[NMPC_numOfOutputs*NMPC_numOfOutputs * NMPC_Horizon
      *NMPC_Horizon];
210 float p[NMPC_numOfOutputs * NMPC_Horizon];
211 float y[NMPC_numOfOutputs * NMPC_Horizon];
212
213 float gdVarValRes[NMPC_startingPointsMaxCount][
      NMPC_numOfOutputs*NMPC_Horizon];
214 float cfRes[NMPC_startingPointsMaxCount];
```

The core of the algorithm is implemented using two nested loops, as shown in Listing 6.4. The outer loop iterates over the search agents, while the inner loop handles the search process. For each new agent, the matrix $\boldsymbol{H}$ is reset to its initial value and the starting location of the agent is chosen stochastically, ensuring that at least one agent starts at the origin. In the inner loop, the agent follows the process described in Algorithm 4.

First, the gradient is calculated and the step direction is determined using the `obtainP` function (lines 263, 267). Subsequently, an inexact line search is performed for fifteen potential values of the step size $\alpha = \{1, 0.75, 0.5, 0.3, 0.2, 0.1, 0.05, 0.025, 0.0125, 0.00625, 0.003125, 0.0015625, 0.00078125, 0.000390625, 0.00001953125\}$ which can be seen on lines 273-293. It was discovered during implementation that satisfactory control results can be obtained even when using just the first Wolfe condition (also known as the Armijo condition). As checking the second condition would increase the computational demands, it was, therefore, omitted from the line search algorithm. The optimization variable (that is, the location of the agent) is then updated with the identified optimal step size `alpha` (line 296). Next, the variable `y` is computed (line 301), and `Hk` is updated via the `obtainHk` function and subsequently assigned back to the `H` matrix used in step direction determination (lines 317-320). Note that during the first iteration of the algorithm, the inverse Hessian approximation is replaced according to (4.1.2) (lines 303-316). The terminating conditions of the inner convergence loop include exceeding the maximum number

of iterations or successfully finding a local minimum where the error is below the `NMPC_errorThreshold`.

The results, including the location of the minimum and its value, are stored in `gdVarValRes` and `cfRes`. If the resulting location of the agent is found to be outside the imposed constraints, a large magnitude value is assigned to `cfRes` to represent the infeasible result.

After all agents complete their searches, the most likely optimum is identified in Listing 6.5 on lines 331-338. The value of the best results is updated accordingly (line 340), and a simple check is performed to ensure its validity (lines 342-345). Finally, the optimization variables corresponding to the voltage increment values in the first prediction horizon step are denormalized (lines 346-347), added to the previous sampling period's voltage values, and assigned as the current control outputs.

Listing 6.4: BFGS in C: optimization

```
245 do {
246     itCount = 0;
247→260     ⋮
261     for (i = 0; i < NMPC_numOfOutputs * NMPC_Horizon; ++i) {
262         gdVarVal[i] = startingLocations[stPointCount * NMPC_numOfOutputs *
                NMPC_Horizon + i];}
263     gradVal(gdGradVal, initialValues[IDPOS], ...);
264     cfVal = costFunctionValue(initialValues[IDPOS], ...);
265
266     while(error > NMPC_ErrorThreshold && itCount <
            NMPC_oneStepMaxIterations) {
267         obtainP(p, H, gdGradVal);
268         error = 0;
269→272         ⋮
273         int lineSearchIt = 0;
274         float pktGf = obtainpktGf(p, gdGradVal);
275         float f_a[NMPC_lineSearchMaxIterations] = { 0 };
276         float alphaT[15] = {1.0f, 0.75f, 0.5f, 0.3f,...};
277         bool lineSearchSuccess[15];
278         // Armijo Backtracking Line Search
279         for (;lineSearchIt<NMPC_lineSearchMaxIterations;++lineSearchIt) {
280             for (j = 0; j < NMPC_numOfOutputs * NMPC_Horizon; ++j) {
281                 gdVarValA[j] = gdVarVal[j]+alphaT[lineSearchIt]*p[j];
282             }
283             f_a[lineSearchIt] = costFunctionValue(initialValues[IDPOS
                    ],...);
284             lineSearchSuccess[lineSearchIt] = f_a[lineSearchIt]<=cfVal+c1*
                    alphaT[lineSearchIt]*pktGf;
285         }
286         alpha = alphaT[14];
287         cfVal = f_a[14];
288         for (i = 13; i >= 0; --i) {
289             if (lineSearchSuccess[i]) {
290                 alpha = alphaT[i];
291                 cfVal = f_a[i];
292             }
293         }
294
```

```
295         for (j = 0; j < NMPC_numOfOutputs * NMPC_Horizon; ++j) {
296             gdVarVal[j] += alpha * p[j];
297         }
298         gradVal(gdGradValNext, initialValues[IDPOS], ...);
299 
300         for (i = 0; i < NMPC_numOfOutputs * NMPC_Horizon; ++i) {
301             y[i] = -gdGradVal[i] + gdGradValNext[i];
302         }
303         if (itCount == 0) {
304             float yTs = y[0]*s[0] + y[1]*s[1] + ... + y[7]*s[7];
305             float yTy = y[0]*y[0] + y[1]*y[1] + ... + y[7]*y[7];
306             float ytsyTy = yTs / yTy;
307             H[0] = ytsyTy;
308→314             ⋮
315             H[63] = ytsyTy;
316         }
317         obtainHk(Hk, H, y, p);
318         for (i = 0; i < NMPC_numOfOutputs * NMPC_numOfOutputs *
                NMPC_Horizon * NMPC_Horizon; ++i) {
319             H[i] = Hk[i];
320         }
321         ++itCount;
322     }
323     errorRes[stPointCount] = cfVal;
324     if (isnan(errorRes[stPointCount]))
325         errorRes[stPointCount] = 1e15;
326     for (i = 0; i < NMPC_numOfOutputs * NMPC_Horizon; ++i) {
327         gdVarValRes[stPointCount][i] = gdVarVal[i];
328     }
329 } while (++stPointCount < NMPC_startingPointsMaxCount);
```

Listing 6.5: BFGS in C: Outputs

```
331 float min_cf = 1e5;
332 int_T id = 0;
333 for (i = 0; i < stPointCount; ++i) {
334     if (cfRes[i] < min_cf) {
335         min_cf = cfRes[i];
336         id = i;
337     }
338 }
339 for (i = 0; i < NMPC_numOfOutputs * NMPC_Horizon; ++i) {
340     gdVarVal[i] = gdVarValRes[id][i];
341 }
342 if (isnan(gdVarVal[0]) || isinf(gdVarVal[0]))
343     gdVarVal[0] = 1;
344 if (isnan(gdVarVal[3]) || isinf(gdVarVal[3]))
345     gdVarVal[3] = 1;
346 gdVarVal[0] *= DUD_MAX;
347 gdVarVal[3] *= DUQ_MAX;
348 out_u0 = ssGetOutputPortRealSignal(S, 0);
349 out_u0[0] = initialValues[4]*UD_MAX + gdVarVal[0];
350 out_u1[1] = initialValues[5]*UQ_MAX + gdVarVal[3];
```

6.2 Parallel implementation

Although the algorithm was implemented sequentially on a central processing unit (CPU) for better tuning and debugging options, to exploit the potential of parallelism, an implementation on a graphics processing unit (GPU) was chosen. The NMPC algorithm was deployed on a GPU based on the Pascal architecture Nvidia GTX 1050Ti, introduced in 2016, with 6.1 CUDA compute capability. It features 768 CUDA cores, 4 GB of GDDR5 memory [47], and a clock speed of up to 1455 MHz in boost mode [3]. Other important parameters relevant to the implementation of the optimization algorithm, as well as a comparison with a newer GPU in a similar category (RTX 4060), are listed in Table 6.2.

Table 6.2: GTX 1050 Ti and RTX 4060 Comparison

Parameter	GTX 1050 Ti (ASUS Expedition O4G)	RTX 4060
Architecture	Pascal (GP107)	Ada Lovelace (AD107)
CUDA Cores	768	3,072
Base Clock	1,290 MHz	1,830 MHz
Boost Clock	1,391 MHz	2,460 MHz
Memory	4 GB GDDR5	8 GB GDDR6
Memory Bandwidth	112 GB/s (128-bit, 7 Gbps)	288 GB/s (128-bit, 15 Gbps)
FP32 Performance	∼2.1 TFLOPS	∼15.1 TFLOPS
FP64 Performance	∼0.066 TFLOPS (1/32 FP32)	∼0.24 TFLOPS (1/64 FP32)
Streaming Multi-processors (SMs)	6	24
L2 Cache	1 MB	24 MB
Compute Capability	6.1	8.9
Power Consumption (TGP)	75W	115W
Sources	[47], [3]	[48], [49], [51]

Were the newer-architecture-based RTX 4060 GPU used to implement the optimization algorithm, it would have the following implications for overall performance. Apart from higher base and boost clock speeds, the RTX 4060 offers four times more cores, enhancing parallel computation capabilities compared to the GTX 1050 Ti. This results in more than a 7× increase in single-precision floating point potential performance. The significantly larger cache size, combined with a larger memory bandwidth, would also noticeably improve the performance of the implementation on the newer device.

The implementation described in the following chapters demonstrates that the approach presented in this thesis enables the control algorithm to run efficiently on even relatively old and comparatively less capable hardware (see Table 6.2) than

current solutions offer. Many viable options are available for implementations of GPU-based algorithms, including pure CUDA code, OpenMP, or OpenACC. For reasons described in more detail later on, an implementation utilizing OpenACC was selected. The main advantages of this approach are great hardware portability and the ability to utilize existing C code to a large extent. In contrast to a potential pure CUDA low-level implementation, where the code is more tailor-made for a narrow spectrum of hardware, the OpenACC code is written in a generally broader manner, and its translation to CUDA is performed by the compiler. Although a CUDA-programmed application can be approximately 40 % faster with large computational grids compared to OpenACC, with smaller computational grids the difference becomes less significant. For the same tasks, OpenMP often performs 80 % or worse than optimized CUDA code. [28]

Since OpenACC's compilers need to translate the constructs into CUDA code first, while CUDA can be run directly and allows for more fine-tuning and optimization, the mean kernel running times are longer with OpenACC for many use cases. [34] However, especially for smaller data sizes, such as the NMPC of the IPMSM problem presented by this implementation, this difference becomes less pronounced. In some problems with smaller data sizes, OpenACC even showcased slightly better performance than a CUDA implementation of the same problem and significantly better performance than OpenMP. [11] Furthermore, the drawback of a potentially slower implementation using OpenACC was considered, for this application, to be balanced out by the portability of the program. It allows the application to run on different GPUs, not necessarily produced by Nvidia, or even multicore CPUs, while only requiring a relatively inexpensive recompilation of the original code. Although OpenMP portability might be slightly higher compared to OpenACC, the average performance of the latter is often significantly better. [61]

The aforementioned advantage of lower demands on the restructuring of the code in OpenACC stems from its directive-based region parallelization. [52] Sections of the code that have the potential to run in parallel can be marked by pragma directives, signaling to the compiler to launch multiple kernels. Such sections are typically loops with an iteration count known at compile time, whose values are independent of each other across the iterations. Considering the for loop from the code, which launches an a priori known number of search agents sequentially, this loop can be marked with an OpenACC directive to launch the agents in parallel to each other, as their values (e.g., optimization variable values or cost function value) are independent for every agent. The inner convergence loop of the BFGS algorithm, on the other hand, cannot be parallelized, as the values at each iteration are obtained from the results of the previous iteration. Data transfers between the host (CPU) and the device (GPU) are also performed by a special data directive initializing the

transfer. Although this task can be implicitly performed by the compiler during the earlier stages of development, it is often unable to successfully determine the correct number of variables that need to be transferred in either direction, which hinders overall performance. It is therefore advisable to optimize the data locality explicitly when targeting short computation times of the parallel algorithm. [50]

6.2.1 Setup

The workflow that utilizes a predictive controller with sequential launch of the search agents implemented in the C mex file and compiled using the supported gcc compiler was described in Chapter 6. However, the OpenACC compilers are not supported by MATLAB and a different approach to the parallelized version of the controller had to be taken. With C remaining the selected implementation language, the code had to be split into two parts. The first presents the core of the predictive controller as a callable C-function with optimization variables and state values as its arguments. The code leverages the OpenACC directives, performs the parallelized optimization, and is compiled as a shared library file using the openACC compilers for the selected device, in this case the Nvidia CUDA Compiler (`nvcc`). The second part of the code serves as an interface between the parallelized code of the controller core and Simulink. This interface is implemented using the rules for C-mex implementation, as discussed previously, and the controller code is called upon at each iteration. This simulation setup made it possible for the parallelized nonlinear model predictive controller to directly compute the voltages for the motor model in Simulink. The tools used during the implementation on Ubuntu 24.04 LTS were Nvidia CUDA compiler version `nvc 25.1-0 64-bit`, OpenACC version 3.3 API and MATLAB and Simulink R2024b. The GPU code performing the NMPC optimization was compiled using the following command:

```
nvc -fPIC -shared -acc -gpu=cc61 -cuda -Minfo=accel
-O3 -Munroll -Mvect=simd -fast -o lib_BFGS_openACC4.so
BFGS_openACC4.c -lcudart -Mipa=fast -Mconcur
-Mfprelaxed -use_fast_math
```

As a part of the preparations of the actual NMPC inputs, the interface also serves as the reference value integrator utilized for zero stead-state control error (see Fig. 8.1). The reference value from the previous simulation step is kept as a discrete state of the C-mex function, as is the integrator gain value K_{wi}.

6.2.2 Code Structure

The code is made up of two main nested loops, the outer loop launching the agents spread over the search domain and the inner loop performing the iterative optimization. The outer loop is marked with `#pragma acc parallel loop gang async` directive which signals to the compiler the beginning of the parallel region. Gang is the highest level of parallelism defined by openACC, typically representing a group of threads, and should be used for the outermost code loops. The clause `async` tells the compiler that the part of the code marked with this directive can be executed asynchronously. The subsequent code can be executed without waiting for the specified operation to complete, which potentially improves the overall performance by overlapping computations and or data transfers. [52] The effect of this clause is demonstrated in Section 9.4. For each agent, a set of private variables (unique to that search agent) is defined and computed before the search begins, as demonstrated in Listing 6.6. Those variables include the Hessian and gradient arrays and also other variables vital to the BFGS algorithm like step direction or current location (optimization variable value). The initial values of gradient (`gradVal()` function) and cost function value (`costFunctionValue()` function) are calculated. These functions are generated by the pruning program and are discussed in Section 5.2.

Listing 6.6: Parallel BFGS: Outer loop

```
65 #pragma acc parallel loop gang async
66     for (stPointCount = 0; stPointCount <
          NMPC_startingPointsMaxCount; stPointCount++) {
67         // Private variables per agent
68         float gdVarVal[8]={0}, gdGradVal[8]={0},
              gdGradValNext[8]={0}, H[64]={0}, Hk[64]={0};
69         float s[8]={0}, y[8]={0}, cfVal=0, c1=0.0001f,
              error=0.0f, var_tunable=8.0f;
70
71         // Initialize Hessian as identity matrix and
              initialize agents
72         H[0] = 1.0f;
73→79           ⋮
80         H[63] = 1.0f;
81         gdVarVal[0]=d_startingLocations[stPointCount*8+0];
82→87           ⋮
88         gdVarVal[7]=d_startingLocations[stPointCount*8+7];
89
90         // Compute initial gradient
91         gradVal(gdGradValNext, initialValues[IDPOS],
              initialValues[IQPOS], initialValues[WPOS],
92                 initialValues[WREFPOS], initialValues[4],
                      initialValues[5],
93                 gdVarVal[0], gdVarVal[1], gdVarVal[2],
                      gdVarVal[3], gdVarVal[4], gdVarVal[5],
                      gdVarVal[6], gdVarVal[7], var_tunable);
94
95         cfVal = costFunctionValue(initialValues[IDPOS],
              initialValues[IQPOS], initialValues[WPOS],
96                 initialValues[WREFPOS], initialValues[4],
                      initialValues[5],
97                 gdVarVal[0], gdVarVal[1], gdVarVal[2],
                      gdVarVal[3], gdVarVal[4], gdVarVal[5],
                      gdVarVal[6], gdVarVal[7], var_tunable);
```

The selection of starting locations using *Halton sequence* is discussed in more detail in Section 8.3. Calling the function `generateStartingLocations()` (see Listing 6.7 line 14), computes the starting location array `float* locations` by using the `halton()` function. First `numAgents`-1 agents, where `numAgents` is their total count, are mapped according to the prime numbers in the range of $[0, 1]$. `numDims` argument represents the size of the optimization variable vector. The starting locations are then scaled to the range of $[-1, 1]$ and for the final agent the vector $\mathbf{x} = [0, 0, \ldots, 0]^T$ is selected as its starting location. The array locations

are saved on the device under the name `d_startingLocations` which are then utilized in Listing 6.6.

Listing 6.7: Parallel BFGS: Generating the starting locations using the Halton sequence

```
1  float halton(int index, int base) {
2      float result = 0.0f;
3      float f = 1.0f / (float)base;
4      int i = index;
5
6      while (i > 0) {
7          result += (i % base) * f;
8          i /= base;
9          f /= (float)base;
10     }
11     return result;
12 }
13
14 void generateStartingLocations(float* locations, int
       numAgents, int numDims) {
15     float primes[10] = {2, 3, 5, 7, 11, 13, 17, 19, 23,
           29};
16     for (int i = 0; i < numAgents-1; i++) {
17         for (int d = 0; d < numDims; d++) {
18             float val = halton(i + 1, primes[d]);
19             // Map [0,1] to [-1,1]
20             val = 2.0f * val - 1.0f;
21             locations[i * numDims + d] = val;
22         }
23     }
24     for (int d = 0; d < numDims; d++) {
25         locations[(numAgents-1) * numDims + d] = 0.0f;
26     }
27 }
```

The search is then started inside the inner loop with a predetermined iteration count of two to reduce the potential branching of the algorithm. With a relatively large number of agents searching for the minimum of the function, it becomes sufficient to perform only a few optimization iterations, and branching of the code caused by checking the current magnitude of the error is infeasible. This loop is implemented as two consecutive blocks of code representing each iteration as demonstrated in Listing 6.8. Inside this convergence loop, the step direction is obtained by calling the `obtainS()` function on line 110. The BFGS step is performed according to Algorithm 4, except for the assessment of step length. Instead of the suggested *Wolfe conditions* check, a more simple approach (discussed in more detail

in Section 6.1) of *Armijo backtracking line-search* was implemented on lines 114-142. First, the vector of potential values for the length of the step α is created (line 114) together with an array of boolean values `lineSearchSuccess` which is used to indicate whether the Armijo condition has been met for each length of the step.

