KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

A General-Purpose Software Framework for Dynamic Optimization

Joel Andersson

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

October 2013

A General-Purpose Software Framework for Dynamic Optimization

Joel ANDERSSON

Supervisory Committee:
Prof. Dr. Ir. Hendrik Van Brussel, chair
Prof. Dr. Moritz Diehl, supervisor
Prof. Dr. Ir. Johan Åkesson, co-supervisor
(Department for Automatic Control, Lund University, Sweden)
Prof. Dr. Ir. Joos Vandewalle
Prof. Dr. Ir. Stefan Vandewalle
Prof. Dr. Ir. Jan Van Impe
Prof. Dr. Ir. Goele Pipeleers
Prof. Dr. Andrea Walther
(Department of Mathematics, University of Paderborn, Germany)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

October 2013

D/2013/7515/135
ISBN 978-94-6018-750-6

Preface

This dissertation explores the building blocks needed to efficiently formulate and solve optimal control problems. The premise of the thesis is that existing general-purpose solvers for optimal control are not powerful enough to treat large classes of industrially-relevant problems. At the same time, the implementation of efficient special-purpose solvers typically requires a prohibitively large programming effort.

To address this, we present a set of high-level building blocks, implemented in freely available software, which can be used to solve optimal control problems using a wide range of state-of-the-art optimal control methods with a comparably modest effort. The ultimate goal of the thesis is to make these methods more accessible users in academia and industry by drastically lowering the effort it takes to implement them.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor, professor Moritz Diehl. I thank you for letting me come to Leuven, for your never ending enthusiasm, for fruitful scientific discussions, for bringing me in contact with world-class researchers from around the world and not least, for allowing me to pursue what interested me the most.

Secondly, I would like to thank my co-supervisor, professor Johan Åkesson for arranging my visit to the Department of Automatic control in Lund, for being a source of much motivation and for great cooperation throughout my PhD.

I also thank all the members of my jury, professor Joos Vandewalle, professor Stefan Vandewalle, professor Jan Van Impe, professor Goele Pipeleers and professor Andrea Walther, whose professional feedback helped to improve both the presentation and the content of this thesis.

Moreover, I would like to thank all my fellow PhD students and post-docs in Leuven, in particular Joris Gillis, who has played a critical role in the development of CasADi and without whom much of the results presented in this thesis would not have been possible. Other colleagues in Leuven that deserve special mention include Carlo Savorgnan, Quoc Tran-Dinh, Boris Houska, Joachim Ferreau, Greg Horn, Attila Kozma, Mario Zanon, Sébastien Gros, Janick Frasch, Milan Vukov, Rien Quirynen, Vyacheslav Kungurtsev and Kurt Geebelen. I also thank Fredrik Magnusson in Lund and my colleagues in the vICERP consortium, in particular Nils Ahlbrink at DLR in Cologne, and my colleagues in the EMBOCON consortium, in particular Sergio Lucia at the T.U. Dortmund.

This work was made possible thanks to generous funding from the Helmholtz Gemeinschaft via vICERP, from the European Union via EMBOCON as well as from the KU Leuven Research Council via OPTEC.

Finally, I would like to thank my family and my friends. A very special word of gratitude goes to my girlfriend Agnieszka.

Joel Andersson
Leuven, October 2013

Abstract

Methods and software for derivative-based numerical optimization in general and simulation-based optimization in particular have seen a large rise in popularity over the past 30 years. Still, due to practical difficulties in implementing many of the methods in a fast and reliable manner, it remains an underused technology both in academia and in industry. To address this, we present a set of methods and tools with the aim of making techniques for dynamic optimization more accessible. In particular, we present CasADi, an open-source software framework for numerical optimization and algorithmic differentiation (AD) that offers a level of abstraction which is lower than algebraic modeling languages, but higher than conventional AD tools. We also discuss several of the many application problems which have already been addressed with CasADi by researchers from diverse fields.

Beknopte samenvatting

Tijdens de voorbije 30 jaar nam de populariteit van methodes en software voor gradiënt-gebaseerde optimalisatie, en meer bepaald simulatie-gebaseerde optimalisatie een hoge vlucht. Praktische moeilijkheden om deze methodes snel en betrouwbaar te implementeren, maken echter dat deze technologie onderbenut blijft in de academische alsook industriële praktijk. Om dit probleem aan te pakken, introduceren we hier een set van tools en methodes met als doel de technieken voor dynamische optimalisatie toegankelijker te maken. In het bijzonder introduceren we CasADi, een open-source software-pakket voor numerieke optimalisatie en algoritmische differentiatie (AD) met een abstractie-niveau lager dan dat van algebraïsche modelleertalen, maar hoger dan dat van conventionele AD tools. We bespreken aansluitend een greep uit de toepassingen die reeds behandeld werden met CasADi door onderzoekers uit diverse onderzoeksdomeinen.

Abbreviations

AD	Algorithmic differentiation
BDF	Backward differentiation formulas
CCS	Compressed column storage
CPU	Central processing unit
CRS	Compressed row storage
DAE	Differential-algebraic equation
END	External numerical differentiation
GN	Gauss-Newton
GPU	Graphics processing unit
IND	Internal numerical differentiation
IP	Interior point
IRK	Implicit Runge-Kutta
IVP	Initial value problem
KKT	Karush-Kuhn-Tucker
LICQ	Linear independence constraint qualification
LP	Linear program or linear programming
MPC	Model predictive control
NLP	Nonlinear program or nonlinear programming
NMPC	Nonlinear model predictive control

OCP	Optimal control problem
ODE	Ordinary differential equation
OO	Operator overloading (for AD)
PDE	Partial differential equation
RFP	Root-finding problem
RK	Runge-Kutta
SCP	Sequential convex programming
SCT	Source code transformation (for AD)
SQP	Sequential quadratic programming
TPBVP	Two-point boundary-value problem
VM	Virtual machine

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In *dynamic optimization* or *optimal control*, solutions are sought for decision-making problems constrained by dynamic equations in the form of ordinary differential equations (ODEs), differential-algebraic equations (DAEs) or partial differential equations (PDEs). Optimal control problems can cover many industrially relevant features, such as multi-stage problem formulations, problems with integer-valued decision variables, problems with multi-point constraints and problems with uncertainty. Because the problem formulation can become very general, along with a large set of different solution algorithms, it is difficult to implement software tools that treat optimal control problems with great generality. While tools do exist that deal with a broad range of problems, as discussed below, the usage of these tools is typically limited to comparably restricted problem formulations.

Optimal control problems

A basic formulation of an *optimal control problem* (OCP) is:

$$\begin{aligned} \underset{x,u,p}{\text{minimize}} \quad & \int_0^T J(x(t),u(t),p)\,dt + E(x(T),p) \\ \text{subject to} \quad & \left.\begin{aligned} &\dot{x}(t) = f(x(t),u(t),p), \\ &u(t) \in \mathcal{U}, \quad x(t) \in \mathcal{X}, \end{aligned}\right\} \quad t \in [0,T] \\ & x(0) \in \mathcal{X}_0, \quad x(T) \in \mathcal{X}_T, \quad p \in \mathcal{P} \end{aligned} \tag{OCP}$$

where $x(\cdot) \in \mathbb{R}^{N_x}$ is the vector of (differential) states, $u(\cdot) \in \mathbb{R}^{N_u}$ is the vector of free control signals and $p \in \mathbb{R}^{N_p}$ is a vector of free parameters in the model.

The OCP here consists of an objective function with a running cost or *Lagrange term* (J), an end-cost or *Mayer term* (E) as well as a *dynamic constraint* (f) in the form of an ODE with initial and terminal conditions ($\mathcal{X}_0$ and $\mathcal{X}_\mathrm{f}$). Finally, there are admissible sets for the states ($\mathcal{X}$), control ($\mathcal{U}$) and parameters ($\mathcal{P}$).

Methods for optimal control

Numerical methods for solving (OCP) and more general variants emerged with the birth of the electronic computer in the 1950's and were in the beginning typically based on a characterization on either the global optimum or the local optimum of the (infinite dimensional) optimal control problem. The characterization of the global optimum, the Hamilton-Jacobi-Bellman equations [130, 167], leads to such methods as *dynamic programming* [34, 38], whereas the characterization of the local optimum, the Pontryagin's maximum principle [155], leads to so-called *indirect methods* for optimal control. However, dynamic programming is normally unable to handle problems of large state-dimension due to Bellman's so-called *curse of dimensionality* [34] and for indirect methods, treating inequality constraints can be prohibitively difficult. Inequality constraints enter in (OCP) as bounds on the state and control trajectories. This fact, combined with the advancement of algorithms for *nonlinear programming*, in particular the emergence of sequential quadratic programming (SQP) [110,156,190], shifted the focus in the early 1980's to *direct methods*, where the control trajectory is parameterized forming a *nonlinear program* (NLP).

Two families of direct optimal control methods could be distinguished early on. In the *sequential approach*, represented by the direct single shooting method, existing methods and software for simulating differential equations were used to eliminate the state trajectory from the problem formulation, leaving the control trajectory and possibly a set of free parameters to be determined by the NLP solver. In the second approach, the *simultaneous approach*, the state trajectory was approximated by polynomials whose coefficients are determined together with the control trajectory in the NLP solver. The simultaneous approach, and in particular the *direct collocation approach* based on the collocation approach for simulating differential equations, was proposed as early as 1970 [142, 143], but only reached widespread popularity in the 1980s through the work of Biegler and Cuthrell [40,67]. An early overview of these approaches can be found in [42]. A key advantage of the simultaneous approach is that it allows handling of unstable systems – where simulating the system might be impossible for the current guess of the control trajectory.

A hybrid approach is the *direct multiple shooting method*, proposed in 1984 by

Bock and Plitt [50,51], in which the state trajectory is only partially eliminated from the NLP. The direct multiple shooting method has some important properties of the simultaneous approach, in particular the handling of unstable systems and the suitability for parallel computations. At the same time, it avoids the need to store the whole state trajectory, which can be prohibitively expensive for large-scale systems arising in e.g. PDE constrained optimization.

Nonlinear programming

As mentioned above, direct methods for optimal control result in *nonlinear programs* (NLPs). The NLPs considered in this thesis are of the general form:

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f(x) \\ \text{subject to} & g(x) = 0, \quad \underline{x} \leq x \leq \overline{x}, \quad \underline{h} \leq h(x) \leq \overline{h}, \end{array} \tag{NLP}$$

where $f : \mathbb{R}^{n_x} \to \mathbb{R}$ is the *objective function*, $g : \mathbb{R}^{n_x} \to \mathbb{R}^{n_g}$ is the *equality constraint function* and $h : \mathbb{R}^{n_x} \to \mathbb{R}^{n_h}$ is the *inequality constraint function*. In (NLP), the *decision variable* $x \in \mathbb{R}^{n_x}$ is constrained by the *simple bounds* $[\underline{x}, \overline{x}]$ and $h(x)$ is constrained by the *nonlinear bounds* $[\underline{h}, \overline{h}]$. We consider NLPs for which all involved functions are twice continuously differentiable, and if not stated otherwise, we allow the bounds ($\underline{x}$, $\overline{x}$, $\underline{h}$ and $\overline{h}$) to take infinite values.

Software for optimal control

The emergence of direct methods greatly facilitated the development of general-purpose software for optimal control. An early general-purpose OCP solver was MUSCOD by Bock in 1984 [51]. This code, which implements the direct multiple shooting method, was later extended and rewritten by Leineweber [131] forming MUSCOD-II in 1999. The MuShROOM code by Kirches [122] as well as the open-source ACADO Toolkit by Houska and Ferreau [115, 116] also implement direct multiple shooting and are specially designed to be used in real-time optimization. The ACADO Toolkit contains the ACADO Code Generation tool, which generates direct multiple shooting based controllers for embedded systems [117]. Tools implementing direct single shooting include DyOS [161] and dsoa [73].

General-purpose OCP tools implementing direct collocation include DIRCOL [177], PROPT [13] and SOCS [15]. The programming effort needed to efficiently implement direct collocation is however typically much lower than for shooting-based methods, largely thanks to algebraic modeling languages

such as AMPL [84], to be discussed in Section 3.5 as well as suitable solvers for the arising large and sparse NLPs such as IPOPT [179], to be discussed in Chapter 4. Direct collocation is therefore often implemented in application-specific codes.

Finally, an important development has been the emergence of derivative-based optimization capabilities within advanced simulation environments. This includes support for dynamic optimization in ASCEND [4], APMonitor [3], gPROMS [157], DYMOLA [6] and JModelica.org [20] as well as several domain-specific tools.

Goal of the thesis

As mentioned above, a popular way to solve optimal control problems using direct collocation is to use an algebraic modeling language. This automates two of the critical steps of the implementation, namely the calculation of derivative information and interfacing with third-party solvers.

The overall goal of the thesis is to allow this approach to be implemented with greater flexibility and extend it to direct and indirect multiple shooting approaches.

1.1 Overview of the thesis and contributions

The remainder of the thesis is structured as follows.

In Chapter 2, we go through the main solution steps of some of the most popular methods for large-scale optimal control. This discussion is not intended to be a self-contained introduction to optimal control, but does provide background and introduces concepts that will be used later in the text.

In Chapter 3, we give a brief but mostly self-contained introduction to algorithmic differentiation (AD). While the first part of this chapter is mainly intended as a service to readers not yet familiar with AD, the second part should be seen as a prelude to Chapter 7.

In Chapter 4, we give an overview of methods and software for derivative-based nonlinear programming. The main contribution of this chapter is an alternative derivation of the lifted Newton method for sequential quadratic programming in Section 4.5. This derivation is notably shorter and hopefully more accessible to readers than the derivation provided in the original paper by Albersmeyer and Diehl [23].

In Chapter 5, we present an novel algorithm for QP condensing, i.e. how to project structured QPs down to smaller but equivalent QPs. We show that the new algorithm in equivalent to the established QP condensing algorithm, widely used in optimization-based control and estimation, but with quadratic rather than cubic complexity in the prediction horizon. Numerical tests confirm that the algorithm is also faster in practice.

In Chapter 6, we present the open-source optimization framework CasADi, which is the main practical contribution of this thesis. CasADi was developed together with Joris Gillis in cooperation with several research groups. CasADi offers a new level of abstraction for solving numerical optimization problems, one that is lower than algebraic modeling languages such as AMPL or GAMS, but higher than conventional AD tools. The tool also shows that there is no inherent contradiction between working in a high-level programming language such as Python, but still getting the speed of optimized C-code. We show that the tool compares favorably with the AMPL for a subset of the CUTEr test suite.

In Chapter 7, we propose a way to efficiently embed solvers of initial-value problems in ordinary or differential-algebraic equations into symbolic expressions. We show that this enables fully automatic forward and adjoint sensitivity analysis to arbitrary order and greatly facilitates the implementation of shooting-based methods for optimal control.

In Chapter 8, we present work towards a general-purpose, structure-exploiting nonlinear programming solver based on the lifted Newton method. The goal is a tool that can efficiently solve the type of nonlinear programming problems arising in optimal control. As shown in the section, fast execution times and easy use in embedded applications are achieved by generating C-code for the computationally expensive parts of the method.

Finally, in Chapter 9, we list some of the applications where CasADi has been successfully applied to date. This includes the implementation of novel methods for optimal control and applications from a wide range of different engineering fields and from multiple research groups.

1.2 Notation

Apart from standard mathematical and engineering notation, we use the following conventions:

Vectors and matrices Vectors $x \in \mathbb{R}^n$ are interchangedly treated as column

vectors or $n - by - 1$ matrices. To declare a vector, the notation (a, b, c) is synonymous with $[a, b, c]^\intercal$. Scalars are also referred to as $1 - by - 1$ matrices.

Arithmetics If x and y are matrices, we shall use $x\,y$ to denote a matrix multiplication, and $x * y$ to denote the elementwise multiplication. If either x or y is a scalar, the notations are equivalent.

Calculus We use $\frac{\partial f}{\partial x} \in \mathbb{R}^{m \times n}$ to denote the Jacobian of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ with respect to x. The gradient, ∇f, is only used for scalar valued functions and always belongs to the same space as x, which can be either vector- or matrix-valued. Time derivatives of a physical quantity x are denoted $\dot{x}$ or $\frac{\partial x}{\partial t}$.

Algorithmic differentiation When considering the directional derivatives of a function $y = f(x)$, we use quantities with hats (e.g. $\hat{x}$) to denote *forward* directional derivatives, $\hat{y} = \frac{\partial f}{\partial x}\,\hat{x}$, and quantities with bars (e.g. $\bar{x}$) to denote *adjoint* directional derivatives, $\bar{x} = \left(\frac{\partial f}{\partial x}\right)^\intercal \bar{y}$, with natural extension to the case when x or y are matrix-valued.

Sets We use calligraphic fonts (e.g. $\mathcal{A}$) to denote sets with $|\cdot|$ denoting the number of elements in the set.

Inequalities An expression such as $a < b < c$ means $a < b$ and $b < c$. Inequalities are elementwise if not stated otherwise.

Chapter 2

The building blocks for dynamic optimization

Let us return to the simple optimal control problem (OCP) as stated in Chapter 1:

$$\begin{aligned} \underset{x,u,p}{\text{minimize}} \quad & \int_0^T J(x(t),u(t),p)\,dt + E(x(T),p) \\ \text{subject to} \quad & \left.\begin{aligned} &\dot{x}(t) = f(x(t),u(t),p), \\ &u(t) \in \mathcal{U}, \quad x(t) \in \mathcal{X}, \end{aligned}\right\} \quad t \in [0,T] \\ & x(0) \in \mathcal{X}_0, \quad x(T) \in \mathcal{X}_T, \quad p \in \mathcal{P} \end{aligned} \tag{OCP}$$

In this chapter, we provide details on some of the aforementioned methods to solve this problem. We will discuss two indirect methods: *indirect single shooting* and *indirect multiple shooting*, as well as three direct methods: *direct single shooting*, *direct multiple shooting* and *direct collocation*. We illustrate the relation between the methods in Figure 2.1. The purpose of this discussion is to expose the building blocks needed for solving real-world optimal control problems. These building blocks will then be discussed in more detail in the subsequent chapters.

To illustrate the methods we begin by introducing a small, yet industrially relevant OCP.

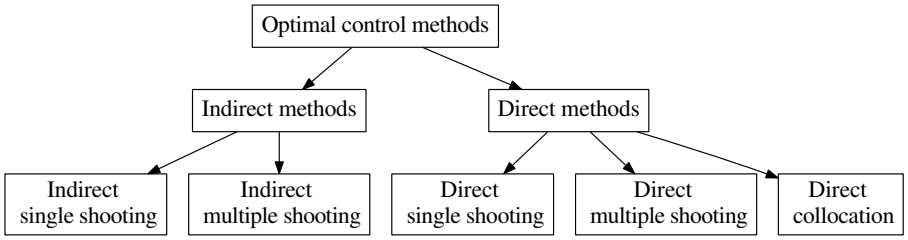

Figure 2.1: Numerical methods for dynamic optimization considered in the chapter

2.1 A guiding example: Minimal-fuel orbit transfer

The motion of a spacecraft in an orbital plane around Earth can be described by the following ODE taken from Cerf et al. [60][1]:

$$\begin{cases} \dot{r}(t) = & v(t)\,\sin\gamma(t) \\ \dot{\phi}(t) = & \dfrac{v(t)}{r(t)}\,\cos\gamma(t) \\ \dot{v}(t) = & -\dfrac{\mu}{r(t)^2}\,\sin\gamma(t) + \dfrac{u_1(t)}{m(t)} \\ \dot{\gamma}(t) = & \left(\dfrac{v(t)}{r(t)} - \dfrac{\mu}{v(t)\,r(t)^2}\right)\cos\gamma(t) + \dfrac{u_2(t)}{m(t)\,v(t)} \\ \dot{m}(t) = & -\dfrac{1}{g_0\,I_{\mathrm{sp}}}\,\sqrt{u_1(t)^2 + u_2(t)^2 + \epsilon} \end{cases} \qquad t \in [0,T] \quad (2.1)$$

where $(r(t), \phi(t))$ is the position of the spacecraft in polar coordinates, $v(t)$ is the speed, $\gamma(t)$ is the angle to the circle (the inclination) and $m(t)$ the mass of the spacecraft. This is illustrated in Figure 2.2. Furthermore, I_{sp} is the specific impulse of the spacecraft, g_0 is standard gravity and μ is Earth's standard gravitational parameter (the gravitational constant multiplied by Earth's mass). The ODE is controlled by the thrust vector $(u_1(t), u_2(t))$, which is constrained by:

$$u_1(t)^2 + u_2(t)^2 \leq u_{\max}^2, \quad t \in [0,T] \quad (2.2)$$

[1]The ϵ, taken to be 10^{-10} in our implementations was not part of the original problem formulation. It is included to keep $\partial\dot{m}/\partial u$ bounded for $u = 0$.

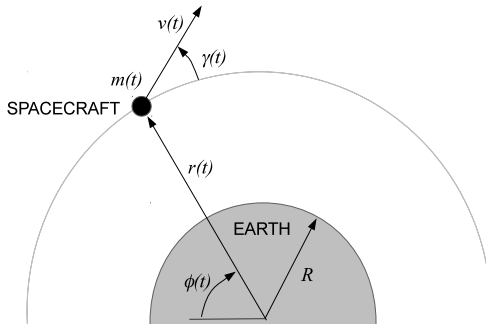

Figure 2.2: Geometry of the orbit transfer problem

As in the Cerf et al. paper [60], we consider the problem of finding a minimal-fuel trajectory for a spacecraft with 40 metric tons initial mass to go from an unstable sun-synchronous orbit to a circular final orbit:

$$\begin{cases} r(0) &= 200\text{ km} + R \\ \phi(0) &= 0 \\ v(0) &= 5.5\text{ km/s} \\ \gamma(0) &= 2\,\pi/180 \\ m(0) &= 40000\text{ kg} \end{cases} \qquad \begin{cases} r(T) &= 800\text{ km} + R \\ \\ v(T) &= 7.5\text{ km/s} \\ \gamma(T) &= 0 \\ \\ \end{cases} \tag{2.3}$$

where R is Earth's radius.

Defining the state vector $x(t) = (r(t), \phi(t), v(t), \gamma(t), m(t))$ and control vector $u(t) = (u_1(t), u_2(t))$ allows us to formulate the minimal-fuel orbit transfer as:

$$\begin{array}{ll} \underset{x,\,u}{\text{minimize}} & E(x(T)) := -m(T) \\ \text{subject to} & \text{Equation (2.1)} \\ & \text{Equation (2.2)} \\ & \text{Equation (2.3)} \end{array} \tag{2.4}$$

2.2 Indirect shooting methods for the orbit transfer problem

In the following, we shall go through the steps of using an indirect approach to solve problem (2.4). For more details and motivation for each step, we refer to

Bryson and Ho [53].

A necessary condition for optimality is given by Pontryagin's maximum principle [155], which for problem (2.4) has the form:

$$u(t) = \arg \max_{\|w\| \leq u_{\max}} H(x(t), \lambda(t), w), \tag{2.5}$$

where $H(x(t), \lambda(t), w)$ is the *Hamiltonian* of the system, which lacking an integral term in the objective is given by:

$$H(x(t), \lambda(t), w) = \lambda(t)^{\intercal} f(x(t), w). \tag{2.6}$$

We have also introduced the *costate* $\lambda(t) = (\lambda_r(t), \lambda_\phi(t), \lambda_v(t), \lambda_\gamma(t), \lambda_m(t))$, which is defined by the differential equation:

$$\dot{\lambda}(t) = -\nabla_x H(x(t), \lambda(t), u(t)). \tag{2.7}$$

The costate ODE is coupled to the original ODE via $x(t)$ and has free initial conditions and terminal conditions fixed whenever the corresponding state is free as explained in e.g. [53]. Here, this means:

$$\lambda_\phi(T) = -\left.\frac{\partial E}{\partial \phi}\right|_{t=T} = 0, \quad \lambda_m(T) = -\left.\frac{\partial E}{\partial m}\right|_{t=T} = 1. \tag{2.8}$$

For problem (2.4), the maximization condition (2.5) can be solved explicitly [60]:

$$u(t) = \begin{cases} \dfrac{u_{\max}\ (v(t)\,\lambda_v(t),\ \lambda_\gamma(t))}{\sqrt{v(t)^2\,\lambda_v(t)^2 + \lambda_\gamma(t)^2}} & \text{if } \sqrt{\lambda_v(t)^2 + \dfrac{\lambda_\gamma(t)^2}{v(t)^2}} > \dfrac{m(t)\,\lambda_m(t)}{g_0\,I_{\text{sp}}} \\ (0, 0) & \text{otherwise} \end{cases}. \tag{2.9}$$

Substituting the expression of the control (2.9) into the ODE (2.1) and costate ODE (2.7) gives a *two-point boundary value problem* (TPBVP). In the *indirect single shooting method*, an *initial-value problem* (IVP) for the *augmented ODE* consisting of the original ODE and the costate ODE is solved, defining the following *shooting function* for the time interval $[t_1, t_2]$:

$$F_{t_1,t_2} : \mathbb{R}^5 \times \mathbb{R}^5 \to \mathbb{R}^5 \times \mathbb{R}^5, \quad (x(t_1), \lambda(t_1)) \mapsto (x(t_2), \lambda(t_2)). \tag{2.10}$$

Using this function to eliminate the state and costate trajectories, the boundary conditions in Equation (2.3) and (2.8), give the system of equations:

$$\begin{pmatrix} r_T - (800\text{ km} + R) \\ \lambda_{\phi,T} \\ v_T - (7.5\text{ km/s}) \\ \gamma_T \\ \lambda_{m,T} - 1 \end{pmatrix} = 0 \tag{2.11}$$

where (x_T, λ_T) are given by evaluating $F_{0,T}(x_0, \lambda_0)$. This is a root-finding problem (RFP) with five equations and as many unknowns.

In practice, the shooting function $F_{0,T}$ is typically a computationally expensive, highly nonlinear function. By partitioning the time horizon into K intervals according to:

$$0 = t_0 < t_1 < \ldots < t_K = T \tag{2.12}$$

and introducing extra degrees of freedom in problem (2.11) corresponding to the state $x_k = x(t_k)$ and costate $\lambda_k = \lambda(t_k)$ at each of these points, we get the equivalent problem:

$$\begin{pmatrix} F_{t_0,t_1}(x_0, \lambda_0) - (x_1, \lambda_1) \\ \vdots \\ F_{t_{K-1},t_K}(x_{K-1}, \lambda_{K-1}) - (x_K, \lambda_K) \\ r_K - (800\text{ km} + R) \\ \lambda_{\phi,K} \\ v_K - (7.5\text{ km/s}) \\ \gamma_K \\ \lambda_{m,K} - 1 \end{pmatrix} = 0, \tag{2.13}$$

which has $5 + 10\,K$ equations and unknowns.

This is the *indirect multiple shooting* approach which was proposed as a solution method for two-point boundary value problems by Morrison [148] and used to solve optimal control problems by e.g. Bulirsch [54]. The formulation in problem (2.13) allows the shooting functions to be evaluated in parallel and can make use of an initial guess for the state and costate trajectory. It is also known to often exhibit faster local convergence than the smaller, more nonlinear problem (2.11) as discussed in e.g. [23, 48].

In terms of implementation, we can conclude that the indirect shooting approach will include at least the following steps:

- Formulation of the costate equations, a process that can be done efficiently and automatically with the reverse mode of algorithmic differentiation, see Chapter 3.
- Integration of the augmented ODE, while taking care to detect discontinuities by using e.g. an ODE integrator with root-finding functionality such as DASSL/DASRT [154] or SUNDIALS [113]. We propose an integrator formulation that allows embedding into gradient-based optimization methods in Chapter 7.
- Solution of the maximization condition either analytically or by embedding an optimizer into the ODE formulation.
- Solution of the root-finding problem (2.11) or (2.13), typically using a gradient-based approach. The structure of problem (2.13) makes it amenable to a Lifted Newton approach, presented for the more general NLP case in Chapter 4 and then explored in software in Chapter 8.

2.3 Direct control parameterization

The indirect approach is also referred to as an *optimize-then-discretize* approach, in that the optimality conditions of (OCP) are considered before a parameterization of the control trajectory is chosen. The converse approach, which starts with the parameterization of the control trajectory is referred to as a direct, or *discretize-then-optimize*, approach.

Assuming that the end time is known and fixed (cf. Section 2.7), we partition the time horizon according to Equation (2.12) and parameterize the control with a low order polynomial on each interval. For simplicity, we shall assume a piecewise constant control:

$$u(t) := u_k \quad \text{for} \quad t \in [t_k, t_{k+1}), \quad k = 0, \ldots, K-1. \tag{2.14}$$

The state trajectory can now be eliminated from the optimization problem by integrating the differential equation forward in time. For problem (OCP), this defines the discrete-time dynamics:

$$\begin{cases} F_k : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_x} \times \mathbb{R}, \\ (x_k, u_k, p) \mapsto (x_{k+1}, \displaystyle\int_{t_k}^{t_{k+1}} J(x(t), u_k, p, t)\, dt), \end{cases} \qquad k = 0, \ldots, K-1, \tag{2.15}$$

where the integral in the objective function is calculated together with the initial value problem solution. This can be done by augmenting the differential equation with one extra state or by using a quadrature formula as in e.g. [113].

This transforms (OCP) into the following discrete-time optimal control problem:

$$\begin{array}{lll} \underset{x,u,p}{\text{minimize}} & \displaystyle\sum_{k=1}^{K} q_k + E(x_K, p) & \\ \text{subject to} & (x_{k+1}, q_{k+1}) = F_k(x_k, u_k, p), & k \in \{0,\dots,K-1\} \\ & x_k \in \mathcal{X}, & k \in \{1,\dots,K-1\} \\ & u_k \in \mathcal{U}, & k \in \{0,\dots,K-1\} \\ & x_0 \in \mathcal{X}_0, \quad x_K \in \mathcal{X}_T, \quad p \in \mathcal{P}, & \end{array} \tag{2.16}$$

where the state constraints have been relaxed to only be enforced at the grid points.

2.4 Direct single shooting

With the initial conditions known, we can use recursion to eliminate the state trajectory, giving the following NLP with $n_p + n_x + K\,n_u$ degrees of freedom:

$$\begin{array}{ll} \underset{\substack{p,\,x_0,\\ u_0,\dots,u_{K-1}}}{\text{minimize}} & \displaystyle\sum_{k=1}^{K} q_k(p, x_0, u_0, \dots, u_{k-1}) + E(x_K(p, x_0, u_0, \dots, u_{K-1})) \\ \text{subject to} & \begin{bmatrix} \underline{p} \\ \underline{x}_0 \\ \underline{u} \\ \vdots \\ \underline{u} \end{bmatrix} \le \begin{bmatrix} p \\ x_0 \\ u_0 \\ \vdots \\ u_{K-1} \end{bmatrix} \le \begin{bmatrix} \overline{p} \\ \overline{x}_0 \\ \overline{u} \\ \vdots \\ \overline{u} \end{bmatrix}, \\ & \begin{bmatrix} \underline{x} \\ \vdots \\ \underline{x} \\ \underline{x}_T \end{bmatrix} \le \begin{bmatrix} x_1(p, x_0, u_0) \\ x_2(p, x_0, u_0, u_1) \\ \vdots \\ x_K(p, x_0, u_0, \dots, u_{K-1}) \end{bmatrix} \le \begin{bmatrix} \overline{x} \\ \vdots \\ \overline{x} \\ \overline{x}_T \end{bmatrix}, \end{array} \tag{2.17}$$

with simplifications when x_0 is given an/or state bounds are absent such as in problem (2.4).

