Hochschule Bielefeld
University of Applied Sciences

Institute for Data Science Solutions of HSBI

Adaptively Refined Mesh for Collocation-Based Dynamic Optimization

Schriftenreihe des Institute for Data Science Solutions

Nr. 1/ 2025

DOI: 10.60802/sidas.2025.1
ISSN: 2943-3509

Hochschule Bielefeld
University of Applied Sciences and Arts
Faculty of Engineering and Mathematics
Optimierung und Simulation

Master's Thesis

Adaptively Refined Mesh for Collocation-Based Dynamic Optimization

Linus Langenkamp
December 10th, 2024

Supervisors:
Prof. Dr., Dipl.-Math. Bernhard Bachmann
M. Sc. Karim Abdelhak

Abstract

This thesis deals with the topic of efficient numerical solutions to dynamic optimization problems using a direct collocation approach with adaptive mesh refinement. A novel h-method is proposed and implemented in a newly developed dynamic optimization framework that utilizes direct collocation with flipped Legendre-Gauss-Radau points. The mesh refinement algorithm aims to uniformize control trajectories by successive bisection based on slope and curvature analysis in each interval. For smooth problems and under suitable convergence, a termination guarantee is established. The algorithm has significant advantages over traditional direct collocation approaches without mesh refinement in terms of accuracy and computation time. The dynamic optimization framework allows for accessible and expressive modeling and proves to be very effective when applied to a variety of example problems, including academic and real-world physical applications. However, the proposed h-method offers room for improvement since no direct error estimates are incorporated, and it is shown that the problem class studied in this thesis allows for an extension to the broad and advanced class of pseudospectral mesh refinement algorithms. Moreover, it is demonstrated that embedding the method in a modern modeling and simulation environment would greatly benefit in terms of performance, symbolic handling, and for error analysis as well as validation of optimal solutions.

Kurzfassung

Diese Arbeit befasst sich mit der effizienten, numerischen Lösung dynamischer Optimierungsprobleme auf Basis direkter Kollokation und unter Verwendung von Algorithmen zur adaptiven Meshverfeinerung. Eine neuartige *h*-Methode, eingebettet in ein neu entwickeltes Framework für dynamische Optimierung, wird vorgestellt. Das Framework basiert auf direkten Kollokationsmethoden mit gespiegelten Legendre-Gauss-Radau Punkten. Das adaptive Meshverfeinerungsverfahren zielt darauf ab, die Trajektorien der Steuervariablen auf Grundlage einer Steigungs- und Krümmungsanalyse in jedem Intervall durch sukzessive Bisektion zu homogenisieren. Unter Annahme der Konvergenz des Kollokationsansatzes für glatte Probleme kann die Terminierung des Verfahrens gezeigt werden. Der Algorithmus weist signifikante Vorteile im Bezug auf Laufzeit und Genauigkeit gegenüber traditionellen direkten Kollokationsmethoden auf. Zusätzlich ermöglicht das Framework eine zugängliche und ausdrucksstarke Modellierung und erweist sich bei einer Vielzahl von akademischen und realen Beispielproblemen als äußerst effektiv. Die vorgeschlagene *h*-Methode bietet jedoch Raum für Verbesserungen, da keine direkten Fehlerabschätzungen verwendet werden. Darüber hinaus wird gezeigt, dass die in dieser Arbeit untersuchte Problemklasse eine Erweiterung der Methode zu einer pseudospektralen Meshverfeinerungsmethode ermöglicht. Zusätzlich birgt die Integration des Algorithmus in eine moderne Modellierungs- und Simulationssoftware erhebliche Vorteile, da dies eine gesteigerte Performance, bessere symbolische Verarbeitung und eine Fehleranalyse sowie Validierung der Optimallösung bieten kann.

Contents

List of Acronyms

AD Automatic Differentiation

BVP Boundary Value Problem

CQ Constrained Qualification

CSE Common Subexpressions

DAE system Differential-Algebraic System of Equations

dGDOP discretized General Dynamic Optimization Problem

fLGR flipped Legendre-Gauss-Radau

GDOP General Dynamic Optimization Problem

HBVP Hamiltonian Boundary Value Problem

IP Interior-Point Method

IVP Initial Value Problem

KKT Karush-Kuhn-Tucker

L2BN L2-Boundary-Norm

LG Legendre-Gauss

LGL Legendre-Gauss-Lobatto

LGR Legendre-Gauss-Radau

LICQ Linear Independence Constrained Qualification

NLP Nonlinear Optimization Problem

NOCP Nonlinear Optimal Control Problem

ODE Ordinary Differential Equation

PMP Pontryagin's Maximum Principle

SQP Sequential Quadratic Progamming

List of Figures

List of Tables

1 Introduction

Dynamic optimization problems arise in many practical applications and industrial areas, such as aerospace and trajectory optimization, chemical and biological engineering, economics, medicine, or robotics. The goal of these optimization problems is to find an optimal control trajectory over a given time horizon that maximizes or minimizes a particular quantity, such as energy, distance, velocity, time, or fuel consumption, and satisfies given constraints. The particular challenge of optimal control problems is that the constraints can contain differential equations, which are often highly nonlinear. Moreover, the optimization is performed over the infinite dimensional space of functions, resulting in extremely hard optimization problems. Nevertheless, there are analytical results providing necessary conditions that optimal solutions must satisfy. The most prominent and well celebrated result is *Pontryagin's maximum principle*, which reduces the infinite dimensional problem to a boundary value problem as well as to the maximization of a Hamiltonian. Under certain convexity conditions the theorem becomes sufficient. However, the arising subproblem is usually very difficult to solve or poorly conditioned, which shows the necessity of numerical methods.[24][16]
One of the most popular and efficient numerical methods is *direct collocation*. In direct collocation methods, both the state and control variables are replaced by discrete approximations at given nodes on the time horizon. These approximations are used to transform the continuous optimal control problem into a large-scale but finite-dimensional nonlinear optimization problem (NLP) that can be solved with conventional NLP solvers such as *Ipopt*[1] or *SNOPT*[25][26]. For this transcription, a special class of Runge-Kutta methods called *collocation schemes* is used to discretize the dynamics of the system. A collocation scheme approximates the state variables on each interval by a polynomial that satisfies the differential equation at the chosen nodes. The nodes are usually roots of orthogonal polynomials, e.g. *Legendre-Gauss (LG)*, *Legendre-Gauss-Radau (LGR)* or *Legendre-Gauss-Lobatto (LGL)* points, since these exhibit excellent stability and high order.[24][50][56]
Direct collocation methods often perform *mesh refinement* algorithms to accurately capture the smooth and non-smooth behavior of the optimal solution and achieve better performance in terms of computation time and error. The principal classes of mesh refinement algorithms are h-, p- and hp-methods. A h-method[50][53][51][52] splits the time horizon into many subintervals and employs rather low fixed degree polynomials on each interval. Convergence is then achieved by increasing the number of intervals or reducing the interval length based on given criteria. h-methods are very flexible and stable, but converge rather slowly compared to other methods for smooth problems. However, h-methods effectively capture non-smooth behavior and are thus well suited for general purpose applications. p-methods commonly use a single interval and a very high degree global polynomial. For an increasing number of collocation nodes and smooth problems, these methods achieve *exponential* convergence. For non-smooth problems, however, these methods produce large error terms and are impractical. Modern hp- or ph-adaptive methods [30][56][46][49][48] aim to combine both approaches, utilizing exponential convergence and the ability to capture non-smooth behavior by varying the number of intervals and the polynomial degree. They show promising results and are implemented in state-of-the-art software such as the proprietary *GPOPS II*[30]. Nevertheless, these hybrid methods also have difficulty finding the placements of switches and kinks effectively and often produce comparably large meshes with more collocation nodes.[50][56]

1.1 Aim of this Thesis

This thesis aims to construct a general-purpose model-based framework for direct collocation-based dynamic optimization using *flipped Legendre-Gauss-Radau (fLGR)* points. Note that this approach is equivalent to the accurate and stable *Radau IIA* Runge-Kutta methods. To solve the arising NLPs, the C++ interface of the well-known nonlinear optimizer Ipopt is used. A central focus of this thesis is the development and implementation of a novel *h*-method mesh refinement algorithm for local collocation. The implementation is intended to be modular and extensible, allowing extensions to broader problem classes and future integrations in third-party tools. Furthermore, the framework is aimed to be very efficient and accurate, producing low error terms. For this reason, case studies on relevant academic and real-world physical dynamic optimization problems will be carried out to evaluate the quality and limitations of the proposed framework.

1.2 Thesis Outline

The present work is divided into 9 chapters and aims to give a comprehensive overview of all key elements that are important for the development of direct collocation methods in dynamic optimization. Chapter 2 motivates and introduces the general problem class considered in this thesis. After this, numerical methods such as Lagrange interpolation and collocation methods, which are necessary to transform the general problem class into a NLP, are described in Chapter 3. Furthermore, in Chapter 4 basic concepts of nonlinear optimization and the important interior-point method Ipopt are presented. Chapter 5 deals with the transcription of the continuous problem into a large-scale NLP using the previously obtained formulas and methods. In Chapter 6, a detailed description and derivation of the proposed mesh refinement algorithm is given. In addition, important attributes of the method are shown. In the following Chapter 7 the implementational details and special features of the framework are presented. The performance of the proposed *h*-method and framework is then analyzed and evaluated for a series of example problems in Chapter 8. The final Chapter 9 gives an overview of the properties, limitations, and potential extensions of the framework.

2 Dynamic Optimization

2.1 Introductory Examples

Dynamic optimization is applied in many different fields, such as robotics, economics, aerospace engineering or the optimization of chemical and biological processes. To motivate the general problem formulation, which enables the modeling of problems in all these fields, two introductory examples are considered. These examples provide an introduction to the subject, show applications of optimal control theory, and most importantly, offer insight into the structure of dynamic optimization problems.

2.1.1 Maximum Travel Distance

The first example is a very simple optimization of the 1-dimensional trajectory of a car. It is adapted from the *OpenModelica User's Guide* [11] and can be found in the section *Optimization with OpenModelica*.

Model 2.1 (Maximum Travel Distance)**.**

$$
\begin{aligned}
&\max_{F(t)} x(t_f) \\
&\quad s.t. \\
&\begin{pmatrix} \dot{x}(t) \\ \dot{v}(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ a(t) \end{pmatrix} \quad \forall t \in [t_0, t_f], \begin{pmatrix} x(t_0) \\ v(t_0) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\
&|F(t) \cdot v(t)| \leq 30 \quad \forall t \in [t_0, t_f] \\
&|F(t)| \leq 10 \quad \forall t \in [t_0, t_f] \\
&F(t) = m \cdot a(t) \quad \forall t \in [t_0, t_f] \\
&v(t_f) = 0 \\
&m = 0.75
\end{aligned}
$$

The goal of this model is to maximize the distance traveled by a car over a fixed time horizon $t \in [t_0, t_f]$ by finding the optimal force trajectory $F(t)$ applied to the wheels of the car. The position of the car at time t is given by $x(t)$, the velocity by $v(t)$ and the acceleration by $a(t)$. Furthermore, the position can be written as the 2nd order differential equation $\ddot{x}(t) = a(t)$ or equivalently as $\dot{x}(t) = v(t)$ and $\dot{v}(t) = a(t)$. Note that all units are omitted to keep the model as simple as possible. Initially the car is at a standstill, i.e. $x(t_0) = 0, v(t_0) = 0$, and it is also desired to reach a final state where the car is holding, i.e. $v(t_f) = 0$. The force can be translated into an acceleration using Newton's 2nd law $F(t) = m \cdot a(t)$ with a fixed mass of $m = 0.75$. In addition, so-called *path constraints* must be satisfied at each time $t \in [t_0, t_f]$. The power $F(t) \cdot v(t)$ cannot exceed 30 in absolute value at any time, thus the constraint $|F(t) \cdot v(t)| \leq 30$ is added to the model. Besides the power limitation, there is also a restriction on the maximum possible torque, which is rewritten in terms of the applied force and given by $|F(t)| \leq 10$. This problem is a classic example of a *Bang-Bang* optimal control, where the control is maximal for half of the time horizon and then switches to the minimal value for the remaining time.[11]

2.1.2 Oil Shale Pyrolysis

The second example is the optimization of an *oil shale pyrolysis*. This process involves several complex temperature-dependent chemical reactions and has been studied in the literature.[12] By heating kerogen, the organic material in oil shale, pyrolytic bitumen is formed. The two materials also react with each other, resulting in the hydrocarbon waste products oil or gas and the carbonaceous residue of the oil shale. The reactions that take place are given by

$$\begin{aligned} x_1 &\xrightarrow{k_1} x_2 \\ x_2 &\xrightarrow{k_2} x_3 \\ x_1 + x_2 &\xrightarrow{k_3} x_2 + x_2 \\ x_1 + x_2 &\xrightarrow{k_4} x_3 + x_2 \\ x_1 + x_2 &\xrightarrow{k_5} x_4 + x_2 \end{aligned}$$

where x_1 is the amount of kerogen, x_2 bitumen, x_3 oil or gas and x_4 carbonaceous residue.

Model 2.2 (Oil Shale Pyrolysis)**.**

$$\begin{aligned} &\max_{T(t)} x_2(t_f) \\ &\quad s.t. \\ &\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{pmatrix} = \begin{pmatrix} -k_1x_1 - (k_3 + k_4 + k_5)x_1x_2 \\ k_1x_1 - k_2x_2 + k_3x_1x_2 \\ k_2x_2 + k_4x_1x_2 \\ k_5x_1x_2 \end{pmatrix} \quad \forall t \in [t_0, t_f], \begin{pmatrix} x_1(t_0) \\ x_2(t_0) \\ x_3(t_0) \\ x_4(t_0) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \\ &\begin{pmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \end{pmatrix} = \begin{pmatrix} \exp\left(8.86 - \frac{10215}{T(t)}\right) \\ \exp\left(24.25 - \frac{18820}{T(t)}\right) \\ \exp\left(23.67 - \frac{17009}{T(t)}\right) \\ \exp\left(18.75 - \frac{14191}{T(t)}\right) \\ \exp\left(20.70 - \frac{15600}{T(t)}\right) \end{pmatrix} \\ &698.15 \leq T(t) \leq 748.15 \end{aligned}$$

The objective of an oil shale pyrolysis is to maximize the amount of pyrolytic bitumen at the final time t_f. In order to achieve this, an optimal temperature control $T(t)$ has to be calculated. As before, all units are omitted. Due to physical limitations, the temperature is bounded by $698.15 \leq T(t) \leq 748.15$. Furthermore, the amounts of the materials are expressed on a relative scale. Since only kerogen is present at the initial time t_0, the initial states are $\boldsymbol{x}(t_0) = \boldsymbol{e}_1$, where $\boldsymbol{e}_1$ is the first unit vector. The dynamics are given by a system of nonlinear differential equations that correspond to the aforementioned chemical reactions. The rate of change of each reaction depends on a coefficient k_i, $i = 1, \ldots, 5$, which in itself depends on the temperature $T(t)$.[12]

2.2 Problem Formulation and Classification

2.2.1 Model-Based Dynamic Optimization

This thesis aims to solve problems belonging to the class of model-based dynamic optimization. The modeling and simulation of dynamic systems is closely related to the optimization of systems. Moreover, simulation is a subset of dynamic optimization where there are neither objectives nor constraints. This thesis demonstrates how the three elements model, constraints and objective, can be considered as the components of model-based dynamic optimization and, in combination, lead to the definition of the General Dynamic Optimization Problem (GDOP).

2.2.1.1 Model Component The *model component* refers to models in simulation environments. Simulation software is used to efficiently study the behavior of real-world systems. These models are typically described as a Differential-Algebraic System of Equations (DAE system).[13][15]

Definition 2.1 (Differential-Algebraic System of Equations)**.** *Given a time horizon $I = [t_0, t_f]$, a time $t \in I$, which evolves over the entire interval I, state variables $\boldsymbol{x}(t) : I \to \mathbb{R}^{n_x}$, differentiated state variables $\dot{\boldsymbol{x}}(t) := \frac{\mathrm{d}\boldsymbol{x}(t)}{\mathrm{d}t}$ with $\dot{\boldsymbol{x}}(t) : I \to \mathbb{R}^{n_x}$, algebraic variables $\boldsymbol{y}(t) : I \to \mathbb{R}^{n_y}$, input variables $\boldsymbol{u}(t) : I \to \mathbb{R}^{n_u}$ and time-invariant input parameters $\boldsymbol{p} \in \mathbb{R}^{n_p}$. Furthermore, let $\boldsymbol{x}_0 \in \mathbb{R}^{n_x}$ denote the initial values for the state $\boldsymbol{x}$ and $\boldsymbol{F} : \mathbb{R}^{2n_x+n_y+n_u+n_p} \times I \to \mathbb{R}^{n_x+n_y}$ be a given function, the system*

$$\begin{aligned} 0 &= \boldsymbol{F}(\dot{\boldsymbol{x}}(t), \boldsymbol{x}(t), \boldsymbol{y}(t), \boldsymbol{u}(t), \boldsymbol{p}, t) \; \forall t \in [t_0, t_f] \\ \boldsymbol{x}(t_0) &= \boldsymbol{x}_0 \end{aligned} \tag{1}$$

is called a differential-algebraic system of equations (DAE system).[16]

The variables in Definition 2.1 can be divided into two categories: $\boldsymbol{u}(t)$ and $\boldsymbol{p}$ are known variables, while $\boldsymbol{x}(t)$ and $\boldsymbol{y}(t)$ are unknowns, which means that they have to be calculated in the solution process. Furthermore, $\boldsymbol{x}(t)$ is determined by an Ordinary Differential Equation (ODE) and its initial value, while $\boldsymbol{y}(t)$ is determined by an algebraic equation. Consequently, the implicit DAE system can be transformed into a semi-explicit DAE system of the form

$$\begin{aligned} \dot{\boldsymbol{x}}(t) &= \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{y}(t), \boldsymbol{u}(t), \boldsymbol{p}, t), \; \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \\ 0 &= \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{y}(t), \boldsymbol{u}(t), \boldsymbol{p}, t). \end{aligned} \tag{2}$$

The transformation into a semi-explicit DAE system can be done by several algorithms, e.g. the so-called *BLT transformation.*[14] This form is crucial for many numerical algorithms, such as integration schemes, since they often require an Initial Value Problem (IVP). In general, this system is not given in closed form, but by iterative processes. In this thesis, however, a closed form is assumed. The semi-explicit DAE system in closed form embodies the *model component* mentioned above.[17][14][15]

When the system is transferred from simulation to optimization, an important difference applies. The input or control variables $\boldsymbol{u}(t)$ and the time-invariant input parameters $\boldsymbol{p}$ are not given, but rather the variables to be optimized, and thus are found in the optimization. This process can be observed in the *oil shale pyrolysis* (Model 2.2), where the temperature control $T(t)$ is the function to be searched for. When transferred to dynamic optimization, the ODEs $\dot{\boldsymbol{x}} = \boldsymbol{f}(\cdot)$ become so-called *dynamic constraints*. For example, in the

maximum travel distance (Model 2.1), the equations $\dot{x} = v$ and $\dot{v} = a$ form the dynamic constraints of the model. In addition, the algebraic constraints $0 = \boldsymbol{g}(\cdot)$ turn into so-called *path constraints*. An example of a path constraint is Newton's 2nd law in Model 2.1. This can be written as $0 = F(t) - m \cdot a(t)$. Both types of constraints must hold at each time $t \in [t_0, t_f]$.

2.2.1.2 Constraint Component In addition to the model itself, it must also be possible to model further constraints that describe physical restrictions and limitations as well as enforce desired behavior. The corresponding component is called *constraint component* in this thesis. There are three types of constraints that belong to this category. The first type of constraint is another *path constraint* that must also hold at all times $t \in [t_0, t_f]$, e.g. the power limitation $|F(t) \cdot v(t)| \leq 30$ in Model 2.1. In general, this constraint can be expressed as

$$\boldsymbol{h}^L \leq \boldsymbol{h}(\boldsymbol{x}(t), \boldsymbol{y}(t), \boldsymbol{u}(t), \boldsymbol{p}, t) \leq \boldsymbol{h}^U \tag{3}$$

with given lower and upper bounds, which can be infinite. The second constraint must hold exclusively at the final time t_f and can be written as

$$\boldsymbol{r}^L \leq \boldsymbol{r}(\boldsymbol{x}(t_f), \boldsymbol{y}(t_f), \boldsymbol{u}(t_f), \boldsymbol{p}, t_f) \leq \boldsymbol{r}^U. \tag{4}$$

This is referred to as a *final constraint* and is used in Model 2.1 to force the final velocity to be zero, i.e. $v(t_f) = 0$. The third type is a special case of the other two constraints. It is a constraint that contains only parameters and is therefore time-invariant. The general form of a *parametric constraint* is given by

$$\boldsymbol{a}^L \leq \boldsymbol{a}(\boldsymbol{p}) \leq \boldsymbol{a}^U. \tag{5}$$

2.2.1.3 Objective Component The last component is called the *objective component* and is used to define a quantity to be minimized. Obviously, it is possible to express maximization problems by multiplying the objective by -1. This component consists of a linear combination of the *Mayer term* and the *Lagrange term*. The complete objective is given by

$$\min_{\boldsymbol{u}(t), \boldsymbol{p}} \underbrace{M(\boldsymbol{x}(t_f), \boldsymbol{y}(t_f), \boldsymbol{u}(t_f), \boldsymbol{p}, t_f)}_{\textit{Mayer term}} + \underbrace{\int_{t_0}^{t_f} L(\boldsymbol{x}(t), \boldsymbol{y}(t), \boldsymbol{u}(t), \boldsymbol{p}, t)\, \mathrm{d}t}_{\textit{Lagrange term}} \tag{6}$$

Note that the *Mayer term* penalizes the final configuration of the system, whereas the *Lagrange term* penalizes an accumulated quantity, such as the integral of power over time. An example of the *Mayer term* is the maximization of pyrolytic bitumen $\max x_2(t_f)$ or equivalently $\min -x_2(t_f)$ in Model 2.2.[13][15][19]

2.2.2 Problem Definition

Based on the different components of model-based dynamic optimization illustrated in Figure 1, the *General Dynamic Optimization Problem (GDOP)* can be defined. The GDOP has the structure of a generic model-based dynamic optimization problem with fixed endpoint. However, it should be noted that free endpoint problems, which are central to models where the minimization of the final time is desired, or problems with

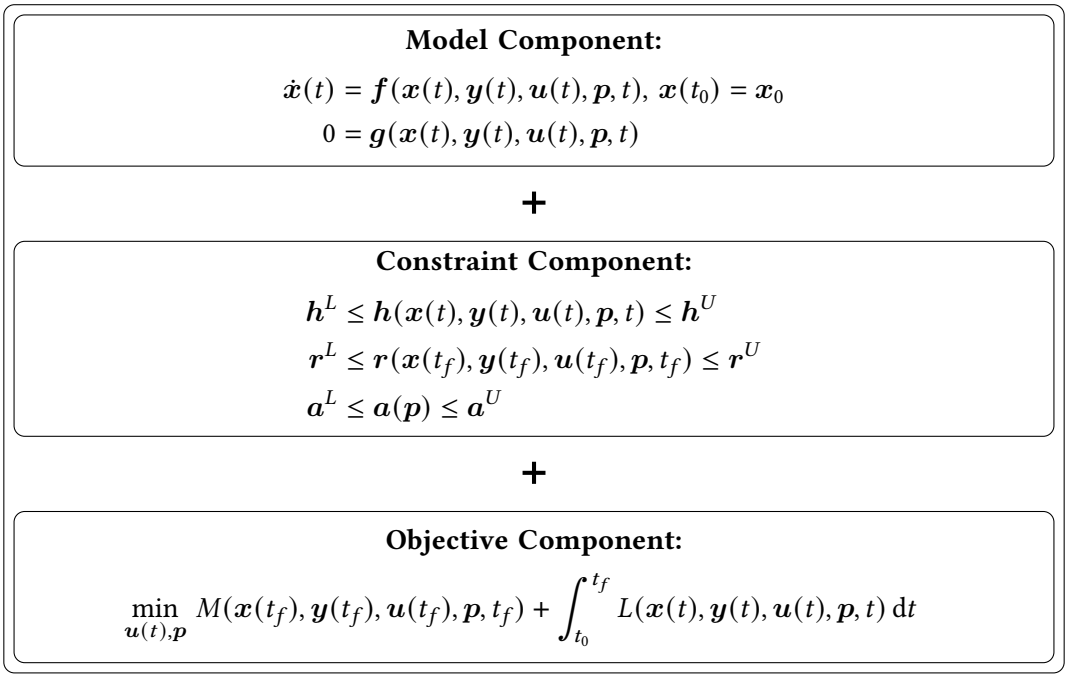


Figure 1: Components of model-based dynamic optimization (adapted from [13])

initial constraints similar to the final constraints could also be considered based on the GDOP.[19] These are beyond the scope of this thesis, but are possible extensions. Furthermore, the main focus is placed on the following fixed endpoint problem formulation.

Definition 2.2 (General Dynamic Optimization Problem (GDOP))**.** *Given a fixed time horizon $I = [t_0, t_f]$, a variable time $t \in [t_0, t_f]$, states $\boldsymbol{x}(t) : I \to \mathbb{R}^{n_x}$ with initial values $\boldsymbol{x}_0 \in \mathbb{R}^{n_x}$, inputs $\boldsymbol{u}(t) : I \to \mathbb{R}^{n_u}$ and parameters $\boldsymbol{p} \in \mathbb{R}^{n_p}$. Let $d := n_x + n_u + n_p$ and given twice continuously differentiable functions $M : \mathbb{R}^d \times I \to \mathbb{R}$, $L : \mathbb{R}^d \times I \to \mathbb{R}$, $\boldsymbol{f} : \mathbb{R}^d \times I \to \mathbb{R}^{n_x}$, $\boldsymbol{g} : \mathbb{R}^d \times I \to \mathbb{R}^{n_g}$, $\boldsymbol{r} : \mathbb{R}^d \times I \to \mathbb{R}^{n_r}$ and $\boldsymbol{a} : \mathbb{R}^{n_p} \to \mathbb{R}^{n_a}$ as well as bounds $\boldsymbol{g}^L, \boldsymbol{g}^U \in (\mathbb{R} \cup \{-\infty, \infty\})^{n_g}$, $\boldsymbol{r}^L, \boldsymbol{r}^U \in (\mathbb{R} \cup \{-\infty, \infty\})^{n_r}$, $\boldsymbol{a}^L, \boldsymbol{a}^U \in (\mathbb{R} \cup \{-\infty, \infty\})^{n_a}$, then*

$$
\begin{aligned}
&\min_{\boldsymbol{u}(t), \boldsymbol{p}} M(\boldsymbol{x}(t_f), \boldsymbol{u}(t_f), \boldsymbol{p}, t_f) + \int_{t_0}^{t_f} L(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t)\, \mathrm{d}t \\
&\quad s.t. \\
&\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t)\ \forall t \in [t_0, t_f] \\
&\boldsymbol{x}(t_0) = \boldsymbol{x}_0 \\
&\quad \boldsymbol{g}^L \leq \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t) \leq \boldsymbol{g}^U\ \forall t \in [t_0, t_f] \\
&\quad \boldsymbol{r}^L \leq \boldsymbol{r}(\boldsymbol{x}(t_f), \boldsymbol{u}(t_f), \boldsymbol{p}, t_f) \leq \boldsymbol{r}^U \\
&\quad \boldsymbol{a}^L \leq \boldsymbol{a}(\boldsymbol{p}) \leq \boldsymbol{a}^U
\end{aligned}
$$

is called General Dynamic Optimization Problem (GDOP) with Mayer term $M(\cdot)$, Lagrange term $\int_{t_0}^{t_f} L(\cdot)\, \mathrm{d}t$, dynamic constraints $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\cdot)$, path constraints $\boldsymbol{g}^L \leq \boldsymbol{g}(\cdot) \leq \boldsymbol{g}^U$, final constraints $\boldsymbol{r}^L \leq \boldsymbol{r}(\cdot) \leq \boldsymbol{r}^U$ and algebraic constraints $\boldsymbol{a}^L \leq \boldsymbol{a}(\cdot) \leq \boldsymbol{a}^U$.

Several simplifications are made in the definition of the GDOP (Definition 2.2). The two types of *path*

constraints are combined into a single vector. This is possible because the algebraic constraints $0 = \boldsymbol{g}(\cdot)$, which stem from the DAE system, can be interpreted as $0 \leq \boldsymbol{g}(\cdot) \leq 0$. The algebraic variables $\boldsymbol{y}(t)$ are also added to the input vector since they are determined by their respective, possibly implicit, algebraic equations anyway, e.g. $a(t) = \frac{F(t)}{m}$ in Model 2.1.
Many optimal control problems can be expressed as a GDOP. This is the case, since it is an extension of the *Nonlinear Optimal Control Problem (NOCP)*, which is implemented in the open source modeling and simulation environment *OpenModelica*.[9][17][20] A NOCP has the form

$$\begin{aligned} &\min_{\boldsymbol{u}(t)} M(\boldsymbol{x}(t_f), \boldsymbol{u}(t_f), t_f) + \int_{t_0}^{t_f} L(\boldsymbol{x}(t), \boldsymbol{u}(t), t)\, \mathrm{d}t \\ &\text{s.t.} \\ &\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t)\ \forall t \in [t_0, t_f] \\ &\boldsymbol{x}(t_0) = \boldsymbol{x}_0 \\ &\boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \leq 0\, \forall t \in [t_0, t_f] \\ &\boldsymbol{r}(\boldsymbol{x}(t_f)) = 0. \end{aligned} \tag{7}$$

It is clear that the GDOP is a generalization of the NOCP, since it provides support for parameter optimization as well as more general constraints. Consequently, the GDOP is an exemplary problem class for modeling and optimization of optimal control problems and dynamic systems based on DAE systems.[17][15][21]

2.3 Solution Methods for Dynamic Optimization Problems

2.3.1 Numerical Methods

Dynamic optimization problems such as the GDOP are notoriously hard to solve and, with the exception of simple problems, must be solved by numerical methods.

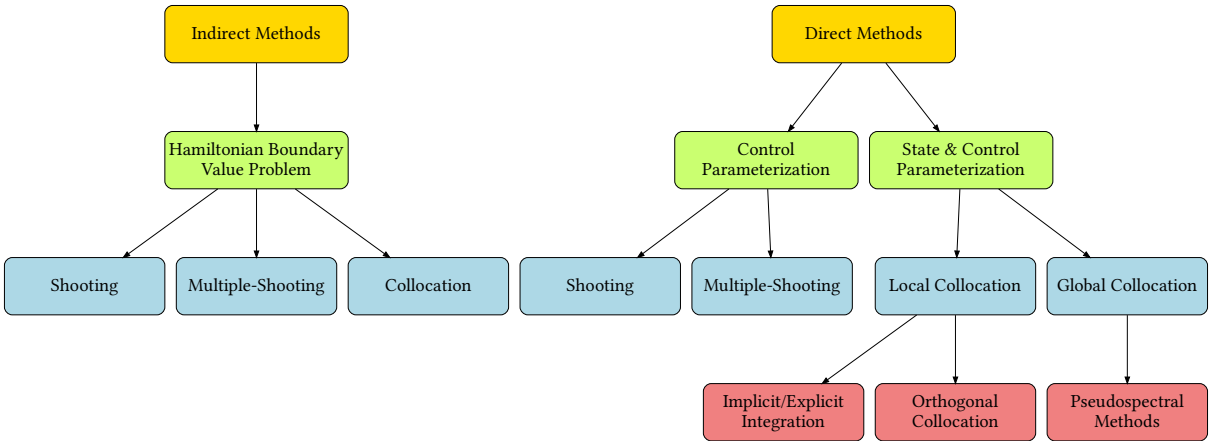


Figure 2: Overview of numerical methods in optimal control theory (adapted from [24])

This is due to the fact that the optimization is performed over the infinite-dimensional space of functions and the problems contain, often nonlinear, differential equations as constraints. Therefore, analytical

solutions are usually not obtainable. The numerical approaches fall into two categories: Indirect methods and direct methods. Both categories with their respective algorithms are displayed in Figure 2.[24]

2.3.1.1 Indirect Methods Indirect methods are usually based on the *calculus of variations*, which is an extension of regular calculus, where functions that depend on functions, also called *functionals*, are optimized. These types of methods use the first-order necessary conditions for extremal solutions to reduce the infinite-dimensional optimal control problem to a system of nonlinear equations. The first-order necessary conditions are usually written in terms of the augmented Hamiltonian $\mathcal{H}$. The Hamiltonian of the GDOP (Definition 2.2) is given by

$$\mathcal{H}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}, \boldsymbol{u}, t) = L + \boldsymbol{\lambda}^T \boldsymbol{f} - \boldsymbol{\mu}^T \boldsymbol{g}, \tag{8}$$

with the *costate* $\boldsymbol{\lambda}(t) : I \to \mathbb{R}^{n_x}$ and Lagrange multipliers $\boldsymbol{\mu}(t) : I \to \mathbb{R}^{n_g}$, which are used for the path constraints. The corresponding necessary conditions are the *Hamiltonian system*

$$\dot{\boldsymbol{x}} = \nabla_{\boldsymbol{\lambda}} \mathcal{H}, \; \dot{\boldsymbol{\lambda}} = -\nabla_{\boldsymbol{x}} \mathcal{H}, \tag{9}$$

the *minimum principle*

$$\boldsymbol{u}^* = \arg\min_{\boldsymbol{u} \in \mathcal{U}} \mathcal{H} \implies 0 = \nabla_{\boldsymbol{u}} \mathcal{H}, \tag{10}$$

where $\mathcal{U}$ denotes the set of all admissible controls, the path and final constraints as well as boundary conditions for $\mathcal{H}$ and $\boldsymbol{\lambda}$ at time t_0 and t_f.[24] The complete problem is called a *Hamiltonian Boundary Value Problem (HBVP)* and can occasionally be solved by analytic methods. This is discussed further in Appendix A for a formulation of *Pontryagin's maximum / minimum principle (PMP)*.

In indirect methods, the HBVP is solved using numerical techniques such as indirect shooting or indirect collocation. Such methods follow the general approach of first optimizing, i.e. obtaining the first-order necessary optimality conditions, and then discretizing the problem. The simplest method is *indirect single shooting*. This method requires an initial guess of the boundary conditions at one end of the interval. Then, the *Hamiltonian System* (8) is integrated to the end of the time horizon using numerical methods. The values at the boundary are compared with the desired values for the boundary conditions. If the error is below a given threshold, the algorithm terminates, otherwise the initial guesses are modified and the process is repeated. Indirect shooting is very sensitive to initial guesses and often fails to converge due to poor conditioning of the Hamiltonian. Therefore, the more stable *indirect multiple shooting* is preferred. This approach divides the time horizon $I = [t_0, t_f]$ into n subintervals. The single shooting method is then performed on each subinterval $[t_i, t_{i+1}]$, while the additional conditions

$$\boldsymbol{x}(t_i^-) = \boldsymbol{x}(t_i^+), \; \boldsymbol{\lambda}(t_i^-) = \boldsymbol{\lambda}(t_i^+) \tag{11}$$

are added to ensure the continuity of the solution. Figure 3 depicts the process of indirect multiple shooting, where the deflects between consecutive arcs must be 0. Although multiple shooting increases the complexity by adding additional constraints to the problem, it is usually a much more stable approach, but, like many indirect methods, still requires good guesses for convergence.

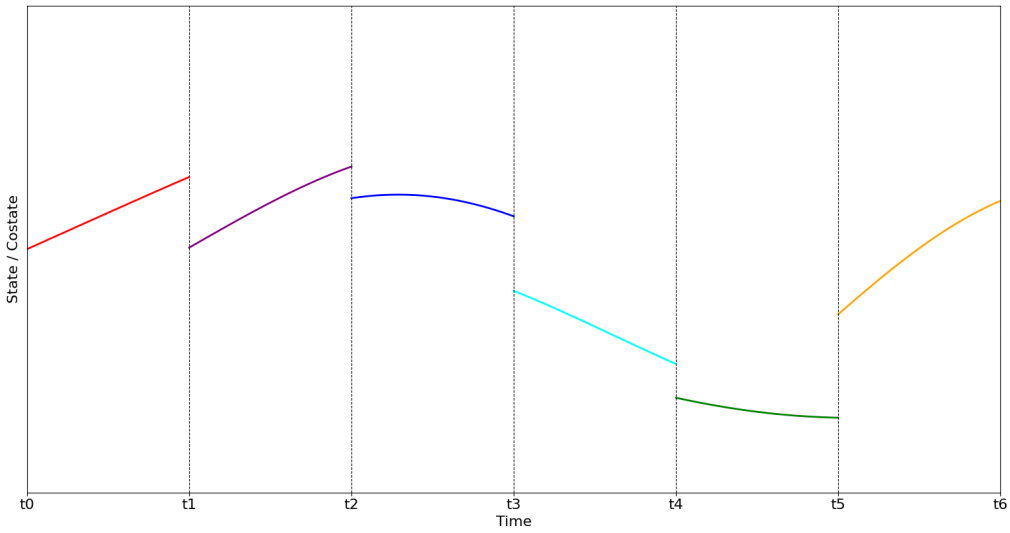


Figure 3: Structure of multiple shooting approaches (adapted from [24])

Another indirect method is *indirect collocation*. In this approach, both state and costate are given as polynomials in each subinterval $[t_i, t_{i+1}]$. These polynomials are defined such that they satisfy the *Hamiltonian system* (8) at certain nodes on the interval. To determine the coefficients of the piecewise polynomials, a system of nonlinear algebraic equations must be solved, resulting in the optimal state and costate trajectories.[24] Overall, indirect methods have several disadvantages, which has led to a shift in focus towards *direct methods*:

- in many instances it is very hard to solve the HBVP
- costates have to be introduced and treated accordingly
- solutions are very sensitive
- insights into the optimal control problem are necessary to obtain realistic guesses[21]

2.3.1.2 Direct Methods Unlike indirect methods, which follow the principle of first optimizing and then discretizing, direct methods follow the principle of first discretizing the continuous optimal control problem and then optimizing the resulting *Nonlinear Optimization Problem (NLP)*. These methods do not use the first-order necessary conditions for extremal solutions, but rather describe the problem directly in discrete form. The discretization can be performed in two ways. In the first variant, the so-called *control parameterization*, only the control variables $\boldsymbol{u}(t)$ are discretized. The second variant discretizes both the control $\boldsymbol{u}(t)$ and the states $\boldsymbol{x}(t)$ and is called *state and control parameterization*. Many modern solvers for dynamic optimization problems are based on direct methods and use state-of-the-art NLP solvers such as *Ipopt*[1], *SNOPT*[25][26] or *KNITRO*[28], which exploit advantageous sparsity patterns and use 2nd order derivatives.[24]

Direct methods based only on control parameterization are *direct single shooting* and *direct multiple shooting* methods. These parameterize the control in a specified functional form with undetermined coefficients, e.g.

$$\boldsymbol{u}(t) \approx \sum_{i=1}^{m} \boldsymbol{a}_i \psi_i(t), \tag{12}$$

with given functions ψ_i and coefficients $\boldsymbol{a}_i$. Given an initial guess for the free parameters, the dynamics of the system are integrated outside the optimization loop to obtain values for the states. These values together with the control, are embedded in a NLP to update the control trajectory until convergence to the optimum is achieved. Similar to indirect methods (Chapter 2.3.1.1), the difference between the two approaches is that single shooting uses only one integration interval, while multiple shooting uses many integration intervals and continuity is ensured by adding the constraints

$$\boldsymbol{x}(t_i^-) = \boldsymbol{x}(t_i^+) \tag{13}$$

for each subinterval. Therefore, direct multiple shooting splits the process of solving the differential equations into many subproblems that can be executed in parallel. The embedding of states and control also results in rather small NLPs compared to *direct collocation* methods. Nevertheless, collocation approaches have considerably better local convergence and are able to handle sparsity efficiently, often resulting in more performant and robust algorithms.[29][20][21][24]

Direct collocation is one of the most widely used and efficient approaches for numerically solving dynamic optimization problems. It describes the direct transcription of a dynamic optimization problem into a NLP when a collocation scheme or the corresponding implicit Runge-Kutta scheme is applied to its dynamics. Thus, both the states and controls are discretized at specific nodes. Depending on the type of collocation scheme, the states are approximated by a set of local, fixed degree, piecewise polynomials (*local collocation*) or a global, high degree polynomial (*global collocation*) and are usually written as a Lagrange interpolating polynomial. Local collocation methods have shown to be very efficient in solving optimal control problems, resulting in large but sparse NLPs.[17][24]

Pseudospectral (global orthogonal) collocation uses orthogonally collocated grid points, such as *Legendre-Gauss (LG)*, *Legendre-Gauss-Radau (LGR)*, *flipped Legendre-Gauss-Radau (fLGR)*, or *Legendre-Gauss-Lobatto (LGL)* points, to construct global high degree polynomials with an excellent accuracy and low oscillations. In recent years, these have become very popular and many pseudospectral methods have been developed. The main advantage of these methods is the *spectral / exponential* convergence when viewed as a function of the number of collocation points, if the optimal solution is smooth. Furthermore, many pseudospectral methods can apply the *Covector Mapping Principle* and thus relate the discrete approximation to the costates of the Hamiltonian. It can therefore be demonstrated that a discrete approximation is an optimal solution to the continuous problem.[30][24] An example of an optimal solution using a global orthogonal collocation method with a degree 70 polynomial based on *flipped Legendre-Gauss-Radau* nodes, is depicted in Figure 4.

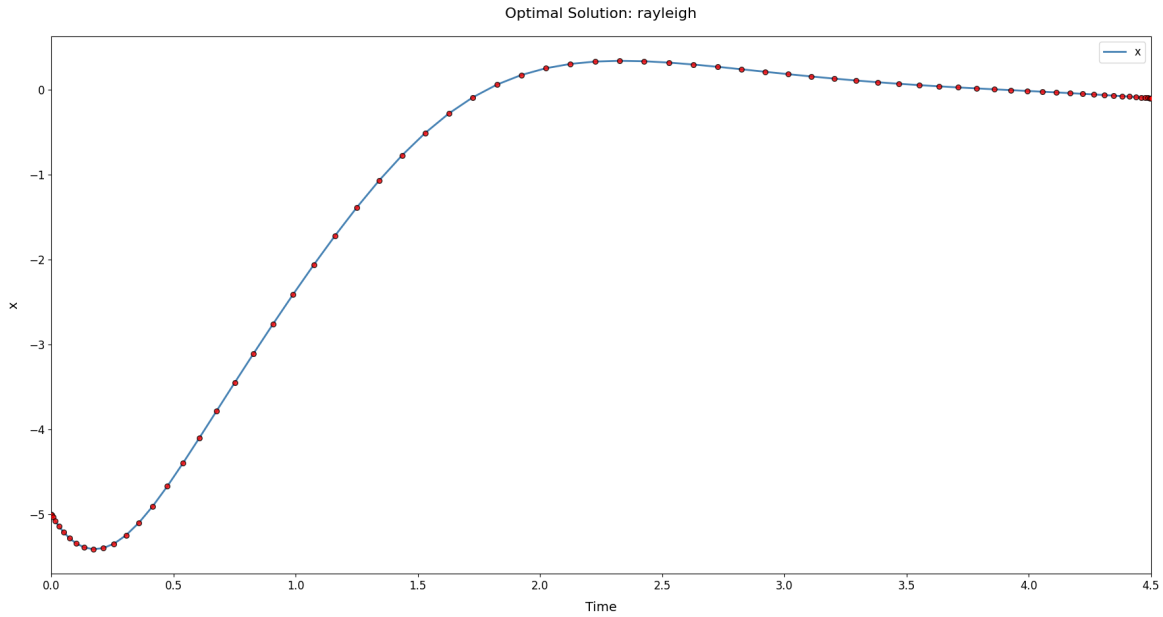


Figure 4: Global collocation state trajectory for problem *Rayleigh* (Appendix F) provided by GDOPT

Both local and global collocation methods typically use adaptive mesh refinement algorithms to achieve more accurate solutions. In local collocation, the placement of grid points or the length of intervals is often changed during the optimization process (*h-method*). Global collocation methods with a single interval usually change the number of orthogonal grid points and thus the degree of the polynomial (*p-method*). Modern pseudospectral approaches combine both methods into so-called *hp-adaptive methods*, in which the placement and length of the intervals as well as the degree of the polynomial on each interval can be changed with respect to the problem structure.[30]

3 Numerical Methods

In this chapter, several numerical methods are presented which are needed to discretize the continuous GDOP (Definition 2.2) with an orthogonal direct collocation approach. These include polynomial interpolation with Lagrange polynomials, numerical quadrature rules to approximate the GDOP Lagrange term, and collocation Runge-Kutta methods for solving the initial value problem.