Listing 6.8: Parallel BFGS: Convergence Loop Part 1

```
101 // FIRST ITERATION -----------
102 gdGradVal[0]=gdGradValNext[0];
103→108     ⋮
109 gdGradVal[7]=gdGradValNext[7];
110 obtainS(s, H, gdGradVal);
111
112 int lineSearchIt = 0;
113
114 float alphaT[15] = {1.0f, 0.75f, 0.5f, 0.3f, 0.2f, 0.1f,
       0.05f, 0.025f, 0.0125f, 0.00625f, 0.003125f, 0.0015625f,
        0.00078125f, 0.000390625f, 0.00001953125f};
115 bool lineSearchSuccess[15];
116 float pktGf = obtainpktGf(s, gdGradVal);
117 float f_a[NMPC_lineSearchMaxIterations] = {0}, gdVarValA
       [8];
118 // Armijo Backtracking Line Search
119 #pragma acc loop
120 for (lineSearchIt=0; lineSearchIt<
       NMPC_lineSearchMaxIterations; ++lineSearchIt) {
121     gdVarValA[0] = gdVarVal[0] + alphaT[lineSearchIt]*s[0];
122→127         ⋮
128     gdVarValA[7] = gdVarVal[7] + alphaT[lineSearchIt]*s[7];
129     f_a[lineSearchIt] = costFunctionValue(initialValues[
           IDPOS], initialValues[IQPOS], initialValues[WPOS],
           initialValues[WREFPOS], initialValues[4],
           initialValues[5], gdVarValA[0], gdVarValA[1],
           gdVarValA[2], gdVarValA[3], gdVarValA[4], gdVarValA
           [5], gdVarValA[6], gdVarValA[7], var_tunable);
130     lineSearchSuccess[lineSearchIt]=f_a[lineSearchIt]<=
           cfVal + c1*alphaT[lineSearchIt]*pktGf;
131 }
132 float alpha = alphaT[14];
133 cfVal = f_a[14];
134 #pragma acc loop
135 for (i = 13; i >= 0; --i) {
136     if (lineSearchSuccess[i]) {
137         alpha = alphaT[i];
138         cfVal = f_a[i];
139 }}
```

The expression $\boldsymbol{p}_k^T \nabla f(\boldsymbol{x}_k)$ is stored in `pktGf` variable. Inside the line-search loop (from line 121 of Listing 6.8) the new optimization variable `gdVarValA` value for the respective step length is determined and the cost function is evaluated at this point. The result of the *Armijo condition* comparison is stored in the `lineSearchSuccess` array. The largest step length that met the condition is then selected (lines 133-142).

The rest of the BFGS iteration tasks are then performed in Listing 6.9. The calculation of the gradients (line 167), the Hessian update (lines 180-191) and the evaluation of the cost functions are performed by the functions created with the pruned NMPC application. As the dimension of the problem is quite small with most vectors, such as those representing the step direction or the gradient difference consisting of eight elements for the control horizon of length $N = 4$, those calculations are often performed sequentially without the introduction of other inner loops. The overhead necessary for launching the parallelized or even sequential loops inside the convergence loops was found to be more costly than the sequential performance of the task. As this part of the code implements the first iteration of the optimization algorithm, the Hessian matrix inversion approximation $\boldsymbol{H}_0$, initially set as a unit matrix, is multiplied by a scalar coefficient based on the obtained values of y and s as described by (4.1.2).

The second iteration of the algorithm is performed in a similar manner. The key difference is that the gradients, the Hessian inverse approximations, and other supporting variables are no longer computed. After the convergence loop finishes, the resulting location is stored in an array `gdVarValRes`, as is the evaluation of the cost function `cfRes`, based on which the best search agent is later identified. As only two of the coordinates of the location reached, representing voltage increments Δu_d and Δu_q, are used as actual control inputs, only those must be stored for further use. As can be seen in Listing 6.10, the search for the best agent is implemented sequentially using the directive `#pragma acc serial`. During testing, this approach found the best agent faster than other parallel versions due to the necessary overhead introduced. After identifying the index `minId` of the best agent, corresponding to the minimal value of the cost function `minVal`, is identified, its achieved results are retrieved from the array `gdVarValRes`.

Listing 6.9: Parallel BFGS: Convergence Loop Part 2

```
158 gdVarVal[0] += alpha * s[0];
159→164     ⋮
165 gdVarVal[7] += alpha * s[7];
166 gradVal(gdGradValNext, initialValues[IDPOS], initialValues[
       IQPOS], initialValues[WPOS], initialValues[WREFPOS],
       initialValues[4], initialValues[5], gdVarVal[0],
       gdVarVal[1], gdVarVal[2], gdVarVal[3], gdVarVal[4],
       gdVarVal[5], gdVarVal[6], gdVarVal[7], var_tunable);
167 y[0] = -gdGradVal[0] + gdGradValNext[0];
168→173     ⋮
174 y[7] = -gdGradVal[7] + gdGradValNext[7];
175 // Update Hessian
176 // Obtain H0 = y' * s / y' * y for subsequent H(1) update
177 float yTs = y[0]*s[0]+y[1]*s[1]+...+y[8]*s[8];
178 float yTy = y[0]*y[0]+y[1]*y[1]+...+y[8]*y[8];
179 float ytsyTy = yTs/yTy;
180 H[0] = ytsyTy;
181→186     ⋮
187 H[63] = ytsyTy;
188
189 obtainHk(Hk, H, y, s);
190 #pragma acc loop seq
191 for (i = 0; i < 64; ++i) {
192     H[i] = Hk[i];
193 }
```

Listing 6.10: Parallel BFGS: Determining the best agent

```
289 #pragma acc serial
290 {
291     float minVal = 1e8;
292     int minId = 0;
293     #pragma acc loop
294     for (i = 0; i < NMPC_startingPointsMaxCount; ++i) {
295         if (cfRes[i] < minVal) {
296             minVal = cfRes[i];
297     }}
298     #pragma acc loop seq
299     for (i = 0; i < NMPC_startingPointsMaxCount; ++i) {
300         if (cfRes[i] == minVal) {
301             minId = i;
302     }}
303     bestVarVal[0] = gdVarValRes[minId*2];
304     bestVarVal[1] = gdVarValRes[minId*2+1];
305 }
```

As was observed during the development of the algorithm, it is often ineffective to parallelize smaller code sections rather than simply computing them in sequence. This is mainly caused by the overhead that is necessarily introduced when parallelizing a loop that does not process enough data for the parallelization to be feasible. In the code structure, this aspect is predominantly represented in three cases. The first is the computations of the BFGS vectors, such as gradient values or gradient differences y. Ideally, if the process was computed in parallel for each element of the vector, the process would complete faster; however, in practice, the number of necessary computations performed is small enough, causing the sequential approach to yield better results. The second case where the sequential run of the code has proven to be faster is the search for the best agent. This is performed by identifying the smallest value in the array that stores the evaluations of the cost function of each agent. Although there are OpenACC constructs specifically designed to handle such scenarios, where each parallel thread should communicate with others to determine the smallest value of an array, the introduced overhead was not justifiable with the number of agents used. However, this would differ with a higher agent count or other hardware utilized for parallel computations. Finally, the code that performs the inexact line search was also implemented using a sequential loop. Parallel approaches were also tested but proved to be inefficient due to a significant slow-down of the overall computations.

7 Benchmarking and Validation

The efficacy of the optimization algorithm will be demonstrated in this chapter through two distinct sets of problems. Its ability to successfully identify global optima of various functions, using its parallel properties, will be tested on multiple constrained and unconstrained test functions. Each features a uniquely designed objective function with a shape that generally poses challenges for optimization. The pruning aspects of the algorithm, which reduce the complexity of the underlying objective function, will be showcased using a simplified NMPC of a PMSM problem, where the optimization accuracy of differently pruned functions will be compared. The algorithm's ability to integrate these two aspects, reducing the complexity of the objective function while preserving its features and behavior, and simultaneously optimizing it rapidly in parallel, is discussed in Section 9.1.

7.1 Optimization Test Functions

A test function is artificially designed to evaluate the ability of a selected optimization algorithm to successfully achieve its optimization goal. [24] The presented functions were chosen due to the challenges they pose to the algorithm, typically involving problems with multiple local minima, often located close to each other, some with small or zero gradients, or a global minimum near a constraint. The latter scenario can present a significant obstacle for an optimization algorithm employing logarithmic barrier functions to represent constraints, as their value increases even before the optimization variable reaches the specified constraint (see Fig. 4.1). Given the gradient-based nature of the optimization algorithm, only test functions that are continuously differentiable were selected. All test functions were optimized with the following algorithm settings, unless otherwise noted. A total of ten search agents were initialized using the *Halton sequence* described in Section 8.3. Each was limited to a maximum of 50 iterations and 15 line search iterations, utilizing the *Armijo backtracking line search* with an initial step length of 1, which was multiplied by a factor of 0.5 after each pass. The terminating error threshold was set to $\varepsilon = 0.01$.

The first set of test functions presents unconstrained optimization problems of varying natures. The Rastrigin test function, visualized in Fig. 7.1, is a nonconvex problem with a relatively large domain and multiple local minima. [64] A two-dimensional version of this problem was employed. As the global minimum of the Rastrigin function, described by (7.1.1), is located at $x_1, x_2 = 0$, no search agent was assigned a starting location there to rigorously assess the performance of the algorithm. A visualization of the movements of the selected agents can be seen in Fig. 7.2, where each agent is represented by a distinct color and a red cross marking

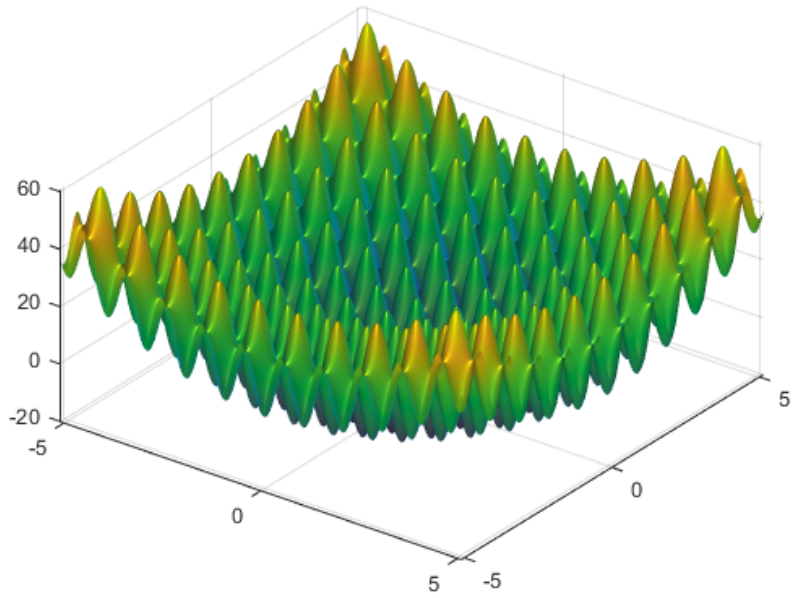

Fig. 7.1: Rastrigin test function

its final location. Its movement was terminated either by successfully identifying a local minimum–when the computed gradient error of the agent fell below the ε threshold–or by reaching the maximum number of iterations allowed for that agent.

$$f(\boldsymbol{x}) = 10n + \sum_{i=1}^{n}[x_i^2 - 10\cos(2\pi x_i)], \quad \boldsymbol{x} \in [-5.12, 5.12]^{n\times 1} \tag{7.1.1}$$

From Fig. 7.2, it is evident that although all the agents displayed located a local minimum of the Rastrigin test function, not all identified the global optimum. Although the distribution of agents across the search domain by the *Halton sequence* is deterministic, it is entirely possible that, had the agents been assigned different starting locations, the global optimum might not have been found. In such a case, the total number of initialized agents would need to be increased. The location of the true global minimum and the one identified can be compared in Table 7.1. It is clear that the true minimum was found within the set accuracy. Reducing the error threshold epsilon would produce more precise results.

A test function by Fletcher and Powell in the form given by (7.1.2) and (7.1.3), was originally proposed as a complicated function with many local minima. [16] As per the original article, the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ multiplying the trigonometric functions consisted of randomly selected integers within the range of $\boldsymbol{A}, \boldsymbol{B} \in \{-100, 100\}^{n\times n}$, as were the arguments of these functions, $\boldsymbol{\alpha}$, which are selected in the range of $\boldsymbol{\alpha} \in [-\pi, \pi]^{n\times 1}$. The search domain of this function spans from

Table 7.1: Comparison of true and achieved global optima of the test functions

Test Function	True Global Minimum $\boldsymbol{x}_0$	Found Minimum $\boldsymbol{x}_f$
Rastrigin	$(0, 0)$	$(1.951 \cdot 10^{-5}, -4.407 \cdot 10^{-5})$
Fletcher-Powell	$(-2.6857, 0.85096)$	$(-2.2740, -0.1258)$
Goldstein-Price	$(0, -1)$	$(-6.415 \cdot 10^{-5}, -1)$
Bird	$(-3.1302, -1.5821)$	$(-3.13, -1.582)$
Rosenbrock restricted 1	$(1, 1)$	$(0.9978, 0.9955)$
Rosenbrock restricted 2	$(0, 0)$	$(-0.002479, -0.007873)$

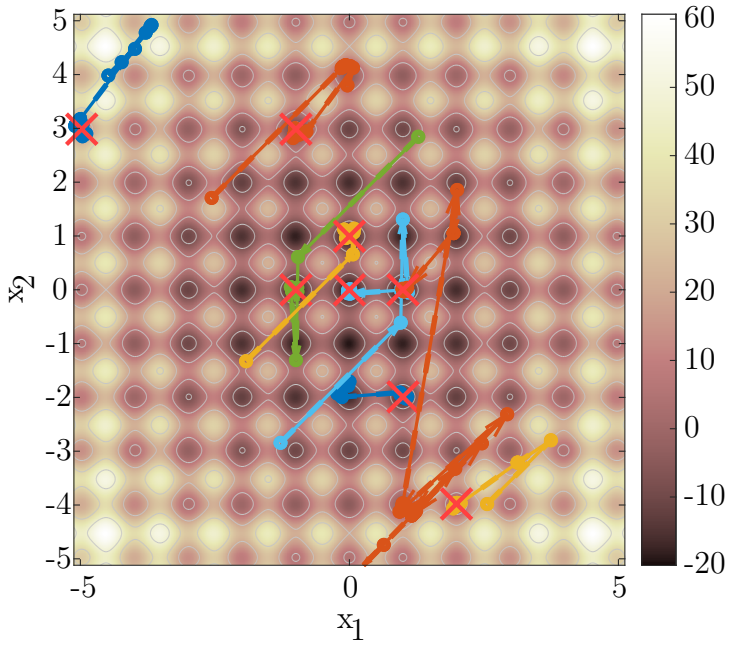

Fig. 7.2: Agents behavior on Rastrigin test function

$-\pi$ to π, with up to 2^n local minima located within this range. Upon inspection of the equation, it is evident that the global optimum of this function lies at $x_j = \alpha_j,\ \forall j \in \{1, 2, \ldots, n\}$. [58]

$$f(\boldsymbol{x}) = \sum_{i=1}^{n} (\boldsymbol{A}_i - \boldsymbol{B}_i(\boldsymbol{x}))^2 \tag{7.1.2}$$

$$\boldsymbol{A}_i = \sum_{j=1}^{n} (a_{ij} \sin \alpha_j + b_{ij} \cos \alpha_j) \tag{7.1.3a}$$

$$\boldsymbol{B}_i(\boldsymbol{x}) = \sum_{j=1}^{n} (a_{ij} \sin x_j + b_{ij} \cos x_j) \tag{7.1.3b}$$

During testing, a two-dimensional version of this function was used to highlight potential drawbacks of the parallelized optimization algorithm implemented using single-precision floating-point variables. The behavior of the search agents on this function is visualized in Fig. 7.3. Although the ideal analytical minimum of this function should lie at the coordinates given by the vector alpha, it was demonstrated that the optimization algorithm identifies a point that yields a lower value of the test function. The function by Fletcher and Powell, evaluated at $\boldsymbol{x}_0 = \boldsymbol{\alpha} = (-2.6857, 0.85096)$, returns $f(\boldsymbol{x_0}) = 1.591 \cdot 10^{-4}$, while at the found $\boldsymbol{x}_f = (-2.2740, -0.1258)$ it gives a value of $f(\boldsymbol{x_f}) = 7.941 \cdot 10^{-5}$. The discrepancy between these two values arises from precision errors in floating-point variables and the evaluation of trigonometric functions. Although the true optimum of the original function was not located in this manner, detecting another point with a very low value proves to be a satisfactory outcome in this scenario. As noted in Section 1.3, the motor predictive controller is based in part on the suboptimality principle, where a slightly worse solution to the underlying function may be more suitable than a slightly better one if it is reached in less time. Thus, minor decimal errors do not affect the overall performance of the algorithm.

The third unconstrained test scenario was developed using the Goldstein-Price test function, which, unlike the previous two, is not based on trigonometric functions. It is an 8^{th}-degree polynomial of two variables with local minima at $\boldsymbol{x}_1 = (1.2, 0.8)$, $\boldsymbol{x}_2 = (1.8, 0.2)$, and $\boldsymbol{x}_3 = (-0.6, -0.4)$, and a global minimum at $\boldsymbol{x}_0 = (0, -1)$. [21] The function is defined as (7.1.4) and presents a problem characterized by a large plateau with small gradients. However, as shown in Fig. 7.4, most agents successfully converged to the global optimum. The local minima are marked with blue crosses in this figure, the one at $\boldsymbol{x}_3 = (-0.6, -0.4)$, causing slight confusion for one of the agents. The true value of the global optimum and the one identified can be compared in Table 7.1.

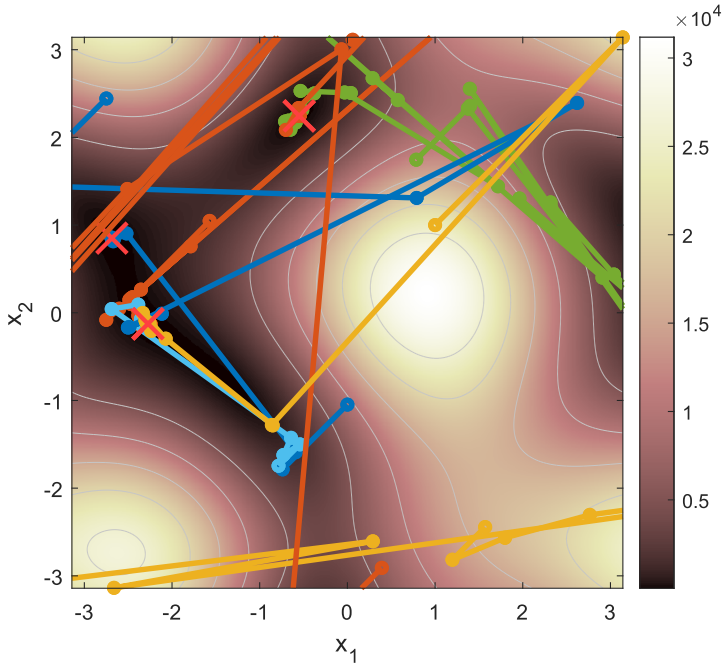

Fig. 7.3: Agents behavior on Fletcher-Powell test function

$$f(x_1, x_2) = \Big[1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2\Big] \cdots \\ \cdots \Big[30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)\Big] \tag{7.1.4}$$

For the constrained optimization problem, the Bird function constrained to a disk was chosen. This function was first defined by Mishra [41] as (7.1.5), features multiple local minima and plateaus with minor gradient values. Fig. 7.5 provides a spatial visualization of the Bird function, whose global minimum is located at approximately $\boldsymbol{x}_0 \approx (-3.1302, -1.5821)$. Its search domain is restricted to $x_1, x_2 \in [-10, 0]$ for both variables, and in this implementation, these are further constrained to a disk defined by: $(x_1 + 5)^2 + (x_2 + 5)^2 = 25$.