This is a *direct single shooting* parameterization. Instead of solving a root-finding problem with embedded ODE integrators as in indirect single and

multiple shooting, requiring only first order derivatives, we now need to solve an NLP, requiring first and (possibly) second order derivative information.

2.5 Direct multiple shooting

While the direct single shooting method results in an NLP that has relatively few degrees of freedom, it has several important drawbacks:

- The objective and constraint functions have a highly nonlinear dependence on the variable u, slowing down the convergence of the NLP solution.
- If an initial guess for $x_1, \ldots, x_K$ is available, it cannot be used to initialize the NLP solver.
- The recursive elimination requires the corresponding functions, F_k in (2.15), to be evaluated sequentially as opposed to in parallel.

These issues can be addressed by including the state trajectory in the NLP, which then has $n_p + n_x + K\,n_u + K\,n_x$ degrees of freedom:

$$
\begin{aligned}
&\underset{\substack{p, x_0, \ldots, x_K,\\ u_0, \ldots, u_{K-1}}}{\text{minimize}} && \sum_{k=1}^{K} q_k + E(x_K, p) \\
&\text{subject to} && \begin{bmatrix} \underline{p} \\ \underline{x}_0 \\ \underline{u} \\ \vdots \\ \underline{x} \\ \underline{u} \\ \underline{x}_T \end{bmatrix} \leq \begin{bmatrix} p \\ x_0 \\ u_0 \\ \vdots \\ x_{K-1} \\ u_{K-1} \\ x_K \end{bmatrix} \leq \begin{bmatrix} \overline{p} \\ \overline{x}_0 \\ \overline{u} \\ \vdots \\ \overline{x} \\ \overline{u} \\ \overline{x}_T \end{bmatrix}, \\
& && \begin{bmatrix} F_0(x_0, u_0, p) - (x_1, q_1) \\ \vdots \\ F_{K-1}(x_{K-1}, u_{K-1}, p) - (x_K, q_K) \end{bmatrix} = 0.
\end{aligned}
\tag{2.18}
$$

This state and control parameterization gives the basis of *Bock's direct multiple shooting method* [51]. It is important to note the special structure of the nonlinear constraint. Even if we do not eliminate the state trajectory as in

the single shooting method, we can still use this special structure in the NLP solver. This will be discussed in Chapter 4 and in Chapter 8 we will present work towards a software implementation addressing this problem class.

2.6 Direct collocation

When we went from direct single shooting to direct multiple shooting we essentially traded nonlinearity for problem size. The NLP in single shooting is small, but often highly nonlinear, whereas the NLP for multiple shooting is larger, but sparser and typically less nonlinear. The *direct collocation method* goes a step further in the same direction, resulting in an even larger, but even sparser and possibly less nonlinear NLP.

To illustrate the method, we return to the initial value problem (2.15). We begin by dividing each control intervals k into $N(k)$ finite elements, $[t_{k,i}, t_{k,i+1}]$, $i \in \{0 \dots N-1\}$. This is illustrated in Figure 2.3. For simplicity we assume the intervals to be equidistant and same for all k:

$$t_{k,i} = t_k + i\, h_k, \quad \text{for } k \in \{0 \dots K-1\}, \quad i \in \{0 \dots N\} \tag{2.19}$$

with $h_k := (t_{k+1} - t_k)/N$.

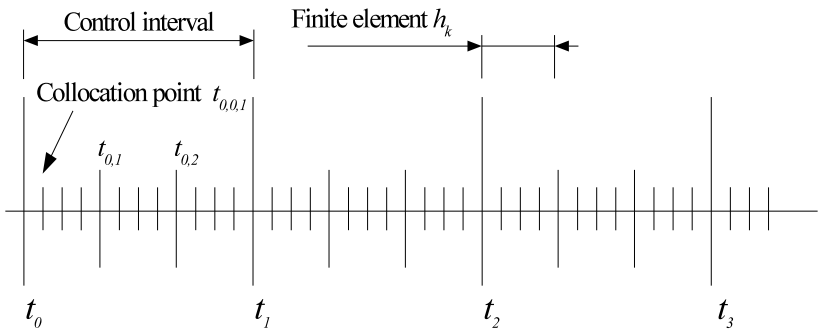

Figure 2.3: Time discretization in direct collocation

On each finite element, we parameterize the state trajectory with a low-order polynomial:

$$x(t) := \sum_{j=0}^{d} L_j\left(\frac{t - t_{k,i}}{h_k}\right) x_{k,i,j} \quad \text{for } t \in [t_{k,i}, t_{k,i+1}], \tag{2.20}$$

where $L_j(\tau)$ is a Lagrangian polynomial basis of order d:

$$L_j(\tau) = \prod_{r=0,\, r\neq j}^{d} \frac{\tau - \tau_r}{\tau_j - \tau_r}, \tag{2.21}$$

In (2.20), $x_{k,i,j}$ are polynomial coefficients, assuming $x_{k,N,0} := x_{k+1,0,0}$. Furthermore, $\tau_0 = 0$ and $\tau_1, \ldots, \tau_d$ in (2.21) are the *collocation points*. One popular choice for these are the *Legendre points* of order $d = 3$:

$$\tau_1 := 0.112702, \quad \tau_2 := 0.500000, \quad \tau_3 := 0.887298, \tag{2.22}$$

defining the time points:

$$\begin{aligned} t_{k,i,j} = t_{k,i} + \tau_j\, h_k \quad \text{for} \quad & k \in \{0 \ldots K-1\}, \\ & i \in \{0 \ldots N-1\}, \quad j \in \{0 \ldots d\}. \end{aligned} \tag{2.23}$$

For discussion on how to select the collocation points, and how this choice impacts the approximation accuracy and numerical stability of the integrator scheme, we refer to Biegler [43].

Since $x_{k,i,j} = x(t_{k,i,j})$ because of the Lagrangian basis, we can differentiate Equation (2.20) to get an approximation of the state derivative at each collocation point, giving the *collocation equations*:

$$\begin{aligned} & f(x_{k,i,j}, u_k, p) - \frac{1}{h_k} \sum_{r=0}^{d} C_{r,j}\, x_{k,i,r} = 0, \\ & \text{for} \quad k \in \{0 \ldots K-1\}, \quad i \in \{0 \ldots N-1\}, \quad j \in \{1 \ldots d\}. \end{aligned} \tag{2.24}$$

where $C_{r,j} := \frac{\partial L_r}{\partial \tau}(\tau_j)$ has been precomputed.

Furthermore, evaluating (2.20) at the end of each finite element gives us the *continuity equations*:

$$\sum_{r=0}^{d} D_r\, x_{k,i,r} - x^{+}_{k,i} = 0, \quad \text{for} \quad k \in \{0 \ldots K-1\}, \quad i \in \{0 \ldots N-1\}, \tag{2.25}$$

where $D_r := L_r(1)$ has been precomputed and $x^+_{k,i}$ is defined to be:

$$x^+_{k,i} = \begin{cases} x_{k+1,0} & \text{if } i = N-1 \\ x_{k,i+1} & \text{if } i < N-1 \end{cases} . \tag{2.26}$$

Finally, by integrating (2.20) from $t_{k,i}$ to $t_{k,i+1}$, we get an approximation of the Lagrange term in the objective function

$$\int_0^T J(x(t), u(t), p)\, dt \approx \sum_{k=0}^{K-1} \sum_{i=0}^{N-1} h_k \sum_{r=0}^{d} B_r\, J(x_{k,i,r}, u_k, p), \tag{2.27}$$

where $B_r := \int_0^1 L_r(\tau)\, d\tau$ has been precomputed.

In the direct collocation method, the collocated state is added as additional degrees of freedom to (2.18) and Equations (2.24) and (2.25) become additional

constraints. This results in the NLP:

$$
\begin{aligned}
&\underset{\substack{p,\,u_0,\ldots,u_{K-1},\\ x_{0,0,0},\ldots,\\ x_{K-1,N-1,d},\,x_{K,0,0}}}{\text{minimize}} && \sum_{k=0}^{K-1}\sum_{i=0}^{N-1} h_k \sum_{r=0}^{d} B_r\, J(x_{k,i,r}, u_k, p) + E(x_{K,0,0}, p) \\
&\text{subject to} && \begin{bmatrix} \underline{p} \\ \underline{x}_0 \\ \underline{x} \\ \vdots \\ \underline{x} \\ \underline{u} \\ \vdots \\ \underline{x} \\ \vdots \\ \underline{x} \\ \underline{u} \\ \underline{x}_T \end{bmatrix} \leq \begin{bmatrix} p \\ x_{0,0,0} \\ x_{0,0,1} \\ \vdots \\ x_{0,N-1,d} \\ u_0 \\ \vdots \\ x_{K-1,0,0} \\ \vdots \\ x_{K-1,N-1,d} \\ u_{K-1} \\ x_{K,0,0} \end{bmatrix} \leq \begin{bmatrix} \overline{p} \\ \overline{x}_0 \\ \overline{x} \\ \vdots \\ \overline{x} \\ \overline{u} \\ \vdots \\ \overline{x} \\ \vdots \\ \overline{x} \\ \overline{u} \\ \overline{x}_T \end{bmatrix}, \\
& && \begin{bmatrix} f(x_{k,i,1}, u_0, t_{k,i,1}) - \frac{1}{h_k} \sum_{r=0}^{d} C_{r,1}\, x_{k,i,r} \\ \vdots \\ f(x_{k,i,d}, u_0, t_{k,i,d}) - \frac{1}{h_k} \sum_{r=0}^{d} C_{r,d}\, x_{k,i,r} \\ \sum_{r=0}^{d} D_r\, x_{k,i,r} - x^{+}_{k,i} \end{bmatrix} = 0, \\
& && \qquad \text{for} \quad k \in \{0 \ldots K-1\}, \quad i \in \{0 \ldots N-1\}
\end{aligned}
\tag{2.28}
$$

where there are now $n_p + n_x + K\, n_u + K\, N\, (d+1)\, n_x$ optimization variables.

Note that the nonlinear constraint in (2.28) can in principle be used to sequentially eliminate the state trajectory. This would result transform the method into either the direct multiple shooting method of Section 2.5 or the direct single shooting method of Section 2.4 using a fixed-step collocation integrator scheme, cf. Section 3.11.

The direct collocation method therefore requires the following:

- A solver for very large and sparse[2] NLPs. These solvers, whether interior point (IP) methods or sequential quadratic programming (SQP) methods as discussed in Chapter 4, typically rely on efficient linear solvers for large and sparse, symmetric, but indefinite problems. An alternative approach, which is a variant of the structure exploiting NLP solution commonly used for the direct multiple shooting method will be discussed in Chapters 4 and 8.
- An efficient way to provide the first and second order derivative information, including sparsity structure, to the NLP solver. This is addressed in Chapter 3.

A comprehensive introduction to direct collocation can be found in the recent books by Biegler [43] and Betts [39].

2.7 Generalizations

The purpose of the presentation so far has been to expose some characteristics about and show the relation between some important methods for large-scale optimal control. Several important details have been left out to simplify the presentation. In the remainder of this chapter, we shall discuss some of those omissions, which can all be implemented with essentially the same set of building blocks.

Free end time

Until know, we have assumed that the end time be fixed. If this is not the case, the end time T can be handled as an additional entry in the parameter vector p and a time transformation can be applied to transform the time horizon to $[0, 1]$:

$$t = T\,\tilde{t} \quad \Rightarrow \quad \frac{\partial x}{\partial \tilde{t}}\left(\tilde{t}\right) = f\left(x(T\,\tilde{t}), u(T\,\tilde{t}), p\right)/T, \quad \tilde{t} \in [0, 1]. \tag{2.29}$$

A free initial time can be handled in the same way, see e.g. [131]. We refer to Bryson and Ho [53] for the treatment of free end time in the context of indirect methods.

[2] That is, the Jacobian of the constraint function and the Hessian of the Lagrangian are sparse, cf. Chapter 4.

Explicit time dependence

The direct collocation method can be trivially extended to handle explicit dependence on time t in the ODE function, as opposed to indirect dependence via the state or control vectors. For the shooting methods, this can be handled by augmenting the state vector with a new component defined by:

$$\dot{x}_{\text{new}}(t) = 1, \quad t \in [0, T], \quad x_{\text{new}}(0) = 0. \tag{2.30}$$

More efficiently, it is also possible to handle the explicit time dependence internally in the IVP solver. For details on this in the context of sensitivity analysis, we refer to e.g. [57].

Multistage problems

Many industrial optimization problems are characterized by different differential equations, or even different number of variables in the different regions of the time domain. Examples of such *multistage* OCPs (also referred to as *multiphase* in the literature) include trajectory optimization for multistage rockets and batch processes in chemical engineering [43]. The direct methods can be handled by scaling each stage to unit length and using different discrete time dynamics in the different regions. If state vector dimensions for the different stages differ, this can be handled by inserting *transition stages* [131]. For indirect methods, we again refer to Bryson and Ho [53].

Differential-algebraic equations

In many practical applications, the system dynamics are not given as ordinary differential equations (ODE) but as differential-algebraic equations (DAE):

$$\begin{cases} \dot{x}(t) &= f_{\text{D}}(x(t), z(t), u(t), p), \\ 0 &= f_{\text{A}}(x(t), z(t), u(t), p), \end{cases} \quad t \in [0, T] \tag{DAE}$$

which we assume to be of *semi-explicit form* and of *index-1* (see e.g. [107]). In this context, it means that the algebraic variable z is defined by an algebraic equation $0 = f_{\text{A}}(x, z, p, t)$ at all times by means of the implicit function theorem, implying in particular that $\partial f_{\text{A}}/\partial z$ must be invertible.

The direct collocation method can be readily extended to handle semi-explicit DAEs, by modifying the collocation equations (2.24) according to:

$$\begin{cases} f_{\mathrm{D}}(x_{k,i,j}, z_{k,i,j}, u_k, p) - \dfrac{1}{h_k} \displaystyle\sum_{r=0}^{d} C_{r,j}\, x_{k,i,r} = 0, \\ f_{\mathrm{A}}(x_{k,i,j}, z_{k,i,j}, u_k, p) = 0, \end{cases}$$

$$\text{for} \quad k \in \{0 \dots K-1\}, \quad i \in \{0 \dots N-1\}, \quad j \in \{1 \dots d\}. \tag{2.31}$$

Here, the collocated algebraic variables, $z_{k,i,j},\ k = 0, \dots, K,\ i = 0, \dots, N$ and $j = 1, \dots, d$, have been introduced as additional variables in the NLP. We leave the algebraic variable undefined in the beginning of each finite element ($j = 0$).

For the direct or indirect shooting methods, the discrete time dynamics (2.15) and the shooting function (2.10), respectively, amounts to solving initial value-problems of the form:

$$\begin{aligned} &F_{t_1,t_2} : \mathbb{R}^{n_x} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q}, \quad (x_1, p_1) \mapsto (x_2, q_2) \\ &\begin{cases} \dot{x}(t) &= f_{\mathrm{D}}(x(t), z(t), p_1), & t \in [t_1, t_2], \\ 0 &= f_{\mathrm{A}}(x(t), z(t), p_1), & x(t_1) = x_1, \quad q(t_1) = 0, \\ \dot{q}(t) &= f_{\mathrm{Q}}(x(t), z(t), p_1), & x_2 = x(t_2), \quad q_2 = q(t_2). \end{cases} \end{aligned} \tag{2.32}$$

We will return on how to solve problem (2.32) in Chapter 7, and in particular how a problem of this form can be efficiently embedded into a symbolic framework. The solution of (2.32) requires the IVP solver to find a solution to the algebraic equation before the start of the integration, which can be difficult and may fail if the function is nonlinear and no good solution guess for the algebraic variable exists. An alternative approach, used by e.g. Leineweber [131], is the *DAE relaxation strategy*. Here, the algebraic variables are included in the NLP formulation and (2.32) is modified according to:

$$\begin{aligned} &F_{t_1,t_2} : \mathbb{R}^{n_x} \times \mathbb{R}^{n_p} \times \mathbb{R}^{n_z} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q}, \quad (x_1, p, z_1) \mapsto (x_2, q_2) \\ &\begin{cases} \dot{x}(t) &= f_{\mathrm{D}}(x(t), z(t), p), \\ 0 &= f_{\mathrm{A}}(x(t), z(t), p) - \delta\left(\frac{t-t_1}{t_2-t_1}\right) f_{\mathrm{A}}(x(t), z_1, p), & t \in [t_1, t_2] \\ \dot{q}(t) &= f_{\mathrm{Q}}(x(t), z(t), p), \end{cases} \\ &x(t_1) = x_1, \quad q(t_1) = 0, \quad x_2 = x(t_2), \quad q_2 = q(t_2), \end{aligned} \tag{2.33}$$

where $\delta(\tau)$ is a monotonically decreasing function with $\delta(0) = 1$ and $\delta(1) = 0$.

This modified IVP has the same structure as (2.32) but with n_p+n_z parameters instead of n_p. Importantly, it has the *known* initial conditions (x_1, z_1), thus alleviating the IVP solver from the problem of solving for the algebraic variables at the initial time. Furthermore, at feasible points for the NLP, it will coincide with the original IVP. For more details, e.g. on how to choose the function $\delta(\tau)$, we refer to Leineweber's dissertation [131].

Control discretization

Instead of piecewise constant control discretization in the direct methods, a piecewise polynomial can be used. The control discretization may also be as fine as the state discretization. This is important for collocation in general, where the finer control discretization comes at little additional cost, and *pseudospectral* collocation methods [35] that employ high order polynomials, in particular.

2.8 Conclusion

As shown in this chapter, the solution of optimal control problems using either the direct or indirect approach can be divided into a number of well-defined tasks. In the subsequent chapters, we will show how these tasks can be performed efficiently and to a large extent automatically from high-level programming languages such as Python or Octave. In particular, this allows any of the methods presented in this chapter, including their generalizations in Section 2.7, to be implemented efficiently with a modest programming effort.

Chapter 3

Algorithmic differentiation and sensitivity analysis

Algorithmic differentiation (AD) is a technique for evaluating derivatives of computer represented functions which has proven useful in nonlinear optimization.

The technique delivers directional derivatives, up to machine precision, of arbitrary differentiable functions for a computational cost of the same order of magnitude as the cost of evaluating the original function. These directional derivatives can be either Jacobian-times-vector products, in the *forward mode*, or (row)vector-times-Jacobian product, in the *reverse mode*.

In this chapter, we will give a general introduction to AD in Sections 3.1 to 3.8. This includes using linear algebra to deduce algorithms for the forward and reverse modes in Sections 3.2 and 3.3 as well as discussing how complete Jacobians and Hessians can be calculated in Section 3.7. For a more complete introduction, we refer to Griewank and Walther [104]. In Section 3.9, we discuss differentiation of algorithms involving implicitly defined functions. Finally, in Sections 3.10 and 3.11 we discuss how discrete-time and continuous-time integrators can be embedded into AD algorithms.

3.1 Introduction to AD

Let us consider a sufficiently smooth nonlinear function $F : \mathbb{R}^{n_0} \to \mathbb{R}^{n_K}$:

$$y = F(x) \tag{3.1}$$

We assume that F is given in the form of an algorithm, typically a computer program, with a number of intermediate variables $z_k \in \mathbb{R}^{n_k}, k = 1, \ldots, K$, each depending on a subset $\mathcal{I}_k$ of the previous variables:

Algorithm 3.1 Calculation of $y := F(x)$ in (3.1)

$z_0 \leftarrow x$
for $k = 1, \ldots, K$ **do**
  $z_k \leftarrow f_k\left(\{z_i\}_{i \in \mathcal{I}_k}\right)$
end for
$y \leftarrow z_K$
return y

By taking the total derivative of every line of the algorithm, we get a new algorithm for calculating both $F(x)$ and its Jacobian $\frac{dF}{dx}(x)$:

Algorithm 3.2 Calculation of $y := F(x)$ and $J := \frac{dF}{dx}(x)$

$z_0 \leftarrow x$
$\frac{dz_0}{dx} \leftarrow I$
for $k = 1, \ldots, K$ **do**
  $z_k \leftarrow f_k\left(\{z_i\}_{i \in \mathcal{I}_k}\right)$
  $\frac{dz_k}{dx} \leftarrow \sum_{i \in \mathcal{I}_k} \frac{\partial f_k}{\partial z_i}\left(\{z_i\}_{i \in \mathcal{I}_k}\right) \frac{dz_i}{dx}$
end for
$y \leftarrow z_K$
$J \leftarrow \frac{dz_K}{dx}$
return y, J

We can write Algorithm 3.2 as the system of linear equations

$$\frac{dz}{dx} = B + L\,\frac{dz}{dx}, \qquad J = A^{\mathsf{T}}\,\frac{dz}{dx}, \tag{3.2}$$

where we have defined:

$$z = \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_K \end{pmatrix}, \quad A = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ I \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} I \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \tag{3.3}$$

with I and 0 of appropriate dimensions, as well as the *extended Jacobian*,

$$L = \begin{pmatrix} 0 & \dots & \dots & 0 \\ \frac{\partial f_1}{\partial z_0} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \frac{\partial f_K}{\partial z_0} & \dots & \frac{\partial f_K}{\partial z_{K-1}} & 0 \end{pmatrix}, \tag{3.4}$$

whose coefficients depend on the intermediate variables defined in Algorithm 3.1.

Since L is strictly lower triangular, $(I - L)$ is invertible, and we can eliminate $\frac{dz}{dx}$ from Equation (3.2), giving an explicit expression for the Jacobian:

$$J = A^\intercal (I - L)^{-1} B \tag{3.5}$$

3.2 The forward mode

In the *forward mode*, we conceptually multiply the Jacobian with a *forward seed matrix* $\hat{X} \in \mathbb{R}^{n_0 \times m}$ from the right. From Equation (3.5):

$$\hat{Y} := J \hat{X} = A^\intercal (I - L)^{-1} B \hat{X} \tag{3.6}$$

The multiplication with $(I - L)^{-1}$ from the right can be performed with a blockwise forward substitution. Also including the calculation of the intermediate variables, this results in the algorithm:

Algorithm 3.3 The forward mode of algorithmic differentiation

$z_0 \leftarrow x$
$\hat{Z}_0 \leftarrow \hat{X}$
for $k = 1, \ldots, K$ **do**
  $z_k \leftarrow f_k\left(\{z_i\}_{i \in \mathcal{I}_k}\right)$
  $\hat{Z}_k \leftarrow \sum_{j \in \mathcal{I}_k} \frac{\partial f_k}{\partial z_j}\left(\{z_i\}_{i \in \mathcal{I}_k}\right) \hat{Z}_j$
end for
$y \leftarrow z_K$
$\hat{Y} \leftarrow \hat{Z}_K$
return $y, \hat{Y}$

In each iteration of the for-loop, in addition to evaluating the non-differentiated function, we evaluate the following *forward derivative propagation* function:

$$\hat{f}_k\left(z_k, \{z_i, \hat{Z}_i\}_{i \in \mathcal{I}_k}\right) := \sum_{j \in \mathcal{I}_k} \frac{\partial f_k}{\partial z_j}\left(z_k, \{z_i, \hat{Z}_i\}_{i \in \mathcal{I}_k}\right) \hat{Z}_j, \tag{3.7}$$

which is assumed cheap either since f_k is an elementary operation with cheap, closed-form expressions for its partial derivatives, or because it can, in turn, be calculated by applying Algorithm 3.3 to the calculation of f_k. Note that we allow $\hat{f}_k$ to depend on z_k, for reasons that will become clear in Section 3.4 and 3.9. Also note that since the forward substitution can be performed simultaneously with the calculation of F, the potentially huge matrix L does not need to be precalculated.

If F is decomposed into a suitable set of atomic operations – see Section 3.4 below – we are able to get upper complexity bounds for some measure of the computational time, which can be formalized as a combination of memory movements and floating point operations, TIME{*operation*}, as well as memory requirements, MEMORY{*operation*}.

- $\text{TIME}\left\{J(x)\,\hat{X}\right\} \leq \eta_1 \cdot m \cdot \text{TIME}\{F(x)\}$
- $\text{MEMORY}\left\{J(x)\,\hat{X}\right\} \leq \eta_2 \cdot m \cdot \text{MEMORY}\{F(x)\}$

where η_1 and η_2 are small constants. For details, we refer to [104, Chapter 4]. In order to limit memory usage, which has a component proportional to the

number of columns p of the seed matrix, the calculation can be performed blockwise with at most $m_{\max}$ columns at a time[3], giving:

- $\text{MEMORY}\left\{J(x)\,\hat{X}\right\} \leq \eta_2 \cdot \min(m, m_{\max}) \cdot \text{MEMORY}\{F(x)\}$

3.3 The reverse mode

In the *reverse mode*, we conceptually multiply the *transpose* of the Jacobian with an *adjoint seed matrix* $\bar{Y} \in \mathbb{R}^{n_K \times m}$ from the right:

$$\bar{X} := J^\intercal\,\bar{Y} = B^\intercal\,(I - L)^{-\intercal}\,A\,\bar{Y} \tag{3.8}$$

This can be calculated by a blockwise backward substitution for $(I - L)^\intercal$, defining the following algorithm:

Algorithm 3.4 The reverse mode of algorithmic differentiation

$z_0 \leftarrow x$
for $k = 1, \ldots, K$ **do** ▷ **"Taping"**
  $z_k \leftarrow f_k\left(\{z_i\}_{i \in \mathcal{I}_k}\right)$
end for
$y \leftarrow z_K$
for $k = 0, \ldots, K-1$ **do** ▷ **"Reset $\bar{Z}$"**
  $\bar{Z}_k \leftarrow 0$
end for
$\bar{Z}_K \leftarrow \bar{Y}$
for $k = K, \ldots, 1$ **do** ▷ **"Backward sweep"**
  for $j \in \mathcal{I}_k$ **do**
    $\bar{Z}_j \leftarrow \bar{Z}_j + \left(\dfrac{\partial f_k}{\partial z_j}\left(\{z_i\}_{i \in \mathcal{I}_k}\right)\right)^\intercal \bar{Z}_k$ ▷ $Note: \bar{Z}_j$ overwritten
  end for
end for
$\bar{X} \leftarrow \bar{Z}_0$
return $y, \bar{X}$

Here, the evaluation depends on the following *adjoint derivative propagation function*:

$$\bar{f}_k\left(z_k, \bar{Z}_k, \{z_i\}_{i \in \mathcal{I}_k}\right) := \left\{\left(\frac{\partial f_k}{\partial z_j}\left(\{z_i\}_{i \in \mathcal{I}_k}\right)\right)^\intercal \bar{Z}_k\right\}_{j \in \mathcal{I}_k} \tag{3.9}$$

[3]In the implementation in Chapter 6, we use $m_{\max} = 64$ by default, cf. Section 3.7

which is cheap if the partial derivatives of f_k are readily available or if (3.9) can, in turn, be calculated using Algorithm 3.4.

The backward substitution requires us to access the columns of L in reverse order, i.e. opposite to the order of calculation. This means that either z_k, as assumed in Algorithm 3.4, or L needs to be stored (or recalculated) when calculating F, a process that is known in the AD literature as *taping*. These values are then accessed during the backward substitution, or the *backward sweep*.

As in the forward mode, we are able to get upper complexity bounds for computational cost and memory requirements:

- $\text{TIME}\left\{J^{\intercal}\bar{Y}\right\} \leq \omega_1 \cdot m \cdot \text{TIME}(F(x))$
- $\text{MEMORY}\left\{J^{\intercal}\bar{Y}\right\} \leq \omega_2 \cdot \min(m, m_{\max}) \cdot \text{MEMORY}\{F(x)\} + \omega_3 \sum_{k=0}^{K} n_k$

where ω_1, ω_2 and ω_3 are small constants, cf. [104, Chapter 4]. To conserve memory, the calculation is assumed performed with at most $m_{\max}$ columns at a time as above.

3.4 Scalar- and matrix-valued atomic operations

When deriving Algorithms 3.3 and 3.4, the intermediate variables z were allowed to be vector-valued. In most implementations, however, the decomposition of function F in (3.1) is made so that the intermediate variables are scalar-valued functions with closed form expressions for its partial derivatives, e.g.

$$u\,v, \quad u+v, \quad -u, \quad 1/u, \quad \sin(u), \quad \log(u), \quad e^u, \quad \text{etc.}$$

The derivative propagation rules for these operations are shown in Table 3.1:

Table 3.1: Essential scalar atomic operations

Unary operation $f(u)$	Forward propagation $\hat{f}(w, u, \hat{u})$	Adjoint propagation $\bar{f}(w, \bar{w}, u)$
$w = -u$	$-\hat{u}$	$\{-\bar{w}\}$
$w = 1/u$	$-w^2\,\hat{u}$	$\{-w^2\,\bar{w}\}$
$w = \sin(u)$	$\cos(u)\,\hat{u}$	$\{\cos(u)\,\bar{w}\}$
$w = \log(u)$	$\hat{u}/u$	$\{\bar{w}/u\}$
$w = e^u$	$w\,\hat{u}$	$\{w\,\bar{w}\}$

Binary operation $f(u, v)$	Forward propagation $\hat{f}(w, u, \hat{u}, v, \hat{v})$	Adjoint propagation $\bar{f}(w, \bar{w}, u, v)$
$w = u\,v$	$\hat{u}\,v + u\,\hat{v}$	$\{v\,\bar{w},\ u\,\bar{w}\}$
$w = u + v$	$\hat{u} + \hat{v}$	$\{\bar{w},\ \bar{w}\}$

We refer to [104, Chapter 2] for a discussion on how to select a suitable set of scalar atomic operations that ensures that the complexity bounds for AD hold.

There are however, good reasons to allow for *vector-* or (by generalizing Algorithms 3.3 and 3.4) *matrix-valued* atomic operations in the AD algorithm. This often has advantages both in terms of memory usage and speed, which can be seen by considering reverse mode AD of a function performing matrix-matrix multiplication:

$$W = f(U, V) = U\,V \tag{3.10}$$

which as shown in e.g. [89] has the particularly simple adjoint propagation rule

$$\bar{f}(W, \bar{W}, U, V) = \left\{\bar{W}\,V^\intercal,\quad U^\intercal\,\bar{W}\right\} \tag{3.11}$$

where the matrix multiplication is performed over the first two dimensions of the now third-order tensor $\bar{W}$.

While this expression can be calculated very efficiently using libraries such as LAPACK [25], a breakdown into scalar-valued operations would cause the number of stored elements to grow rapidly with the matrix dimensions and have much worse cache utilization.

Other matrix-valued operations that have analytic forms for forward as well as adjoint derivative propagations include matrix inverses, determinants, solutions of linear systems of equations, singular value decompositions (SVD), Cholesky factorizations and matrix norms as shown in e.g. [89, 90, 174]. We list the propagation rules for the most essential matrix operations in Table 3.2.