3.1 Lagrange Interpolation

Polynomial interpolation with Lagrange basis polynomials is an important technique used throughout this thesis. The goal of polynomial interpolation is to find a polynomial $p \in P^n$ that satisfies the conditions

$$p(t_i) = x_i,\ i = 0, \dots, n, \tag{14}$$

where $t_0, \dots, t_n \in \mathbb{R}$ are $n+1$ distinct nodes and $x_0, \dots, x_n \in \mathbb{R}$ are the corresponding values at the nodes. The idea of Lagrange interpolation is to write the interpolating polynomial as a linear combination of the so-called *Lagrange basis polynomials*. The j-th Lagrange basis polynomial is defined as

$$l_j(t) := \prod_{\substack{k=0 \\ k \neq j}}^{n} \frac{t - t_k}{t_j - t_k}\ \forall j = 0, \dots, n \tag{15}$$

such that the following lemma holds.

Lemma 3.1. *Let $t_0, \dots, t_n \in \mathbb{R}$ be distinct nodes, then the Lagrange basis polynomials with these nodes satisfy*

$$l_j(t_i) = \delta_{ji} = \begin{cases} 1, & \text{if } j = i, \\ 0, & \text{if } j \neq i \end{cases} \tag{16}$$

Proof. trivial. □

Based on Lemma 3.1, the Lagrange interpolating polynomial can be constructed as a linear combination of the basis polynomials.

Definition 3.1 (Lagrange Interpolating Polynomial)**.** *Given distinct nodes $t_0, \dots, t_n \in \mathbb{R}$ and values $x_0, \dots, x_n \in \mathbb{R}$, the Lagrange interpolating polynomial $p \in P_n$ is defined as*

$$p(t) = \sum_{j=0}^{n} x_j l_j(t). \tag{17}$$

It is clear that this polynomial satisfies (14), since

$$p(t_i) = \sum_{j=0}^{n} x_j l_j(t_i) = \sum_{j=0}^{n} x_j \delta_{ji} = x_i. \tag{18}$$

Furthermore, the polynomial is the one unique polynomial p with $\deg p \leq n$ that meets (14) as seen in Theorem 3.2. There are also other ways to interpolate polynomials, e.g. *Newton interpolation*, which yield

different representations of the same polynomial.

Theorem 3.2 (Uniqueness)**.** *Given distinct nodes $t_0, \dots, t_n \in \mathbb{R}$ and values $x_0, \dots, x_n \in \mathbb{R}$, then exists an unique interpolating polynomial $p \in P_n$ that satisfies* (14)*.*

Proof. Assume it exists another polynomial $q \in P_n$, such that $p(t_i) = q(t_i) = x_i$, $i = 0, \dots, n$. Then the polynomial $d := p - q \in P_n$ has $n+1$ roots at $t_0, \dots, t_n$. By the fundamental theorem of algebra $d \equiv 0$, which implies $p(t) = q(t)\ \forall t \in \mathbb{R}$. □

In many cases, polynomial interpolation is used to approximate a function f with a polynomial p based on sample points $f(t_0), \dots, f(t_n)$, $i = 0, \dots, n$. Therefore, it is useful to have an error bound on the absolute difference between p and f. This bound is provided by Theorem 3.3.

Theorem 3.3 (Interpolation Error)**.** *Let all distinct nodes $t_0, \dots, t_n \in \mathbb{R}$ be inside an interval $[a, b]$ and let $f \in C^{n+1}[a, b]$ with $x_i = f(t_i)$, $i = 0, \dots, n$. Furthermore, p is the interpolating polynomial that satisfies $p(t_i) = x_i$ and $\omega(t) := (t - t_0)(t - t_1) \cdots (t - t_n)$, then for every $\tilde{t} \in [a, b]$ exist a $\xi \in (a, b)$ such that*

$$f(\tilde{t}) - p(\tilde{t}) = \omega(\tilde{t}) \frac{f^{(n+1)}(\xi)}{(n+1)!} \tag{19}$$

and thus

$$\left|f(\tilde{t}) - p(\tilde{t})\right| \leq \frac{\left|\omega(\tilde{t})\right|}{(n+1)!} \max_{\xi \in [a,b]} \left|f^{(n+1)}(\xi)\right| \leq \frac{(b-a)^{n+1}}{(n+1)!} \max_{\xi \in [a,b]} \left|f^{(n+1)}(\xi)\right|. \tag{20}$$

Proof. If $\tilde{t} = t_k$ for some $k \in \{0, \dots, n\}$, then $f(\tilde{t}) - p(\tilde{t}) = 0$. Therefore, let $t \neq t_k$ for all $k \in \{0, \dots, n\}$ and define a polynomial $\tilde{p} \in P^{n+1}$, which interpolates the pairs (t_i, x_i), $i = 0, \dots, n$ and $(\tilde{t}, f(\tilde{t}))$. Since $\omega(\tilde{t}) \neq 0$, $\tilde{p}$ can be written as

$$\tilde{p}(t) = p(t) + K\omega(t) \tag{21}$$

with $K = \frac{f(\tilde{t}) - p(\tilde{t})}{\omega(\tilde{t})} \in \mathbb{R}$. Note that $F := f - \tilde{p}$, $F \in C^{n+1}[a, b]$ has $n + 2$ distinct roots on $[a, b]$. By Rolle's theorem F' has at least $n + 1$ roots, F'' has at least n roots, …, and $F^{(n+1)}$ has at least 1 root. Thus, for every root ξ of $F^{(n+1)}$

$$\begin{aligned} 0 = F^{(n+1)}(\xi) &= f^{(n+1)}(\xi) - p^{(n+1)}(\xi) - K\omega^{(n+1)}(\xi) = f^{(n+1)}(\xi) - K(n+1)! \\ &\implies K = \frac{f^{(n+1)}(\xi)}{(n+1)!} \\ &\implies f(\tilde{t}) - p(\tilde{t}) = \omega(\tilde{t}) \frac{f^{(n+1)}(\xi)}{(n+1)!}. \text{[32]} \end{aligned} \tag{22}$$

□

Although the theorem gives an error bound, it does not show how to choose the nodes $t_0, \dots, t_n$ in a way to minimize the error. This is achieved by using *Chebyshev points*, which are not studied in this work. Furthermore, this result is the starting point for the construction of accurate quadrature rules, where given functions are approximated by an interpolating polynomial, which is then exactly integrated.[32]

3.1.1 Barycentric Representation

In addition to the standard formulation of Lagrange interpolation, there is a *barycentric* representation, which has advantages in computational speed and accuracy. The j-th Lagrange basis polynomial in barycentric form is given by

$$l_j(t) = \frac{\frac{\lambda_j}{t-t_j}}{\sum_{k=0}^{n} \frac{\lambda_k}{t-t_k}} \tag{23}$$

with the weights

$$\lambda_j = \prod_{\substack{k=0 \\ k\neq j}}^{n} \frac{1}{t_j - t_k}. \tag{24}$$

The Lagrange interpolating polynomial in barycentric form can then be written in the same way as (17).[33]

3.1.2 Differentiation Matrices

Using the barycentric representation, it is possible to efficiently construct *differentiation matrices* of the Lagrange basis polynomials. These are defined as follows:

Definition 3.2 (Differentiation Matrix)**.** *Let $l_j(t)$ be the j-th Lagrange basis polynomial corresponding to the distinct nodes $t_0, \ldots, t_n \in \mathbb{R}$. The matrix $D^{(m)} \in \mathbb{R}^{(n+1)\times(n+1)}$ with*

$$D_{ij}^{(m)} = \left(\frac{\mathrm{d}^m l_j(t)}{\mathrm{d}t^m}\right)_{t_i} = l_j^{(m)}(t_i) \tag{25}$$

is called the m-th differentiation matrix.[33]

Thus, the m-th differentiation matrix contains the m-th derivatives of the Lagrange basis polynomials evaluated at the nodes. Since the derivatives of a Lagrange interpolating polynomial p can be written as

$$\frac{\mathrm{d}^m p(t)}{\mathrm{d}t^m} = \sum_{j=0}^{n} p(t_j) \frac{\mathrm{d}^m l_j(t)}{\mathrm{d}t^m} = \sum_{j=0}^{n} x_j \frac{\mathrm{d}^m l_j(t)}{\mathrm{d}t^m}, \tag{26}$$

it is easy to obtain the derivatives at the nodes via the simple vector-vector or matrix-vector products

$$p^{(m)}(t_i) = \sum_{j=0}^{n} D_{ij}^{(m)} x_j \quad \text{or} \quad \boldsymbol{x}^{(m)} = D^{(m)}\boldsymbol{x}, \tag{27}$$

where $\boldsymbol{x}^{(m)} = (p^{(m)}(t_0), p^{(m)}(t_1), \ldots, p^{(m)}(t_n))^T$ and $\boldsymbol{x} = (p(t_0), p(t_1), \ldots, p(t_n))^T$. Note that the m-th differentiation matrix can be computed as the m-th power of the first matrix, i.e. $D^{(m)} = (D^{(1)})^m$, since the product $D^{(1)}\boldsymbol{x}$ yields the values of the first derivative at the nodes $\boldsymbol{x}^{(1)}$. These values also define an unique polynomial (the derivative) that can be differentiated with $D^{(1)}$ again. Obviously, this process can be generalized. Another fact about differentiation matrices is that $D^{(m)}\mathbf{1} = \mathbf{0}$ and thus $\det\left(D^{(m)}\right) = 0$, because the vector $\mathbf{1}$ defines the constant polynomial $p(t) \equiv 1$ with $p'(t) \equiv 0$ and, in particular, $p'(t_j) = 0$. In the course of this work, it will be useful to have formulas for the first and second differentiation matrices.

These have been evaluated in [34] and are written in terms of the barycentric weights (24):

$$D_{ij}^{(1)} = \begin{cases} \dfrac{\lambda_j}{\lambda_i}\dfrac{1}{t_i - t_j}, & \text{if } i \neq j, \\ -\displaystyle\sum_{\substack{k=0 \\ k\neq i}}^{n} \dfrac{\lambda_k}{\lambda_i}\dfrac{1}{t_i - t_k}, & \text{if } i = j, \end{cases} \tag{28}$$

and

$$D_{ij}^{(2)} = \begin{cases} 2D_{ij}^{(1)}\left(D_{ii}^{(1)} - \dfrac{1}{t_i - t_j}\right), & \text{if } i \neq j, \\ 2(D_{ii}^{(1)})^2 + 2\displaystyle\sum_{\substack{k=0 \\ k\neq i}}^{n} D_{ik}^{(1)}\dfrac{1}{t_i - t_k}, & \text{if } i = j. \end{cases} \tag{29}$$

This makes it possible to compute each of the differentiation matrices in $O(n^2)$, which is significantly faster than naively applying the product rule to (15).

3.2 Quadrature

Numerical integration, also known as *quadrature*, deals with the problem of numerically integrating a given function over an interval. Quadrature rules are widely used because not all integrals can be evaluated analytically. In addition, functions are often not given explicitly, but rather as a set of sample points. The general problem is to find the value of

$$I = \int_a^b f(t)\,\mathrm{d}t \tag{30}$$

by evaluating the function at specified nodes $t_1, \ldots, t_n$ and weighting the expression with certain weights $w_1, \ldots, w_n$, i.e.

$$\tilde{I} = \sum_{j=1}^{n} w_j f(t_j) \approx \int_a^b f(t)\,\mathrm{d}t. \tag{31}$$

Note that to make the transition to Runge-Kutta methods more gradual, from now on n nodes are used instead of $n + 1$.[32]

3.2.1 Interpolatory Quadrature

A simple way to numerically integrate a function f is to replace it with an interpolating polynomial p, e.g. a Lagrange polynomial, and exactly integrate the polynomial. This approach is called *interpolatory quadrature*. Given n distinct nodes $t_1, \ldots, t_n$ and the values $f(t_1), \ldots, f(t_n)$, the Lagrange polynomial (Definition 3.1) is defined as

$$p(t) = \sum_{j=1}^{n} f(t_j) l_j(t). \tag{32}$$

Thus, the integral approximation becomes

$$\tilde{I} = \int_a^b p(t)\,\mathrm{d}t = \int_a^b \sum_{j=1}^{n} f(t_j) l_j(t)\,\mathrm{d}t = \sum_{j=1}^{n} f(t_j) \int_a^b l_j(t)\,\mathrm{d}t \tag{33}$$

and it is clear that the weights are given by

$$w_j = \int_a^b l_j(t)\,\mathrm{d}t = \int_a^b \prod_{\substack{k=1 \\ k\neq j}}^{n} \frac{t - t_k}{t_j - t_k}\,\mathrm{d}t. \tag{34}$$

A simple class of quadrature rules are the *Newton-Cotes formulas*, which choose the nodes to be equidistant. There are two types of Newton-Cotes formulas: Closed formulas, where the boundary points a, b are used as nodes, and open formulas, where a and b are not used as nodes.[32] To show the principle procedure, the simplest closed Newton-Cotes formula, the so-called *trapezoidal rule*, is constructed.

Example 3.1 (Construction of the trapezoidal rule)**.** *Choose the nodes $t_1 = a$ and $t_2 = b$, then*

$$w_1 = \int_a^b \frac{t-b}{a-b}\,\mathrm{d}t = \frac{b-a}{2}, w_2 = \int_a^b \frac{t-a}{b-a}\,\mathrm{d}t = \frac{b-a}{2} \tag{35}$$

and thus

$$\int_a^b f(t)\,\mathrm{d}t \approx \frac{b-a}{2}\left(f(a) + f(b)\right). \tag{36}$$

In order to evaluate and compare quadrature rules, accuracy measures must be established. For quadrature rules, the *exactness*, i.e. the degree of the polynomials that are exactly integrated, is an appropriate measure. In addition the *order* and the *error* of quadrature rules are defined, which are central concepts later on.

Definition 3.3 (Exactness, Degree of Exactness)**.** *A quadrature rule has the degree of exactness $m \in \mathbb{N}$, if it exactly integrates all polynomials $p \in P^m$ and m is maximal.*

Definition 3.4 (Order)**.** *A quadrature rule is of order $m \in \mathbb{N}$, if it has degree of exactness $m - 1$.*

Definition 3.5 (Error)**.** *Given a quadrature rule with nodes $t_1, \ldots, t_n$ and weights $w_1, \ldots, w_n$, the error E of the method for a given function f is*

$$E[f] = \int_a^b f(t)\,\mathrm{d}t - \sum_{j=1}^{n} w_j f(t_j). \tag{37}$$

Based on Definition 3.3 a lower bound on the degree of exactness for all interpolatory quadrature rules can be established.

Theorem 3.4. *An interpolatory quadrature rule with n distinct nodes $t_1, \ldots, t_n \in [a, b]$ has at least degree of exactness $n - 1$ and order n.*

Proof. Given the interpolating polynomial p with $p(t_i) = f(t_i)$ for all $i = 1, \ldots, n$. By construction and Theorem 3.3

$$E[f] = \int_a^b f(t) - p(t)\,\mathrm{d}t = \frac{1}{n!}\int_a^b \omega(t) f^{(n)}(\xi(t))\,\mathrm{d}t \text{ with } \omega(t) = \prod_{k=1}^{n}(t - t_k) \tag{38}$$

and $\xi \in [a, b]$. If $f \in P^m$ with $m < n$, then $f^{(n)} \equiv 0$ and thus $E[f] = 0$.[32] □

By Theorem 3.4, the trapezoidal rule exactly integrates at least linear functions. In general, the error term of the trapezoidal rule can be written as

$$E[f] = -\frac{(b-a)^3}{12} f''(\xi), \text{ if } f \in C^2[a,b]. \tag{39}$$

Another closed Newton-Cotes formula is *Simpson's rule*

$$\int_a^b f(t)\,\mathrm{d}t \approx \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right). \tag{40}$$

By construction, it has at least degree of exactness 2, but in fact it also exactly integrates cubic polynomials, making it a method of order 4. The error term is

$$E[f] = -\frac{(b-a)^5}{2880} f^{(4)}(\xi), \text{ if } f \in C^4[a,b]. \tag{41}$$

Newton-Cotes formulas are only stable for a small number of nodes n, because the error in the polynomial interpolation grows due to the equidistant grid, and additionally weights become negative. Therefore, the interval $[a,b]$ is usually divided into many subintervals. The quadrature rule is then applied to each subinterval.[31]

3.2.2 Gaussian Quadrature

An additional approach for the construction of quadrature rules is presented. It is based on the theory of orthogonal polynomials (Appendix B) and allows for the construction of quadrature rules of (possibly) maximum order that contain only positive weights. These methods are called *Gaussian quadratures* and were first introduced by *C. F. Gauss*. In contrast to Newton-Cotes formulas, where the nodes are always placed to be equidistant, these quadrature rules make extensive use of placing nodes in a non-equidistant fashion to achieve high orders. The formulas are usually constructed on the interval $[-1,1]$. Thus, the interval $[a,b]$ is rescaled appropriately and the process can be undone later on.

3.2.2.1 Radau Quadrature Although the *Gauss-Legendre quadrature* (Appendix C) achieves an optimal ratio between the order and the number of nodes, other quadrature rules are used for the discretization of the GDOP. The *Radau quadrature* [37], which was first introduced by *R. Radau* in 1880, is very similar to the *Gauss-Legendre quadrature*. It is also based on roots of orthogonal polynomials, but fixes one node at either endpoint -1 or 1. In this thesis only the quadrature rule with fixed node $t_n = 1$ is studied, since the corresponding *Runge-Kutta method* has highly desirable properties. This quadrature rule is based on the *Jacobi polynomials*, which are defined as follows.

Definition 3.6. *The polynomials* $(P_n^{(\alpha,\beta)})_{n\in\mathbb{N}_0}$*, which satisfy Definition B.1 with the weighting function* $w(t) = (1-t)^\alpha(1+t)^\beta$*,* $\alpha,\beta > -1$*, i.e.*

$$\begin{aligned} &\int_{-1}^{1} (1-t)^\alpha (1+t)^\beta P_n^{(\alpha,\beta)} P_m^{(\alpha,\beta)}\,\mathrm{d}t = 0,\ n \neq m \\ &\int_{-1}^{1} (1-t)^\alpha (1+t)^\beta P_n^{(\alpha,\beta)} P_n^{(\alpha,\beta)}\,\mathrm{d}t \neq 0,\ \forall n \geq 0 \end{aligned} \tag{42}$$

and $P_0^{(\alpha,\beta)} = 1$ *are called Jacobi polynomials.[35]*

Only the *Jacobi polynomials* with $\alpha = 1$ and $\beta = 0$ are used for the construction of the *Radau quadrature*. This specific system of orthogonal polynomials, i.e. $(P_n^{(1,0)})_{n\in\mathbb{N}_0}$, satisfies the three-term recurrence relation

$$P_n^{(1,0)}(t) = \frac{1}{(n+1)(2n-1)}\left[((2n+1)(2n-1)t+1)\,P_{n-1}^{(1,0)}(t) - (n-1)(2n+1)P_{n-2}^{(1,0)}(t)\right],\ n \geq 2 \tag{43}$$

with $P_0^{(1,0)}(t) = 1$ and $P_1^{(1,0)}(t) = \frac{1}{2}(3t+1)$.[36] The next few polynomials are $P_2^{(1,0)}(t) = \frac{1}{2}(5t^2+2t-1), P_3^{(1,0)}(t) = \frac{1}{8}(35t^3+15t^2-15t-3), P_4^{(1,0)}(t) = \frac{1}{8}(63t^4+28t^3-42t^2-12t+3)$. The roots of the polynomial $(1-t)P_n^{(1,0)}(t)$ are called *flipped Legendre-Gauss-Radau (fLGR)* points. The standard *Legendre-Gauss-Radau (LGR)* points are obtained by simply switching the sign of each fLGR point and thus correspond to the quadrature rules with fixed endpoint -1.[49] Based on properties of these polynomials and the fLGR points, a n node quadrature rule of order $2n-1$ can be constructed.

Theorem 3.5 (Radau Quadrature)**.** *The quadrature rule*

$$\sum_{j=1}^{n} w_j f(t_j) \tag{44}$$

with n nodes t_j *chosen as roots of the polynomial* $(1-t)P_{n-1}^{(1,0)}(t)$ *and the weights*

$$w_j = \int_{-1}^{1} \prod_{\substack{k=1\\k\neq j}}^{n} \frac{t-t_k}{t_j-t_k}\,\mathrm{d}t\ j = 1,\dots,n \tag{45}$$

obtains order $2n-1$.

Proof. Let $p \in P^{2n-2}$ be arbitrary. Then p can be divided by $\Phi(t) = (1-t)P_{n-1}^{(1,0)}(t)$, which yields

$$p(t) = q(t)\Phi(t) + r(t), \tag{46}$$

with $q \in P^{n-2}$ and $r \in P^{n-2}$. Thus

$$\int_{-1}^{1} p(t)\,\mathrm{d}t = \int_{-1}^{1} q(t)\Phi(t)\,\mathrm{d}t + \int_{-1}^{1} r(t)\,\mathrm{d}t = \underbrace{\int_{-1}^{1}(1-t)P_{n-1}^{(1,0)}(t)q(t)\,\mathrm{d}t}_{=0} + \int_{-1}^{1} r(t)\,\mathrm{d}t = \int_{-1}^{1} r(t)\,\mathrm{d}t, \tag{47}$$

since the Jacobi polynomials form a basis and each Jacobi polynomial $P_m^{(\alpha,\beta)}$ with degree $m < n-1$ is orthogonal to $P_{n-1}^{(\alpha,\beta)}$ with weight $(1-t)^\alpha(1+t)^\beta$, $\alpha,\beta > -1$ (Definition B.1). Since the zeros of orthogonal polynomials are simple and lie inside the interval (Lemma B.2), all roots of Φ are distinct and lie on the interval $[-1,1]$. Choosing these roots as nodes and constructing an interpolatory quadrature rule yields

$$\sum_{j=1}^{n} w_j p(t_j) = \sum_{j=1}^{n} w_j q(t_j)\underbrace{\Phi(t_j)}_{=0} + \sum_{j=1}^{n} w_j r(t_j) = \sum_{j=1}^{n} w_j r(t_j) = \int_{-1}^{1} r(t)\,\mathrm{d}t = \int_{-1}^{1} p(t)\,\mathrm{d}t, \tag{48}$$

since $r(t) \in P^{n-2}$ can be exactly integrated with any n node interpolatory quadrature rule (Theorem 3.4).

Therefore, the interpolatory quadrature rule with the n zeros of $(1-t)P_{n-1}^{(1,0)}(t)$ chosen as nodes has degree of exactness $2n-2$ and order $2n-1$. (adapted from[32]) □

Additionally, the error term of the quadrature rule can be expressed as

$$E[f] = \frac{2^{2n} n[(n-1)!]^4}{2[(2n-1)!]^3} f^{(2n-1)}(\xi),\ \xi \in (-1,1) \tag{49}$$

for sufficiently smooth $f \in C^{2n-1}$.[38] By Theorem C.3 all weights of the *Radau quadrature* are positive, which makes the method well conditioned. An example is considered to show the first few formulas of the quadrature rule.

Example 3.2. *The Radau quadrature with $n = 1, 2, 3$ nodes is given by:*

$$\begin{aligned}
n = 1 &: \int_{-1}^{1} f(t)\,\mathrm{d}t \approx 2f(1) \\
n = 2 &: \int_{-1}^{1} f(t)\,\mathrm{d}t \approx \frac{3}{2} f\left(-\frac{1}{3}\right) + \frac{1}{2} f\,(1) \\
n = 3 &: \int_{-1}^{1} f(t)\,\mathrm{d}t \approx \frac{16-\sqrt{6}}{18} f\left(-\frac{1+\sqrt{6}}{5}\right) + \frac{16+\sqrt{6}}{18} f\left(\frac{1-\sqrt{6}}{5}\right) + \frac{2}{9} f\,(1)
\end{aligned} \tag{50}$$

3.3 Runge-Kutta Methods

The following section presents methods for discretizing the dynamic of the GDOP. *Runge-Kutta methods* are an important class of numerical methods for solving initial value problems (IVP) of the form

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(t, \boldsymbol{x}(t)),\ \boldsymbol{x}(t_0) = \boldsymbol{x}_0. \tag{51}$$

From now on, it is assumed that (51) has an unique solution on the interval $[t_0, t_f]$. At first, the IVP is transformed into an equivalent integral equation

$$\boldsymbol{x}(\tau) = \boldsymbol{x}(t_0) + \int_{t_0}^{\tau} \boldsymbol{f}(t, \boldsymbol{x}(t))\,\mathrm{d}t. \tag{52}$$

Splitting the time horizon $[t_0, t_f]$ into n intervals $[t_0, t_1], [t_1, t_2], \ldots, [t_{n-1}, t_n]$ with $t_n = t_f$ yields the iterative process

$$\boldsymbol{x}(t_{i+1}) = \boldsymbol{x}(t_i) + \int_{t_i}^{t_{i+1}} \boldsymbol{f}(t, \boldsymbol{x}(t))\,\mathrm{d}t,\ \ i = 0, \ldots, n-1. \tag{53}$$

For simplicity, each subinterval $[t_i, t_{i+1}]$ has constant length $h \equiv t_{i+1} - t_i$. Furthermore, m nodes $t_{ij} = t_i + c_j h, j = 1, \ldots, m$ with $c_j \in [0,1]$ are introduced. Replacing the exact solution $\boldsymbol{x}(t_i)$ by a numerical approximation $\boldsymbol{x}_i$ and applying a quadrature rule (31) results in

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + h \sum_{j=1}^{m} b_j \boldsymbol{f}(t_{ij}, \boldsymbol{x}(t_{ij})), \tag{54}$$

where b_j are the weights of the quadrature rule. It is apparent that the values $x(t_{ij})$ are not given. Thus, approximations $\boldsymbol{x}_{ij} \approx \boldsymbol{x}(t_{ij})$ are calculated via a similar approach

$$\boldsymbol{x}_{ij} = \boldsymbol{x}_i + h \sum_{l=1}^{m} a_{il} \boldsymbol{f}(t_{il}, \boldsymbol{x}_{il})), \;\; j = 1, \ldots, m \tag{55}$$

for given values $a_{il} \in \mathbb{R}$. Combining equations (54) and (55) leads to the definition of *Runge-Kutta methods*.[39]

Definition 3.7 (Runge-Kutta Method)**.** *Let $b_i, a_{ij} \in \mathbb{R}$, $i, j = 1, \ldots, m$ and let $c_i = \sum_{j=1}^{m} a_{ij}$. A m-stage Runge-Kutta method is given by*

$$\begin{aligned} \boldsymbol{k}_j &= \boldsymbol{f}(t_0 + c_j h, \boldsymbol{x}_0 + h \sum_{l=1}^{m} a_{il} \boldsymbol{k}_l), \;\; j = 1, \ldots, m \\ \boldsymbol{x}_1 &= \boldsymbol{x}_0 + h \sum_{j=1}^{m} b_j \boldsymbol{k}_j. \end{aligned} \tag{56}$$

In Definition 3.7 only a single step of the method is presented. In practice this process is repeated n times as seen in (54). The method defining coefficients b_j and c_j are called weights and nodes. Furthermore, the matrix A with entries a_{ij} is called *Runge-Kutta* or *Butcher matrix*. The coefficients are usually written in a compact *Butcher-tableau*:

$$\begin{array}{c|c} \boldsymbol{c} & A \\ \hline & \boldsymbol{b}^T \end{array} = \begin{array}{c|ccc} c_1 & a_{11} & \ldots & a_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ c_m & a_{m1} & \ldots & a_{mm} \\ \hline & b_1 & \ldots & b_m \end{array} \tag{57}$$

Runge-Kutta methods can be either *explicit* or *implicit*. In explicit methods the slopes k_j are all evaluated directly based on previously calculated slopes. Thus, $a_{ij} = 0$, $j \geq i$ holds for all explicit methods. This can be visualized as the *Butcher matrix* being a lower triangular matrix without diagonal entries. Implicit methods can have arbitrarily structured matrices A. This implies that in each step a nonlinear system of equation has to be solved, making implicit methods more numerically expensive than explicit methods. However, these offer crucial advantages in terms of order and stability and play a major role in the course of this thesis.

3.3.1 Order and Construction

To compare and construct Runge-Kutta methods, the order of a method is introduced.

Definition 3.8. *A Runge-Kutta method has order p, if for all sufficiently regular problems* (51) *the local error $\boldsymbol{x}_1 - \boldsymbol{x}(t_0 + h)$ satisfies*

$$\boldsymbol{x}_1 - \boldsymbol{x}(t_0 + h) = O(h^{p+1}) \text{ as } h \to 0. \tag{58}$$

This concept of order is a generalization of the order for quadrature rules, because the error of a quadrature rule (31) with order p is always of the form $E[f] = C \cdot h^{p+1} f^{(p)}(\xi)$, where $h > 0$ is the length of the interval $[a, b]$ and $C \in \mathbb{R}$ is some constant.

By applying a Taylor expansion to $\boldsymbol{x}_1$ and $\boldsymbol{x}(t_0 + h)$ around $h = 0$, algebraic conditions can be obtained, that a Runge-Kutta method of order p has to satisfy. The conditions for $p = 1, 2, 3$ are

$$\begin{aligned} p = 1 &: \sum_{j=1}^{m} b_j = 1 \\ p = 2 &: \sum_{j=1}^{m} b_j c_j = \frac{1}{2} \text{ and } \sum_{j=1}^{m} b_j c_j^2 = \frac{1}{3} \\ p = 3 &: \sum_{i=1}^{m} \sum_{j=1}^{m} b_j a_{ij} c_j = \frac{1}{6}. \end{aligned} \tag{59}$$

Note that a method has order p, iff all previous conditions are met and the condition for p is met. The process of generating algebraic conditions to construct high order Runge-Kutta methods can be extended. Nevertheless, the number of conditions grows rapidly and solving the system of equations symbolically is very hard. Therefore, general construction formulas for high order methods are very important.[40][41]
A few examples of explicit methods are considered. The simplest Runge-Kutta method is the *forward Euler method* with the *Butcher-tableau*

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array} \tag{60}$$

and iteration $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + h\boldsymbol{f}(t_i, \boldsymbol{x}_i)$. The method has order $p = 1$ and can be interpreted as following the direction of the vector field $\boldsymbol{f}$ at each step. Other examples of explicit methods are the *explicit trapezoidal method* with 2 stages and order $p = 2$, *Ralston's method* with 3 stages and order 3 as well as the famous method of *Kutta* with 4 stages and order $p = 4$, which is a generalization of *Simpson's rule*:

$$\begin{array}{c|cc} 0 & \\ 1 & 1 & \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \qquad \begin{array}{c|ccc} 0 & & & \\ \frac{1}{2} & \frac{1}{2} & & \\ \frac{3}{4} & 0 & \frac{3}{4} & \\ \hline & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} \end{array} \qquad \begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array} \tag{61}$$

3.3.2 Stability

In many practical applications these methods, especially the higher order ones, work quite well. However, in the case of stiff differential equations explicit methods have major disadvantages. To introduce the concept of stability, the scalar test equation

$$\dot{x} = \lambda x, \ \lambda \in \mathbb{C} \tag{62}$$

is considered. When performing a single step $h > 0$, the exact solution $x(t)$ is scaled by a factor of $\exp z$ with $z := \lambda h$. The solution provided by a Runge-Kutta method is also scaled by some function $R(z)$, i.e. $x_1 = R(\lambda h)x_0$.

Definition 3.9 (Stability Function)**.** *Given a Runge-Kutta method and a step size $h > 0$, the stability function $R(z)$, $z \in \mathbb{C}$ is the function which satisfies $x_1 = R(\lambda h)x_0$, when the method is applied to the differential equation*

$\dot{x} = \lambda x, \ \lambda \in \mathbb{C}$.

It is of particular interest to study this function for $\operatorname{Re} z < 0$, because then $\exp z$ has a damping effect on the solution. Therefore, the *stability function* should satisfy the equation $|R(z)| \leq 1$ to ensure that the solution does not grow. If this property does not apply, the solution is unstable and may oscillate heavily. This concept is summarized in the following definition, introduced by *Dahlquist*.

Definition 3.10 (A-stable)**.** *A Runge-Kutta method is called A-stable, if* $|R(z)| \leq 1$ *for all* $z \in \mathbb{C}$ *with* $\operatorname{Re} z < 0$.

The stability function can be calculated efficiently with the following representation.

Lemma 3.6. *The stability function of a Runge-Kutta method is given by*

$$R(z) = 1 + z\boldsymbol{b}^T(I - zA)^{-1}\mathbf{1}. \tag{63}$$

Proof. If the method is applied to the test equation (62) for a step size $h > 0$, then $\boldsymbol{k} = (k_1, \ldots, k_m)^T = \lambda(\mathbf{1} + hA\boldsymbol{k})x_0$. Thus, $\boldsymbol{k} = \lambda(I - \lambda hA)^{-1}\mathbf{1}x_0$ and $x_1 = x_0 + h\boldsymbol{b}^T\lambda(I - \lambda hA)^{-1}\mathbf{1}x_0 = (1 + \lambda h\boldsymbol{b}^T(I - \lambda hA)^{-1}\mathbf{1})x_0$. Setting $z := \lambda h$ results in

$$x_1 = R(z)x_0 = (1 + z\boldsymbol{b}^T(I - zA)^{-1}\mathbf{1})x_0. \tag{64}$$

□

The stability function for the *forward Euler method* is $R(z) = 1 + z$ and that for the *explicit trapezoidal method* is $R(z) = 1 + z + \frac{1}{2}z^2$. For explicit methods, $R(z)$ is a polynomial approximation of the exponential function, which makes these methods not A-stable. Therefore, the step size h has to be chosen sufficiently small for the method to converge. The regions of stability, i.e. $\{z \in \mathbb{C} \,|\, |R(z)| \leq 1\}$, for all aforementioned methods are displayed below.[41][43]

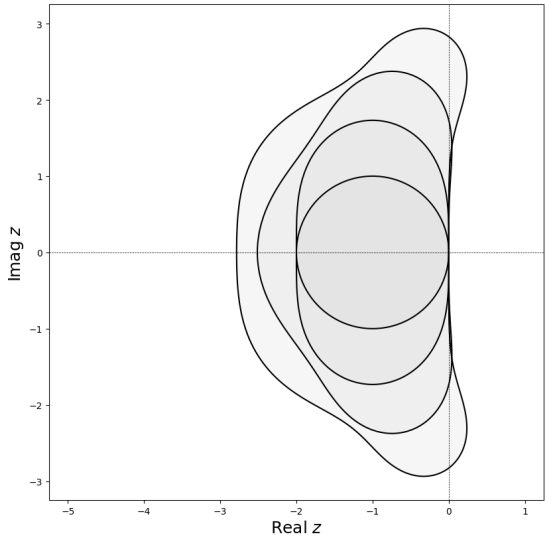


Figure 5: Stability regions of explicit Runge-Kutta methods

3.4 Collocation Methods

Now, the concept of *collocation*, a systematic approach to generate implicit Runge-Kutta methods of arbitrary high orders is presented. Again, the idea is to replace the exact solution by a polynomial approximation. In

this case the approximation has to satisfy the initial value and the differential equation at given nodes c_i.

Definition 3.11 (Collocation)**.** *Let $c_1, \dots, c_m$ be distinct real numbers (usually $0 \leq c_i \leq 1$). The collocation polynomial $\boldsymbol{q}(t)$ is a polynomial of degree m satisfying*

$$\begin{aligned} \boldsymbol{q}(t_0) &= \boldsymbol{x}_0 \\ \dot{\boldsymbol{q}}(t_0 + c_i h) &= \boldsymbol{f}(t_0 + c_i h, \boldsymbol{q}(t_0 + c_i h)), \;\; i = 1, \dots, m, \end{aligned} \tag{65}$$

and the numerical solution of the collocation method is defined by $\boldsymbol{x}_1 = \boldsymbol{q}(t_0 + h)$.[40]

A great feature of the *collocation* approach is that it produces a continuous approximation, unlike the standard Runge-Kutta methods, which only provide a discrete approximation. This is a major benefit, especially when running simulations and reasoning about the behavior of real-world systems. It is not clear that Definition 3.11 leads to a Runge-Kutta method. However, every set of distinct nodes $c_1, \dots, c_m$ defines an unique implicit method as shown in the following theorem.

Theorem 3.7. *A collocation method (Definition 3.11) is equivalent to the m-stage Runge-Kutta method (Definition 3.7) with coefficients*

$$a_{ij} = \int_0^{c_i} l_j(\tau)\, \mathrm{d}\tau, \qquad b_j = \int_0^1 l_j(\tau)\, \mathrm{d}\tau, \tag{66}$$

where $l_i(\tau)$ is the Lagrange basis polynomial (15) $l_i(\tau) = \prod_{l \neq i} (\tau - c_l)/(c_i - c_l)$.

Proof. Let $\boldsymbol{q}(t)$ be the collocation polynomial and define $\boldsymbol{k}_j := \dot{\boldsymbol{q}}(t_0 + c_i h)$. Writing $\dot{\boldsymbol{q}}$ as a *Lagrange interpolating polynomial* (Definition 3.1) leads to

$$\dot{\boldsymbol{q}}(t_0 + \tau h) = \sum_{j=1}^{m} \boldsymbol{k}_j l_j(\tau). \tag{67}$$

Applying an integration results in

$$\boldsymbol{q}(t_0 + c_i h) = \boldsymbol{x}_0 + h \sum_{j=1}^{m} \boldsymbol{k_j} \int_0^{c_i} l_j(\tau)\, \mathrm{d}\tau. \tag{68}$$

Comparing the coefficients with the definition of Runge-Kutta methods (Definition 3.7) yields the equation $a_{ij} = \int_0^{c_i} l_j(\tau)\, \mathrm{d}\tau$. Furthermore, integration from 0 to 1 results in $b_j = \int_0^1 l_j(\tau)\, \mathrm{d}\tau$.[40] □

Consequently, the coefficients a_{ij} and b_j are completely determined by the chosen nodes c_j. It is therefore of particular interest to choose the nodes, such that the method has a high order and good stability properties. Two fundamental theorems are stated below. These show that collocation methods provide excellent approximations to the exact solution of initial value problems.