$$f(x_1, x_2) = \sin(x_2) \exp\Big[(1 - \cos x_1)^2\Big] + \cos(x_1) \exp\Big[(1 - \sin x_2)^2\Big] \tag{7.1.5}$$

Fig. 7.6 visualizes three distinct cases of agent behavior based on their respective starting locations. Those closer to the local minimum at approximately $\boldsymbol{x}_1 \approx (-3.2, -7.8)$ converged there, while others successfully located the global minimum. The subsequent selection of the best agent, based on the evaluation of the function in the final locations of the agents, determines the superior result. A special case of an agent's movement is evident in the top left corner of this figure, where

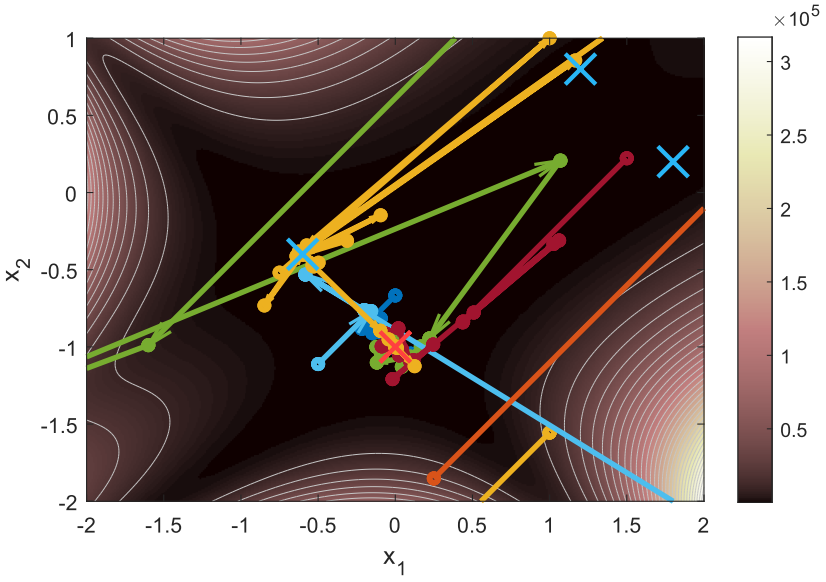

Fig. 7.4: Agents behavior on the Goldstein-Price test function

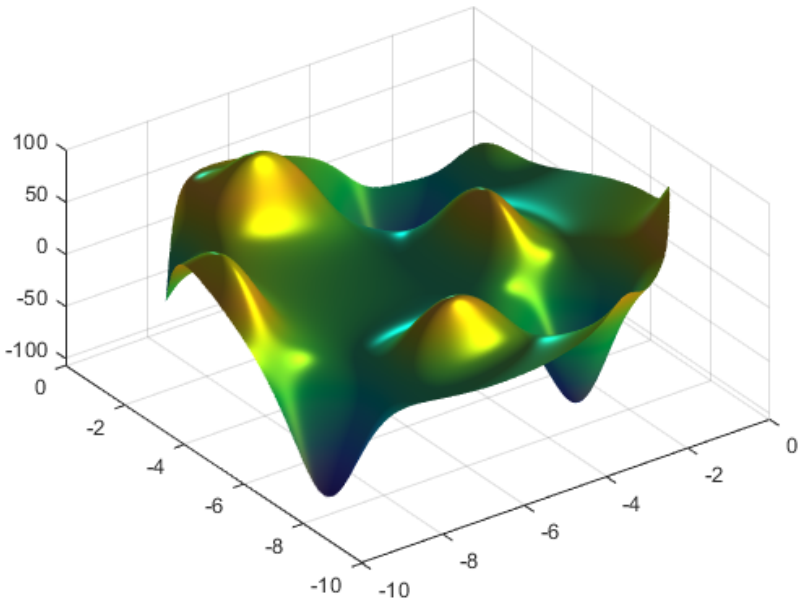

Fig. 7.5: Bird test function

an agent was initialized within the allowed region of $[-10, 0]^{2\times1}$ but outside the constraining circle. After its location was checked during the first iteration of its optimization run, further progress was terminated.

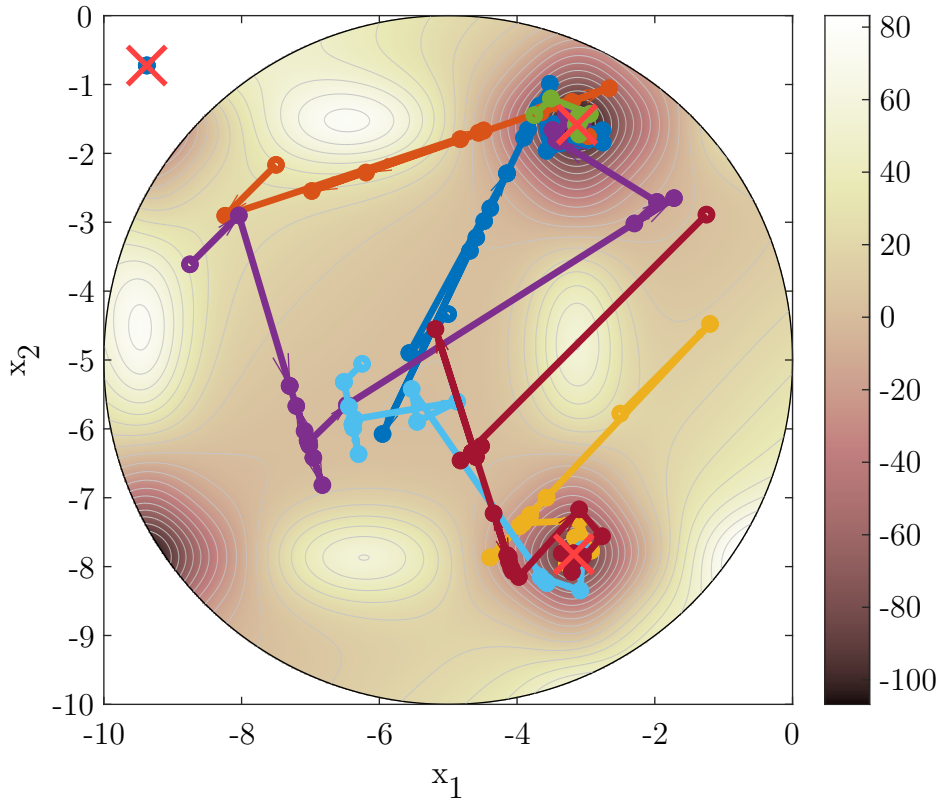

Fig. 7.6: Agents behavior on the Bird test function

The optimization algorithm was also evaluated using Rosenbrock's function [57], defined by (7.1.6), which has a global minimum located within a narrow valley shaped by parabola. The original search domain of $x_1, x_2 \in [-1.5, 1.5]$ was constrained by (7.1.7), representing a circle, in the first of two constraint scenarios.

$$f(x_1, x_2) = (1 - x_1)^2 + 100(x_1^2 - x_2)^2 \tag{7.1.6}$$

$$x_1^2 + x_2^2 - 2 \leq 0 \tag{7.1.7}$$

A pattern can be observed in Fig. 7.7 depicting the behavior of agents, who generally converge to the narrow valley and then locate its deepest point, situated at the edge of the restricted region. The true minimum within this valley lies at the coordinates $\boldsymbol{x}_0 = (1, 1)$, which according to Table 7.1 was not precisely found. This slight deviation is attributed to logarithmic barrier functions, which

modify the behavior of the function near or at the constraints. In this case, a coefficient of $c = 0.001$ multiplying the logarithmic barrier was applied. Further reducing this coefficient could minimize the difference between the true and found optimum of the function but might also introduce undesirable numerical instability in the optimization process. A solution achieved with this level of accuracy remains acceptable in the context of the algorithm's use case, similar to the minor floating-point operation errors observed with the Fletcher-Powell test function.

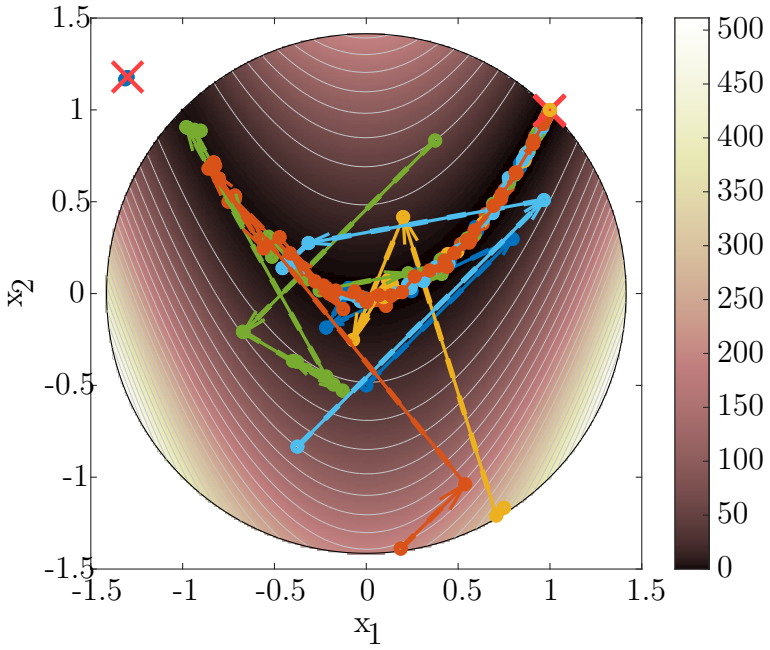

Fig. 7.7: Agents behavior on the Rosenbrock's test function, first constraint scenario

The second constraint scenario for the Rosenbrock function was implemented using a set of two inequalities (7.1.8). A pattern similar to the previous scenario is visible in Fig. 7.8 Most agents converge to the valley and identify its minimum within the restricted region. Some agents terminate after initializing outside this region. One agent began its optimization run within the permitted area but took an optimization step that resulted in an out-of-bounds position, leading to the termination of its subsequent iterations.

$$(x_1 - 1)^3 - x_2 + 1 \leq 0 \tag{7.1.8a}$$

$$x_1 + x_2 - 2 \leq 0 \tag{7.1.8b}$$

To test the capability of the algorithm to handle optimization problems of a higher dimension, a modification was made to the original Rastrigin function (7.1.1).

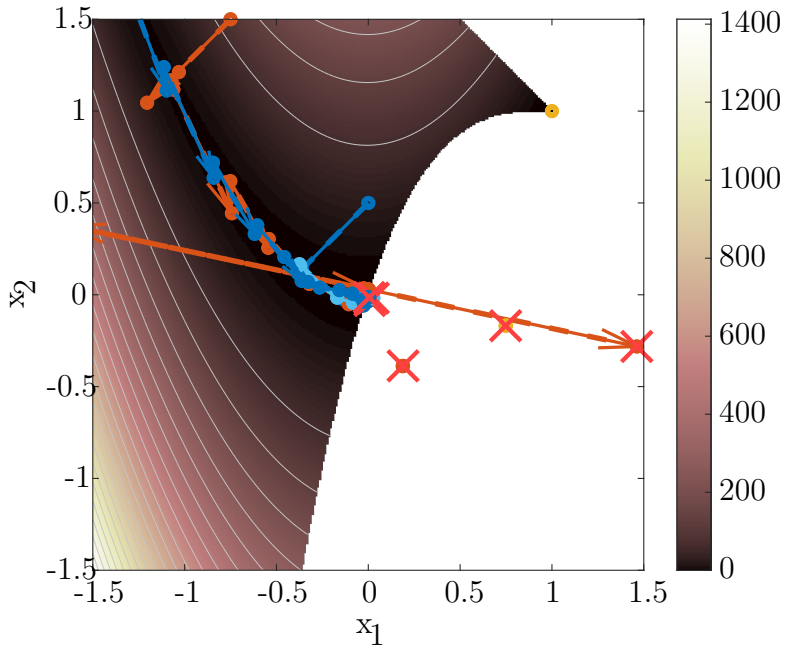

Fig. 7.8: Agents behavior on the Rosenbrock's test function, second constraint scenario

The number of dimensions was increased to $n = 30$ and subsequently the function was optimized with 150 parallel agents. The limit of optimization and line-search iterations was kept at 50 and 15 respectively. The error threshold remained $\varepsilon = 0.01$. The parallelized optimization algorithm managed to find the global optimum of the function with a precision higher than $1 \cdot 10^{-7}$, as can be seen in Table 7.2 featuring the location achieved by the best agent.

The problem of $n = 30$ dimensions already presents a significant computational burden as the size of the Hessian inverse approximation grows to 30×30, presenting a total of 900 elements that need to be calculated. Although this approach is well suited for the implementation of NMPC with a shorter prediction horizon, like the one of IPMSM presented in this thesis, larger optimization problems could pose a problem. For such cases, the original BFGS optimization method can be replaced with a suitable alternative, as is the limited-memory BFGS (L-BFGS) designed to handle such problems. [36]

Table 7.2: Rastrigin function in 30 dimensions

Dimension	Result	Dimension	Result
1	$3.746 \cdot 10^{-8}$	16	$3.746 \cdot 10^{-8}$
2	$2.090 \cdot 10^{-8}$	17	$2.096 \cdot 10^{-8}$
3	$2.043 \cdot 10^{-8}$	18	$1.250 \cdot 10^{-8}$
4	$1.250 \cdot 10^{-8}$	19	$1.250 \cdot 10^{-8}$
5	$1.300 \cdot 10^{-8}$	20	$1.252 \cdot 10^{-8}$
6	$1.252 \cdot 10^{-8}$	21	$4.607 \cdot 10^{-8}$
7	$4.603 \cdot 10^{-8}$	22	$4.607 \cdot 10^{-8}$
8	$-9.172 \cdot 10^{-9}$	23	$-9.172 \cdot 10^{-9}$
9	$-8.696 \cdot 10^{-9}$	24	$-8.635 \cdot 10^{-9}$
10	$-8.685 \cdot 10^{-9}$	25	$-1.353 \cdot 10^{-8}$
11	$-1.658 \cdot 10^{-8}$	26	$-1.669 \cdot 10^{-8}$
12	$-1.606 \cdot 10^{-8}$	27	$-1.607 \cdot 10^{-8}$
13	$-1.606 \cdot 10^{-8}$	28	$-1.603 \cdot 10^{-8}$
14	$-1.705 \cdot 10^{-8}$	29	$-1.705 \cdot 10^{-8}$
15	$-1.705 \cdot 10^{-8}$	30	$-1.705 \cdot 10^{-8}$

7.2 Accuracy of the Pruned Function

In this chapter, the algorithm's approach to reducing the complexity of the optimization problem is tested and validated. The NMPC of the IPMSM problem, with a prediction horizon of three steps and constrained by the total voltage and current equations, is considered. The shorter prediction horizon was selected because of the vast complexity of the original cost function for $N = 4$, which consists of 844,302 terms just for its standard part. The observations presented in this chapter were partially published in [60].

The importance of cost function pruning is demonstrated using the aforementioned problem, with the entire function evaluated. Although this approach is not necessary during the standard operation of the pruning utility described in Section 5.1, for this purpose, the complete cost function, comprising 2,399 terms, was obtained. The complexity of the resulting problem is illustrated in Fig. 7.9, where the absolute values of the numeric coefficients M of each term (4.3.4) were divided into logarithmic bins along the x-axis. The height of each bin represents the maximum cumulative impact of the respective set of terms on the overall value of the cost function. Here, the most extreme case is considered, where the normalized variables in each term take an absolute value of 1. The number of terms in each bin is indicated by its color saturation. The y-axis of the plot is also logarithmic. The

overlaid yellow line was obtained by cumulatively summing the terms of the cost function from the smallest to the largest. The total value of the cost function, if each normalized variable were to take an absolute value of 1, is thus the value reached by this line at its right end. A potential relative pruning threshold of $\delta = 10^4$ is marked with a dashed line.

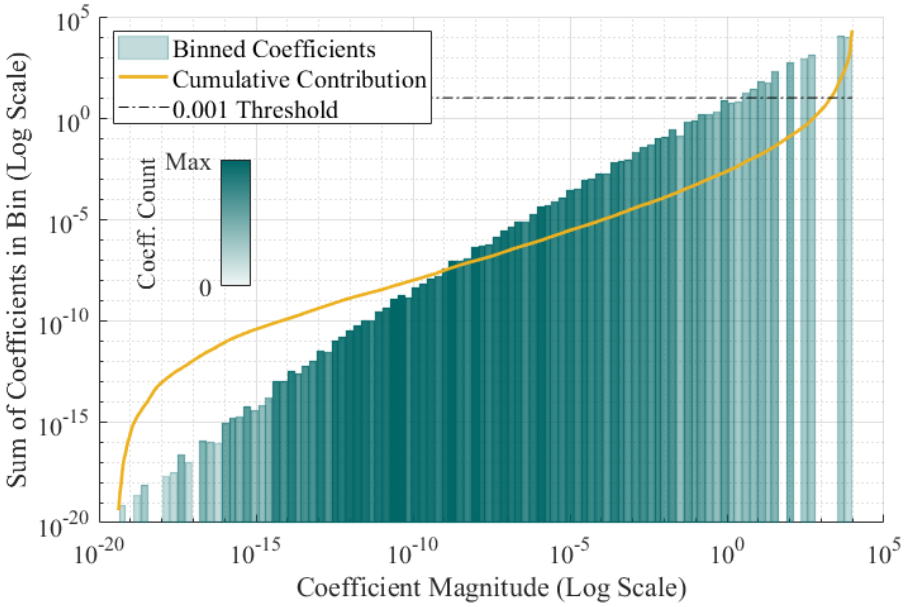

Fig. 7.9: Impact of the terms on the cost function value for $N = 3$

If the original function were not rewritten in the form of (4.3.3), a comparison between the pruned and original functions can be made based on the number of multiplications and additions required for evaluation. The resulting comparison is presented in Table 7.3 for the prediction horizon of length four, where the rewritten original function (4.3.3) and the pruned function are considered to be prepared without the precomputation of recurring terms described in Section 5.1. This approach was adopted to list the total required number of operations for each function in a comparative manner, as the recurring term predefinition could potentially also be applied to the original function. It is evident that even if the original function without additional rewriting is considered, a significant reduction of the number of necessary operations occurs after pruning.

The pruning process must simultaneously achieve two objectives: significantly reducing the complexity of the pruned function while preserving its accuracy and precision. Selecting the appropriate pruning threshold therefore involves a trade-off between these two goals. To quantitatively analyze this trade-off, a set of two indicators was developed.

The Pareto-Efficiency-Inspired Metric (PEM) is based on the *Pareto efficiency*

Table 7.3: Pruned functions comparison for $N = 4$

Function	Original	Original Rewritten	Pruned
Terms Standard	-	844,302	938
Terms Constraints	-	71,981,561	341
Additions Standard	1,264	79,290	420
Multiplications Standard	1,990	172,928	734

principle used in multi-objective optimization. [39] Reallocating resources to improve one objective is considered *Pareto efficient* if it does not impair other objectives. PEM is defined as (7.2.1) by multiplying two terms. The first term represents the balance between the complexity and precision of the function. Both objectives, complexity and error, should ideally be zero, which is symbolically indicated by subtracting zero in both parentheses of the denominator of the first term. N_{pruned} and N_{original} denote the total number of terms in the pruned and original functions, respectively, and their ratio reflects the complexity of the resulting function. MAE represents the *mean absolute error* of the pruned function and C_{original} is the mean of the absolute values of the output of the original cost function. The ratio $\frac{\text{MAE}}{C_{\text{original}}}$ indicates the average error of the current solution. The crowding distance (CD), present in the second term of the formula and defined by (7.2.2), assesses the density of solutions in the trade-off region. This term facilitates the comparison of different tested pruning threshold values and discourages their clustering at extremes. Consequently, solutions with very low complexity but higher mean error, or those with unjustifiably high complexity and very low error, are not favored by the Pareto efficiency-inspired metric.

$$\text{PEM} = \frac{1}{\sqrt{\left(\frac{N_{\text{pruned}}}{N_{\text{original}}} - 0\right)^2 + \left(\frac{\text{MAE}}{|\overline{C_{\text{original}}|}} - 0\right)^2}} \cdot \frac{1}{1 + CD} \tag{7.2.1}$$

$$\text{CD} = \left|\frac{N_{\text{pruned},i+1} - N_{\text{pruned},i-1}}{N_{\text{original}}}\right| + \left|\frac{\text{MAE}_{i+1} - \text{MAE}_{i-1}}{\overline{|C_{\text{original}}|}}\right| \tag{7.2.2}$$

The Relative Efficiency Metric (REM), similar to PEM, is used to balance the trade-off between complexity and accuracy. REM is defined as (7.2.3), where the numerator represents the reduction in computational complexity, and the denominator reflects the reduction in error of the current solution. This is measured by normalizing the mean absolute error of the solution relative to other tested pruning thresholds. In addition, a regularization coefficient ε is introduced to avoid singularities. It is recommended to set this coefficient to a small magnitude, such as $\varepsilon = 0.01$.