Table 3.2: Essential matrix atomic operations (cf. [89])

Unary operation[a] $f(U)$	Forward propagation $\hat{f}(W, U, \hat{U})$	Adjoint propagation $\bar{f}(W, \bar{W}, U)$
$W = U^\intercal$	$\hat{U}^\intercal$	$\bar{W}^\intercal$
$W = U^{-1}$	$-W\,\hat{U}\,W$	$-W^\intercal\,\bar{W}\,W^\intercal$
$w = \det(U)$	$w\,\mathrm{tr}(U^{-1}\,\hat{U})$	$(\bar{w}\,w)\,U^{-\intercal}$
$w = \|U\|_\mathrm{F}$	$(1/w)\,\mathrm{tr}(U^\intercal\,\hat{U})$	$(\bar{w}/w)\,U$

Binary operation $f(U, V)$	Forward propagation $\hat{f}(W, U, \hat{U}, V, \hat{V})$	Adjoint propagation $\bar{f}(W, \bar{W}, U, V)$
$W = U\,V$	$\hat{U}\,V + U\,\hat{V}$	$\{\bar{W}\,V^\intercal,\, U^\intercal\,\bar{W}\}$
$w = \mathrm{tr}(U^\intercal\,V)$	$\mathrm{tr}(\hat{U}^\intercal\,V) + \mathrm{tr}(U^\intercal\,\hat{V})$	$\{\bar{w}\,V,\, \bar{w}\,U\}$
$W = U^{-1}\,V$	$U^{-1}\,(\hat{V} - \hat{U}\,W)$	$\{-(U^{-\intercal}\,\bar{W})\,W^\intercal,\, U^{-\intercal}\,\bar{W}\}$

[a] Lower-case used for scalar-valued variables

Again, the operations are performed over the first two dimensions of the third-order tensors $\hat{U}$, $\hat{V}$ and $\bar{W}$.

3.5 Implementation of AD

There are two classical ways to implement AD:

- In the *operator overloading* (OO) approach, Algorithms 3.3 or 3.4 are evaluated during or after the numerical evaluation of the original (non-differentiated) function. The typical way to implement it is through *operator overloading* in languages such as C++, which allows mathematical operations to also include the forward seed propagation in (Algorithm 3.3) or the recording of the intermediate variables (Algorithm 3.4, first part). The OO approach can be thought of as an *evaluate-then-differentiate* approach.
- In *source code transformation* (SCT) approach, Algorithms 3.3 or 3.4 are performed on the expression graphs defining a function. The expression graphs are typically given as the parsed source code inside a compiler. This will result in new expression graphs that can be used to produce new source code for the differentiated expressions. The SCT approach can be thought of as a *differentiate-then-evaluate* approach.

For a discussion about OO and SCT and their pros and cons, we refer to [46].

Virtual machines for operator overloading AD

Operator overloading tools are typically able to record a so-called *operation trace*, which is a sorted sequence of all encountered operations. It contains all the information needed to numerically evaluate the function as long as the control flow path does not change. The trace can be traversed either in the order of calculation, for forward mode AD, or in the reverse order, for reverse mode AD. This evaluation by traversing the operation trace can be described as an evaluation in a *virtual machine* (VM). The VM can be either *registry-based* or *stack-based*. In a registry-based VM, the values of the intermediate operations are stored in a random access data structure we shall call the *work vector*. The instructions of the virtual machine then refer to operations acting on the work vector. To limit the size of the work vector, and hence the overall memory use, it is important to reuse the elements when possible. This can be done by analyzing when elements go out of scope, the *live variable ranges* of the work vector elements. In a stack-based VM, in contrast, the values of the intermediate operations are stored on a *stack*, i.e. a *last in, first out* type data structure.

Software tools

Conventional AD tools that use the OO approach include ADOL-C [102], CppAD [5], Sacado [19] and FADBAD++ [8] for C/C++ and MAD [79] for MATLAB. Tools that use the SCT approach include ADIC [1] for C/C++, ADIFOR [45] for Fortran, ADiMat [2] for MATLAB as well as OpenAD [12] and TAPENADE [16] for both C/C++ and Fortran.

The first popularized implementation of AD with matrix-valued operations was done in MATLAB by Verma and Coleman through their ADMAT/ADMIT packages [63, 174]. More recent implementations, also in MATLAB, include the ADiMat package [47], MAD [79] and MSAD [120]. Implementations in other programming languages include the Python-based Theano package [36].

3.6 Higher order derivatives

Until now we have only considered first order directional derivatives. Higher order derivatives can be calculated by repeatedly applying the source-code transformation approach to the algorithm of a function. This assumes that the atomic operations in Section 3.4 forms a set which is closed under forward

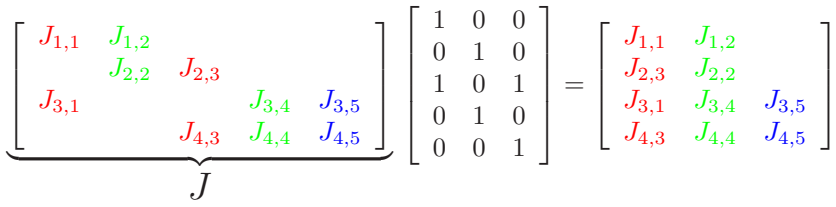

Figure 3.1: Jacobian compression via Curtis-Powell-Reid seeding

and reverse mode differentiation, i.e. differentiating the algorithm will only give rise to operations from the same set.

Higher-order derivatives can also be calculated in the context of operator overloading, although implementations beyond second order are rare. For further discussion on this, we refer to Griewank and Walther [104].

3.7 Calculating complete Jacobians

To calculate complete Jacobians, Algorithm 3.3 can be executed with $\hat{X}$ being the identity matrix or, alternatively, Algorithm 3.4 can be executed with $\bar{Y}$ being the identity matrix. The cost for evaluating the Jacobian this way is proportional to the number of independent variables, in the forward mode, or the dimension of the function, in the reverse mode.

When the Jacobian is sparse and has a sparsity pattern as illustrated by the matrix J in Figure 3.1, Algorithm 3.3 can be made cheaper by compressing the seed matrix $\hat{X}$ after identifying subsets of *structurally orthogonal* columns of the Jacobian. Similarly, Algorithm 3.4 can be made cheaper by identifying subsets of structurally orthogonal rows of the Jacobian. This was first shown by Curtis, Powell, and Reid in 1974 [66], who also presented a greedy algorithm that today can be classified as a *direct*, *unidirectional* graph coloring algorithm for bipartite graphs. *Direct* in this context means that the nonzeros of J are given directly in the right hand side in Figure 3.1, in contrast to an *indirect* compression method which requires a solve after the compressed Jacobian has been calculated. *Unidirectional* means that either the rows or the columns are compressed as opposed to a *bidirectional* compression, which relies on a combination of forward and adjoint directional derivatives. The latter can be beneficial e.g. when the Jacobian contains both dense rows and dense columns.

The Hessian of a scalar-valued function can be obtained with a *forward-over-reverse* approach. Here the reverse mode is used to derive an algorithm for calculating the gradient and then the forward mode is applied to the gradient algorithm giving a Hessian-times-vector product for a computational cost proportional to calculating the original function. When calculating a complete Hessian, i.e. the Jacobian of the gradient, symmetry can be exploited in the Jacobian calculation. One popular such algorithm, used in this work, is the *star-coloring* algorithm by Fertin, Raspaud, and Reid [76].

Since the problem of finding the Jacobian compression requiring the least amount of directional derivatives is known to be NP-hard [134], practical algorithms are typically greedy heuristics, processing the rows and/or columns in a certain order. The quality of the resulting compression depends on this order and can often be improved by first permutating the Jacobian in a so-called *preordering step*. For more discussion on this and other topics related to graph coloring for Jacobian compression, we refer to Gebremedhin et al. [86] and the references therein.

The graph coloring requires prior knowledge of the sparsity pattern of the Jacobian. Automatically detecting the sparsity pattern is a nontrivial problem and for sparse problems often the most expensive step in the Jacobian calculation as reported in e.g. [150].

The major approaches for this are the *sparse-vector approach* [150] and the related *index domain approach* [101], where sets of indices are propagated through the algorithm, as well as the *bitvector* approach, where the AD algorithms – forward or reverse – are evaluated using a boolean data type. The latter approach is described in e.g. [88] and is also adopted in this thesis.

3.8 AD in domain-specific languages

All the AD tools mentioned in Section 3.5 have been designed to be able to handle existing user source code with minimal modifications. But AD can also be implemented inside domain-specific languages for computer algebra, mathematical programming and physical modeling. This is not to be confused with *symbolic differentiation*, which is a simpler, recursive algorithm to obtain symbolic expressions for the derivatives.

The implementation of true AD requires that the expression graphs can contain *shared subexpressions*, i.e. multiple references to the same expression must be allowed to be formed without this resulting in a copy of the expression. We

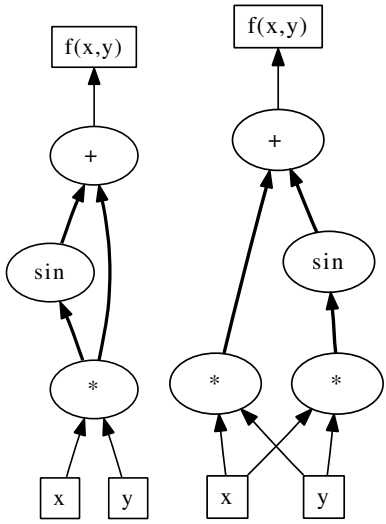

Figure 3.2: The expression graph for $f(x, y) = x\,y + \sin(x\,y)$ with (left) and without (right) treating $(x\,y)$ as a shared subexpression

illustrate this in Figure 3.2 where we show an expression defining the function $f(x, y) = x\,y + \sin(x\,y)$ with and without $x\,y$ as a shared subexpression.

Symbolic expressions in conventional computer algebra systems, including Maple, SymPy and MATLAB's Symbolic Math Toolbox do not allow shared subexpressions. We illustrate this with the following code segment for MATLAB's Symbolic Math Toolbox:

```
x = sym('x');
for i=1:100
  x = x*sin(x);
end
```

Since each iteration of the for-loop will cause a copy of the expression for `x` to be made, the final expression graph, if successfully formed, would contain some $2^{100} \approx 10^{30}$ nodes.

As a consequence, AD in e.g. Maple (see [147, 175]), is only supported for algorithms defined in the form of functions.

Algebraic modeling languages

One particular type of symbolic tools that implements AD is *algebraic modeling languages*. Algebraic modeling languages allow users to formulate mathematical programming problems in a representation that closely resembles standard mathematical notation. For example, an optimization problem such as

$$\begin{array}{ll} \underset{x,y}{\text{minimize}} & x\,y \\ \text{subject to} & x \geq 0, \quad y \geq 0, \quad x^2 + y^2 \leq 1, \end{array} \tag{3.12}$$

may be formulated in the algebraic modeling language AMPL [84] as:

```
var x >= 0;
var y >= 0;
maximize Cost: x*y;
subject to UnitCircle: x*x + y*y <= 1;
```

This problem is then parsed and reformulated to a canonical form such as (NLP) from Chapter 4. Using the symbolic representation of this canonical form, tools such as AMPL [84] and GAMS [10] then use AD to calculate exact first and second order derivative information and pass this information on to interfaced solvers.

3.9 Embedded implicit functions

It is possible to allow the atomic operations in AD to be the solution to a system of equations, as noted by e.g. [30, 33, 62]. In particular, let us assume that $w = f(u)$ is defined as the solution to a parametric root-finding problem of the form:

$$g(w) - A\,u = 0 \Leftrightarrow w := f(u), \tag{3.13}$$

where A is a known and constant matrix. Without loss of generality, we have assumed that u enters linearly in the equations, possibly by extending w with a dummy variable defined by $\tilde{w} = u$. An expression for the Jacobian can be obtained from the implicit function theorem (assuming regularity conditions are met):

$$\frac{\partial f}{\partial u}(u) = \left(\frac{\partial g}{\partial w}(w(u))\right)^{-1} A. \tag{3.14}$$

Conceptually multiplying this expression from the right with $\hat{U}$, allows us to define a forward derivative propagation rule:

$$\hat{f}(w, u, \hat{U}) = \left(\frac{\partial g}{\partial w}(w)\right)^{-1} A\,\hat{U}, \tag{3.15}$$

which can be calculated by a matrix-vector product followed by a linear solve for the Jacobian $\frac{\partial g}{\partial w}$. Note that if Newton's method is used to solve the original root-finding problem (3.13), then a method for calculating and factorizing this Jacobian is readily available. Equation (3.15) naturally extends to implicit functions with multiple inputs.

Similarly, we can derive an adjoint derivative propagation rule by conceptually multiplying the transpose of (3.14) with $\bar{W}$:

$$\bar{f}(w, \bar{W}, u) = \left\{ A^{\mathsf{T}} \left(\frac{\partial g}{\partial w}(w)\right)^{-\mathsf{T}} \bar{W} \right\}, \tag{3.16}$$

which can be calculated as a linear solve for the transpose of the Jacobian $\frac{\partial g}{\partial w}$, followed by a matrix-vector product. We note that common factorizations of the Jacobians such as LU or QR also allow solving for the transposed system. As in the forward case, Equation (3.16) naturally extends to implicit functions with multiple inputs.

Implicit functions with inexact Jacobians

The above formulas assume that an exact and up-to-date factorization of $\frac{\partial g}{\partial w}$ is available. When this is not the case, or when it is too expensive to update the factorization every time (3.19) or (3.20) is invoked, we can forgo this requirement as described in the following.

We begin by extending the problem formulation (3.13), to allow the function output to be a linear combination of the solution to the root-finding problem:

$$\begin{cases} 0 = g(z) - A\,u \\ w = B^{\mathsf{T}}\,z \end{cases} \quad \Leftrightarrow \quad w = f(u), \tag{3.17}$$

where A and B are given matrices.

We get the following expression for the Jacobian of this modified f:

$$\frac{\partial f}{\partial u}(u) = B^{\mathsf{T}} \left(\frac{\partial g}{\partial z}(z(u))\right)^{-1} A, \tag{3.18}$$

and hence the following forward and adjoint derivative propagation rules:

$$\hat{f}(w, u, \hat{U}) = B^{\mathsf{T}} \left(\frac{\partial g}{\partial z}(z(u)) \right)^{-1} A \hat{U}, \tag{3.19}$$

$$\bar{f}(w, \bar{W}, u) = \left\{ A^{\mathsf{T}} \left(\frac{\partial g}{\partial z}(z(u)) \right)^{-\mathsf{T}} B \bar{W} \right\}. \tag{3.20}$$

We can interpret (3.17) jointly with (3.19) as a new implicit function of the form (3.17):

$$\begin{aligned} 0 &= \begin{pmatrix} g(z) \\ \frac{\partial g}{\partial z}(z)\, \hat{z}_1 \\ \vdots \\ \frac{\partial g}{\partial z}(z)\, \hat{z}_m \end{pmatrix} - \begin{pmatrix} A & & & \\ & A & & \\ & & \ddots & \\ & & & A \end{pmatrix} \begin{pmatrix} u \\ \hat{u}_1 \\ \vdots \\ \hat{u}_m \end{pmatrix} \\ \begin{pmatrix} w \\ \hat{w}_1 \\ \vdots \\ \hat{w}_m \end{pmatrix} &= \begin{pmatrix} B & & & \\ & B & & \\ & & \ddots & \\ & & & B \end{pmatrix}^{\mathsf{T}} \begin{pmatrix} z \\ \hat{z}_1 \\ \vdots \\ \hat{z}_m \end{pmatrix}, \end{aligned} \tag{3.21}$$

where $\hat{u}_1, \ldots, \hat{u}_m$ are the columns of $\hat{U}$ and $\hat{w}_1, \ldots, \hat{w}_m$ are the columns of $\hat{W}$.

Similarly, we can interpret (3.17) jointly with (3.20) as the implicit function:

$$\begin{aligned} 0 &= \begin{pmatrix} g(z) \\ \frac{\partial g}{\partial z}(z)^{\mathsf{T}}\, \bar{z}_1 \\ \vdots \\ \frac{\partial g}{\partial z}(z)^{\mathsf{T}}\, \bar{z}_m \end{pmatrix} - \begin{pmatrix} A & & & \\ & B & & \\ & & \ddots & \\ & & & B \end{pmatrix} \begin{pmatrix} u \\ \bar{w}_1 \\ \vdots \\ \bar{w}_m \end{pmatrix} \\ \begin{pmatrix} w \\ \bar{u}_1 \\ \vdots \\ \bar{u}_m \end{pmatrix} &= \begin{pmatrix} B & & & \\ & A & & \\ & & \ddots & \\ & & & A \end{pmatrix}^{\mathsf{T}} \begin{pmatrix} z \\ \bar{z}_1 \\ \vdots \\ \bar{z}_m \end{pmatrix}, \end{aligned} \tag{3.22}$$

where $\bar{u}_1, \ldots, \bar{u}_m$ are the columns of $\bar{U}$ and $\bar{w}_1, \ldots, \bar{w}_m$ are the columns of $\bar{W}$.

We can then use an inexact Newton-type method to solve the augmented implicit functions (3.21) or (3.22). We refer to [113] and references therein for a discussion on how to solve root-finding problems of this form.

3.10 Embedded discrete-time integrators

As explored in Chapter 2, solving OCPs with shooting-based methods requires solvers for initial value problems (IVP) in differential equations to be embedded into either a root-finding problem or into an NLP. Before moving on to the continuous-time case in Section 3.11, we first treat the discrete-time case in the following.

Explicit one-step integration methods

Consider the function $F : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q}$ which is defined by Algorithm 3.5. It corresponds to a discrete-time one-step integrator with dynamics given by the functions $\Phi(x, u)$ and $\Psi(x, u)$.

Algorithm 3.5 Definition of $F : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q}$

input (x_0, u)
$q_0 = 0$
for $k = 0, \ldots, N-1$ **do**
  $x_{k+1} = \Phi(x_k, u)$
  $q_{k+1} = \Psi(x_k, u) + q_k$
end for
return (x_N, q_N)

We will refer to $x_k \in \mathbb{R}^{n_x}$ as the *state*, $u \in \mathbb{R}^{n_u}$ as the *parameter* and $q_k \in \mathbb{R}^{n_q}$ as the *summation state*.

Problems in this form appear e.g. as the result of (explicit) Runge-Kutta schemes for ODEs as noted in Section 3.11 below. We have also seen one example in Section 2.2.

The function F defined by Algorithm 3.5 has two vector-valued inputs and two vector-valued outputs. We can can write it more compactly as:

$$\{x_N, q_N\} = F(x_0, u) \tag{3.23}$$

In the following, we make a brief discussion how to obtain its forward and adjoint derivative propagation functions. That is, we seek ways of calculating

$$\begin{bmatrix} \hat{X}_N \\ \hat{Q}_N \end{bmatrix} := \begin{bmatrix} \frac{\partial x_N}{\partial x_0}(x_0, u) & \frac{\partial x_N}{\partial u}(x_0, u) \\ \frac{\partial q_N}{\partial x_0}(x_0, u) & \frac{\partial q_N}{\partial u}(x_0, u) \end{bmatrix} \begin{bmatrix} \hat{X}_0 \\ \hat{U} \end{bmatrix} \tag{3.24}$$

and

$$\begin{bmatrix} \bar{X}_0 \\ \bar{U} \end{bmatrix} := \begin{bmatrix} \frac{\partial x_N}{\partial x_0}(x_0, u) & \frac{\partial x_N}{\partial u}(x_0, u) \\ \frac{\partial q_N}{\partial x_0}(x_0, u) & \frac{\partial q_N}{\partial u}(x_0, u) \end{bmatrix}^{\mathsf{T}} \begin{bmatrix} \bar{X}_N \\ \bar{Q}_N \end{bmatrix} \tag{3.25}$$

respectively.

We get a forward derivative propagation rule for F by applying the forward mode of AD to (3.5). This results in function $\hat{F}$ defined by Algorithm 3.6.

Algorithm 3.6 Definition of
$\hat{F} : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_u \times m} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_q \times m}$

input $(x_0, u, \hat{X}_0, \hat{U})$
$q_0 = 0$
$\hat{Q}_0 = 0$
for $k = 0, \ldots, N-1$ **do**
  $x_{k+1} = \Phi(x_k, u)$
  $q_{k+1} = \Psi(x_k, u) + q_k$
  $\hat{X}_{k+1} = \frac{\partial \Phi}{\partial x}(x_k, u)\, \hat{X}_k + \frac{\partial \Phi}{\partial u}(x_k, u)\, \hat{U}$
  $\hat{Q}_{k+1} = \frac{\partial \Psi}{\partial x}(x_k, u)\, \hat{X}_k + \frac{\partial \Psi}{\partial u}(x_k, u)\, \hat{U} + \hat{Q}_k$
end for
return $(x_N, q_N, \hat{X}_N, \hat{Q}_N)$

We note that Algorithm 3.6 has the same structure as Algorithm 3.5 if we form the augmented variables $\tilde{x} := (x, \mathrm{vec}(\hat{X}))$, $\tilde{q} := (q, \mathrm{vec}(\hat{Q}))$ and $\tilde{u} := (u, \mathrm{vec}(\hat{U}))$.

Similarly, we get an adjoint derivative propagation rule by applying the reverse mode of AD to Algorithm 3.5, resulting in Algorithm 3.7.

Algorithm 3.7 Definition of $\bar{F} : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_q \times m} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_u \times m}$

input $(x_0, u, \bar{X}_N, \bar{Q})$
$q_0 = 0$
for $k = 0, \ldots, N-1$ **do**
  $x_{k+1} = \Phi(x_k, u)$
  $q_{k+1} = \Psi(x_k, u) + q_k$
end for
$\bar{U}_N = 0$
for $k = N-1, \ldots, 0$ **do**
  $\bar{X}_k = \left[\frac{\partial \Phi}{\partial x}(x_k, u)\right]^\intercal \bar{X}_{k+1} + \left[\frac{\partial \Psi}{\partial x}(x_k, u)\right]^\intercal \bar{Q}$
  $\bar{U}_k = \left[\frac{\partial \Phi}{\partial u}(x_k, u)\right]^\intercal \bar{X}_{k+1} + \left[\frac{\partial \Psi}{\partial u}(x_k, u)\right]^\intercal \bar{Q} + \bar{U}_{k+1}$
end for
return $(x_N, q_N, \bar{X}_0, \bar{U}_0)$

In contrast to the forward mode case, Algorithm 3.7 does *not* have the same structure as Algorithm 3.5, due to the inclusion of the backward sweep.

Implicit one-step integration methods

We can extend Algorithm 3.5 to include an algebraic variable $z_k \in \mathbb{R}^{n_z}$ for each discrete time point $k = 0, \ldots, N-1$. We let it be implicitly defined by the equation $0 = \Theta(x_k, z_k, u)$ as shown in Algorithm 3.8.

Algorithm 3.8 Definition (implicit case) of $F : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q}$

input (x_0, u)
$q_0 = 0$
for $k = 0, \ldots, N-1$ **do**
  $0 = \Theta(x_k, z_k, u)$ ▷ Solve for z_k
  $x_{k+1} = \Phi(x_k, z_k, u)$
  $q_{k+1} = \Psi(x_k, z_k, u) + q_k$
end for
return (x_N, q_N)

Problems in this form appear e.g. as the result of implicit Runge-Kutta schemes for ODEs or DAEs as noted in Section 3.11 below.

By using the derivative propagation rules for the implicit functions as described in Section 3.9, the forward propagation rule defined in Algorithm 3.6 extends to:

Algorithm 3.9 Definition (implicit case) of
$\hat{F} : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_u \times m} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_q \times m}$

input $(x_0, u, \hat{X}_0, \hat{U})$
$q_0 = 0$
$\hat{Q}_0 = 0$
for $k = 0, \ldots, N-1$ **do**
  $0 = \Theta(x_k, z_k, u)$ ▷ Solve for z_k
  $0 = \frac{\partial \Theta}{\partial x}(x_k, z_k, u)\, \hat{X}_k + \frac{\partial \Theta}{\partial z}(x_k, z_k, u)\, \hat{Z}_k + \frac{\partial \Theta}{\partial u}(x_k, z_k, u)\, \hat{U}$ ▷ Solve for $\hat{Z}_k$
  $x_{k+1} = \Phi(x_k, z_k, u)$
  $\hat{X}_{k+1} = \frac{\partial \Phi}{\partial x}(x_k, z_k, u)\, \hat{X}_k + \frac{\partial \Phi}{\partial z}(x_k, z_k, u)\, \hat{Z}_k + \frac{\partial \Phi}{\partial u}(x_k, z_k, u)\, \hat{U}$
  $q_{k+1} = \Psi(x_k, z_k, u) + q_k$
  $\hat{Q}_{k+1} = \frac{\partial \Psi}{\partial x}(x_k, z_k, u)\, \hat{X}_k + \frac{\partial \Psi}{\partial z}(x_k, z_k, u)\, \hat{Z}_k + \frac{\partial \Psi}{\partial u}(x_k, z_k, u)\, \hat{U} + \hat{Q}_k$
end for
return $(x_N, q_N, \hat{X}_N, \hat{Q}_N)$

We note that the introduced help variables $\hat{Z}_0, \ldots, \hat{Z}_{N-1}$ are uniquely defined if the implicit function is well-posed, which in particular implies that $\frac{\partial \Theta}{\partial z}$ is nonsingular.

The corresponding adjoint derivative propagation rule becomes:

Algorithm 3.10 Definition (implicit case) of
$\bar{F} : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_q \times m} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_u \times m}$

input $(x_0, u, \bar{X}_N, \bar{Q})$
$q_0 = 0$
for $k = 0, \ldots, N-1$ **do**
  $0 = \Theta(x_k, z_k, u)$ ▷ Solve for z_k
  $x_{k+1} = \Phi(x_k, z_k, u)$
  $q_{k+1} = \Psi(x_k, z_k, u) + q_k$
end for
$\bar{U}_N = 0$
for $k = N-1, \ldots, 0$ **do**
  $0 = \left[\frac{\partial \Phi}{\partial z}(x_k, z_k, u)\right]^{\intercal} \bar{X}_{k+1} + \left[\frac{\partial \Theta}{\partial z}(x_k, z_k, u)\right]^{\intercal} \bar{Z}_k + \left[\frac{\partial \Psi}{\partial z}(x_k, z_k, u)\right]^{\intercal} \bar{Q}$
  ▷ Solve for $\bar{Z}_k$
  $\bar{X}_k = \left[\frac{\partial \Phi}{\partial x}(x_k, z_k, u)\right]^{\intercal} \bar{X}_{k+1} + \left[\frac{\partial \Theta}{\partial x}(x_k, z_k, u)\right]^{\intercal} \bar{Z}_k + \left[\frac{\partial \Psi}{\partial x}(x_k, z_k, u)\right]^{\intercal} \bar{Q}$
  $\bar{U}_k = \left[\frac{\partial \Phi}{\partial u}(x_k, z_k, u)\right]^{\intercal} \bar{X}_{k+1} + \left[\frac{\partial \Theta}{\partial u}(x_k, z_k, u)\right]^{\intercal} \bar{Z}_k + \left[\frac{\partial \Psi}{\partial u}(x_k, z_k, u)\right]^{\intercal} \bar{Q}$
    $+\bar{U}_{k+1}$
end for
return $(x_N, q_N, \bar{X}_0, \bar{U}_0)$

As in the explicit case, we see that the implicit one-step integrator retains its structure under forward mode, but not in reverse mode differentiation.

3.11 Embedded continuous-time integrators

We now proceed with the continuous-time case. In the following, we consider parametric IVP in *ordinary differential equations* (ODE),

$$
\begin{aligned}
& f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q}, \quad (x_0, u) \mapsto (x_1, q_1) \\
& \left\{ \begin{array}{rcll} \dot{x}(t) & = & \phi(x(t), u) & \\ \dot{q}(t) & = & \psi(x(t), u) & \end{array} \right. \qquad t \in [0, 1], \\
& x(0) = x_0, \quad q(0) = 0, \quad x_1 = x(1), \quad q_1 = q(1),
\end{aligned}
\tag{3.26}
$$

or, more generally, in (semi-explicit) *differential-algebraic equations* (DAE),

$$\begin{aligned} &f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q}, \quad (x_0, u) \mapsto (x_1, q_1) \\ &\begin{cases} \dot{x}(t) &= \phi(x(t), z(t), u) \\ 0 &= \theta(x(t), z(t), u) \qquad t \in [0, 1], \\ \dot{q}(t) &= \psi(x(t), z(t), u) \end{cases} \\ &x(0) = x_0, \quad q(0) = 0, \quad x_1 = x(1), \quad q_1 = q(1), \end{aligned} \tag{3.27}$$

where $x(\cdot) \in \mathbb{R}^{n_x}$ is the (differential) state, $z(\cdot) \in \mathbb{R}^{n_z}$ is the algebraic variable, $u \in \mathbb{R}^{n_u}$ is the parameter and $q(\cdot) \in \mathbb{R}^{n_q}$ is what we shall call the *quadrature state*, i.e. a differential state that does not enter in the ODE/DAE right hand side. A solver for (3.26) or (3.27) will be referred to as an *integrator*. The process of calculating derivatives of an integrator will be referred to as *sensitivity analysis*.

We shall assume that the functions ϕ, θ and ψ are sufficiently smooth and that $\frac{\partial \theta}{\partial z}$ is invertible, which in particular means the algebraic variable z is implicitly defined by the algebraic equation $0 = \theta(x(t), z(t), u)$. We also assume that the integration take place over the unit interval $[0, 1]$. A time transformation can be performed if this is not the case.

Numerical solution

The numerical solution to problems of the form (3.26) and (3.27) has been studied since the advent of the electronic computer. For a comprehensive overview of methods, we refer to Hairer, Nørsett and Wanner [107,108]. Popular solution methods, in the context of optimal control, include (explicit) *Runge-Kutta* (RK) methods and implicit methods such as *backward differentiation formulas* (BDF) and *implicit Runge-Kutta* (IRK). Popular software tools include `ode45`, `ode15s`, `ode15i` (and related solvers) in MATLAB, the integrators in the GNU Scientific Library (GSL) [83], the DASSL code by Petzold [154] as well as the CVODES and IDAS integrators from the SUNDIALS suite [113].

The AD approach to sensitivity analysis

In Section 2.6, we used a collocation scheme to eliminate the integrators from the multiple shooting NLP formulation, and delegated the solution of this integration problem to the NLP solver. But, as noted in that section, the collocation equations can also be used to eliminate the state trajectory from

the NLP optimization variables. That defines a *fixed-step implicit Runge-Kutta integrator*, which takes the shape of the implicit discrete-time integrator in Algorithm 3.8. The function Θ in Algorithm 3.8 will then contain the collocation equations (2.24) for ODEs or (2.31) for DAEs. The now affine functions Φ and Ψ in Algorithm 3.8 will contain the continuity equations (2.25) and the quadratures (2.27), respectively.

The same discretization approach can also be performed with other one-step integrator schemes resulting in discrete-time problems either of the form of Algorithm 3.5, for explicit integrator schemes, or of Algorithm 3.8 for implicit integrator schemes. Sensitivity analysis can then be performed as described in Section 3.10.

We shall refer to this way of calculating derivatives of an ODE or DAE integrator as the *AD approach to sensitivity analysis*. The approach is also known as *internal numerical differentiation* (IND) [21,31,49] and the traditional way to implement it is to use a black box AD tool to differentiate an existing code for the IVP solution, but more sophisticated approaches also exist [22]. The term IND is in particular often used when derivatives of the underlying DAE right hand side are approximated using finite differences. A simpler approach, where finite differences is applied two the whole IVP solver, including step size selection and Newton algorithm for implicit schemes, is known as *external numerical differentiation* (END) [21]. Since the AD approach attempts to calculate the exact derivative of an approximate solution to the IVP, it can be thought of as an "integrate-then-differentiate" approach.