Theorem 3.8. *The collocation polynomial $\boldsymbol{q}(t)$ is an approximation of order m to the exact solution of* (51) *on the whole interval, i.e.*

$$\|\boldsymbol{q}(t) - \boldsymbol{x}(t)\| \leq C h^{m+1}, \;\; t \in [t_0, t_0 + h] \tag{69}$$

and for sufficiently small h. Moreover, the derivatives of $\boldsymbol{q}(t)$ satisfy for $t \in [t_0, t_0 + h]$

$$\left\| \boldsymbol{q}^{(k)}(t) - \boldsymbol{x}^{(k)}(t) \right\| \leq C h^{m+1-k}, \;\; k = 0, \dots, m. \tag{70}$$

Proof. see [40]. □

Theorem 3.9 (Superconvergence)**.** *The order of a collocation method is identical to that of the underlying quadrature rule.*

Proof. see [40]. □

Especially Theorem 3.9 is of utmost importance. It shows that choosing the nodes in accordance to the nodes of the *Gauss-Legendre* (Theorem C.2) or *Radau quadrature* (Theorem 3.5) leads to m-stage methods of order $2m$ and $2m-1$, respectively. Thus, it is possible to construct methods of arbitrary high order. The only difference to the nodes in quadrature rules is that the nodes have to be rescaled to the interval $[0,1]$ by the transformation $t \mapsto \frac{1}{2}(t+1)$.[42][40]

3.4.1 Radau IIA

This chapter introduces the class of *Radau IIA* methods. These methods are the already mentioned collocation methods, which are generated by choosing the rescaled fLGR points, which are the nodes of the *Radau quadrature rule* (Theorem 3.5). Radau IIA methods have order $2m-1$ and excellent stability properties. It is common to write RadauP, where P is the order of the method, when referring to a specific Radau IIA method. Although Gauss-Legendre methods have a higher order of $2m$, it will be shown that there are decisive benefits to using Radau IIA schemes. The *Butcher-tableaus* of Radau IIA for $m = 1, 2, 3$ are

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \qquad \begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array} \qquad \begin{array}{c|ccc} \frac{2}{5}-\frac{\sqrt{6}}{10} & \frac{11}{45}-\frac{7\sqrt{6}}{360} & \frac{37}{225}-\frac{169\sqrt{6}}{1800} & -\frac{2}{225}-\frac{\sqrt{6}}{75} \\ \frac{2}{5}+\frac{\sqrt{6}}{10} & \frac{37}{225}+\frac{169\sqrt{6}}{1800} & \frac{11}{45}+\frac{7\sqrt{6}}{360} & -\frac{2}{225}+\frac{\sqrt{6}}{75} \\ 1 & \frac{4}{9}-\frac{\sqrt{6}}{36} & \frac{4}{9}+\frac{\sqrt{6}}{36} & \frac{1}{9} \\ \hline & \frac{4}{9}-\frac{\sqrt{6}}{36} & \frac{4}{9}+\frac{\sqrt{6}}{36} & \frac{1}{9} \end{array} \tag{71}$$

The method with 1-stage is equivalent to the *implicit* or *backward Euler method.* This is one of the most stable Runge-Kutta methods, but has only order 1. In Figure 6, the 3-stage Radau5 is applied to an example IVP with a quite large step size of $h = 0.36$. As stated in Theorem 3.8, the *collocation method* is a very accurate approximation on the entire interval. Furthermore, even though a large step size is chosen, only a small error to the exact solution is present.[42][44]

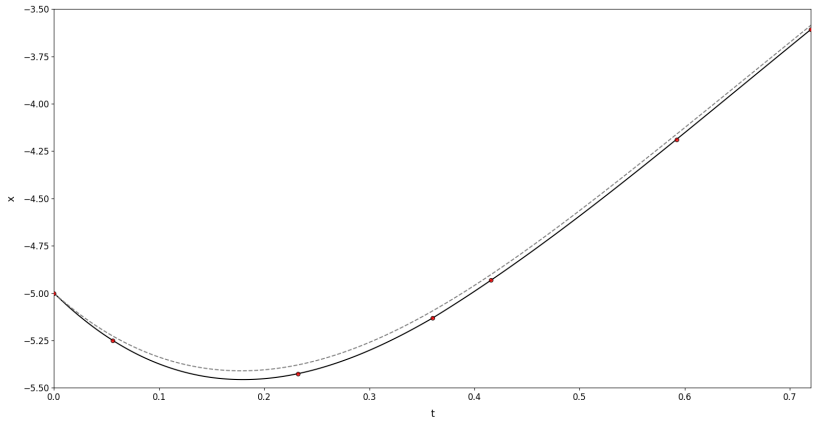


Figure 6: Piecewise polynomial approximation with the 3-stage Radau IIA to the dotted exact solution

3.4.1.1 Simplified Representation Radau IIA methods can be rewritten in a simplified form. By Definition 3.11 and Theorem 3.7

$$\begin{aligned} \boldsymbol{x}_{0,i} &:= \boldsymbol{q}(t_0 + c_i h) = \boldsymbol{x}_0 + h \sum_{j=1}^{m} a_{ij} \boldsymbol{k}_j, \;\; i = 1, \dots, m \\ \implies \boldsymbol{x}_{0,i} - \boldsymbol{x}_0 &= h \sum_{j=1}^{m} a_{ij} \boldsymbol{k}_j = h \begin{pmatrix} a_{i1} I & \dots & a_{im} I \end{pmatrix} \begin{pmatrix} \boldsymbol{k}_1 \\ \vdots \\ \boldsymbol{k}_m \end{pmatrix}, \end{aligned} \tag{72}$$

where $I \in \mathbb{R}^{n_x \times n_x}$ is the identity matrix. Applying this for all $\boldsymbol{x}_{0,1}, \dots, \boldsymbol{x}_{0,m}$ yields the representation

$$\begin{pmatrix} \boldsymbol{x}_{0,1} - \boldsymbol{x}_0 \\ \vdots \\ \boldsymbol{x}_{0,m} - \boldsymbol{x}_0 \end{pmatrix} = h \begin{pmatrix} a_{11} I & \dots & a_{1m} I \\ \vdots & \ddots & \vdots \\ a_{m1} I & \dots & a_{mm} I \end{pmatrix} \begin{pmatrix} \boldsymbol{f}(t_{0,1}, \boldsymbol{x}_{0,1}) \\ \vdots \\ \boldsymbol{f}(t_{0,m}, \boldsymbol{x}_{0,m}) \end{pmatrix} = h(A \otimes I) \begin{pmatrix} \boldsymbol{f}(t_{0,1}, \boldsymbol{x}_{0,1}) \\ \vdots \\ \boldsymbol{f}(t_{0,m}, \boldsymbol{x}_{0,m}) \end{pmatrix} \tag{73}$$

with $t_{0,i} = t_0 + c_i h$ and the *Kronecker product*

$$G \otimes H = \begin{pmatrix} g_{11} H & \dots & g_{1m} H \\ \vdots & \ddots & \vdots \\ g_{n1} H & \dots & g_{nm} H \end{pmatrix}, \; G \in \mathbb{R}^{n \times m}. \tag{74}$$

At a later stage the method will be embedded in a NLP. For this reason it is important to have as few variables and equations as possible. When using a collocation method which does not use the endpoint $c_m = 1$ as a node, e.g. the Gauss-Legendre method, the slopes $\boldsymbol{k}_1, \dots, \boldsymbol{k}_m$ as well as the value $\boldsymbol{x}_1$ have to be calculated. This results in $m + 1$ equation and $m + 1$ variables per interval. Since the Radau IIA method uses the endpoint as a node and is of collocation type,

$$\boldsymbol{x}_1 = \boldsymbol{q}(t_0 + h) = \boldsymbol{q}(t_0 + c_m h) = \boldsymbol{x}_{0,m} \tag{75}$$

holds. Thus, the value $\boldsymbol{x}_0$ is given from the previous interval, which results in only m variables per interval. Furthermore, only m equations per interval are needed as well, because the new approximation is exactly the value of the *collocation polynomial* at the final node, i.e. $\boldsymbol{x}_{0,m} = \boldsymbol{x}_1$. Therefore, Radau IIA has a better ratio of order to number of equations and variables than the Gauss-Legendre methods.[42]

3.4.1.2 Stability Additionally, Radau IIA has favorable stability properties. Since it is not known whether the IVP is stiff, it is a desirable property to be accurate even for stiff problems. The stability regions of the first 5 methods are displayed in Figure 7. All Radau IIA as well as Gauss-Legendre methods are A-stable, i.e. $|R(z)| \leq 1$ for $\operatorname{Re} z < 0$.

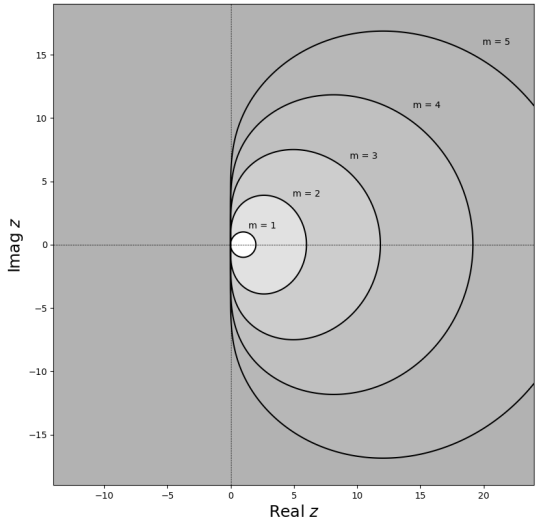


Figure 7: Stability regions of Radau IIA methods

In addition to A-stability, there are other concepts that an integration scheme should fulfill in order to solve stiff differential equations efficiently. Two important stability concepts are defined below.

Definition 3.12 (B-stable)**.** *A Runge-Kutta method is called B-stable, if the contractivity condition*

$$(\boldsymbol{f}(t,\boldsymbol{x}) - \boldsymbol{f}(t,\boldsymbol{y}))^T (\boldsymbol{x} - \boldsymbol{y}) \leq 0 \tag{76}$$

implies for all $h \geq 0$

$$\|\boldsymbol{x}_1 - \hat{\boldsymbol{x}}_1\| \leq \|\boldsymbol{x}_0 - \hat{\boldsymbol{x}}_0\|, \tag{77}$$

where $\boldsymbol{x}_1, \hat{\boldsymbol{x}}_1$ *are the numerical approximations after one step and* $\boldsymbol{x}_0, \hat{\boldsymbol{x}}_0$ *the corresponding initial values.[42]*

Definition 3.13 (L-stable)**.** *A Runge-Kutta method is called L-stable if it is A-stable and in addition*

$$\lim_{z \to \infty} R(z) = 0. \tag{78}$$

L-stability is the property to very quickly dampen out the effects of extremely stiff parts and B-stability is concerned with whether the Runge-Kutta method inherits certain characteristics of the differential equation. The Gauss-Legendre class is only A and B-stable, while Radau IIA methods are A, B and L-stable, which makes them suitable even for highly stiff problems.[41][42]

4 Nonlinear Optimization

This chapter offers an outline of optimality conditions and relevant algorithms in nonlinear optimization. The principle workflow of the Sequential Quadratic Progamming (SQP) and the Interior-Point Method (IP) are presented in this context. Based on the *filter line-search interior-point method* implemented in the open source nonlinear optimizer *Ipopt*, the large-scale NLP resulting from the direct collocation approach is solved. Therefore, a brief overview of the C++ Ipopt interface is also included. The following general NLP is considered throughout this section.

Definition 4.1 (Nonlinear Optimization Problem (NLP))**.** *The optimization problem*

$$\begin{aligned} &\min_{\boldsymbol{x}} f(\boldsymbol{x}) \\ &\text{s.t.} \\ &\boldsymbol{h}(\boldsymbol{x}) = \mathbf{0} \\ &\boldsymbol{c}(\boldsymbol{x}) \leq \mathbf{0}, \end{aligned} \tag{79}$$

with $\boldsymbol{x} \in \mathbb{R}^n$ and continuously differentiable functions $f : \mathbb{R}^n \to \mathbb{R}, \boldsymbol{h} : \mathbb{R}^n \to \mathbb{R}^p, \boldsymbol{c} : \mathbb{R}^n \to \mathbb{R}^m$ is called nonlinear optimization problem (NLP).[45]

4.1 Necessary and Sufficient Optimality Conditions

In order to obtain necessary and sufficient conditions that an optimal solution of the general NLP (Definition 4.1) must satisfy, several concepts are introduced. For some point $\boldsymbol{x}^* \in \mathbb{R}^n$ an inequality constraint $c_i(\boldsymbol{x}^*) \leq 0$ can be either $c_i(\boldsymbol{x}^*) = 0$ or $c_i(\boldsymbol{x}^*) < 0$. In the case of $c_i(\boldsymbol{x}^*) < 0$ the constraint is (by continuity of c_i) also satisfied in a region around $\boldsymbol{x}^*$. Because this does not apply to $c_i(\boldsymbol{x}^*) = 0$, it leads to the splitting of inequality constraints into *active* and *inactive* constraints.

Definition 4.2 (Active Constraint, Active Set)**.** *Let $\boldsymbol{c} : \mathbb{R}^n \to \mathbb{R}^m$ define the set $X = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{c}(\boldsymbol{x}) \leq \mathbf{0}\}$ and given a point $\boldsymbol{x}^* \in X$. For each $i = 1, \ldots, m$ the constraint c_i is said to be active at $\boldsymbol{x}^*$ if $c_i(\boldsymbol{x}^*) = 0$ and is said to be inactive at $\boldsymbol{x}^*$ if $c_i(\boldsymbol{x}^*) < 0$. Furthermore, the set of actives constraints at $\boldsymbol{x}^*$ is denoted by*

$$A(\boldsymbol{x}^*) := \{i \mid c_i(\boldsymbol{x}^*) = 0\}. \tag{80}$$

Note that not every local optimum satisfies the *Karush-Kuhn-Tucker (KKT)* conditions, which are necessary conditions for optimality introduced in Theorem 4.1. Therefore, the notion of *regularity* is needed. A *Constrained Qualification (CQ)* is an additional condition that, if satisfied by a point, makes the point *regular* and thus the KKT conditions necessary conditions. Such a condition guarantees that the shape of the feasible region in a neighborhood of a point is captured by linear approximations of the feasible region. One possible CQ is the *Linear Independence Constrained Qualification (LICQ)*. This states that a point is regular, if the gradients of the equality and active inequality constraints are linearly independent.

Definition 4.3 (Regular Point, LICQ)**.** *Let $\boldsymbol{h} : \mathbb{R}^n \to \mathbb{R}^p$ and $\boldsymbol{c} : \mathbb{R}^n \to \mathbb{R}^m$ be continuously differentiable and $X = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{h}(\boldsymbol{x}) = \mathbf{0} \wedge \boldsymbol{c}(\boldsymbol{x}) \leq \mathbf{0}\}$ be the set of feasible points for Definition 4.1. A point $\boldsymbol{x}^* \in X$ is a regular point of the constraints if the gradients $\nabla h_k(\boldsymbol{x}^*), k = 1, \ldots, p$, and $\nabla c_i(\boldsymbol{x}^*), i \in A(\boldsymbol{x}^*)$, are linearly independent.*

Theorem 4.1 (First- and Second-Order Necessary Conditions)**.** *Given a NLP (Definition 4.1) with twice continuously differentiable $f : \mathbb{R}^n \to \mathbb{R}, \boldsymbol{h} : \mathbb{R}^n \to \mathbb{R}^p, \boldsymbol{c} : \mathbb{R}^n \to \mathbb{R}^m$. If $\boldsymbol{x}^*$ is a local minimum and a regular point of the constraints, then there exist unique vectors $\boldsymbol{\lambda}^* \in \mathbb{R}^p$ and $\boldsymbol{\mu}^* \in \mathbb{R}^m$ such that the Karush-Kuhn-Tucker (KKT) conditions*

$$\nabla f(\boldsymbol{x}^*) + \nabla \boldsymbol{h}(\boldsymbol{x}^*)^T \boldsymbol{\lambda}^* + \nabla \boldsymbol{c}(\boldsymbol{x}^*)^T \boldsymbol{\mu}^* = \mathbf{0} \tag{81a}$$

$$\boldsymbol{h}(\boldsymbol{x}^*) = \mathbf{0} \tag{81b}$$

$$\boldsymbol{c}(\boldsymbol{x}^*) \leq \mathbf{0} \tag{81c}$$

$$\boldsymbol{\mu}^* \geq \mathbf{0} \tag{81d}$$

$$(\boldsymbol{\mu}^*)^T \boldsymbol{c}(\boldsymbol{x}^*) = \mathbf{0} \tag{81e}$$

hold. In addition

$$\boldsymbol{y}^T \left(\nabla^2 f(\boldsymbol{x}^*) + \nabla^2 \boldsymbol{h}(\boldsymbol{x}^*)^T \boldsymbol{\lambda}^* + \nabla^2 \boldsymbol{c}(\boldsymbol{x}^*)^T \boldsymbol{\mu}^* \right) \boldsymbol{y} \geq 0, \tag{82}$$

for all $\boldsymbol{y} \in \mathbb{R}^n$ such that $\nabla \boldsymbol{h}(\boldsymbol{x}^*)^T \boldsymbol{y} = \mathbf{0}$ and $\nabla c_i(\boldsymbol{x}^*)^T \boldsymbol{y} = 0, i \in A(\boldsymbol{x}^*)$.

Proof. see [45]. □

Condition (81a) is called *stationarity*, (81b) and (81c) denote *primal feasibility*, (81d) *dual feasibility* and (81e) are the *complementary conditions*. These conditions are very important for designing efficient numerical algorithms to solve NLPs. However, they are only necessary conditions and not every point that satisfies (81a) - (81e) and (82) is a local optimum. In general, no statement can be made about global optima. Nevertheless, in the special case that the NLP is a convex optimization problem, i.e. f, c_i are convex and h_i are affine, these conditions are even sufficient for a global optimum.[45]
Now the *Lagrangian* of a NLP is introduced. It allows compact representations of the first- and second-order necessary conditions.

Definition 4.4 (Lagrangian)**.** *Given a NLP (Definition 4.1) with $f : \mathbb{R}^n \to \mathbb{R}, \boldsymbol{h} : \mathbb{R}^n \to \mathbb{R}^p, \boldsymbol{c} : \mathbb{R}^n \to \mathbb{R}^m$, the Lagrangian $\mathcal{L} : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^m \to \mathbb{R}$ is defined as*

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) := f(\boldsymbol{x}) + \boldsymbol{\lambda}^T \boldsymbol{h}(\boldsymbol{x}) + \boldsymbol{\mu}^T \boldsymbol{c}(\boldsymbol{x}), \tag{83}$$

with $\boldsymbol{\lambda} \in \mathbb{R}^p$ and $\boldsymbol{\mu} \in \mathbb{R}^m$.

Thus, (81a) can be written as $\nabla_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) = \mathbf{0}$ and and (82) as $\boldsymbol{y}^T \nabla^2_{\boldsymbol{xx}} \mathcal{L}(\boldsymbol{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) \boldsymbol{y} \geq 0$.
There are also sufficient conditions regarding the local optimality of a point. If a point is a KKT point and in addition the reduced Hessian of the Lagrangian $\nabla^2_{\boldsymbol{xx}} \mathcal{L}$ is positive definite on the *tangent space* of the active constraints (85a), (85b), (85c), then the point is a strict local minimum.

Theorem 4.2 (Second-Order Sufficient Conditions)**.** *Given a NLP (Definition 4.1) with twice continuously differentiable $f : \mathbb{R}^n \to \mathbb{R}, \boldsymbol{h} : \mathbb{R}^n \to \mathbb{R}^p, \boldsymbol{c} : \mathbb{R}^n \to \mathbb{R}^m$. If there exist $\boldsymbol{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*$ satisfying the KKT conditions* (81a)*,* (81b)*,* (81c)*,* (81d)*,* (81e) *and*

$$\boldsymbol{y}^T \nabla^2_{\boldsymbol{xx}} \mathcal{L}(\boldsymbol{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*) \boldsymbol{y} > 0, \tag{84}$$

for all $\boldsymbol{y} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$ *such that*

$$\nabla c_i(\boldsymbol{x}^*)^T \boldsymbol{y} = 0,\ i \in A(\boldsymbol{x}^*)\ \textit{with}\ \mu_i^* > 0, \tag{85a}$$

$$\nabla c_i(\boldsymbol{x}^*)^T \boldsymbol{y} \leq 0,\ i \in A(\boldsymbol{x}^*)\ \textit{with}\ \mu_i^* = 0, \tag{85b}$$

$$\nabla \boldsymbol{h}(\boldsymbol{x}^*)^T \boldsymbol{y} = \mathbf{0}, \tag{85c}$$

then $\boldsymbol{x}^*$ *is a strict local minimum.*

Proof. see [45]. □

4.2 Sequential Quadratic Programming

This section presents a basic overview of *sequential quadratic programming (SQP)*, which is one of the most effective methods for numerical solutions to NLPs. SQP methods are implemented in many available solvers such as *SNOPT*[25][26] or *KNITRO*[28]. To motivate the approach, it is useful to recall *Newton's method* for solving an algebraic system of equations.

Algorithm 4.1: Newton's Method [45]

Input: $\boldsymbol{F} : \mathbb{R}^n \to \mathbb{R}^n, \boldsymbol{F} \in C, \boldsymbol{x}_0 \in \mathbb{R}^n, \varepsilon > 0$

Output: $\boldsymbol{x}^*$ with $\|\boldsymbol{F}(\boldsymbol{x}^*)\|_\infty \leq \varepsilon$

```
k ← 0;
while ‖F(x*)‖_∞ > ε do
    Solve ∇F(x_k) d_k = −F(x_k);
    x_{k+1} ← x_k + d_k;
    k ← k + 1;
end
return x_k;
```

Newton's method solves the algebraic system of equations $\boldsymbol{F}(\boldsymbol{x}) = \mathbf{0}$ by finding a search direction $\boldsymbol{d}_k$ that solves $\nabla \boldsymbol{F}(\boldsymbol{x}_k)\boldsymbol{d}_k = -\boldsymbol{F}(\boldsymbol{x}_k)$ in every step. The formula can be derived by doing a simple Taylor expansion $\mathbf{0} = \boldsymbol{F}(\boldsymbol{x}_k + \boldsymbol{d}_k) \approx \boldsymbol{F}(\boldsymbol{x}_k) + \nabla \boldsymbol{F}(\boldsymbol{x}_k)\boldsymbol{d}_k + O(\|\boldsymbol{d}_k\|^2)$. The method has quadratic convergence if the initial guess $\boldsymbol{x}_0$ is sufficiently "good". In addition, there is also the *damped Newton's method*, which does not perform a full step $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{d}_k$, but rather updates the solution as $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$ with $\alpha_k \in (0, 1)$.[45]

For the SQP method, a problem with only equality constraints is considered, since it allows an easier interpretation of the method.

$$\begin{aligned} &\min_{\boldsymbol{x}} f(\boldsymbol{x}) \\ &\text{s.t.} \\ &\boldsymbol{h}(\boldsymbol{x}) = \mathbf{0} \end{aligned} \tag{86}$$

It is assumed that CQ are satisfied. Thus, the KKT conditions are necessary for optimality. Because the

problem contains no inequalities, the KKT conditions reduce to (81a) and (81b) and yield

$$\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{\lambda}) := \begin{pmatrix} \nabla f(\boldsymbol{x}) + \nabla \boldsymbol{h}(\boldsymbol{x}^*)^T \boldsymbol{\lambda} \\ \boldsymbol{h}(\boldsymbol{x}) \end{pmatrix} = \boldsymbol{0}. \tag{87}$$

The Lagrangian of the problem is given by $\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) := f(\boldsymbol{x}) + \boldsymbol{\lambda}^T \boldsymbol{h}(\boldsymbol{x})$. Calculating the Jacobian of $\boldsymbol{F}$ results in

$$\nabla \boldsymbol{F}(\boldsymbol{x}, \boldsymbol{\lambda}) = \begin{pmatrix} \nabla^2_{\boldsymbol{xx}} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}) & \nabla \boldsymbol{h}(\boldsymbol{x})^T \\ \nabla \boldsymbol{h}(\boldsymbol{x}) & 0 \end{pmatrix}. \tag{88}$$

Applying Newton's method (Algorithm 4.1) to the KKT conditions (87) yields the step calculations

$$\begin{pmatrix} \boldsymbol{x}_{k+1} \\ \boldsymbol{\lambda}_{k+1} \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}_k \\ \boldsymbol{\lambda}_k \end{pmatrix} + \begin{pmatrix} \boldsymbol{d}^{\boldsymbol{x}}_k \\ \boldsymbol{d}^{\boldsymbol{\lambda}}_k \end{pmatrix}. \tag{89}$$

The search directions $\boldsymbol{d}^{\boldsymbol{x}}_k, \boldsymbol{d}^{\boldsymbol{\lambda}}_k$ are obtained by solving the system

$$\begin{pmatrix} \nabla^2_{\boldsymbol{xx}} \mathcal{L}(\boldsymbol{x}_k, \boldsymbol{\lambda}_k) & \nabla \boldsymbol{h}(\boldsymbol{x}_k)^T \\ \nabla \boldsymbol{h}(\boldsymbol{x}_k) & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{d}^{\boldsymbol{x}}_k \\ \boldsymbol{d}^{\boldsymbol{\lambda}}_k \end{pmatrix} = \begin{pmatrix} -\nabla f(\boldsymbol{x}_k) - \nabla \boldsymbol{h}(\boldsymbol{x}_k)^T \boldsymbol{\lambda}_k \\ -\boldsymbol{h}(\boldsymbol{x}_k) \end{pmatrix}. \tag{90}$$

Note that the KKT matrix (90) is symmetric. Therefore specific solvers that exploit this structure can be applied. It can also be shown that the matrix is nonsingular, if the LICQ is met and the current solution is close to the optimum. The update step for $\boldsymbol{\lambda}_{k+1}$ can be removed by substituting $\boldsymbol{d}^{\boldsymbol{\lambda}}_k = \boldsymbol{\lambda}_{k+1} - \boldsymbol{\lambda}_k$ and thus

$$\begin{pmatrix} \nabla^2_{\boldsymbol{xx}} \mathcal{L}(\boldsymbol{x}_k, \boldsymbol{\lambda}_k) & \nabla \boldsymbol{h}(\boldsymbol{x}_k)^T \\ \nabla \boldsymbol{h}(\boldsymbol{x}_k) & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{d}^{\boldsymbol{x}}_k \\ \boldsymbol{\lambda}_{k+1} \end{pmatrix} = \begin{pmatrix} -\nabla f(\boldsymbol{x}_k) \\ -\boldsymbol{h}(\boldsymbol{x}_k) \end{pmatrix}. \tag{91}$$

This system of equations actually defines the first-order necessary (KKT) conditions of the quadratic problem

$$\begin{aligned} \min_{\boldsymbol{d}} \; & f(\boldsymbol{x}_k) + \nabla f(\boldsymbol{x}_k)^T \boldsymbol{d} + \frac{1}{2} \boldsymbol{d}^T \nabla^2_{\boldsymbol{xx}} \mathcal{L}(\boldsymbol{x}_k, \boldsymbol{\lambda}_k) \boldsymbol{d} \\ \text{s.t.} \; & \\ & \boldsymbol{h}(\boldsymbol{x}_k) + \nabla \boldsymbol{h}(\boldsymbol{x}_k)^T \boldsymbol{d} = \boldsymbol{0}. \end{aligned} \tag{92}$$

The Lagrange multiplier and the search direction can thus be considered as the solution to the subproblem (92) or the linear system (91). It is easy to generalize the method by adding inequality constraints to the subproblem, i.e.

$$\begin{aligned} \min_{\boldsymbol{d}} \; & f(\boldsymbol{x}_k) + \nabla f(\boldsymbol{x}_k)^T \boldsymbol{d} + \frac{1}{2} \boldsymbol{d}^T \nabla^2_{\boldsymbol{xx}} \mathcal{L}(\boldsymbol{x}_k, \boldsymbol{\lambda}_k, \boldsymbol{\mu}_k) \boldsymbol{d} \\ \text{s.t.} \; & \\ & \boldsymbol{h}(\boldsymbol{x}_k) + \nabla \boldsymbol{h}(\boldsymbol{x}_k)^T \boldsymbol{d} = \boldsymbol{0} \\ & \boldsymbol{c}(\boldsymbol{x}_k) + \nabla \boldsymbol{c}(\boldsymbol{x}_k)^T \boldsymbol{d} \leq \boldsymbol{0} \end{aligned} \tag{93}$$

General NLPs can be solved in this way.[45]

Algorithm 4.2: Prototype Sequential Quadratic Programming (Equality Constraints) [45]

Input: NLP of the form (86), $\boldsymbol{x}_0, \boldsymbol{\lambda}_0$

Output: $\boldsymbol{x}^*$

$k \leftarrow 0$;
while *stopping condition is not met* **do**
  Evaluate $f(\boldsymbol{x}_k), \nabla f(\boldsymbol{x}_k), \boldsymbol{h}(\boldsymbol{x}_k), \nabla \boldsymbol{h}(\boldsymbol{x}_k), \nabla^2_{\boldsymbol{xx}} \mathcal{L}(\boldsymbol{x}_k, \boldsymbol{\lambda}_k)$;
  Solve the subproblem (92) or the linear system (91) and obtain $\boldsymbol{d}^{\boldsymbol{x}}_k, \boldsymbol{\lambda}_{k+1}$;
  $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \boldsymbol{d}^{\boldsymbol{x}}_k$;
  $k \leftarrow k + 1$;
end
return $\boldsymbol{x}_k$;

The resulting algorithm is only a very primitive prototype of real SQP solvers. These perform line searches and second-order corrections, use merit functions and are able to handle infeasible subproblems. These advanced topics are beyond the scope of this thesis.[45]

4.3 Interior-Point Methods

Another important class of numerical NLP solvers are so-called *interior-point methods*. These methods introduce barriers to penalize infeasible solutions and reward feasibility. These have shown to be very robust, offer desirable local as well as global convergence and are well suited for large-scale nonlinear problems. Again, only the principle algorithm is outlined in this section. Consider the following NLP:

$$\begin{aligned} &\min_{\boldsymbol{x}} f(\boldsymbol{x}) \\ &\text{s.t.} \\ &\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0} \\ &\boldsymbol{x} \geq \boldsymbol{0} \end{aligned} \tag{94}$$

It is clear that this problem is equivalent to Definition 4.1, since each inequality constraint $c_i(\boldsymbol{x}) \leq 0$ can be rewritten as $c_i(\boldsymbol{x}) + s = 0$ by introducing a slack variable $s \geq 0$. In interior-point methods the problem (94) is transformed into a so-called *barrier problem*, which depends on a scalar parameter $\mu \geq 0$. This problem contains logarithmic barriers to penalize variables that are too close to the boundary. The principle idea of interior-point methods is to solve the barrier problem iteratively for a decreasing sequence $(\mu_j)_{j \in \mathbb{N}_0}$ and thus obtain a feasible and locally optimal solution.[1]

$$\begin{aligned} &\min_{\boldsymbol{x}} \phi_\mu(\boldsymbol{x}) := f(\boldsymbol{x}) - \mu \sum_{i=1}^{n} \ln(x_i) \\ &\text{s.t.} \\ &\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0} \end{aligned} \tag{95}$$

Adding the dual variables $z_i = \mu/x_i, i = 1, \dots, n$ and applying the KKT conditions (81a) and (81b) to the barrier problem (95), the system

$$\begin{aligned} \nabla f(\boldsymbol{x}) + \nabla \boldsymbol{h}(\boldsymbol{x})^T \boldsymbol{\lambda} - \boldsymbol{z} &= \mathbf{0} \\ \boldsymbol{h}(\boldsymbol{x}) &= \mathbf{0} \\ XZ\mathbf{1} - \mu\mathbf{1} &= \mathbf{0} \end{aligned} \tag{96}$$

with $X := \operatorname{diag}(\boldsymbol{x}), Z := \operatorname{diag}(\boldsymbol{z})$ is obtained. An alternative, elegant interpretation of the system is that it represents a homotopy method applied to the KKT conditions of the original problem (94). Using the damped Newton's method (Algorithm 4.1), the system can be solved by calculating

$$\begin{pmatrix} \boldsymbol{x}_{k+1} \\ \boldsymbol{\lambda}_{k+1} \\ \boldsymbol{z}_{k+1} \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}_k \\ \boldsymbol{\lambda}_k \\ \boldsymbol{z}_k \end{pmatrix} + \begin{pmatrix} \alpha_k \boldsymbol{d}_k^{\boldsymbol{x}} \\ \alpha_k \boldsymbol{d}_k^{\boldsymbol{\lambda}} \\ \alpha_k^{\boldsymbol{z}} \boldsymbol{d}_k^{\boldsymbol{z}} \end{pmatrix}. \tag{97}$$

The search directions are the solution to the linear system

$$\begin{pmatrix} W_k & \nabla \boldsymbol{h}(\boldsymbol{x}_k)^T & -I \\ \nabla \boldsymbol{h}(\boldsymbol{x}_k) & 0 & 0 \\ Z_k & 0 & X_k \end{pmatrix} \begin{pmatrix} \boldsymbol{d}_k^{\boldsymbol{x}} \\ \boldsymbol{d}_k^{\boldsymbol{\lambda}} \\ \boldsymbol{d}_k^{\boldsymbol{z}} \end{pmatrix} = - \begin{pmatrix} \nabla f(\boldsymbol{x}_k) + \nabla \boldsymbol{h}(\boldsymbol{x}_k)^T \boldsymbol{\lambda}_{\boldsymbol{k}} - \boldsymbol{z}_{\boldsymbol{k}} \\ \boldsymbol{h}(\boldsymbol{x}_k) \\ X_k Z_k \mathbf{1} - \mu \mathbf{1} \end{pmatrix} =: \boldsymbol{F}(\boldsymbol{x}_k, \boldsymbol{\lambda}_k, \boldsymbol{z}_k), \tag{98}$$

where $W_k = \nabla^2_{\boldsymbol{x}\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{z}) = \nabla^2 f(\boldsymbol{x}_k) + \nabla^2 \boldsymbol{h}(\boldsymbol{x}_k)^T \boldsymbol{\lambda}_k$ is the Hessian of the Lagrangian for the original problem. The resulting matrix is obviously nonsymmetric. By adding X_k^{-1} multiplied with the last row to the first row, the system can be split into a smaller symmetric system

$$\begin{pmatrix} W_k + \Sigma_k & \nabla \boldsymbol{h}(\boldsymbol{x}_k)^T \\ \nabla \boldsymbol{h}(\boldsymbol{x}_k) & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{d}_k^{\boldsymbol{x}} \\ \boldsymbol{d}_k^{\boldsymbol{\lambda}} \end{pmatrix} = - \begin{pmatrix} \nabla \phi_\mu(\boldsymbol{x}_k) + \nabla \boldsymbol{h}(\boldsymbol{x}_k)^T \boldsymbol{\lambda}_{\boldsymbol{k}} \\ \boldsymbol{h}(\boldsymbol{x}_k) \end{pmatrix}, \tag{99}$$

because $\nabla \phi_\mu(\boldsymbol{x}) = \nabla f(\boldsymbol{x}) - \mu X_k^{-1} \mathbf{1}$, and with $\Sigma_k = X_k^{-1} Z_k$. The search direction $\boldsymbol{d}_k^{\boldsymbol{z}}$ is then evaluated by

$$\boldsymbol{d}_k^{\boldsymbol{z}} = \mu X_k^{-1} \mathbf{1} - z_k - \Sigma_k \boldsymbol{d}_k^{\boldsymbol{x}}. \tag{100}$$

Given a fixed $\mu > 0$, the Newton steps (99), (100), (97) are usually performed until the error of the nonlinear algebraic system (96) is below some threshold $\varepsilon > 0$. Only then the parameter μ is decreased and the process repeated. In this way, a very stable convergence to an optimum can be achieved, while remaining feasible due to the logarithmic barriers. The step sizes of Newton's method α_k and $\alpha_k^{\boldsymbol{z}}$ are typically determined by performing a line search. Nevertheless, the Algorithm 4.3, which is described in this thesis, does not contain such a routine. As before, the presented algorithm is just a prototype version of an interior-point method that lacks most core features efficient implementations offer, although the principle approach is the same to state-of-the-art solvers such as Ipopt.[1]

Algorithm 4.3: Prototype Interior-Point Method [45]

Input: NLP of the form (94), $\boldsymbol{x}_0, \boldsymbol{\lambda}_0, \boldsymbol{z}_0, \mu_0 > 0, \sigma \in (0,1)$, a decreasing sequence $\varepsilon_k > 0$

Output: $\boldsymbol{x}^*$

$k \leftarrow 0$;
while *stopping condition is not met* **do**
  $j \leftarrow 0$;
  $\boldsymbol{x}_{k,0}, \boldsymbol{\lambda}_{k,0}, \boldsymbol{z}_{k,0} \leftarrow \boldsymbol{x}_k, \boldsymbol{\lambda}_k, \boldsymbol{z}_k$;
  while $\left\|\boldsymbol{F}(\boldsymbol{x}_{k,j}, \boldsymbol{\lambda}_{k,j}, \boldsymbol{z}_{k,j})\right\|_\infty > \varepsilon_k$ **do**
    Solve the systems (99), (100) using the point $\boldsymbol{x}_{k,j}, \boldsymbol{\lambda}_{k,j}, \boldsymbol{z}_{k,j}$ and obtain $\boldsymbol{d}^{\boldsymbol{x}}_{k,j}, \boldsymbol{d}^{\boldsymbol{\lambda}}_{k,j}, \boldsymbol{d}^{\boldsymbol{z}}_{k,j}$;
    Update $\boldsymbol{x}_{k,j}, \boldsymbol{\lambda}_{k,j}, \boldsymbol{z}_{k,j}$ with $\boldsymbol{d}^{\boldsymbol{x}}_{k,j}, \boldsymbol{d}^{\boldsymbol{\lambda}}_{k,j}, \boldsymbol{d}^{\boldsymbol{z}}_{k,j}$ according to (97);
    $j \leftarrow j + 1$;
  end
  $\mu_{k+1} \leftarrow \sigma\mu_k$;
  $k \leftarrow k + 1$;
end
return $\boldsymbol{x}_k$;

4.3.1 Ipopt

The NLP solver *Ipopt (Interior Point OPTimizer)*[1] is a state-of-the-art EPL 2.0 licensed open source software package for large-scale nonlinear optimization. Ipopt implements a primal-dual interior-point filter line search algorithm that provides feasibility restoration as well as second-order and inertia corrections. The algorithm has proven to be very efficient and is considered standard software in the field of nonlinear optimization. The package uses the coordinate format (COO) to efficiently process and handle sparse Jacobians and Hessians. There are several linear solvers available that can be used by Ipopt, e.g. the free to use default option *MUMPS*[2] or the proprietary *HSL*[3] solvers (*MA27*, *MA57*, *MA77*, *MA86*, *MA97*). Ipopt is used in many collocation-based dynamic optimization implementations such as *OpenModelica*[17], *JModelica*[16], *GPOPS II*[30], *PSOPT*[27] or *SPARTAN*[49].

The software package has a C++ interface that won the *2011 J. H. Wilkinson Prize for Numerical Software*. This interface is used in the proposed framework *GDOPT*. For this reason, a brief overview of the interface is provided in the following. The NLP formulation that Ipopt requires as input is

$$\begin{aligned} &\min_{\boldsymbol{x}} f(\boldsymbol{x}) \\ &\text{s.t.} \\ &\quad \boldsymbol{g}^L \leq \boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{g}^U \\ &\quad \boldsymbol{x}^L \leq \boldsymbol{x} \leq \boldsymbol{x}^U \end{aligned} \tag{101}$$

with $\boldsymbol{x} \in \mathbb{R}^n$ and twice continuously differentiable functions $f : \mathbb{R}^n \to \mathbb{R}, \boldsymbol{g} : \mathbb{R}^n \to \mathbb{R}^m$ and $\boldsymbol{x}^L \in \{\mathbb{R} \cup \{-\infty\}\}^n, \boldsymbol{x}^U \in \{\mathbb{R} \cup \{\infty\}\}^n, \boldsymbol{g}^L \in \{\mathbb{R} \cup \{-\infty\}\}^m, \boldsymbol{g}^U \in \{\mathbb{R} \cup \{\infty\}\}^m$. Additionally, equality constraints can be modeled by setting the same lower and upper bounds. This NLP formulation is very general, and

no reformulations are needed to model standard NLPs, as in the Definition 4.1.

The C++ interface offers a virtual base class `Ipopt::TNLP` with 8 essential purely virtual methods: `get_nlp_info`, `get_bounds_info`, `get_starting_point`, `eval_f`, `eval_grad_f`, `eval_g`, `eval_jac_g`, and `eval_h`. When modeling a NLP with this interface, a derived class for the problem must be created that implements all of these methods. The only exception is if a *quasi-Newton / BFGS Hessian approximation* is used. In this case, the method `eval_h` is not needed.[10]

It is refrained from presenting the methods with detailed inputs and outputs. These can be found in the Ipopt documentation.[10] However, all problem defining information that is required by Ipopt will be listed as in the documentation:

1. Problem dimensions
    1. number of variables: n
    2. number of constraints: m
2. Problem bounds
    1. variable bounds: $\boldsymbol{x}^L, \boldsymbol{x}^U$
    2. constraint bounds: $\boldsymbol{g}^L, \boldsymbol{g}^U$
3. Initial starting point
    1. initial values for the primal variables: $\boldsymbol{x}_0$
    2. initial values for the multipliers (only required for a warm start option): $\boldsymbol{\lambda}_0$
4. Problem Structure
    1. number of nonzeros in the Jacobian of the constraints: $nnz\left(\nabla \boldsymbol{g}(\boldsymbol{x})\right)$
    2. number of nonzeros in the Hessian of the Lagrangian: $nnz\left(\sigma_f \nabla^2 f(\boldsymbol{x}) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(\boldsymbol{x})\right)$
    3. sparsity structure of the Jacobian of the constraints
    4. sparsity structure of the Hessian of the Lagrangian (only lower triangular part)
5. Evaluation of Problem Functions based on a given point $(\boldsymbol{x}, \boldsymbol{\lambda}, \sigma_f)$
    1. objective function: $f(\boldsymbol{x})$
    2. gradient of the objective: $\nabla f(\boldsymbol{x})$
    3. constraint function values: $\boldsymbol{g}(\boldsymbol{x})$
    4. Jacobian of the constraints: $\nabla \boldsymbol{g}(\boldsymbol{x})$
    5. Hessian of the Lagrangian: $\sigma_f \nabla^2 f(\boldsymbol{x}) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(\boldsymbol{x})$

The Jacobian and Hessian sparsity patterns are usually provided in the very first evaluation of the matrices. These sparse matrices are implemented as coordinate format arrays: `int* iRow, int* jCol, double* values`. Additionally, the Hessian of the Lagrangian contains an additional factor σ_f weighting the Hessian of the objective. By setting $\sigma_f = 0$, Ipopt can solely ask for the Hessian of the constraints, if needed.[10][1]

5 Discretization of the GDOP

With all the preliminary work done, the continuous GDOP (Definition 2.2) can be discretized using an orthogonal direct collocation approach. For this, the class of Radau IIA collocation schemes is used, because they have excellent stability and accuracy as presented in Chapter 3.4.1. Furthermore, Radau collocation approaches are widely used in existing dynamic optimization frameworks and have shown exceptional results when applied in dynamic optimization.[30][46][47][17] The large-scale NLP resulting from the discretization is then solved using Ipopt (Chapter 4.3.1). Therefore, the NLP must be of the form (101) and certain derivative information has to be provided. For this reason the required sparse derivatives are also calculated in this chapter.

5.1 Transcription with Direct Collocation

Firstly, recall the formulation of the GDOP (Definition 2.2)

$$
\begin{aligned}
&\min_{\boldsymbol{u}(t),\boldsymbol{p}} M(\boldsymbol{x}(t_f),\boldsymbol{u}(t_f),\boldsymbol{p},t_f) + \int_{t_0}^{t_f} L(\boldsymbol{x}(t),\boldsymbol{u}(t),\boldsymbol{p},t)\,\mathrm{d}t \\
&\quad \text{s.t.} \\
&\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t),\boldsymbol{u}(t),\boldsymbol{p},t)\ \forall t \in [t_0,t_f] \\
&\boldsymbol{x}(t_0) = \boldsymbol{x}_0 \\
&\quad \boldsymbol{g}^L \le \boldsymbol{g}(\boldsymbol{x}(t),\boldsymbol{u}(t),\boldsymbol{p},t) \le \boldsymbol{g}^U\ \forall t \in [t_0,t_f] \\
&\quad \boldsymbol{r}^L \le \boldsymbol{r}(\boldsymbol{x}(t_f),\boldsymbol{u}(t_f),\boldsymbol{p},t_f) \le \boldsymbol{r}^U \\
&\quad \boldsymbol{a}^L \le \boldsymbol{a}(\boldsymbol{p}) \le \boldsymbol{a}^U.
\end{aligned}
$$

The time horizon $[t_0,t_f]$ is divided into n subintervals $[t_i,t_{i+1}]$ for $i=0,\dots,n$. The concrete placement of intervals will be examined in the following chapter on mesh refinement. For now, it is sufficient to assume that the nodes t_i are all equidistantly distributed. In each subinterval m collocation nodes $t_{i,j} = t_i + \Delta t_i c_j$ for $j=1,\dots,m$ with $\Delta t_i = t_{i+1} - t_i$ are introduced. Since the presented framework has a strong focus on local collocation techniques, in this formulation the number of collocation nodes is constant for all intervals. However, the subsequent considerations are also possible with varying numbers of nodes per interval as in pseudospectral collocation methods. The collocation nodes c_j are the *flipped Legendre-Gauss-Radau* (fLGR) points rescaled to the interval $[0,1]$. As shown in Chapter 3.2.2.1 and Chapter 3.4.1, these are the m roots of $(1-t)P_{m-1}^{(1,0)}(t)$ rescaled from $[-1,1]$ to $[0,1]$, i.e. the nodes c_j satisfy $(1-c_j)P_{m-1}^{(1,0)}(2c_j-1)=0$, where $P_{m-1}^{(1,0)}$ is the $(m-1)$-th Jacobi polynomial with $\alpha=1,\beta=0$. The state and input variables $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$ are discretized at the nodes as $\boldsymbol{x}(t_{i,j}) \approx \boldsymbol{x}_{i,j}$ and $\boldsymbol{u}(t_{i,j}) \approx \boldsymbol{u}_{i,j}$ for $i=0,\dots,n, j=1,\dots,m$. Note that the node $c_j=1$ is always contained in the Radau IIA scheme and thus $\boldsymbol{x}(t_{i+1}) \approx \boldsymbol{x}_{i+1} = \boldsymbol{x}_{i,m}$ as well as $\boldsymbol{x}(t_f) = \boldsymbol{x}(t_{n+1}) \approx \boldsymbol{x}_{n,m}$. This fact is very important, because no additional variable must be introduced at the start of each interval. It is also noteworthy that the node 0 does not belong to the Radau IIA scheme. Consequently, the value of $\boldsymbol{u}(t_0)$ is not obtained in the optimization. The problem can be addressed by using a Lobatto scheme, which fixes the nodes $c_1=0$ and $c_m=1$ on the first interval as in [17] or by simply interpolating the control backwards. In this thesis the latter strategy is used. The parameters $\boldsymbol{p}$ are not discretized, since they are time-invariant by definition.