$$\mathrm{REM} = \frac{\frac{N_{\mathrm{original}} - N_{\mathrm{pruned}}}{N_{\mathrm{original}}}}{\frac{\mathrm{MAE} - \mathrm{MAE}_{\mathrm{min}}}{\mathrm{MAE}_{\mathrm{max}} - \mathrm{MAE}_{\mathrm{min}}} + \varepsilon} \tag{7.2.3}$$

Both indicators quantify the trade-off in slightly different ways, as shown in Table 7.4. REM, with its emphasis on error, is particularly well suited for performance-critical applications where complexity reduction is a secondary (but not insignificant) objective. In contrast, PEM balances the two objectives more generally, avoiding favoritism toward extreme solutions. Table 7.4 provides a statistical comparison of different pruning scenarios for the scenario of $N = 3$.

The pruning thresholds were set relative to values consistent with all threshold types $\boldsymbol{\delta} = \{\delta_S, \delta_{SC}, \delta_{CF}, \delta_{\nabla}, \delta_{\nabla C}, \delta_{CFC}\}$ that were described in Section 5.1. The functions evaluated a total of $n = 1000$ different input sequences. To compare the different pruning threshold values, additional metrics were used in addition to PEM and REM. The mean absolute error (MAE) measures the average magnitude of the error of the current solution when compared to the original function, the root mean squared error (RMSE) penalizes larger magnitude errors more heavily, and the mean relative error (MRE) contextualizes errors relative to the magnitude of the original values. The average bias of the pruned solution is represented by the mean difference $\overline{d}$ from the output of the original function, and the standard deviation of these differences is marked with s_d. An advantage of both PEM and REM is that they do not require assessing the performance of the original, often overly complex, cost function but instead compare available solutions among themselves.

Table 7.4: Cost function pruning threshold values comparison for $N = 3$

$\boldsymbol{\delta}$	100	1000	10^4	10^5	10^6
$\overline{d}$	111.6655	3.3757	-0.0556	0.0209	0.0065
s_d	50.7002	5.0960	0.4595	0.0448	0.0160
MAE	111.6655	4.7109	0.3397	0.0362	0.0117
RMSE	122.6259	6.1105	0.4626	0.0494	0.0173
MRE	18.3812	0.6666	0.0486	0.0053	0.0011
N_{pruned}	119	181	246	337	454
REM	0.94	17.75	69.37	**84.11**	81.08
PEM	7.50	**11.41**	9.12	6.55	5.04

A visual representation of the difference caused by the change in the pruning threshold by an order of magnitude can be examined in the Bland-Altman plot Fig. 7.10 of the two data sets. The figure plots the difference of the outputs of the original and pruned functions for each data point along the y-axis. The horizontal axis represents the average value of the evaluated cost function. The mean values

for each selection of the pruning threshold are plotted with solid blue and red lines for $\delta = 10^3$ and $\delta = 10^4$, respectively. Expectedly, setting the pruning threshold higher means a significant reduction in the deviations from the mean. Furthermore, the mean values of the differences for $\delta = 10^4$ were aligned with zero better than for $\delta = 10^3$.

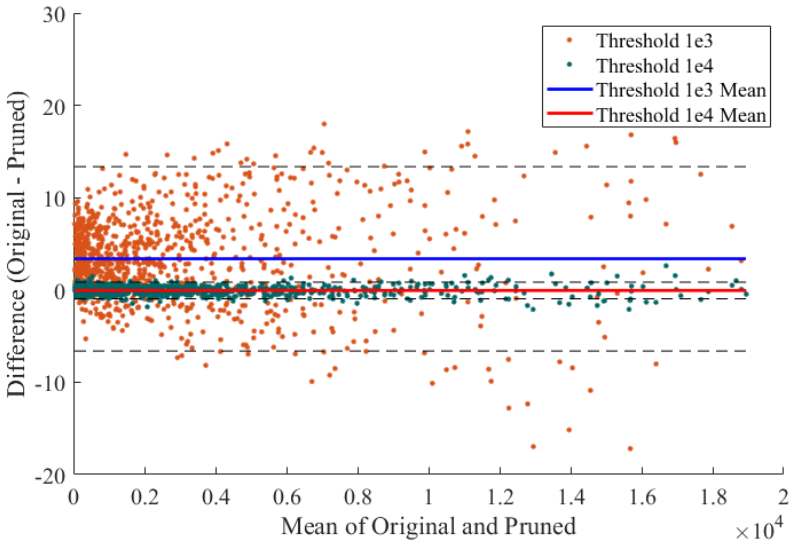

Fig. 7.10: Bland-Altman Comparison of $\delta = 10^3$ and $\delta = 10^4$

As demonstrated by Table 7.4, the pruning algorithm can significantly reduce the complexity of the cost function, represented by the remaining terms N_{pruned} compared to the original rewritten function with $N_{\text{original}} = 2399$, while maintaining the precision quantified by the mean relative error (MRE). Based on REM and PEM metrics, the optimal pruning threshold selection for the general scenario of NMPC of IPMSM is approximately in the range of $\delta = [10^3, 10^4]$. A significant reduction in the necessary multiplication and addition operations can be observed in the pruned function even compared to the original function, which was not rewritten (see Table 7.3).

8 Algorithms Parameters and Details

The number of parameters of the algorithm that need to be determined increased significantly with the introduction of the cost function preparation (discussed in Chapter 5) compared to a standard MPC approach. Another set of tunable parameters is contributed by the optimization problem, like the iteration count or line search settings. The resulting parameters of the algorithm can therefore be divided into three categories: those relating to the cost function based on the state-space model, those stemming from the subsequent pruning, and those adhering to the optimization of the cost function. The first two need to be decided during the offline preparation process, while the latter can be tuned after the cost function is put together. In this chapter, the three categories of the parameters of the algorithm are discussed with a focus on the parallelized version of the algorithm with field-weakening capabilities. The IPMSM model described in Section 3.1 was used with a sampling period of $t_s = 200\,\mu$s. The simulation results, the quality of the control, and other aspects of each implementation are discussed in Chapter 9.

8.1 Cost Function Parameters

The list of settings and other adjustments utilized in the creation of the cost function is common to all MPC approaches and, in this case, to those based on the state-space model of the controlled process. After examining the form of the cost function used in this approach (4.2.1) and (4.2.2), a set of required parameters can be identified. Three matrices — $\boldsymbol{P}$, $\boldsymbol{Q}$, and $\boldsymbol{R}$ — of appropriate dimensions, weighting the states and inputs over the prediction horizon are constructed to penalize the values of the corresponding states and inputs. They provide a structure for the predictive controller, reflecting the respective priorities each state or input should receive during optimization. A higher weight coefficient results in the respective state or input value being penalized more heavily during the optimization process. For the final implementation of the algorithm, the weighting matrices $\boldsymbol{P}$, $\boldsymbol{Q}$, and $\boldsymbol{R}$ were chosen as outlined by (8.1.1).

To balance computational complexity (and thus time efficiency) with the capabilities and control accuracy of the controller, a prediction horizon of a length $N = 4$ was selected.

$$\boldsymbol{P} = \begin{bmatrix} p_{i_d} & 0 & 0 & 0 & 0 & 0 \\ 0 & p_{i_q} & 0 & 0 & 0 & 0 \\ 0 & 0 & p_\omega & -p_\omega & 0 & 0 \\ 0 & 0 & -p_\omega & p_\omega & 0 & 0 \\ 0 & 0 & 0 & 0 & p_{u_d} & 0 \\ 0 & 0 & 0 & 0 & 0 & p_{u_q} \end{bmatrix} = \begin{bmatrix} 5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3500 & -3500 & 0 & 0 \\ 0 & 0 & -3500 & 3500 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.001 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.1.1a}$$

$$\boldsymbol{Q} = \begin{bmatrix} q_{i_d} & 0 & 0 & 0 & 0 & 0 \\ 0 & q_{i_q} & 0 & 0 & 0 & 0 \\ 0 & 0 & q_\omega & -q_\omega & 0 & 0 \\ 0 & 0 & -q_\omega & q_\omega & 0 & 0 \\ 0 & 0 & 0 & 0 & q_{u_d} & 0 \\ 0 & 0 & 0 & 0 & 0 & q_{u_q} \end{bmatrix} = \begin{bmatrix} 0.01 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3500 & -3500 & 0 & 0 \\ 0 & 0 & -3500 & 3500 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.001 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.1.1b}$$

$$\boldsymbol{R} = \begin{bmatrix} r_{\Delta u_d} & 0 \\ 0 & r_{\Delta u_q} \end{bmatrix} = \begin{bmatrix} 100 & 0 \\ 0 & 100 \end{bmatrix} \tag{8.1.1c}$$

Although longer horizons could potentially enhance process control quality and robustness, such an approach did not yield significantly better results that would justify the noticeably higher complexity. Furthermore, the speed reference provided to the controller is assumed to be constant, as the control is generally causal and therefore should not anticipate future speed requirements. In scenarios where such future reference could be provided to the controller, the NMPC algorithm would benefit from a prolonged prediction horizon. Although the entire reference is available a priori in this thesis, a real-life causal scenario is assumed, and therefore ω_{ref} is held constant for each iteration. However, the prediction horizon cannot be too short for the controller to fully leverage the potential of the dynamic system. As described by [32], a prediction horizon of at least four steps is necessary to successfully achieve field weakening, which is typically a desired aspect of IPMSM control.

Given that all variables in the optimization problem are normalized to a range of $[-1, 1]$, the appropriate magnitude of the coefficients of the logarithmic barrier functions is quite small. The coefficient values used in the final implementation of the algorithm are listed in Table 8.1. In order to tune the controller to successfully handle field weakening operations of the IPMSM, the coefficients were selected with a very low magnitude. Bigger values caused the controller to behave suboptimally and the control results were not satisfactory.

Table 8.1: Logarithmic barrier functions coefficient values

Constraint	Coefficient value ρ
$\sqrt{u_{dn}^2 + u_{qn}^2} \leq 1$	$5 \cdot 10^{-6}$
$\sqrt{i_{dn}^2 + i_{qn}^2} \leq 1$	$5 \cdot 10^{-6}$

8.2 Pruning Parameters

As described in Section 5.1, the pruning application presents six different thresholds for term pruning, each with a distinct impact on the cost function. During the tuning phase of the controller, it is appropriate to first set the parameters of the cost function itself with a higher pruning threshold, resulting in fewer terms being omitted from the computations. After this process, once the control results are satisfactory, the pruning threshold can be adjusted more aggressively to further reduce the complexity of the function while still maintaining the desired control accuracy. The set of pruning thresholds used in the final implementation is listed in Table 8.2. All pruning was performed with relative settings, where terms with coefficients $\delta\times$ smaller than the maximum coefficient present are removed. The thresholds mentioned in Table 8.2 have the following meanings: δ_S state equations pruning, δ_{SC} constraint equations pruning, δ_∇ standard part of the cost function differentiation with respect to the control inputs pruning, $\delta_{\nabla C}$ denominator parts of the cost function differentiation with respect to the control inputs pruning, δ_{CF} evaluation of the standard part of the cost function pruning, and δ_{CFC} arguments of the logarithms of the cost function pruning.

Table 8.2: Pruning threshold values

Threshold	Value
δ_S	10^4
δ_{SC}	10^3
δ_{CF}	10^3
δ_∇	500
$\delta_{\nabla C}$	500
δ_{CFC}	10^3

As outlined in Section 7.2, suitable pruning threshold settings can significantly reduce the complexity of the cost function while preserving the accuracy of the solution. The values of Table 8.2 were selected for the final implementation based on the statistical performance examination of different pruning threshold values (see Table 7.4), where the magnitude of the most suitable thresholds for a similar

application was found to be around the range of 10^3 and 10^5 based on the REM and PEM values. After additional empirical testing, some of the thresholds could be lowered further without losing accuracy and while reducing the computational burden. For this case, the following improvements were achieved. If no pruning was applied, resulting in a function composed of unpruned state equations and constraint terms, the cost function, described as a sum of terms (4.3.4), would contain more than 800,000 such terms just in its standard part, rendering this case highly impractical. Pruning with the threshold values listed in (8.2), the number of terms present in each equation is drastically reduced. This is illustrated in Table 8.3, which lists the number of terms present in the state equations of i_d, i_q and ω for steps three and four of the prediction horizon.

Table 8.3: Number of terms of the original and the pruned state equations comparison

Function	$i_d(k+3)$	$i_q(k+3)$	$\omega(k+3)$	$i_d(k+4)$	$i_q(k+4)$	$\omega(k+4)$
Original	209	228	183	6,087	5,902	7,201
Pruned	22	21	20	34	28	31

8.3 Optimization Parameters

For the optimization process, an additional set of parameters was determined based on the parallelized BFGS algorithm (Algorithm 4). A total of 512 search agents were initialized in each iteration, and the static optimization-iteration count was established at two. To fully exploit the parallelism of the optimization, data from previous iterations were not used for the location initialization of new agents. Although reusing the values of the optimization variable achieved in the previous iteration could accelerate the convergence of a particular agent initialized with these values, it would not enhance the overall convergence of the algorithm. The remaining agents would still need to explore potentially new areas of the cost function, resulting in more iterations required to locate a local minimum. The maximum number of modified line searches performed by each agent during each iteration of the BFGS optimization was set to five, and the constant c utilized during the line search (see Eq. (2.2.2)) was chosen as $c = 1 \cdot 10^{-4}$.

With a prediction horizon length of $N = 4$ and two control inputs, the vector of the optimization variable consists of eight elements, each constrained to a range of $[-1, 1]$. This defines the search domain as an eight-dimensional hypercube centered at $[0, 0, \cdots, 0]$ with a side length of 2. Although 256 search agents could theoretically cover this hypercube equally, with one agent initialized at each corner, this approach

would not yield satisfactory results. Instead, a *Halton sequence* was employed to cover the search domain as uniformly as possible, with one agent always initialized at zero coordinates. The *Halton sequence* algorithm generates starting locations using a vector of prime numbers of length n, where n corresponds to the required dimensionality of the space. [23] The algorithm utilizes *radical inverse function* (8.3.1) which maps a non-negative integer $Z \in \mathbb{Z}^+$ to a value in $[0, 1)$ using a prime number base b. The coefficients $a_j(k)$ are calculated by expressing Z in the base b as $Z = a_0 + a_1 b + a_2 b^2 + \cdots + a_m b^m$ where $a_j \in \{0, 1, \ldots, b-1\}$. The fractional number $\phi_b(k)$ is then obtained by writing the digits of $\boldsymbol{a}$ in reverse order: $\phi_b(k) = a_m b^{-1} + a_{m-1} b^{-2} + \cdots + a_0 b^{-(m+1)}$.

$$\phi_b(Z) = \sum_{j=0}^{\infty} a_j(k) b^{-(j+1)} \tag{8.3.1}$$

Although the locations appear to be randomly generated, the outcome is deterministic. The low discrepancy of the sequence, which indicates that the agents are distributed across the space as evenly and uniformly as possible, is best achieved with a vector of consecutive prime numbers starting from the smallest one. [30] As each prime is used to generate one dimension, a vector $b = \{2, 3, 5, 7, 11, 13, 17, 19\}$, with a length of eight elements, was selected. The starting locations are generated once at the start of the simulation using a dedicated function described in Listing 6.7 and stored on the device (GPU).

8.4 Other implementation notes

An integrator, which enables the controller to better achieve a zero steady-state control error, as discussed in Section 6.2.1, introduces an additional parameter to be set. The integrator structure can be seen in Fig. 8.1 and its gain was set to $K_{wi} = 5$. The time difference between two consecutive integration steps is the controller sampling time, $t_s = 200\,\mu s$.

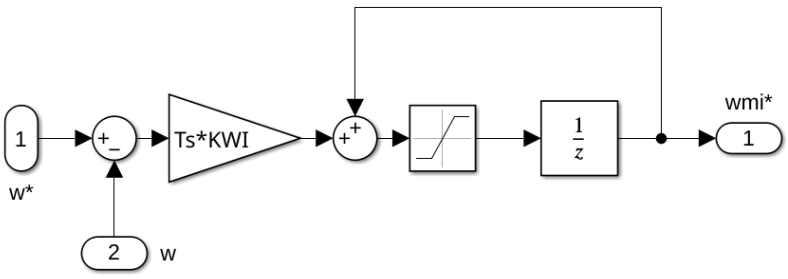

Fig. 8.1: Structure of the reference integrator

As described in Section 5.2, the pruning utility offers the cost function to be evaluated using logarithmic barrier functions and also by directly evaluating the constraining functions. In the final implementation of the algorithm the direct evaluation of the constraints was not used and only the logarithmic function was computed during the evaluation of the cost function.

Another feature of the pruning utility is the pre-computation of recurring terms. This approach was also tested, but for the scenario of NMPC of IPMSM, especially when implemented on the GPU, proved to be inefficient. This might be caused by the fact that if all computed terms are different combinations of a smaller number of variables, those can be stored in the register with faster access times than the global memory and are further exploited by multiply-accumulate operations. The final implementation of the algorithm therefore does not utilize any pre-computing capabilities of the designed pruning utility.

Although the implementation of constraints using logarithmic barriers ensures that the variable always lies strictly inside the constraint, it also presents a disadvantage. If the optimization takes on value that the inequality constraints imposed on the variable or its functions are almost violated, oscillations of the variable or its function occur. This is caused by the fact that the gradient of the underlying optimization function is very steep near the constraints, and the Hessian matrix can become ill-defined. [69] In the NMPC of the IPMSM scenario, this is manifested by the oscillations of the total voltage during field weakening operations. If the voltage is very close to the constraint, a slight ripple can be observed (see Fig. 8.2), which is more pronounced on the currents. Therefore, a control signal filter was implemented to relax the behavior of the voltage near the limits. This filter rounds the voltage increment values Δu_d and Δu_q to a precision of 0.2 if both conditions (8.4.1) are met. That is, the voltage can change only in 0.2 increments if the absolute value of the speed reference is high enough and the direct part of the current is below the threshold. The constants present in these inequalities were obtained empirically.