The variational approach to sensitivity analysis

The converse approach, i.e. "differentiate-then-integrate", is the *variational approach to sensitivity analysis*. It starts by differentiating the differential equations and uses calculus of variations to form a new IVP (in the forward case) or a two-point boundary value-problem (in the adjoint case), whose solution gives the derivatives. In the following, we shall informally derive these equations from the discrete-time case in Section 3.10. For a more formal treatment, we refer to Maly and Petzold [145] and Cao el al. [57].

The forward sensitivity equations

By defining $\Phi(x,u) := x + \frac{1}{N}\,\phi(x,u)$ and $\Psi(x,u) := \frac{1}{N}\,\psi(x,u)$, we can interpret Algorithm 3.5 as an explicit Euler method for solving (3.26). Similarly, we can interpret Algorithm 3.5 as an explicit Euler method for solving the following

augmented IVP with $(1+m)$ times as many states as the original problem:

$$
\begin{aligned}
&\hat{f} : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_u \times m} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_q \times m}, \\
&\quad (x_0, u, \hat{X}_0, \hat{U}) \mapsto (x_1, q_1, \hat{X}_1, \hat{Q}_1) \\
&\begin{cases}
\dot{x}(t) &= \phi(x(t), u) \\
\dot{q}(t) &= \psi(x(t), u) \\
\dot{\hat{X}}(t) &= \frac{\partial \phi}{\partial x}(x(t), u)\, \hat{X}(t) + \frac{\partial \phi}{\partial u}(x(t), u)\, \hat{U} \\
\dot{\hat{Q}}(t) &= \frac{\partial \psi}{\partial x}(x(t), u)\, \hat{X}(t) + \frac{\partial \psi}{\partial u}(x(t), u)\, \hat{U}
\end{cases} \qquad t \in [0, 1], \\
&x(0) = x_0, \quad q(0) = 0, \quad x_1 = x(1), \quad q_1 = q(1), \\
&\hat{X}(0) = \hat{x}_0, \quad \hat{Q}(0) = 0, \quad \hat{X}_1 = \hat{X}(1), \quad \hat{Q}_1 = \hat{Q}(1),
\end{aligned}
\tag{3.28}
$$

Since the approximations will be exact as N tends to infinity, we can use a solver for (3.28) for calculating the forward sensitivities for (3.26).

The approach naturally extends to DAEs in semi-explicit form, by defining $\Phi(x, z, u) := x + \frac{1}{N}\phi(x, z, u)$, $\Theta(x, z, u) := \theta(x, z, u)$ and $\Psi(x, z, u) := \frac{1}{N}\psi(x, z, u)$ in Algorithms 3.8 and 3.9. For a more formal treatment as well as an overview of methods to efficiently solve (3.28), we refer to Hindmarsh et al. [113] and references therein.

The adjoint sensitivity equations

Equivalently, using Algorithm 3.7 and with the same definitions of Φ and Ψ, we obtain the following problem, whose solution gives the reverse mode derivative propagation rule:

$$
\begin{aligned}
&\bar{f} : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_q \times m} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_u \times m}, \\
&\quad (x_0, u, \bar{X}_1, \bar{Q}) \mapsto (x_1, q_1, \bar{X}_0, \bar{U}_0) \\
&\begin{cases}
\dot{x}(t) &= \phi(x(t), u) \\
\dot{q}(t) &= \psi(x(t), u) \\
-\dot{\bar{X}}(t) &= \left[\frac{\partial \phi}{\partial x}(x(t), u)\right]^\intercal \bar{X}(t) + \left[\frac{\partial \psi}{\partial x}(x(t), u)\right]^\intercal \bar{Q} \\
-\dot{\bar{U}}(t) &= \left[\frac{\partial \phi}{\partial u}(x(t), u)\right]^\intercal \bar{X}(t) + \left[\frac{\partial \psi}{\partial u}(x(t), u)\right]^\intercal \bar{Q}
\end{cases} \qquad t \in [0, 1], \\
&x(0) = x_0, \quad q(0) = 0, \quad x_1 = x(1), \quad q_1 = q(1), \\
&\bar{X}_1(1) = \bar{X}_1, \quad \bar{U}(1) = 0, \quad \bar{X}_0 = \bar{X}(0), \quad \bar{U}_0 = \bar{U}(0),
\end{aligned}
\tag{3.29}
$$

This is a two-point boundary-value problem with initial conditions on x and q as well as terminal constraints on $\bar{X}$ and $\bar{U}$. It can be solved by a forward integration storing the trajectory of x, followed by an integration backwards in time. Again, the approach naturally extends to DAEs in semi-explicit form. For a more formal treatment and in particular a discussion on how to limit the memory associated with storing the trajectory of x, we again refer to Hindmarsh et al. [113] and references therein.

3.12 Conclusion

Efficiently and accurately calculating derivative information is of paramount importance in numerical optimization and algorithmic differentiation (AD) is the method of choice for this.

We showed that AD, most commonly implemented for algorithms made up by scalar-valued operations, extends readily to algorithms where the operations are matrix-valued. We also showed how to handle constructs relevant for optimal such as implicitly defined functions and ODE/DAE integrators in the context of AD.

Chapter 4

Structure exploiting nonlinear programming

We have seen in the previous chapter how optimal control problems can be reformulated to either root-finding problems of the form:

$$g(x) = 0, \tag{RFP}$$

or nonlinear programs (NLPs) of the form:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g(x) = 0, \quad \underline{x} \leq x \leq \overline{x}, \quad \underline{h} \leq h(x) \leq \overline{h}. \end{aligned} \tag{NLP}$$

If not stated otherwise, we shall assume $f : \mathbb{R}^{n_x} \to \mathbb{R}$, $g : \mathbb{R}^{n_x} \to \mathbb{R}^{n_g}$ and $h : \mathbb{R}^{n_x} \to \mathbb{R}^{n_h}$ to be twice continuously differentiable in x and allow the bounds ($\underline{x}$, $\overline{x}$, $\underline{h}$ and $\overline{h}$) to take infinite values.

In this chapter, we provide some essential definitions and review the necessary conditions for optimality of (NLP) in Section 4.1. We also give a brief introduction into the two most widespread numerical methods for solving it; sequential quadratic programming in Section 4.2 and interior point methods in Section 4.3. A more thorough introduction to nonlinear programming can be found e.g. in the recent textbooks by Nocedal and Wright [151], Biegler [43] and Bertsekas [37].

We also present the *lifted Newton method*, a structure-exploiting variant of the classical Newton's method for solving (RFP) and (NLP) in Section 4.5. We

end the chapter with a brief overview of existing numerical tools for NLP in Section 4.6.

4.1 Essentials

We shall use the formulation (NLP) to denote an NLP throughout this text, which corresponds well to the form expected by numerical tools. We point out that other formulations, in particular without the lower bounds on h, are more suitable to discuss important properties such as convexity (see e.g. [52]).

The following is a set of definitions for (NLP):

Definition 4.1 (Feasibility)**.** *A point $x \in \mathbb{R}^{n_x}$ is called a* feasible point *for (NLP) if it satisfies:*

$$g(x) = 0, \quad \underline{x} \leq x \leq \overline{x}, \quad \underline{h} \leq h(x) \leq \overline{h},$$

The set of all feasible points is the feasible set*:*

$$\mathcal{F} := \{x \in \mathbb{R}^{n_x} : x \text{ feasible}\}$$

Definition 4.2 (Optimality)**.** *A point $x^* \in \mathbb{R}^{n_x}$ is called* globally optimal *for (NLP) if it is feasible and satisfies:*

$$f(x^*) \leq f(x) \quad \forall x \in \mathcal{F}$$

A point x^* is called locally optimal *if there exists an open neighborhood $\mathcal{N}$ around x^* in which the point is globally optimal:*

$$f(x^*) \leq f(x) \quad \forall x \in \mathcal{F} \cap \mathcal{N}$$

If not stated otherwise, the term *optimal* will be used in the sense locally optimal.

Definition 4.3 (Lagrangian function, Lagrange multipliers)**.** *The* Lagrangian function *is defined by:*

$$\mathcal{L}(x, \lambda_g, \lambda_x, \lambda_h) := f(x) + \lambda_g^T\, g(x) + \lambda_x^T\, x + \lambda_h^T\, h(x), \tag{4.1}$$

where λ_g, λ_x and λ_h are the Lagrange multipliers *corresponding to the constraints $g(x) = 0$, $\underline{x} \leq x \leq \overline{x}$ and $\underline{h} \leq h(x) \leq \overline{h}$ respectively.*

Definition 4.4 (Active constraint)**.** *A bound constraint will be called* inactive *if both the upper and lower bound hold with strict inequality. Otherwise it is*

called active. *An active constraint is called* weakly active *if the corresponding Lagrange multiplier is zero. We denote by* $\mathcal{A}_x$ *and* $\mathcal{A}_h$ *the index sets corresponding to active constraints and use* $x_{\mathcal{A}}$ *and* $h_{\mathcal{A}}$ *to denote vectors with the corresponding elements of* x *and* h.

Definition 4.5 (Regularity)**.** *A point is called* regular *if it is feasible and the matrix:*

$$\begin{bmatrix} \dfrac{\partial g}{\partial x} \\ \dfrac{\partial h_{\mathcal{A}}}{\partial x} \end{bmatrix}$$

has full row rank. We say that the linear independence constraint qualification *(LICQ) holds.*

The Karush-Kuhn-Tucker conditions

The following theorem, due to Karush, Kuhn and Tucker [119, 127], is paramount in numerical optimization and states the necessary conditions for optimality of (NLP):

Theorem 4.1 (The KKT conditions)**.** *If* x^* *is regular and a locally optimal point of (NLP), then a solution* $(\lambda_g^*, \lambda_x^*, \lambda_h^*)$ *exists to the set of equations:*

$$\nabla_x \mathcal{L}(x^*, \lambda_g, \lambda_x, \lambda_h) = 0, \tag{4.2a}$$

$$g(x^*) = 0, \quad \underline{x} \leq x^* \leq \overline{x}, \quad \underline{h} \leq h(x^*) \leq \overline{h}, \tag{4.2b}$$

$$(\lambda_x^*)_i \begin{cases} \leq 0 & \text{if the } i\text{-th lower bound of } x \text{ is active} \\ \geq 0 & \text{if the } i\text{-th upper bound of } x \text{ is active} \\ = 0 & \text{otherwise} \end{cases} \tag{4.2c}$$

$$(\lambda_h^*)_i \begin{cases} \leq 0 & \text{if the } i\text{-th lower bound of } h(x) \text{ is active} \\ \geq 0 & \text{if the } i\text{-th upper bound of } h(x) \text{ is active} \\ = 0 & \text{otherwise} \end{cases} \tag{4.2d}$$

If there are no weakly active constraints at the solution, we say that strict complementarity *holds and the solution is unique.*

Proof. By adapting the proof found in e.g. [151] to problems of the form (*NLP*). □

We say that x^* is the *primal* solution and that $(\lambda_g^*, \lambda_x^*, \lambda_h^*)$ is the *dual solution* to (NLP).

Parametric NLPs

Definition 4.6 (Parametric NLP, parametric sensitivity)**.** *An NLP, where a subset of the variables are equality constrained, i.e., can be written* $(\underline{x}, \overline{p}) \leq (x, p) \leq (\overline{x}, \overline{p})$*, is called a* parametric NLP*, and we shall write it in the form:*

$$\begin{array}{ll} \textit{minimize} \\ x, p & f(x,p) \\ \textit{subject to} & g(x,p) = 0, \quad p = \overline{p}, \quad \underline{x} \leq x \leq \overline{x}, \quad \underline{h} \leq h(x,p) \leq \overline{h}, \end{array} \tag{4.3}$$

The Lagrange multiplier λ_p^* *corresponding to* $p = \overline{p}$ *at the optimal solution is the* parametric sensitivity *of (4.3).*

Corollary 4.1 (Calculation of parametric sensitivity)**.** *If* $(x^*, \lambda_g^*, \lambda_h^*)$ *belong to a primal-dual solution to (4.3), then*

$$-\lambda_p^* = \nabla_p f + \left(\frac{\partial g}{\partial p}\right)^\intercal \lambda_g^* + \left(\frac{\partial h}{\partial p}\right)^\intercal \lambda_h^* \tag{4.4}$$

Proof. Follows directly from the KKT conditions and the definition of the Lagrangian. □

4.2 Sequential quadratic programming

In *sequential quadratic programming* (SQP) methods, the KKT-conditions are linearized at a current primal (x) and dual (λ_g, λ_h) solution guess. This gives rise to a *quadratic program* (QP), which is typically required to be convex. The solution of this QP gives a new iterate or a search direction for a new iterate. SQP was originally proposed by Wilson [190] but reached widespread popularity mainly through the work of Han [109, 110] and Powell [156]. For a comprehensive introduction into SQP, we refer to the standard textbooks for nonlinear programming. For solving the large and sparse NLPs arising in optimal control, two major approaches exist to solve the arising QP subproblems:

General-purpose, sparsity-exploiting QP solvers This includes solvers such as CPLEX [118] or the open-source OOQP [87]. These solvers are

usually *interior-point methods* (cf. [151]) that rely on direct methods for sparse linear systems (cf. [68]). This has proven successful in the context of direct collocation methods (Section 2.6).

Special-purpose, structure-exploiting QP solvers These solvers can be either interior-point or *active-set methods* (cf. [151]). This has proven successful in the context of direct multiple shooting methods (Section 2.5). A comprehensive treatment of this topic can be be found in the dissertations of Leineweber [131] and Kirches [122]. QP solvers written to exploit the block structure arising from multiple shooting discretization include qpOASES [75] (in combination with *condensing*, cf. [131]), FORCES [72] and qpDUNES [81].

Constrained Gauss-Newton method

A common class of NLPs is constrained least squares problems, i.e., problems with a least squares objective function:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \tfrac{1}{2}\left\|F(x)\right\|_2^2 \\ \text{subject to} \quad & g(x) = 0, \quad \underline{x} \leq x \leq \overline{x}, \quad \underline{h} \leq h(x) \leq \overline{h}. \end{aligned} \tag{4.5}$$

For many such problems, the Hessian of the Lagrangian function (5.7) is well approximated by the *Gauss-Newton* (GN) Hessian, i.e.

$$\nabla^2 \mathcal{L}(x, \lambda_g, \lambda_h, \lambda_x) \approx \left(\frac{\partial F}{\partial x}\right)^\intercal \left(\frac{\partial F}{\partial x}\right). \tag{4.6}$$

If this approximation is used in a SQP method, the quadratic nature (4.6) will ensure that the resulting QP is convex, as required by most QP-solvers. Furthermore, since the right hand side of (4.6) does not depend on the Lagrange multipliers, they can be left out of the SQP method (although they can still be useful to assess optimality and as a stopping criterion). This gives the basis of the *constrained Gauss-Newton method* [151].

Sequential convex programming

A generalization of SQP is to allow the subproblem to be a more generic *convex* program such as a *quadratically constrained quadratic program* (QCQP), *second-order cone program* (SOC) or a *semidefinite program* (SDP) (see [52]). This results in a *sequential convex programming* (SCP) method.

As an example, in the orbital transfer example from Section 2.1, an SQP method would need to linearize the constraint on the control:

$$u_1(t)^2 + u_2(t)^2 \leq T_{\max}^2, \quad t \in [0, T], \tag{2.2}$$

which may result in a degenerate QP if e.g. $u_1(t) = u_2(t) = 0$ in some region. If (2.2) is left untouched and only the dynamic equation (2.1) linearized, the subproblem becomes a (convex) QCQP.

For an overview of SCP method, especially in the context of optimal control, we refer to the dissertation of Tran-Dinh [169] and references therein.

4.3 Interior point methods for NLP

In interior point (IP) methods for NLP, which have evolved out of IP methods for linear (LP) and quadratic programming (QP), the inequality constrained (NLP) is replaced with a sequence of equality constrained problems. In their implementation, interior point methods typically eliminate constraints of the form $\underline{h} \leq h(x) \leq \overline{h}$ by introducing slack variables and additional equality constraints. After locating an *interior point*, i.e., a point where $\underline{x} < x < \overline{x}$ holds with strict inequality, which may require reformulating some decision variables as parameters, they formulate a *barrier problem* of the form:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) - \mu \sum_{i=1}^{n_x} (\log(\overline{x}_i - x_i) + \log(x_i - \underline{x}_i)) \\ \text{subject to} \quad & g(x) = 0. \end{aligned} \tag{4.7}$$

Sequences of barrier problems for increasing values of the *barrier parameter* μ are then solved with Newton-type methods. Using sparsity exploiting linear algebra (similar to the IP methods for LP or QP), the IP method for NLP has proven very successful in the context of direct collocation methods (Section 2.6), much thanks to the open-source implementation IPOPT [179]. For more details on the implementation of IP methods for nonlinear programming we refer to recent NLP textbooks as well as Wächter and Biegler's paper on the implementation of IPOPT [179].

4.4 Globalization techniques

Neither SQP nor IP methods are guaranteed to converge to a local minimum from an arbitrary starting point. To get global convergence, they need to be

combined with a globalization strategy such as *line-search* or *trust-region*. For details on this we refer to [64, 151].

More recently, Fletcher and Leyffer [78], proposed *filter methods*, as a less conservative version of the original trust-region method. Wächter and Biegler [180, 181] then adopted this approach to a filter line-search method which did not suffer from the problem of selecting a penalty parameter in the merit function.

4.5 The lifted Newton method

While general-purpose sparsity-exploiting linear algebra has proven very successful for collocation methods, both in SQP and IP settings, state-of-the-art implementations of direct multiple shooting have relied on customized, structure-exploiting linear algebra such as the QP condensing technique by Bock and Plitt [51,131]. Using this technique, the size of the QP can be reduced to the size of a QP arising from direct single shooting (Section 2.4). We shall return to this approach in Chapter 5, where we propose a novel implementation of QP condensing.

Albersmeyer and Diehl [23] later generalized what is known as *Schlöder's trick* [164] in condensing to a general-purpose, structure-exploiting technique called the *lifted Newton method*.

In its most basic form, the lifted Newton method can be viewed as a way to efficiently *lift* a root-finding problem formulation that contains a series of intermediate expressions, to a less nonlinear problem of higher dimension. Under certain conditions, this is known to have advantages both in terms of local and global convergence as explored in the original paper [23] and references therein. An equivalent viewpoint, which is closer to the QP condensing approach and which we shall employ here, is to view it as a structure-exploiting solution method for root-finding problems with the structure:

$$g(x) = \begin{pmatrix} \phi_1(x_0) & - & x_1 \\ \phi_2(x_0, x_1) & - & x_2 \\ \vdots & & \\ \phi_{N-1}(x_0, x_1, \ldots, x_{N-2}) & - & x_{N-1} \\ \phi_N(x_0, x_1, \ldots, x_{N-1}) & & \end{pmatrix} = 0, \tag{4.8}$$

where x is partitioned into $x_i \in \mathbb{R}^{n_i}$, $i = 0, \ldots, N-1$.

Note that this structure includes the root-finding problem arising in indirect multiple shooting (2.13). Furthermore, this structure is common in the DAE

simulation community and arises when so-called *tearing* strategies are applied to break up large and sparse nonlinear systems of equations into a sequence of smaller linear or nonlinear systems of equations. For more details on this and their implementation in Modelica-based tools, we refer to Cellier and Kofman [59] or Fritzson [82].

In the following section, we shall derive the method in the context of sequential quadratic programming. We refer to the original paper [23] for a description of the method for solving root-finding problems of the form (4.8).

Lifted sequential quadratic programming

We consider structured NLPs of the form:

$$\begin{aligned}
\underset{x}{\text{minimize}} \quad & f(x) \\
\text{subject to} \quad & \phi_1(x_0) - x_1 = 0, \\
& \phi_2(x_0, x_1) - x_2 = 0, \\
& \quad \vdots \\
& \phi_N(x_0, \ldots, x_{N-1}) - x_N = 0, \\
& \\
& g(x) = 0, \\
& \underline{x} \leq x \leq \overline{x}, \\
& \underline{h} \leq h(x) \leq \overline{h},
\end{aligned} \tag{4.9}$$

where x has been decomposed into $(x_0, \ldots, x_N)$. This structure arises in direct multiple shooting (2.18) and, with $\phi_i(\cdot)$ defined implicitly, in direct collocation (2.28).

To simplify the presentation, we shall assume that the objective is linear and that $g(x)$ and $h(x)$ are absent. We also introduce a dummy variable defined by $u := x_0$:

$$\begin{aligned}
\underset{u,\, x}{\text{minimize}} \quad & c^\intercal x \\
\text{subject to} \quad & u - x_0 = 0, \\
& \phi_1(x_0) - x_1 = 0, \\
& \phi_2(x_0, x_1) - x_2 = 0, \\
& \quad \vdots \\
& \phi_N(x_0, \ldots, x_{N-1}) - x_N = 0, \\
& \\
& \underline{x} \leq x \leq \overline{x},
\end{aligned} \tag{4.10}$$

It is always possible to reformulate (4.9) into (4.10) by introducing slack variables and allowing the lower bounds $\underline{x}$ and upper bounds $\overline{x}$ to be equal for a subset of the variables x.

We define the Lagrangian of (4.10) to be:

$$\mathcal{L}(u, x, \lambda_x, \nu) = c^\intercal x + \lambda_x^\intercal x + \nu_0 (u - x_0) + \sum_{j=1}^{N} \nu_j^\intercal \left(\phi_j(x_0, \ldots, x_{j-1}) - x_j\right), \quad (4.11)$$

where we have introduced the multipliers λ_x corresponding to the bound constraint $\underline{x} \leq x \leq \overline{x}$ and $\nu = (\nu_0, \ldots, \nu_N)$ corresponding to the equality constraints $u - x_0 = 0$ and $\phi_j(x_0, \ldots, x_{j-1}) - x_j = 0$ for $j = 1, \ldots, N$.

In each iteration of an (exact Hessian, full step) SQP method to solve (4.10), with $(u^{(k)}, x^{(k)}, \lambda_x^{(k)}, \nu^{(k)})$ being the current primal-dual solution guess, the solution guess for the next iteration $(u^{(k+1)}, x^{(k+1)}, \lambda_x^{(k+1)}, \nu^{(k+1)})$ is given from solving the QP:

$$\begin{aligned} \underset{\Delta u, \Delta x}{\text{minimize}} \quad & \frac{1}{2} \Delta x^\intercal H \Delta x + h^\intercal \Delta x \\ \text{subject to} \quad & G \Delta u - L \Delta x = -g, \\ & \underline{x} - x^{(k)} \leq \Delta x \leq \overline{x} - x^{(k)}, \end{aligned} \quad (4.12)$$

where we have defined $\Delta u := u^{(k+1)} - u^{(k)}$, $\Delta x := x^{(k+1)} - x^{(k)}$, $H := \nabla_x^2 \mathcal{L}(u^{(k)}, x^{(k)}, \nu^{(k)})$, $g := (\phi_1(x^{(k)}) - x_1^{(k)}, \ldots, \phi_N(x^{(k)}) - x_N^{(k)})$ as well as:

$$L := \begin{bmatrix} I & & & \\ -\frac{\partial \phi_1}{\partial x_0} & I & & \\ \vdots & \ddots & \ddots & \\ -\frac{\partial \phi_N}{\partial x_0} & \cdots & -\frac{\partial \phi_N}{\partial x_{N-1}} & I \end{bmatrix}, \quad G := \begin{bmatrix} I \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \quad (4.13)$$

Since L is invertible, we can write Δx in terms of Δu:

$$\Delta x = L^{-1} (G \Delta u + g) := A \Delta u + a. \quad (4.14)$$

This allows us to eliminate Δx from (4.12) giving a QP in only Δu:

$$\begin{aligned} \underset{\Delta u}{\text{minimize}} \quad & \frac{1}{2} \Delta u^\intercal B \Delta u + b^\intercal \Delta u \\ \text{subject to} \quad & \underline{x} - x^{(k)} - a \leq A \Delta u \leq \overline{x} - x^{(k)} - a, \end{aligned} \quad (4.15)$$

where

$$\begin{aligned} A &= L^{-1}\, G, \qquad & B &= G^\intercal\, L^{-\intercal}\, H\, A, \\ a &= L^{-1}\, g, & b &= G^\intercal\, L^{-\intercal}\, (H\, a + c), \end{aligned} \tag{4.16}$$

We can calculate a in (4.16) using a forward substitution for the lower triangular matrix L defining Algorithm 4.1:

Algorithm 4.1 Calculation of a in (4.16)

$a_0 \leftarrow g_0$
for $i = 1, \ldots, N$ **do**
$$a_i \leftarrow g_i + \sum_{j=0}^{k-1} \left(\frac{\partial \phi_i}{\partial x_j}(x^{(k)}) \right) a_j$$
end for

Algorithm 4.1 can be interpreted as a forward directional derivative:

$$a = \left(\frac{\partial z}{\partial d}(u^{(k)}, g) \right) g, \tag{4.17}$$

where the function $z(u, d)$ is defined by Algorithm 4.2:

Algorithm 4.2 Modified function $z(u, d)$

$z_0 \leftarrow u - d_0$
$z_1 \leftarrow \phi_1(z_0) - d_1$
$\vdots$
$z_N \leftarrow \phi_N(z_0, \ldots, z_{N-1}) - d_N$

With a in (4.16) given, $H\, a$ in (4.16) can be calculated efficiently as a forward-over-adjoint directional derivative:

$$w := H\, a = (\nabla_x^2 \mathcal{L}(u, x, \lambda_x, \nu))\, a = (\nabla_x^2 r(x))\, a, \tag{4.18}$$

where the scalar function $r(x)$ consists of the non-vanishing terms in the Lagrangian:

$$r(x) = \sum_{j=1}^{N} \nu_j^\intercal\, \phi_j(x_0, \ldots, x_{j-1}). \tag{4.19}$$

This allows us to calculate b in in (4.16) using a backward substitution for $L^\intercal$ as shown in Algorithm 4.3:

Algorithm 4.3 Calculation of b in (4.16)

$\tilde{b}_N \leftarrow w_N + c_N$
for $i = N, \ldots, 0$ **do**
$$\tilde{b}_i \leftarrow w_i + c_i + \sum_{j=0}^{i-1} \left(\frac{\partial \phi_j}{\partial x_i}(x^{(k)}) \right)^\intercal \tilde{b}_j$$
end for
$b \leftarrow \tilde{b}_0$

We can identify this algorithm as the calculation of an adjoint directional derivative of $z(u, d)$, giving the following expression for b:

$$b = \left(\frac{\partial z}{\partial u}(u^{(k)}, g) \right)^\intercal (w + c). \tag{4.20}$$

The same approach used to calculate a and b above can also be used to calculate the product of A and B with an arbitrary matrix V, i.e. in turn calculate:

$$A' := A\,V = \left(\frac{\partial z}{\partial d}(u^{(k)}, g) \right) (G\,V), \tag{4.21a}$$

$$W' := H\,A' = (\nabla_x^2 r(x))\,A', \tag{4.21b}$$

$$B' := B\,V = \left(\frac{\partial z}{\partial u}(u^{(k)}, g) \right)^\intercal W'. \tag{4.21c}$$

This can in particular be used to retrieve the complete matrices A and B by setting $V := I$.

With B, b, A and a now given, the reduced space QP (4.15) can be formulated and solved using either a dense QP solver or a matrix-free QP solver, in the latter case relying on (4.21). This gives the step in the reduced variables, $\Delta u^{(k+1)}$ as well as the new multipliers of the inequality constraints $\lambda_x^{(k+1)}$.

We can retrieve the step in the eliminated variables from (4.14):

$$\Delta x = A\,\Delta u + a = \left(\frac{\partial z}{\partial d}(u^{(k)}, g) \right) (G\,\Delta u) + a, \tag{4.22}$$

either by performing the matrix-vector multiplication or by calculating the directional derivative.

The new multipliers for the assignment conditions, $\nu^{(k+1)}$, can be retrieved from noting that at an optimal solution of (4.12), we have $0 = \nabla_x \mathcal{L}(u^{(k+1)}, x^{(k+1)}, \lambda_x^{(k+1)}, \nu^{(k+1)})$ from the KKT conditions. From (4.11):

$$\nabla_x \mathcal{L}(u^{(k+1)}, x^{(k+1)}, \lambda_x^{(k+1)}, \nu^{(k+1)}) = c + \lambda_x^{(k+1)} - L^\intercal \nu^{(k+1)} = 0$$

$$\Leftrightarrow \quad \nu^{(k+1)} = L^{-\intercal}(c + \lambda_x^{(k+1)}) = \left(\frac{\partial z}{\partial d}(u^{(k)}, g)\right)^\intercal (c + \lambda_x^{(k+1)}). \qquad (4.23)$$

For an alternative derivation, we refer to [23].

4.6 Numerical tools

A large number of codes for NLP has been written over the years. A comprehensive survey of SQP-type methods before the year 2000 can be found in Conn, Gould and Toint [64]. Early SQP codes include the original LANCELOT code [65], SNOPT [91, 92] and NPSOL [93]. Early, but still relevant, IP codes are LOQO [173] and KNITRO (both IP and SQP) [56]. filterSQP implements the original filter trust region method [78]. The open-source IPOPT [179], originally in Fortran by Wächter and later reimplemented in C++ by Laird, implements Wächter and Biegler's filter line-search method. LANCELOT-B, which is free for academic use, is a reimplementation of LANCELOT, and implements a trust region SQP method with exploitation of partial separability [97]. WORHP [55], also free for academic use, is a large-scale line-search SQP code. The open-source ALGENCAN solver [29] is a matrix-free augmented Lagrangian type method. An implementation of the lifted Newton method can be found in LiftOpt [23].

4.7 Conclusion

Direct methods for optimal control require the solution of structured nonlinear programs (NLPs). In this chapter, we reviewed the two major approaches for solving NLPs, namely sequential quadratic programming (SQP) and interior point (IP) methods. We also presented the lifted Newton method in the context of SQP.

The lifted Newton method is able to efficiently address large and block sparse NLPs arrising from direct multiple shooting discretization. These NLPs are often not sparse enough to be handled efficiently by general-purpose sparsity-exploiting NLP solvers. We presented a novel way of deriving the method.

Chapter 5

A Condensing Algorithm for Nonlinear MPC with a Quadratic Runtime in Horizon Length

The following chapter presents a novel algorithm for QP condensing. *The text was prepared together with Janick Frasch, Milan Vukov and Moritz Diehl and has been submitted to the Automatica journal. It is presented in the following in mostly unaltered form.*

A large number of practical algorithms for optimal control problems (OCP) relies on a so-called *condensing* procedure to exploit the given structure in the quadratic programming (QP) subproblems. While the established structure-exploiting condensing algorithm is of cubic complexity in the horizon length, in this technical note we propose a novel algorithm that is only of quadratic complexity in horizon length. We present numerical results confirming that the proposed algorithm is faster in practice and discuss implications for related QP solution algorithms that rely on the elimination of the state sequence. In particular, we show that it is possible to decrease the runtime complexity from quadratic to linear in the length of time horizon for a class of fast-gradient based algorithms widely used in practice.