Combining all variables $\boldsymbol{x}_{i,j}, \boldsymbol{u}_{i,j}, \boldsymbol{p}$ results in a large vector of variables that are passed to Ipopt, i.e.

$$\boldsymbol{x}_{NLP} := \begin{pmatrix} \boldsymbol{x}_{0,1}, \boldsymbol{u}_{0,1}, \\ \vdots \\ \boldsymbol{x}_{0,m}, \boldsymbol{u}_{0,m}, \\ \boldsymbol{x}_{1,1}, \boldsymbol{u}_{1,1}, \\ \vdots \\ \boldsymbol{x}_{1,m}, \boldsymbol{u}_{1,m}, \\ \vdots \\ \boldsymbol{x}_{n,1}, \boldsymbol{u}_{n,1}, \\ \vdots \\ \boldsymbol{x}_{n,m}, \boldsymbol{u}_{n,m}, \\ \boldsymbol{p} \end{pmatrix} = \begin{pmatrix} x_{0,1}^{(1)}, x_{0,1}^{(2)}, \dots, x_{0,1}^{(d_x)}, u_{0,1}^{(1)}, u_{0,1}^{(2)}, \dots, u_{0,1}^{(d_u)}, \\ \vdots \\ x_{0,m}^{(1)}, x_{0,m}^{(2)}, \dots, x_{0,m}^{(d_x)}, u_{0,m}^{(1)}, u_{0,m}^{(2)}, \dots, u_{0,m}^{(d_u)}, \\ x_{1,1}^{(1)}, x_{1,1}^{(2)}, \dots, x_{1,1}^{(d_x)}, u_{1,1}^{(1)}, u_{1,1}^{(2)}, \dots, u_{1,1}^{(d_u)}, \\ \vdots \\ x_{1,m}^{(1)}, x_{1,m}^{(2)}, \dots, x_{1,m}^{(d_x)}, u_{1,m}^{(1)}, u_{1,m}^{(2)}, \dots, u_{1,m}^{(d_u)}, \\ \vdots \\ x_{n,1}^{(1)}, x_{n,1}^{(2)}, \dots, x_{n,1}^{(d_x)}, u_{n,1}^{(1)}, u_{n,1}^{(2)}, \dots, u_{n,1}^{(d_u)}, \\ \vdots \\ x_{n,m}^{(1)}, x_{n,m}^{(2)}, \dots, x_{n,m}^{(d_x)}, u_{n,m}^{(1)}, u_{n,m}^{(2)}, \dots, u_{n,m}^{(d_u)}, \\ p^{(1)}, p^{(2)}, \dots, p^{(d_p)} \end{pmatrix}, \tag{102}$$

where the raised index denotes the component of the variables. At first, the dynamic is discretized. Applying the simplified representation of the Radau IIA collocation scheme (73) to the dynamic of the GDOP yields

$$\begin{pmatrix} \boldsymbol{x}_{i,1} - \boldsymbol{x}_i \\ \vdots \\ \boldsymbol{x}_{i,m} - \boldsymbol{x}_i \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}_{i,1} - \boldsymbol{x}_{i-1,m} \\ \vdots \\ \boldsymbol{x}_{i,m} - \boldsymbol{x}_{i-1,m} \end{pmatrix} = \Delta t_i (A \otimes I) \begin{pmatrix} \boldsymbol{f}(\boldsymbol{x}_{i,1}, \boldsymbol{u}_{i,1}, \boldsymbol{p}, t_{i,1}) \\ \vdots \\ \boldsymbol{f}(\boldsymbol{x}_{i,m}, \boldsymbol{u}_{i,m}, \boldsymbol{p}, t_{i,m}) \end{pmatrix} \tag{103}$$

for $i = 0, \dots, n$. The value $\boldsymbol{x}_{-1,m}$ is not a variable of the problem, but rather the given starting value $\boldsymbol{x}_0$ that is provided for the specific GDOP. However, this formulation offers a poor sparsity pattern, because the sparse structure is destroyed when multiplied with the dense Butcher matrix A. Since the Butcher matrix A is invertible for Radau IIA schemes and furthermore $(A \otimes I)^{-1} = (A^{-1} \otimes I)$,

$$(A^{-1} \otimes I) \begin{pmatrix} \boldsymbol{x}_{i,1} - \boldsymbol{x}_{i-1,m} \\ \vdots \\ \boldsymbol{x}_{i,m} - \boldsymbol{x}_{i-1,m} \end{pmatrix} = \Delta t_i \begin{pmatrix} \boldsymbol{f}(\boldsymbol{x}_{i,1}, \boldsymbol{u}_{i,1}, \boldsymbol{p}, t_{i,1}) \\ \vdots \\ \boldsymbol{f}(\boldsymbol{x}_{i,m}, \boldsymbol{u}_{i,m}, \boldsymbol{p}, t_{i,m}) \end{pmatrix} \tag{104}$$

follows. Bringing the equation to residual form results in

$$(A^{-1} \otimes I) \begin{pmatrix} \boldsymbol{x}_{i,1} - \boldsymbol{x}_{i-1,m} \\ \vdots \\ \boldsymbol{x}_{i,m} - \boldsymbol{x}_{i-1,m} \end{pmatrix} - \Delta t_i \begin{pmatrix} \boldsymbol{f}(\boldsymbol{x}_{i,1}, \boldsymbol{u}_{i,1}, \boldsymbol{p}, t_{i,1}) \\ \vdots \\ \boldsymbol{f}(\boldsymbol{x}_{i,m}, \boldsymbol{u}_{i,m}, \boldsymbol{p}, t_{i,m}) \end{pmatrix} = \boldsymbol{0}. \tag{105}$$

Clearly, the representation (105) has a way favorable sparsity pattern. Now, not all variables of an interval i can be contained in a single equation, but rather those corresponding to the specific node j and the ones resulting from the matrix-vector multiplication. Equation (105) can be rewritten in the form

$$0 = \sum_{k=1}^{m} \tilde{a}_{jk} (\boldsymbol{x}_{i,k} - \boldsymbol{x}_{i-1,m}) - \Delta t_i \boldsymbol{f}(\boldsymbol{x}_{i,j}, \boldsymbol{u}_{i,j}, \boldsymbol{p}, t_{i,j}), \;\; i = 1, \dots, n, \, j = 1, \dots, m, \tag{106}$$

where $\tilde{a}_{jk}$ are the entries of the inverse Butcher matrix A^{-1}. In order to emphasize that $\boldsymbol{x}_0 = \boldsymbol{x}_{i-1,m}$ is not a variable, but rather a given starting value,

$$0 = \sum_{k=1}^{m} \tilde{a}_{jk}(\boldsymbol{x}_{0,k} - \boldsymbol{x}_0) - \Delta t_0 \boldsymbol{f}(\boldsymbol{x}_{0,j}, \boldsymbol{u}_{0,j}, \boldsymbol{p}, t_{0,j}), \;\; j = 1, \ldots, m, \tag{107}$$

is written for the first interval.[17]
The discretization of the path $\boldsymbol{g}^L \leq \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t) \leq \boldsymbol{g}^U \; \forall t \in [t_0, t_f]$ and final constraints $\boldsymbol{r}^L \leq \boldsymbol{r}(\boldsymbol{x}(t_f), \boldsymbol{u}(t_f), \boldsymbol{p}, t_f) \leq \boldsymbol{r}^U$ is straightforward. These algebraic constraints simply have to be satisfied at each discrete time $t_{i,j}$ on the time horizon. Therefore, the constraints become

$$\begin{aligned} &\boldsymbol{g}^L \leq \boldsymbol{g}(\boldsymbol{x}_{i,j}, \boldsymbol{u}_{i,j}, \boldsymbol{p}, t_{i,j}) \leq \boldsymbol{g}^U, \;\; \forall i = 0, \ldots, n \;\; \forall j = 1, \ldots, m \\ &\boldsymbol{r}^L \leq \boldsymbol{r}(\boldsymbol{x}_{n,m}, \boldsymbol{u}_{n,m}, \boldsymbol{p}, t_{n,m}) \leq \boldsymbol{r}^U. \end{aligned} \tag{108}$$

Since the parametric constraints $\boldsymbol{a}^L \leq \boldsymbol{a}(\boldsymbol{p}) \leq \boldsymbol{a}^U$ are time-invariant anyway, they are not affected by the discretization. The Mayer term $M(\boldsymbol{x}(t_f), \boldsymbol{u}(t_f), \boldsymbol{p}, t_f)$ of the objective function is, just like the final constraints, replaced by its discretized version and becomes

$$M(\boldsymbol{x}_{n,m}, \boldsymbol{u}_{n,m}, \boldsymbol{p}, t_{n,m}). \tag{109}$$

In order to discretize the Lagrange term $\int_{t_0}^{t_f} L(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t) \, \mathrm{d}t$, the Radau quadrature rule (Theorem 3.5) is applied on every interval i. It yields

$$\sum_{i=0}^{n} \Delta t_i \sum_{j=1}^{m} b_j L(\boldsymbol{x}_{i,j}, \boldsymbol{u}_{i,j}, \boldsymbol{p}, t_{i,j}), \tag{110}$$

where b_j are the rescaled weights of the Radau quadrature or equivalently the weights of the Radau IIA collocation scheme.[17] The combination of all discretized functions and constraints results in the *discretized General Dynamic Optimization Problem (dGDOP).*

Definition 5.1 (dGDOP)**.** *Given a GDOP (Definition 2.2) and a m-stage Radau IIA collocation scheme (Chapter 3.4.1), the direct collocation NLP with $\boldsymbol{x}^L \leq \boldsymbol{x}_{i,j} \leq \boldsymbol{x}^U, \boldsymbol{u}^L \leq \boldsymbol{u}_{i,j} \leq \boldsymbol{u}^U, \boldsymbol{p}^L \leq \boldsymbol{p} \leq \boldsymbol{p}^U$ given by*

$$\min_{\boldsymbol{x}_{i,j}, \boldsymbol{u}_{i,j}, \boldsymbol{p}} \; M(\boldsymbol{x}_{n,m}, \boldsymbol{u}_{n,m}, \boldsymbol{p}, t_{n,m}) + \sum_{i=0}^{n} \Delta t_i \sum_{j=1}^{m} b_j L(\boldsymbol{x}_{i,j}, \boldsymbol{u}_{i,j}, \boldsymbol{p}, t_{i,j})$$

s.t.

$$\begin{aligned} \mathbf{0} &= \sum_{k=1}^{m} \tilde{a}_{jk}(\boldsymbol{x}_{0,k} - \boldsymbol{x}_0) - \Delta t_0 \boldsymbol{f}(\boldsymbol{x}_{0,j}, \boldsymbol{u}_{0,j}, \boldsymbol{p}, t_{0,j}), \;\; j = 1, \ldots, m \\ \mathbf{0} &= \sum_{k=1}^{m} \tilde{a}_{jk}(\boldsymbol{x}_{i,k} - \boldsymbol{x}_{i-1,m}) - \Delta t_i \boldsymbol{f}(\boldsymbol{x}_{i,j}, \boldsymbol{u}_{i,j}, \boldsymbol{p}, t_{i,j}), \;\; i = 1, \ldots, n, j = 1, \ldots, m \\ \boldsymbol{g}^L &\leq \boldsymbol{g}(\boldsymbol{x}_{i,j}, \boldsymbol{u}_{i,j}, \boldsymbol{p}, t_{i,j}) \leq \boldsymbol{g}^U, \;\; \forall i = 0, \ldots, n \;\; \forall j = 1, \ldots, m \\ \boldsymbol{r}^L &\leq \boldsymbol{r}(\boldsymbol{x}_{n,m}, \boldsymbol{u}_{n,m}, \boldsymbol{p}, t_{n,m}) \leq \boldsymbol{r}^U \\ \boldsymbol{a}^L &\leq \boldsymbol{a}(\boldsymbol{p}) \leq \boldsymbol{a}^U \end{aligned} \tag{111}$$

is called discretized General Dynamic Optimization Problem (dGDOP).

5.2 Derivatives of the Nonlinear Optimization Problem

Solving the dGDOP with Ipopt requires certain derivative information (Chapter 4.3.1). This includes the gradient of the objective function, Jacobian of the constraints and Hessian of the Lagrangian with respect to the NLP variables (102). In this chapter the necessary partial derivatives and sparsity patterns are calculated. To get a shorter notation, the following abbreviations are introduced:

$$\begin{aligned}
\boldsymbol{z}_{i,j} &:= (\boldsymbol{x}_{i,j}, \boldsymbol{u}_{i,j}, \boldsymbol{p}, t_{i,j}),\ \ \boldsymbol{v}_{i,j} := (\boldsymbol{x}_{i,j}, \boldsymbol{u}_{i,j}) \\
L_{i,j}(\cdot) &:= L(\boldsymbol{x}_{i,j}, \boldsymbol{u}_{i,j}, \boldsymbol{p}, t_{i,j}),\ \ M_{n,m}(\cdot) := M(\boldsymbol{x}_{n,m}, \boldsymbol{u}_{n,m}, \boldsymbol{p}, t_{n,m}) \\
\boldsymbol{F}_{0,j}(\cdot) &:= \sum_{k=1}^{m} \tilde{a}_{jk}(\boldsymbol{x}_{0,k} - \boldsymbol{x}_0) - \Delta t_0 \boldsymbol{f}(\boldsymbol{x}_{0,j}, \boldsymbol{u}_{0,j}, \boldsymbol{p}, t_{0,j}),\ \ j = 1, \ldots, m \\
\boldsymbol{F}_{i,j}(\cdot) &:= \sum_{k=1}^{m} \tilde{a}_{jk}(\boldsymbol{x}_{i,k} - \boldsymbol{x}_{i-1,m}) - \Delta t_i \boldsymbol{f}(\boldsymbol{x}_{i,j}, \boldsymbol{u}_{i,j}, \boldsymbol{p}, t_{i,j}),\ \ i = 1, \ldots, n,\ j = 1, \ldots, m \\
\boldsymbol{g}_{i,j}(\cdot) &:= \boldsymbol{g}(\boldsymbol{x}_{i,j}, \boldsymbol{u}_{i,j}, \boldsymbol{p}, t_{i,j}),\ \ \forall i = 0, \ldots, n\ \ \forall j = 1, \ldots, m \\
\boldsymbol{r}_{n,m}(\cdot) &:= \boldsymbol{g}(\boldsymbol{x}_{n,m}, \boldsymbol{u}_{n,m}, \boldsymbol{p}, t_{n,m})
\end{aligned} \tag{112}$$

The abbreviations allow interpreting the derivatives of the NLP as callbacks to the derivatives of the continuous GDOP evaluated at some point $\boldsymbol{z}_{i,j}$. It is important to note that these derivatives may themselves be sparse and that this structure can be exploited in an implementation. The constraints must be sorted in an appropriate way to obtain proper block-structured Jacobians and Hessians. In this thesis the constraint vector is sorted as

$$\boldsymbol{g}_{NLP} := \begin{pmatrix} \boldsymbol{F}_{0,1}(\cdot) \\ \boldsymbol{g}_{0,1}(\cdot) \\ \vdots \\ \boldsymbol{F}_{0,m}(\cdot) \\ \boldsymbol{g}_{0,m}(\cdot) \\ \boldsymbol{F}_{1,1}(\cdot) \\ \boldsymbol{g}_{1,1}(\cdot) \\ \vdots \\ \boldsymbol{F}_{1,m}(\cdot) \\ \boldsymbol{g}_{1,m}(\cdot) \\ \vdots \\ \boldsymbol{F}_{n,1}(\cdot) \\ \boldsymbol{g}_{n,1}(\cdot) \\ \vdots \\ \boldsymbol{F}_{n,m}(\cdot) \\ \boldsymbol{g}_{n,m}(\cdot) \\ \boldsymbol{r}_{n,m}(\cdot) \\ \boldsymbol{a}(\cdot) \end{pmatrix} \quad \text{with} \quad \begin{pmatrix} \mathbf{0} \\ \boldsymbol{g}^L \\ \vdots \\ \mathbf{0} \\ \boldsymbol{g}^L \\ \mathbf{0} \\ \boldsymbol{g}^L \\ \vdots \\ \mathbf{0} \\ \boldsymbol{g}^L \\ \vdots \\ \mathbf{0} \\ \boldsymbol{g}^L \\ \vdots \\ \mathbf{0} \\ \boldsymbol{g}^L \\ \boldsymbol{r}^L \\ \boldsymbol{a}^L \end{pmatrix} \leq \begin{pmatrix} \boldsymbol{F}_{0,1}(\cdot) \\ \boldsymbol{g}_{0,1}(\cdot) \\ \vdots \\ \boldsymbol{F}_{0,m}(\cdot) \\ \boldsymbol{g}_{0,m}(\cdot) \\ \boldsymbol{F}_{1,1}(\cdot) \\ \boldsymbol{g}_{1,1}(\cdot) \\ \vdots \\ \boldsymbol{F}_{1,m}(\cdot) \\ \boldsymbol{g}_{1,m}(\cdot) \\ \vdots \\ \boldsymbol{F}_{n,1}(\cdot) \\ \boldsymbol{g}_{n,1}(\cdot) \\ \vdots \\ \boldsymbol{F}_{n,m}(\cdot) \\ \boldsymbol{g}_{n,m}(\cdot) \\ \boldsymbol{r}_{n,m}(\cdot) \\ \boldsymbol{a}(\cdot) \end{pmatrix} \leq \begin{pmatrix} \mathbf{0} \\ \boldsymbol{g}^U \\ \vdots \\ \mathbf{0} \\ \boldsymbol{g}^U \\ \mathbf{0} \\ \boldsymbol{g}^U \\ \vdots \\ \mathbf{0} \\ \boldsymbol{g}^U \\ \vdots \\ \mathbf{0} \\ \boldsymbol{g}^U \\ \vdots \\ \mathbf{0} \\ \boldsymbol{g}^U \\ \boldsymbol{r}^U \\ \boldsymbol{a}^U \end{pmatrix}. \tag{113}$$

It will be demonstrated in the following chapters that this sorting together with the sorting of the variables (102) results in an advantageous block structure.

5.2.1 Gradient of the Objective Function

At first, the objective function

$$\phi(\cdot) := M_{n,m}(\cdot) + \sum_{i=0}^{n} \Delta t_i \sum_{j=1}^{m} b_j L_{i,j}(\cdot) \tag{114}$$

is differentiated with respect to all variables $\boldsymbol{x}_{NLP}$ (102). For $(i,j) \neq (n,m)$ a straightforward calculation leads to

$$\frac{\partial \phi}{\partial \boldsymbol{x}_{i,j}} = \Delta t_i b_j \nabla_{\boldsymbol{x}} L\Big|_{\boldsymbol{z}_{i,j}}, \quad \frac{\partial \phi}{\partial \boldsymbol{u}_{i,j}} = \Delta t_i b_j \nabla_{\boldsymbol{u}} L\Big|_{\boldsymbol{z}_{i,j}}. \tag{115}$$

For $(i,j) = (n,m)$ the gradient is

$$\frac{\partial \phi}{\partial \boldsymbol{x}_{n,m}} = \nabla_{\boldsymbol{x}} M\Big|_{\boldsymbol{z}_{n,m}} + \Delta t_n b_m \nabla_{\boldsymbol{x}} L\Big|_{\boldsymbol{z}_{n,m}}, \quad \frac{\partial \phi}{\partial \boldsymbol{u}_{n,m}} = \nabla_{\boldsymbol{u}} M\Big|_{\boldsymbol{z}_{n,m}} + \Delta t_n b_m \nabla_{\boldsymbol{u}} L\Big|_{\boldsymbol{z}_{n,m}} \tag{116}$$

and the derivative with respect to the parameters is

$$\frac{\partial \phi}{\partial \boldsymbol{p}} = \nabla_{\boldsymbol{p}} M\Big|_{\boldsymbol{z}_{n,m}} + \sum_{i=0}^{n} \Delta t_i \sum_{j=1}^{m} b_j \nabla_{\boldsymbol{p}} L\Big|_{\boldsymbol{z}_{i,j}}. \tag{117}$$

Thus, the gradient is a dense vector, but becomes sparse if the objective consists only of the Mayer term $M(\cdot)$. This has no effect on the implementation, because the gradient is required as a dense vector by Ipopt.

5.2.2 Jacobian of the Constraints

The constraint Jacobian $\frac{\mathrm{d}\boldsymbol{g}_{NLP}}{\mathrm{d}\boldsymbol{x}_{NLP}}$ can be divided into the blocks J_0, J_1, J_2 and P_0, P_1, P_2. Blocks beginning with J are present in every optimal control problem and contain all derivatives of the state and control variables. The blocks starting with P are only used if parameters are also optimized, as they contain the derivatives with respect to the parameters. In general, the Jacobian has the following sparsity pattern:

constrs\vars	$\boldsymbol{v}_{0,:}$	$\boldsymbol{v}_{1,:}$	$\dots$	$\boldsymbol{v}_{n,:}$	$\boldsymbol{p}$
$\boldsymbol{F}_{0,:}, \boldsymbol{g}_{0,:}$	J_0	0	0	0	P_0
$\boldsymbol{F}_{1,:}, \boldsymbol{g}_{1,:}$	0	J_1	0	0	P_0
$\vdots$	0	0	$\ddots$	0	$\vdots$
$\boldsymbol{F}_{n,:}, \boldsymbol{g}_{n,:}$	0	0	0	J_1	P_0
$\boldsymbol{r}_{n,m}$	0	0	0	J_2	P_1
$\boldsymbol{a}$	0	0	0	0	P_2

(118)

The Jacobian is obviously very sparse and follows a cyclic structure. The components of each block and all nonzero derivatives are presented below. For $i = 0, \dots, n$, $j = 1, \dots, m$ and $q \in \{1, \dots, m\}$ the derivatives of

the discretized dynamic are

$$\frac{\partial \boldsymbol{F}_{i,j}}{\partial \boldsymbol{x}_{i,q}} = \begin{cases} \tilde{a}_{jd} I, & \text{if } q \neq j, \\ \tilde{a}_{jj} I - \Delta t_i \nabla_{\boldsymbol{x}} \boldsymbol{f}\Big|_{\boldsymbol{z}_{i,j}}, & \text{if } q = j \end{cases}, \quad \frac{\partial \boldsymbol{F}_{i,j}}{\partial \boldsymbol{u}_{i,j}} = -\Delta t_i \nabla_{\boldsymbol{u}} \boldsymbol{f}\Big|_{\boldsymbol{z}_{i,j}}, \quad \frac{\partial \boldsymbol{F}_{i,j}}{\partial \boldsymbol{p}} = -\Delta t_i \nabla_{\boldsymbol{p}} \boldsymbol{f}\Big|_{\boldsymbol{z}_{i,j}}. \tag{119}$$

For $i = 0, \ldots, n$ and $j = 1, \ldots, m$ the derivatives of the path constraints are

$$\frac{\partial \boldsymbol{g}_{i,j}}{\partial \boldsymbol{x}_{i,j}} = \nabla_{\boldsymbol{x}} \boldsymbol{g}\Big|_{\boldsymbol{z}_{i,j}}, \quad \frac{\partial \boldsymbol{g}_{i,j}}{\partial \boldsymbol{u}_{i,j}} = \nabla_{\boldsymbol{u}} \boldsymbol{g}\Big|_{\boldsymbol{z}_{i,j}}, \quad \frac{\partial \boldsymbol{g}_{i,j}}{\partial \boldsymbol{p}} = \nabla_{\boldsymbol{p}} \boldsymbol{g}\Big|_{\boldsymbol{z}_{i,j}}. \tag{120}$$

The blocks J_0 and P_0 contain (119) and (120) for $i = 0$. Because of the derivative $\frac{\partial \boldsymbol{F}_{i,j}}{\partial \boldsymbol{x}_{i,q}}$, many block diagonal matrices are present in each block. Furthermore, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$

$$\frac{\partial \boldsymbol{F}_{i,j}}{\partial \boldsymbol{x}_{i-1,m}} = -\sum_{k=1}^{m} \tilde{a}_{jk} I \tag{121}$$

is also part of the Jacobian. Therefore the matrix J_1 consists of $\frac{\partial \boldsymbol{F}_{i,j}}{\partial \boldsymbol{x}_{i-1,m}}$ and all derivatives that are contained in J_0. Since the structure of the derivatives does not change for $i > 0$ and arbitrary j, the sparsity pattern is repeated many times. This fact can be seen in Figure 8, where the sparsity pattern of Model 2.1 with $n = 24$ and $m = 3$ is visualized.

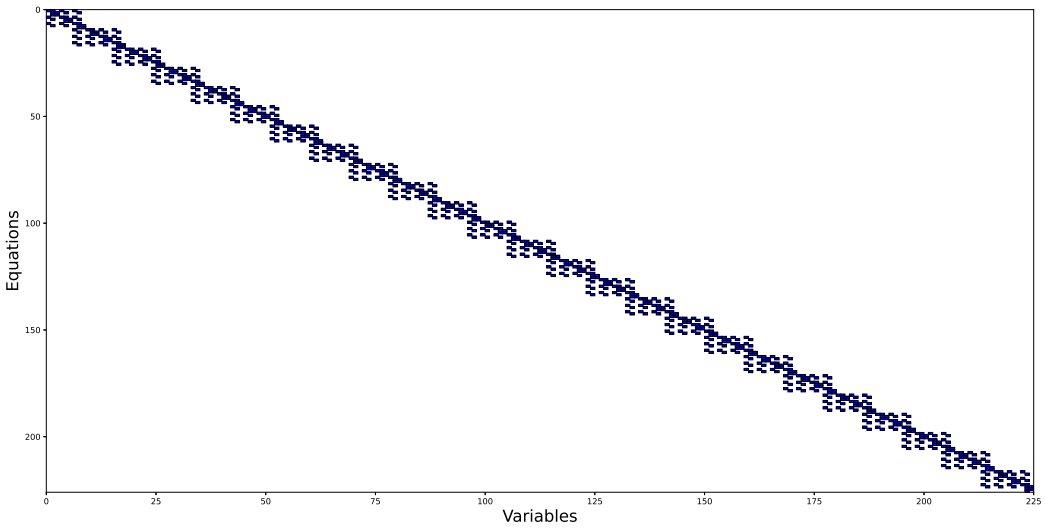


Figure 8: Sparse Jacobian of Model 2.1

In Figure 8 the derivatives of the final constraints

$$\frac{\partial \boldsymbol{r}_{n,m}}{\partial \boldsymbol{x}_{n,m}} = \nabla_{\boldsymbol{x}} \boldsymbol{r}\Big|_{\boldsymbol{z}_{n,m}}, \quad \frac{\partial \boldsymbol{r}_{n,m}}{\partial \boldsymbol{u}_{n,m}} = \nabla_{\boldsymbol{u}} \boldsymbol{r}\Big|_{\boldsymbol{z}_{n,m}}, \quad \frac{\partial \boldsymbol{r}_{n,m}}{\partial \boldsymbol{p}} = \nabla_{\boldsymbol{p}} \boldsymbol{r}\Big|_{\boldsymbol{z}_{n,m}} \tag{122}$$

are also visible in the very last row, since no parameters are present. In general, these form the block matrices J_2, which include $\frac{\partial \boldsymbol{r}_{n,m}}{\partial \boldsymbol{x}_{n,m}}$ and $\frac{\partial \boldsymbol{r}_{n,m}}{\partial \boldsymbol{u}_{n,m}}$, as well as $P_1 = \frac{\partial \boldsymbol{r}_{n,m}}{\partial \boldsymbol{p}}$.

The remaining block P_2 contains the derivatives of the parametric constraints

$$\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{p}} = \nabla_{\boldsymbol{p}} \boldsymbol{a}\Big|_{\boldsymbol{p}}. \tag{123}$$

5.2.3 Hessian of the Lagrangian

Because of *Schwarz's theorem*, only the lower triangular part of the Lagrangian has to be provided. Recall that Ipopt requires the Lagrangian $\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \sigma_f) = \sigma_f \nabla^2 f(\boldsymbol{x}) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(\boldsymbol{x})$, where the functions g_i are the constraints (113) and λ_i are the corresponding Lagrangian multipliers, which are provided by the solver in every iteration (see Chapter 4.3.1). To keep the notation short, the shift function

$$s(i, j) = (im + j - 1)(d_{\boldsymbol{x}} + d_{\boldsymbol{g}}) \tag{124}$$

is introduced. Given an interval i and a node j, the shift function $s(i, j)$ is constructed in a way, that it returns the index of the constraint $\boldsymbol{F}_{i,j}$ in the constraint vector $\boldsymbol{g}_{NLP}$ and thus the index of the corresponding Lagrangian multiplier. The Hessian has a very straightforward sparsity structure, which is built purely with the sparsity patterns of the continuous functions, since the linear terms $\sum_{k=1}^{m} \tilde{a}_{jk}(\boldsymbol{x}_{i,k} - \boldsymbol{x}_{i-1,m})$ vanish in the second derivative. The Hessian has the following general sparsity pattern:

vars\vars	$\boldsymbol{v}_{0,1}$	$\dots$	$\boldsymbol{v}_{n,m-1}$	$\boldsymbol{v}_{n,m}$	$\boldsymbol{p}$
$\boldsymbol{v}_{0,1}$	A	0	0	0	B
$\vdots$	0	$\ddots$	0	0	$\vdots$
$\boldsymbol{v}_{n,m-1}$	0	0	A	0	B
$\boldsymbol{v}_{n,m}$	0	0	0	$\hat{A}$	$\hat{B}$
$\boldsymbol{p}$	B	$\dots$	B	$\hat{B}$	C

(125)

Here A and $\tilde{A}$ are the block matrices, which stem from the derivatives with respect to the states and controls, i.e. $\nabla^2_{\boldsymbol{xx}}, \nabla^2_{\boldsymbol{ux}}, \nabla^2_{\boldsymbol{uu}}$. The matrices $B, \tilde{B}, C$ originate from (mixed) derivatives with respect to the parameters, i.e. $\nabla^2_{\boldsymbol{px}}, \nabla^2_{\boldsymbol{pu}}, \nabla^2_{\boldsymbol{pp}}$. Note that the matrix $\tilde{A}$ contains A completely, but also consists of the final constraints and the Mayer term derivatives. The same holds for the block matrices $\tilde{B}$ and B. Since parameters can be used in every function and constraint of the GDOP, block C, which contains the derivatives $\nabla^2_{\boldsymbol{pp}}$, is basically a weighted sum over all continuous Hessians evaluated at some points.
Now, the evaluation of block A and $\tilde{A}$ is presented. The Hessian block matrices $B, \tilde{B}$ and C are outlined in the Appendix D. For $(i, j) \neq (n, m)$, the derivatives

$$\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{x}_{i,j} \partial \boldsymbol{x}_{i,j}} = \sigma_f \Delta t_i b_j \nabla^2_{\boldsymbol{xx}} L\Big|_{\boldsymbol{z}_{i,j}} - \Delta t_i \sum_{d=1}^{d_{\boldsymbol{x}}} \lambda_{s(i,j)+d} \nabla^2_{\boldsymbol{xx}} f^{(d)}\Big|_{\boldsymbol{z}_{i,j}} + \sum_{d=1}^{d_{\boldsymbol{g}}} \lambda_{s(i,j)+d_{\boldsymbol{x}}+d} \nabla^2_{\boldsymbol{xx}} g^{(d)}\Big|_{\boldsymbol{z}_{i,j}} \tag{126}$$

$$\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{u}_{i,j} \partial \boldsymbol{x}_{i,j}} = \sigma_f \Delta t_i b_j \nabla^2_{\boldsymbol{ux}} L\Big|_{\boldsymbol{z}_{i,j}} - \Delta t_i \sum_{d=1}^{d_{\boldsymbol{x}}} \lambda_{s(i,j)+d} \nabla^2_{\boldsymbol{ux}} f^{(d)}\Big|_{\boldsymbol{z}_{i,j}} + \sum_{d=1}^{d_{\boldsymbol{g}}} \lambda_{s(i,j)+d_{\boldsymbol{x}}+d} \nabla^2_{\boldsymbol{ux}} g^{(d)}\Big|_{\boldsymbol{z}_{i,j}} \tag{127}$$

$$\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{u}_{i,j} \partial \boldsymbol{u}_{i,j}} = \sigma_f \Delta t_i b_j \nabla^2_{\boldsymbol{uu}} L\Big|_{\boldsymbol{z}_{i,j}} - \Delta t_i \sum_{d=1}^{d_{\boldsymbol{x}}} \lambda_{s(i,j)+d} \nabla^2_{\boldsymbol{uu}} f^{(d)}\Big|_{\boldsymbol{z}_{i,j}} + \sum_{d=1}^{d_{\boldsymbol{g}}} \lambda_{s(i,j)+d_{\boldsymbol{x}}+d} \nabla^2_{\boldsymbol{uu}} g^{(d)}\Big|_{\boldsymbol{z}_{i,j}} \tag{128}$$

form matrix A. The block matrix $\tilde{A}$ also contains the derivatives of the final constraints and the Mayer term and is evaluated as

$$\begin{aligned}\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{x}_{n,m} \partial \boldsymbol{x}_{n,m}} &= \sigma_f \nabla^2_{\boldsymbol{xx}} M\Big|_{\boldsymbol{z}_{n,m}} + \sigma_f \Delta t_n b_m \nabla^2_{\boldsymbol{xx}} L\Big|_{\boldsymbol{z}_{n,m}} - \Delta t_n \sum_{d=1}^{d_{\boldsymbol{x}}} \lambda_{s(n,m)+d} \nabla^2_{\boldsymbol{xx}} f^{(d)}\Big|_{\boldsymbol{z}_{n,m}} \\ &\quad + \sum_{d=1}^{d_{\boldsymbol{g}}} \lambda_{s(n,m)+d_{\boldsymbol{x}}+d} \nabla^2_{\boldsymbol{xx}} g^{(d)}\Big|_{\boldsymbol{z}_{n,m}} + \sum_{d=1}^{d_{\boldsymbol{r}}} \lambda_{s(n+1,0)+d} \nabla^2_{\boldsymbol{xx}} r^{(d)}\Big|_{\boldsymbol{z}_{n,m}}\end{aligned} \tag{129}$$

$$\begin{aligned}\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{u}_{n,m} \partial \boldsymbol{x}_{n,m}} &= \sigma_f \nabla^2_{\boldsymbol{ux}} M\Big|_{\boldsymbol{z}_{n,m}} + \sigma_f \Delta t_n b_m \nabla^2_{\boldsymbol{ux}} L\Big|_{\boldsymbol{z}_{n,m}} - \Delta t_n \sum_{d=1}^{d_{\boldsymbol{x}}} \lambda_{s(n,m)+d} \nabla^2_{\boldsymbol{ux}} f^{(d)}\Big|_{\boldsymbol{z}_{n,m}} \\ &\quad + \sum_{d=1}^{d_{\boldsymbol{g}}} \lambda_{s(n,m)+d_{\boldsymbol{x}}+d} \nabla^2_{\boldsymbol{ux}} g^{(d)}\Big|_{\boldsymbol{z}_{n,m}} + \sum_{d=1}^{d_{\boldsymbol{r}}} \lambda_{s(n+1,0)+d} \nabla^2_{\boldsymbol{ux}} r^{(d)}\Big|_{\boldsymbol{z}_{n,m}}\end{aligned} \tag{130}$$

$$\begin{aligned}\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{u}_{n,m} \partial \boldsymbol{u}_{n,m}} &= \sigma_f \nabla^2_{\boldsymbol{uu}} M\Big|_{\boldsymbol{z}_{n,m}} + \sigma_f \Delta t_n b_m \nabla^2_{\boldsymbol{uu}} L\Big|_{\boldsymbol{z}_{n,m}} - \Delta t_n \sum_{d=1}^{d_{\boldsymbol{x}}} \lambda_{s(n,m)+d} \nabla^2_{\boldsymbol{uu}} f^{(d)}\Big|_{\boldsymbol{z}_{n,m}} \\ &\quad + \sum_{d=1}^{d_{\boldsymbol{g}}} \lambda_{s(n,m)+d_{\boldsymbol{x}}+d} \nabla^2_{\boldsymbol{uu}} g^{(d)}\Big|_{\boldsymbol{z}_{n,m}} + \sum_{d=1}^{d_{\boldsymbol{r}}} \lambda_{s(n+1,0)+d} \nabla^2_{\boldsymbol{uu}} r^{(d)}\Big|_{\boldsymbol{z}_{n,m}}.\end{aligned} \tag{131}$$

An example Hessian sparsity pattern of the model *Satellite* (Appendix G) with $n = 7, m = 3$ is considered. The model does not contain any parameters as seen by the absence of blocks $B, \tilde{B}$ and C. Furthermore, the matrix is extremely sparse and the cyclic structure of the Hessian can be observed. The block A is repeated $(n+1)m - 1 = 23$ times until block $\tilde{A}$ is reached. $\tilde{A}$ is particularly noticeable because the Mayer term of the problem contains all states quadratically, which results in a diagonal in the last block.

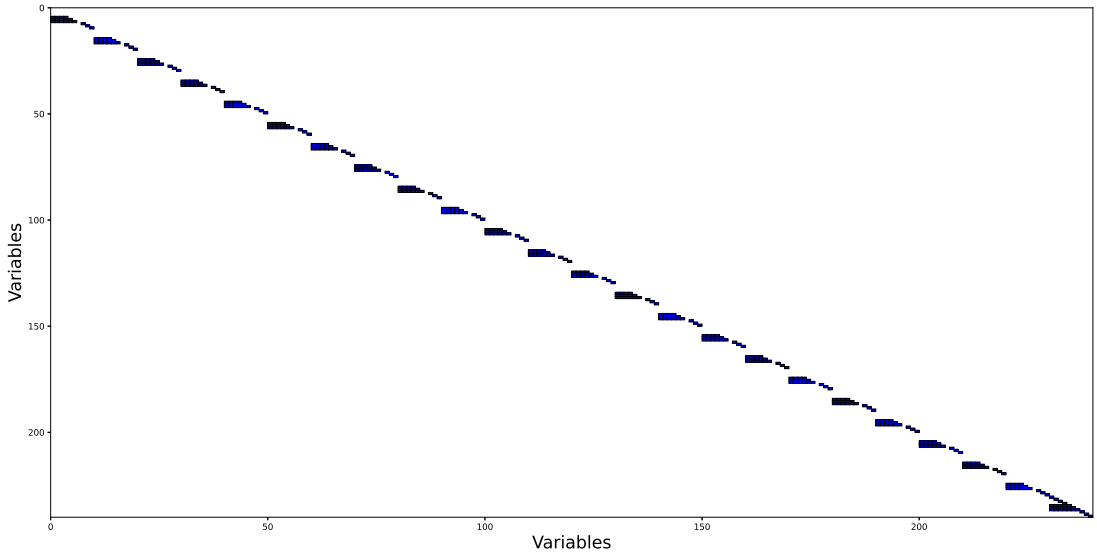


Figure 9: Sparse Hessian of Model *Satellite* (Appendix G)

5.3 Equivalence of the dGDOP and fLGR Pseudospectral Collocation

In Chapter 5.1 the discretization of the continuous GDOP was performed using the Radau IIA Runge-Kutta collocation method, because this guarantees highly favorable stability and accuracy properties. Now, another equivalent formulation is constructed based on this discretization. The new formulation of the dGDOP is expressed in terms of the first differentiation matrix $D^{(1)}$ (Chapter 3.1.2) at the fLGR points rescaled to $[0, 1]$. In pseudospectral collocation, the formulation based on fLGR points (without rescaling) is commonly used. It is shown that if the number of collocation nodes and thus the polynomial degree per interval was allowed to vary, the dGDOP would be equivalent to the transcription of dynamic optimization problems in pseudospectral collocation methods.[49][46][48] Therefore, the proposed framework can be easily extended in this way and can be interpreted as a Runge-Kutta, local and global collocation method. The formulation based on the first differentiation matrix $D^{(1)}$ also allows for a very fast computation of the required coefficients. Recall that the discretized dynamic of the GDOP has the form

$$(A^{-1} \otimes I) \begin{pmatrix} \boldsymbol{x}_{i,1} - \boldsymbol{x}_{i-1,m} \\ \vdots \\ \boldsymbol{x}_{i,m} - \boldsymbol{x}_{i-1,m} \end{pmatrix} - \Delta t_i \begin{pmatrix} \boldsymbol{f}(\boldsymbol{x}_{i,1}, \boldsymbol{u}_{i,1}, \boldsymbol{p}, t_{i,1}) \\ \vdots \\ \boldsymbol{f}(\boldsymbol{x}_{i,m}, \boldsymbol{u}_{i,m}, \boldsymbol{p}, t_{i,m}) \end{pmatrix} = \boldsymbol{0}. \tag{132}$$

Since $\boldsymbol{x}_{i-1,m}$ is subtracted in each component of the vector, the expression can be written as

$$\begin{pmatrix} \gamma_1 I & \tilde{a}_{11} I & \dots & \tilde{a}_{1m} I \\ \vdots & \vdots & & \vdots \\ \gamma_m I & \tilde{a}_{m1} I & \dots & \tilde{a}_{mm} I \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_{i-1,m} \\ \boldsymbol{x}_{i,1} \\ \vdots \\ \boldsymbol{x}_{i,m} \end{pmatrix} - \Delta t_i \begin{pmatrix} \boldsymbol{f}(\boldsymbol{x}_{i,1}, \boldsymbol{u}_{i,1}, \boldsymbol{p}, t_{i,1}) \\ \vdots \\ \boldsymbol{f}(\boldsymbol{x}_{i,m}, \boldsymbol{u}_{i,m}, \boldsymbol{p}, t_{i,m}) \end{pmatrix} = \boldsymbol{0} \tag{133}$$

with $\gamma_d = -\sum_{k=1}^{m} \tilde{a}_{dk}$. Because a collocation approach is chosen, the state variables can by Definition 3.11 be written as $\boldsymbol{x}_i(\tau) = \sum_{j=0}^{m} \boldsymbol{x}_{i,j} l_j(\tau)$ on every interval i. Here, l_j denotes the j-th Lagrange basis polynomial with respect to the nodes $t_i, t_i + \Delta t_i c_1, \dots, t_i + \Delta t_i c_m = t_{i+1}$. Differentiating the polynomial yields

$$\frac{\mathrm{d}\boldsymbol{x}_i(\tau)}{\mathrm{d}\tau} = \sum_{j=0}^{m} \boldsymbol{x}_{i,j} \frac{\mathrm{d}l_j(\tau)}{\mathrm{d}\tau}. \tag{134}$$

By Definition 3.11, the collocation polynomial has to satisfy the differential equation at every node c_k for $k = 1, \dots, m$ and thus

$$\frac{\mathrm{d}\boldsymbol{x}_i}{\mathrm{d}\tau}(t_i + \Delta t_i c_k) = \sum_{j=0}^{m} \boldsymbol{x}_{i,j} \frac{\mathrm{d}l_j}{\mathrm{d}\tau}(t_i + \Delta t_i c_k) = \begin{pmatrix} \boldsymbol{f}(\boldsymbol{x}_{i,1}, \boldsymbol{u}_{i,1}, \boldsymbol{p}, t_{i,1}) \\ \vdots \\ \boldsymbol{f}(\boldsymbol{x}_{i,m}, \boldsymbol{u}_{i,m}, \boldsymbol{p}, t_{i,m}) \end{pmatrix}. \tag{135}$$

Applying the affine transformation $\tilde{\tau} = \frac{\tau - t_i}{\Delta t_i}$, the expression can be written as

$$\frac{1}{\Delta t_i} \sum_{j=0}^{m} \boldsymbol{x}_{i,j} \frac{\mathrm{d}\tilde{l}_j}{\mathrm{d}\tilde{\tau}}(c_k) = \begin{pmatrix} \boldsymbol{f}(\boldsymbol{x}_{i,1}, \boldsymbol{u}_{i,1}, \boldsymbol{p}, t_{i,1}) \\ \vdots \\ \boldsymbol{f}(\boldsymbol{x}_{i,m}, \boldsymbol{u}_{i,m}, \boldsymbol{p}, t_{i,m}) \end{pmatrix}, \tag{136}$$

where $\tilde{l}_j$ is the j-th Lagrange polynomial with respect to the nodes $0, c_1, \ldots, c_m = 1$. By Definition 3.2 $D^{(1)}_{k,j} = \frac{\mathrm{d}\tilde{l}_j}{\mathrm{d}\tilde{\tau}}(c_k)$. Rearranging terms results in

$$\begin{pmatrix} D^{(1)}_{1,0} I & D^{(1)}_{1,1} I & \ldots & D^{(1)}_{1,m} I \\ \vdots & \vdots & & \vdots \\ D^{(1)}_{m,0} I & D^{(1)}_{m,1} I & \ldots & D^{(1)}_{m,m} I \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_{i-1,m} \\ \boldsymbol{x}_{i,1} \\ \vdots \\ \boldsymbol{x}_{i,m} \end{pmatrix} - \Delta t_i \begin{pmatrix} \boldsymbol{f}(\boldsymbol{x}_{i,1}, \boldsymbol{u}_{i,1}, \boldsymbol{p}, t_{i,1}) \\ \vdots \\ \boldsymbol{f}(\boldsymbol{x}_{i,m}, \boldsymbol{u}_{i,m}, \boldsymbol{p}, t_{i,m}) \end{pmatrix} = \boldsymbol{0}. \tag{137}$$

By comparing the coefficients with (133) the equivalence is established, since all other functions and constraints in pseudospectral collocation are equivalent to the dGDOP anyway (see [49][46][30]). Additionally, in pseudospectral methods the time horizon $[t_0, t_f]$ is always scaled to the normalized interval $[-1, 1]$, because the fLGR points are defined on $[-1, 1]$. In this thesis, the process is equivalently described on the arbitrary time horizon $[t_0, t_f]$ by adding a factor of Δt_i to the dynamic and using the rescaled fLGR points. The computation of the Butcher matrix A and its subsequent inversion also becomes obsolete, since the differentiation matrices allow for a new representation of the inverse Butcher matrix $\tilde{a}_{ij} = D^{(1)}_{ij}$, which can be computed extremely fast with (28).