$$i_{dn} < -0.87 \tag{8.4.1a}$$

$$|\omega_{ref}| > 0.73 \tag{8.4.1b}$$

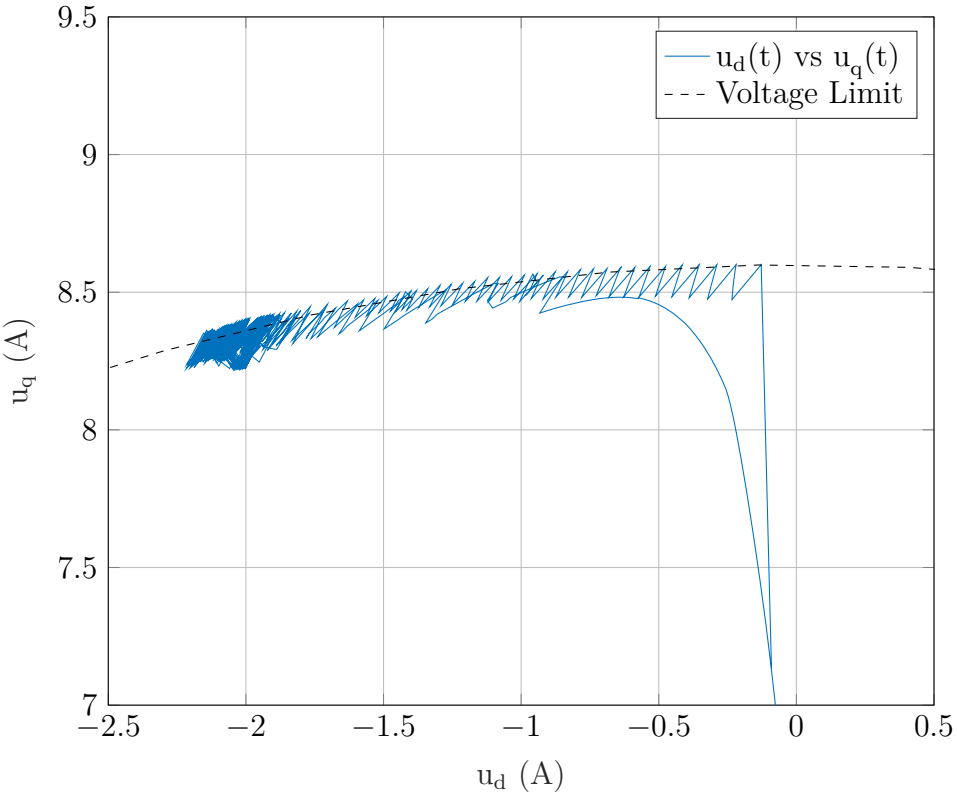

Fig. 8.2: Logarithmic-barrier-induced oscillations

9 NMPC Validation

Two distinct versions of the BFGS-based NMPC for IPMSM were created, with sequential and parallel implementation of the algorithm. Their behavior and the achieved control results will be discussed in this chapter. First, the sequential implementation in pure C as described in Section 6.1 was implemented on a Windows machine and ran on one core of the i5 8400 Coffee Lake (2017) CPU, the parallel version ran on the Nvidia GTX 1050 Ti and was described in more detail in Section 6.2. The sequential and parallel versions were implemented with the same parameters (see Chapter 8). The simulations were performed in Simulink using a fixed size step of $t_{sim} = 20\,\mu$s and automatic solver selection.

The speed reference the IPMSM controller should track, shown in Fig. 9.1, combines ramps, sudden step changes, and constant signal regions for both positive and negative speeds. The ramps are reaching higher speed values than can be achieved by the motor, indicating those regions where the controller should weaken the magnetic field of the stator naturally instead of continuing to strictly follow the MTPA curve. Sudden step changes occur between positive values and zero speed as well as negative values and zero speed, later also very large changes between positive and negative speeds directly. The goal of the designed NMPC controller was to track the reference without excessive aggressiveness and with minimal ripple. Step changes should be tracked as fast as possible without significant reference overshoots and while staying within the constraints placed on voltages and currents. Those versions of the control algorithm that implement field weakening should utilize it during higher magnitude values of reference during ramping and subsequent saturation at $150\,\mathrm{rad}\cdot\mathrm{s}^{-1}$. At the simulation time of $t = 2.75\,\mathrm{s}$ a load torque of $M = 0.1\,\mathrm{Nm}$ was applied to the motor to test the controller's ability to compensate for this control disturbance. The total length of the reference signal is 3.5 seconds.

9.1 Control Results

The behavior of the controller tracking the speed reference, configured according to Chapter 8, is illustrated in Fig. 9.2. The controller tracks constant values without ripple, and the settling time depends on the reference change preceding the stationary part. The graph can be quantitatively analyzed by comparing the settling times for negative and positive value changes of varying magnitudes. Table 9.2 lists these values, with $t_{st}(x \to y)$ denoting the settling time for a reference change from x to y $\mathrm{rad}\cdot\mathrm{s}^{-1}$ within a $2\,\%$ error range in relation to $50\,\mathrm{rad}\cdot\mathrm{s}^{-1}$. The times between the first reference crossing and speed settling within $\pm 2\,\%$ of the constant reference, denoted as $t_c(x \to y)$ with a notation similar to the settling times, are also listed

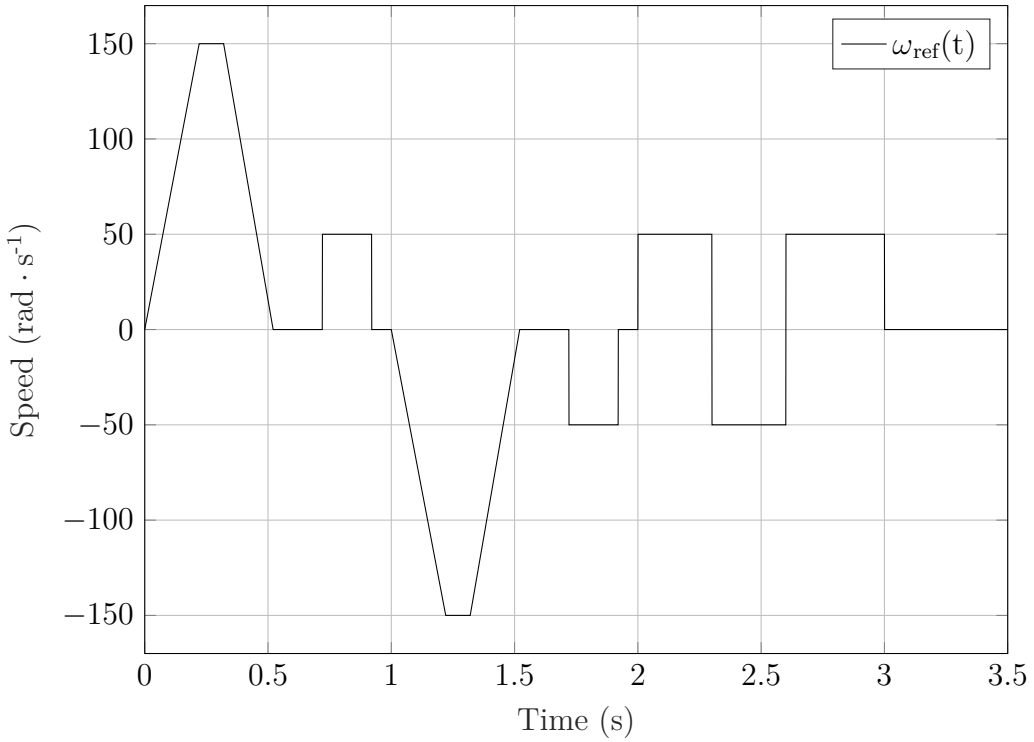

Fig. 9.1: Speed reference

in Table 9.2. The interval between the time when the load torque is applied and when the speed returns within the error range is denoted by t_{sL}. The intervals t_{FW+} and t_{FW-} represent the time difference between the moment when the motor speed rises above $111\,\text{rad}\cdot\text{s}^{-1}$ and when it reaches its maximum during field weakening for positive and negative reference values, respectively.

The scenario depicted in Fig. 9.2, applies an additional load torque of $T_L = 0.1\,\text{Nm}$ at $t = 2.75\,\text{s}$, marked with a dashed-dotted line. This demonstrates the ability of the controller to effectively eliminate introduced control disturbances. The error induced by the load torque was corrected in 10 ms, with the motor speed settling within a 2 % error range of the reference signal. During periods where the tracked reference signal exceeds the motor's nominal speed, reaching $150\,\text{rad}\cdot\text{s}^{-1}$, the controller seamlessly transitions from the MTPA regime to field weakening operations and the motor speed reaches $117.2\,\text{rad}\cdot\text{s}^{-1}$. The field weakening operation is clearly visible in Fig. 9.3, which displays the stator currents in *dq* coordinates. In these regions, the direct part of the current peaks at negative values, while it otherwise remains near zero, which is the desired behavior during MTPA operations. This behavior was achieved without imposing additional constraints on the direct part of the current.

When a steep change in the reference value occurs, a sudden peak in the quadra-

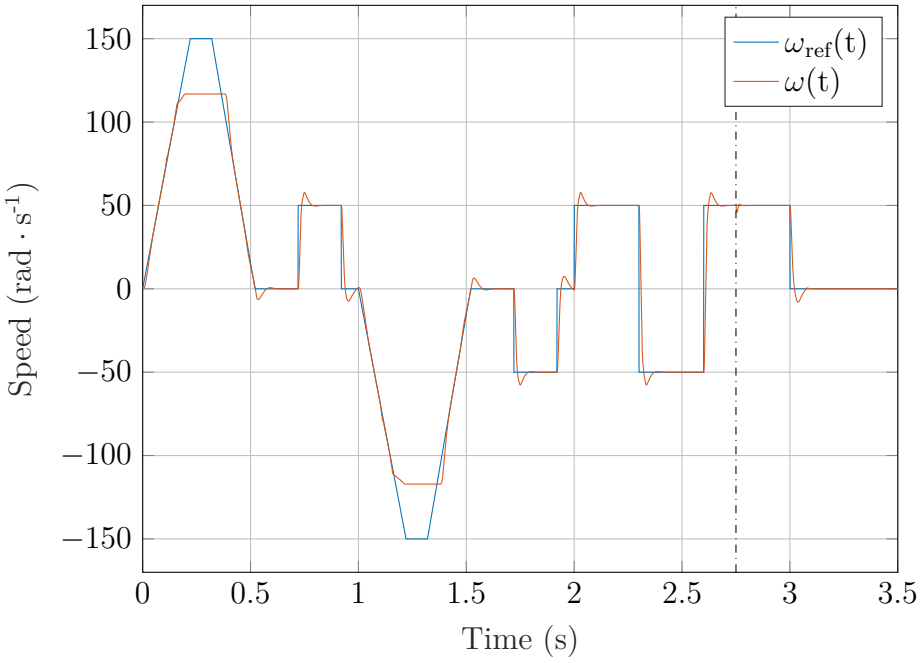

Fig. 9.2: NMPC of IPMSM: Motor Speed

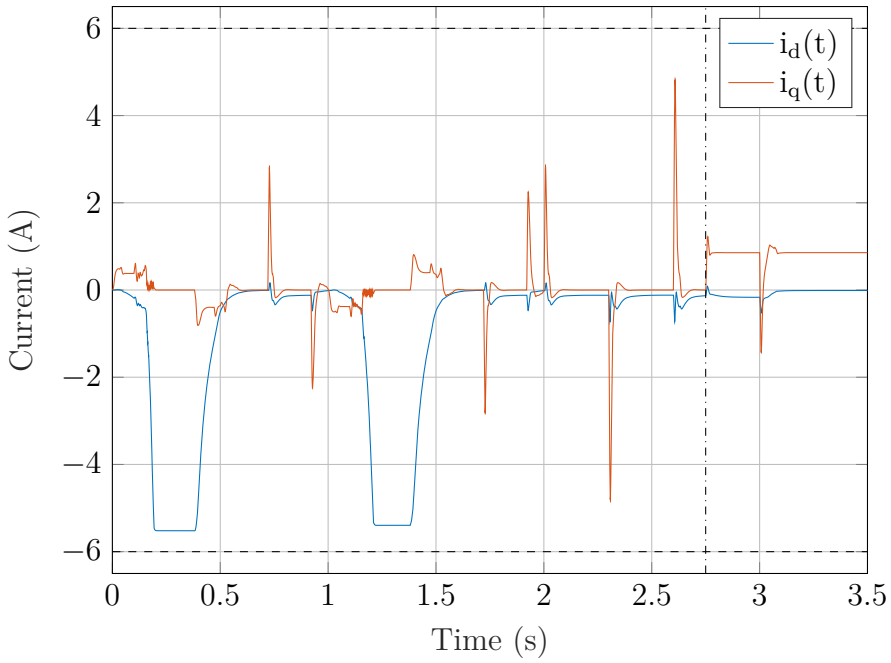

Fig. 9.3: NMPC of IPMSM: Motor Currents

ture part of the current is observed. This leads to rapid speed changes, as dictated by the motor speed equation (3.1.3c). The quadrature part remains within the imposed current constraints and subsequently returns to its previous value. At $t = 2.75\,\mathrm{s}$, a change in the mean value of the quadrature part is noticeable, occurring when the load torque is applied, and the controller compensates for the this disturbance.

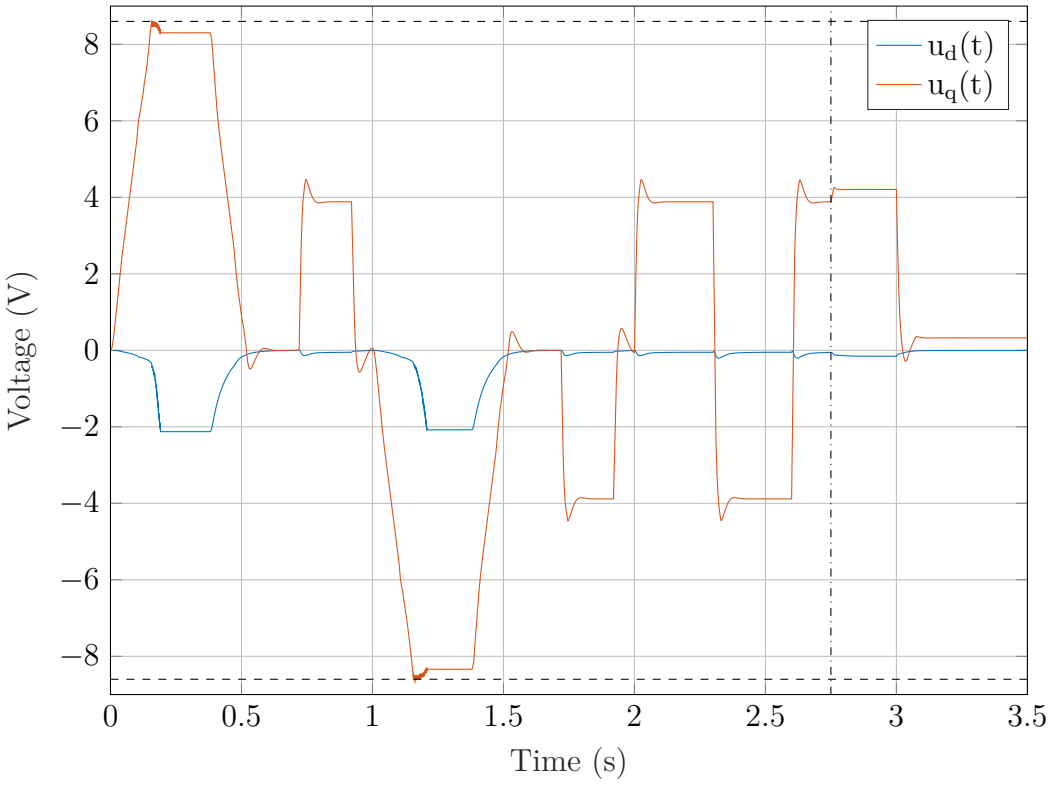

Fig. 9.4: NMPC of IPMSM: Motor Voltages

Fig. 9.5 displays the voltage value in the *dq* coordinates. The total voltage limit of 8.6 V is marked with a dotted line, demonstrating that the imposed constraints were not violated. The figure shows that during MTPA operations, the direct part of the voltage remains near zero, and the quadrature part rises in either direction. Only at the very limit, when the controller transitions to field-weakening operations, does the magnitude of the direct part increase, and the quadrature part decreases only slightly to compensate for the total voltage value.

The current constraint is also upheld, as shown in Fig. 9.6 together with the maximum torque per Ampere curve. After inspecting this visualization, it is evident that the current does not reach its maximum allowed value. During its development, a more aggressive controller tuning was also tested to further enhance its tracking ability. This iteration introduced smaller voltage-increment and currents penalization. The more conservative option chosen in the final implementation, however, is more stable and balances the behavior of the controller during different operation

scenarios in a more robust way. This approach also reduces the stress caused to the motor components. The motor therefore does not reach the reference after a sudden step change in the shortest time possible, as is demonstrated in the comparison with a vector controller in Section 9.2, but does not overshoot this reference value as much, which was selected as the preferred behavior.

The momentary electric torque produced by the motor is shown in Fig. 9.7, clearly indicating a sudden change in the mean value at $t = 2.75\,\mathrm{s}$, where compensation for the applied load torque occurs.