5.1 Introduction

In this chapter we consider the solution of the following linear time-varying optimal control problem (LTV-OCP) with quadratic objective, in n_x states and n_u controls on a time horizon of length N:

$$\underset{\substack{u_0,\ldots,u_{N-1}\\ x_1,\ldots,x_N}}{\text{minimize}} \quad \sum_{k=0}^{N-1} \left(\frac{1}{2} u_k^\intercal R_k u_k + x_k^\intercal S_k u_k + u_k^\intercal r_k \right)$$

$$+ \sum_{k=1}^{N} \left(\frac{1}{2} x_k^\intercal Q_k x_k + x_k^\intercal q_k \right) \tag{5.1a}$$

$$\text{subject to} \quad x_{k+1} = A_k x_k + B_k u_k + c_k, \quad k = 0, \ldots, N-1, \tag{5.1b}$$

$$\underline{u}_k \leq u_k \leq \overline{u}_k, \quad k = 0, \ldots, N-1, \tag{5.1c}$$

$$\underline{x}_k \leq x_k \leq \overline{x}_k, \quad k = 1, \ldots, N. \tag{5.1d}$$

For the ease of presentation, we assume a fixed initial value x_0 and only bound constraints, while the algorithms presented in this chapter can be extended in a straightforward way to more general LTV-OCPs. Such LTV-OCP with quadratic objective are a structured form of quadratic programs (QP) and originate from a wide class of problem formulations in optimal control, notably in solution algorithms for linear-quadratic model predictive control (MPC) problems [159], linear moving horizon estimation (MHE) problems [158], as well as their nonlinear counterparts and general nonlinear optimal control problems (OCPs) and dynamic parameter estimation problems solved by sequential quadratic programming (SQP) based algorithms [41, 49, 71, 126].

Interior-point (IP) algorithms can exploit the block-banded structure of LTV QPs well internally [159], but have well-known difficulties in exploiting the similarity between subsequently solved QPs in an SQP or online solution context. Active-set (AS) methods on the other hand can exploit the similarity between subsequently solved QPs very well, but typically do not benefit from the known problem structure to the same extent as IP methods. One way to exploit structure in AS methods is to make use of a so-called *condensing* preprocessing step [49, 51] to eliminate the over-parameterized state sequence from the QP optimization variables. This can lead to drastically reduced problem sizes if the number of states is relatively large compared to the number of controls, which is the case for large classes of control and estimation problems.

Recent comparisons indicate that efficient condensing-based AS methods outperform efficient IP methods on time horizons of short and medium length, but are being outperformed on longer time horizons [178]. This is largely due to the expensive condensing step, which needs to be performed in every iteration for LTV systems originating from the linearization of nonlinear MPC problems. Up to now, the condensing step was believed to be of cubic complexity in the horizon length of the optimization problem [44,61,80,123,131,132], as described in Section 5.2.

The contribution of this chapter is a novel algorithm that is equivalent to the established condensing algorithm but with quadratic rather than cubic complexity in the horizon length.

5.2 QP condensing

The constraints (5.1b) can be written in matrix form:

$$\begin{aligned} A\,x = B\,u + c \Leftrightarrow & \begin{bmatrix} I & & & \\ -A_1 & I & & \\ & \ddots & \ddots & \\ & & -A_{N-1} & I \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \\ = & \begin{bmatrix} B_0 & & & \\ & B_1 & & \\ & & \ddots & \\ & & & B_{N-1} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{N-1} \end{bmatrix} + \begin{bmatrix} A_0\,x_0 + c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix}, \end{aligned} \tag{5.2}$$

giving the following equivalent form of (5.1):

$$\begin{aligned} \underset{x,u}{\text{minimize}} \quad & \frac{1}{2}\,u^\mathsf{T} R\,u + x^\mathsf{T} S\,u + u^\mathsf{T} r + \frac{1}{2}\,x^\mathsf{T} Q\,x + x^\mathsf{T} q \\ \text{subject to} \quad & A\,x = B\,u + c \\ & \underline{u} \le u \le \overline{u} \\ & \underline{x} \le x \le \overline{x}, \end{aligned} \tag{5.3}$$

where, in addition to the quantities in (5.2), we define the vectors $r := (r_0^\mathsf{T}, \ldots, r_{N-1}^\mathsf{T})^\mathsf{T}$, $q := (q_1^\mathsf{T}, \ldots, q_N^\mathsf{T})^\mathsf{T}$, $\underline{x} := (\underline{x}_1^\mathsf{T}, \ldots, \underline{x}_N^\mathsf{T})^\mathsf{T}$, $\overline{x} := (\overline{x}_1^\mathsf{T}, \ldots, \overline{x}_N^\mathsf{T})^\mathsf{T}$ as well as the matrices $R := \operatorname{diag}(R_0, \ldots, R_{N-1})$, $S := \operatorname{diag}(S_0, \ldots, S_{N-1})$ and $Q := \operatorname{diag}(Q_0, \ldots, Q_N)$.

Rather than solving the full space QP (5.1) directly, we seek to solve a so-called *condensed* QP:

$$\begin{aligned} \underset{u}{\text{minimize}} \quad & \frac{1}{2}\, u^\intercal\, H\, u + u^\intercal\, h \\ \text{subject to} \quad & \underline{u} \le u \le \overline{u} \\ & \underline{x} - g \le G\, u \le \overline{x} - g, \end{aligned} \tag{5.4}$$

which is equivalent, but with $n_u\, N$ instead of $(n_u + n_x)\, N$ optimization variables.

Since A is nonsingular, we can use (5.2) to explicitly eliminate x from (5.3), giving the following expressions for g, G, h and H in (5.4):

$$g := A^{-1}\, c, \tag{5.5a}$$

$$G := A^{-1}\, B, \tag{5.5b}$$

$$h := r + G^\intercal\, (q + Q\, g) + S^\intercal\, g, \tag{5.5c}$$

$$H := R + G^\intercal\, Q\, G + S^\intercal\, G + G^\intercal\, S, \tag{5.5d}$$

where $g := (g_1^\intercal, \ldots, g_N^\intercal)^\intercal$, $h := (h_0^\intercal, \ldots, h_{N-1}^\intercal)^\intercal$ and

$$\begin{aligned} G &:= \begin{bmatrix} G_{1,0} & & \\ \vdots & \ddots & \\ G_{N,0} & \cdots & G_{N,N-1} \end{bmatrix}, \\ H &:= \begin{bmatrix} H_{0,0} & \cdots & H_{0,N-1} \\ \vdots & \ddots & \vdots \\ H_{N-1,0} & \cdots & H_{N-1,N-1} \end{bmatrix}. \end{aligned} \tag{5.6}$$

For future reference, we define the Lagrangian of (5.1),

$$\begin{aligned} \mathcal{L}(u, x, \mu, \lambda, \nu) &:= \sum_{k=0}^{N-1} \left(\frac{1}{2}\, u_k^\intercal\, R_k\, u_k + x_k^\intercal\, S_k\, u_k + u_k^\intercal\, r_k \right) \\ &+ \sum_{k=1}^{N} \left(\frac{1}{2}\, x_k^\intercal\, Q_k\, x_k + x_k^\intercal\, q_k \right) + \sum_{k=0}^{N-1} \mu_k^\intercal u_k + \sum_{k=1}^{N} \lambda_k^\intercal x_k \\ &+ \sum_{k=0}^{N-1} \nu_{k+1}^\intercal\, (A_k\, x_k + B_k\, u_k + c_k - x_{k+1}), \end{aligned} \tag{5.7}$$

where we have introduced the multipliers ν, μ, and λ for the constraints (5.1b), (5.1c) and (5.1d) respectively.

5.3 The classical condensing algorithm

The classical algorithm for a structure-exploiting computation of (5.5) can be derived as follows, cf. [44, 61, 80, 123, 131, 132].

Since A in (5.2) lower triangular, g can be calculated from (5.5a) using a blockwise forward substitution:

Algorithm 5.1 Calculating g from (5.5a)

$g_0 \leftarrow x_0$
for $k = 0, \ldots, N-1$ **do**
  $g_{k+1} \leftarrow A_k\, g_k + c_k$
end for

Analogously, G from (5.5b) is computed using blockwise forward substitutions exploiting the sparsity structure of the right-hand-side B:

Algorithm 5.2 Calculating G from (5.5b)

for $i = 0, \ldots, N-1$ **do**
  $G_{i+1,i} \leftarrow B_i$
  for $k = i+1, \ldots, N-1$ **do**
    $G_{k+1,i} \leftarrow A_k\, G_{k,i}$
  end for
end for

Subsequently, h is obtained by performing the matrix-vector multiplications given in (5.5c):

Algorithm 5.3 Calculating h from (5.5c), $\mathcal{O}(N^2)$ complexity

for $i = 0, \ldots, N-1$ **do**
  $h_i \leftarrow r_i + S_i^\intercal\, g_{i+1}$
  for $k = i+1, \ldots, N$ **do**
    $h_i \leftarrow h_i + G_{k,i}^\intercal\, (q_k + Q_k\, g_k)$
  end for
end for

Analogously, H is calculated by carrying out the blockwise matrix operations in (5.5d), exploiting the special structure of R, Q, S and G. Due to symmetry of H, only the upper triangular part needs to be calculated. To limit the floating

point operations, we precalculate $W := Q\,G$, a step that can be avoided if e.g. Q is diagonal.

Algorithm 5.4 Calculating H from (5.5d), $\mathcal{O}(N^3)$ complexity

for $i = 0, \dots, N-1$ **do**
  for $k = 1, \dots, N$ **do**
    $W_{i,k} \leftarrow Q_k\, G_{k,j}$
  end for
  for $j = 0, \dots, i-1$ **do**
    $H_{i,j} \leftarrow S_i^\intercal\, G_{i+1,j}$
    for $k = i+1, \dots, N$ **do**
      $H_{i,j} \leftarrow H_{i,j} + G_{k,i}^\intercal\, W_{k,j}$
    end for
  end for
  $H_{i,i} \leftarrow R_i$
  for $k = i+1, \dots, N$ **do**
    $H_{i,i} \leftarrow H_{i,i} + G_{k,i}^\intercal\, W_{k,i}$
  end for
end for

The Algorithms 5.1, 5.2, 5.3 and 5.4 together allow to formulate the condensed QP (5.4), which can be solved using a dense QP solver yielding the primal solution u and multipliers for the inequality constraints λ.

To recover x and ν, Constraints (5.1b) can be used in combination with $\nabla_{x_k} \mathcal{L} = 0$, $k = N, \dots, 1$, which are necessary conditions of optimality:

Algorithm 5.5 Recovering x and ν in (5.1)

for $k = 0, \dots, N-1$ **do**
  $x_{k+1} \leftarrow A_k\, x_k + B_k\, u_k + c_k$
end for
$\nu_N \leftarrow \lambda_N + Q_N\, x_N$
for $k = N-1, \dots, 1$ **do**
  $\nu_k \leftarrow \lambda_k + Q_k\, x_k + A_k^\intercal\, \nu_{k+1} + S_k\, u_k + q_k$
end for

Table 5.1: Complexity analysis, $\mathcal{O}(N^3)$ condensing

Step	Cost [FLOPs], leading term(s)
G (Algorithm 5.2)	$\frac{1}{2}\,N^2\,n_x^2\,n_u$
H (Algorithm 5.4)	$\frac{1}{6}\,N^3\,n_x\,n_u^2 + N^2\,n_x^2\,n_u$
g (Algorithm 5.1)	$N\,n_x^2$
h (Algorithm 5.3)	$\frac{1}{2}\,N^2\,n_x^2\,n_u + \frac{1}{2}\,N^2\,n_x\,n_u^2$
u, λ (Solving QP)	$\frac{1}{3}\,N^3\,n_u^3$
s,ν (Algorithm 5.5)	$3\,N\,n_x^2 + 2\,N\,n_x\,n_u$

Complexity analysis

The computational complexity of the established condensing algorithm can be assessed by counting the number of floating point operations (FLOPs)[4] in the corresponding algorithms. A summary of the complexity can be found in Table 5.1. The entries of the table are consistent with the complexity observed, e.g., by Kirches et al. [122]. When special structures occur, like Q diagonal or S zero, the coefficients of the table may become lower.

5.4 A condensing algorithm with N^2 complexity

An alternative algorithm for h can be derived by exploiting the special structure of G. From (5.5):

$$h = r + G^\intercal\,(q + Q\,g) + S^\intercal\,g = r + B^\intercal\,\underbrace{A^{-\intercal}\,(q + Q\,g)}_{:=w} + S^\intercal\,g, \tag{5.8}$$

which can be calculated cheaply using matrix-vector multiplications as well as a blockwise backward substitution for the block upper triangular matrix $A^\intercal$, we get the following algorithm:

[4] *Multiply-accumulate* (MAC) is assumed to be a single cycle operation. This holds for modern processors.

Algorithm 5.6 Calculating h from (5.5c), $\mathcal{O}(N)$ complexity

$w_N \leftarrow q_N + Q_N\, g_N$
for $k = N-1, \ldots, 1$ **do**
  $h_k \leftarrow r_k + S_k^\intercal\, g_k + B_k^\intercal\, w_{k+1}$
  $w_k \leftarrow q_k + Q_k\, g_k + A_k^\intercal\, w_{k+1}$
end for
$h_0 \leftarrow r_0 + S_0^\intercal\, x_0 + B_0^\intercal\, w_1$

From (5.5d), we also get an alternative way of calculating H:

$$\begin{aligned} H &= R + G^\intercal\, Q\, G + S^\intercal\, G + G^\intercal\, S \\ &= R + B^\intercal \underbrace{A^{-\intercal}\,(Q\, G)}_{:=W} + S^\intercal\, G + (S^\intercal\, G)^\intercal. \end{aligned} \tag{5.9}$$

Using blockwise backward substitution to calculate W and exploiting the sparsity structure of the matrices, we get an algorithm to calculate the lower triangular part of H:

Algorithm 5.7 Calculating H from (5.5d), $\mathcal{O}(N^2)$ complexity

for $i = 0, \ldots, N-1$ **do**
  $W_{N,i} \leftarrow Q_N\, G_{N,i}$
  for $k = N-1, \ldots, i+1$ **do**
    $H_{k,i} \leftarrow S_k^\intercal\, G_{k,i} + B_k^\intercal\, W_{k+1,i}$
    $W_{k,i} \leftarrow Q_k\, G_{k,i} + A_k^\intercal\, W_{k+1,i}$
  end for
  $H_{i,i} \leftarrow R_i + B_i^\intercal\, W_{i+1,i}$
end for

The equivalence between Algorithms 5.3 and 5.6 and the equivalence between Algorithms 5.4 and 5.7 are given from the manner in which the algorithms were derived.

Complexity analysis

For the proposed algorithm, the complexity in terms of floating point operations, again to the highest order term, is shown in Table 5.2. We see that the complexity decreases from cubic to quadratic in the calculation of H

Table 5.2: Complexity analysis, $\mathcal{O}(N^2)$ condensing

Step	Cost [FLOPs], leading terms
H (Algorithm 5.7)	$N^2\, n_x^2\, n_u + N^2\, n_x\, n_u^2$
h (Algorithm 5.6)	$2\, N\, n_x^2 + 2\, N\, n_x\, n_u$

and from quadratic to linear in the calculation of h. The complexities in n_x and n_u are unchanged.

5.5 Implications for gradient-based optimization

From (5.5b) and (5.5d), we can also use the above technique to derive an algorithm for multiplying H by an arbitrary vector $v = (v_0^\intercal, \ldots, v_{N-1}^\intercal)^\intercal$. Solving

$$g' := G\, v = A^{-1}\, B\, v,$$

$$h' := H\, v = R\, v + B^\intercal \underbrace{A^{-\intercal}\, (Q\, g' + S\, v)}_{:=w} + S^\intercal\, g',$$

by a blockwise backward substitution, we get:

Algorithm 5.8 Multiplying G from (5.5b) and H from (5.5d) with a vector v

$g'_0 \leftarrow 0$
for $k = 0, \ldots, N-1$ **do**
  $g'_{k+1} \leftarrow A_k\, g'_k + B_k\, v_k$ ▷ $g' = Gv$
end for
$w_N \leftarrow Q_N\, g'_N$
for $k = N-1, \ldots, 1$ **do**
  $h'_k \leftarrow R_k\, v_k + S_k^\intercal\, g'_k + B_k^\intercal\, w_{k+1}$ ▷ $h' = Hv$
  $w_k \leftarrow S_k\, v_k + Q_k\, g'_k + A_k^\intercal\, w_{k+1}$
end for
$h'_0 \leftarrow R_0\, v_0 + B_0^\intercal\, w_1$

Algorithm 5.8 can be used in QP solvers that require reduced Hessians-times-vector products to be calculated, particularly in the tailored adaptation of Nesterov's *fast gradient method* presented by Richter et al. [160]. Using Algorithm 5.8, the central Hessian-vector product now can be performed at the

cost of $N\,(3\,n_x^2+4\,n_x\,n_u+n_u^2)-2\,n_x\,n_u-n_x^2$ FLOPs instead of $\mathcal{O}(N^2\,n_u^2)$ FLOPs as stated in [160], since the main computational cost is the Hessian-vector product. For problems with long control horizons this may be a significant improvement. All other features of the algorithm remain unchanged. It should be noted that the result of computing products of a reduced Hessian times a vector at a complexity of $\mathcal{O}(N)$ FLOPs has been used implicitly before, see, e.g., Kögel and Findeisen [124].

5.6 Numerical results

To assess the proposed condensing algorithm, we consider the benchmark from nonlinear MPC presented in [191]. A chain of masses connected by springs with one end fixed are steered to an equilibrium position with a wall constituting a hard constraint on the state trajectory. With 9 masses, we obtain a system with $n_x = 57$ states and $n_u = 3$ controls. We present the execution time of Algorithms 5.1, 5.2, 5.5, 5.6 and 5.7 for control horizons up to 100 in Figure 5.1. The timings are compared against the standard $\mathcal{O}(N^3)$ condensing algorithm, i.e. Algorithm 5.1, 5.2, 5.3, 5.4 and 5.5. We have also included the runtime ratio between the old and new algorithms. Both algorithms have been implemented in ANSI-C as part of the ACADO Code Generation tool [117]. The results, which have been produced on a 3GHz Intel Q9650 based desktop computer running Linux, confirm significant savings of 60 % or more (i.e. a speedup of 2.5 or greater) for all considered horizon lengths.

5.7 Conclusions and outlook

For the benchmark considered, the new condensing algorithm proposed in this chapter outperformed the classical one for all considered control horizons with a factor of at least 2.5. This suggests that the proposed condensing algorithm is not only superior for long control horizons, but for short to medium ones as well. We see that the curve for the speed-up factor in Figure 5.1 becomes asymptotically linear for growing horizons, confirming the one order of magnitude improvement.

When regarding the complete algorithm to solve the full space QP, there is one remaining step that is of cubic complexity: the solution of the condensed QP. This cost arises from the Cholesky factorization of the Hessian before the first active set change, while all active set changes have a cost of $\mathcal{O}(N^2\,n_u^2)$. Though this cost is independent of the state dimension n_x and in most practical

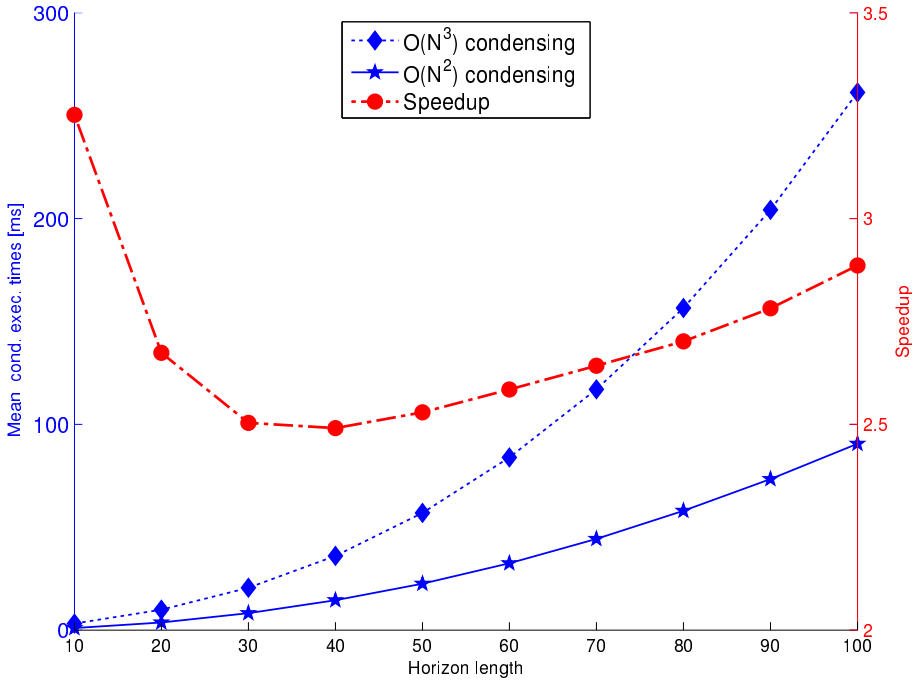

Figure 5.1: Execution times vs. horizon length for benchmark [191]

instances still lower than the new condensing algorithm of quadratic complexity, it would be desirable to find an active set algorithm for the condensed QP with quadratic complexity, which is an interesting topic for future research.

Chapter 6

CasADi – A framework for algorithmic differentiation and numerical optimization

In this chapter, the open-source software package CasADi is presented. We will give a general overview of the tool and provide details on the implementation of CasADi, including algorithmic novelties that we believe are of interest to the AD community. After stating the scope of the project in Section 6.1, we introduce the syntax and usage of the tool in Section 6.2. For a more in-depth discussion, we refer to CasADi's user guide [28]. In Section 6.3, we discuss the software architecture of CasADi's symbolic core, which includes how expressions are represented and evaluated numerically or symbolically in virtual machines (VM). We then discuss the AD approach in Section 6.4, which includes both the operator overloading (OO) and source code transformation (SCT) approach (cf. Section 3.5) and a graph coloring approach for calculation of complete Jacobians and Hessians. In Section 6.5, we discuss a novel approach to calculate the sparsity structure by using graph coloring for large, but structured Jacobians. In Section 6.6, we discuss how the numerical evaluation or sparsity pattern calculation can be accelerated through the use of code generation. In Section 6.7, we assess the performance of the tool by comparing different ways of evaluating symbolic expression and compare the result the virtual machine of AMPL, representing the state-of-the-art for high-level optimization modeling. We end with an outlook on future developments in Section 6.8.

6.1 Scope of the project

The aim of CasADi is to make state-of-the-art methods for optimal control, nonlinear programming and algorithmic differentiation, as presented in the previous chapters, more accessible to the users and developers of such methods. The tool has been implemented jointly with fellow PhD student Joris Gillis and in close contact with users from industry and other research groups.

The project addresses the solution of optimal control problems using a different approach than the existing packages presented in Chapter 1. Rather than providing the end user with a "black box" OCP solver, i.e. a convenient way of entering an OCP and have it solved automatically, the focus is to offer a generic set of building blocks for OCP solution. This allows advanced users to easily implement a wide range of methods without compromising efficiency. While the tool has found uses that were not originally intended, the main target groups of the tool are:

Advanced users of optimal control, who wish to tailor an OCP solver to a problem with special structure or a nonstandard problem formulation

Method researchers who wish to be able to rapidly prototype novel algorithms for nonlinear programming or optimal control

Software developers of general purpose or specific purpose optimal control problem solvers that need base functionalities appearing in any derivative-based optimal control code

CasADi is written in self-contained C++, relying on nothing but the C++ Standard Library. It can be used either from C++ directly, or via full-featured front-ends to Python and Octave, generated automatically using the *Simplified Wrapper and Interface Generator* (SWIG) [32] framework. CasADi is distributed open-source under the permissive *GNU Lesser General Public License* (LGPL) via its website [18].

6.2 Syntax and usage

CasADi is designed to have the *look-and-feel* of a computer algebra system (CAS), with rudimentary support for manipulating expressions in symbolic form. In contrast to conventional CAS-tools however, it uses an expression graph representation with support for shared subexpressions as discussed in

Section 3.5. To illustrate this, consider the following code segment for CasADi's Octave front-end:

```
x = ssym('x');
for i=1:100
  x = x*sin(x);
end
```

The meaning is the same as the corresponding code for MATLAB's Symbolic Math Toolbox in Section 3.5, but instead of 2^{100} nodes, this code will result in 200 nodes, corresponding to 100 multiplications and 100 sine operations.

Expression graphs with scalar-valued operations

The symbolic framework allows the user to construct symbolic expressions using two different graph representations. In the first, *scalar* graph representation, the atomic operations of the expression graphs are scalar-valued expressions, similar to the representation being used in most AD tools. They are exposed to the user via a MATLAB inspired *everything-is-a-matrix* data type, i.e. vectors are treated as n-by-1 matrices and scalars as 1-by-1 matrices. All matrices are sparse and the elements consisting of scalar symbolic expressions are stored in a general sparse format, namely *compressed row storage* (CRS)[5] [25]. To illustrate the usage, the following code calculates an expression for the gradient of the determinant function, $f(x) = \det(x), x \in \mathbb{R}^{3\times 3}$, using CasADi's Python front-end:

Listing 6.1: Gradient of the determinant, scalar atomic operations

```
from casadi import *
x = ssym("x",3,3)
f = det(x)
print gradient(f,x)
```

which outputs (formatted for clarity):

$$\begin{bmatrix} x_4\,x_8 - x_7\,x_5 & x_5\,x_6 + x_8\,(-x_3) & x_4\,(-x_6) + x_7\,x_3 \\ -(x_1\,x_8 - x_7\,x_2) & x_2\,(-x_6) + x_8\,x_0 & x_1\,x_6 + x_7\,(-x_0) \\ x_1\,x_5 - x_4\,x_2 & x_2\,x_3 + x_5\,(-x_0) & x_1\,(-x_3) + x_4\,x_0) \end{bmatrix}.$$

In the code above, `x = ssym("x",3,3)` declares a 3-by-3 matrix with scalar symbolic primitives, $x_0, \ldots, x_8$ and `f = det(x)` forms a symbolic expression for

[5] Future versions of CasADi may switch to the related, but more conventional *compressed column storage* (CCS) [25].

the determinant using minor expansion. Finally, the function `gradient(f,x)` calculates the gradient of the scalar (i.e. 1-by-1) expression `f` with respect to entries of the matrix `x` and returns the result as a matrix with the same dimension as `x`. Internally, the reverse mode of AD using SCT, cf. Section 3.5, is used for the calculation.

Expression graphs with matrix-valued operations

In the second, *matrix* graph representation, the atomic operations of the expression graph are multiple-input, multiple-output where all inputs and outputs are sparse matrix-valued. The syntax is designed to mirror the scalar graph representation as much as possible. For the example above, we get:

Listing 6.2: Gradient of the determinant, matrix atomic operations

```
from casadi import *
x = msym("x",3,3)
f = det(x)
print gradient(f,x)
```

where `ssym` has been replaced by `msym`, and the output is now instead (again formatted for clarity):

$$\det(x)\, x^{-\intercal}$$

as can be expected from the derivative results for the determinant (cf. [89]).

The rationale behind having two different expression graph formulations in parallel is to allow the low level operations such as the evaluation of a DAE right hand side in a small-to-medium size problem to be evaluated with minimal overhead and maximal symbolic simplifications, but then use the more general graph to represent high level operations for algorithms that are naturally vector- or matrix-valued. The matrix graph representation also acts as a "glue" allowing high level operations such as implicit functions (Section 3.9) and ODE/DAE integrators (Chapter 7) into the expression graphs.

Both expression graphs can be used to formulate function objects, also known as functors [24], corresponding to multiple-input, multiple-output functions with inputs and outputs being sparse matrices. Calls to these functors can then be embedded into the matrix expression graph above, providing an intuitive way for the user to do *checkpointing*, cf. [104, Chapter 12].

Function objects, which can be embedded into symbolic expressions, need not be defined by an expression graph. They can also be an external C,

C++ or Python function or an ODE/DAE integrator, or a solver of linear, quadratic, semidefinite or nonlinear programs. Note that embedding such generic functions may result in expressions that are only piecewise differentiable or non-differentiable.

External solver interfaces

For the development of optimization code, several solver packages have been written or interfaced. These solvers all take the form of function objects, allowing them to be embedded into expression graphs, and providing them with a uniform way for interacting with the user. Interfaced tools include ODE/DAE integrators, in particular the SUNDIALS suite [113], where directional derivatives are calculated through automatic formulation and solution of the forward and adjoint sensitivity equations (see Chapter 7). Other interfaced solvers are QP solvers such as qpOASES [75], OOQP [87] and CPLEX (as a QP solver) [118] and NLP solvers such as IPOPT [179], KNITRO [56] and WORHP [55]. Similar to algebraic modeling languages (Section 3.5), derivative information in the form requested by the solver in question is automatically generated and passed on to the solver.

Model import

In addition to formulating problems in CasADi directly, it is also possible to import NLPs formulated in the algebraic modeling language AMPL [84], which we mentioned in Section 3.5. AMPL consists of two parts, a closed-source parser and an open-source runtime known as the *AMPL Solver Library* (ASL). To pass model data from the parser to the runtime, AMPL uses the so-called *.nl* file format [85]. CasADi supports import of NLPs formulated in *.nl* files, and the symbolic nature of the format allows CasADi to construct a symbolic representation of the NLPs. The NLPs can then be manipulated and solved with the NLP solvers contained in or interfaced with CasADi. Generation of *.nl* files is also possible via the open-source modeling environment Pyomo [111].

Model import is also possible from the physical modeling language Modelica [17] via the open-source compiler JModelica.org [20, 26]. The import, which uses an extension of the standard *Functional Mock-up Interface* (FMI) standard [9], allows CasADi to build up a symbolic representation of the OCPs. Different symbolic manipulations can be applied to an imported models, including scaling, sorting of variables and equations and reformulation of DAEs from fully-implicit to semi-explicit form.

6.3 Implementation of the symbolic core

In the following we provide some detail on the implementation of CasADi.

An *everything-is-a-sparse-matrix* data type

As mentioned above, the symbolic data types exposed to the user are sparse matrices. The rationale for this is that for symbolic calculations, the cost for the linear algebra is hardly going to be a bottleneck. By treating all symbolic scalars, vectors and dense matrices as sparse matrices, the number of data types exposed to the user can be minimized. This flattens the learning curve and facilitates the maintenance of the software. In addition, existing packages for linear algebra, including uBLAS [11] and Eigen [7] in C++ and NumPy/SciPy in Python, are typically not suitable for storing or manipulating matrices with symbolic entries. For example, conventional linear algebra packages would expect that the result of a comparison such as $(x \geq y)$ be either *true* or *false*. If x and y are symbolic expressions, however, the result of $(x \geq y)$ is likely to be an expression that evaluates to either *true* or *false*. Restrictions like these means that the linear algebra algorithms must be modified accordingly. As an example, this could mean finding alternatives to *partial pivoting* (see e.g. [69]) during the solution of linear systems of equations.

We note that it is in general cheap to detect if a sparse matrix has a certain structure, e.g. having no structural zeros, being diagonal matrix etc. This can then be exploited, e.g. in specialized algorithms for matrix multiplication.

Scalar graph representation and VM design

The elements of the above matrix class can be either floating point variables (e.g. `double`) or scalar-valued symbolic expressions. The scalar expression type has been implemented using operator overloading [187] in C++.