6 Mesh Refinement

Using the structure and derivative information of the dGDOP (Definition 5.1), which have been calculated in Chapter 5.1, the problem can be implemented and solved with a generic NLP solver such as Ipopt or SNOPT. Consequently, an initial guess on the control, states and parameters as well as the number of intervals and collocation nodes must be provided. The resulting algorithm is very simple and basically replaces solving the continuous GDOP with solving the dGDOP on some mesh $\mathcal{M} = \{t_0, t_1, \ldots, t_{n-1}, t_n\}$ with $t_n = t_f$. Furthermore, on each interval $[t_i, t_{i+1}]$ a collocation scheme based on the rescaled fLGR nodes $c_1, \ldots, c_m$ is used. Without any knowledge of the problem, it is clearly best to choose an equidistant grid, i.e. $\Delta t_i = t_{i+1} - t_i \equiv const, i = 0, \ldots, n$. This approach works quite well for general problems. However, the algorithm has several key disadvantages, all of which can be overcome by the use of so-called *mesh refinement algorithms*, introduced in this section. In addition, the novel mesh refinement algorithm *L2-Boundary-Norm* is also presented in this context.

6.1 Iterative Mesh Refinement

At first, some observations about solving the problem with NLP solvers are made. In general, the initial guess for the control variables $\boldsymbol{u}_{i,j}$ is rather poor, e.g. constant everywhere, if no additional information about the problem is known. This makes the optimization very expensive, because the optimizer has to perform a lot of steps to converge, if it converges at all. Note that interior-point methods are pretty robust and therefore often converge even for poor initial guesses, but may take a significant amount of time. For computational reasons, it is therefore beneficial to solve only very small problems with the initial guess.
In addition, the optimal solution may contain discontinuities or switches, kinks, bends and steep sections. If any of these occur and the mesh resolution at these points is too low, it is very likely that the error of the control and states to the exact solution will be abnormally high. The solution obtained by the optimization thus becomes unrealistic. Therefore, a very high resolution, i.e. a dense mesh, is required at the points where such behavior occurs. If the control does not contain such behavior, a high resolution is not required, although the step size should not be taken too large.
These observations lead directly to an iterative mesh refinement. A mesh refinement algorithm is a procedure that solves the initial NLP on a coarse mesh, refines the mesh and updates the initial values based on the previous optimal solution. It then calls the optimizer again until a stopping condition is met. All previous considerations are fulfilled. The initial mesh can be chosen to be arbitrarily small but must reflect the principle behavior of the optimal control and states. Consequently, the initial optimization is very fast even for poor initial guesses. The behavior of the optimal solution is then analyzed in order to find switches, kinks, bends and steep sections. In general, the goal is to find a new mesh such that the new optimal solution has a lower error compared to the exact solution. If the error in the current iteration is low enough, the algorithm terminates. Otherwise the old optimal solution is interpolated on the new mesh and the optimizer is run again. Because the old optimal solution is feasible and optimal for the previous problem and both problems are very similar, the primal and dual infeasibility of the old solution for the new problem is likely to be very small. If the problem is sensitive or chaotic, this may not be the case. For well conditioned problems though, only a few iterations of the optimizer are needed to obtain the new optimal solution. This makes the procedure very fast. The resulting prototype mesh refinement algorithm

for the GDOP is given below.

Algorithm 6.4: Prototype Iterative Mesh Refinement Algorithm

Input: GDOP (Definition 2.2), guesses $\boldsymbol{x}_{ij}$, $\boldsymbol{u}_{ij}$, $\boldsymbol{p}$, mesh $\mathcal{M}_0$, number of collocation nodes m

Output: $\boldsymbol{x}^*_{ij}, \boldsymbol{u}^*_{ij}, \boldsymbol{p}^*$

```
k ← 0;
x*_ij, u*_ij, p* ← Solve the dGDOP on M_0 with m collocation nodes and guess x_ij, u_ij, p;
while stopping condition is not met do
    M_{k+1} ← Update the the mesh M_k with the optimal solution x*_ij, u*_ij, p*;
    x_ij, u_ij, p ← Interpolate x*_ij, u*_ij, p* on M_{k+1};
    x*_ij, u*_ij, p* ← Solve the dGDOP on M_{k+1} with m collocation nodes and guess x_ij, u_ij, p;
    k ← k + 1;
end
return x*_ij, u*_ij, p*;
```

An important fact about the prototype algorithm is that in this formulation the number of collocation nodes remains constant throughout all iterations, because in the definition of the dGDOP (Definition 5.1) these are set to be constant. In general, this must not apply, as varying the polynomial degree per interval can greatly reduce the error. The number of collocation nodes m is also an input to the algorithm. It is therefore important to chose m adequately. To investigate this further, the behavior of smooth and non-smooth problems for varying numbers of n and m are considered in the following chapter. These considerations give rise to major classes of mesh refinement algorithms as well as their combinations. [58][46][56]

6.2 Classes of Mesh Refinement Algorithms

6.2.1 Convergence of Radau Collocation

To introduce the principle classes of mesh refinement algorithms for direct collocation-based dynamic optimization, an important theoretical result is stated first. As shown by *Kameswaran* and *Biegler* in [47] or by *Hager* in [59], the Radau collocation approach offers great convergence to the optimal solution. Moreover, under suitable conditions and for sufficiently smooth problems the error of Radau collocation with n intervals and m collocation nodes is at least $O(h^m)$ for all continuous variables, i.e. states, control and even costates. This is a generalization of Theorem 3.8, because the control is bounded in this way as well. Since $O(h) = O\left(\frac{1}{n}\right)$, the error can be written as $O(n^{-m})$. This result implies that the error of smooth problems decays at the so-called *spectral rate*, i.e. *exponentially fast*, as a function of the number of collocation nodes m. The result can also be generalized to the broader class of orthogonal collocation methods, such as Gauss or Lobatto global collocation. If only the number of intervals n is increased, then the error decreases just polynomially. Nevertheless, this is still a relatively fast convergence towards the optimal solution.

To illustrate this result, two examples are considered. First, the hypersensitive optimal control problem (Model A.1) that has a smooth analytical solution 163 164 is solved without any mesh refinement and for a final time of $t_f = 25$. The problem is solved for a varying number of collocation nodes with fixed $n = 1$ as

well as with fixed polynomial degree $m = 3$ and varying number of intervals. The numerical optimal state solution $x(t)$ as well as the actual optimal state $x^*(t)$ are depicted in Figure 10.

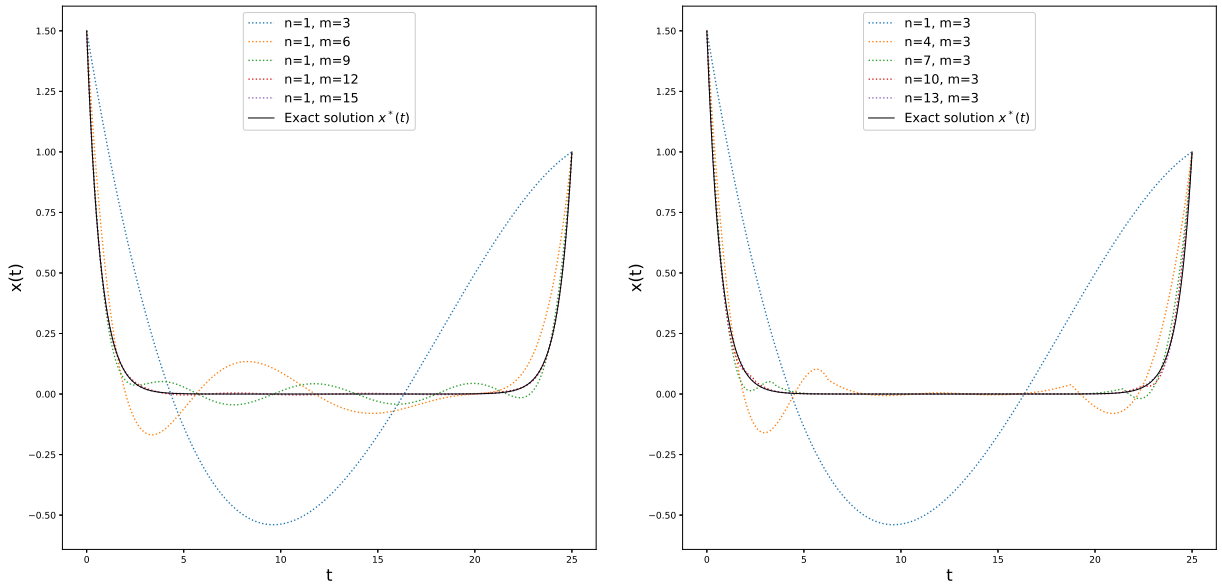


Figure 10: Optimal state trajectories for varying n and m of the smooth Model A.1

It can be seen that in both cases, for varying numbers of n and m, the solutions rapidly converge to the exact state solution. Furthermore, the maximum error at the collocation nodes is shown in Table 1 in the column $\|x^* - x\|_\infty$. Just as proven in theory, the error of a Radau collocation method decreases exponentially for an increasing number of nodes m. The error of the fixed degree approach with varying number of intervals n also decreases, but obviously not at an exponential rate.

n	m	$\|x^* - x\|_\infty$	$\|v^* - v\|_\infty$
1	3	$1.01090 \cdot 10^{-1}$	$1.44329 \cdot 10^{-1}$
1	6	$1.23319 \cdot 10^{-1}$	$1.09776 \cdot 10^{-1}$
1	9	$1.92360 \cdot 10^{-2}$	$1.00248 \cdot 10^{-1}$
1	12	$4.40250 \cdot 10^{-3}$	$1.52441 \cdot 10^{-1}$
1	15	$2.56256 \cdot 10^{-4}$	$9.36149 \cdot 10^{-2}$
1	18	$3.23635 \cdot 10^{-5}$	$2.47793 \cdot 10^{-2}$
1	21	$8.02432 \cdot 10^{-7}$	$2.89962 \cdot 10^{-2}$
1	30	$3.50543 \cdot 10^{-11}$	$6.99020 \cdot 10^{-2}$

(a) Errors for a varying number of nodes m

n	m	$\|x^* - x\|_\infty$	$\|v^* - v\|_\infty$
1	3	$1.01090 \cdot 10^{-1}$	$1.44329 \cdot 10^{-1}$
4	3	$1.07285 \cdot 10^{-1}$	$3.60824 \cdot 10^{-2}$
7	3	$4.53019 \cdot 10^{-2}$	$2.06185 \cdot 10^{-2}$
10	3	$2.29132 \cdot 10^{-2}$	$1.44330 \cdot 10^{-2}$
13	3	$1.28503 \cdot 10^{-2}$	$1.11022 \cdot 10^{-2}$
16	3	$7.60206 \cdot 10^{-3}$	$9.02061 \cdot 10^{-3}$
19	3	$4.69443 \cdot 10^{-3}$	$7.59631 \cdot 10^{-3}$
31	3	$9.43812 \cdot 10^{-4}$	$4.65580 \cdot 10^{-3}$

(b) Errors for a varying number of intervals n

Table 1: Errors for smooth and non-smooth problems

Given that a considerable number of optimal control problems have non-smooth optimal solutions, because of path or final constraints, it is important to also consider this case. Therefore, Model 2.1 is examined. It is easy to check that for $t_f = 0.5$ the optimal force is given by $F(t) = \begin{cases} 10, & \text{if } t \leq 0.25, \\ -10, & \text{if } t > 0.25 \end{cases}$. This is a classical example of a *bang-bang* control, where the control is maximal at first and becomes minimal after some

switching time. To make the solution non-symmetric, the lower bound on the force is set to -5, i.e. the box constraint on the force becomes $-5 \leq F(t) \leq 10$. Thus, the switching point is located at $\frac{t_f}{3}$. The problem is solved with the same number of intervals n and collocation nodes m as before. Figure 11 compares the optimal numerical velocity with the exact optimal solution. The column $\|v^* - v\|_\infty$ of Table 1 gives the errors of the exact solutions compared to the numerical solutions evaluated at the nodes t_{ij}.

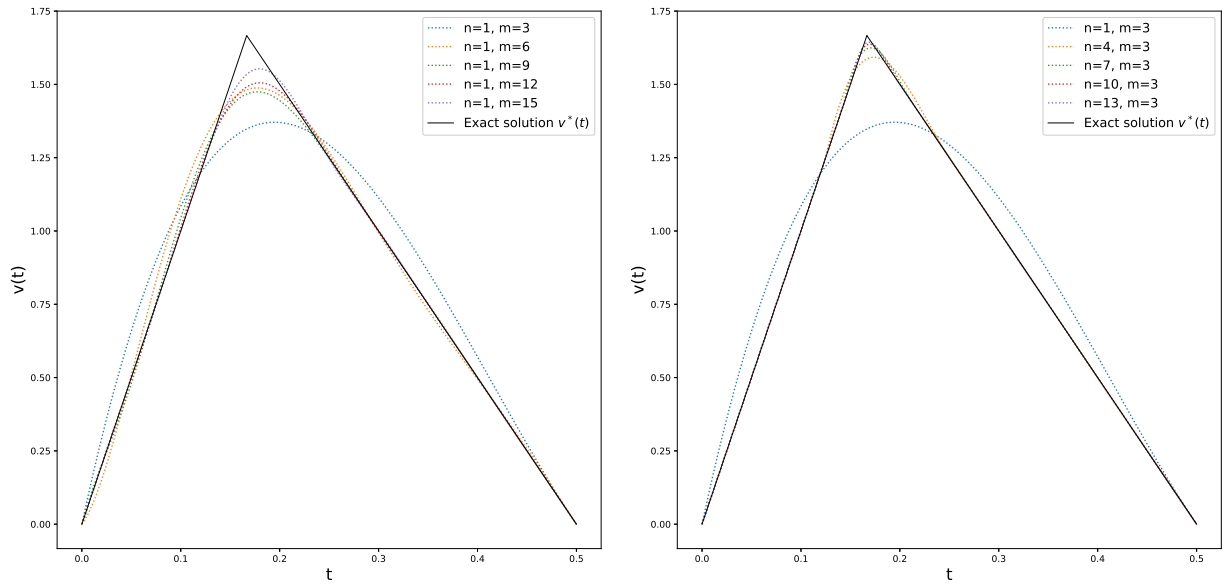


Figure 11: Optimal velocity for varying n and m of the non-smooth Model 2.1

Although increasing the polynomial degree m results in exponential convergence to the optimal solution for smooth problems, this does not apply to non-smooth problems. Table 1 displays that the error to the optimal solution only slightly decreases with an increase in the number of nodes m. In contrast, if the number of intervals n is increased, the error decreases significantly faster. Note that this is also due to the fact that the total number of variables $n \cdot m$ is larger in case of Table 1 (b) compared to Table 1 (a). Nevertheless, it is generally advantageous to increase the number of intervals in case of non-smooth problems and increase the polynomial degree in case of smooth problems.[47][59][56]

6.2.2 *h*-Methods

h-methods are mesh refinement algorithms that utilize the concept of increasing the number of intervals n, while imposing a polynomial of fixed degree m on each interval. These methods are basically described by Algorithm 6.4. The name stems from the fact that these methods adaptively change the positions of the intervals and their respective lengths h. *h*-methods do not achieve exponential convergence, since the polynomial degree is fixed. However, by choosing a rather high degree m, the convergence can become very rapid for smooth problems. It is noteworthy that these methods are also very capable of solving non-smooth optimal control problem as seen in Chapter 6.2.1. Furthermore, *h*-methods do not just increase the number of intervals and employ an equidistant grid. In the case of the non-smooth problem in Figure 11, it is possible to start with an equidistant mesh and insert a single interval in every iteration at the switching point. Thus, the error to the true switching time decays exponentially as a function of the number of

mesh iterations. By starting with an initial equidistant mesh with $n = 15$ and performing 5 mesh iterations, resulting in a final number of just 20 intervals, the error becomes $\|v^* - v\|_\infty = 1.66209 \cdot 10^{-8}$. This approach is clearly far superior to the previous results in Table 1(a) and Table 1(b).

In general, *h*-methods change the location and add mesh points in every iteration so that the mesh is dense where high accuracy is needed, such as at discontinuities, kinks, bends and steep sections. This process can cause numerical instabilities, because the interval lengths might become very small. Since rather low degree polynomials are used, they are very capable of reflecting non-smooth behavior. A variety of *h*-methods have been developed and can be found in the literature. These methods have shown to be very efficient in solving dynamic optimization problems because they satisfy the criteria mentioned in Chapter 6.1 and lead to very sparse NLPs. Note that number of non-zero elements in the Jacobian blocks (119) is $O(m^2)$. Therefore, the number of non-zeros in the full Jacobian is of order $O(m^2 n)$ for $O(nm)$ variables and thus, using local collocation results in sparse NLPs. In the following some *h*-methods are presented:[50][53][56]

In [50] a *multiresolution technique* was developed, which calculates the error between the current control trajectory and a trajectory based on non-oscillatory (ENO) interpolation. In this way, non-smooth section are detected and the mesh is refined. It also merges adjacent intervals, if the control is constant on both intervals. A similar multiresolution technique was developed in [51]. In [52] an integer programming technique is used to minimize the maximum error in each refinement iteration. Another approach used in direct[53] and indirect methods [54] is based on *density functions*. These methods iteratively generate meshes by investigating a density and its corresponding distribution function, which encode the curvature of the control trajectories. The basic idea of the density function and multiresolution approaches is presented in Chapter 6.3, where a novel mesh refinement algorithm that utilizes these is proposed.

6.2.3 *p*-Methods

p-methods are mesh refinement algorithms that iteratively increase the number of collocation nodes *m* and thus the polynomial degree. The number of intervals is usually set to 1, resulting in a single global high-degree polynomial. The name *p*-method stem from the degree *p* of the global polynomial. These methods offer very limited performance for general dynamic optimization problems, since they converge poorly for non-smooth problems. In the case of smooth problems, however, they achieve the aforementioned spectral convergence as a function of the number of collocation nodes, hence the name *pseudospectral methods*. In order to achieve this, *p*-methods utilize the roots of orthogonal polynomials, such as the *Legendre-Gauss (LG)*, *Legendre-Gauss-Radau (LGR)*, *flipped Legendre-Gauss-Radau (fLGR)* or *Legendre-Gauss-Lobatto (LGL)* points, as the nodes of an interpolating polynomial. Furthermore, pseudospectral methods can usually apply the *covector mapping principle* to relate the optimal solution resulting from the direct collocation approach with an indirect approach and thus obtain a costate estimation of the continuous problem.[55] In addition to poor convergence for non-smooth problems, these also provide a rather dense Jacobian, since the number of non-zero entries is $O(m^2)$ and the number of variables is $O(m)$ when using a single interval $n = 1$. Therefore, solving the resulting NLP is considerably more expensive than in *h*-methods.[30]

6.2.4 *hp*- and *ph*-Methods

hp- and *ph*-methods are hybrid pseudospectral methods, which combine the advantages of both the *h*- and *p*-methods. These adjust the degree of the polynomials as well as the number of intervals and their

placements. The nodes are, just like in p-methods, roots of orthogonal polynomials such as the LG, LGR, LGL or fLGR points. This approach is very flexible, since it allows exponential convergence for smooth sections and also effectively handles non-smooth behavior. Furthermore, hybrid methods usually yield direct costate estimations. hp-methods employ a p mesh refinement in sections that have been detected as smooth and add points and low degree polynomials in non-smooth sections, thereby performing a h-method. The difference between both types of hybrid methods is that in a ph-method the degree of the polynomial is increased first. Therefore, the algorithm aims to utilize the spectral convergence and splits intervals only if necessary. In contrast, the hp-method first divides intervals and only later increases the polynomial degree. These types of methods are the state-of-the-art adaptive mesh refinement algorithms and are implemented e.g. in the proprietary software package *GPOPS II* [30]. However, these methods have considerable drawbacks. Because hp- and ph-method effectively perform p- and h-methods, the sparsity of the Jacobian is worse than for pure h-methods. This results in large and more expensive computations. Furthermore, the NLP may be poorly conditioned since high degree polynomials are used. Another disadvantage is that the mesh refinement algorithm has a greater level of algorithmic complexity, than pure h-methods, since the error estimations must include both the polynomial degrees and interval lengths. It is therefore much more difficult to adjust the free parameters appropriately to achieve the desired convergence.[56][30][57]

6.3 L2-Boundary-Norm

As stated in Chapter 6.2.2, there are a lot of h-method that have already been developed. In this chapter, the novel h-method *L2-Boundary-Norm (L2BN)* is presented, that effectively uniformizes the control trajectory, similar to *density functions*.[53] The aim of this process is to achieve a distribution of the error that is approximately uniform. L2BN performs a consecutive bisection mesh refinement, similar to the *multiresolution* technique [50], to capture the non-smooth behavior of the problem and is able to converge rapidly to the exact solution as a function of the mesh iterations. Furthermore, the method is able to detect discontinuities, kinks, bends and steep sections. The algorithm shows promising runtimes, since subsequent initial values are obtained by interpolation of the previous optimal solution and the method itself requires only a runtime of $O(d_u n m^2)$. Since the h-method uses a fixed polynomial degree m on each interval, the degree has to be chosen appropriately for the specific problem. Note that it is possible to use rather high degree polynomials, e.g. $m \geq 5$ for smooth and well conditioned problems, while low degree polynomials with $m \leq 4$ can be used for poorly conditioned and non-smooth problems. Thus, the convergence of each NLP in the mesh refinement process can become of very high polynomial order for smooth problems. The termination of the mesh refinement algorithm is also guaranteed for smooth problems and under suitable convergence conditions, as will be demonstrated in this chapter. However, L2BN has several drawbacks: It does not use the spectral convergence of the fLGR points and therefore may require a lot of mesh iterations as well as many intervals. This can lead to numerical instabilities. Furthermore, the algorithm does not directly incorporate error estimates of the solution. Note that this may result in certain behavior not being handled appropriately by the algorithm. It is emphasized that this fact can be addressed by incorporating simulation software directly in the framework. Based on this, both a local simulation for each interval and a global simulation for the entire time horizon could be carried out, leading to very accurate error estimates. This is computationally expensive and could also be replaced by using analytical results on the

error or by re-evaluating the solution with a new mesh as shown in hybrid methods such as [46]. Overall, it is important to point out that these drawbacks can be eliminated by suitable extensions. Nonetheless, the proposed algorithm proves to be very efficient and produces small error terms for a variety of real-world applications and benchmark problems, as will be shown in Chapter 8.

6.3.1 Prerequisites and Related Work

To derive and motivate L2BN, the necessary prerequisites and related work are now described. At first, the definition of a density function is needed.[53]

Definition 6.1 (Density Function)**.** *A mesh density function is a non-negative function $\bar{f}(t)$ that satisfies*

$$\int_0^{t_f} \bar{f}(t)\,\mathrm{d}t = 1$$

and is zero (at most) at countably many points. The corresponding distribution function is given by

$$\bar{F}(t) = \int_0^{t} \bar{f}(\tau)\,\mathrm{d}\tau.$$

In the density function approach described in [53], the density function $\bar{f}(t)$ is chosen as the normalized curvature of a given control trajectory. Furthermore, the mesh $\mathcal{M} = \{t_0, t_1, \ldots, t_n\}$ is then obtained by the sequence satisfying

$$\int_{t_i}^{t_{i+1}} \bar{f}(\tau)\,\mathrm{d}\tau = \frac{1}{n},\ \ i = 0, \ldots, n-1 \tag{138}$$

with the initial mesh point $t_0 = 0$. In this way, the structure of the mesh $\mathcal{M}$ is directly encoded and imposed by the distribution function. Another prerequisite are the so-called *dyadic meshes.* A dyadic mesh contains a set of mesh points obtained by successive bisection of an initial equidistant mesh $\mathcal{M}_0 = V_0$ containing $n+1$ points. For $k \geq 0$, the k-th dyadic mesh V_k is given by

$$V_k = \left\{ t_f \frac{i}{2^k n} \middle| i = 0, \ldots, 2^k n \right\}. \tag{139}$$

Furthermore, the mesh $V_{k+1} = V_k \cup W_k$ with

$$W_k = \left\{ t_f \frac{2i+1}{2^{k+1} n} \middle| i = 0, \ldots, 2^k n - 1 \right\}. \tag{140}$$

W_k denotes all points that are added to V_{k+1}. In the k-th mesh iteration of the multiresolution technique in [50], the algorithm determines for all $\tau \in W_k$ whether an interpolation error at τ is greater than a given threshold. If so, τ and neighboring points from W_k are added to the current mesh $\mathcal{M}_k$. Since the algorithm iterates over the full set W_k and $\mathcal{M}_k$ does not necessarily contain the full mesh V_k, the method overcomes dyadic limitations that would result from simply testing all intermediate points in the current mesh $\mathcal{M}_k$.[50] Since the splitting condition of L2BN is constructed in a different way, it will be seen that L2BN does not need to overcome the dyadic limitations and can straightforwardly bisect intervals from the current mesh $\mathcal{M}_k$. Nevertheless, for L2BN it applies that $\mathcal{M}_k \subseteq V_k$ and the points with maximum resolution which are added to $\mathcal{M}_k$ stem from W_k.

6.3.2 On-Interval Condition

In the following two chapters the bisection conditions of L2BN are presented. These conditions are essentially split into two parts: The *on-interval / L2* condition and the *boundary* condition. The principle idea is to inspect the current solution of the control variables $\boldsymbol{u}(t)$ within each mesh interval $[t_i, t_{i+1}]$ of the mesh $\mathcal{M}_k$ individually and estimate the *change* and *curvature* over the entire interval. If the *change* or *curvature* is too large, the interval is bisected and thus, the point $\frac{t_i+t_{i+1}}{2}$ is added to the new mesh $\mathcal{M}_{k+1}$.
Let $[t_i, t_{i+1}]$ be an arbitrary interval on the mesh $\mathcal{M}_k$ and given the current optimal solution of a control variable $\hat{u}$ on this interval. The optimal solution is given by $m+1$ sample points at $t_i + \Delta t_i c_j$, $j = 0, \dots, m$ (setting $c_0 = 0$). The first step is to construct an interpolating polynomial $p(t)$ of the control variable $\hat{u}$ on the nominal interval $[0, 1]$, i.e. $p(c_j) = \hat{u}\left(t_i + \Delta t_i c_j\right)$, $j = 0, \dots, m$ and $p \in P^m$. This rescaling is performed because the length of the interval Δt_i should not affect the condition. Moreover, the interval length is actually the quantity that needs to be refined and the approach aims to uniformize the values of $\hat{u}$ on each interval. It will be seen that otherwise the estimates for the *change* and *curvature* would be proportional to the interval length, which is undesired and would result in a way weaker condition.
The next step is to approximate the *change* and *curvature* of p on the nominal interval $[0, 1]$. This is performed by evaluating norms of $\dot{p}$ and $\ddot{p}$, such as L^1, L^2 or L^∞. In this case, the L^q norms are given by $\|p\|_{L^q} := \left(\int_0^1 |p(t)|^q \, \mathrm{d}t\right)^{\frac{1}{q}}$. L^1 and L^∞ are not chosen, since oscillations in the functions have an huge impact on the L^1 norm and the essential supremum does not yield enough insights into the behavior on the interval. Therefore, the L^2 norm is used, because it is a good middle ground between the other two norms and offers excellent properties that will be demonstrated later.
Overall, the *on-interval / L2* condition says: If for a given interval $[t_i, t_{i+1}]$ exists at least one control variable $\hat{u}$, such that the interpolating polynomial $p(t)$ of $\hat{u}$ on the nominal interval $[0, 1]$ satisfies

$$\|\dot{p}(t)\|_{L^2} \geq TOL_1 \text{ or } \|\ddot{p}(t)\|_{L^2} \geq TOL_2 \tag{141}$$

for specified tolerances TOL_1, TOL_2, then the interval is bisected.

6.3.2.1 Fast Computation Now, a procedure is presented that allows for a rapid computation with exact precision of the *on-interval* condition for an arbitrary number of collocation nodes m. By construction $\deg(\dot{p}) = m-1, \deg(\ddot{p}) = m-2$ and thus $\deg(\dot{p}^2) = 2m-2, \deg(\ddot{p}^2) = 2m-4$ hold for $m \geq 2$. Note that if $m = 1$, the second derivative of p vanishes and the first is constant. By Theorem 3.5, the degree of exactness of the Radau quadrature is $2m-2$ for m nodes. This implies that both $\dot{p}(t)^2$ and $\ddot{p}(t)^2$ are exactly integrated for all m and hence

$$\|\dot{p}(t)\|_{L^2} = \left(\int_0^1 \dot{p}(t)^2 \, \mathrm{d}t\right)^{\frac{1}{2}} = \left(\sum_{k=1}^m b_k \dot{p}^2(c_k)\right)^{\frac{1}{2}} \text{ and } \|\ddot{p}(t)\|_{L^2} = \left(\int_0^1 \ddot{p}(t)^2 \, \mathrm{d}t\right)^{\frac{1}{2}} = \left(\sum_{k=1}^m b_k \ddot{p}^2(c_k)\right)^{\frac{1}{2}} \tag{142}$$

with b_k being the Radau quadrature weights for the interval $[0, 1]$. Clearly, p can be written as a Lagrange interpolating polynomial (Definition 3.1), i.e. $p(t) = \sum_{j=0}^m \hat{u}_j l_j(t)$ with $\hat{u}_j := \hat{u}(t_i + \Delta t_i c_j), c_0 = 0$ and the

basis polynomials $l_j(t) = \prod_{\substack{k=0 \\ k\neq j}}^{n} \frac{t-c_k}{c_j-c_k} \ \forall j = 0,\dots,n$. Thus, $\dot{p}(t) = \sum_{j=0}^{m} \hat{u}_j \dot{l}_j(t)$ and consequently

$$\|\dot{p}(t)\|_{L^2} = \left(\sum_{k=1}^{m} b_k \dot{p}^2(c_k)\right)^{\frac{1}{2}} = \left(\sum_{k=1}^{m} b_k \left(\sum_{j=0}^{m} \hat{u}_j \dot{l}_j(c_k)\right)^2\right)^{\frac{1}{2}} = \left(\sum_{k=1}^{m} b_k \left(\sum_{j=0}^{m} \hat{u}_j D^{(1)}_{kj}\right)^2\right)^{\frac{1}{2}}, \tag{143}$$

using the first differentiation matrix (Definition 3.2) $D^{(1)}$ of the rescaled fLGR points. Since this matrix is constant and can be precomputed, a very simple and fast algorithm is obtained that exactly evaluates the desired quantity $\|\dot{p}(t)\|_{L^2}$. Note that repeating the process for $\ddot{p}$ and using the second differentiation matrix $D^{(2)}$ yields an analogous formula. Utilizing the property $D^{(2)} = (D^{(1)})^2$, the following algorithm requiring $O(m^2)$ operations is obtained.

Algorithm 6.5: Fast On-Interval Computation

Input: Sample values $\hat{\boldsymbol{u}} = (\hat{u}(t_i), \hat{u}(t_i + \Delta t_i c_1), \dots, \hat{u}(t_i + \Delta t_i c_m))^T$, m-step Radau IIA collocation scheme and differentiation matrix $D^{(1)}$

Output: $\|\dot{p}(t)\|_{L^2}$, $\|\ddot{p}(t)\|_{L^2}$

$\boldsymbol{p}' \leftarrow D^{(1)}\hat{\boldsymbol{u}}$;
$\boldsymbol{q}' \leftarrow \left((p'_1)^2, \dots, (p'_m)^2\right)^T$;
$\|\dot{p}(t)\|_{L^2} \leftarrow \sqrt{\boldsymbol{b}^T \boldsymbol{q}'}$;
$\boldsymbol{p}'' \leftarrow D^{(1)}\boldsymbol{p}'$;
$\boldsymbol{q}'' \leftarrow \left((p''_1)^2, \dots, (p''_m)^2\right)^T$;
$\|\ddot{p}(t)\|_{L^2} \leftarrow \sqrt{\boldsymbol{b}^T \boldsymbol{q}''}$;
return $\|\dot{p}(t)\|_{L^2}, \|\ddot{p}(t)\|_{L^2}$;

6.3.2.2 Convergence and Termination

Another useful property of the on-interval condition is that the mesh refinement algorithm terminates for smooth problems and under suitable convergence properties, i.e. only finitely many refinements are needed until $\|\dot{p}(t)\|_{L^2} < TOL_1$ and $\|\ddot{p}(t)\|_{L^2} < TOL_2$ hold. This fact can be deduced from the following lemma, where the L^2 norm is again taken on the interval $[0,1]$.

Lemma 6.1. *Let $p \in P^m$ be a polynomial and $p_b(t) = p\left(\frac{t}{2}\right) + \varepsilon(t)$ be a variation of $p\left(\frac{t}{2}\right)$ by $\varepsilon \in P^m$, then for a given $\delta < 1$*

$$\frac{\|\dot{p}_b(t)\|_{L^2}}{\|\dot{p}(t)\|_{L^2}} < \delta < 1, \quad \text{if} \quad \|\dot{\varepsilon}(t)\|_{L^2} < \frac{\sqrt{2}\delta - 1}{2}\left\|\dot{p}\left(\frac{t}{2}\right)\right\|_{L^2}. \tag{144}$$

Proof.

$$\|\dot{p}(t)\|^2_{L^2} = \int_0^1 \dot{p}(t)^2 \,\mathrm{d}t = \int_0^{\frac{1}{2}} \dot{p}(t)^2 \,\mathrm{d}t + \int_{\frac{1}{2}}^1 \dot{p}(t)^2 \,\mathrm{d}t = \frac{1}{2}\left(\int_0^1 \dot{p}\left(\frac{t}{2}\right)^2 \,\mathrm{d}t + \int_0^1 \dot{p}\left(\frac{t+1}{2}\right)^2 \,\mathrm{d}t\right)$$

$$\implies \|\dot{p}(t)\|_{L^2} = \frac{1}{\sqrt{2}}\left(\int_0^1 \dot{p}\left(\frac{t}{2}\right)^2 \,\mathrm{d}t + \int_0^1 \dot{p}\left(\frac{t+1}{2}\right)^2 \,\mathrm{d}t\right)^{\frac{1}{2}} = \frac{1}{\sqrt{2}}\left(\left\|\dot{p}\left(\frac{t}{2}\right)\right\|^2_{L^2} + \left\|\dot{p}\left(\frac{t+1}{2}\right)\right\|^2_{L^2}\right)^{\frac{1}{2}} \tag{145}$$

Also $\dot{p}_b(t) = \frac{1}{2}\dot{p}\left(\frac{t}{2}\right) + \dot{\varepsilon}(t)$ and by the *Minkowski inequality* $\|\dot{p}_b(t)\|_{L^2} \leq \frac{1}{2}\left\|\dot{p}\left(\frac{t}{2}\right)\right\|_{L^2} + \|\dot{\varepsilon}(t)\|_{L^2}$, implying

$$\frac{\|\dot{p}_b(t)\|_{L^2}}{\|\dot{p}(t)\|_{L^2}} \leq \frac{\frac{1}{2}\left\|\dot{p}\left(\frac{t}{2}\right)\right\|_{L^2} + \|\dot{\varepsilon}(t)\|_{L^2}}{\frac{1}{\sqrt{2}}\left(\left\|\dot{p}\left(\frac{t}{2}\right)\right\|_{L^2}^2 + \left\|\dot{p}\left(\frac{t+1}{2}\right)\right\|_{L^2}^2\right)^{\frac{1}{2}}} \leq \frac{1}{\sqrt{2}}\frac{\left\|\dot{p}\left(\frac{t}{2}\right)\right\|_{L^2} + 2\|\dot{\varepsilon}(t)\|_{L^2}}{\left\|\dot{p}\left(\frac{t}{2}\right)\right\|_{L^2}} = \frac{1}{\sqrt{2}}\left(1 + \frac{2\|\dot{\varepsilon}(t)\|_{L^2}}{\left\|\dot{p}\left(\frac{t}{2}\right)\right\|_{L^2}}\right) \tag{146}$$

and thus the preposition follows by rearranging terms. □

Assuming that a polynomial $p(t)$ violates (141), then the interval is bisected and in the next iteration both subintervals are investigated. Without loss of generality, the value of the on-interval condition on the first subinterval is inspected, since the case for the second interval is virtually identical with $p_b(t) = p\left(\frac{t+1}{2}\right) + \varepsilon(t)$. The new polynomial can be written in the form $p_b(t) = p\left(\frac{t}{2}\right) + \varepsilon(t)$, because rescaling the values in $[0, \frac{1}{2}]$ of the old polynomial p to $[0, 1]$, as mandatory for the L2-condition, yields $p\left(\frac{t}{2}\right)$ and furthermore, an error ε between the old and new polynomials must be taken into account. By Lemma 6.1 the new L^2 norm on the subinterval is smaller than the old one, if $\|\dot{\varepsilon}(t)\|_{L^2} < \frac{\sqrt{2}\delta - 1}{2}\|\dot{p}_b(t)\|_{L^2}$. This means that the on-interval condition leads to a decreasing sequence of L^2 norms if the difference of the polynomials $p_b(t), p\left(\frac{t}{2}\right)$ and thus the difference of the old and new control variables on the subinterval is bounded proportional to the change on the interval itself. Assuming the convergence of the orthogonal collocation method, the errors $\varepsilon, \dot{\varepsilon}$ and $\ddot{\varepsilon}$ approach 0 if the problem is smooth. Since the polynomial in Lemma 6.1 is arbitrary, the same considerations also hold for the second derivative $\ddot{p}$ on either subinterval. Backed by numerical evidence in Chapter 8, this shows the termination of the method after a finite number of refinement iterations under suitable convergence conditions and for smooth problems.

It is important to note that the property only holds for smooth problems. This can be seen by investigating the sample values $\hat{u} = (0, 0, 1)^T$ for the 2-step Radau scheme and assuming the optimal solution is 0 for $t < 1$ and 1 for $t \geq 1$. In this way, the bisection of the interval will yield the first subinterval $\hat{u}_{b1} = (0, 0, 0)^T$ and the second $\hat{u}_{b2} = (0, 0, 1)^T = \hat{u}$, which generates the same polynomial and thus leads to the same L^2 norm. This process will be repeated for each mesh iteration k.

However, for smooth controls this would not be possible since the function is continuous. Furthermore, heuristic convergence constants can be obtained for the on-interval condition. Assuming that the new polynomial is equal to the previous one on either of the two subintervals, e.g. $p_b(t) = p\left(\frac{t}{2}\right)$, the convergence becomes $\frac{\|\dot{p}_b(t)\|_{L^2}}{\|\dot{p}(t)\|_{L^2}} \leq \frac{1}{\sqrt{2}}$. If additionally the norms of both subintervals are equal, i.e. $\left\|\dot{p}\left(\frac{t}{2}\right)\right\|_{L^2} = \left\|\dot{p}\left(\frac{t+1}{2}\right)\right\|_{L^2}$ and the polynomial does not change between iterations, then $\frac{\|\dot{p}_b(t)\|_{L^2}}{\|\dot{p}(t)\|_{L^2}} \leq \frac{1}{2}$. Although the constants are obtained by assuming that $\varepsilon \equiv 0$, these considerations can actually be observed for example problems and will be backed by numerical evidence in Chapter 8.