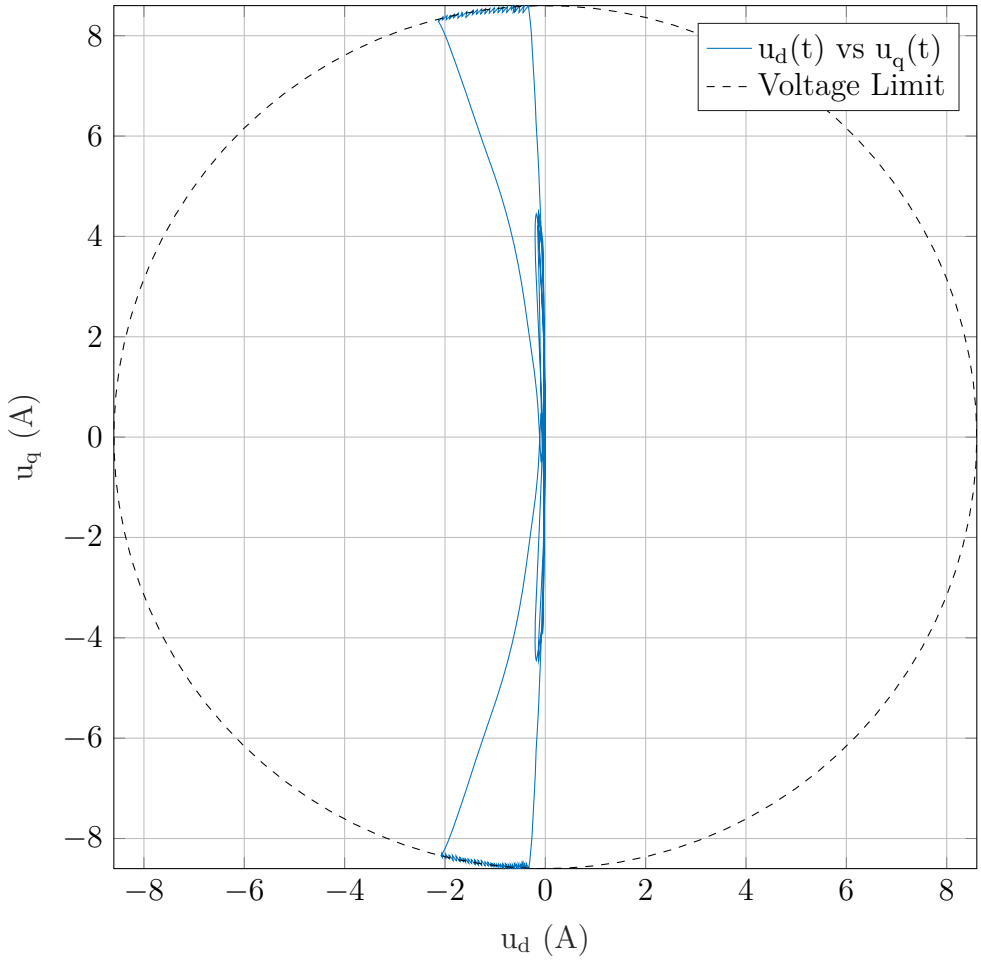

Fig. 9.5: NMPC of IPMSM: Voltages in *dq* coordinates

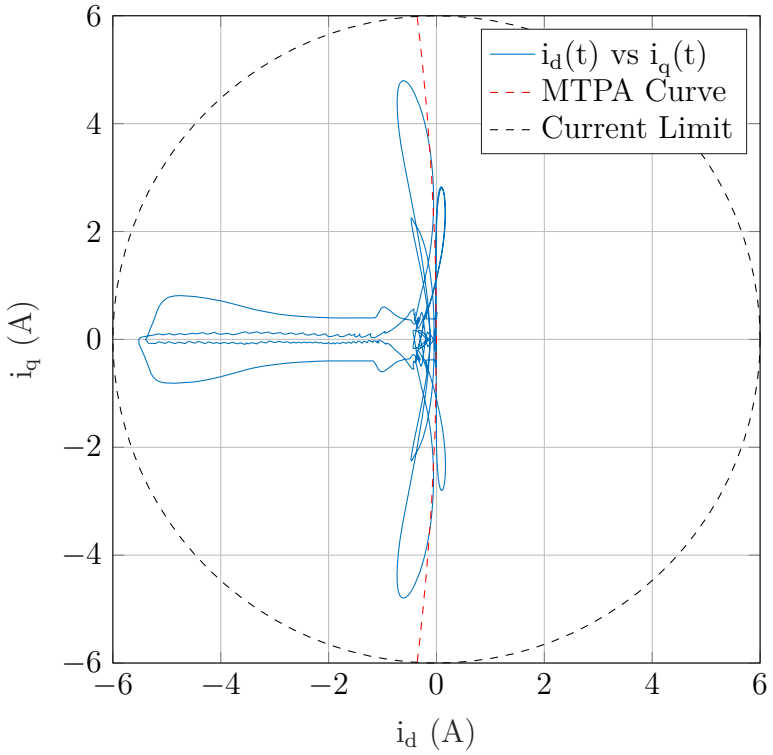

Fig. 9.6: NMPC of IPMSM: Currents in *dq* coordinates

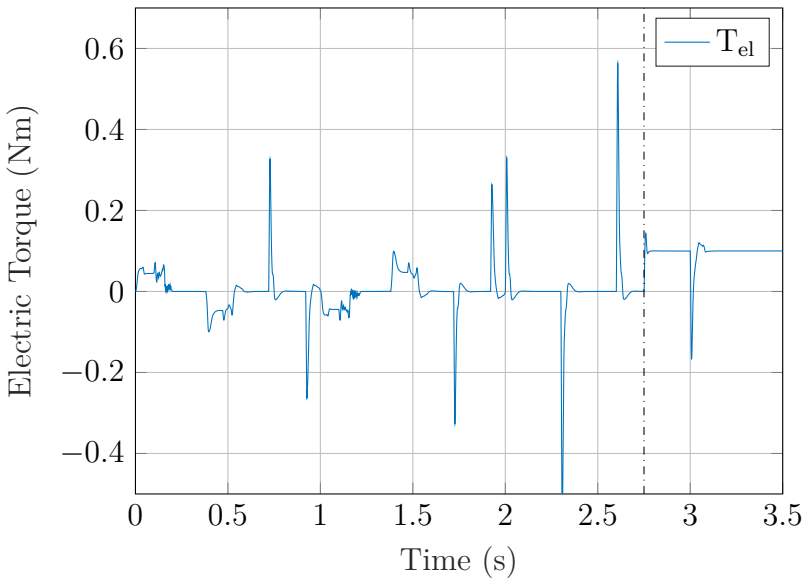

Fig. 9.7: NMPC of IPMSM: Electric Torque

9.2 Vector Control

In order to compare the control capabilities of the designed NMPC against a popular alternative, a vector controller (VC), also known as field-oriented controller (FOC) was designed. Vector control is a popular approach to IPMSM control utilizing a set of two control loops. The first regulates the speed of the motor while the second regulates the stator currents dq, in contrast to another popular method of direct torque control (DTC). In general, VC has better steady-state performance than DTC, which on the other hand has better dynamics. [70]

The designed vector controller uses the structure shown in Fig. 9.8. The electric torque reference is obtained as the output of the speed controller and supplied to the look-up table (LUT) generator. This block is a part of Simulink's motor control blockset and creates references for the current controllers based on the provided mechanical speed and electric torque references. As the imposed voltage and current limits are set very tightly, this control block was not able to generate the current reference LUTs properly for the field-weakening regions and the comparison of NMPC and VC was therefore performed disregarding field-weakening capabilities. The block was set in the linear IPMSM model mode and provided with the motor parameters. Simulink-integrated PI controllers were set in ideal form, in discrete time domain with a sampling period of $t_s = 200\,\mu$ s, and the anti-windup mechanism was enabled using the back-calculation method[1] The current controllers generated a set of voltages u'_d and u'_q that were decoupled using equations (9.2.1). The controllers were tuned to a system response with settling times comparable to those obtained with NMPC and their parameters are listed in Table 9.1. The initial tuning parameters were obtained from [32] and further tuned to this specific scenario.

$$u_d = u'_d - \omega L_q i_q \tag{9.2.1a}$$

$$u_q = u'_q + \omega L_d i_d + \omega \Psi_{PM} \tag{9.2.1b}$$

Table 9.1: Vector control: Parameters of the PI controllers

PI Controller	P	I	Kb
T_e^*	0.3	94	5.5866
i_d^*	0.36	$1.07 \cdot 10^{-3}$	3759.4
i_q^*	0.9	0.075	22.779

[1] https://www.mathworks.com/help/releases/R2024b/simulink/slref/discretepidcontroller.html

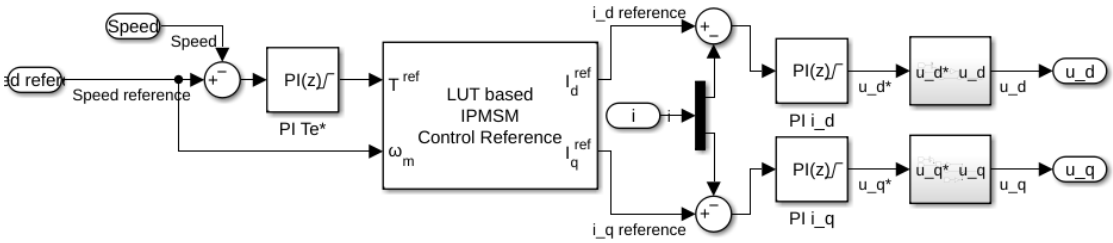

Fig. 9.8: Structure of the vector controller

The comparison of the motor speeds between NMPC and VC can be seen in Fig. 9.9. Although the settling times listed in Table 9.2 are similar for both control methods, the speed of the vector-controlled motor overshoots its reference more significantly after step changes of the reference. The overshoot values ω_{os} for the NMPC and VC are compared in Table 9.3. In contrast, the VC settles faster when the reference is $\pm 50\,\mathrm{rad} \cdot \mathrm{s}^{-1}$ and suddenly returns to zero. The settling times $t_{s_{LT}}$ after the load torque is applied at $t = 2.75\,\mathrm{s}$ are omitted for the vector controller, as the speed stayed in the 2 % error range. All comparison figures are rendered with the speed reference, which is not in scale.

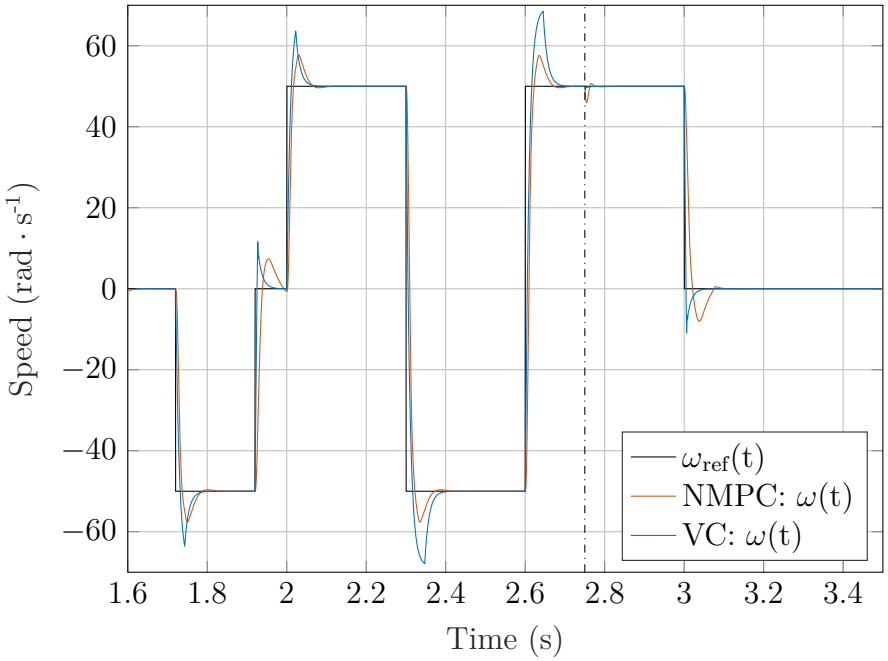

Fig. 9.9: VC Comparison: Speed

The visualization of current was divided into two figures for better readability, Fig. 9.10 for the direct part of the current and Fig. 9.11 for the quadrature. Although the current limit was explicitly established and LUTs were created according to

them, vector-controlled IPMSM tends to overshoot the maximum value of 6 A, which does not occur with NMPC. It is also evident that the control approach of NMPC and VC is slightly different, with VC generating more sudden and more aggressive changes of i_q. This is also apparent after examining the behavior of u_q in Fig. 9.13. The quadrature part of the stator voltage is smoother when generated using the NMPC compared to the VC. As the part where the reference speed rises above the base speed of the motor was omitted, and thus no field weakening operations are performed, no significant observations can be made from the visualization u_d Fig. 9.12.

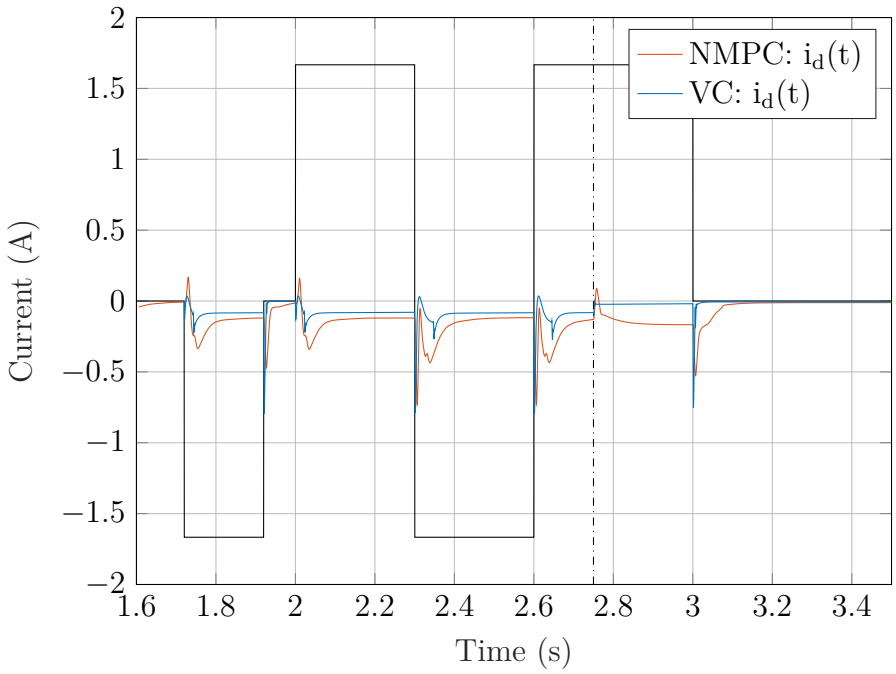

Fig. 9.10: VC Comparison: Direct part of current

The overshooting of the imposed current limit that was discussed earlier is also clearly visible in Fig. 9.14 showing the currents in the dq coordinates together with the ideal MTPA curve. The vector-controlled motor adheres to this curve slightly better, but does not stay within the imposed bounds. Similar visualization of the behavior of the dq voltages can be examined in Fig. 9.15 where the aggressiveness of the vector controller is also apparent.

Another criterion for the comparison of NMPC and VC is the total energy consumption summarized in Table 9.4. The quantities present in the table have the following meaning: E is the total energy consumption obtained by (9.2.2a) as an integral of the instantaneous electrical power $P(t)$, E_D is the total energy drawn from the power source (9.2.2b) without regeneration. The numbers demonstrate a 8.77 % reduction in the energy consumption by the NMPC and a 9.01 % drop in the drawn

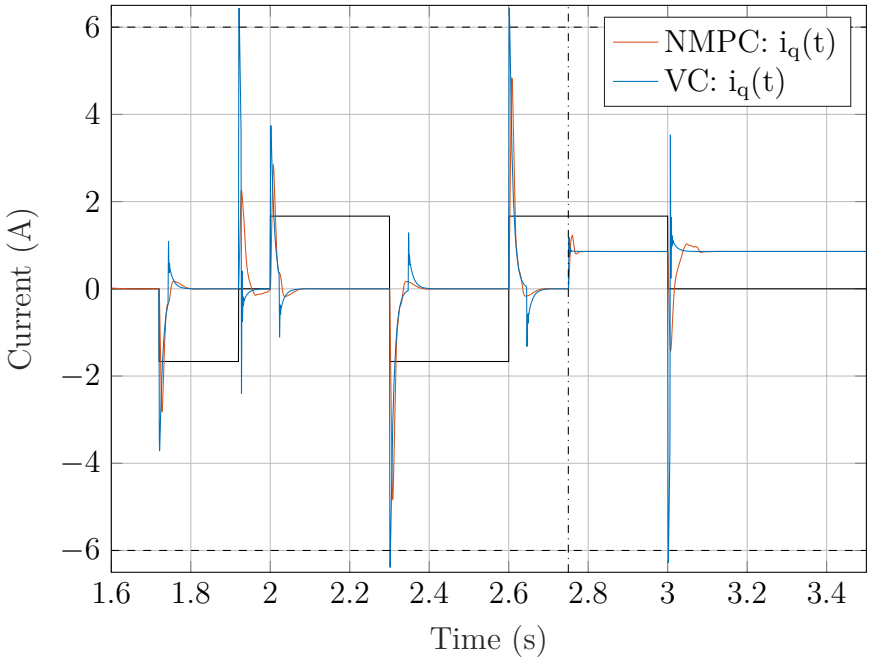

Fig. 9.11: VC Comparison: Quadrature part of current

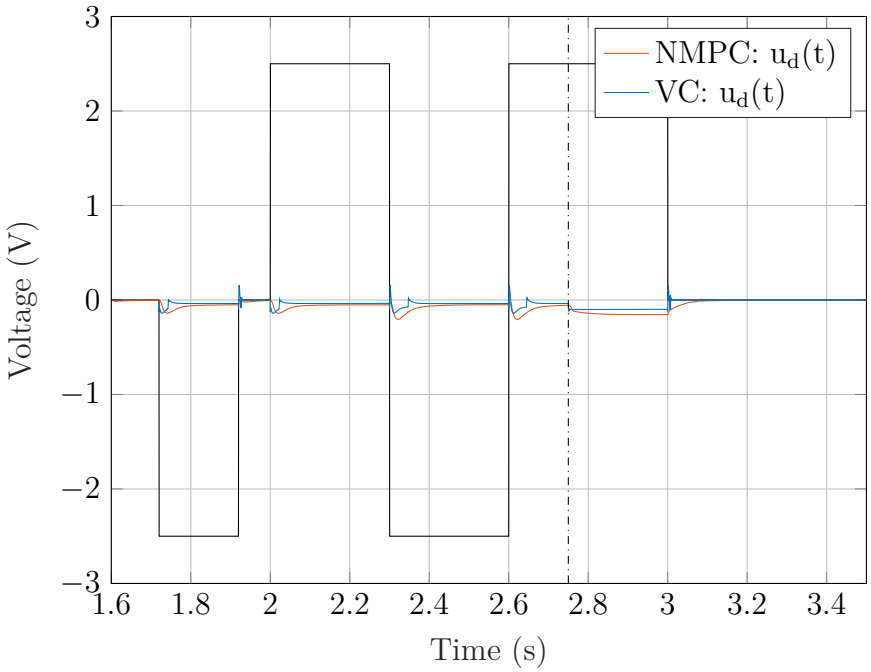

Fig. 9.12: VC Comparison: Direct part of voltage

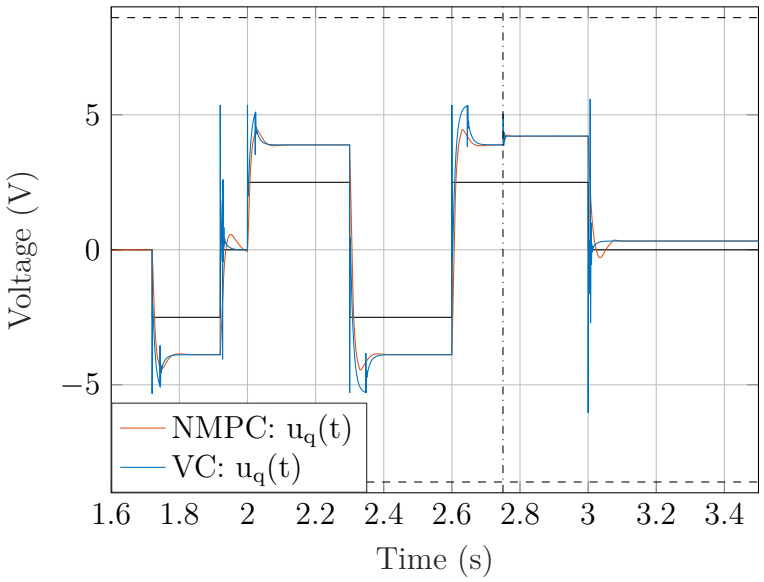

Fig. 9.13: VC Comparison: Quadrature part of voltage

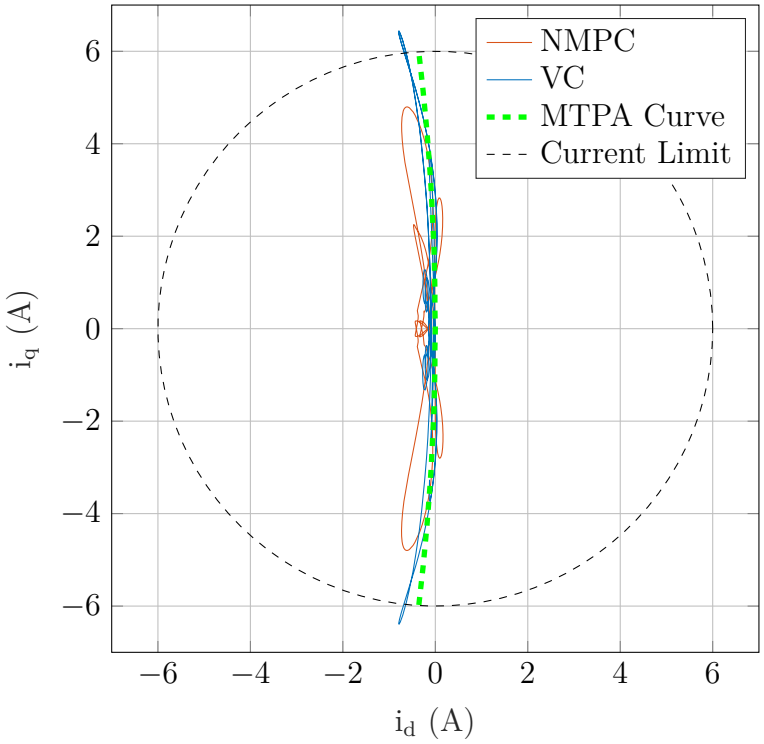

Fig. 9.14: VC Comparison: Currents in *dq* coordinates

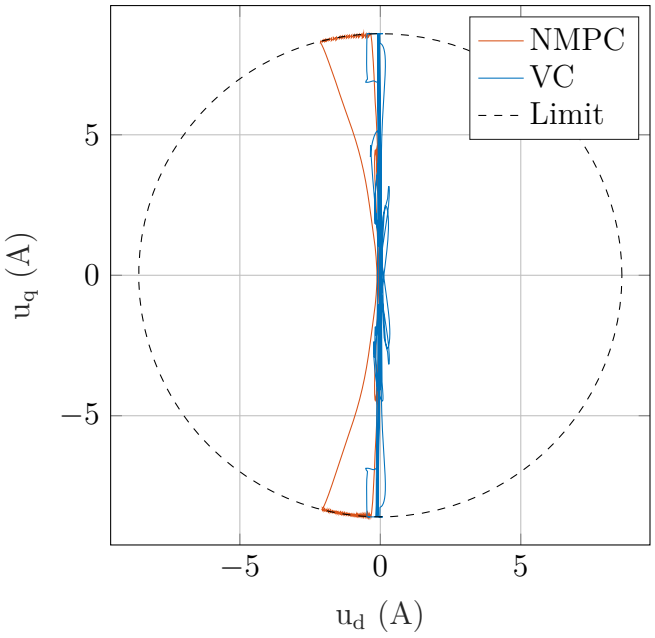

Fig. 9.15: VC Comparison: Voltages in dq coordinates

energy. The instantaneous electrical power for both NMPC and VC is visualized in Fig. 9.16. For positive regions of power, NMPC and VC behave similarly with VC being slightly more aggressive. However, for negative regions, the VC overshoots are more significant.