In contrast to corresponding data types in traditional operator overloading AD tools as presented in Section 3.5, the purpose of the type is only to formulate symbolic expressions. It is not designed to be a "drop-in replacement" for a floating point type as is the case for e.g. `adouble` in ADOL-C [102] or `AD<double>` in CppAD [5].

In practice it means that code that contains control flow instructions such *if*-statements, must always be manually reformulated. Consider e.g. the code segment:

```
double x,f;
x = ...
if(x >= 0){
  f = x;
} else {
  f = 5;
}
```

This code segment can be readily used in a traditional OO tool by only replacing `double` for e.g. `adouble` in ADOL-C. When executing the code, the condition `x >= 0` will be determined by the current numerical value for `x`. Typically, this means that the symbolic expression for `f` is only valid for certain numerical values and if `x >= 0` changes value, the symbolic expression needs to be regenerated, or *retraced*, cf. [104]. In CasADi, the same segment would read:

```
CasADi::SX x,f;
x = ...
f = if_else(x >= 0, x, 5);
```

Note that similar constructs do exist in convensional AD tools, to avoid retracing, but are not *required* as in CasADi.

Requirements like these makes CasADi fundamentally unsuitable to use "black-box" on an existing C-code when compared to ADOL-C or CppAD. It also disqualifies the use of linear algebra packages as explained in the previous section. On the other hand, it also guarantees that a symbolic expression never needs to be regenerated once formed and enables more aggressive simplifications of the expressions to be made.

The scalar symbolic type `CasADi::SX` is not intended to be used directly by the end user. Instead, as explained in Section 6.3 above, the user typically works with matrices of `CasADi::SX`.

```
CasADi::SXMatrix x,f;
x = ...
f = if_else(x >= 0, x, 5);
```

The primary purpose of constructing symbolic expressions in CasADi is to define function objects that allow numerical and symbolical evaluation. The first step in creating such a function object is a *depth-first search* topological sort of the corresponding expressions. The sorted expression graph is then used to form an array of instructions to be performed on a work array. Such an instruction is stored in the following data structure that takes up 16 bytes on typical architectures:

```
struct ScalarAtomic{
  int op;       // Operator index
  int i0;
  union{
```

```
      double d;
      struct{ int i1,i2; };
    };
  };
```

Each instruction – which can be either a function input or output storage instruction, a double precision constant or a unary/binary instruction – contains an identifier ("op" above) identifying the type of instruction as well as either three integers or one integer and a double precision floating point value.

This array of instructions (implemented as `std::vector<ScalarAtomic>`) is then evaluated in a register-based virtual machine, cf. Section 3.5. The core of the VM is a switch-statement inside a for-loop of the form:

Listing 6.3: Layout of the scalar graph virtual machine

```
for(auto i=algorithm.begin(); i!=algorithm.end(); ++i){
  switch(i->op){
    case OP_INPUT:    work[i->i0] = input[i->i1][i->i2];          break;
    case OP_OUTPUT:   output[i->i0][i->i2] = work[i->i1];         break;
    case OP_CONSTANT: work[i->i0] = i->d;                         break;
    case OP_ADD:      work[i->i0] = work[i->i1] + work[i->i2];    break;
    case OP_SIN:      work[i->i0] = sin(work[i->i1]);             break;
    ...
  }
}
```

On typical architectures, this allows the switch-statement translates to an efficient lookup table in the compiled code.

The locations in the work vector are determined by analyzing the live variable ranges following the topological sorting of the expression graph, and work vector elements are then reused on a *last in, first out* basis.

Matrix graph representation and VM design

While the scalar graph representation is designed for minimal overhead, the design goal of the matrix graph representation is maximum generality. This means that in addition to (now matrix-valued) constants and (now elementwise) unary/binary operations, several additional atomic operations are needed. The most important of these are shown in Table 6.1.

Table 6.1: Atomic operations in the matrix graph representation with interdependencies for forward and reverse mode AD by SCT

	Operation	Definition	Interdependencies Forward	Adjoint
1	Constant	$0, I, [1.2, 4]$, etc.		
2	Elementwise operation	$\sin(X), X * Y$, etc.		3 [a]
3	Inner product	$\mathrm{tr}(X^\intercal Y)$ [b]		2
4	Transpose	$X^\intercal$		
5	Matrix multiplication	$X + Y\,Z^\intercal$		4
6	Reshape	(changes dimension)		
7	Sparse assignment	(changes sparsity)		
8	Submatrix access	$X(\mathit{elements})$		9
9	Submatrix assignment	$X(\mathit{elements}) = Y$ [c]		8
10	Inverse	X^{-1}	4,5	4,5
11	Determinant	$\det(X)$	2,3,4,10	2,4,10
12	Function call	(see text)	(4,5)	(4,5)
13	Linear solve	$X\,Y^{-1}$ or $X\,Y^{-\intercal}$	4,5,16,17	4,5,16,17
14	Implicit functions	(Section 3.9)	4,5,12,13	4,5,12,13
15	Frobenius norm	$\|X\|_\mathrm{F}$	2,3	2
16	Vertical concatenation	$[x_1; x_2; ..., x_n]$		17
17	Vertical split	$[z_1; z_2; ..., z_n] = X$		16

[a] Dependency appears for e.g. $x * Y$, when x is a scalar and Y is a matrix
[b] Efficiently evaluated by the equivalent expression $\sum_{i,j} X_{i,j}\,Y_{i,j}$
[c] More precisely, the two step evaluation $F := X$, $F(\mathit{elements}) = Y$

The set of atomic operations is selected to ensure that both forward and adjoint derivative propagations can be expressed with the same set of operations. For example, using the notation of Chapter 3, the matrix multiplication, $f(X, Y, Z) := X + Y\,Z^\intercal$, will result in:

$$\hat{f}(X, \hat{X}, Y, \hat{Y}, Z, \hat{Z}) = \hat{X} + \hat{Y}\,Z^\intercal + Y\,\hat{Z}^\intercal = f(f(\hat{X}, \hat{Y}, Z), Y, \hat{Z}) \tag{6.1}$$

in the forward mode, and

$$\begin{aligned}
&\left\{\bar{X}^{\text{after}}, \bar{Y}^{\text{after}}, \bar{Z}^{\text{after}}\right\} \\
&= \left\{\bar{X}^{\text{before}}, \bar{Y}^{\text{before}}, \bar{Z}^{\text{before}}\right\} + \bar{f}(X, Y, Z, \bar{F}) \\
&= \left\{\bar{X}^{\text{before}}, \bar{Y}^{\text{before}}, \bar{Z}^{\text{before}}\right\} + \left\{\bar{F}, \quad \bar{F}\,Z, \quad \bar{F}^\intercal\,Y\right\} \\
&= \left\{\bar{X}^{\text{before}} + \bar{F}, \quad f(\bar{Y}^{\text{before}}, \bar{F}, Z^\intercal), \quad f(\bar{Z}^{\text{before}}, \bar{F}^\intercal, Y^\intercal)\right\}
\end{aligned} \tag{6.2}$$

in the reverse mode. Therefore, if an expression contains matrix multiplications, the expression for its derivative will contain matrix multiplications and, for the reverse mode, additions and matrix transposes. This dependency is indicated in the last two columns of Table 6.1, which list the interdependencies between the atomic operations. For details on how to derive this and other derivative propagation rules, we refer to [89, 90].

A central operation is the *function call* operation, which corresponds to a call to an arbitrary multiple input, multiple output function, that may or may not have been defined by a symbolic expression. The function call operation, when differentiated, gives rise to a new function call operation to a different function that calculates f as well as $\hat{f}$ and/or $\bar{f}$ for a number of derivative directions simultaneously. The derivative calculation can be interrupted when only a subset of these directions is needed using the partial evaluation discussed in Section 6.3. To limit memory use, the functions for calculating directional derivatives are cached allowing an expression graph to contain multiple calls to the same derivative function.

The matrix graph representation may also contain calls to the solution of linear systems of equations, using any of the linear solvers interfaced with CasADi. Derivatives are then calculated using the propagation rules from Table 3.2. The same linear solver instances and, when possible, factorizations are then reused for the derivative calculation. Calls to nonlinear equation solvers are supported as suggested in Section 3.9, first part.

As for the scalar expression graph, the matrix expression graph corresponding to a function is topologically sorted using a depth-first search for evaluation in a register-based virtual machine operating on a work vector consisting of a set of sparse matrices. The work vector is kept small by again reusing work vector elements on a last in, first out basis (now with one "stack" per unique sparsity pattern). This makes sure that an operation of the form:

$$X^{\text{after}} = X^{\text{before}} + Y\,Z^{\intercal}, \tag{6.3}$$

resulting from the the reverse mode propagation for matrix multiplication shown in (6.2), will have X^{after} and X^{before} point to the same location in memory (provided that X^{before} is not needed later in the algorithm). This is then exploited by the multiplication routine. Another way of expressing this is that we allow the elements of the work vector in the matrix VM to be *mutable* objects.

For some operations, this *in-place* calculation is critical as not to violate the complexity bounds for the reverse mode of AD. For instance, the *submatrix access* operation:

$F \leftarrow X(elements)$

will give rise to a submatrix assignment operation illustrated with the algorithm:

$\bar{X}^{\text{after}} \leftarrow \bar{X}^{\text{before}}$

$\bar{X}^{\text{after}}(elements) \leftarrow \bar{X}^{\text{after}}(elements) + \bar{F}$

If the dimension of X is much larger than F, the computational cost will be dominated by the trivial assignment and not by the addition. This means that the reverse mode applied to an algorithm involving submatrix accesses can give an algorithm that is arbitarly more expensive than the original algorithm, unless the assignment is avoided. The use in-place operations avoids trivial assignments like these whenever possible.

6.4 AD implementation

CasADi supports AD by both the operator overloading approach and the source code transformation approach for both expression graph representations. The operator overloading approach is implemented by executing Algorithms 3.3 or 3.4 with a numerical data type calculating up to (by default) 64 forward or adjoint directional derivatives at a time. For the reverse mode, we include a forward sweep to either store the partial derivatives of f (for the scalar expression graph) or the arguments of f (for the matrix expression graphs)[6]. AD by source code transformation is implemented by executing Algorithms 3.3 or 3.4 with the corresponding symbolic data type giving new expressions for the directional derivatives. To limit the creation of duplicate expression, it reuses the nodes of the expression graph used to define the function as much as possible.

Jacobians are calculated by first determining their sparsity structure using the bitvector approach explained in Section 3.7. This is implemented by propagating sets of (on most architectures) 64 booleans, implemented by reusing data structures for the double precision (i.e. 64 bit) numerical evaluation to hold arrays of 64-bit integer types (via `reinterpret_cast<unsigned long long*>` in C++). Hessians are calculated by using a symmetry-exploiting variant of the same algorithm applied to the gradient, which in turn has been calculated with reverse mode AD by source code transformation. To limit the number of passes needed to determine the sparsity pattern, a novel seeding approach, described in Section 6.5 below, is used. There exist also the possibility to just-in-time compile the sparsity propagation algorithm on central or graphics processing units (GPU/CPU) as described in Sections 6.6.

[6]To be more precise, the arguments that are overwritten (or *spilled*) from the work vector are stored to a stack.

With the sparsity pattern known, graph coloring is used to determine a set of forward or adjoint directional derivatives needed to retrieve the Jacobian entries. Two algorithms are implemented for non-symmetric and symmetric Jacobians respectively; a (greedy, distance-2) unidirectional algorithm [86, Algorithm 3.1] and a (greedy, distance-2) star coloring algorithm [86, Algorithm 4.1] with a *largest-first* ordering [86, 183][7]. For the unidirectional coloring, both coloring of rows and columns are tested, starting with the one most likely to produce a good coloring and interrupting the second coloring prematurely if worse than the first. This will give an expression for the Hessian or Jacobian containing (possibly inlined) calls to functions for calculating directional derivatives using either an operator overloading or source code transformation approach.

6.5 Hierarchical sparsity pattern calculation

As noted in Section 3.7, determining the sparsity pattern of a Jacobian is often the most expensive step in the Jacobian calculation. When using the bitvector approach, even when calculating the nonzeros of 64 rows or columns at a time, there is still a linear dependence on either the number of columns of the Jacobian – when propagating dependencies forward through the algorithm – or on the number of rows – when propagating dependencies backward. For a large NLP with hundreds of thousands of rows *and* columns, the cost can be prohibitive.

The sparsity pattern calculation can be sped up by using probabilistic methods as shown by Griewank and Mitev [103], which in particular makes sense when the nonzeros of the Jacobian are found at more or less random locations. The sparsity patterns of Hessians and Jacobians arising when solving optimal control problems, be they the Hessian of the Lagrangian and Jacobian of the constraints of an NLP or the Jacobian of an ODE/DAE right-hand-side function inside an integrator, seldom have nonzeros located at random locations. The typical situation is that they possess some kind of block sparsity with vast regions of only structural zeros and more or less sparse blocks. We show in Figure 6.1 an example of such a sparsity pattern, corresponding to the Hessian of the Lagrangian of an NLP arising from direct collocation.

To exploit block sparsity, we propose first to determine a crude sparsity pattern. In particular, by partitioning the function inputs of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ into 64 groups, the Jacobian can be written as a 64-by-m block matrix. We can determine which of these blocks have *any* nonzeros with a single forward pass

[7] At the time of writing, a new star coloring algorithm with lower complexity proposed by Gebremedhin et al. [86], was being implemented.

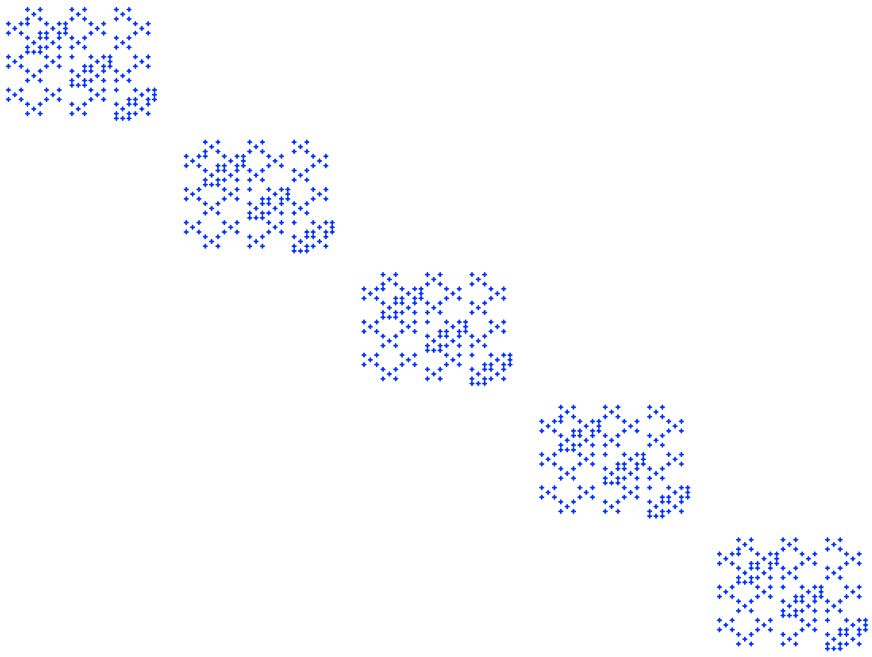

Figure 6.1: The sparsity pattern of the Hessian of the Lagrangian of an NLP arising from direct collocation

through the algorithm. Alternatively, we can partition the function outputs into 64 groups resulting in a n-by-64 block respresentation of the Jacobian. With a single reverse pass through the agorithm, we can determine which of those blocks have any nonzeros. We thereafter apply the graph coloring techniques explained in Section 6.5 to determine if any blocks can be calculated in parallel. This information is then used when calculating a finer sparsity pattern. The approach can be applied hierarchically with successively finer sparsity patterns until the true sparsity pattern is given.

The above approach was implemented by Joris Gillis. A more detailed description along with numerical results is planned for his dissertation.

6.6 Just-in-time compilation

There are limits to how fast virtual machines can be made. Heavily optimized operator overloading based AD tools like ADOL-C and CppAD and as well as the virtual machines of AMPL and CasADi, are still an order of magnitude slower than compiled C-code generated from a source-code transformation tool. To bridge this performance gap, it is necessary to generate machine code in one way or another.

Generating compiler intermediate representation

The most direct way, but also the most low-level is to directly build up the compiler intermediate representation. The open-source LLVM compiler framework [129] (formerly *low-level virtual machine*) provides a way to accomplish this while still retaining platform independence. In Section 6.7, we show that just-in-time compilation using the LLVM can give a speedup of factor 2-3 over the corresponding highly optimized virtual machine. By enabling code optimization in LLVM, another factor 2-3 can be obtained, although this can come at the cost of much longer compilation times.

C-code generation

Generating compiler intermediate representation tends to become complex when the code to be generated involves basic features such as branches and scopes. The situation is not helped by the fact that LLVM at the time of writing *does not maintain a stable application programming interface (API).* An alternative is to generate C-code, which offers a standardized higher-level API, compile it and finally load it as a dynamically linked library (DLL). This also represents a more indirect approach since it adds the steps of writing the code on disk and then use the parser of the C-compiler to internally construct the intermediate representation.

A practical difficulty with this approach includes the need to call a bitcode-compatible C-compiler. This can be overcome by linking with a C-compiler as opposed to calling it via the operating system. In particular, the Clang front-end to LLVM offers a way to do this with a permissive open-source software license. At the time of writing, CasADi supported C-code generation from a subset of classes (in particular not including interfacing to third party codes) and had a functionality to dynamically load the corresponding DLL on POSIX-

based operating systems (e.g. Linux and Mac OS X) as well as on Microsoft Windows.

Generating OpenCL kernels

The Open Computing Language (OpenCL) [121] is a C-based language for writing functions, called *kernels*, that can run on multiple architectures including central processing units (CPU) and graphics processing units (GPU). It represents both an extension and a restriction to conventional C. Extension, since compared to standard C99, it includes a set of new data types and a set of new keywords for defining the physical memory region of data. Restriction, since it omits many central C concepts such as function pointers and only implements a subset of the standard C library.

For algorithmic differentiation, OpenCL is particularly interesting since it allows certain data parallel tasks such as derivative calculation in multiple directions [98] or sparsity-pattern propagation to be performed using general-purpose computing on GPUs. OpenCL can also be used as an alternative to conventional C-code generation for non-data parallel tasks since it is platform independent and has a stable API. Since OpenCL is essentially C-code as far as code generation is concerned, it is relatively easy to maintain both conventional C-code generation and OpenCL-code generation, both supported in CasADi.

6.7 Benchmarking

Although the main goal of CasADi is to be more *accessible*, as opposed to *faster*, than existing packages for algorithmic differentiation and numerical optimization, raw numerical performance is also of importance. In the following, we show how the different ways of evaluating numerical expressions compare against each other and benchmark CasADi against AMPL Solver Library.

A speed benchmark problem

To assess the performance of the algorithmic differentiation in CasADi, we consider the problem of calculating the determinant of an n-by-n matrix:

$$f(X) = \det(X), \quad X \in \mathbb{R}^{n \times n}. \tag{6.4}$$

The simplest, but by no means the most efficient, way to calculate f is to use minor expansion, an algorithm with a complexity that grows exponentially with

the dimension of the matrix. In Python, minor expansion can be implemented as follows:

```
def f(A):
  # Get dimension, quick return if scalar
  n = A.shape[0]
  if n==1: return A

  # Expand along the first column
  R = 0; sign = 1
  for i in range(n):
    M = A[range(i)+range(i+1,n),1:,] # Minor
    R += sign*A[i,0]*f(M); sign = -sign
  return R
```

which is valid syntax for NumPy/SciPy (standard numerical packages for Python) as well as for CasADi.

A well-known benchmark example from algorithmic differentiation is to calculate the gradient of this function using reverse mode AD. Note that we already know the beginning of the chapter that the gradient is given by the explicit expression:

$$\nabla f(X) = \det(X)\, X^{-\mathsf{T}} \tag{6.5}$$

Using the scalar graph representation

We begin by calculating ∇f using CasADi's scalar representation and an OO approach. Since f is scalar-valued, a single adjoint sweep is enough to calculate the gradient. The code required is:

```
from casadi import *           # Import CasADi
from numpy.random import rand # A random matrix generator
X = ssym("X",n,n)              # Input expression
F = SXFunction([X],[f(X)])     # Create a function object
F.init()                       # Initialize for evaluation
F.setInput(rand(n,n))          # Let X be a random matrix
F.setAdjSeed(1.0)              # Set adjoint seed
F.evaluate(0,1)                # Evaluate with 0 forward
                               # and 1 adjoint direction
print F.adjSens()              # Print the gradient
```

In Figure 6.2 (line with squares, "□"), we see how the cost of the numerical evaluation increases with the dimension. For $n = 8$ the number of nodes in the expression graph is around 140 000 and the gradient calculation takes 4 ms. For $n = 11$, which corresponds to more than 100 million nodes, CasADi runs out of memory. All calculations have been performed on a Dell Latitude E6400 laptop equipped with a 2.4 GHz Intel Core duo processor (one core used) and 4 GB of RAM.

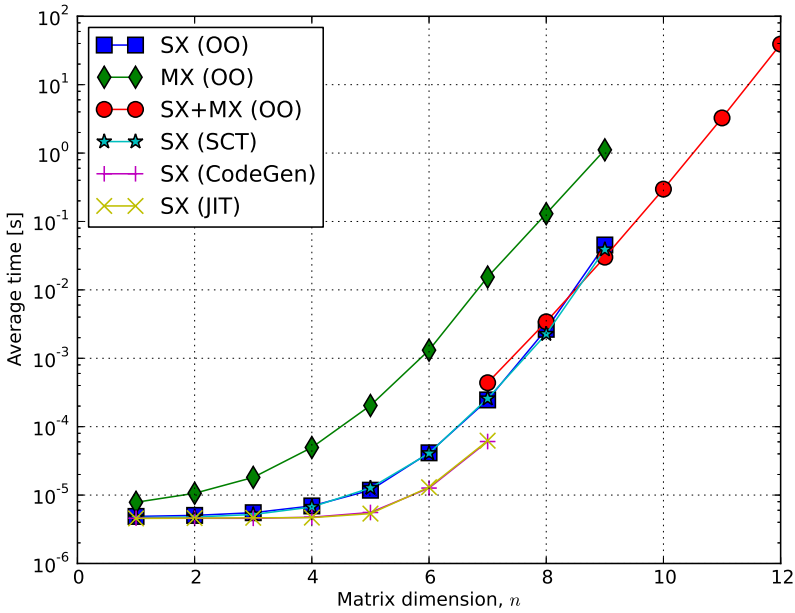

Figure 6.2: Gradient of the function $f(X) = \det(X)$, $X \in \mathbb{R}^{n \times n}$, for increasing n, calculated in different ways

Using the matrix graph representation

The same problem can be solved with CasADi's matrix representation, simply by replacing `ssym` with `msym` and `SXFunction` with `MXFunction` above. The result is also shown in Figure 6.2 (line with diamonds, "$\diamond$"). We note that the approach is significantly slower. For $n = 8$, the solution time is 132 ms, close to two orders of magnitudes slower than the scalar graph. This is not surprising, since the matrix representation has not been optimized for small computational overhead to the same extent as the scalar representation. Indeed, the main purpose of the matrix representation is to "glue" operations together that are by themselves expensive to evaluate, like numerical linear algebra operations or function evaluations.

To use the matrix graph formulation more wisely, we first define a function object using the scalar representation that calculates the determinant of a smaller problem, e.g. an 7-by-7 matrix. Let us call this function object `F_small`.

We can then build up an expression with function calls to `F_small` by replacing the line

```
if n==1: return A
```

with

```
if n==8:
  [d] = F_small.call([A]) # Create a function call to F_small
  return d                # Return the function call reference
```

Since this allows most of the computationally expensive work to take place in the (fast) virtual machine for the scalar representation, the approach is only slightly more expensive than the scalar graph approach, as seen in Figure 6.2 (line with circles, "$\circ$"). We essentially need one extra evaluation of the determinant (without directional derivatives) since the intermediate variables of `F_small` get overwritten between the calls. When memory runs out using this approach, we can define a new function for determinants of say 12-by-12 matrices and repeat the approach. The combined scalar-matrix graph approach, which in AD terminology can be considered a *checkpointing scheme*, hence allows us to limit the memory use at the expense of extra floating point operations.

Source code transformation approach

The gradient can also be calculated using a SCT approach. Indeed, simply invoking "`gradient(f(X),X)`", will create a new symbolic expression for the gradient using this approach. This expression can then be used to define a function object, which can be numerically evaluated as above. The result of this calculation is shown in Figure 6.2 (line with stars, "$\star$").

There are three situations when the SCT approach is particularly interesting:

- When higher order derivatives are requested, since the OO approach only supports first order derivatives. The SCT approach can be applied recursively to any order.
- When complete Jacobians or Hessians are calculated, since the sparsity pattern then only needs to be considered when constructing the graphs, not when numerically evaluating them. Simple but common operations such as multiplications by zero or one are eliminated from the graph altogether.
- When we want to generate code for the derivative calculation using C-code generation or just-in-time compilation.

The line with upright crosses (+) in Figure 6.2 shows the timings for a C-code generation approach, followed by compilation to a shared library using the gcc 4.6 compiler (`-O3` optimization enabled), in turn loaded into CasADi as a dynamically linked library (DLL). Finally, the line with diagonal crosses (×) shows the results using the just-in-time compilation approach, using LLVM version 3.0.

The main downside of using code generation in CasADi are long compilation times when dealing with very large symbolic expressions. This is also true for the just-in-time compilation approach, at least for the example considered here. If we use Clang, a C-compiler built on top of the LLVM framework (and hence optimized for fast compilation), the difference between just-in-time compilation and C-code generation is small.

Nonlinear programming

To assess the performance of CasADi for nonlinear programming, we use Bob Vanderbei's AMPL version of the CUTEr test problems suite. We use AMPL to parse the test problems and to generate *.nl* files that are imported into CasADi as described in Section 6.2. We then use CasADi to calculate first and second order derivative information, including sparsity and to solve the NLP using CasADi's interface to IPOPT [179] (Version 3.10 using MA27 as a linear solver). We compare the results to IPOPT's interface to the AMPL Solver Library (ASL). From the 111 nonlinear problems successfully solved, out of a total of 135, we select the problems where the iterates are identical for CasADi and ASL – a different number in the iterates can often be explained by the finite precision floating point arithmetics. In Table 6.2, we present the ten problems where IPOPT reports more than 80 ms solution time for either CasADi or ASL. All calculations have been performed on a Dell Latitude E6400 laptop with an Intel Core Duo processor (only one core was used) of 2.4 GHz, with 4 GB of RAM, 3072 KB of L2 Cache and 128 kB of L1 cache, running Ubuntu 10.4.

In the table, we show the total solution time as reported by IPOPT and the part of this time actually spent for automatic differentiation in ASL or CasADi. Most of the remaining time is spent in the linear solver. For eight of the ten benchmarks, the time spent in CasADi is about half or less than that of ASL. Note that there is no compiler in the loop for neither CasADi nor ASL. Significantly faster function evaluations would indeed be possible through dynamic compilation as described in Section 6.6, though, as seen in the table, this does not necessarily result in significantly faster NLP solution as a large part of the solution time is spent inside the linear solver.

Table 6.2: Benchmarking against AMPL Solver Library (ASL). The total solution time as well as the time spent by the AD framework (i.e. NLP callback functions) are displayed.

Benchmark	NLP dimensions		Iterations	Time ASL [s]		Time CasADi [s]		Change
	Variables	Constraints		Total	AD	Total	AD	AD part
gpp	250	498	22	0.492	0.272	0.500	0.264	-3 %
reading1	10001	5000	22	0.712	0.408	0.306	0.104	-76 %
porous2	4900	4900	8	1.916	0.188	1.736	0.036	-81 %
orthrgds	10003	5000	16	0.949	0.568	0.512	0.164	-71 %
clnlbeam	1499	1000	205	0.776	0.184	0.784	0.184	0 %
svanberg	5000	5000	30	2.492	0.520	2.300	0.272	-48 %
orthregd	10003	5000	6	0.332	0.208	0.160	0.060	-71 %
trainh	20000	10002	69	3.932	1.984	2.804	0.896	-55 %
orthrgdm	10003	5000	6	0.328	0.208	0.156	0.068	-67 %
dtoc2	5994	3996	10	0.296	0.124	0.224	0.048	-61 %

6.8 Conclusion and outlook

As we have seen in this chapter, there is no inherent contradiction between working in a high-level environment such as Python and evaluation at the speed of optimized C-code. We also showed that using a combination of two virtual machines, AD can be implemented in a manner that minimizes computational and memory overhead while at the same time supporting high level operations. We also discussed different ways of speeding up the numerical calculations via code generation.

We end this chapter by listing some future plans for the tool:

- It is the intention to continue to evolve the symbolic core until it essentially becomes a *turing-complete functional programming language* [185, 189]. This includes adding high-level operations such as *folds* [184] for flow control as well as *monads* [186] for user interaction and third-party interfaces.
- The strict depth-first ordering of the expression graph means that the corresponding functions, $F(x_1, \ldots, x_n) \rightarrow y_1, \ldots, y_m$, can be partially evaluated when only a subset, $\{y_1, \ldots, y_i\}, i < m$, of the outputs are needed. This feature is important to limit the memory use for AD by source code transformation.
- CasADi contains some support for parallel evaluation of expressions on shared memory architectures, e.g. integrators in a multiple shooting context. This implementation uses OpenMP. Extensions include support for distributed memory architectures using MPI and graphics processing units using OpenCL [121].

Chapter 7

Automatic sensitivity analysis for ODE and DAE

In the presentation of the forward and reverse mode differentiation of algorithms containing implicit functions in Section 3.9, we saw that derivative propagation rules could be formulated as augmented root-finding problems with the same structure. This is important as it allows the calculation of second and higher order derivatives of such algorithms in a source-code-transformation context. The same did however not hold true for the discrete-time or continuous-time integrators of Sections 3.10 and 3.11, since the reverse mode resulted in a two-point boundary-value problem or its discrete-time equivalent, as opposed to an initial-value problem.

In this chapter, we propose an extension of the discrete-time and continuous-time integrator formulations that overcomes this limitation. Using this extension, we can treat an ODE or DAE integrator as any other atomic operation in a symbolic expression, and in particular efficiently calculate derivative information to arbitrary order.

We present the extension in a discrete-time setting in Section 7.1 and thereafter in continuous-time in Section 7.2. In Section 7.3, we discuss the implementation of the approach in CasADi and in Section 7.4, we illustrate it with an example. In Section 7.5, we discuss future extensions.

7.1 An extended discrete-time integrator

Let us augment the discrete-time dynamics of Algorithm 3.5 with a second dynamic equation that depends point-wise on the first but goes backward rather than forward in time. The result is shown in Algorithm 7.1, where we for simplicity of the presentation have left out the summation state q and the parameter u.

Algorithm 7.1 Definition of extended $F : \mathbb{R}^{n_x} \times \mathbb{R}^{n_r} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_r}$

input (x_0, r_N)
for $k = 0, \ldots, N-1$ **do**
  $x_{k+1} = \Phi(x_k)$
end for
for $k = N-1, \ldots, 0$ **do**
  $r_k = \Phi^*(x_k, r_{k+1})$
end for
return (x_N, r_0)

The appended "backward dynamics" described by $\Phi^*(x, r)$ can be, but need not be, related to the adjoint directional derivatives of $\Phi(x)$. Applying forward mode AD to Algorithm 7.1, as described in Section 3.2 results in Algorithm 7.2.