Moreover, this property allows to interpret the method as converging to a final mesh with n intervals, which is approximately generated by a density function. Given a problem with a scalar optimal smooth control trajectory $u^*(t)$. Since $\|\dot{p}(t)\|_{L^2} < TOL_1$ and p is an approximation of u, the equation

$$\int_{t_i}^{t_{i+1}} \frac{\dot{u}^*(t)^2}{\Delta t_i}\,\mathrm{d}t \leq TOL_1^2, \;\; i = 0, \ldots, n-1 \tag{147}$$

is obtained by substitution. As constructed by scaling to the nominal interval $[0, 1]$, (147) shows that the on-interval condition is independent of the interval length because Δt_i is canceled out. Thus, only the

values of the function are taken into account. By multiplying with $\frac{1}{n \cdot TOL_1^2}$ the inequality

$$\int_{t_i}^{t_{i+1}} \frac{\dot{u}^*(t)^2}{n \cdot TOL_1^2 \cdot \Delta t_i} \, \mathrm{d}t \leq \frac{1}{n} \;\; i = 0, \ldots, n-1 \tag{148}$$

results, which is very similar to density functions mesh refinement (138). Moreover, the inequality effectively shows that the *on-interval* condition is at least as strong as some density function (148). Clearly, this approximation of a density function is only of theoretical nature, since the method iteratively bisects the mesh and thus $\mathcal{M}_n \subseteq V_n$ for all $n \geq 0$. Such a limitation would not apply when actually using density functions. Furthermore, the proposed approach uses other conditions, such as the L^2 norm of the second derivative, applies the conditions to any number of control variables, and also the set of previously obtained mesh points $\mathcal{M}_{n-1}$ is always contained in $\mathcal{M}_n$. Nevertheless, (148) offers a very close relation between the proposed L2BN and density function approaches.

6.3.3 Boundary Condition

The proposed method could be implemented using only the *on-interval* condition. However, this approach has a critical flaw that will be analyzed in the following.

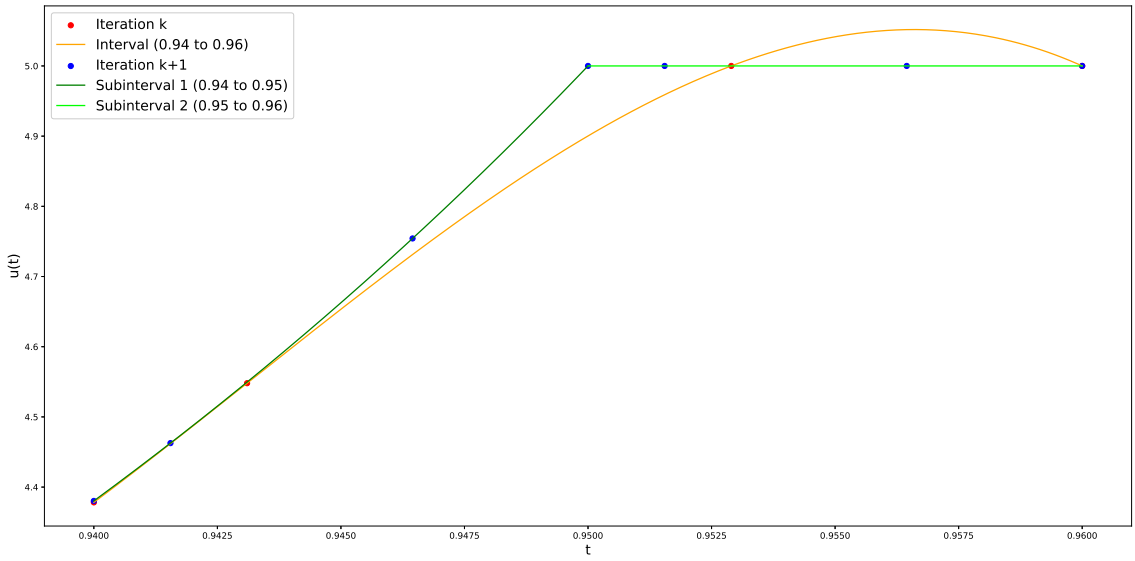


Figure 12: Corner in the control trajectory due to the *on-interval* condition

Consider Figure 12, where the interval $[0.94, 0.96]$ has been bisected into the first subinterval $[0.94, 0.95]$ and the second subinterval $[0.95, 0.96]$ with the *on-interval* condition. Furthermore, the interpolating polynomials on each interval based on the 3-step Radau collocation method are visualized. It can be seen that the first subinterval still has a large slope on the entire interval and thus this interval is likely to be bisected again. The second subinterval, on the other hand, is basically constant on the entire interval and therefore will not trigger another bisection. This is a critical flaw since the boundary of both intervals describes a sharp corner. For this reason, the algorithm should achieve a higher resolution and eliminate the corner or at least find the approximately correct time at which the corner is located. Note that this may

not be at exactly 0.95. To eliminate this drawback, a certain *boundary* condition is introduced. For this the first and second derivatives of the interpolating polynomials p_1 and p_2 for subinterval 1 and 2 are compared on the boundary. The measure could be taken as either the relative or absolute error, but both fail at certain tasks. The error measure should behave like the relative error for large values, while it should behave like the absolute error for small values. This has the advantage that minor oscillations in the solution do not trigger a bisection as well as that the magnitude of the values is taken into account. The error term that is used in the *boundary* condition is the so-called *P1 error / plus-1 error*, which satisfies both conditions: $P_1(x,y) := \frac{|x-y|}{1+\min\{|x|,|y|\}}$. Overall, the *boundary* condition says: Given two adjacent intervals with their respective interpolating polynomials p, q, then both intervals are bisected if

$$\frac{|\dot{p}(t_i) - \dot{q}(t_i)|}{1 + \min\{|\dot{p}(t_i)|, |\dot{q}(t_i)|\}} \geq CTOL_1 \quad \text{or} \quad \frac{|\ddot{p}(t_i) - \ddot{q}(t_i)|}{1 + \min\{|\ddot{p}(t_i)|, |\ddot{q}(t_i)|\}} \geq CTOL_2 \tag{149}$$

for the common boundary point t_i and specified corner tolerances $CTOL_1, CTOL_2$. It is important to point out that this condition does not interfere with the termination for smooth problems derived in Chapter 6.3.2.2, since the control trajectory converges to the exact smooth solution and therefore will not contain corners or kinks.

6.3.4 Resulting Algorithm

The previous considerations and the combination of the *on-interval* and *boundary* conditions lead to the proposed mesh refinement algorithm *L2-Boundary-Norm (L2BN)*, which is an implementation of the generic iterative mesh refinement algorithm (Algorithm 6.4). The version of L2BN implemented in the framework *GDOPT* has five free parameters. These are the maximum number of mesh iterations k_{max} and the interpolation method, e.g. linear splines or polynomial interpolation. In addition, the number of full bisections $B \in \mathbb{N}_0$ must be provided. This parameter forces all mesh intervals to be bisected for the first B mesh iterations. In this way, the algorithm is able to solve the initial NLP on a very course mesh and even for poor initial guesses extremely rapidly. Moreover, this parameter allows to set a minimum desired resolution, even if the initial NLP cannot be solved with this resolution due to initial guesses or poor conditioning. Furthermore, the level of the mesh $\Lambda \in \mathbb{R}$ and the corner tolerance $C > 0$ must be provided. The level Λ directly affects the tolerances of the *on-interval* condition. For a given control variable $\hat{u}$, the tolerances are given by $TOL_1(\hat{u}) = \frac{\text{range}(\hat{u})}{n} 10^{-\Lambda}$ and $TOL_2(\hat{u}) = \frac{\text{range}(\hat{u})}{2n} 10^{-\Lambda}$, with the number of initial intervals n and $\text{range}(\hat{u}) := \max_{t\in[t_0,t_f]} \hat{u}(t) - \min_{t\in[t_0,t_f]} \hat{u}(t)$. The level is defined on a logarithmic scale, so increasing it by 1 makes the *on-interval* condition 10 times more sensitive. The default value is $\Lambda = 0$ and the typical range is $-2.5 \leq \Lambda \leq 2.5$. The parameter C determines the corner tolerances $CTOL_1$ and $CTOL_2$ as $C = CTOL_1 = CTOL_2$. The default value is $C = 0.1$ and the typical range is $0.05 \leq C \leq 0.5$. To make the algorithm more stable, this version of the algorithm also bisects both adjacent intervals, if condition (141) is triggered. In addition, in the presented version only the control variables are analyzed. Note that the same process can be performed for state variables, but may be inefficient. The total runtime of each refinement iteration is $O(d_u n m^2)$, which is very rapid, as can be seen from the fact that the number of discretized control variables is $d_u n m$. The principle workflow of L2BN is given by:

Algorithm 6.6: L2-Boundary-Norm (L2BN)

Input: GDOP (Definition 2.2), guesses $\boldsymbol{x}_{ij}$, $\boldsymbol{u}_{ij}$, $\boldsymbol{p}$, mesh $\mathcal{M}_0$, number of collocation nodes m, mesh iterations k_{max}, interpolation method, mesh level Λ, corner tolerance C, full bisections B

Output: $\boldsymbol{x}^*_{ij}, \boldsymbol{u}^*_{ij}, \boldsymbol{p}^*$

$k \leftarrow 0$;
$\boldsymbol{x}^*_{ij}, \boldsymbol{u}^*_{ij}, \boldsymbol{p}^* \leftarrow$ Solve the dGDOP on $\mathcal{M}_0$ with m collocation nodes and guess $\boldsymbol{x}_{ij}$, $\boldsymbol{u}_{ij}$, $\boldsymbol{p}$;
$k \leftarrow k + 1$;
while $k \leq k_{max}$ **do**
  if $k \leq B$ **then**
    $S \leftarrow \{0, \ldots, n\}$;
  else
    $S \leftarrow \{\}$;
    for $i = 0, \ldots, n$ **do**
      for $d = 1, \ldots, d_{\boldsymbol{u}}$ **do**
        $\|\dot{p}(t)\|_{L^2}, \|\ddot{p}(t)\|_{L^2} \leftarrow$ Calculate the L^2 norms of $u^{(d)}$ on interval i with Algorithm 6.5;
        if $\|\dot{p}(t)\|_{L^2} \geq TOL_1\left(u^{(d)}\right) \vee \|\ddot{p}(t)\|_{L^2} \geq TOL_2\left(u^{(d)}\right)$ **then**
          $S \leftarrow S \cup \{i-1, i, i+1\}$;
        end
        if $i > 0$ **then**
          $\dot{p}(t_i), \ddot{p}(t_i), \dot{q}(t_i), \ddot{q}(t_i) \leftarrow$ Evaluate the 1st and 2nd derivatives of the interpolating polynomials of $u^{(d)}$ on the intervals i and $i-1$ at their common boundary point t_i;
          if $\frac{|\dot{p}(t_i) - \dot{q}(t_i)|}{1+\min\{|\dot{p}(t_i)|, |\dot{q}(t_i)|\}} \geq C \vee \frac{|\ddot{p}(t_i) - \ddot{q}(t_i)|}{1+\min\{|\ddot{p}(t_i)|, |\ddot{q}(t_i)|\}} \geq C$ **then**
            $S \leftarrow S \cup \{i-1, i\}$;
          end
        end
      end
    end
    if $S == \{\}$ **then**
      return $\boldsymbol{x}^*_{ij}, \boldsymbol{u}^*_{ij}, \boldsymbol{p}^*$;
    end
  $\hat{S}_k \leftarrow$ Bisect the intervals in S;
  $\mathcal{M}_{k+1} \leftarrow \mathcal{M}_k \cup \hat{S}_k$;
  $\boldsymbol{x}_{ij}, \boldsymbol{u}_{ij}, \boldsymbol{p} \leftarrow$ Interpolate $\boldsymbol{x}^*_{ij}, \boldsymbol{u}^*_{ij}, \boldsymbol{p}^*$ on $\mathcal{M}_{k+1}$ according to the interpolation method;
  $\boldsymbol{x}^*_{ij}, \boldsymbol{u}^*_{ij}, \boldsymbol{p}^* \leftarrow$ Solve the dGDOP on $\mathcal{M}_{k+1}$ with m collocation nodes and guess $\boldsymbol{x}_{ij}$, $\boldsymbol{u}_{ij}$, $\boldsymbol{p}$;
  $k \leftarrow k + 1$;
end
return $\boldsymbol{x}^*_{ij}, \boldsymbol{u}^*_{ij}, \boldsymbol{p}^*$;

7 Framework - GDOPT

7.1 Overview of the Framework

This chapter presents a basic overview of the proposed open source framework *GDOPT (General Dynamic Optimizer)*, which implements the novel mesh refinement algorithm *L2-Boundary-Norm (L2BN)* (Algorithm 6.6). *GDOPT* is licensed under LGPL-3.0 and publicly available on *GitHub*.[60] The framework is split into two components: The Python frontend *gdopt* that allows expressive and accessible modeling of GDOPs and the C++ backend *libgdopt* that performs the computationally intensive task of solving the large-scale nonlinear problems with *Ipopt*[1]. Furthermore, *Ipopt* itself relies on efficient linear solvers such as the supported *MUMPS*[2] and *HSL*[3] solvers. The principle workflows in *GDOPT* are visualized in Figure 13.

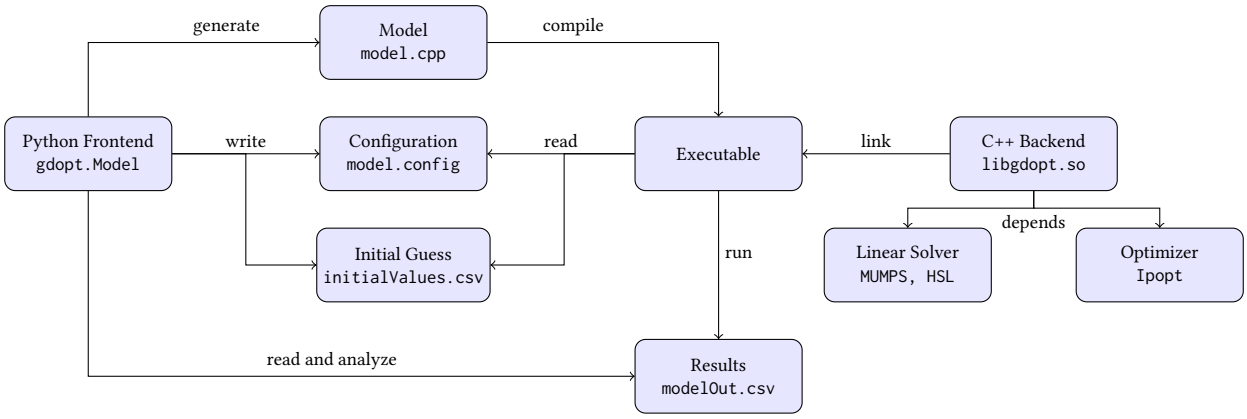


Figure 13: Overview of principal workflows in *GDOPT*

7.1.1 Modeling

The first step is to model a GDOP in the Python frontend using the package *gdopt* as shown in Appendix H for Model 2.2. In addition to the GDOP itself, this modeling environment allows the user to provide initial guesses and nominal values for the variables and functions as well as many flags and options. A key feature of *gdopt* is that it utilizes the Python interface of the C++ library *SymEngine*[4], which allows for fast symbolic handling and manipulations in the frontend. Note that modeling with *gdopt* is very extensive and therefore not within the scope of this thesis. For a detailed introduction to modeling and an overview of all flags, it is recommended to examine the *GDOPT User's Guide*.[61] Currently only a guide for *GDOPT v.0.1.3* exists, thus several new features and flags are missing in the guide.

7.1.2 Code Generation

When the modeling of the GDOP is completed, two pipelines must be accessed to provide the model to the backend. At first, the pipeline `generate` calculates the first and second derivatives as well as the adjacency structures of all functions and derivatives with *SymEngine*. Because the derivative calculations are performed symbolically and zeros in the derivatives are directly detected, there is no overestimate on the number of non-zero elements. This is a major advantage over other tools. In *GPOPS II* the sparsity pattern of the Hessian is calculated as struct $\left(S_J^T S_J\right)$ with S_J being the sparsity of the Jacobian and furthermore, the Hessian is only calculated numerically with finite differences.[30] Additionally, *SymEngine* is used to

find *Common Subexpressions (CSE)* for all functions and calculated derivatives. This has a huge impact on the evaluation of the symbolic derivatives, since no calculation needs to be performed multiple times and thus, the execution time of the framework is reduced by a large amount. After completing the derivative and adjacency calculations, the entire model is translated to the equivalent backend C++ formulation. For every function of the model a class is generated, which includes the adjacency structures for the first and second derivatives `adj`, `adjDiff`, a method for the evaluation of the function `eval` and methods for the first and second derivatives `evalDiff`, `evalDiff2`. All function evaluations use the previously obtained CSE to reduce the number of calculations per method. This process can be seen in Appendix K, where the generated first dynamic equation of Model 2.2 is displayed. Note that the elements of the adjacency structures correspond to the entries of the derivative vectors and indicate the indices of the variables with respect to which the derivatives are computed. Besides these necessary components of the GDOP, nominal values for the variables and functions are also written to the C++ file, if provided by the user. *GDOPT* is capable of including nominal values and appropriately scales the variables, constraints and the objective of the NLP with the builtin Ipopt interface. It is of high importance that all variables and functions are approximately $O(1)$ in order to have a well-conditioned NLP. The generated C++ file `model.cpp` is then compiled while linking with the backend library *libgdopt*. The standard compiler is GCC with default flags `-O3 -ffast-math`.

7.1.3 Optimization

In order to run the resulting executable, the pipeline `optimize` must be accessed. This pipeline is used to generate a configuration file as well as to provide an initial guess to the optimizer. The config and the initial guess will then be read by the executable at runtime. Note that `generate` and `optimize` are split into two pipelines such that parameters of the model can be changed after compilation. The corresponding frontend method takes several arguments, including the final time t_f, number of intervals n, number of collocation nodes m, a dictionary of flags and a dictionary of mesh flags. All of these are written to the configuration file `model.config`, which follows a simple macro-like syntax. An example configuration file for Model 2.2 is given in Appendix L. Most of the flags or the entries in the configuration respectively are straightforward to comprehend. However, some of the entries will be further emphasized. The `REFINEMENT_METHOD` determines the way in which the new initial values are interpolated after a mesh iteration, e.g. linear splines or polynomial interpolation. In the `[constant derivatives]` section, several values are set to notify Ipopt if some of the NLP derivatives are constant. Another important section is `[optionals: ipopt flags]`, where optional Ipopt flags will be written. In this case, the flags set in the frontend cause Ipopt to switch from the default `adaptive` to the `monotone` μ-Strategy with $\mu_0 = 10^{-16}$ for all refinements, i.e. every iteration except the very first. Note that such a small initial value of the barrier parameter μ results in the barrier problem being basically equal to the original NLP. Since it is also assumed that the new interpolated initial values are almost optimal, only very few iterations are needed to obtain a new optimal solution. This strategy is highly recommended when using adaptive mesh refinement algorithms. In addition to the predefined flags, the config can also contain so-called *RuntimeParameters*, which correspond to parameters of standard modeling environments. These custom constants can be used anywhere in the modeling process, e.g. in functions, as starting values, nominal values, guesses, and are generated into a global variable in `model.cpp`. This makes it possible to change values in the model without recompiling, which

is useful in many cases, such as benchmarks or parameter sweeps. In addition to the configuration, the `optimize` pipeline also obtains initial values for the states and writes them to the file `initialValues.csv`. This is done by solving the IVP using the initial guesses for the control variables and parameters provided by the user, i.e. the initial values of the states are obtained by simulating the dynamics of the problem. By default, the simulation is performed with the Python package *SciPy*[5], which contains several integrators such as *Radau5*, *BDF* or *LSODA*. Other options for initialization are to set all values to a constant or to use one of the explicit integrators implemented in the backend itself, i.e. *explicit Euler* or *RK4*. Note that these may lead to initial guesses that are unusable due to poor conditioning or stiffness, but are very fast compared to solving the IVP with *SciPy*. If the previous steps have been completed, the executable can be run. The executable reads in the configuration and initial guesses and then performs the optimization. If the optimization is successful, the optimal values are written to the file `modelOut.csv`.

7.1.4 Results and Analysis

The optimal solution `modelOut.csv` is automatically read by the frontend. This gives the user direct access to the optimal trajectories. In addition, *gdopt* implements extensive plotting features for standard and parametric plots, mesh refinement plots, sparsity patterns, etc. using the Python package *matplotlib*[8]. Some of the native plotting capabilities of *GDOPT* are presented in Chapter 8, but can also be found in the *GDOPT User's Guide* [61].

7.2 libgdopt

The following section gives an overview of the solver infrastructure and the components of the C++ backend *libgdopt*. In essence, *libgdopt* is an extended implementation of Algorithm 6.6, where each of the NLPs is solved with an Ipopt implementation of the dGDOP. Note that the interfaced Python frontend can be exchanged for other interfaces that provide the necessary derivatives, the configuration and possibly initial guesses to *libgdopt*. Moreover, an interface with a sophisticated modeling environment would allow to solve more general GDOPs, where the DAE system must not be semi-explicit by default. This is the case since modeling software can transform an implicit DAE system into an explicit DAE system as stated in Chapter 2.2.1.1. Additionally, such an extension would allow for faster initial guesses, since the simulation could be performed with the generated code and called directly from *libgdopt* instead of from the frontend. This would also make it possible to use simulation-based error estimates for the current solution and to refine certain intervals based on these errors.

7.2.1 Helper Classes and Structures

In Figure 14 a simplified class diagram of *libgdopt* is depicted. The library has two adjacency structures `Adjacency` and `AdjacencyDiff` that contain the indices of the non-zero first and second derivatives of a function. These correspond to the adjacency structure of the continuous problem. The class `Expression` holds both as an attribute and defines the methods `eval`, `evalDiff` and `evalDiff2`, which are the evaluation of the function and the first and second derivatives, respectively. The output of `evalDiff` and `evalDiff2` is sorted corresponding to the sorting of the adjacency structures. An example `Expression` can be seen in Appendix K, where the generated code for the first dynamic equation of Model 2.2 is displayed. In addition,

a class `Constraint` extends `Expression` by adding lower and upper bounds. Both classes are utilized in the `Problem` class. `Problem` is essentially a direct translation of the continuous GDOP (Definition 2.2) and contains callback functions for each mathematical expression used in the GDOP. It also holds the variable bounds and nominal values of the functions and variables.

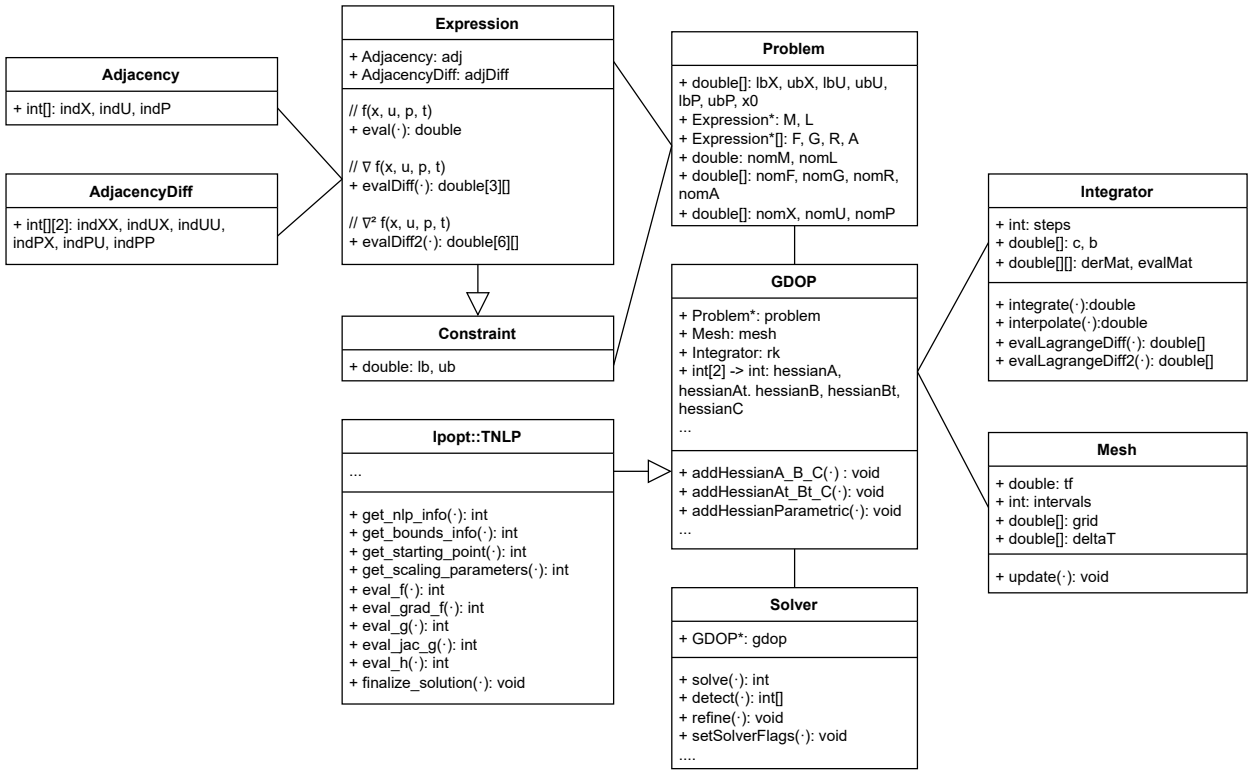


Figure 14: Simplified class diagram of *libgdopt*

The `Integrator` class contains all coefficients of a m-stage Radau IIA collocation scheme. These have been calculated with the construction script presented in Appendix E and are hard-coded into the class. Furthermore, the `Integrator` is able to perform the Radau quadrature rule with the method `integrate` and `evalLagrangeDiff` and `evalLagrangeDiff2` are used to evaluate the first and second derivatives of the Lagrange interpolating polynomials at the collocation nodes. The class `Mesh` contains all values related to the current mesh such as t_i, Δt_i, the number of intervals n and the final time t_f. It is possible to `update` the mesh based on the index set S of intervals to be bisected (Algorithm 6.6).

7.2.2 Ipopt Implementation

The class `GDOP` is an implementation of the already presented virtual base class `Ipopt::TNLP` (Chapter 4.3.1). In a nutshell, `GDOP` implements a single dGDOP in Ipopt and thus forms a complete algorithm for solving dynamic optimization problems without mesh refinement. The class has a continuous problem, a mesh and a Radau IIA collocation scheme as attributes. Furthermore, the virtual methods of `Ipopt::TNLP` are implemented according to the derived dGDOP problem structure (Definition 5.1) and its derivatives (Chapter 5.2). The computations inside the code are very complicated and therefore not in the scope of this thesis. Nevertheless, the most important procedures are explained at a high level. In general, each

function or derivative evaluation of the NLP can be reduced to values of the mesh and integrator or the evaluation of a continuous function or derivative at a given point $\boldsymbol{z}_{i,j} := (\boldsymbol{x}_{i,j}, \boldsymbol{u}_{i,j}, \boldsymbol{p}, t_{i,j})$. Therefore, the instances of the `Constraint` and `Expression` classes are evaluated using the presented callback methods. Because the evaluations are purely symbolic, the quality of the provided derivatives is very high compared to approaches with finite differences. Additionally, *Automatic Differentiation (AD)* is not employed, as the proposed framework already incorporates code generation and compilation, which are time-consuming processes. Consequently, the calculation of symbolic derivatives are inexpensive, given that these are performed prior to the compilation and models for dynamic optimization are quite small. Note that since the adjacency structure is minimal and CSE are used, the evaluations of the callback functions are extremely rapid. In addition, the possibly sparse derivatives of the continuous problem are heavily exploited, because the derivatives are vectorized and evaluated simultaneously. However, the constraints itself are not vectorized. This extension would be beneficial, because then the CSE of all constraints could be considered, which results in even less calculations. Another important process is to supply Ipopt with derivative sparsity patterns. As mentioned in Chapter 4.3.1, the Jacobian and the Hessian must be provided in coordinate format. *libgdopt* performs this process in two steps. First, the concrete sparsity pattern of the problem is discovered and saved. After that, only the values of the matrices are provided in each Ipopt iteration. For the Jacobian, this process is quite simple, because the sparsity pattern can be easily constructed by iterating over all constraints (113) in order. On the other hand, the Hessian sparsity is generated in several stages. However, this process only takes time proportional to the number of non-zeros in the Hessian, which is obviously optimal. At first, the sparsity patterns of the block matrices $A, \hat{A}, B, \hat{B}, C$ (125) are obtained as dense matrices with entries 0 or 1. By iterating over each dense block matrix, a hashmap is constructed, which holds all non-zero derivative indices and maps the variable pair to the index in the coordinate format vector. These maps can be found in Figure 14 as `hessianA`, ..., `hessianC`. The hashmaps define the entire sparse Hessian and by using index shifts and pointer arithmetic, the entire Hessian sparsity pattern can be constructed.

7.2.3 Solving

Similar to the considerations that motivated the prototype mesh refinement algorithm (Algorithm 6.4), a basic wrapper is constructed around the solving of the `GDOP`. This is performed by the class `Solver`, which has a `GDOP` instance as a member. Moreover, `Solver` has the core method `solve`, which resembles the L2BN mesh refinement algorithm (Algorithm 6.6). Utilizing this method, the full execution process can be described: At first, the generated configuration file is read and all global variables and *RuntimeParameters* are set. The specific `Problem`, the initial `Mesh` as well as the `Integrator` and corresponding initial `GDOP` are instantiated. Based on the `GDOP`, a `Solver` instance is created, which calls `solve`. Firstly, this method sets all Ipopt flags with `setSolverFlags` and solves the `GDOP` with the initial guesses. After that, the mesh refinement of Algorithm 6.6 is performed. All intervals that have to be refined are detected with `detect`. If the set of interval indices S is empty, the algorithm terminates. Otherwise the mesh method `update` is called and the previous optimal solution is interpolated according to the set interpolation method with `refine`. Then, a new `GDOP` is instantiated, the initial guess and solver flags are set, and the `GDOP` is solved again. This process is repeated until the number of maximum mesh iterations is exceeded or the set of interval indices is empty. Finally, `finalize_solution` is called and the optimal values are written to a file.

8 Performance of the Framework

GDOPT and its novel mesh refinement algorithm L2BN are now applied to four example problems taken from the open literature. All GDOPT problem implementations are part of the GDOPT GitHub repository [60]. The first example is the optimization of an oil shale pyrolysis[12], that has been introduced in Chapter 2.1.2. This problem shows the ability of GDOPT to converge to the optimal solution for poor initial guesses. Furthermore, the error between the optimal solution and a resimulation with OpenModelica is considered, to investigate the quality and accuracy of the solution provided. The second example is the hypersensitive optimal control problem (Model A.1) found in [46] and demonstrates that L2BN can capture the hypersensitive nature of the problem and rapidly converge to the true optimal solution using only a small final mesh. The third example is the optimization of a diesel-electric powertrain[18] and illustrates that GDOPT can reproduce optimal solutions for real physical applications. Furthermore, GDOPT is compared to the dynamic optimization solver implemented in OpenModelica[17], showing the striking advantages of the proposed mesh refinement algorithm as well as the efficiency of the implementation itself. The fourth example is the reentry trajectory optimization of a reusable launch vehicle, i.e. space shuttle, taken from [58]. It also shows the ability of GDOPT to solve real physical systems, reproduce results from the literature, and moreover, incorporate nominal values to appropriate scale the NLPs and converge to meaningful, realistic optimal solutions. All problems are computed on a platform with a 4.9 GHz Intel Core i5-12600KF, 32 GB 4800 MHz DDR5 RAM, running on Ubuntu 22.04.3 and using GCC v11.4.0 with flags `-O3 -ffast-math` for compilation.

8.1 Oil Shale Pyrolysis

Consider the *Oil Shale Pyrolysis* (Model 2.2) studied in Chapter 2.1.2. The GDOPT implementation of the model can be found in Appendix H. The L2BN mesh refinement (Algorithm 6.6) is run with $n = 25, m = 3, k_{max} = 6, B = 2$. Furthermore, linear interpolation is used as the interpolation method and the already introduced monotone μ-strategy with $\mu_0 = 10^{-14}$ is employed for all mesh iterations $k \geq 1$. The linear solver is *MA57* which is part of the *HSL*[3] library. Since no analytic solution is available, the quality of the solution is evaluated by the maximum error between the optimal state $x_2(t)$ (pyrolytic bitumen) provided by GDOPT and the values of $x_2(t)$ that are obtained by resimulating the model with advanced simulation software, where the optimal control trajectory is set as an input. If this error is sufficiently small, it is clear that the solution provided by the framework is very close to the true optimal solution. The simulation is performed with the open-source modeling and simulation environment *OpenModelica* using a tolerance of 10^{-14}, 2500 intervals and the BDF method *DASSL*. The initial guess on the control variable is the average of the lower and upper bound, i.e. $T(t) \equiv \frac{748.15+698.15}{2}$.

GDOPT spent 0.0019 seconds for the derivative calculations and code generation, 0.9852 seconds for the compilation and 0.0293 seconds for obtaining the initial guess for the states. Clearly, the times in *SymEngine* and *SciPy* are negligible compared to the compilation. Running the executable produced the following mesh refinement history (Table 2), where k denotes the mesh iteration, $|\mathcal{M}_k|$ the number of intervals, ϕ^* the optimal objective, t_{Ipopt} the time in Ipopt excluding the function evaluations, t_{eval} the time for the function evaluations, $\left\| x_2^{(opt)}(t) - x_2^{(sim)}(t) \right\|_\infty$ the error between the simulation and the values provided by GDOPT, and $\left| \phi^{(opt)} - \phi^{(sim)} \right|$ the error in the objective. Moreover, t_{eval} is the time Ipopt spent evaluating

the callback Jacobian, Hessian, etc., and t_{Ipopt} is the time in Ipopt without t_{eval}.

k	$\lvert\mathcal{M}_k\rvert$	$\phi^{(opt)}$	t_{Ipopt} in s	t_{eval} in s	$\left\lVert x_2^{(opt)}(t) - x_2^{(sim)}(t)\right\rVert_\infty$	$\lvert\phi^{(opt)} - \phi^{(sim)}\rvert$
0	25	$-3.5365032296 \cdot 10^{-1}$	0.05688	0.00399	$5.71439 \cdot 10^{-4}$	$1.35618 \cdot 10^{-6}$
1	50	$-3.5365028487 \cdot 10^{-1}$	0.00989	0.00100	$2.31212 \cdot 10^{-4}$	$3.18765 \cdot 10^{-7}$
2	100	$-3.5365157438 \cdot 10^{-1}$	0.02196	0.00260	$6.95880 \cdot 10^{-5}$	$1.18472 \cdot 10^{-7}$
3	132	$-3.5365157567 \cdot 10^{-1}$	0.02226	0.00258	$2.61763 \cdot 10^{-5}$	$1.30685 \cdot 10^{-9}$
4	174	$-3.5365157881 \cdot 10^{-1}$	0.02514	0.00301	$8.10317 \cdot 10^{-6}$	$7.40748 \cdot 10^{-10}$
5	230	$-3.5365157964 \cdot 10^{-1}$	0.03188	0.00413	$2.14007 \cdot 10^{-7}$	$7.77591 \cdot 10^{-11}$
6	307	$-3.5365157953 \cdot 10^{-1}$	0.03882	0.00492	$3.07730 \cdot 10^{-7}$	$6.95770 \cdot 10^{-11}$
Final / Σ	307	$-3.5365157953 \cdot 10^{-1}$	0.20683	0.02223	$3.07730 \cdot 10^{-7}$	$6.95770 \cdot 10^{-11}$

Table 2: L2BN mesh refinement history for Model 2.2

In total 0.24788 seconds were spent in the backend, 0.20683 seconds in Ipopt and MA57, 0.02223 seconds for the function evaluations, 0.00882 seconds in GDOPT algorithms, e.g. L2BN, initializations, etc., and 0.01000 seconds for I/O operations. This shows that the framework is extremely fast and the proposed mesh refinement algorithm and the derivative evaluations are implemented very efficiently. Moreover, Table 2 illustrates that subsequent mesh iterations take very little time to solve. Note that the last solved NLP contains 307 intervals, but takes only 0.04374 seconds to solve, which is less than the initial NLP with only 25 intervals. In addition, the proposed mesh refinement algorithm L2BN iteratively reduces the error to the simulation except for the last iteration. The optimal temperature control $T(t)$ for $k = 6$ and a visualization of the refinement are depicted in Figure 15.

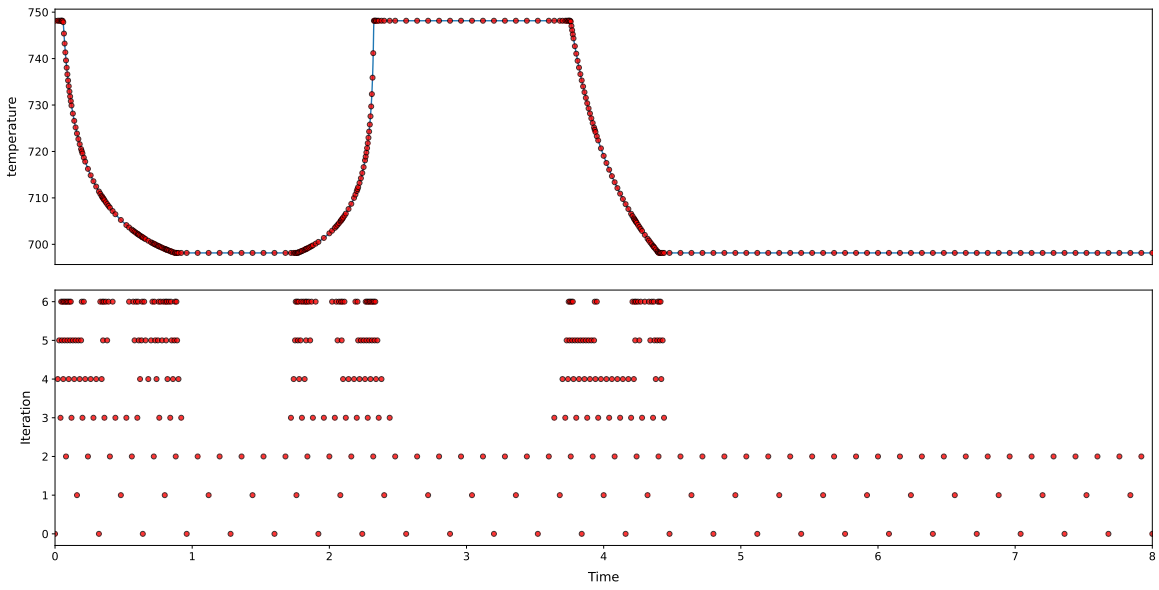


Figure 15: Optimal temperature control and mesh refinement for Model 2.2 ($t_{i,j}$ drawn)

This plot illustrates the placement of the inserted base points t_i for a given mesh iteration. As constructed, it can be seen that L2BN correctly detects corners and steep sections in the control trajectory, bisects the corresponding intervals and thus, decreases the error. Additionally, the error for $k = 6$ is shown in Figure 16.

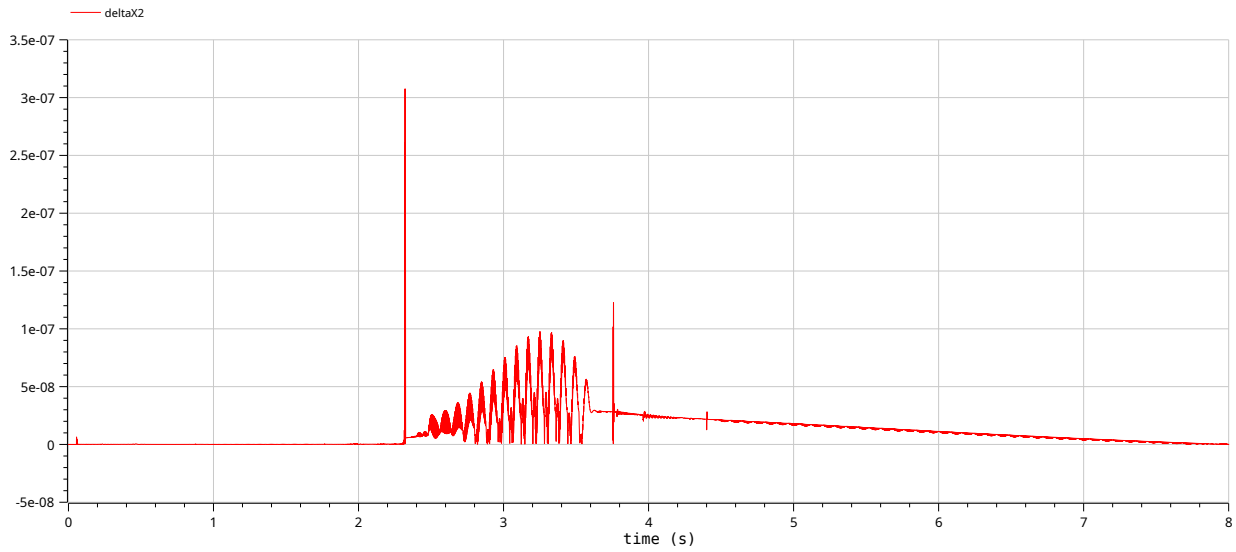


Figure 16: Error between the simulated and provided optimal state $\left|x_2^{(opt)}(t) - x_2^{(sim)}(t)\right|$ for Model 2.2

Now, all aforementioned parameters of the model remain the same, but the problem is solved on an equidistant mesh for a varying number of intervals and without any mesh refinement. Since the initial guess is so poor that the algorithm converged very slowly, a new initial guess $u(t) \equiv 700$ is used for $n = 400, 800, 1200$ intervals.

$n = \lvert\mathcal{M}_0\rvert$	$\phi^{(opt)}$	t_{Ipopt} in s	t_{eval} in s	$\left\|x_2^{(opt)}(t) - x_2^{(sim)}(t)\right\|_\infty$	$\left\|\phi^{(opt)} - \phi^{(sim)}\right\|$
50	$-3.5365028487 \cdot 10^{-1}$	0.12248	0.01003	$2.31212 \cdot 10^{-4}$	$3.18765 \cdot 10^{-7}$
100	$-3.5365157438 \cdot 10^{-1}$	0.48809	0.04142	$6.95880 \cdot 10^{-5}$	$1.18472 \cdot 10^{-7}$
200	$-3.5365157570 \cdot 10^{-1}$	1.50942	0.12893	$2.61960 \cdot 10^{-5}$	$1.35812 \cdot 10^{-9}$
400	$-3.5365157884 \cdot 10^{-1}$	0.39726	0.04560	$8.10256 \cdot 10^{-6}$	$8.18108 \cdot 10^{-10}$
800	$-3.5365157967 \cdot 10^{-1}$	3.11860	0.27875	$2.08376 \cdot 10^{-7}$	$2.59384 \cdot 10^{-11}$
1200	$-3.5365157960 \cdot 10^{-1}$	2.11181	0.22395	$7.77407 \cdot 10^{-7}$	$3.70373 \cdot 10^{-11}$

Table 3: Performance of the default collocation method without mesh refinements

Comparing the solution of Table 3 and Table 2 shows that the proposed mesh refinement algorithm is superior to conventional equidistant approaches in every aspect. L2BN requires less than a tenth of the execution time to obtain a solution that has a comparably low error. This is possible because the interpolation of the optimal solution results in almost optimal initial guesses. By also setting μ_0 to be very small, the interior-point optimizer has to perform very few steps. Moreover, the adaptive mesh refinement algorithm can adequately handle the extremely poor initial guess, because the first NLP is very small. L2BN is also able to correctly identify the relevant sections of the control trajectory that have to be refined, as seen in Figure 15, and by the fact that the final mesh is almost 3 times smaller, than the equidistant mesh with a comparably low error.