$$E = \frac{3}{2} \int P(t)\ \mathrm{d}t, \text{ where } P(t) = u_d(t)i_d(t) + u_q(t)i_q(t) \tag{9.2.2a}$$

$$E_D = \frac{3}{2} \int P^+(t)\mathrm{d}t, \text{ where } P^+(t) = \max\{u_d(t)i_d(t) + u_q(t)i_q(t), 0\} \tag{9.2.2b}$$

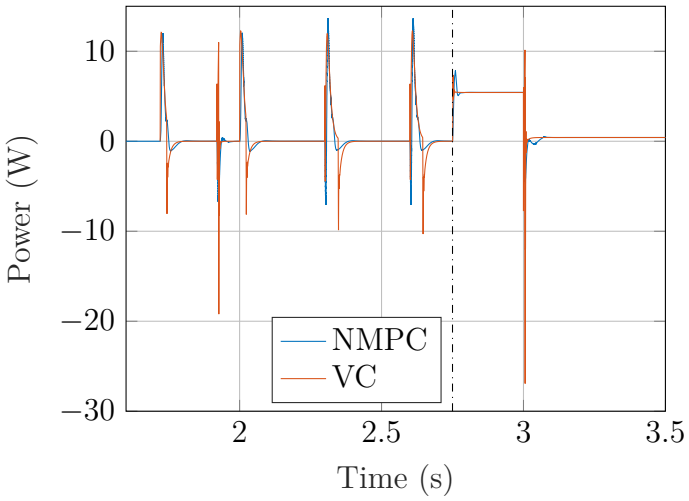

Fig. 9.16: VC Comparison: Instantaneous electrical power

Table 9.2: Settling times comparison for NMPC and VC, values are listed in seconds

	NMPC	VC
$t_{st}(0 \to 50)$	0.0576	0.0515
$t_{st}(-50 \to 0)$	0.0661	0.0319
$t_{st}(50 \to 0)$	0.0663	0.0318
$t_{st}(0 \to -50)$	0.0577	0.0502
$t_{st}(50 \to -50)$	0.0637	0.0764
$t_{st}(-50 \to 50)$	0.0631	0.0789
$t_c(0 \to 50)$	0.0406	0.0395
$t_c(-50 \to 0)$	0.0473	0.0266
$t_c(50 \to 0)$	0.0476	0.0265
$t_c(0 \to -50)$	0.0407	0.0387
$t_c(50 \to -50)$	0.0408	0.0600
$t_c(-50 \to 50)$	0.0407	0.0621
t_{sL}	0.0105	0
t_{FW+}	0.0370	-
t_{FW-}	0.0553	-

Table 9.3: Speed overshoots comparison for NMPC and VC, values are listed in rad·s^{-1}

	NMPC Parallel	VC
$\omega_{os}(0 \to 50)$	7.722	13.3714
$\omega_{os}(-50 \to 0)$	7.380	11.620
$\omega_{os}(50 \to 0)$	7.378	11.627
$\omega_{os}(0 \to -50)$	7.619	13.555
$\omega_{os}(50 \to -50)$	7.629	18.531
$\omega_{os}(-50 \to 50)$	7.636	17.793

9.3 Parallel and sequential implementation comparison

The parallel implementation of the NMPC described in Section 6.2 and the sequential implementation from Section 6.1 are compared in the following. The parameters outlined in Chapter 8 were used for both implementations. The two versions of the implementation showed very similar overall behavior, as can be seen in Fig. 9.17 when comparing the speed of the IPMSM. The similarity can be further demonstrated in Table 9.4 listing the total energy consumption during the simulations. Both quantities E and E_D were computed over the entire duration of the simulation and also just for the shorter period used during the VC comparisons starting from the simulation time $t = 1.6\,\mathrm{s}$ onward. The implementations produce similar results, the parallel being slightly better if the field weakening regions are included, while the sequential performing better if they are omitted.

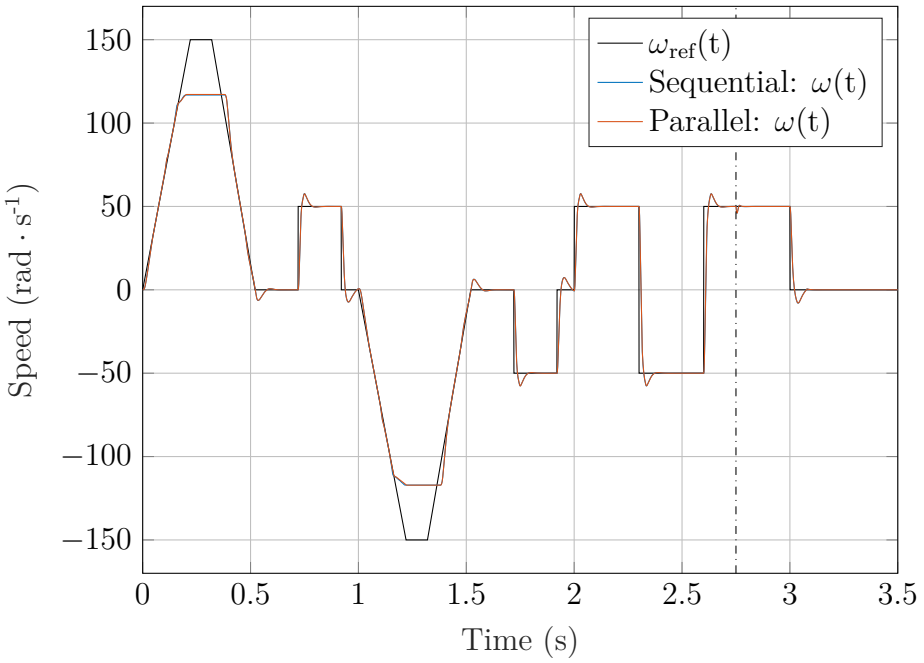

Fig. 9.17: Sequential comparison: Speed

A difference between the two versions can be observed when the total voltage value gets near the imposed constraints. Fig. 9.18 visualizes a detail of the voltage behavior in the first field weakening region. The oscillations that occur when the controller reduces the value of u_q to accommodate the drop in u_d, are less pronounced with sequential implementation. The steady-state value of u_q during field weakening is slightly higher for the sequential version. This is probably caused by the different

approach to the evaluation of the logarithmic function, whose value has a significant impact in this region.

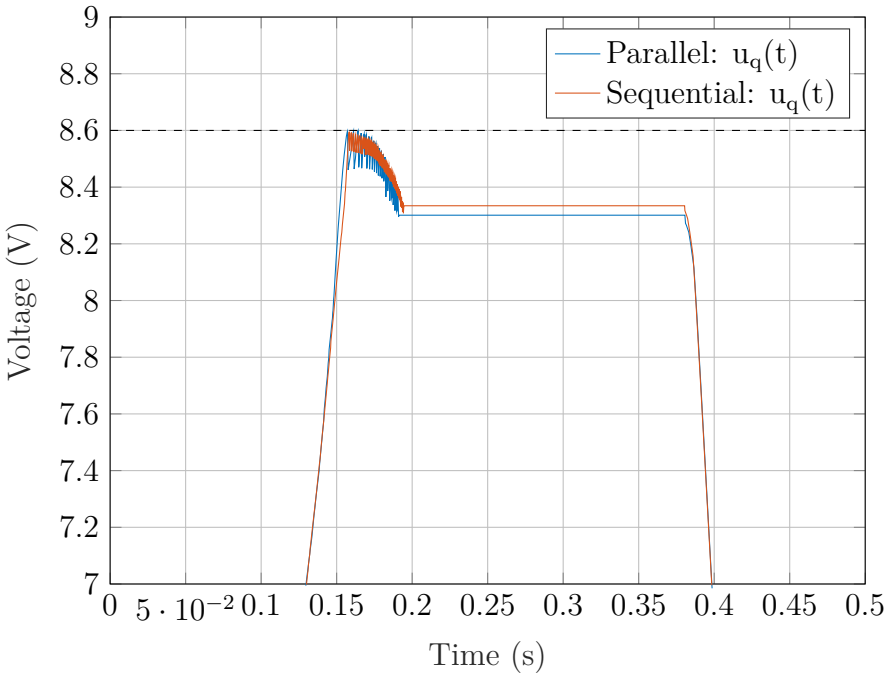

Fig. 9.18: Sequential comparison: u_q detail

Another detailed comparison is visualized for the value of i_d in Fig. 9.19. The figure shows the maximum negative value reached by each controller during field weakening. It is evident that each sets the value of i_d lower in a different field-weakening region, and the difference is approximately 0.1 A for both cases. During most operations, both versions behave identically, although for some specific scenarios with potentially higher demands placed on the precision of floating-point operations, a difference, such as in Fig. 9.20, is observable.

Table 9.4: Energy consumption comparison for parallel NMPC, sequential NMPC and VC

	NMPC Parallel	NMPC Sequential	VC
E (J)	9.6732	9.75987	-
E_D (J)	10.1223	10.1865	-
E (J), $t > 1.6\,\mathrm{s}$	1.9501	1.9247	2.1376
E_D (J) $t > 1.6\,\mathrm{s}$	2.1867	2.1663	2.4031

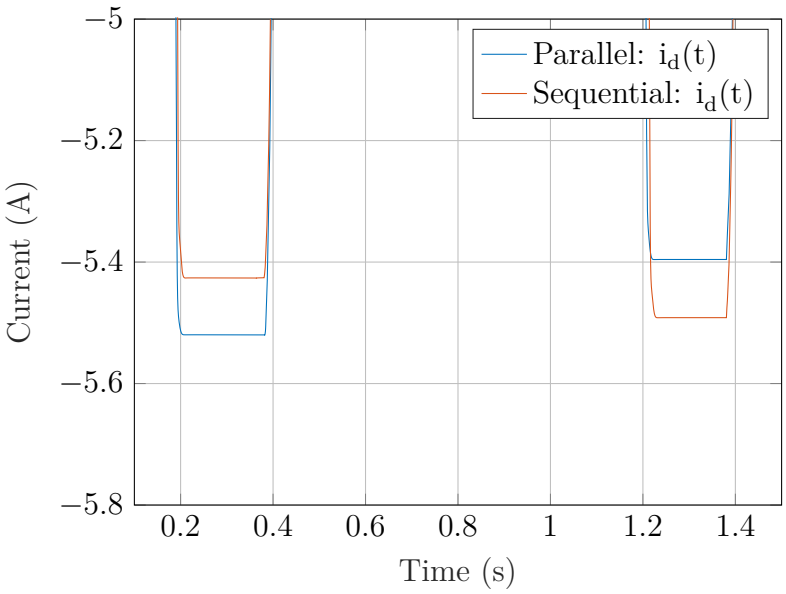

Fig. 9.19: Sequential comparison: i_d detail

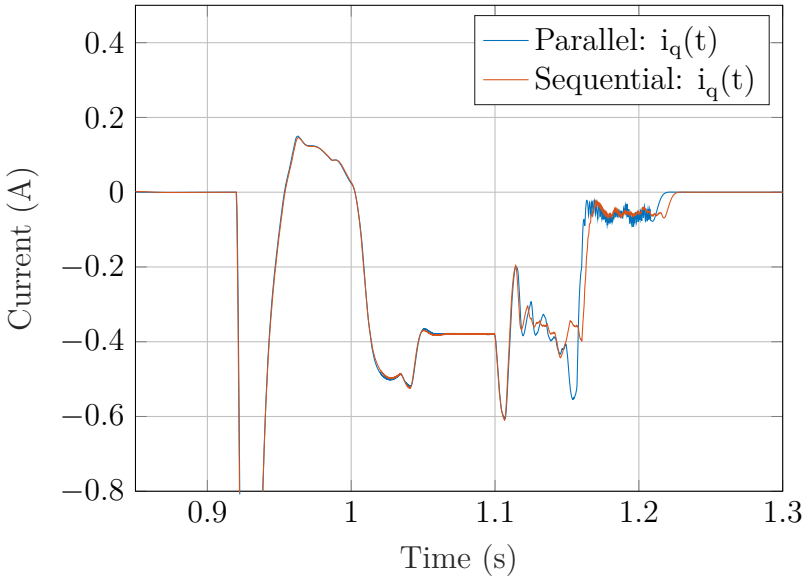

Fig. 9.20: Sequential comparison: i_q detail

9.4 Time Efficiency

To determine the actual time demands of the algorithm, the code was isolated and ran on the GPU while eliminating most of the potentially interfering processes. To test it on real data, the actual inputs to the NMPC during the coarse of the simulation were captured and stored in a file. The program iteratively launched the NMPC algorithm with the data read from this file used as the input was created. This approach suppressed the demands that would otherwise be posed on the device if the NMPC algorithm was launched from Simulink. The compute mode of the Nvidia GTX1050 Ti was set to *Exclusive thread* using the `sudo nvidia-smi -c 3` command. As this GPU does not support direct user clock management, the code was run four times in total, with the first three iterations being used to preheat the device, and the final one was then used for the measurements. The computation times obtained in this way are visualized in Fig. 9.21 in blue. The moving average filter shows the mean value of the last $n = 10$ iterations. The average value of all iteration times is $\overline{t_{IT}} = 56.0868\,\mu$s.

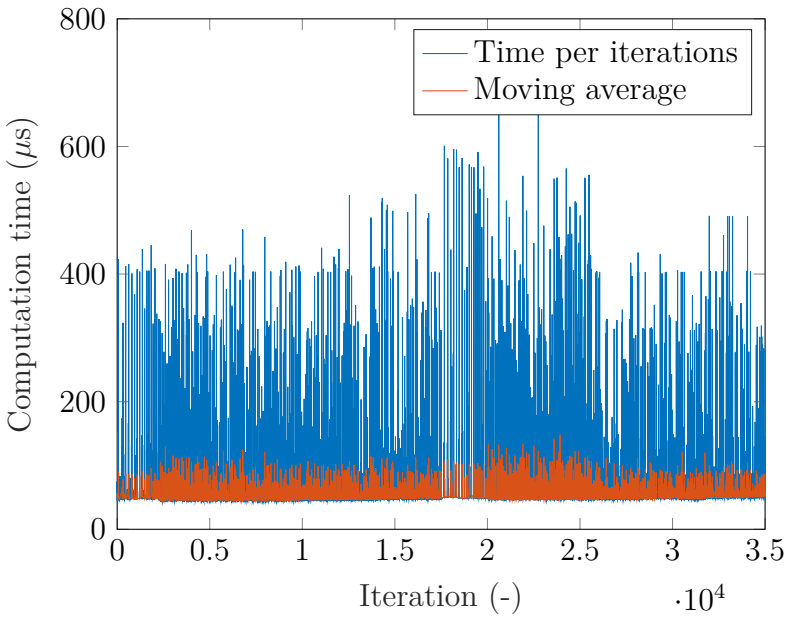

Fig. 9.21: Computational burden of the NMPC algorithm

The representation of the computational burden in Fig. 9.21 can be misrepresented due to the total number of 35,000 iterations, where each outlier is clearly visible. This was partially demonstrated by the moving average filter value. Furthermore, the times needed to compute each iteration of the NMPC algorithm were divided into bins with a width of 50 μs each and plotted in a logarithmic histogram. As is apparent from Fig. 9.22, most iterations were computed faster than 50 μs. Of

the total of 35,000 iterations, in 590 cases the time required to compute the iteration was greater than 200 μs, representing 1.686 % of the cases.

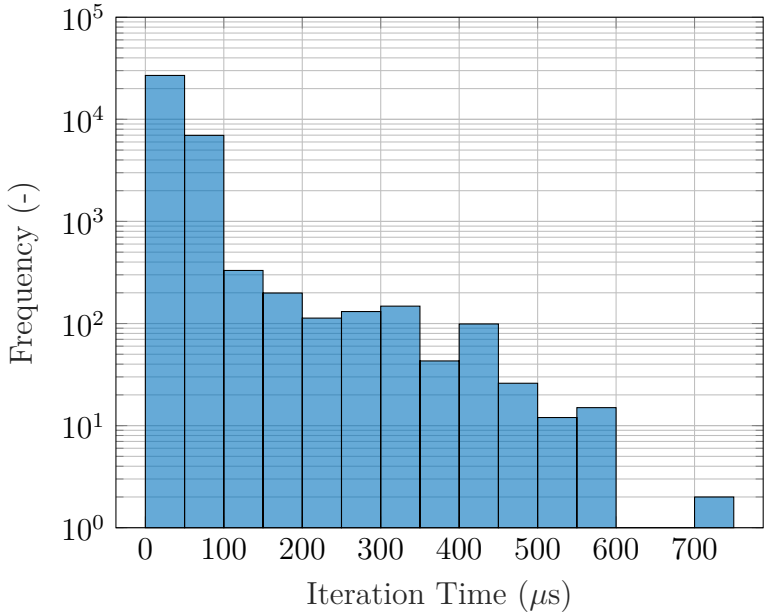

Fig. 9.22: Distribution of the computation times

A more detailed overview of the time efficiency of the tasks performed during each iteration was obtained using the Nvidia Nsight Systems Profiler. Fig. 9.23 represents an average iteration of the parallelized program implemented as described in Section 6.2. In this figure, shown in red, the data transfers between the host and the device can be observed at the beginning and at the end of the iteration. Although the transferred data volume is very small, these operations take up a large percentage of the total time of the iteration. The upload of the `float initialValues` array of six elements on the device takes $\sim 12\,\mu$s, while the download of the `float bestVarVal` array presents a similar burden. Between these two data-transfer operations, the actual optimization on the device is performed. The highlighted section represents the parallelized search, while the second section is the subsequent determination of the best agent. The visual representation of these two tasks in the profiler can be misleading due to the asynchronous nature of the search loop (see Listing 6.6). The task of identifying the best search agent needs to wait for all the agents to finish the optimization, and therefore takes up more time in the profile of the iteration. The total time of these two operations, that is, from when the agents are initialized to the moment when the optimal control input is found, is approximately 30 μs.

Although in more than 98 % cases the NMPC iterations are computed faster than the set sampling period of $t_s = 200\,\mu$s, and therefore demonstrate the real-

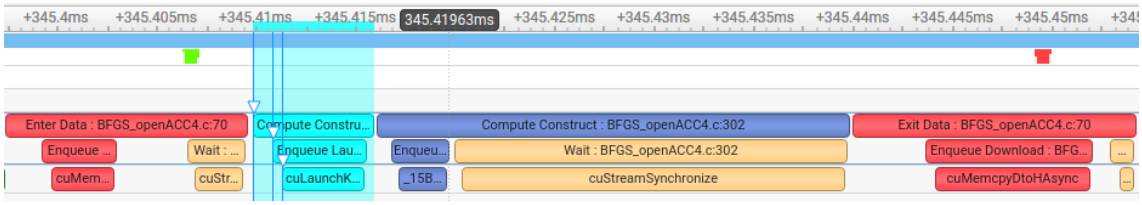

Fig. 9.23: Profile of the NMPC iteration

time potential of the algorithm, the code isolation described earlier in the section was not sufficient for all iterations to meet this threshold.