Algorithm 7.2 Definition of extended $\hat{F}$:
$\mathbb{R}^{n_x} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_r \times m} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_r \times m}$

input $(x_0, r_N, \hat{X}_0, \hat{R}_N)$
for $k = 0, \ldots, N-1$ **do**
  $x_{k+1} = \Phi(x_k)$
  $\hat{X}_{k+1} = \frac{\partial \Phi}{\partial x}(x_k)\, \hat{X}_k$
end for
for $k = N-1, \ldots, 0$ **do**
  $r_k = \Phi^*(x_k, r_{k+1})$
  $\hat{R}_{k+1} = \frac{\partial \Phi^*}{\partial x}(x_k, r_{k+1})\, \hat{X}_k + \frac{\partial \Phi^*}{\partial r}(x_k, r_{k+1})\, \hat{R}_{k+1}$
end for
return $(x_N, r_0, \hat{X}_N, \hat{R}_0)$

This algorithm has the same structure as Algorithm 7.1, if we define the augmented (forward) state $\tilde{x}_k := (x_k, \mathrm{vec}(\hat{X})_k)$, $k = 0, \ldots, N-1$ and the augmented backward state $\tilde{r}_k := (r_k, \mathrm{vec}(\hat{R})_k)$, $k = N-1, \ldots, 0$. The

augmented dynamic equations can be efficiently calculated by applying forward mode AD to $\Phi(x)$ and $\Phi^*(x, r)$.

Similarly, applying reverse mode AD to Algorithm 7.1, as described in Section 3.3, results in Algorithm 7.3.

Algorithm 7.3 Definition of extended $\bar{F}$:
$\mathbb{R}^{n_x} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_r \times m} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_r \times m}$

input $(x_0, r_N, \bar{X}_N, \bar{R}_0)$
for $k = 0, \ldots, N-1$ **do**
  $x_{k+1} = \Phi(x_k)$
end for
for $k = N-1, \ldots, 0$ **do**
  $r_k = \Phi^*(x_k, r_{k+1})$
end for
for $k = 0, \ldots, N-1$ **do**
  $\bar{R}_{k+1} = \left[\frac{\partial \Phi^*}{\partial r}(x_k, r_{k+1})\right]^\intercal \bar{R}_k$
end for
for $k = N-1, \ldots, 0$ **do**
  $\bar{X}_k = \left[\frac{\partial \Phi}{\partial x}(x_k, r_{k+1})\right]^\intercal \bar{X}_{k+1} + \left[\frac{\partial \Phi^*}{\partial x}(x_k, r_{k+1})\right]^\intercal \bar{R}_k$
end for
return $(x_N, r_0, \bar{X}_0, \bar{R}_N)$

We now restrict $\Phi^*(x, r)$ to be an affine function in r, though not necessarily in x, i.e.:

$$\frac{\partial \Phi^*}{\partial r}(x, z) = \frac{\partial \Phi^*}{\partial r}(x, 0) = \frac{\partial \Phi^*}{\partial r}(x). \tag{7.1}$$

We note that this requirement will in particular be fulfilled whenever $\Phi^*(x, r)$ is the result of a reverse mode differentiation as in Section 3.10. This allows us to rewrite Algorithm 7.3 as shown in Algorithm 7.4.

Algorithm 7.4 Definition (after restriction) of extended $\bar{F}$:
$\mathbb{R}^{n_x} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_r \times m} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_r \times m}$

input $(x_0, r_N, \bar{X}_N, \bar{R}_0)$
for $k = 0, \ldots, N-1$ **do**
  $x_{k+1} = \Phi(x_k)$
  $\bar{R}_{k+1} = \left[\frac{\partial \Phi^*}{\partial r}(x_k)\right]^\mathsf{T} \bar{R}_k$
end for
for $k = N-1, \ldots, 0$ **do**
  $r_k = \Phi^*(x_k, r_{k+1})$
  $\bar{X}_k = \left[\frac{\partial \Phi}{\partial x}(x_k, r_{k+1})\right]^\mathsf{T} \bar{X}_{k+1} + \left[\frac{\partial \Phi^*}{\partial x}(x_k, r_{k+1})\right]^\mathsf{T} \bar{R}_k$
end for
return $(x_N, r_0, \bar{X}_0, \bar{R}_N)$

After the simplification, the algorithm for reverse mode differentiation has indeed the same structure as Algorithm 7.1, if we define the augmented forward state $\tilde{x}_k := (x_k, \mathrm{vec}(\bar{R})_k)$, $k = 0, \ldots, N-1$ and the augmented backward state $\tilde{r}_k := (r_k, \mathrm{vec}(\bar{X})_k)$, $k = N-1, \ldots, 0$. The augmented dynamic equations, which clearly satisfy (7.1), can be efficiently calculated by applying reverse mode AD to $\Phi(x)$ and $\Phi^*(x, r)$.

Generalization to parametric integrators with summation states

The above extended discrete-time integrator formulation naturally generalizes to explicit and implicit parametric integrator formulations with summation states. We show the implicit version in Algorithm 7.5, where we require that $\Theta^*(x, z, u, r, s, v)$, $\Phi^*(x, z, u, r, s, v)$ and $\Psi^*(x, z, u, r, s, v)$ are affine in r, s and v, i.e.:

$$\begin{aligned}
\frac{\partial \Phi^*}{\partial r}(x, z, u, r, s, v) &= \frac{\partial \Phi^*}{\partial r}(x, z, u, 0, 0, 0) = \frac{\partial \Phi^*}{\partial r}(x, z, u), \\
\frac{\partial \Phi^*}{\partial s}(x, z, u, r, s, v) &= \frac{\partial \Phi^*}{\partial s}(x, z, u, 0, 0, 0) = \frac{\partial \Phi^*}{\partial s}(x, z, u), \\
\frac{\partial \Phi^*}{\partial v}(x, z, u, r, s, v) &= \frac{\partial \Phi^*}{\partial v}(x, z, u, 0, 0, 0) = \frac{\partial \Phi^*}{\partial v}(x, z, u),
\end{aligned} \tag{7.2}$$

and equivalently for $\Theta^*(x, z, u, r, s, v)$ and $\Psi^*(x, z, u, r, s, v)$.

Algorithm 7.5 Definition of proposed F :
$\mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_v} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_p}$

input (x_0, u, r_N, v)
$q_0 = 0$
for $k = 0, \ldots, N-1$ **do**
  $0 = \Theta(x_k, z_k, u)$ ▷ Solve for z_k
  $x_{k+1} = \Phi(x_k, z_k, u)$
  $q_{k+1} = \Psi(x_k, z_k, u) + q_k$
end for
$p_N = 0$
for $k = N-1, \ldots, 0$ **do**
  $0 = \Theta^*(x_k, z_k, u, r_{k+1}, s_k, v)$ ▷ Solve for s_k
  $r_k = \Phi^*(x_k, z_k, u, r_{k+1}, s_k, v)$
  $p_k = \Psi^*(x_k, z_k, u, r_{k+1}, s_k, v) + p_{k+1}$
end for
return (x_N, q_N, r_0, p_0)

We propose Algorithm 7.5 as a standard formulation that efficiently handles a large class of problems originating from optimal control. Note in particular that it can be used to efficiently implement the AD approach to (continuous-time) sensitivity analysis presented in Section 3.11. The forward and reverse mode derivative propagation rules for Algorithm 7.5 can be found in Appendix B.1.

7.2 An extended continuous-time integrator

The approach of the previous section extends to two-point boundary value problems of the form:

$$\begin{aligned} &f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_r} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_r}, \quad (x_0, r_1) \mapsto (x_1, r_0) \\ &\begin{cases} \dot{x}(t) &= \phi(x(t)), \\ -\dot{r}(t) &= \phi^*(x(t), r(t)), \end{cases} \qquad t \in [0,1], \\ &x(0) = x_0, \quad r(1) = r_1, \quad x_1 = x(1), \quad r_0 = r(0), \end{aligned} \tag{7.3}$$

with the affinity restriction on the backward dynamics, cf. (7.1):

$$\frac{\partial \phi^*}{\partial r}(x, z) = \frac{\partial \phi^*}{\partial r}(x, 0) = \frac{\partial \phi^*}{\partial r}(x). \tag{7.4}$$

By defining $\Phi(x) := x + \frac{1}{N}\,\phi(x)$ and $\Phi^*(x, r) := r + \frac{1}{N}\,\phi^*(x, r)$, we interpret Algorithm 7.1 as the explicit Euler method for calculating (7.3). This gives (cf.

Section 3.11) the following equation for the forward derivatives of (7.3):

$$
\begin{aligned}
&\hat{f} : \mathbb{R}^{n_x} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_r \times m} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_r \times m},\\
&\quad (x_0, r_1, \hat{X}_0, \hat{R}_1) \mapsto (x_1, r_0, \hat{X}_1, \hat{R}_0)\\
&\begin{cases}
\dot{x}(t) &= \phi(x(t)),\\
\dot{\hat{X}}(t) &= \frac{\partial \phi}{\partial x}(x(t))\, \hat{X}(t)\\
-\dot{r}(t) &= \phi^*(x(t), r(t)),\\
-\dot{\hat{R}}(t) &= \frac{\partial \phi^*}{\partial x}(x(t), r(t))\, \hat{X}(t) + \frac{\partial \phi^*}{\partial r}(x(t))\, \hat{R}(t),
\end{cases} \qquad t \in [0,1],\\
&x(0) = x_0, \quad r(1) = r_1, \quad x_1 = x(1), \quad r_0 = r(0),\\
&\hat{X}(0) = \hat{X}_0, \quad \hat{R}(1) = \hat{R}_1, \quad \hat{X}_1 = \hat{X}(1), \quad \hat{R}_0 = \hat{R}(0),
\end{aligned}
\tag{7.5}
$$

which can be efficiently and automatically formulated using forward mode AD.

We also get the following equation for the adjoint derivatives of (7.3):

$$
\begin{aligned}
&\bar{f} : \mathbb{R}^{n_x} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_r \times m} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_r \times m},\\
&\quad (x_0, r_1, \bar{X}_1, \bar{R}_0) \mapsto (x_1, r_0, \bar{X}_0, \bar{R}_1)\\
&\begin{cases}
\dot{x}(t) &= \phi(x(t)),\\
\dot{\bar{R}}(t) &= \left[\frac{\partial \phi^*}{\partial r}(x(t))\right]^{\mathsf{T}} \bar{R}(t) \qquad t \in [0,1],\\
-\dot{r}(t) &= \phi^*(x(t), r(t)),\\
-\dot{\bar{X}}(t) &= \left[\frac{\partial \phi}{\partial x}(x(t), r(t))\right]^{\mathsf{T}} \bar{X}(t) + \left[\frac{\partial \phi^*}{\partial x}(x(t), r(t))\right]^{\mathsf{T}} \bar{R}(t),
\end{cases}\\
&x(0) = x_0, \quad r(1) = r_1, \quad x_1 = x(1), \quad r_0 = r(0),\\
&\bar{R}(0) = \bar{R}_0, \quad \bar{X}(1) = \bar{X}_1, \quad \bar{R}_1 = \bar{R}(1), \quad \bar{X}_0 = \bar{X}(0),
\end{aligned}
\tag{7.6}
$$

which can be efficiently and automatically formulated using reverse mode AD. Both (7.5) and (7.6) have the same structure as (7.3), allowing them to be used as an atomic operation in a source-code-transformation setting.

Generalization to parametric integrators with quadrature states

The above extended continuous-time integrator formulation also generalizes and to explicit and implicit parametric integrator formulations with quadrature

states. In the implicit case, we get:

$$
\begin{aligned}
&f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_v} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_p}, \\
&\quad (x_0, u, r_1, v) \mapsto (x_1, q_1, r_0, p_0) \\
&\left\{ \begin{array}{rcll}
\dot{x}(t) &=& \phi(x(t), z(t), u), & \\
0 &=& \theta(x(t), z(t), u), & \\
\dot{q}(t) &=& \psi(x(t), z(t), u), & \\
-\dot{r}(t) &=& \phi^*(x(t), z(t), u, r(t), s(t), v), & t \in [0, 1], \\
0 &=& \theta^*(x(t), z(t), u, r(t), s(t), v), & \\
-\dot{p}(t) &=& \psi^*(x(t), z(t), u, r(t), s(t), v), &
\end{array} \right. \\
&x(0) = x_0, \quad r(1) = r_1, \quad q(0) = 0, \quad p(1) = 0, \\
&x_1 = x(1), \quad q_1 = q(1), \quad r_0 = r(0), \quad p_0 = p(0),
\end{aligned}
\tag{7.7}
$$

where we require that $\theta^*(x, z, u, r, s, v)$, $\phi^*(x, z, u, r, s, v)$ and $\psi^*(x, z, u, r, s, v)$ are affine in r, s and v, cf. (7.2).

As its discrete-time counterpart in the previous section, we propose (7.7) as a standard formulation that efficiently handles a large class of problems originating from optimal control, but here using the variational approach to sensitivity analysis presented in Section 3.11. The forward and reverse mode derivative propagation rules for (7.7) can be found in Appendix B.2.

7.3 Implementation in CasADi

The approach presented in Section 7.2 has been implemented in CasADi, relying on AD by source-code-transformation to automatically generate augmented problems following the rules listed in Appendix B.2.

This enables fully automatic sensitivity analysis to arbitrary order, using any combination of forward and adjoint differentiation. In particular, it can be used to efficiently calculate first and second order derivative information for direct or indirect shooting-based optimal control methods.

The implementation has been made in CasADi's integrator base class, making the functionality available to all integrators. In addition to integrators written in CasADi, this includes the SUNDIALS solvers, both CVODES for explicit ODEs and IDAS for DAEs [113]. CasADi's interfaces to CVODES and IDAS automatically generate all derivative information requested by the solvers, which can then be used in the direct or iterative linear solvers inside CVODES

and IDAS. In addition, linear solvers interfaced to CasADi, in particular sparse direct solvers such as CSparse [68], can be used by SUNDIALS solvers, either directly in the Newton solver, or as a preconditioner for the iterative linear solvers. The SUNDIALS solvers have been written to handle large-scale ODEs and DAEs.

7.4 Usage example

We illustrate the automatic sensitivity analysis by considering initial-value problem in the matrix-valued ODE $\dot{X}(t) = X^{-1}$, with $X(t) \in \mathbb{R}^{n\times n}$ and $t \in [0,1]$:

$$
\begin{aligned}
&f : \mathbb{R}^{n\times n} \to \mathbb{R}^{n\times n} \times \mathbb{R} \quad X_0 \mapsto (X_1, q_1)\\
&\begin{cases} \dot{X}(t) &= Z(t), \\ 0 &= Z(t)\,X(t) - I, \qquad t \in [0,1], \\ \dot{q}(t) &= \|Z(t)\|_{\mathrm{F}}, \end{cases} \qquad (7.8)\\
&X(0) = X_0, \quad q(0) = 0, \quad X_1 = X(1), \quad q_1 = q(1).
\end{aligned}
$$

Choosing $n = 3$, this problem can be coded in CasADi-Python as follows:

```
from casadi import *
n = 3
X = ssym("X",n,n)
Z = ssym("Z",n,n)
arg = [X, Z, None]
rhs = [Z, mul(X,Z)-DMatrix.eye(n), norm_F(Z)]
dae = SXFunction(arg,rhs)
f = IdasIntegrator(dae)
f.init()
```

This code starts by creating two matrix-valued symbolic variables for X and Z and then uses these variables to define a function object with three inputs and three outputs. The inputs correspond to state, algebraic variable and parameter (here absent) and the outputs to the differential, algebraic and quadrature right-hand-sides. Here we use CasADi's matrix expression graph but by replacing `msym` with `ssym` and `MXFunction` with `SXFunction` this would be valid syntax for the scalar expression graph and operations would be expanded in scalar operations.

This function object is then used to create a new function object using CasADi's interface to IDAS from the SUNDIALS suite [113]. The IDAS function object, in turn, has the four inputs and four outputs as shown in Equation (7.7).

Next, we create a function object corresponding to the gradient of the (scalar-valued) quadrature state at the final time with respect to all inputs. This function object is created using the adjoint mode by source code transformation as follows:

```
grad_f = f.gradient("x0","qf")
grad_f.init()
```

Internally, this will invoke the reverse mode derivative propagation rule as stated in Appendix B.2.

Finally we evaluate this gradient function numerically, while at the same time calculating one forward mode directional derivative (i.e. $m = 1$ in Section 3.2) using the operator overloading approach. Internally, the forward mode derivative propagation rule as stated in Appendix B.2 is invoked. The syntax for this is, cf. [28]:

```
X0 = DMatrix.eye(n) + 1
X0_hat = DMatrix.zeros(n,n)
X0_hat[0,0] = 1
grad_f.setInput(X0,"x0")
grad_f.setFwdSeed(X0_hat,"x0")
grad_f.evaluate(1,0)
print grad_f.output("grad")
print grad_f.fwdSens("grad")
```

where we have used the following numerical values and directional derivative seed:

$$X_0 = \begin{bmatrix} 3 & 1 & 1 \\ 1 & 3 & 1 \\ 1 & 1 & 3 \end{bmatrix}, \qquad \hat{X}_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

That is, we perturb the upper left element of X_0 and see how this influences $\nabla_{X_0} q_1(X_0)$. This corresponds to an *forward-over-adjoint* sensitivity analysis [152] and returns:

$$\nabla_{x_0} q_1 = \begin{bmatrix} -0.086406 & 0.037633 & 0.037633 \\ 0.037633 & -0.086406 & 0.037633 \\ 0.037633 & 0.037633 & -0.086406 \end{bmatrix},$$

$$\widehat{\nabla_{x_0} q_1} = \begin{bmatrix} 0.069795 & -0.022014 & -0.022014 \\ -0.022014 & -0.003099 & 0.013247 \\ -0.022014 & 0.013247 & -0.003099 \end{bmatrix}.$$

7.5 Conclusion and outlook

In this chapter, we have proposed to extend the standard ODE/DAE integrator formulation to also include a coupled problem which depends on the first but is integrator backward in time. Using this as a standard formulation, arbitrary order sensitivity analysis can be calculated efficiently and automatically using any combination of forward or adjoint sensitivity analysis.

We conclude with a discussion on some topics that have not yet been implemented in code, but are relevant for efficiently solving large-scale optimal control problems.

Discrete-time formulation

As discussed in Section 3.11, the AD approach to sensitivity analysis results in the type of discrete-time initial-value problems treated in Sections 3.10 and 7.1. While this kind of problem can be readily expanded into its constituting operations, possibly including implicit functions as in Section 3.9, it makes sense to handle it by a dedicated atomic operation. In particular, it would enable exploiting the similarity between subsequent (discrete) time points.

Sparsity pattern propagation

In our implementation, we have so far assumed that all states depend on all other states. Certain dynamic systems, for example weakly coupled dynamic systems such as river networks [163], will give to non-dense sparsity patterns, which can be exploited. Such structure also arises from differentiating. Rules for sparsity pattern propagation can be formulated by studying the bipartite graph defined by the sparsity patterns of the Jacobians of ϕ, θ, ψ, ϕ^*, θ^* and ψ^* in (7.7) with respect to x, z, u, r, s and v.

Fully implicit DAEs

In this presentation, and in code, we have so far assumed that the DAE be in semi-explicit form and of index-1. This in particular means that DAEs given in fully-implicit form, e.g. originating from physical modeling languages such as Modelica (cf. Section 6.2), need to be reformulated before they can be used by the integrators. This reformulation can be done by symbolically or numerically solving for the differential state derivatives, by adding dummy

algebraic variables or by using an index-reduction algorithm such as Pantelides Algorithm [153]. For more detail on such reformulations, we refer to Cellier and Kofman [59] or Fritzson [82].

An alternative to reformulation is to extend the automatic sensitivity analysis presented here to fully implicit DAEs of index-1, see e.g. [57].

Structure exploitation

The augmented DAE that needs to be solved for both forward and adjoint sensitivity analysis, has special structure that can be exploited by the linear solver. In current implementation, structure exploitation is only partly supported. This is particularly important when calculating complete Jacobians and Hessians of expressions with embedded integrators.

Multi-point integrator outputs

In (3.27), we considered only two time points, the initial time $t = 0$ and the end time $t = 1$. This means that if we are interested in the output at a set of points, we need to call the integrator repeatedly. In many situations, e.g. in parameter estimation or if we wish to enforce path constraints at a finer grid than the control discretization in direct multiple shooting, it is beneficial to have the integrator output the state at a set of points.

Chapter 8

Towards a structure-exploiting nonlinear programming solver

In this chapter we report progress towards a code-generating NLP solver that goes under the name *SCPgen*. SCPgen currently implements the lifted Newton method for SQP as explained in Section 4.5. In contrast to the only other implementation of lifted Newton at the time of writing, the LiftOpt package [23] which relies on the operator overloading AD tool ADOL-C, SCPgen uses AD by source code transformation in CasADi to generate symbolic expressions for the mapping to and from a reduced-space QP subproblem. These symbolic expressions can then either be evaluated in CasADi's virtual machines or be code-generated as C-code.

The goal is to extend SCPgen to become a parametric sequential convex programming (SCP) solver generator, cf. Section 4.2, which can generate structure-exploiting SCP solvers as self-contained C-code.

In the following, we present the usage and implementation of the solver in Sections 8.1 and 8.2. We then demonstrate the effectiveness of the approach by solving a large-scale parameter estimation problem in Section 8.3. We end the chapter with an outlook for the tool, which at the time of writing was still under development, in Section 8.4.

8.1 Syntax and usage

As explained in Chapters 6 and 7 and demonstrated in Appendix A.3, CasADi provides a very compact way of transcribing an OCP using a direct single shooting approach. To allow a user to adopt such a code to a multiple shooting transcription, we extend the CasADi symbolic environment with an operator *lift*:

$$\textit{lifted expression} = \text{lift}(\textit{expression to lift}, \textit{expression for the initial guess})$$

This operator allows the user to instruct a built-in or interfaced solver to treat a particular node in the computational graph as an equality constraint rather than an assignment. In addition to the node to be lifted, it also takes a second argument corresponding to an expression for the initial guess of the lifted variable. This expression can e.g. be a measurement value, a simulation or a parameter in the NLP solver.

The detection and reformulation of expressions containing *lift* instructions has been implemented in SCPgen. To the user, SCPgen has the same syntax as a solver for solving the reduced space NLP, while internally solving the full space NLP. In particular, this allows a single shooting transcription like that presented in Appendix A.3, to be modified into a multiple shooting transcription by adding a single line of code.

8.2 Implementation

SCPgen implements the lifted Newton method for SQP as explained in Section 4.5, generalized to support parametric NLPs – cf. (4.3) – and to allow nonlinear bound constraints and a nonlinear objective function. It supports both exact second derivatives or a Gauss-Newton Hessian approximation. As a globalization strategy, it currently implements a line-search algorithm based on an ℓ_1 merit function. If no expressions are marked using the *lift* operator introduced above, it behaves like a standard, sparsity-exploiting parametric SQP code.

SCPgen is distributed along with CasADi and requires no additional dependencies. As we shall see in the following, it relies heavily on CasADi's ability to do AD by source code transformation and, for best performance, C-code generation.

Symbolic preprocessing

During the initialization of the NLP solver, the user-provided NLP is first decomposed into its defining symbolic expressions. By traversing the NLP expression graph and identifying all *lift* instructions, two functions are generated symbolically:

- a lifted parametric NLP
- a function that calculates an initial guess for the lifted variables

The latter is a function with the structure $f_{\text{guess}} : \mathbb{R}^{n_u} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_x}$ that takes the reduced space variables $u = x_0 \in \mathbb{R}^{n_u}$ (cf. Section 4.5) and parameters $p \in \mathbb{R}^{n_p}$ and calculates the initial guess provided as the second argument to the *lift* instruction.

The lifted parametric NLP is essentially a parametric version of (4.9):

$$\begin{aligned} \underset{x,\,p}{\text{minimize}} \quad & f(x,p) \\ \text{subject to} \quad & \phi_1(x_0,p) - x_1 = 0, \\ & \phi_2(x_0,x_1,p) - x_2 = 0, \\ & \quad\vdots \\ & \phi_N(x_0,\ldots,x_{N-1},p) - x_N = 0, \\ & p = \overline{p}, \quad \underline{x} \le x \le \overline{x}, \quad \underline{h} \le h(x,p) \le \overline{h}. \end{aligned} \tag{8.1}$$

The steps presented in Section 4.5 are then performed on (8.1), generalized to the allow nonlinear bounds and a nonlinear objective function as well as the parameter p, i.e.:

- Symbolically creating the function $z(u,d,p)$, a generalization of $z(u,d)$ defined by Algorithm 4.2.
- Generating symbolic expressions for (the generalized) a and $A\,v$, for an unknown vector v using the forward mode of AD by SCT as shown in (4.17) and, with $V = v$, (4.21a).
- Using the reverse mode of AD by SCT to generate new expressions corresponding to the gradient of the Lagrangian $\nabla_x \mathcal{L}(x,\lambda_x,\lambda_h,\nu)$, cf. (4.18). As a by product of this calculation, we also get expressions for calculating the parametric sensitivities, cf. (4.4).

- Generating expressions for w and W' using the forward mode of AD by SCT as shown in (4.18) and (4.21b).
- Generating expressions for (the generalized) b and $B\,v$ using the reverse mode of AD by SCT as shown in (4.20) and (4.21c).
- Generating expressions for the potentially sparse matrices A and B using the standard routines for Jacobian calculation by SCT. For B, symmetry is exploited.
- Generating an expression for calculating the full space step Δx given the reduced space solution, cf. (4.22), using forward mode AD by SCT.
- Generating an expression for calculating the new multipliers $\nu^{(k+1)}$ given the reduced space solution, cf. (4.23), using reverse mode AD by SCT.

We refer to Albersmayer and Diehl [23] for a discussion on how to handle the Gauss-Newton case.

The final result of the symbolic preprocessing is five function objects for calculating

1. an initial guess for the lifted variables,
2. the matrices of the condensed QP,
3. the vectors of the condensed QP,
4. the primal-dual solution of the full space QP given the solution to the condensed QP as well as
5. the parametric sensitivities as well as the residual in the KKT conditions.

Code-generation

The five functions resulting from the symbolic preprocessing represent, along with the QP solution, the vast majority of the computational effort in the SQP algorithm. This effort can be made up to an order of magnitude smaller by generating C-code for the functions as explained in Section 6.6.

SQP main loop

The remainder of the code is basic SQP algorithm with a line-search algorithm based on the ℓ_1 merit function. This part of the code is currently not code generated.

8.3 Example: Parameter estimation for the shallow-water equations

To assess the performance of the code, we consider the following two-dimensional hyperbolic PDE describing the propagation of water waves in a basin [23, 188]:

$$\left\{\begin{array}{rcl} \dfrac{\partial u}{\partial t}(t,x,y) & = & g\,\dfrac{\partial h}{\partial x}(t,x,y) - p_1\,u(t,x,y), \\ \dfrac{\partial v}{\partial t}(t,x,y) & = & g\,\dfrac{\partial h}{\partial y}(t,x,y) - p_1\,v(t,x,y), \\ \dfrac{\partial h}{\partial t}(t,x,y) & = & -p_2\left(\dfrac{\partial h}{\partial x}(t,x,y) + \dfrac{\partial h}{\partial y}(t,x,y)\right), \end{array}\right. \tag{8.2}$$

for $t \in [0,T]$ and $(x,y) \in \Omega$. We see how velocity (u,v) and height h of the water surface evolve in time t and space (x,y). In addition to standard gravity g, the model has the parameters p_1 corresponding to viscous drag and p_2 corresponding to mean height.

As a spatial domain, we choose a quadratic region with side length $L = 0.02$ m, i.e. $\Omega = [0,L] \times [0,L]$. We assume that the walls are reflecting, i.e. u and v are zero at the boundary of Ω.

As initial conditions we take:

$$\left\{\begin{array}{rcl} u(0,x,y) & = & 0, \\ v(0,x,y) & = & 0, \\ h(0,x,y) & = & \left\{\begin{array}{ll} h_{\mathrm{sp}}\,\cos\left(3\,\pi\,d(x,y)/2\,r_{\mathrm{sp}}\right) & \text{if } d(x,y) \le r_{\mathrm{sp}}/3, \\ 0 & \text{otherwise,} \end{array}\right. \end{array}\right. \tag{8.3}$$

with $d(x,y) = \sqrt{(x-x_{\mathrm{sp}})^2 + (y-y_{\mathrm{sp}})^2}$. This models a water drop with radius r_{sp} and height h_{sp} falling to an otherwise calm surface. It hits the surface at the point $(x_{\mathrm{sp}}, y_{\mathrm{sp}})$ in the (x,y) plane. In our calculations, we shall use $r_{\mathrm{sp}} = 0.03$ m, $h_{\mathrm{sp}} = 0.01$ m, $x_{\mathrm{sp}} = y_{\mathrm{sp}} = 0.04$ m.

In Figure 8.1, we illustrate the dynamic model (8.2) with initial conditions given by (8.3).

To solve the problem, we use a uniform discretization in space with n-by-n grid points. We use $U(t) \in \mathbb{R}^{n+1\times n}$, $V(t) \in \mathbb{R}^{n\times n+1}$ and $H(t) \in \mathbb{R}^{n\times n}$ to denote the corresponding "meshed" state variables. This allows us to calculate the

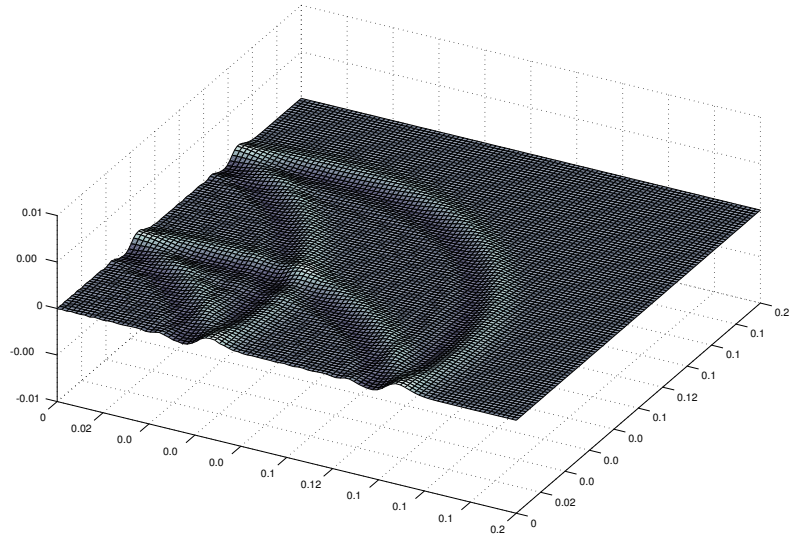

Figure 8.1: The shallow water equations; Snapshot of the water basin at $t = 370$ ms. For the simulation, $p_1 = 4$ s^{-1} and $p_2 = 0.01$ m were used.

following discrete-time dynamics of an interval $\Delta t = 0.01$ s:

$$
\begin{aligned}
&F : \mathbb{R}^{n+1\times n} \times \mathbb{R}^{n\times n+1} \times \mathbb{R}^{n\times n} \times \mathbb{R}^2 \to \mathbb{R}^{n+1\times n} \times \mathbb{R}^{n\times n+1} \times \mathbb{R}^{n\times n} \\
&\qquad (U_1, V_1, H_1, p) \mapsto (U_2, V_2, H_2) \\
&\begin{cases}
\text{PDE (8.2)}, & & & \\
p = (p_1, p_2), & & & \\
U(t_1) = U_1, & V(t_1) = V_1, & H(t_1) = H_1, & \\
U_2 = U(t_2), & V_2 = V(t_2), & H_2 = H(t_2). &
\end{cases}
\end{aligned}
\tag{8.4}
$$

As in Albersmayer and Diehl [23], we discretize time using a fixed-step leapfrog integration scheme with 100 time steps of length 0.0001 s. This results in an explicit discrete-time one-step integrator scheme as encountered in Sections 3.10 and 7.1. The integrator scheme can be efficiently implemented using only high level matrix operations in CasADi, giving a symbolic representation of the NLP with only matrix-valued operations.