8.2 Hypersensitive Optimal Control Problem

Now, the hypersensitive optimal control problem (Model A.1) is considered. As shown in Appendix A.1, the problem has a smooth analytic optimal solution (163), (164) and is an excellent problem to test adaptive

mesh refinement algorithms that has been studied in the literature, e.g. in [46]. Since the optimal solution is almost 0 everywhere except near the start and end points, adaptive mesh refinement algorithms work very well because very few mesh points are actually needed. In the following, the problem is solved with the proposed framework for a final time of $t_f = 10000$. The L2BN parameters are $n = 25, m = 7, k_{max} = 20$ and $B = 0$. Note that rather high degree polynomials with degree $m = 7$ are used, because the problem has a smooth solution and therefore a rapid decrease in error can be excepted. As before, the monotone μ-strategy with $\mu_0 = 10^{-14}$ is employed. The initial guess is $u(t) \equiv 0$ and the initial states are obtained by performing a simulation of the dynamic with *SciPy*. The linear solver that is used is the default option *MUMPS*.

In Table 4 the mesh refinement history of the model in Appendix I is shown. The column $\#it$ denotes the number of iterations Ipopt needed to solve, inf_pr and inf_du are the primal and dual infeasibilities of the initial guess provided to Ipopt and err_x and err_u are the error between the numerical optimal solution and the true optimal solution, i.e. $err_x = \left\|x^*(t) - x^{(opt)}(t)\right\|_\infty$ and $err_u = \left\|u^*(t) - u^{(opt)}(t)\right\|_\infty$. It is important that the problem is a quadratic optimization problem (QP) and thus very easy to solve, compared to general NLPs. Moreover, in each iteration Ipopt performs a single step to achieve optimality, except for the last two iterations, where the primal and dual infeasibilities of the initial guess are so small, that they are below the optimality tolerance of 10^{-14} and are therefore returned immediately. Overall, the problem is solved extremely fast in about 0.042 seconds and the error of the state and control decreases rapidly. This shows that the proposed method efficiently captures the hypersensitive nature of the problem. In Figure 17, the optimal solution and the corresponding mesh refinement are visualized for the interval $[9980, 10000]$. Once again, the dots in the graph identify the base points t_i of every interval. It is clear that GDOPT correctly detects the huge slope at the end of the time horizon and reduces the error in this way. Additionally, solving the problem with an equidistant mesh up to the precision obtained by L2BN is virtually impossible. Even with an initial mesh containing $n = 10000$ intervals, the error to the exact solution is $err_x = 1.5727 \cdot 10^{-8}$, while solving the problem in a single iteration also took 0.26305 seconds.

k	$\lvert\mathcal{M}_k\rvert$	$\#it$	inf_pr	inf_du	t_{Ipopt} in s	t_{eval} in s	err_x	err_u
0	25	1	$3.21 \cdot 10^{1}$	$1.81 \cdot 10^{-4}$	0.00759	0.00004	$4.0957 \cdot 10^{-2}$	$1.3521 \cdot 10^{0}$
1	31	1	$1.12 \cdot 10^{2}$	$9.07 \cdot 10^{0}$	0.00153	0.00004	$8.2728 \cdot 10^{-2}$	$1.2907 \cdot 10^{0}$
2	38	1	$5.26 \cdot 10^{1}$	$4.29 \cdot 10^{0}$	0.00109	0.00004	$1.4478 \cdot 10^{-1}$	$1.1707 \cdot 10^{0}$
3	46	1	$2.29 \cdot 10^{1}$	$1.87 \cdot 10^{0}$	0.00196	0.00007	$1.4362 \cdot 10^{-1}$	$9.4482 \cdot 10^{-1}$
4	54	1	$8.45 \cdot 10^{0}$	$6.65 \cdot 10^{-1}$	0.00161	0.00006	$6.4403 \cdot 10^{-2}$	$5.7797 \cdot 10^{-1}$
5	61	1	$2.23 \cdot 10^{0}$	$2.04 \cdot 10^{-1}$	0.00187	0.00007	$1.4952 \cdot 10^{-2}$	$1.9296 \cdot 10^{-1}$
6	67	1	$3.21 \cdot 10^{-1}$	$5.18 \cdot 10^{-2}$	0.00205	0.00007	$1.2344 \cdot 10^{-3}$	$2.4417 \cdot 10^{-2}$
7	72	1	$1.91 \cdot 10^{-2}$	$4.34 \cdot 10^{-3}$	0.00326	0.00018	$3.3416 \cdot 10^{-5}$	$1.1210 \cdot 10^{-3}$
8	78	1	$4.35 \cdot 10^{-4}$	$1.09 \cdot 10^{-4}$	0.00385	0.00018	$3.7683 \cdot 10^{-7}$	$2.3996 \cdot 10^{-5}$
9	85	1	$4.65 \cdot 10^{-6}$	$1.15 \cdot 10^{-6}$	0.00285	0.00010	$2.5480 \cdot 10^{-9}$	$3.2073 \cdot 10^{-7}$
10	93	1	$3.11 \cdot 10^{-8}$	$7.41 \cdot 10^{-9}$	0.00291	0.00012	$4.9793 \cdot 10^{-10}$	$6.8914 \cdot 10^{-9}$
11	102	1	$1.60 \cdot 10^{-10}$	$3.69 \cdot 10^{-11}$	0.00285	0.00015	$4.9793 \cdot 10^{-10}$	$6.8914 \cdot 10^{-9}$
12	111	1	$8.41 \cdot 10^{-13}$	$1.77 \cdot 10^{-13}$	0.00313	0.00012	$4.9793 \cdot 10^{-10}$	$6.8914 \cdot 10^{-9}$
13	119	0	$7.47 \cdot 10^{-15}$	$2.57 \cdot 10^{-15}$	0.00224	0.00006	$4.9793 \cdot 10^{-10}$	$6.8914 \cdot 10^{-9}$
14	125	0	$7.47 \cdot 10^{-15}$	$2.20 \cdot 10^{-15}$	0.00193	0.00006	$4.9793 \cdot 10^{-10}$	$6.8914 \cdot 10^{-9}$
Final / Σ	125	13	-	-	0.04072	0.00133	$4.9793 \cdot 10^{-10}$	$6.8914 \cdot 10^{-9}$

Table 4: L2BN mesh refinement history for Model A.1

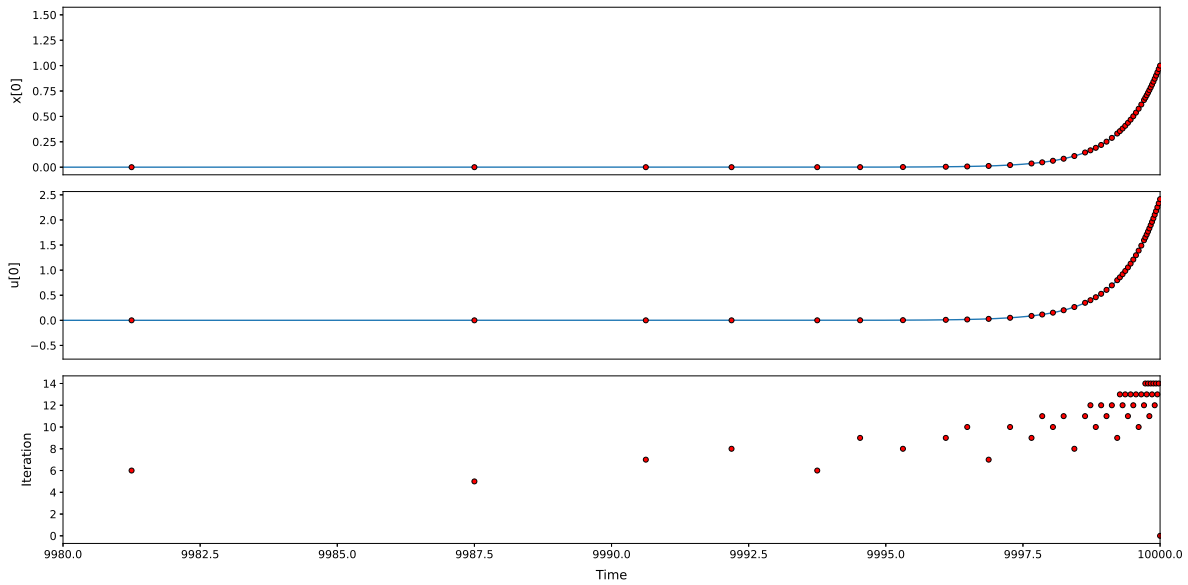


Figure 17: Optimal control and mesh refinement for Model A.1 (t_i drawn)

An important property of the problem is that it has a smooth optimal control. As investigated in Chapter 6.3.2.2, L2BN should terminate after a finite number of iterations. Since the algorithm terminated after 14 iterations and $k_{max} = 20$, this is clearly fulfilled and the control trajectory is uniformized as desired. Furthermore, Table 5 shows the *L2 / on-interval* history of the last interval. This interval has the largest L^2 value of all intervals for each iteration and is denoted by L2_max = $\max_i \|\dot{p}(t)\|_{L^2}$. The second row of the table contains L2_frac, which is the value of L2_max in the current iteration divided by the L2_max of the previous iteration. At first, the values of L2_max grow because the resolution is too low in the first few iterations. After iteration 5, however, the error decreases continuously until it falls below the threshold $TOL_1 = 0.121321$. Moreover, the quotient L2_frac that determines the speed of the linear convergence approaches 0.5, as heuristically obtained in Chapter 6.3.2.2.

k	0	1	2	…	5	6	7	8	9	10	11	12	13	14
L2_max	1.03	4.43	4.78	…	6.13	4.97	3.58	2.52	1.69	1.04	0.585	0.312	0.161	0.081
L2_frac	-	4.30	1.07	…	1.14	0.811	0.720	0.704	0.671	0.615	0.563	0.533	0.516	0.503

Table 5: History of the on-interval condition for Model A.1

8.3 Diesel Motor

The next example is taken from the OpenModelica testsuite[63] and has been extensively studied in [18] and solved in [20]. This real-world physical optimal control problem deals with the fuel optimal start-up of a diesel-electric powertrain from an idling condition to a given power level. Since the optimal control problem contains very complicated differential equations, the model is not presented in this thesis. Moreover, the formulation of the problem in GDOPT can be found in the GitHub repository.[60] In order to evaluate the performance of GDOPT compared to other solvers for dynamic optimization problems, the optimal control problem is solved with GDOPT as well as OpenModelica v1.23.0 (OM). The OpenModelica

implementation[17] also uses Ipopt as the NLP solver and utilizes Radau IIA schemes, but only supports $m = 1$ and $m = 3$ collocation nodes. Furthermore, it does not incorporate adaptive mesh refinement algorithms and has no symbolic Hessian support in the present implementation. The optimal solution is calculated on the same hardware and in the case of OpenModelica using *OMEdit* as the interface.
In both cases, the problem is solved with an Ipopt tolerance of 10^{-14}, using the free linear solver MUMPS and for the same initial guesses. The other settings are set to default, but GDOPT again performs the monotone μ-strategy with $\mu_0 = 10^{-14}$, if mesh refinement is performed. The problem has been solved for various numbers of configurations and the performance measures are given in Table 6. GDOPT took 0.0208 seconds for the derivative calculations and code generation, 1.1986 seconds for compiling the model and 0.0337 seconds for obtaining the initial guess by simulating the dynamics. Note that these timings are independent of the specific configuration.

Algorithm	k_{max}	n	m	$\lvert\mathcal{M}_{final}\rvert$	$\phi^{(opt)}$	t_{Ipopt} in s	t_{eval} in s	Termination
OM	-	25	3	25	$1.11171326863 \cdot 10^{-3}$	0.191	0.096	optimal
GDOPT	0	25	3	25	$1.11171326863 \cdot 10^{-3}$	0.069	0.011	optimal
OM	-	100	3	100	$1.11155875712 \cdot 10^{-3}$	0.337	0.235	optimal
GDOPT	0	100	3	100	$1.11155875712 \cdot 10^{-3}$	0.146	0.036	optimal
OM	-	250	3	250	$1.11155972241 \cdot 10^{-3}$	0.873	0.686	optimal
GDOPT	0	250	3	250	$1.11155972241 \cdot 10^{-3}$	0.336	0.079	optimal
OM	-	1000	3	1000	$1.11155920322 \cdot 10^{-3}$	4.157	3.195	acceptable
GDOPT	0	1000	3	1000	$1.11155856386 \cdot 10^{-3}$	1.757	0.391	optimal
GDOPT	5	25	3	114	$1.11155856029 \cdot 10^{-3}$	0.189	0.041	optimal
GDOPT	5	25	5	116	$1.11155852132 \cdot 10^{-3}$	0.558	0.097	optimal
GDOPT	5	25	7	117	$1.11155856606 \cdot 10^{-3}$	0.859	0.110	optimal
GDOPT	5	100	3	292	$1.11155853676 \cdot 10^{-3}$	0.559	0.143	optimal
GDOPT	5	100	5	293	$1.11155853568 \cdot 10^{-3}$	1.326	0.242	optimal
GDOPT	5	100	7	299	$1.11155853848 \cdot 10^{-3}$	3.382	0.384	optimal

Table 6: Performances of GDOPT and OpenModelica for the Model *Diesel Motor*

For $n = 25, 100, 250, 1000$ the same configuration without mesh refinement is used for GDOPT and OM. It can be seen that the optimal objectives coincide for $n = 25, 100, 250$, but not for $n = 1000$. This is caused by the fact that OpenModelica terminates with an *acceptable* optimal solution, since Ipopt could not reach the set tolerance of 10^{-14}. Additionally, in the other three cases, the time taken by Ipopt t_{Ipopt} is nearly 3 times less and the function evaluations are roughly 7 times faster when performed by GDOPT. The origin of the apparent time difference in t_{Ipopt} is unclear and needs further investigation, although it may be caused by a mismatch in the way compilation is performed, or by GDOPT using Ipopt flags more effectively by default. However, this is a noteworthy outcome. Even if the Ipopt time is not taken into account, the time taken by the callback functions is extremely small in the case of the proposed framework, showing that the derivative information is deployed in an efficient manner. It should be noted that the function evaluations in this case are very computationally expensive as can be seen by the fact that GDOPT generated 289 and 279 CSE for the continuous Hessians of two dynamic equations. However, GDOPT took only 0.0208 seconds for the derivative calculations and code generation, as presented before.
Although the comparison of the performance for equidistant meshes is of interest from a theoretical standpoint, GDOPT is able to perform adaptive mesh refinement with Algorithm 6.6. For this reason, the problem

is also solved for a varying number of initial mesh sizes n, collocation nodes m and a given number of maximum mesh iterations $k_{max} = 5$. The performances and the final mesh sizes $\left|\mathcal{M}_{final}\right|$ are also available in Table 6. The optimal control of GDOPT with adaptive mesh refinement and OpenModelica for sufficiently many intervals n is virtually indistinguishable as can be seen in Appendix M. Moreover, employing L2BN allows GDOPT to capture the non-smooth behavior of the optimal solution and increase the resolution at the discontinuities and steep sections. Note that high resolutions are not needed everywhere, since the optimal control is constant in many sections. The optimal objective of this problem is approximately $\phi^* \approx 1.1115585 \cdot 10^{-3}$, which is obtained by GDOPT with 1000 intervals in over 2 seconds. However, using L2BN the same objective can be found on a final mesh with 114 intervals and in about a tenth of the time.

8.4 Reusable Launch Vehicle

Now a classical, highly nonlinear benchmark problem, the crossrange maximization for a space shuttle reentry trajectory, is considered. The problem has been studied in [58] and [30] as a free endpoint optimal control problem. In this thesis, it is refrained from presenting the model in detail. A in-depth description of the model can be found in [58]. The GDOPT formulation in SI units is given in Appendix J. The control variables of the model are the angle of attack $\alpha(t)$ and the bank angle $\sigma(t)$ of the space shuttle. Objective is to maximize the crossrange or equivalently the final latitude, while the vehicle satisfies final constraints on the flight path angle $\psi(t)$, altitude $h(t)$ and velocity $v(t)$. Initially the altitude, velocity and flight path angle are given by $h(0) = 79248\,\text{m}, v(0) = 7803\,\text{m/s}, \psi(0) = \frac{-\pi}{180}$. The final constraints on these states are $h(t_f) = 24384\,\text{m}, v(t_f) = 762\,\text{m/s}, \psi(t_f) = \frac{-5\pi}{180}$.

Prior to optimization, several observations are made about the model. First, the final time t_f of this model is originally chosen to be free, but GDOPT does not support free endpoint problems. Therefore, the problem is converted to a fixed endpoint problem by choosing the final time as the optimal endpoint provided in the literature, i.e. $t_f = 2009.35\,\text{s}$. Additionally, this problem contains astronomical constants, such as the gravitational parameter μ or the radius of the earth R_e, and variables with large orders of magnitude. Hence, a proper scaling of the problem is essential, because the framework diverges otherwise. Note that GDOPT has native support for nominal values and the problem does not need to be adjusted by hand.

L2BN is run on a coarse initial mesh with $n = 5, m = 4, k_{max} = 10, B = 0$, using linear interpolation, a monotone μ-strategy with $\mu_0 = 10^{-14}$ for all refinement iterations, the linear solver MUMPS, and the initial control guess as the average of the lower and upper bounds. GDOPT took 0.005 seconds for the derivative calculations and code generation, 1.1353 seconds for compiling generated C++ code and 0.3077 seconds for solving the IVP with *SciPy*. Clearly, simulating the dynamics is very slow compared to the previous examples. The mesh refinement history of Model J is shown in Table 7.

It can be observed that the algorithm terminates before the maximum number of mesh iterations is reached, which is due to the fact that the problem has a smooth optimal control trajectory. This fact can be observed in Appendix N, where the mesh refinement with all base points t_i is depicted. Table 7 also demonstrates that the refinement iterations were performed very fast compared to the initial NLP, which had a poor and unrealistic initial guess. The other timings of the framework are 0.00443 seconds for the GDOPT algorithms and 0.00682 seconds for I/O operations, giving a total execution time of 0.50939 seconds.

As seen in the table, GDOPT converged to the optimal solution $\phi^{(opt)} = -5.9627639619 \cdot 10^{-1}$, which agrees to all in [30] provided digits. Moreover, the optimal solution of GDOPT in Figure 18 is practically

k	$\lvert\mathcal{M}_k\rvert$	$\phi^{(opt)}$	#it	t_{Ipopt} in s	t_{eval} in s
0	5	$-5.9631893105 \cdot 10^{-1}$	406	0.33636	0.01389
1	10	$-5.9628169368 \cdot 10^{-1}$	14	0.01126	0.00066
2	20	$-5.9627636467 \cdot 10^{-1}$	10	0.01303	0.00115
3	40	$-5.9627639362 \cdot 10^{-1}$	9	0.01907	0.00145
4	46	$-5.9627639617 \cdot 10^{-1}$	8	0.01958	0.00151
5	53	$-5.9627639619 \cdot 10^{-1}$	8	0.02289	0.00181
6	60	$-5.9627639619 \cdot 10^{-1}$	8	0.02475	0.00195
7	63	$-5.9627639619 \cdot 10^{-1}$	8	0.02632	0.00244
Final / Σ	63	$-5.9627639619 \cdot 10^{-1}$	471	0.47327	0.02488

Table 7: L2BN mesh refinement history for Model J

indistinguishable from the solutions in [58] and [30]. However, a similar objective does not necessarily imply that the solution is realistic. Since this optimal control problem has final constraints, it is very easy to check whether the solution is plausible by running a sophisticated simulation of the optimal control with OpenModelica.

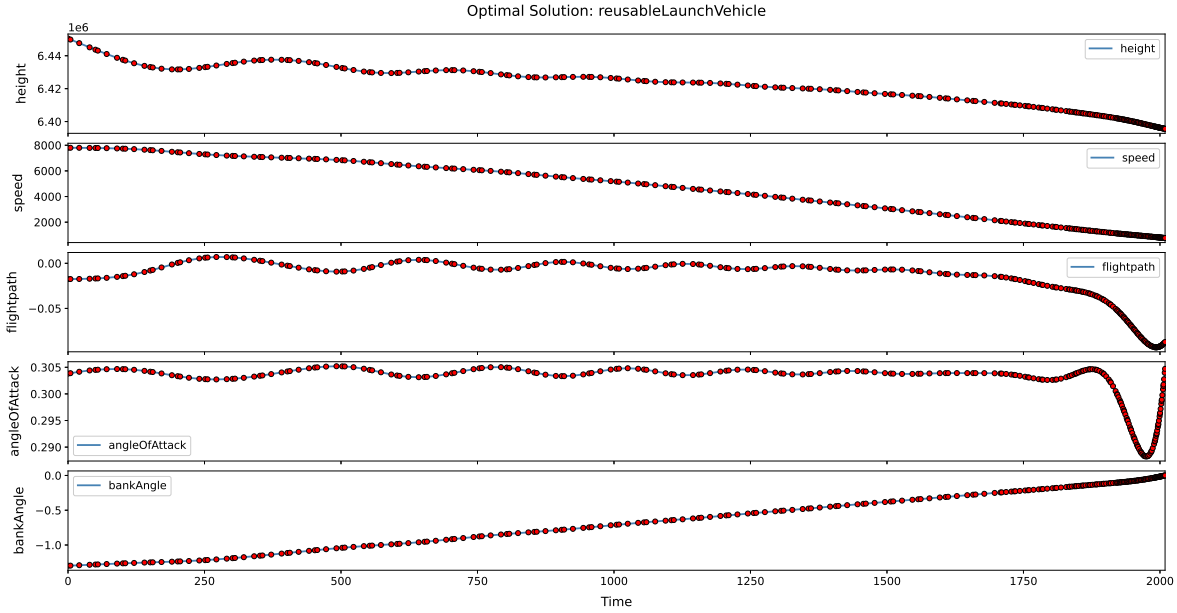
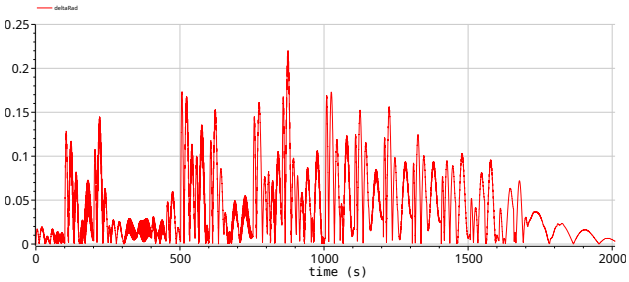
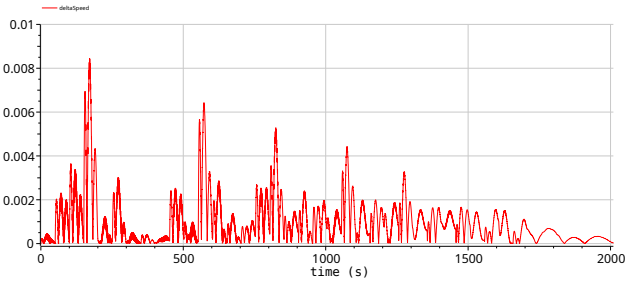


Figure 18: Optimal height $r(t) = R_e + h(t)$, velocity $v(t)$, flight path angle $\psi(t)$, angle of attack $\alpha(t)$ and bank angle $\sigma(t)$ for Model J ($t_{i,j}$ drawn)

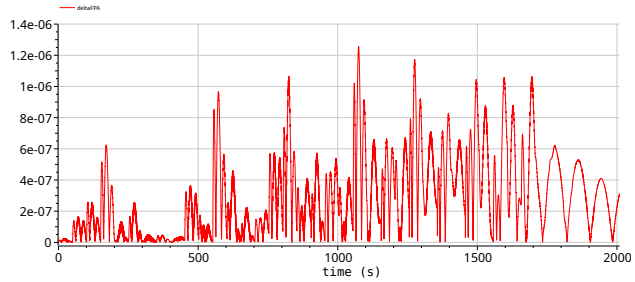
First, the simulation was performed for the initial optimal solution $k = 0$. The error between the altitude of the solution and the desired altitude of 24384 m at the final time t_f is 498.1 m, which is unreasonably inaccurate. Next, the optimal solution provided by GDOPT is investigated. The error for the altitude, velocity and flight path angle over time can be seen in Figure 19. The maximum error in altitude over the entire time horizon is $\left\|h^{(opt)}(t) - h^{(sim)}(t)\right\|_\infty = 0.2196\,\text{m}$ and the error at the final time is given by $\left|24384\,\text{m} - h^{(sim)}(t_f)\right| = 3.295 \cdot 10^{-3}\,\text{m}$. In addition, the errors of the velocity are $\left\|v^{(opt)}(t) - v^{(sim)}(t)\right\|_\infty = 8.453 \cdot 10^{-3}\,\text{m/s}$ and $\left|762\,\text{m/s} - v^{(sim)}(t_f)\right| = 3.256 \cdot 10^{-5}\,\text{m/s}$, and the error terms for the flight path angle are given by $\left\|\psi^{(opt)}(t) - \psi^{(sim)}(t)\right\|_\infty = 1.257 \cdot 10^{-6}$ and $\left|\frac{-5\pi}{180} - \psi^{(sim)}(t_f)\right| = 3.712 \cdot 10^{-7}$.



(a) Absolute error on the altitude $h(t)$ in m



(b) Absolute error on the velocity $v(t)$ in m/s



(c) Absolute error on the flight path angle $\psi(t)$ in rad

Figure 19: Error between simulated and provided optimal states for Model J

Clearly, all these values are sufficiently small and the optimal solution is highly significant. Note that it is possible to improve further, e.g. using the same parameters but an initial mesh of $n = 15$ intervals and $m = 7$ collocation nodes, the framework produced a maximum altitude error of merely 2 cm. Overall, this example shows the efficiency of the proposed framework by converging and terminating for a poor initial guess, incorporating nominal values, as well as in terms of computational time and accuracy for a prominent benchmark problem.

9 Final Remarks

9.1 Summary

In this thesis, a comprehensive overview and implementation of adaptive mesh refinement for direct collocation-based dynamic optimization has been carried out. At first, the principle components of model-based dynamic optimization, i.e. model, constraints and objective, were introduced and led to the General Dynamic Optimization Problem (GDOP). In order to transcribe the continuous infinite dimensional dynamic optimization problem into a discrete NLP, several concepts from the fields of numerical mathematics and nonlinear optimization were needed. In this context, Lagrange interpolation, quadrature rules and Runge-Kutta methods were presented. Based on these foundations and fundamental results about methods of collocation type, the important class of Radau IIA Runge-Kutta collocation methods was constructed. These formulas exhibit exceptional accuracy and stability, and are used in existing dynamic optimization frameworks. Additionally, necessary concepts of nonlinear optimization as well as two important algorithmic classes were introduced, i.e. sequential quadratic programming and interior-point methods. Using the Radau IIA collocation scheme or equivalently the *flipped Legendre-Gauss-Radau (fLGR)* points, the continuous GDOP was transcribed to a large-scale nonlinear optimization problem. Furthermore, it was shown that this dGDOP is equivalent to the transcription in pseudospectral collocation with fixed degree polynomials on each interval.

In the next chapter, important classes of mesh refinement algorithms were motivated and moreover, the novel mesh refinement algorithm *L2-Boundary-Norm (L2BN)* was proposed. This is an h-method that aims to uniformize the control trajectories by iteratively bisecting intervals containing a large slope or curvature, such as discontinuities, steep sections or kinks. It was shown that for smooth problems and under suitable convergence conditions, the termination of the algorithm is guaranteed. In this way, the algorithm can be interpreted as an approximation of a certain density function. Furthermore, the proposed mesh refinement algorithm can be implemented in a very low polynomial time, as it exploits key properties of the Radau quadrature rule and differentiation matrices. The runtime of L2BN was shown to be proportional to the number of control variables in the NLP times the constant number of collocation nodes per interval.

The proposed mesh refinement algorithm is implemented in the novel open-source dynamic optimization framework *GDOPT*[60], which is split into an accessible and expressive frontend modeling environment *gdopt* as well as a powerful and performant C++ backend *libgdopt*. The backend is an extensive implementation of L2BN and utilizes the state-of-the-art interior-point optimizer Ipopt[1] to solve the arising NLPs. *libgdopt* employs a minimal adjacency structure and computes the sparse Hessian and Jacobian by evaluating callback functions of the symbolic derivatives. These are calculated and generated to C++ code by the frontend and use common subexpressions computed by *SymEngine*[4] to reduce evaluation time. In addition, the necessary coefficients of the Radau IIA collocation scheme are calculated to machine precision and hard-coded. GDOPT has many features such as support for nominal values, initial guesses and special functions, plotting features, runtime parameters, and numerous additional flags and options, which are described in the *GDOPT User's Guide*[61].

The performance of the framework has been tested on a number of academic and real-world physical models. It has been shown that GDOPT efficiently captures discontinuities, steep sections, bends, kinks, and is considerably faster compared to traditional equidistant strategies. Moreover, GDOPT can converge

extremely fast even for poor initial guesses, since these are only used on comparably small meshes. The subsequent mesh iterations become very rapid, because the new initial solution is almost optimal and an adequate μ-strategy for the interior-point optimizer can be employed. Additionally, the solutions obtained by the proposed framework were validated using sophisticated simulation software and it was possible to show that the error is significantly smaller compared to traditional collocation-based dynamic optimization. GDOPT was able to reproduce known optimal solutions of real-world physical optimization problems that are highly nonlinear, and shows promising results in terms of execution time and accuracy.

9.2 Limitations and Potential Extensions

Although the framework and the novel mesh refinement algorithm offer decent results when applied to typical dynamic optimization problems, further studies and enhancements are warranted. The framework has not yet been applied to large real world systems as well as been compared to other mesh refinement algorithms. Intensive case studies in these areas would be extremely useful to reveal potential improvements and limitations of GDOPT. In any case, it is reasonable to embed the C++ library *libgdopt* into a sophisticated modeling and simulation software such as *OpenModelica*[9], which would result in many benefits over the present frontend. GDOPT can greatly benefit from the symbolic machinery and expressiveness of existing simulation software and modeling languages, and thus, a wider range of dynamic optimization problems could be formulated, e.g. implicit DAE systems. In addition to extending the GDOP problem class to implicit DAE systems, other extensions can be considered. For example, general initial constraints are currently not supported. These would allow to model boundary value problems where the initial states are not known or moreover, are actually variables to be optimized. Furthermore, support for problems with a free final time t_f is very important to correctly model problems like the *Reusable Launch Vehicle*. These features can be added to GDOPT with relatively limited effort and would greatly improve the expressiveness of the framework. In addition, the integration of simulation software allows solving the IVP, which is used to obtain initial guesses for the state variables, in the backend and using the already generated and compiled model. This reduces overhead and improves performance. Since GDOPT does not directly estimate error terms, an extension of L2BN in this way would be very useful. This could be done by estimating the error using theoretical results similar to [46] or by using efficient and accurate simulation software to perform local or global simulations and therefore, directly calculate the error of the current optimal solution. Although this process may be quite expensive, such a procedure would have great benefits in terms of the quality of the solutions and would allow to verify the validity of the results. Furthermore, as proven in this thesis, the dGDOP is an equivalent transcription to *pseudospectral* direct collocation methods. Therefore, it is reasonable to extend the present *h*-method to a *hp*- or *ph*-adaptive method, that utilizes the spectral convergence for smooth problems by changing the number of collocation nodes per interval individually, while also being able to precisely capture non-smooth behavior. Such an extension to a pseudospectral method is likely to be very time consuming, but would enable GDOPT to compete with state-of-the-art direct collocation frameworks.

References

[1] Wächter, A., Biegler, L. T. (2006). On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming. *Mathematical Programming*, 106(1), 25–57.

[2] Amestoy, P. R., Duff, I. S., L'Excellent, J.-Y., Koster, J. (2001). A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1), 15–41.

[3] HSL. (2013). A collection of Fortran codes for large-scale scientific computation. Available at: http://www.hsl.rl.ac.uk.

[4] SymEngine Developers. SymEngine: A Fast Symbolic Manipulation Library. GitHub repository. Available at: https://github.com/symengine/symengine, accessed September 27, 2024.

[5] Virtanen, P., et al. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3), 261–272.

[6] Meurer, A., Smith, C.P., Paprocki M., et. al. (2017) SymPy: symbolic computing in Python. PeerJ Computer Science. 3:e103. https://doi.org/10.7717/peerj-cs.103

[7] The mpmath development team (2023). mpmath: a Python library for arbitrary-precision floating-point arithmetic. http://mpmath.org

[8] Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3), 90–95.

[9] Fritzson, P., Pop, A., Abdelhak, K., et al. (2020). The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development. *Modeling, Identification and Control*, 41(4), 241–285. ISSN 1890-1328. Available at: https://www.mic-journal.no/PDF/2020/MIC-2020-4-1.pdf.

[10] Ipopt. (n.d.). Ipopt Documentation. Retrieved from https://coin-or.github.io/Ipopt/.

[11] OpenModelica. (n.d.). OpenModelica User's Guide. Retrieved from https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/optimization.html.

[12] Wen, C. S., Yen T. F. (1977). Optimization of oil shale pyrolysis. *Chemical Engineering Science*, 32(3), 346–349. ISSN: 0009-2509.

[13] Biegler, L. T. (2010). Nonlinear programming. Concepts, algorithms, and applications in chemical processes. *SIAM*.

[14] Åkesson, J., Braun, W., Lindholm, P., & Bachmann, B. (2012). Generation of Sparse Jacobians for the Function Mock-Up Interface 2.0. In *Proceedings of the 9th International Modelica Conference* (pp. 185–196). The Modelica Association. https://doi.org/10.3384/ecp12076185.

[15] Mengist, A., Gebremedhin, M., Ruge, V., & Bachmann, B. (2013). Model-Based Dynamic Optimization with OpenModelica and CasADi. Conference paper. Available at: https://www.researchgate.net/publication/271849556.

[16] Magnusson, F., & Åkesson, J. R. (2015). Dynamic Optimization in JModelica.org. *Processes*, June 2015. https://doi.org/10.3390/pr3020471. Available at: https://www.researchgate.net/publication/303933195.

[17] Ruge, V., Braun, W., Bachmann, B., Walther, A. & Kulshreshtha, K. (2014). Efficient Implementation of Collocation Methods for Optimization using OpenModelica and ADOL-C. Conference paper. Available at: https://www.researchgate.net/publication/269235513.

[18] Sivertsson, M., Eriksson, L. (2012). Optimal power response of a diesel-electric powertrain. IFAC Proceedings Volumes, Volume 45, Issue 30, Pages 262-269.

[19] Friesz, T. L. (2010). *Dynamic Optimization and Differential Games*. Springer. https://doi.org/10.1007/978-1-4419-5780-7.

[20] Bachmann, B., Ochel, L., Ruge, V., et. al. (2012). Parallel Multiple-Shooting and Collocation Optimization with OpenModelica. Conference paper. Available at: https://www.researchgate.net/publication/268003988.

[21] Aburajabaltamimi, J. (2011). Development of Efficient Algorithms for Model Predictive Control of Fast Systems. PhD thesis. Technische Universität Ilmenau.

[22] Evans, L. C. An Introduction to Mathematical Optimal Control Theory. University of California Berkeley. Available at: https://math.berkeley.edu/%7Eevans/control.course.pdf, accessed October 21, 2024.

[23] Peng, H., Gao, Q., Wu, Z. (2014). Symplectic algorithms with mesh refinement for a hypersensitive optimal control problem. *International Journal of Computer Mathematics* 92(11):1-17. https://doi.org/10.1080/00207160.2014.979810.

[24] Rao, A. V. (2010). A Survey of Numerical Methods for Optimal Control. *Advances in the Astronautical Sciences* 135(1). Available at: https://www.researchgate.net/publication/268042868.

[25] P. E. Gill, W. Murray, M. A. Saunders, Elizabeth Wong. SNOPT 7.7 User's Manual. CCoM Technical Report 18-1, Center for Computational Mathematics, University of California, San Diego.

[26] P. E. Gill, W. Murray and M. A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. SIAM Review 47 (2005), 99-131.

[27] Becerra, V.M. (2010). Solving complex optimal control problems at no cost with PSOPT. Proc. IEEE Multi-conference on Systems and Control, Yokohama, Japan, September 7-10, 2010, pp. 1391-1396.

[28] R. H. Byrd, J. Nocedal, and R. A. Waltz. KNITRO: An Integrated Package for Nonlinear Optimization, Large Scale Nonlinear Optimization, Springer Verlag, 2006, pp. 35–59.

[29] Quirynen, R., Gros, S., Houska, B. et al. Lifted collocation integrators for direct optimal control in ACADO toolkit. Math. Prog. Comp. 9, 527–571 (2017). https://doi.org/10.1007/s12532-017-0119-0

[30] Patterson, M. A., Rao, A. V. GPOPS-II: A MATLAB Software for Solving Multiple-Phase Optimal Control Problems Using hp-Adaptive Gaussian Quadrature Collocation Methods and Sparse Nonlinear

Programming. ACM Transactions on Mathematical Software. 41(1):1-37 (2014). https://dl.acm.org/doi/10.1145/2558904

[31] Bachmann, B. (2021). Numerische Mathematik. Skript. Fachhochschule Bielefeld.

[32] Schwarz, H. R., Köckler, N. (2009). Numerische Mathematik. Springer. https://link.springer.com/book/10.1007/978-3-8348-8166-3

[33] Baltensperger, R. (2000). Improving the accuracy of the matrix differentiation method for arbitrary collocation points. Applied Numerical Mathematics. 33(1):143-149. http://dx.doi.org/10.1016/S0168-9274(99)00077-X

[34] Schneider, C., Werner, W. (1986). Some New Aspects of Rational Interpolation. Math. Comp. 47, 285-299. https://www.ams.org/journals/mcom/1986-47-175/S0025-5718-1986-0842136-8/

[35] Foupouagnigni, M., Koepf W. (2020). Orthogonal Polynomials. 2nd AIMS-Volkswagen Stiftung Workshop, Douala, Cameroon, 5-12 October, 2018. https://link.springer.com/book/10.1007/978-3-030-36744-2

[36] Marcellán, F., Branquinho, A. & Petronilho, J. Classical orthogonal polynomials: A functional approach. Acta Appl Math 34, 283–303 (1994). https://doi.org/10.1007/BF00998681

[37] Radau, R. Etude sur les formules d'approximation qui servent à calculer la valeur numérique d'une intégrale définie. J. Math. Pures et Appl. , 6 (1880) pp. 283–336

[38] Abramowitz, M., Stegun, I. A., (1968). Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables. United States Government Printing Office. ISBN: 978-0160002021.

[39] Hermann, M. (2020). Numerische Mathematik Band 2: Analytische Probleme, 4. Auflage. De Gruyter.

[40] Hairer, E., Lubich, C., Wanner, G. (2006). Geometric Numerical Integration. Springer. ISBN: 978-3-540-30663-4

[41] Butcher, J. C. (2003), Numerical Methods for Ordinary Differential Equations. Wiley.

[42] Hairer, E., Wanner, G. (2000). Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems. Springer.

[43] Hairer, E., Nørsett, S. P., Wanner, G. (1993). Solving Ordinary Differential Equations I: Nonstiff Problems. Springer.

[44] Hairer, E., Wanner, G. (1999). Stiff differential equations solved by Radau methods. Journal of Computational and Applied Mathematics 111 (1999) 93-111.

[45] Andrei, N. (2022). Modern Numerical Nonlinear Optimization. Springer. https://doi.org/10.1007/978-3-031-08720-2

[46] Liu, F., Hager, W. W., Rao, A. V. (2015). Adaptive mesh refinement method for optimal control using nonsmoothness detection and mesh size reduction. Journal of the Franklin Institute 352 4081–4106. https://doi.org/10.1016/j.jfranklin.2015.05.028

[47] Kameswaran, S., Biegler, L. T. (2008). Convergence rates for direct transcription of optimal control problems using collocation at Radau points. Comput. Optim. Appl. 41 (1) 81–126.

[48] Garg, D., Patterson, M. A. et al. (2011). Direct Trajectory Optimization and Costate Estimation of General Optimal Control Problems Using a Radau Pseudospectral Method. Computational Optimization and Applications 49(2):335-358.

[49] Sagliano, M., Theil, S., D'Onofrio V., Bergsma M. (2018). SPARTAN: A Novel Pseudospectral Algorithm for Entry, Descent, and Landing Analysis. Advances in Aerospace Guidance, Navigation and Control. http://doi.org/10.1007/978-3-319-65283-2_36

[50] Zhao, J., Shang, T. (2018). Dynamic Optimization Using Local Collocation Methods and Improved Multiresolution Technique. Appl. Sci. 2018, 8(9), 1680. https://doi.org/10.3390/app8091680

[51] Jain, S., Tsiotras, P. (2008). Trajectory Optimization Using Multiresolution Techniques. Journal of Guidance Control and Dynamics. Journal of Guidance, Control, and Dynamics 31(5).

[52] Betts, J.T., Huffman, W.P. (1998). Mesh refinement in direct transcription methods for optimal control. Optimal Control Applications and Methods, 19 (1) 1-21.

[53] Zhao, Y., Tsiotras, Panagiotis. (2009). Mesh Refinement Using Density Function for Solving Optimal Control Problems. AIAA Infotech at Aerospace Conference and Exhibit and AIAA. 10.2514/6.2009-2019

[54] Peng, H., Gao, Q., Wu, Z., Zhong, W. (2014). Symplectic algorithms with mesh refinement for a hypersensitive optimal control problem. International Journal of Computer Mathematics. 92. 1-17. 10.1080/00207160.2014.979810

[55] Benson, D., Huntington, G., Thorvaldsen, T., Rao, A. (2006). Direct Trajectory Optimization and Costate Estimation via an Orthogonal Collocation Method. Journal of Guidance Control and Dynamics. 29. 1435-1440. https://doi.org/10.2514/1.20478

[56] Patterson, M. A., Hager, W. W., and Rao, A. V. (2015). A ph mesh refinement method for optimal control. Optim. Control Appl. Meth., 36, 398–421.

[57] Darby, C. L. (2011). hp–Pseudospectral Method for Solving Continuous-Time Nonlinear Optimal Control Problems. PhD thesis. University of Florida.