When the computation times of the parallel implementation Fig. 9.21 are compared to the computational burden of the sequential version shown in Fig. 9.24, an expected conclusion can be drawn. The parallelized approach significantly speeds up the computations, in this case roughly by a factor of more than 200. Interestingly, even though no correlation between the speed reference value and computational demands is apparent for the parallel NMPC, the per-iteration time for the sequential implementation is clearly higher in the field weakening regions. As the optimization iteration count is fixed to two, this is probably caused by the general computational demands posed during the field weakening operations. The computational time of large expressions in C can vary depending on the specific values of the variables present, as can the memory access pattern to those variables or other intermediate results. [43]

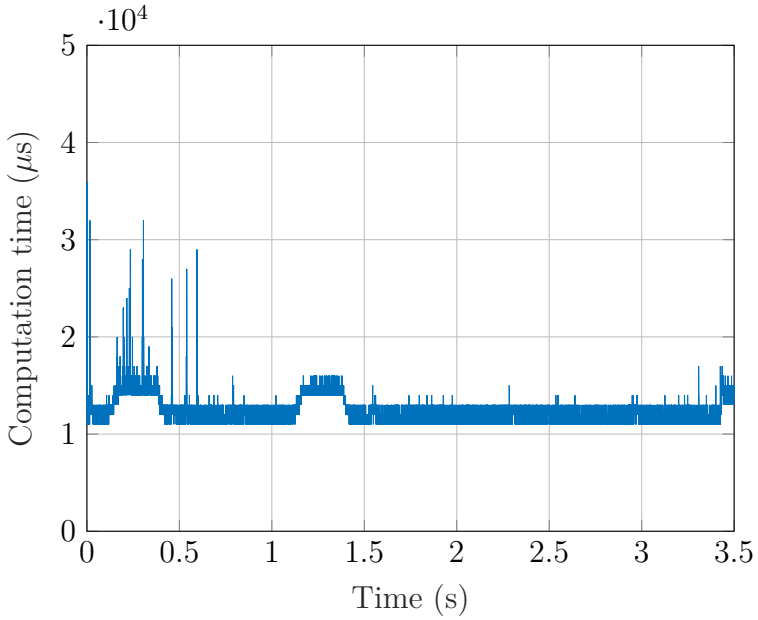

Fig. 9.24: Sequential comparison: Computational burden

Conclusion

This thesis introduces a novel approach to NMPC implementation with a focus on synchronous motor control. The significant computational challenges that limit the real-time application of NMPC for very fast systems were addressed by reducing the complexity of the problem while preserving the robustness of the solution and by exploiting the capabilities of parallel computations. The designed parallelized optimization algorithm was tailored to the specific needs of NMPC of IPMSM but the underlying principles tackled could be used to address a broader field of optimization-based problems. To fully take advantage of the parallel capabilities of the hardware, a dedicated optimization solver was developed.

The NMPC was used to control the internal permanent magnet synchronous motor utilizing its state-space model. The objective was to minimize the cost function, which was tuned to ensure the correct behavior of the motor. Because of the complex nonlinear dynamics of the IPMSM system and the necessary voltage and current constraints imposed, this function represents a nonconvex constrained optimization problem, which poses significant computational-related demands on the controller. The algorithm described in Chapter 4 addresses the real-time NMPC issue both for offline preparations of the control mechanism and also for the online regulation. A specialized utility was implemented in Chapter 5 to reduce the computational burden of the optimization by identifying parts of the cost function that can be safely eliminated from future calculations. As was demonstrated in Section 7.2, a significant reduction in complexity can be achieved in this way without sacrificing the accuracy and precision of the function.

The parallel abilities of the controller implemented on a GPU, which were partly introduced to speed up the optimization tasks and partly to address the common issues of nonconvex optimization, were validated in Section 7.1. A set of various benchmarking functions was successfully optimized and their global minima were identified. The capabilities of the algorithm implemented to seamlessly combine the reduction in complexity of offline preparations and the parallel properties of online optimization were tested in Chapter 9. The NMPC controller handled various control objectives well and, compared to the popular vector control of IPMSM, it proved to be more energy efficient in the tested scenario.

The computational demands were addressed in Section 9.4. Although promising real-time capabilities were demonstrated, additional hardware capabilities in the area of real-time computations would be necessary for further development.

Bibliography

[1] Almér, S., Besselmann, T., and Ferreau, J. Nonlinear model predictive torque control of a load commutated inverter and synchronous machine. In *2014 International Power Electronics Conference (IPEC-Hiroshima 2014 - ECCE ASIA)* (2014), pp. 3563–3567.

[2] Armijo, L. Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific Journal of Mathematics 16*, 1 (1966), 1–3.

[3] ASUS. ASUS Expedition GeForce® GTX 1050 Ti OC edition eSports gaming graphics card 4GB GDDR5. `https://web.archive.org/web/20170314191119/https://www.asus.com/Graphics-Cards/EX-GTX1050TI-O4G/`. Accessed: 2025/04/04.

[4] Bock, H. G., Diehl, M. M., Leineweber, D. B., and Schlöder, J. P. A Direct Multiple Shooting Method for Real-Time Optimization of Nonlinear DAE Processes. In *Nonlinear Model Predictive Control* (Basel, 2000), F. Allgöwer and A. Zheng, Eds., Birkhäuser Basel, pp. 245–267.

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

[6] Broyden, C. G. The Convergence of a Class of Double-rank Minimization Algorithms. *IMA Journal of Applied Mathematics 6*, 3 (Sept. 1970), 222–231.

[7] Camacho, E., and Bordons, C. *Model Predictive Control*, vol. 13. Springer, Jan. 2004.

[8] Camacho, E. F., Berenguel, M., and Rubio, F. R. *Advanced control of solar plants*. Springer London, 1997.

[9] Camacho, E. F., and Bordons, C. Nonlinear Model Predictive Control: An Introductory Review. In *Assessment and future directions of nonlinear model predictive control*, R. Findeisen, F. Allgower, and L. T. Biegler, Eds., 2007 ed., Lecture notes in control and information sciences. Springer, Berlin, Germany, July 2007, ch. 1, pp. 1–16.

[10] Chen, Z., Lai, J., Li, P., Awad, O. I., and Zhu, Y. Prediction Horizon-Varying Model Predictive Control (MPC) for Autonomous Vehicle Control. *Electronics 13*, 8 (2024).

[11] Cleverson, L., Ledur, D., Zeve, C., and Anjos, D. Comparative Analysis of OpenACC, OpenMP and CUDA using Sequential and Parallel Algorithms. In *WSPPD 2013* (Aug. 2013).

[12] Diwan, S., and Deshpande, S. Nonlinear Model Predictive Controller for the Real-Time control of Fast Dynamic System. In *2019 International Conference on Communication and Electronics Systems* (July 2019), pp. 289–294.

[13] Findeisen, R., and Allgöwer, F. An introduction to nonlinear model predictive control. *21st Benelux Meeting on Systems and Control* (Jan. 2002).

[14] Fletcher, R. A new approach to variable metric algorithms. *The Computer Journal 13*, 3 (Jan. 1970), 317–322.

[15] Fletcher, R. *Practical Methods of Optimization*. John Wiley & Sons, Ltd, 2000, ch. Newton-Like Methods, pp. 44–79.

[16] Fletcher, R., and Powell, M. J. D. A Rapidly Convergent Descent Method for Minimization. *The Computer Journal 6*, 2 (Aug. 1963), 163–168.

[17] Geyer, T. *Model Predictive Control of High Power Converters and Industrial Drives*. John Wiley & Sons, Ltd, 2016, ch. Introduction, pp. 1–28.

[18] Goldfarb, D. A Family of Variable-Metric Methods Derived by Variational Means. *Mathematics of Computation 24*, 109 (1970), 23–26.

[19] Goldfeld, S. M., Quandt, R. E., and Trotter, H. F. Maximization by Quadratic Hill-Climbing. *Econometrica 34*, 3 (1966), 541–551.

[20] Goldstein, A. A. On Steepest Descent. *Journal of the Society for Industrial and Applied Mathematics Series A Control 3*, 1 (1965), 147–151.

[21] Goldstein, A. A., and Price, J. F. On Descent from Local Minima. *Mathematics of Computation 25*, 115 (1971), 569–574.

[22] Grüne, L., and Pannek, J. *Nonlinear Model Predictive Control: Theory and Algorithms*. Springer London, Jan. 2011.

[23] Halton, J. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numerische Mathematik 2* (1960), 84–90.

[24] Haupt, R. L., and Haupt, S. E. *Practical Genetic Algorithms*. John Wiley & Sons, Ltd, 2003, ch. Appendix I: Test Functions, pp. 205–209.

[25] Hestenes, M. R. Multiplier and gradient methods. *Journal of Optimization Theory and Applications 4* (1969), 303–320.

[26] Ji, W. e. a. Electromagnetic Design of High-Power and High-Speed Permanent Magnet Synchronous Motor Considering Loss Characteristics. *Energies 14* (06 2021), 3622.

[27] Karmanov, V. G. *Mathematical programming*. Mir Publishers, 1989.

[28] Khalilov, M., and Timoveev, A. Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU. *Journal of Physics: Conference Series 1740*, 1 (Jan. 2021), 012056.

[29] Khan, S., and Guivant, J. Fast nonlinear model predictive planner and control for an unmanned ground vehicle in the presence of disturbances and dynamic obstacles. *Scientific Reports 12* (July 2022), 12135.

[30] Kocis, L., and Whiten, W. J. Computational investigations of low-discrepancy sequences. *ACM Trans. Math. Softw. 23*, 2 (June 1997), 266–294.

[31] Kozubík, M. Aplikace nelineárního prediktivního řízení pro pohon se synchronním motorem [NMPC Application for PMSM Drive Control]. Master's thesis, Brno University of Technology, 2018.

[32] Kozubík, M., Veselý, L., Aufderheide, E., and Václavek, P. Parallel Computing Utilization in Nonlinear Model Predictive Control of Permanent Magnet Synchronous Motor. *IEEE Access 12* (2024), 128187–128200.

[33] Kuhn, H. W., and Tucker, A. W. Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability, 1950* (1951), University of California Press, pp. 481–492.

[34] Li, X., and Shih, P.-C. An Early Performance Comparison of CUDA and OpenACC. *MATEC Web of Conferences 208* (Jan. 2018), 05002.

[35] Lin, F.-J., Liao, Y.-H., Lin, J.-R., and Lin, W.-T. Interior Permanent Magnet Synchronous Motor Drive System with Machine Learning-Based Maximum Torque per Ampere and Flux-Weakening Control. *Energies 14*, 2 (2021).

[36] Liu, D. C., and Nocedal, J. On the limited memory BFGS method for large scale optimization. *Mathematical Programming 45* (1989), 503–528.

[37] Lu, Q. e. a. Targeting Posture Control With Dynamic Obstacle Avoidance of Constrained Uncertain Wheeled Mobile Robots Including Unknown Skidding

and Slipping. *IEEE Transactions on Systems, Man, and Cybernetics: Systems 51*, 11 (2021), 6650–6659.

[38] Ma, H., Gao, Y., Yang, Y., and Xu, S. Improved Nonlinear Model Predictive Control Based Fast Trajectory Tracking for a Quadrotor Unmanned Aerial Vehicle. *Drones 8*, 8 (2024).

[39] Mas-Colell, A., Whinston, M. D., and Green, J. R. *Microeconomic Theory*. Oxford University Press, Oxford, 1995. Chapter 16: Equilibrium and Its Basic Welfare Properties, pp. 547–554.

[40] Mayne, D. Nonlinear Model Predictive Control: Challenges and Opportunities. In *Nonlinear Model Predictive Control* (Basel, 2000), F. Allgöwer and A. Zheng, Eds., Birkhäuser Basel, pp. 23–44.

[41] Mishra, S. K. Some New Test Functions for Global Optimization and Performance of Repulsive Particle Swarm Method. *University Library of Munich, Germany, MPRA Paper* (Aug. 2006).

[42] Morari, M., and Lee, J. H. Model predictive control: past, present and future. *Computers & Chemical Engineering 23* (1999), 667–682.

[43] Muller, J.-M. e. a. *Handbook of Floating-Point Arithmetic*. Birkhäuser, Jan. 2010.

[44] Mynář, Z., Veselý, L., and Václavek, P. PMSM Model Predictive Control With Field-Weakening Implementation. *IEEE Transactions on Industrial Electronics 63*, 8 (2016), 5156–5166.

[45] Nemirovski, A., and Ben-Tal, A. Lecture notes in Optimization III, 2023. Chapter Inexact Line Search. Georgia Institute of Technology, Technion – Israel Institute of Technology.

[46] Nocedal, J., and Wright, S. *Numerical Optimization*, 2 ed. Springer Series in Operations Research and Financial Engineering. Springer, New York, NY, July 2006.

[47] Nvidia. GeForce GTX 1050. `https://web.archive.org/web/20161222152032/https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1050/`. Accessed: 2025/04/04.

[48] Nvidia. GeForce RTX 4060. `https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4060-4060ti/`. Accessed: 2025/04/04.

[49] Nvidia. Your GPU Compute Capability. https://developer.nvidia.com/cuda-gpus. Accessed: 2025/04/04.

[50] Nvidia. *PGI Compiler Tools: Profiler OpenACC Tutorial*, 2019.

[51] Nvidia. *CUDA C++ Programming Guide*, release 12.8 ed., Feb. 2025.

[52] OpenACC. OpenACC Programming and Best Practices Guide. https://openacc-best-practices-guide.readthedocs.io/en/latest/. Accessed: 2025/04/04.

[53] Patne, V., Ingole, D., and Sonawane, D. Towards fast nonlinear model predictive control for embedded applications. *IFAC-PapersOnLine 55* (01 2022), 304–309.

[54] Patne, V. e. a. Ultra-Fast Nonlinear Model Predictive Control for Motion Control of Autonomous Light Motor Vehicles. *World Electric Vehicle Journal 15*, 7 (2024).

[55] Qin, S. J., and Badgwell, T. A. A survey of industrial model predictive control technology. *Control Engineering Practice 11*, 7 (2003), 733–764.

[56] Richalet, J., Rault, A., Testud, J., and Papon, J. Model predictive heuristic control: Applications to industrial processes. *Automatica 14*, 5 (1978), 413–428.

[57] Rosenbrock, H. H. An Automatic Method for Finding the Greatest or Least Value of a Function. *The Computer Journal 3*, 3 (Jan. 1960), 175–184.

[58] Schwefel, H.-P. *Numerische Optimierung von Computermodellen mittels der Evolutionsstrategie [Computer-model Numerical Optimization through Evolutionary Algorithms]*, vol. 26. Birkhauser, Jan. 1977.

[59] Scokaert, P., Mayne, D., and Rawlings, J. Suboptimal model predictive control (feasibility implies stability). *IEEE Transactions on Automatic Control 44*, 3 (1999), 648–654.

[60] Sedlář, J. Efficient Optimization through Normalized Variable Scaling and Selective Term Pruning. In *Proceedings of the 31st Student EEICT 2025* (2025), Brno University of Technology, Faculty of Electrical Engineering and Communication. To appear.

[61] Shan, B., and Araya-Polo, M. Evaluation of Programming Models and Performance for Stencil Computation on Current GPU Architectures, 2024.

[62] Shanno, D. F. Conditioning of quasi-newton methods for function minimization. *Mathematics of Computation 24*, 111 (1970), 647–656.

[63] Sorensen, D. C. Newton's Method with a Model Trust Region Modification. *SIAM Journal on Numerical Analysis 19*, 2 (1982), 409–426.

[64] Törn, A., and Žilinskas, A. *Global Optimization*, 1989 ed. Lecture notes in computer science. Springer, Berlin, Germany, Feb. 1989.

[65] Tran-Dinh, Q., and van Dijk, M. Chapter 1 - Gradient descent-type methods: Background and simple unified convergence analysis. In *Federated Learning*, L. M. Nguyen, T. N. Hoang, and P.-Y. Chen, Eds. Academic Press, 2024, pp. 3–28.

[66] Velarde-Gomez, S., and Giraldo, E. Nonlinear Control of a Permanent Magnet Synchronous Motor Based on State Space Neural Network Model Identification and State Estimation by Using a Robust Unscented Kalman Filter. *Eng 6* (02 2025), 30.

[67] Wolfe, P. Convergence Conditions for Ascent Methods. *SIAM Review 11*, 2 (1969), 226–235.

[68] Wright, S. J. Continuous Optimization (Nonlinear and Linear Programming). In *The Princeton Companion to Applied Mathematics* (2012).

[69] Wright, S. J., and Orban, D. Properties of the Log-Barrier Function on Degenerate Nonlinear Programs. *Mathematics of Operations Research 27*, 3 (2002), 585–613.

[70] Xiong, S., Pan, J., and Yang, Y. Robust Decoupling Vector Control of Interior Permanent Magnet Synchronous Motor Used in Electric Vehicles with Reduced Parameter Mismatch Impacts. *Sustainability 14*, 19 (2022).

[71] Zheng, A., and Zhang, A. W. *Non-linear Predictive Control: Theory and practice*. Control, Robotics and Sensors. Institution of Engineering and Technology, Oct. 2001, ch. Computationally efficient non linear predictive control algorithm for control of constrained nonlinear systems, pp. 173–188.

[72] Zorich, V. A. *Mathematical analysis I*, 2 ed. Universitext. Springer, Berlin, Germany, Mar. 2016.

A Contents of the electronic attachment

The electronic attachment to this thesis is organized as follows: both sequential and parallel implementations of the NMPC algorithm are stored in the folder named `NMPC_Implementation` within their respective subfolders. For the parallel implementation, `BFGS_Precise.c` represents the MATLAB mex file, `BFGS_NMPC4.h` the common header file and `BFGS_openACC4` files are the actual implementation of the parallelized NMPC to be compiled as a shared library. In the sequential-implementation folder, the `BFGS_Precise4.cpp` is the .mex file containing also the NMPC implementation. Simulation files are located in the `Simulation` directory with Simulink files for serial and parallel NMPC stored again in their respective subfolders. Vector control has another dedicated folder in this directory, where `MTPA_calc.m` file is used to calculate the ideal MTPA curve of the IPMSM. Files common to all simulation scenarios, namely `refSig.mat` representing the speed reference signal and `PMSM_init.m` initializing the motor parameters, are also stored in the simulations-related folder. Finally, the term-pruning utility is stored in the dedicated `pruningApp` folder.

```
/ ..... Root
├── NMPC_Implementation ..... Algorithm Implementation
│   ├── Parallel
│   │   ├── BFGS_NMPC4.h
│   │   ├── BFGS_openACC4.h
│   │   ├── BFGS_openACC4.c
│   │   ├── BFGS_Precise.c
│   │   └── make_custom_solver_sfunction.m
│   └── Sequential
│       ├── BFGS_Precise4.cpp
│       ├── BFGS_NMPC4.h
│       └── make_custom_solver_sfunction.m
├── Simulation ..... Simulation files
│   ├── Parallel
│   │   └── Simulation.slx
│   ├── Sequential
│   │   └── Simulation.mdl
│   ├── VC
│   │   ├── IPMSM_FOC.slx
│   │   └── MTPA_calc.m
│   ├── refSig.mat
│   └── PMSM_init.m
└── pruningApp ..... Term pruning utility
    └── app_revX.mlapp
```