[58] Betts, J. T. (2010). Practical Methods for Optimal Control and Estimation Using Nonlinear Programming, Second Edition. Society for Industrial and Applied Mathematics. 10.1137/1. 9780898718577

[59] Hager, W. W. et. al. (2019). Convergence rate for a Radau hp collocation method applied to constrained optimal control. Computational Optimization and Applications (2019) 74:275–314. https://doi.org/10.1007/s10589-019-00100-1

[60] Langenkamp, L. (2024). GDOPT. GitHub Repository. https://github.com/linuslangenkamp/GDOPT

[61] Langenkamp, L. (2024). GDOPT - General Dynamic Optimizer v.0.1.3 (A Python Environment for Optimizing Dynamic Models). User's Guide. Retrieved from https://github.com/linuslangenkamp/GDOPT/blob/master/usersguide/usersguide.pdf

[62] Langenkamp, L. (2024). ConstructionRadauIIA. GitHub Repository. https://github.com/linuslangenkamp/ConstructionRadauIIA

[63] Diesel Motor Model from the OpenModelica Testsuite. https://github.com/OpenModelica/OpenModelica/blob/master/testsuite/openmodelica/cruntime/optimization/basic/DM.mo

Appendix A - Maximum Principle

Pontryagin's Maximum Principle (PMP) is one of the most important and influential theorems in optimal control theory. It provides necessary conditions that the optimal solution of a control problem must satisfy. Under certain convexity conditions, PMP even provides sufficient conditions. The theorem reduces the infinite-dimensional optimization problem to a two-point Boundary Value Problem (BVP) as well as to the maximization of a Hamiltonian, which is a much easier problem to solve. Thus, it is possible to solve optimal control problems entirely by analytic methods and to obtain analytic solutions. While the theorem is not a main focus of this thesis, it is a useful tool to evaluate analytic solutions and compare them with the numerical solution of the proposed framework.[22][19]
There are many different formulations of PMP in the literature. Here it is given for the following optimal control problem:

$$\begin{aligned} &\max_{\boldsymbol{u}(t)} M(\boldsymbol{x}(t_f)) + \int_{t_0}^{t_f} L(\boldsymbol{x}(t), \boldsymbol{u}(t))\,\mathrm{d}t \\ &\quad \text{s.t.} \\ &\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t)\ \forall t \in [t_0, t_f],\ \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \\ &\boldsymbol{u}(t) \in \mathcal{U} \end{aligned} \tag{150}$$

This problem is a fixed endpoint maximization optimal control problem with neither path constraints nor final constraints. The set $\mathcal{U}$ denotes the set of all admissible controls. In order to keep the notation as simple as possible, the explicit dependencies on the time t will be omitted for the rest of the chapter.[22]

Theorem A.1 (Pontryagin's Maximum Principle)**.** *Let $\boldsymbol{u}^*(t)$ be the optimal control to problem* (150) *and $\boldsymbol{x}^*(t)$ the corresponding state vector, then exists a function $\boldsymbol{\lambda}^*(t)$, such that the Hamiltonian*

$$\mathcal{H}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{\lambda}) = L(\boldsymbol{x}, \boldsymbol{u}) + \boldsymbol{\lambda}^T \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})$$

satisfies

$$\dot{\boldsymbol{x}}^* = \nabla_{\boldsymbol{\lambda}} \mathcal{H}(\boldsymbol{x}^*, \boldsymbol{u}^*, \boldsymbol{\lambda}^*) \tag{151}$$

$$\dot{\boldsymbol{\lambda}}^* = -\nabla_{\boldsymbol{x}} \mathcal{H}(\boldsymbol{x}^*, \boldsymbol{u}^*, \boldsymbol{\lambda}^*) \tag{152}$$

$$\boldsymbol{\lambda}^*(t_f) = \nabla M(\boldsymbol{x}^*(t_f)) \tag{153}$$

$$H(\boldsymbol{x}^*, \boldsymbol{u}^*, \boldsymbol{\lambda}^*) = \max_{\boldsymbol{u} \in \mathcal{U}} \mathcal{H}(\boldsymbol{x}^*, \boldsymbol{u}, \boldsymbol{\lambda}^*) \tag{154}$$

$$0 = \nabla_{\boldsymbol{u}} \mathcal{H}(\boldsymbol{x}^*, \boldsymbol{u}^*, \boldsymbol{\lambda}^*) \tag{155}$$

Proof. see [22] □

A.1 Hypersensitive Optimal Control Problem

To illustrate PMP an example problem is considered. This is an excellent problem to test adaptive mesh refinement algorithms that are the primary focus in this thesis. In Chapter 8.2 the analytic solution of Model A.1 will be compared with the numeric solution to evaluate the performance of the proposed algorithm.

Model A.1 (Hypersensitive Optimal Control Problem)**.**

$$\begin{aligned}
&\max_{u(t)} \int_0^{t_f} -\frac{1}{2}\left(x^2+u^2\right)\,\mathrm{d}t \\
&\quad s.t. \\
&\qquad \dot{x} = -x+u \;\; \forall t \in [0,t_f] \\
&x(0) = \frac{3}{2},\; x(t_f) = 1
\end{aligned}$$

The unspecified final time t_f is usually chosen to be way larger than 25, i.e. $t_f \gg 25$. The model then becomes a hypersensitive optimal control problem with a three-phase structure consisting of take-off, cruise, and landing phases. Only in the take-off and landing phases do the control and states evolve significantly to satisfy the initial and final constraints, while both control and state are close to 0 for most of the time horizon to maximize the Lagrange integrand $-\frac{1}{2}\left(x^2+u^2\right)$.[23]
Model A.1 is now solved using PMP. The Hamiltonian is given by

$$\mathcal{H}(x,u,\lambda) = -\frac{1}{2}\left(x^2+u^2\right) + \lambda(-x+u). \tag{156}$$

By Theorem A.1

$$\dot{x}^* = \nabla_\lambda \mathcal{H}(x^*,u^*,\lambda^*) = -x^* + u^* \tag{157}$$

$$\dot{\lambda}^* = -\nabla_x \mathcal{H}(x^*,u^*,\lambda^*) = -(-x^* - \lambda^*) = x^* + \lambda^* \tag{158}$$

$$0 = \nabla_u \mathcal{H}(x^*,u^*,\lambda^*) = -u^* + \lambda^* \implies u^* = \lambda^*. \tag{159}$$

Substituting λ^* by u^* in (158) leads to the linear homogeneous system of differential equations with constant coefficients

$$\begin{pmatrix} \dot{x}^* \\ \dot{u}^* \end{pmatrix} = \begin{pmatrix} -1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x^* \\ u^* \end{pmatrix}. \tag{160}$$

It is easy to check that the general solution for x^* is

$$x^* = C_1 \exp\left(\sqrt{2}t\right) + C_2 \exp\left(-\sqrt{2}t\right),\; C_1, C_2 \in \mathbb{R}. \tag{161}$$

The free coefficients can be calculated using the boundary conditions $x^*(0) = \frac{3}{2}, x^*(t_f) = 1$ and solving the arising linear system of equations

$$\begin{pmatrix} 1 & 1 \\ \exp\left(\sqrt{2}t_f\right) & \exp\left(-\sqrt{2}t_f\right) \end{pmatrix} \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} = \begin{pmatrix} \frac{3}{2} \\ 1 \end{pmatrix}. \tag{162}$$

Inserting the coefficients into the general solution (161) yields the optimal state trajectory

$$x^*(t) = \frac{\frac{3}{2}\exp\left(\sqrt{2}\left(t-2t_f\right)\right) - \exp\left(\sqrt{2}\left(t-t_f\right)\right) + \exp\left(\sqrt{2}\left(-t-t_f\right)\right) - \frac{3}{2}\exp\left(-\sqrt{2}t\right)}{\exp\left(-2\sqrt{2}t_f\right) - 1}. \tag{163}$$

The corresponding optimal control is given by

$$u^*(t) = \frac{\frac{3}{2}(1+\sqrt{2})\exp\left(\sqrt{2}\left(t-2t_f\right)\right) - (1+\sqrt{2})\exp\left(\sqrt{2}\left(t-t_f\right)\right)}{\exp\left(-2\sqrt{2}t_f\right)-1} + \frac{(1-\sqrt{2})\exp\left(\sqrt{2}\left(-t-t_f\right)\right) - \frac{3}{2}(1-\sqrt{2})\exp\left(-\sqrt{2}t\right)}{\exp\left(-2\sqrt{2}t_f\right)-1}. \tag{164}$$

Appendix B - Orthogonal Polynomials

Some basic definitions and theorems about orthogonal polynomials are obtained. Since orthogonal polynomials are not the main focus of this thesis, the theorems are simply stated and not proven. Firstly, it can be observed that for two polynomials $p, q \in P^n$

$$\langle p, q \rangle = \int_{-1}^{1} p(t)q(t)w(t)\,\mathrm{d}t, \text{ with } w(t) > 0 \tag{165}$$

defines an *inner product.*[36] Thus, the concept of orthogonality can be defined for polynomials.

Definition B.1 (Orthogonal Polynomial System)**.** *Any sequence* $(p_n)_{n\in\mathbb{N}_0}$ *with* $\deg p_n = n$ *satisfying*

$$\begin{aligned} \int_{-1}^{1} p_n(t)p_m(t)w(t)\,\mathrm{d}t = 0,\ n \neq m \\ \int_{-1}^{1} p_n(t)p_n(t)w(t)\,\mathrm{d}t \neq 0,\ \forall n \geq 0 \end{aligned} \tag{166}$$

is said to be orthogonal w.r.t. to a weighting function $w(t) > 0$*, and called an orthogonal polynomial system.[35]*

From Definition B.1 it is clear, that any *orthogonal polynomial system* $(p_n)_{n\in\mathbb{N}_0}$ forms a basis for polynomials of degree $\leq n$. Given initial polynomials p_0 and p_1 that satisfy Definition B.1, all orthogonal polynomials with $n > 1$ can be constructed by the *Gram-Schmidt process.* Furthermore, orthogonal polynomials always satisfy a three-term recurrence relation and have simple roots inside the interval $[-1, 1]$.

Lemma B.1. *Let* $(p_n)_{n\in\mathbb{N}_0}$ *be an orthogonal polynomial system, then there exist two complex sequences* $(\beta_n)_{n\in\mathbb{N}_0}$ *and* $(\gamma_n)_{n\in\mathbb{N}_0}$*, with* $\gamma_n \neq 0$ *for every n, such that*

$$\begin{aligned} tp_n(t) &= p_{n+1}(t) + \beta_n p_n(t) + \gamma_n p_{n-1}(t) \text{ for } \forall n \geq 1 \\ p_0(t) &= 1, p_1(t) = t - \beta_0. \end{aligned} \tag{167}$$

Proof. see [36]. □

Lemma B.2. *Let* $(p_n)_{n\in\mathbb{N}_0}$ *be an orthogonal polynomial system, then for every* $n \in \mathbb{N}$ *all zeros of* p_n *are simple and lie inside the interval* $[-1, 1]$.

Proof. see [35]. □

Appendix C - Gauss-Legendre Quadrature

Using n steps, an optimal quadrature rule of the form (31) with order $2n$ will be constructed. This method is called the *Gauss-Legendre quadrature* and based on the orthogonal *Legendre polynomials*.

Definition C.1. *The polynomials $(p_n)_{n\in\mathbb{N}_0}$, which satisfy Definition B.1 with the constant weighting function $w(t) \equiv 1$, i.e.*

$$\begin{aligned} \int_{-1}^{1} p_n(t)p_m(t)\,\mathrm{d}t &= 0,\ n \neq m \\ \int_{-1}^{1} p_n(t)p_n(t)\,\mathrm{d}t &\neq 0,\ \forall n \geq 0 \end{aligned} \tag{168}$$

and $p_0(t) = 1$ are called Legendre polynomials.

These can be constructed interatively with the *Gram-Schmidt process*. The first five *Legendre polynomials* are: $p_0(t) = 1, p_1(t) = t, p_2(t) = \frac{1}{2}(3t^2 - 1), p_3(t) = \frac{1}{2}(5t^3 - 3t)$ and $p_4(t) = \frac{1}{8}(35t^4 - 30t^2 + 3)$. The roots of the Legendre polynomials are also called *Legendre-Gauss (LG)* points. To show that the *Gauss-Legendre quadrature* obtains the maximum possible order, an upper bound on the maximum possible order is established.

Theorem C.1. *The order of a quadrature rule with n nodes can not exceed 2n.*

Proof. Define $q(t) =: \prod_{k=1}^{n}(t - t_k)^2 \in P^{2n}$ with $t_1, \ldots, t_n$ being the nodes of the quadrature rule. Then $I = \int_{-1}^{1} q(t)\,\mathrm{d}t > 0$, but, since $q(t_k) = 0$ for all $k = 1, \ldots, n$, the quadrature rule yields $\tilde{I} = \sum_{j=1}^{n} w_j q(t_j) = 0$. Therefore, no quadrature rule with degree of exactness $2n$ or order $2n + 1$ can exist.[32] □

With the preliminary work done, the very elegant construction of the *Gauss-Legendre quadrature* can be demonstrated.

Theorem C.2 (Gauss-Legendre Quadrature)**.** *It exists a unique quadrature rule with n nodes*

$$\sum_{j=1}^{n} w_j f(t_j) \tag{169}$$

that obtains the maximum order 2n. The nodes t_j are the roots of the n-th Legendre polynomial $p_n(t)$ and the weights w_j are given by

$$w_j = \int_{-1}^{1} \prod_{\substack{k=1 \\ k\neq j}}^{n} \frac{t - t_k}{t_j - t_k}\,\mathrm{d}t \quad j = 1, \ldots, n. \tag{170}$$

Proof. Only the construction of the method is presented. For the proof of uniqueness see [32]. Let $p \in P^{2n-1}$ be arbitrary. Then p can be divided by the n-th Legendre polynomial p_n, which yields

$$p(t) = q(t)p_n(t) + r(t), \tag{171}$$

with $q \in P^{n-1}$ and $r \in P^{n-1}$. Thus

$$\int_{-1}^{1} p(t)\,\mathrm{d}t = \underbrace{\int_{-1}^{1} q(t)p_n(t)\,\mathrm{d}t}_{=0} + \int_{-1}^{1} r(t)\,\mathrm{d}t = \int_{-1}^{1} r(t)\,\mathrm{d}t, \tag{172}$$

since the Legendre polynomials form a basis and every Legendre polynomial p_m with degree $m < n$ is orthogonal to p_n (Definition B.1). Because the zeros of the Legendre polynomials are simple (Lemma B.2) and chosen as nodes, constructing an interpolatory quadrature rule yields

$$\sum_{j=1}^{n} w_j p(t_j) = \sum_{j=1}^{n} w_j q(t_j) \underbrace{p_n(t_j)}_{=0} + \sum_{j=1}^{n} w_j r(t_j) = \sum_{j=1}^{n} w_j r(t_j) = \int_{-1}^{1} r(t)\,\mathrm{d}t = \int_{-1}^{1} p(t)\,\mathrm{d}t, \tag{173}$$

since $r(t) \in P^{n-1}$ can be exactly integrated with any n node interpolatory quadrature rule (Theorem 3.4). Therefore, the interpolatory quadrature rule with the n zeros of the Legendre polynomial p_n chosen as nodes has degree of exactness $2n-1$ and order $2n$.[32] □

Theorem C.3. *The weights of an interpolatory quadrature rule with n nodes are all positive, if the exactness of the quadrature rule is at least* $2n-2$.

Proof. By definition the weights are given by the integral over the Lagrange basis polynomials $l_j \in P^{n-1}$, i.e.

$$w_j = \int_{-1}^{1} l_j(t)\,\mathrm{d}t = \int_{-1}^{1} \prod_{\substack{k=1 \\ k\neq j}}^{n} \frac{t-t_k}{t_j-t_k}\,\mathrm{d}t \; j = 1, \ldots, n. \tag{174}$$

Since $l_j^2 \in P^{2n-2}$ will be exactly integrated with the quadrature rule,

$$0 < \int_{-1}^{1} l_j(t)^2\,\mathrm{d}t = \sum_{\nu=1}^{n} w_\nu l_j(t_\nu)^2 = \sum_{\nu=1}^{n} w_\nu \delta_{\nu j}^2 = w_j \tag{175}$$

holds. (adapted from [32]) □

This property is extremely favorable, since negative weights can cause numerical instabilities and thus convergence issues. For demonstration purposes an example is considered, which shows the structure of the *Gauss-Legendre quadrature* with 2 nodes.

Example C.1. *The Gauss-Legendre quadrature with* $n = 2$ *nodes and order* 4 *is given by*

$$\int_{-1}^{1} f(t)\,\mathrm{d}t \approx f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right). \tag{176}$$

The substitution $\tau = \frac{a+b}{2} + t\frac{b-a}{2}$ *results in the general quadrature rule*

$$\int_{a}^{b} f(\tau)\,\mathrm{d}\tau \approx \frac{b-a}{2}\left[f\left(\frac{a+b}{2} - \frac{b-a}{2\sqrt{3}}\right) + f\left(\frac{a+b}{2} + \frac{b-a}{2\sqrt{3}}\right)\right]. \tag{177}$$

Appendix D - Hessian Calculations for Blocks $B, \tilde{B}, C$ of the dGDOP

Block B: For $(i, j) \neq (n, m)$:

$$\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{p} \partial \boldsymbol{x}_{i,j}} = \sigma_f \Delta t_i b_j \nabla^2_{\boldsymbol{px}} L \Big|_{\boldsymbol{z}_{i,j}} - \Delta t_i \sum_{d=1}^{d_{\boldsymbol{x}}} \lambda_{s(i,j)+d} \nabla^2_{\boldsymbol{px}} f^{(d)} \Big|_{\boldsymbol{z}_{i,j}} + \sum_{d=1}^{d_{\boldsymbol{g}}} \lambda_{s(i,j)+d_{\boldsymbol{x}}+d} \nabla^2_{\boldsymbol{px}} g^{(d)} \Big|_{\boldsymbol{z}_{i,j}} \tag{178}$$

$$\frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{p} \partial \boldsymbol{u}_{i,j}} = \sigma_f \Delta t_i b_j \nabla^2_{\boldsymbol{pu}} L \Big|_{\boldsymbol{z}_{i,j}} - \Delta t_i \sum_{d=1}^{d_{\boldsymbol{x}}} \lambda_{s(i,j)+d} \nabla^2_{\boldsymbol{pu}} f^{(d)} \Big|_{\boldsymbol{z}_{i,j}} + \sum_{d=1}^{d_{\boldsymbol{g}}} \lambda_{s(i,j)+d_{\boldsymbol{x}}+d} \nabla^2_{\boldsymbol{pu}} g^{(d)} \Big|_{\boldsymbol{z}_{i,j}} \tag{179}$$

Block $\tilde{B}$:

$$\begin{aligned} \frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{p} \partial \boldsymbol{x}_{n,m}} = {} & \sigma_f \nabla^2_{\boldsymbol{px}} M \Big|_{\boldsymbol{z}_{n,m}} + \sigma_f \Delta t_n b_m \nabla^2_{\boldsymbol{px}} L \Big|_{\boldsymbol{z}_{n,m}} - \Delta t_n \sum_{d=1}^{d_{\boldsymbol{x}}} \lambda_{s(n,m)+d} \nabla^2_{\boldsymbol{px}} f^{(d)} \Big|_{\boldsymbol{z}_{n,m}} \\ & + \sum_{d=1}^{d_{\boldsymbol{g}}} \lambda_{s(n,m)+d_{\boldsymbol{x}}+d} \nabla^2_{\boldsymbol{px}} g^{(d)} \Big|_{\boldsymbol{z}_{n,m}} + \sum_{d=1}^{d_{\boldsymbol{r}}} \lambda_{s(n+1,0)+d} \nabla^2_{\boldsymbol{px}} r^{(d)} \Big|_{\boldsymbol{z}_{n,m}} \end{aligned} \tag{180}$$

$$\begin{aligned} \frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{p} \partial \boldsymbol{u}_{n,m}} = {} & \sigma_f \nabla^2_{\boldsymbol{pu}} M \Big|_{\boldsymbol{z}_{n,m}} + \sigma_f \Delta t_n b_m \nabla^2_{\boldsymbol{pu}} L \Big|_{\boldsymbol{z}_{n,m}} - \Delta t_n \sum_{d=1}^{d_{\boldsymbol{x}}} \lambda_{s(n,m)+d} \nabla^2_{\boldsymbol{pu}} f^{(d)} \Big|_{\boldsymbol{z}_{n,m}} \\ & + \sum_{d=1}^{d_{\boldsymbol{g}}} \lambda_{s(n,m)+d_{\boldsymbol{x}}+d} \nabla^2_{\boldsymbol{pu}} g^{(d)} \Big|_{\boldsymbol{z}_{n,m}} + \sum_{d=1}^{d_{\boldsymbol{r}}} \lambda_{s(n+1,0)+d} \nabla^2_{\boldsymbol{pu}} r^{(d)} \Big|_{\boldsymbol{z}_{n,m}} \end{aligned} \tag{181}$$

Block C:

$$\begin{aligned} \frac{\partial^2 \mathcal{L}}{\partial \boldsymbol{p}^2} = {} & \sum_{i=0}^{n} \sum_{j=1}^{m} \sigma_f \Delta t_i b_j \nabla^2_{\boldsymbol{pp}} L \Big|_{\boldsymbol{z}_{i,j}} - \Delta t_i \sum_{d=1}^{d_{\boldsymbol{x}}} \lambda_{s(i,j)+d} \nabla^2_{\boldsymbol{pp}} f^{(d)} \Big|_{\boldsymbol{z}_{i,j}} + \sum_{d=1}^{d_{\boldsymbol{g}}} \lambda_{s(i,j)+d_{\boldsymbol{x}}+d} \nabla^2_{\boldsymbol{pp}} g^{(d)} \Big|_{\boldsymbol{z}_{i,j}} \\ & + \sigma_f \nabla^2_{\boldsymbol{pp}} M \Big|_{\boldsymbol{z}_{n,m}} + \sum_{d=1}^{d_{\boldsymbol{r}}} \lambda_{s(n+1,0)+d} \nabla^2_{\boldsymbol{pp}} r^{(d)} \Big|_{\boldsymbol{z}_{n,m}} + \sum_{d=1}^{d_{\boldsymbol{a}}} \lambda_{s(n+1,0)+d_{\boldsymbol{r}}+d} \nabla^2_{\boldsymbol{pp}} a^{(d)} \Big|_{\boldsymbol{p}} \end{aligned} \tag{182}$$

Appendix E - Radau IIA Construction

The C++ implementation of Algorithm 6.6 with Ipopt requires certain constant coefficients of the Radau IIA collocation scheme, i.e. the collocation nodes c_j, quadrature weights b_j, inverse Butcher matrix A^{-1}, differentiation matrix $D^{(1)}$ and the evaluation matrix $E_{ij} = \left(l_j(\tilde{c}_i)\right)_{i,j}$ with $\tilde{c}_i = 0, \frac{c_1}{2}, \ldots, \frac{c_m}{2}, \frac{1+c_1}{2}, \ldots, \frac{1+c_m}{2}$, which allows for a fast computation of the interpolating polynomial on the subintervals. Note that the inverse Butcher matrix must not be calculated, since its a submatrix of $D^{(1)}$ as proven in Chapter 5.3. In the present implementation all these coefficients $c_j, b_j, D^{(1)}_{ij}, E_{ij}$ are hard-coded for $m = 1, \ldots, 70$ collocation nodes and thus enable $O(1)$ access time. This results in a slightly larger binary, but makes the dGDOP function evaluations faster compared to tools where these constants are calculated at runtime. The construction of these (high-order) collocation schemes is performed by a *Python* script that utilizes the packages *SymPy*[6] for symbolic handling and *mpmath*[7] for arbitrary precision floating point arithmetic. Therefore, the coefficients are calculated up to machine precision, which is very valuable for the framework. The script can be found in the *GitHub* repository *ConstructRadauIIA*[62] under the name *ConstructRadauIIA.py*. The principle workflow is to first set the floating point precision to 150 digits and iteratively compute the Jacobi polynomials $P_m^{(1,0)}$ with (43). Then the nodes c_j are calculated as roots of $(1-t)P_m^{(1,0)}(2t-1)$ with *SymPy* routines. However, by studying the three-term recurrence relation (43) the roots can be expressed as the spectrum of a specific symmetric tridiagonal matrix, which can be computed very efficiently using the QR algorithm. Currently, this process is not performed, but can be implemented for higher order methods in the future. Based on the nodes, all other coefficients can be obtained in a straightforward way. To efficiently calculate $D^{(1)}$ (28) is used and the quadrature weights b_j are obtained by using the formula of Theorem 3.5 for the interval $[0, 1]$. To verify the validity of the results, known analytical constants such as the quadrature weights at $c_m = 1$ are compared with the numerical values. After that, the coefficients have been cropped to `long double` precision and hard-coded into the C++ `Integrator` class.

Appendix F - Rayleigh Optimal Control Problem in GDOPT

```
from gdopt import *

model = Model("rayleigh")

x = model.addState(start=-5, symbol="x")
y = model.addState(start=-5, symbol="y")

u = model.addInput()

# x' = y, y' = -x + y * (1.4 - 0.14 * y**2) + 4 * u
model.addDynamic(x, y)
model.addDynamic(y, -x + y * (1.4 - 0.14 * y**2) + 4 * u)

model.addLagrange(x**2 + u**2)

model.generate()

model.optimize(
        tf=4.5,
        steps=1,
        rksteps=70,
        flags={
                "linearSolver": LinearSolver.MA57,
                "initVars": InitVars.SOLVE_EXPLICIT,
        },
)
```

Appendix G - Satellite Optimal Control Problem in GDOPT

```
from gdopt import *

model = Model("satellite")

I1, I2, I3 = 1000000, 833333, 916667

T1S, T2S, T3S = 550, 50, 550

M1, M2, M3, M4 = 0.70106, 0.0923, 0.56098, 0.43047

x1 = model.addState(start=0)
x2 = model.addState(start=0)
x3 = model.addState(start=0)
x4 = model.addState(start=1)
x5 = model.addState(start=0.01)
x6 = model.addState(start=0.005)
x7 = model.addState(start=0.001)

u1 = model.addInput(nominal=0.005)
u2 = model.addInput(nominal=0.00005)
u3 = model.addInput(nominal=0.005)

model.addDynamic(x1, 0.5 * (x5 * x4 - x6 * x3 + x7 * x2))
model.addDynamic(x2, 0.5 * (x5 * x3 + x6 * x4 - x7 * x1))
model.addDynamic(x3, 0.5 * (-x5 * x2 + x6 * x1 + x7 * x4))
model.addDynamic(x4, -0.5 * (x5 * x1 + x6 * x2 + x7 * x3))
model.addDynamic(x5, ((I2 - I3) * x6 * x7 + T1S * u1) / I1)
model.addDynamic(x6, ((I3 - I1) * x7 * x5 + T2S * u2) / I2)
model.addDynamic(x7, ((I1 - I2) * x5 * x6 + T3S * u3) / I3)

model.addMayer((x1 - M1) ** 2 + (x2 - M2) ** 2 + (x3 - M3) ** 2
               + (x4 - M4) ** 2 + x5**2 + x6**2 + x7**2,
               Objective.MINIMIZE,
)

model.addLagrange(0.5 * (u1**2 + u2**2 + u3**2), Objective.MINIMIZE)

model.generate()

model.optimize(steps=25, rksteps=3, tf=100,
       flags={"linearSolver": LinearSolver.MUMPS, "tolerance": 1e-12})
```

Appendix H - Oil Shale Pyrolysis in GDOPT

```
from gdopt import *

model = Model("oilShalePyrolysis")

x1 = model.addState(start=1, symbol="kerogen")
x2 = model.addState(start=0, symbol="pyrolytic bitumen")
x3 = model.addState(start=0, symbol="oil \& gas")
x4 = model.addState(start=0, symbol="organic carbon")

T = model.addInput(lb=698.15, ub=748.15, symbol="temperature", nominal=700,
    guess=(748.15 + 698.15) / 2)

k1 = exp(8.86 - (20300 / 1.9872) / T)
k2 = exp(24.25 - (37400 / 1.9872) / T)
k3 = exp(23.67 - (33800 / 1.9872) / T)
k4 = exp(18.75 - (28200 / 1.9872) / T)
k5 = exp(20.70 - (31000 / 1.9872) / T)

model.addDynamic(x1, -k1 * x1 - (k3 + k4 + k5) * x1 * x2)
model.addDynamic(x2, k1 * x1 - k2 * x2 + k3 * x1 * x2)
model.addDynamic(x3, k2 * x2 + k4 * x1 * x2)
model.addDynamic(x4, k5 * x1 * x2)

model.addMayer(x2, Objective.MAXIMIZE)

model.generate()

model.optimize(
       tf=8,
       steps=25,
       rksteps=3,
       flags={
              "tolerance": 1e-15,
              "linearSolver": LinearSolver.MA57
       },
       meshFlags={
              "algorithm": MeshAlgorithm.L2_BOUNDARY_NORM,
              "iterations": 6,
              "muStrategyRefinement": MuStrategy.MONOTONE,
              "muInitRefinement": 1e-14,
              "fullBisections": 2,
       },
)
```

Appendix I - Hypersensitive Optimal Control Problem in GDOPT

```
from gdopt import *

model = Model("analyticHypersensitive")

x = model.addState(start=1.5)
u = model.addInput()

model.addDynamic(x, -x + u)
model.addFinal(1.0 - x, eq=0)
model.addLagrange(0.5 * (x**2 + u**2))

model.generate()

model.optimize(
        tf=10000,
        steps=25,
        rksteps=7,
        flags={
                "linearSolver": LinearSolver.MUMPS,
                "quadraticObjective": True,
                "linearConstraints": True,
        },
        meshFlags={
                "iterations": 20,
                "muStrategyRefinement": MuStrategy.MONOTONE,
                "muInitRefinement": 1e-16,
                "refinementMethod": RefinementMethod.POLYNOMIAL,
        },
)
```

Appendix J - Reusable Launch Vehicle in GDOPT

```
from gdopt import *

model = Model("reusableLaunchVehicle")

# conversion to SI
cft2m = 0.3048 # feet to meters
cft2km = cft2m / 1000 # feet to kilometers
cslug2kg = 14.5939029 # slugs to kilograms

# constants
Re = 20902900 * cft2m
S = 2690 * cft2m**2
cl1, cl2 = -0.2070, 1.6756
cd1, cd2, cd3 = 0.0785, -0.3529, 2.04
b1, b2, b3 = 0.07854, -0.061592, 0.00621408
H = 23800 * cft2m
al1, al2 = -0.20704, 0.029244
rho0 = 0.002378 * cslug2kg / cft2m**3
mu = 1.4076539e16 * cft2m**3
mass = 6309.433 * cslug2kg

# initial values
alt0 = 260000 * cft2m
rad0 = alt0 + Re
lon0 = 0
lat0 = 0
speed0 = 25600 * cft2m
fpa0 = -1 * pi / 180
azi0 = 90 * pi / 180

# final values
altf = 80000 * cft2m
radf = altf + Re
speedf = 2500 * cft2m
fpaf = -5 * pi / 180

# nominal values
nomRad = (rad0 + Re) / 2
nomSpeed = 45010 / 2

rad = model.addState(start=rad0, lb=Re, ub=rad0, symbol="height", nominal=nomRad)
speed = model.addState(start=speed0, lb=10, ub=45000, symbol="speed",
    nominal=nomSpeed)
lon = model.addState(start=lon0, lb=-pi, ub=pi, symbol="longitude")
```

```
lat = model.addState(start=lat0, lb=-70 * pi / 180, ub=70 * pi / 180,
    symbol="latitude")
fpa = model.addState(start=fpa0, lb=-80 * pi / 180, ub=80 * pi / 180,
    symbol="flightpath")
azi = model.addState(start=azi0, lb=-pi, ub=pi, symbol="azimuth")

aoa = model.addInput(lb=-pi / 2, ub=pi / 2, guess=0, symbol="angleOfAttack")
bank = model.addInput(lb=-pi / 2, ub=1 * pi / 180, guess=-89 / 360 * pi,
    symbol="bankAngle")

altitude = rad - Re

CD = cd1 + cd2 * aoa + cd3 * aoa**2
rho = rho0 * exp(-altitude / H)
CL = cl1 + cl2 * aoa
gravity = mu / rad**2
dynamic_pressure = 0.5 * rho * speed**2
D = dynamic_pressure * S * CD / mass
L = dynamic_pressure * S * CL / mass

model.addDynamic(rad, speed * sin(fpa), nominal=nomRad)
model.addDynamic(speed, -D - gravity * sin(fpa), nominal=nomSpeed)
model.addDynamic(lon, speed * sin(fpa) * sin(azi) / (rad * cos(lat)))
model.addDynamic(lat, speed * cos(fpa) * cos(azi) / rad)
model.addDynamic(fpa, (L * cos(bank) - cos(fpa) * (gravity - speed**2 / rad)) / speed)
model.addDynamic(azi,(L * sin(bank) / cos(fpa) + speed**2 * cos(fpa) * sin(azi) *
    tan(lat) / rad)/ speed)

model.addFinal(rad, eq=radf, nominal=nomRad)
model.addFinal(speed, eq=speedf, nominal=nomSpeed)
model.addFinal(fpa, eq=fpaf)

model.addMayer(-lat)

model.meshIterations = 8
model.muStrategyRefinement = MuStrategy.MONOTONE
model.muInitRefinement = 1e-14

model.generate()

model.optimize(tf=2009.35, steps=5, rksteps=4)
```

Appendix K - Generated First Dynamic Equation of Model 2.2

```
class F0oilShalePyrolysis : public Expression {
       public:
       static std::unique_ptr<F0oilShalePyrolysis> create() {
              Adjacency adj{{0, 1}, {0}, {}};
              AdjacencyDiff adjDiff{{{1, 0}}, {{0, 0}, {0, 1}}, {{0, 0}}, {}, {}, {}};
              return std::unique_ptr<F0oilShalePyrolysis>(new
                  F0oilShalePyrolysis(std::move(adj), std::move(adjDiff)));
       }

       double eval(const double *x, const double *u, const double *p, double t)
           override {
              const double x0 = pow(u[0], -1);
              return -exp(8.86 - 10215.3784219002*x0)*x[0] - x[0]*x[1]*(exp(18.75 -
                  14190.8212560386*x0) + exp(20.7 - 15599.8389694042*x0) + exp(23.67
                  - 17008.8566827697*x0));
       }

       std::array<std::vector<double>, 3> evalDiff(const double *x, const double *u,
           const double *p, double t) override {
              const double x0 = pow(u[0], -1);
              const double x1 = exp(8.86 - 10215.3784219002*x0);
              const double x2 = exp(20.7 - 15599.8389694042*x0);
              const double x3 = exp(18.75 - 14190.8212560386*x0);
              const double x4 = exp(23.67 - 17008.8566827697*x0);
              const double x5 = x2 + x3 + x4;
              const double x6 = pow(u[0], -2);
              return {std::vector<double>{-x1 - x5*x[1], -x5*x[0]},
                  {-10215.3784219002*x1*x6*x[0] - x[0]*x[1]*(15599.8389694042*x2*x6 +
                  14190.8212560386*x3*x6 + 17008.8566827697*x4*x6)}, {}};
       }

       std::array<std::vector<double>, 6> evalDiff2(const double *x, const double *u,
           const double *p, double t) override {
              const double x0 = pow(u[0], -1);
              const double x1 = exp(23.67 - 17008.8566827697*x0);
              const double x2 = pow(u[0], -2);
              const double x3 = exp(18.75 - 14190.8212560386*x0);
              const double x4 = exp(20.7 - 15599.8389694042*x0);
              const double x5 = 17008.8566827697*x2*x1 + 14190.8212560386*x2*x3 +
                  15599.8389694042*x2*x4;
              const double x6 = exp(8.86 - 10215.3784219002*x0);
              const double x7 = -10215.3784219002*x2*x6 - x5*x[1];
              const double x8 = -x5*x[0];
              const double x9 = pow(u[0], -4);
```

```
36              const double x10 = pow(u[0], -3);
37              const double x11 = x6*x[0];
38              return {std::vector<double>{-(x1 + x3 + x4)}, {x7, x8},
                    {20430.7568438003*x11*x10 - 104353956.302623*x9*x11 -
                    x[0]*x[1]*(-34017.7133655395*x1*x10 + 289301205.655*x1*x9 -
                    28381.6425120773*x3*x10 + 201379407.920838*x3*x9 -
                    31199.6779388084*x4*x10 + 243354975.871341*x4*x9)}, {}, {}, {}};
39      }
40      private:
41      F0oilShalePyrolysis(Adjacency adj, AdjacencyDiff adjDiff) :
            Expression(std::move(adj), std::move(adjDiff)) {}
42  };
```

Appendix L - Configuration File of Model 2.2

```
[standard model parameters]
FINAL_TIME 8
INTERVALS 50
RADAU_INTEGRATOR 3
LINEAR_SOLVER MA57
INIT_VARS SOLVE
TOLERANCE 1e-15
MAX_ITERATIONS 5000
MESH_ALGORITHM L2_BOUNDARY_NORM
MESH_ITERATIONS 5
REFINEMENT_METHOD LINEAR_SPLINE
USER_SCALING false

[constant derivatives]
LINEAR_OBJECTIVE false
QUADRATIC_OBJECTIVE_LINEAR_CONSTRAINTS false
LINEAR_CONSTRAINTS false

[optionals: ipopt flags]
MU_INIT_REFINEMENT 1e-16
MU_STRATEGY_REFINEMENT monotone

[optionals: output]
EXPORT_OPTIMUM_PATH ".generated/oilShalePyrolysis"
INITIAL_STATES_PATH ".generated/oilShalePyrolysis"

[optionals: mesh refinement]

[runtime parameters]
```

Appendix M - Plots for Model Diesel Motor

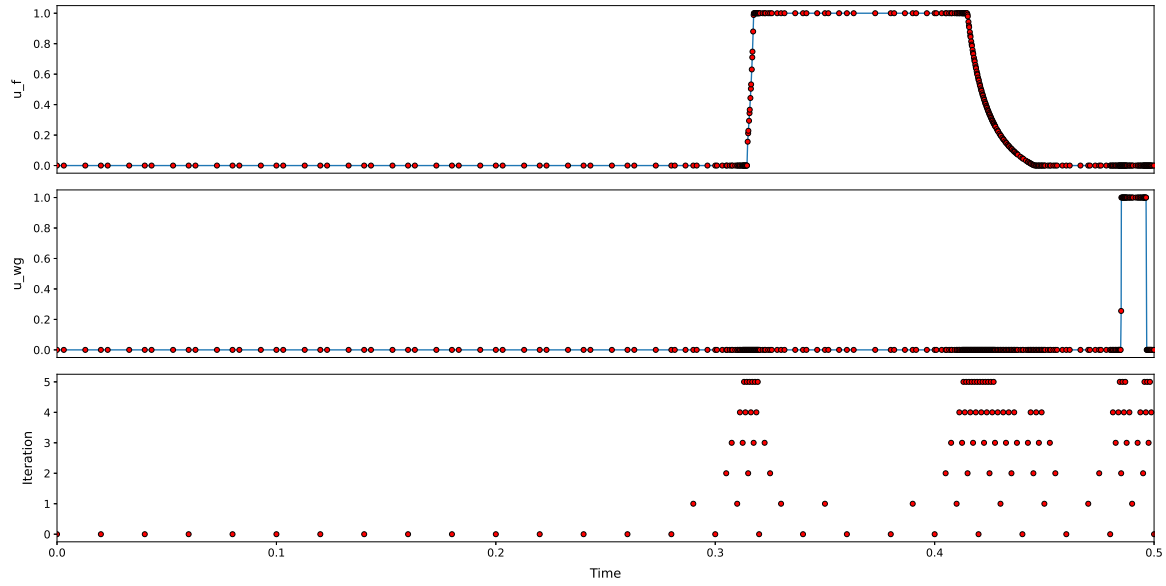


Figure 20: Optimal controls for $n = 25, m = 3, k_{max} = 5$ provided by GDOPT ($t_{i,j}$ drawn)

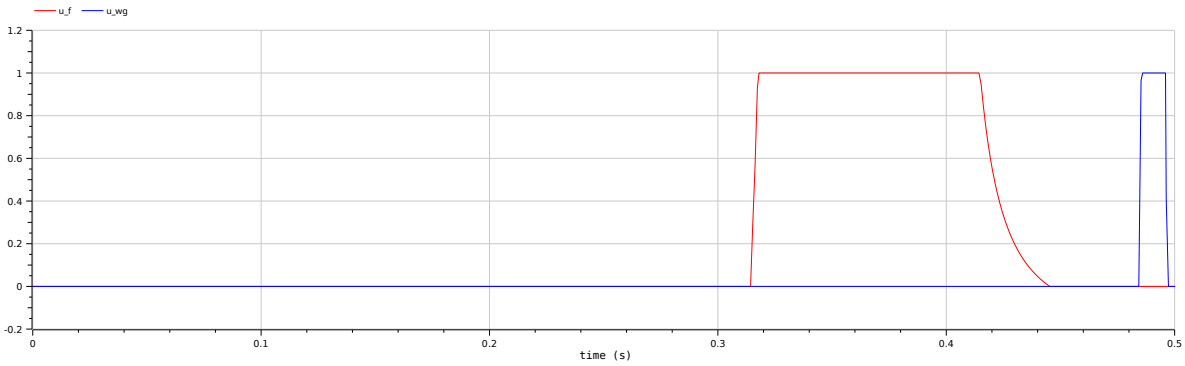


Figure 21: Optimal controls for $n = 250, m = 3$ provided by OpenModelica

Appendix N - Plots for Model Reusable Launch Vehicle

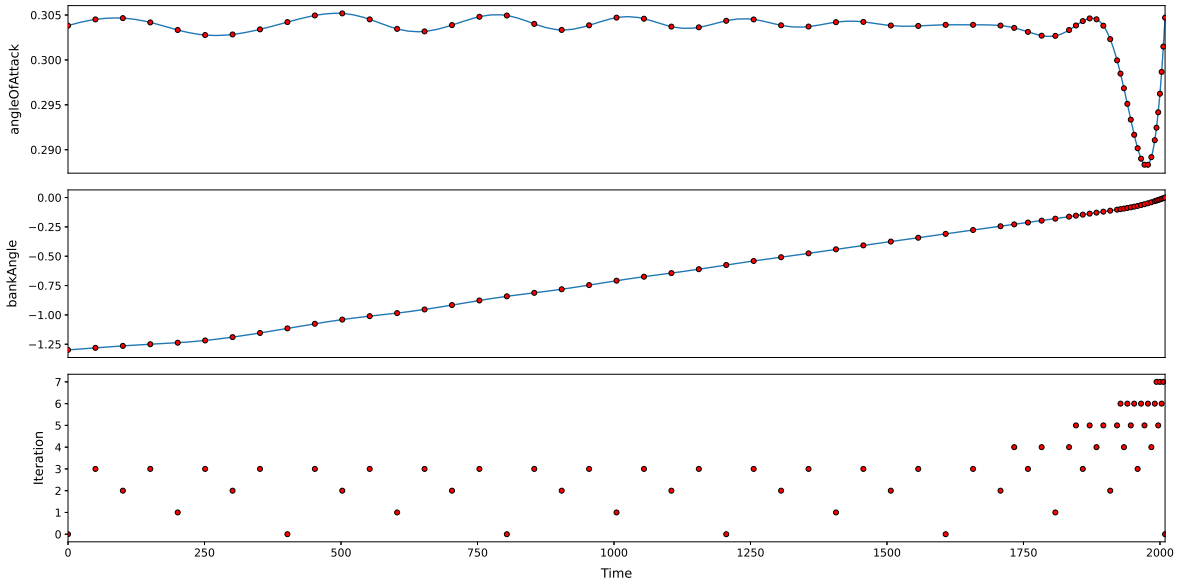


Figure 22: Mesh refinement history for the Reusable Launch Vehicle provided by GDOPT (t_i drawn)