This NLP can be written compactly as:

$$\begin{array}{ll} \underset{U,V,H,p}{\text{minimize}} & \displaystyle\sum_{k=1}^{100} \frac{1}{2}\|H_k - \overline{H}_k\|_{\mathrm{F}}^2 \\ \text{subject to} & U_0 = 0, \quad V_0 = 0, \quad H_0 = \overline{H}_0, \\ & (U_{k+1}, V_{k+1}, H_{k+1}) = F(U_k, V_k, H_k, p), \quad k = 0, \ldots, 99 \\ & \underline{p} \leq p \leq \overline{p}, \\ & \underline{H}_k \geq 0, \quad k = 1, \ldots 100, \end{array} \tag{8.5}$$

where $\overline{H}_0$ is the discretized initial condition for $H(t)$ given from Equation (8.3) and $\overline{H}_k, k = 1, \ldots, 100$ are given measurement values for the $H(t)$. We simulate measurement values in advance by adding noise to a forward simulation with $p_1 = 2$ s^{-1} and $p_2 = 0.01$ m.

We solve (8.5) using SCPgen in two ways. First, we solve it using single shooting, i.e. by sequentially eliminating the state trajectory, leaving only p are free parameters. Secondly we solve it using multiple shooting, by *lifting* U_k, V_k and H_k at all 100 intervals. We note at this point at in Albersmeyer and Diehl [23], only H_k is lifted. Another important difference compared to Albersmeyer and Diehl is that we use dual feasibility as opposed to step size as a stopping criterion in Gauss-Newton SQP.

The results for a 30-by-30 grid and using a Gauss-Newton Hessian approximation is shown in Table 8.1. We use a forward simulation to initialize the lifted variables. Note that in the lifted case, this corresponds to an NLP with 274142 variables and 364140 constraints. The full-space Hessian, never formed explicitly, has a block diagonal structure consisting of 100 *dense* blocks of 30200-by-30200 variables. This means that there are around 10^{11} nonzeros in the Hessian making it unsuitable to solve with general-purpose sparsity-exploiting QP solvers.

The results in Table 8.1 largely agree with those reported in [23], but the time per iteration is of the order four times shorter compared to LiftOpt. This can to a large extent be attributed to the code generation. In Table 8.2, we see the same table, but using an exact Hessian.

Using an exact Hessian, the solution times are around 5 times greater than for Gauss-Newton and the memory overhead is larger.

A spatial discretization of 30-by-30 points is a very crude discretization for the problem at hand. In Tables 8.3 and 8.4, we show the results for a finer grid

Table 8.1: Parameter estimation for the 2D shallow water equations with Gauss-Newton Hessian, 30-by-30 grid points

drag	depth	iter ss	time ss	iter ms	time ms
0.5	0.01	5	24.56	4	33.92
5	0.01	5	23.82	4	34.01
15	0.01	7	32.31	6	47.74
30	0.01	10	45.51	7	55.15
2	0.005	17	73.74	6	47.87
2	0.02	17	74.98	6	49.44
2	0.1	23	98.31	20	147.6
0.2	0.001	18	79.41	9	69.44
1	0.005	14	62.38	6	47.83
4	0.02	14	62.3	6	47.98
1	0.02	16	70.32	6	48.09
20	0.001	16	70.51	6	48.46

Table 8.2: Parameter estimation for the 2D shallow water equations with exact Hessian, 30-by-30 grid points

drag	depth	iter ss	time ss	iter ms	time ms
0.5	0.01	6	85.89	5	147.2
5	0.01	6	85.91	5	141.4
15	0.01	9	124.2	6	166.2
30	0.01	16	225.2	7	192.5
2	0.005	22	301.8	6	171
2	0.02	24	318.8	6	170.1
2	0.1	21	291.3	45	1227
0.2	0.001	22	295.7	∞	∞
1	0.005	25	342	7	198.9
4	0.02	19	254.6	7	197.9
1	0.02	14	187.3	6	169.8
20	0.001	25	336.4	9	245.5

with 100-by-100 points. The multiple shooting NLP then has of the order 30 million variables.

It is interesting to note that due to the high level formulation of the problem, using only matrix operations, the most expensive function corresponding to the evaluation of the matrices in the condensed Hessian, can be represented as a C-function with only 2369 lines of code. The compilation time using Clang and `-O3` code optimization, is only 7 seconds. This is small compared to the around 40 seconds for a single iteration of the NLP.

Table 8.3: Parameter estimation for the 2D shallow water equations with Gauss-Newton, 100-by-100 grid points

drag	depth	iter ss	time ss	iter ms	time ms
0.5	0.01	5	313.7	4	415.8
5	0.01	5	316	5	502.1
15	0.01	7	428.6	6	589.1
30	0.01	8	471	7	675.4
2	0.005	14	829.8	8	761.8
2	0.02	13	747.2	7	674.4
2	0.1	30	1545	32	2861
0.2	0.001	19	1049	9	852.6
1	0.005	16	879.5	8	760.9
4	0.02	13	718.3	7	672.7
1	0.02	16	854.9	8	759.2
20	0.001	14	763.5	7	674.6

Table 8.4: Parameter estimation for the 2D shallow water equations with exact Hessian, 100-by-100 grid points

drag	depth	iter ss	time ss	iter ms	time ms
2	0.01	0	49.62	2	238.9
0.5	0.01	5	299.2	9	858.5
5	0.01	5	302.5	8	792.5
15	0.01	7	406.6	10	982
30	0.01	8	456.7	11	1089
2	0.005	14	744.7	11	1044
2	0.02	13	692.5	13	1215
2	0.1	30	1516	45	4200
0.2	0.001	19	1009	12	1132
1	0.005	16	842.5	10	943.5
4	0.02	13	696.7	15	1428
1	0.02	16	839	14	1306
20	0.001	14	755.6	11	1054

8.4 Conclusion and outlook

In this chapter, we have presented a novel implementation of the lifted Newton method for SQP presented in Section 4.5. The implementation, called SCPgen, uses AD by source code transformation in CasADi and supports generating optimized C-code for the condensed QP.

As mentioned in the introduction of the chapter, SCPgen is still under development and a substantial programming effort is still required before it can be considered competitive with existing state-of-the-art NLP solvers. In the following, we list some of the extensions planned for SCPgen.

Globalization

SCPgen currently implements a very basic line-search globalization using an ℓ_1 merit function. While this does indeed work satisfactory for many problems, it often requires manual tuning of the line-search parameters. Globalization techniques that do not suffer from this include filter line-search and trust-region methods as mentioned in Section 4.4.

Regularization

Another related issue is regularization, also known as convexification, of the QP. This is required since most QP solvers cannot handle indefinite QPs, and even if a solution is found, it can not be guaranteed to give a descent direction for the line-search. SCPgen currently uses a rudimentary regularization based on shifting all the eigenvalues (by adding a multiple of the identity) until the Hessian is guaranteed to be positive semi-definite according to Greshgorin's circle theorem. This leads to an overly conservative approach. As an

alternative, the indefinite QP can be solved locally using e.g. IPOPT [179] (configured to work as a QP solver). Clearly, this is not a satisfactory approach.

Scaling

Scaling of the full-space NLP is important, in particular to avoid that the reduced space QP becomes too ill-conditioned. In SCPgen, the scaling can be done as part of the symbolic preprocessing.

Matrix-free QP solution

As mentioned in Section 4.5, the lifted Newton method for SQP can also be used together with a matrix-free QP solution. This represents an attractive alternative when the reduced space QP is neither small nor very sparse.

Sequential convex programming

The lifted Newton method for SQP readily extends to SCP. This is an attractive extension, especially since the symbolic framework in CasADi can be made to automatically detect convex subproblems present in NLPs. The detection and reformulation can rely on the rules for *disciplined convex programming* used to solve convex optimization problems in nonstandard form in e.g. CVX [99, 100].

Code-generation of the main optimization loop

Currently, the main optimization loop including the line-search and regularization is not code-generated. By adding code-generation also for this part as well as for the interface to the QP solver, it is possible to generate an SCP method in completely self-contained C-code to be used e.g. on an embedded system.

Real-time iterations

The code generation currently generates separate functions for evaluating the matrices and the vectors in the reduced space QP. In nonlinear model predictive control (NMPC) implementing *real-time iterations* [70], the calculation of the matrices is moved to a so-called *preparation phase*. This allows the NMPC controller to provide faster feedback.

Chapter 9

Applications

The methods presented in previous chapters and their implementation in CasADi has been used successfully both for method development and in applications at multiple research groups.

Notable applications include trajectory optimization for tethered airfoils [105, 192], parameter estimation in systems biology [114, 168], quadcopter flight control [95,96], remote sensing for unmanned aerial vehicles [112], time-optimal path-following flight control [170, 171, 172], geothermal electricity production [182], control of chemical reactors [137], control of combined-cycle power plants [128], optimization of production from oil wells [149] and spacecraft trajectory optimization for gravity assisted transfers to lunar orbit [106].

Notable non-standard optimal control methods whose implementation was made possible using CasADi include adjoint-based distributed multiple-shooting [125, 162], robust periodic optimal control [94] and non-conservative robust NMPC [137].

Finally, CasADi serves as the basis of other software codes, including several optimal control solvers inside JModelica.org [20, 133, 144]. Applications spurred indirectly via the CasADi-based solvers in JModelica.org include multi-objective MPC [146] and automotive control [141]. CasADi is also used in the RAWESOME code for airborne wind energy optimization and control [14] and gILC [176] for iterative learning control. Finally, CasADi has been used to implement direct collocation methods for OpenModelica [166].

In this last chapter, we provide some detail on two of these applications. In Section 9.1, we present a robustified state and control parameterization and

show how it can be used to control an industrial batch polymerization process. The second highlighted application is the startup optimization of a combined-cycle power plant, using a model written in the Modelica modeling language in Section 9.2.

9.1 Robust control of an industrial batch reactor

In the following we present one way of implementing a robust controller for an industrial reactor. We start by describing the dynamic model for the reactor and then describe the optimal control problem, which amounts to a *nonlinear model predictive control* (NMPC) formulation with economic cost function. To get a controller that is robust against uncertainties in the model parameters, we use a robust NMPC formulation as suggested by Lucia et al. [139,140]. This allows us to define and efficiently implement robustified versions of the three direct methods presented in Chapter 2. Finally, we discuss numerical results in the open-loop case. This application is the result of joint work with BASF SE and Sergio Lucia at the Technische Universität Dortmund. It was was carried out as part of the EMBOCON European Project for embedded optimization and control.

The focus of the following discussion is the formulation of the OCP and transcription into an NLP. For more discussion on the numerical results, we refer to [137].

A batch polymerization process model

The process under consideration is a batch polymerization reactor for production of complex chemical molecules (polymers) out of simpler molecules (monomers). The reactor consists of a jacketed vessel which is cooled by a constant flow of cooling water through the jacket. In addition to the cooling water, an external heat exchanger can be used to heat or cool the reactor. The process begins when a mixture of water and monomer is injected into the reactor and ends when a sufficient fraction of the monomer has been polymerized and extracted from the reactor. This is an exothermic process with a reaction rate that has a highly nonlinear temperature dependence. The process is also subject to considerable uncertainties.

The dynamic behavior of the reactor is modeled using the following system of ordinary differential equations, provided by BASF SE. It corresponds to an

industrial reactor for production of polypropylene from propene:

$$\dot{m}_{\mathrm{W}} = \mu_{\mathrm{F}}\,(1 - w_{\mathrm{M}}) \tag{9.1a}$$

$$\dot{m}_{\mathrm{M}} = \mu_{\mathrm{F}}\,w_{\mathrm{M}} - k_{\mathrm{R}}\,\tilde{m}_{\mathrm{M}} - k_{\mathrm{P}}\,m_{\mathrm{X}}\,\frac{m_{\mathrm{M}}}{m} \tag{9.1b}$$

$$\dot{m}_{\mathrm{P}} = k_{\mathrm{R}}\,\tilde{m}_{\mathrm{M}} + k_{\mathrm{P}}\,m_{\mathrm{X}}\,\frac{m_{\mathrm{M}}}{m} \tag{9.1c}$$

$$\begin{aligned}\dot{T}_{\mathrm{R}} = \frac{1}{c_{\mathrm{R}}\,m}\,&\left[\mu_{\mathrm{F}}\,c_{\mathrm{F}}\,\left(T_{\mathrm{F}} - T_{\mathrm{R}}\right) - h\,A\,\left(T_{\mathrm{R}} - T_{\mathrm{V}}\right)\right.\\ &\left.-\mu_{\mathrm{X}}\,c_{\mathrm{R}}\,\left(T_{\mathrm{R}} - T_{\mathrm{P}}\right) + h_{\mathrm{R}}\,k_{\mathrm{R}}\,\tilde{m}_{\mathrm{M}}\right]\end{aligned} \tag{9.1d}$$

$$\dot{T}_{\mathrm{V}} = \frac{1}{c_{\mathrm{V}}\,m_{\mathrm{V}}}\,\left[h\,A\,\left(T_{\mathrm{R}} - T_{\mathrm{V}}\right) - h\,A\,\left(T_{\mathrm{V}} - T_{\mathrm{J}}\right)\right] \tag{9.1e}$$

$$\dot{T}_{\mathrm{J}} = \frac{1}{c_{\mathrm{W}}\,m_{\mathrm{MC}}}\,\left[\mu_{\mathrm{MC}}\,c_{\mathrm{W}}\,\left(T_{\mathrm{J}}^{\mathrm{IN}} - T_{\mathrm{J}}\right) + h\,A\,\left(T_{\mathrm{V}} - T_{\mathrm{J}}\right)\right] \tag{9.1f}$$

$$\dot{T}_{\mathrm{P}} = \frac{1}{c_{\mathrm{R}}\,m_{\mathrm{X}}}\,\left[\mu_{\mathrm{X}}\,c_{\mathrm{W}}\,\left(T_{\mathrm{R}} - T_{\mathrm{P}}\right)\right] - \alpha\,(T_{\mathrm{P}} - T_{\mathrm{X}}) + k_{\mathrm{P}}\,\frac{m_{\mathrm{M}}}{m}\,m_{\mathrm{X}}\,h_{\mathrm{R}}\right] \tag{9.1g}$$

$$\dot{T}_{\mathrm{X}} = \frac{1}{c_{\mathrm{W}}\,m_{\mathrm{XC}}}\,\left[\phi_{\mathrm{XC}}\,c_{\mathrm{W}}\,(T_{\mathrm{X}}^{\mathrm{IN}} - T_{\mathrm{X}}) - \alpha\,(T_{\mathrm{X}} - T_{\mathrm{P}})\right] \tag{9.1h}$$

Here, the first three equations describe mass balances for the water in the reactor (9.1a), the monomer (9.1b) and the polymer (9.1c). The next four equations describe the energy balances for the reactor (9.1d), the vessel (9.1e), the jacket (9.1f), the polymer exiting the external heat exchanger (9.1g) and the coolant leaving the external heat exchanger (9.1h).

In Table 9.1, we summarize the variables and constants entering in (9.1). The actual numerical values have been left out as requested by BASF SE. We have also defined the following auxiliary variables:

$$\begin{aligned} h &= \frac{m_{\mathrm{W}}}{m}\,h_{\mathrm{WS}} + \frac{m_{\mathrm{M}}}{m}\,h_{\mathrm{MS}} + \frac{m_{\mathrm{P}}}{m}\,h_{\mathrm{PS}}, & \tilde{m}_{\mathrm{M}} &= m_{\mathrm{M}}\,\left(1 - \frac{m_{\mathrm{X}}}{m}\right),\\ k &= k_1\,\frac{m_{\mathrm{M}}}{m_{\mathrm{M}} + m_{\mathrm{P}}} + k_2\,\frac{m_{\mathrm{P}}}{m_{\mathrm{M}} + m_{\mathrm{P}}}, & m &= m_{\mathrm{W}} + m_{\mathrm{M}} + m_{\mathrm{P}},\\ k_{\mathrm{R}} &= k_0\,k\,\exp\left(-\frac{E_{\mathrm{a}}}{R\,T_{\mathrm{R}}}\right), & k_{\mathrm{P}} &= k_0\,k\,\exp\left(-\frac{E_{\mathrm{a}}}{R\,T_{\mathrm{P}}}\right). \end{aligned}$$

Table 9.1: Variables and constants in the polymerization model

States x	Description
m_{W}	Water mass
m_{M}	Monomer mass
m_{P}	Polymer mass
T_{R}	Reactor temperature
T_{V}	Vessel temperature
T_{J}	Jacket temperature
T_{P}	External heat exchanger polymer temperature
T_{X}	External heat exchanger coolant temperature

Controls u	Description
μ_{F}	Feed flow of the water/monomer mixture
$T_{\mathrm{J}}^{\mathrm{IN}}$	Jacket inlet temperature
$T_{\mathrm{X}}^{\mathrm{IN}}$	External heat exchanger inlet temperature

Constants	Description
Uncertain:	
h_{R}	Specific reaction enthalpy
k_0	Specific reaction rate
Certain:	
A	Jacket surface
c_{W}	Specific heat capacity of water (at constant pressure)
c_{R}	Specific heat capacity of the reactor (at constant pressure)
c_{F}	Specific heat capacity of the feed (at constant pressure)
c_{V}	Specific heat capacity of the vessel (at constant pressure)
E_{a}	Activation energy
h_{WS}	Heat transfer coefficient, water to steel
h_{MS}	Heat transfer coefficient, monomer to steel
h_{PS}	Heat transfer coefficient, polymer to steel
k_1, k_2	Reaction parameters
m_{X}	External heat exchanger polymer mass
m_{XC}	External heat exchanger coolant mass
m_{MC}	Jacket coolant mass
m_{V}	Vessel steel mass
R	Gas constant
T_{F}	Feed temperature
w_{M}	Monomer mass fraction in feed
α	Product of surface area and heat transfer coefficient for external heat exchanger
μ_{XC}	External heat exchanger coolant flow
μ_{MC}	Coolant flow through the jacket
μ_{X}	Polymer flow through the jacket

Optimal control problem formulation

The problem under consideration is to control the plant to minimize the total batch time while respecting hard constraints on the temperatures. We shall use an NMPC scheme, trying to find the optimal controls for a time horizon of t_f. Since the total batch time is defined as the point in time when the polymer mass m_P reaches a certain value, we choose as the objective function to maximize m_P at the end of the time horizon. We get the following problem formulation:

$$
\begin{aligned}
\underset{x,u}{\text{minimize}} \quad & -m_\mathrm{P}(t_\mathrm{f}) \\
\text{subject to} \quad & \text{Equation (9.1)} \\
& x(0) = x_0 \\
& \underline{u} \le u(t) \le \overline{u}, \quad \underline{x} \le x(t) \le \overline{x}, \qquad t \in [0, t_\mathrm{f}] \\
& \int_0^{t_\mathrm{f}} \mu_\mathrm{F}(t)\, dt \le m_\mathrm{F}^{\max} \\
& \int_0^{t_\mathrm{f}} \left(\frac{h_\mathrm{R}\, \dot{m}_\mathrm{M}}{m\, c_\mathrm{R}} - \frac{m_\mathrm{M}\, h_\mathrm{R}\, (\dot{m}_\mathrm{M} + \dot{m}_\mathrm{W} + \dot{m}_\mathrm{P})}{m^2\, c_\mathrm{R}} + \dot{T}_\mathrm{R} \right) dt \le T_\mathrm{ad}^{\max}
\end{aligned}
\tag{9.2}
$$

In addition to the differential equation (9.1) with given initial conditions, there are bounds on the state and control trajectories. There are also two constraints formulated as quadratures. The first one represents an upper bound on the total amount of monomer remaining to be injected into the reactor and the second one is a safety constraint that ensures that the exothermic reaction will not cause the plant to "blow up", should the cooling system break down.

Solving (9.2) gives a solution trajectory for the controls u. Using the model predictive control paradigm, the first part of this optimal control trajectory is then fed into the system. The time horizon is then shifted and the problem resolved for new initial conditions.

A non-conservative robust NLP transcription

In the OCP formulation (9.2), we assumed all the model parameters (which are constants in the OCP) to be known with certainty. In reality, there are considerable uncertainties which can have large implications on the calculated

optimal control trajectories. When it is not possible to estimate such parameters, a popular way to take this into account is to use a min-max formulation as proposed by e.g. Scokaert and Mayne [165]. This means searching for the best solution (in some sense) that satisfies the constraints for any realization of the uncertain model parameters. In practice, however, a min-max formulation can be overly conservative and may fail to find any feasible solution. Therefore, Lucia et al. [139,140] proposed a less conservative approach to robust NMPC, which we shall employ here. Rather than to search for a solution for any realization of the uncertain parameters, we shall look for a solution that satisfies the constrains for a finite set of parameter combinations.

For the polymerization process, there are two uncertain model parameters, namely (h_{R}, k_0) as indicated in Table 9.1. These are given with nominal values $(h_{\mathrm{R}}^{\mathrm{nom}}, k_0^{\mathrm{nom}})$ as well as lower $(h_{\mathrm{R}}^{\mathrm{min}}, k_0^{\mathrm{min}})$ and upper $(h_{\mathrm{R}}^{\mathrm{max}}, k_0^{\mathrm{max}})$ bounds. This allows us to define the following set of realizations of the uncertain model parameters:

$$\begin{aligned}\mathcal{R} = \{ \quad & (h_{\mathrm{R}}^{\mathrm{nom}}, k_0^{\mathrm{nom}}), \quad (h_{\mathrm{R}}^{\mathrm{nom}}, k_0^{\mathrm{min}}), \quad (h_{\mathrm{R}}^{\mathrm{nom}}, k_0^{\mathrm{max}}), \\ & (h_{\mathrm{R}}^{\mathrm{min}}, k_0^{\mathrm{nom}}), \quad (h_{\mathrm{R}}^{\mathrm{min}}, k_0^{\mathrm{min}}), \quad (h_{\mathrm{R}}^{\mathrm{min}}, k_0^{\mathrm{max}}), \\ & (h_{\mathrm{R}}^{\mathrm{max}}, k_0^{\mathrm{nom}}), \quad (h_{\mathrm{R}}^{\mathrm{max}}, k_0^{\mathrm{min}}), \quad (h_{\mathrm{R}}^{\mathrm{max}}, k_0^{\mathrm{max}}) \quad \}\end{aligned} \tag{9.3}$$

Robust direct control parameterization

Our task is thus to find a control trajectory that gives a feasible solution for all of these nine combinations. For this we modify the traditional direct control discretization described in Section 2.3 taking into account the possible parameter realization. This gives rise to a *scenario tree* with different scenarios for the state trajectory as illustrated for the first two control intervals in Figure 9.1. For comparison, we also show standard (non-robust) NMPC in Figure 9.2. For clarity of the presentation, we show only three out of the nine parameter realizations.

In Figure 9.1, we get different possible values for the state at $t = t_1$ corresponding to the same value for the control but with different parameter realizations. It also, perhaps counter-intuitively, splits up further for $t = t_2$. While this is mainly important to be able to handle time-varying disturbances, it is also useful to be able to prove *recursive feasibility* and *stability* for the NMPC scheme. For details on this, we refer to [138].

To avoid an exponential increase in the number of states, branching of the state trajectory is only done during the robust horizon, i.e. the first L control intervals. For the remaining control interval, the uncertainty realization is kept constant. Using $x_k^{(i_1,\ldots,i_k)}$ and $u_k^{(i_1,\ldots,i_k)}$ to denote the state and control

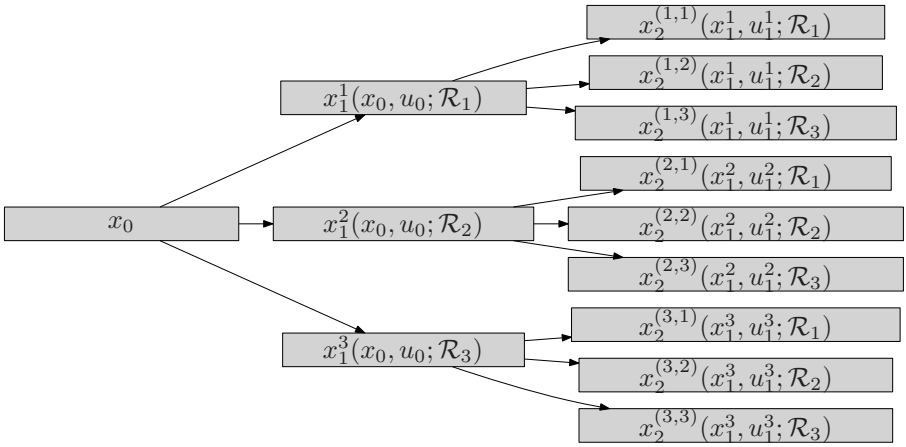

Figure 9.1: Robust NMPC: Scenario tree for the first two control intervals

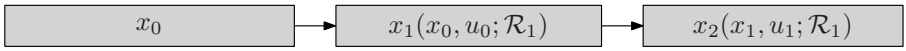

Figure 9.2: Standard NMPC: Two first control intervals

values corresponding to the uncertainty realization $(\mathcal{R}_{i_1}, \ldots, \mathcal{R}_{i_k})$ for the first k control intervals, this gives discrete time dynamics of the form:

$$\begin{cases} x_{k+1}^{(i_1,\ldots,i_k,j)} = F_k(x_k^{(i_1,\ldots,i_k)}, u_k^{(i_1,\ldots,i_k)}; \mathcal{R}_j), \\ \forall (i_1, \ldots, i_k, j) \in [1,9]^{k+1}, \end{cases} \qquad k = 0, \ldots, L-1$$

$$\begin{cases} x_{k+1}^{(i_1,\ldots,i_L)} = F_k(x_k^{(i_1,\ldots,i_L)}, u_k^{(i_1,\ldots,i_L)}; \mathcal{R}_{i_L}), \\ \forall (i_1, \ldots, i_L) \in [1,9]^{L}, \end{cases} \qquad k = L, \ldots, K-1 \tag{9.4}$$

We also impose the path constraints accordingly, cf. (2.16) from Chapter 2:

$$\begin{cases} \underline{u} \le u_k^{(i_1,\ldots,i_k)} \le \overline{u}, & \forall (i_1, \ldots, i_k) \in [1,9]^k, & k = 0, \ldots, L-1 \\ \underline{u} \le u_k^{(i_1,\ldots,i_L)} \le \overline{u}, & \forall (i_1, \ldots, i_L) \in [1,9]^L, & k = L, \ldots, K-1 \\ \underline{x} \le x_k^{(i_1,\ldots,i_k)} \le \overline{x}, & \forall (i_1, \ldots, i_k) \in [1,9]^k, & k = 1, \ldots, L \\ \underline{x} \le x_k^{(i_1,\ldots,i_L)} \le \overline{x}, & \forall (i_1, \ldots, i_L) \in [1,9]^L, & k = L+1, \ldots, K \end{cases} \tag{9.5}$$

Furthermore augmenting the state x with two new entries for the two quadratures in (9.2) gives a discrete time OCP of the form:

$$\begin{array}{ll} \underset{x,\,u}{\text{minimize}} & -\sum_{s\in[1,9]^L} \omega^s \left(m_{\mathrm{P}}\right)_K^s \\ \text{subject to} & x_0 = \overline{x}_0 \\ & \text{Equation (9.4)} \\ & \text{Equation (9.5)} \\ & h(x_K^s) \leq \overline{h}, \end{array} \tag{9.6}$$

where we have used a weighted combination of all the scenarios as the cost function. The weighting factors ω^s are chosen to give all scenarios equal weight.

This allows us to modify the direct optimal control methods from Chapter 2 giving three new methods, namely

1. a *robustified direct single shooting* method by completely eliminating the state trajectory from the NLP,
2. a *robustified direct multiple shooting* method by keeping the state trajectory in the NLP as well as
3. a *robustified direct collocation* method by also including the state at the integrator steps in the NLP.

For more detail on the here presented robust NMPC formulation, we refer to Lucia et al. [140].

Implementation

Both a robustified direct multiple shooting and a robustified direct collocation were implemented and used to solve the problem. In the direct multiple shooting case, CasADi's interface to CVODES [113]. In the direct collocation case, Radau collocation points were used with interpolating polynomial of second order and two finite elements per collocation interval. In both cases a prediction horizon of $K = 30$ steps was chosen. The resulting NLP was solved with IPOPT [179]. For more detail on the implementation, we refer [137].

Conclusions and outlook

In Figure 9.3, we show numerical results for the proposed controller. The topmost plot shows the evolution of the reactor temperature T_{R}, which is

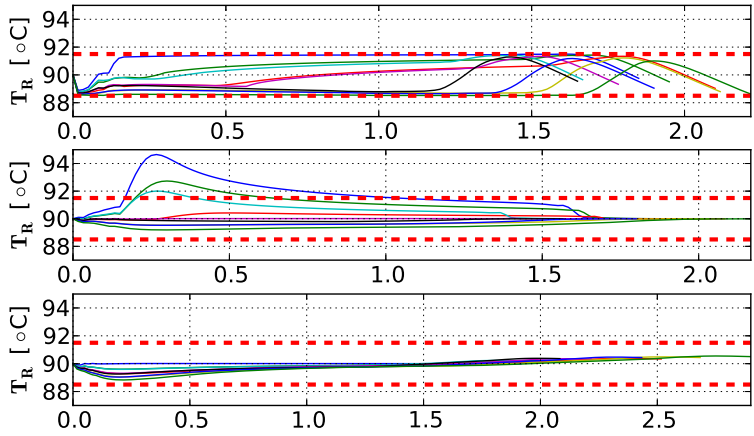

Figure 9.3: Evolution of the reactor temperature for the proposed non-conservative robust NMPC (top), standard NMPC (middle) and worst-case NMPC (bottom). Different trajectories correspond to different realizations of the model uncertainty.

subject to safety critical bounds depicted with dashed lines. The nine curves in the figure correspond to different realizations of the uncertain parameters. We see that all realizations remain within the safety-critical bounds. This result is contrasted with the same curves for standard NMPC in the middle figure and worst-case (conservative) NMPC in the bottom figure. We see that while standard NMPC fails to satisfy the temperature bounds for some uncertainty realizations, worst-case NMPC behaves overly conservatively, hence sacrificing optimality.

For more detailed results, we refer [137].

At the time of writing, experimental validation of the proposed controller was underway at the BASF SE facility in Ludwigshafen, Germany.

9.2 Startup optimization of a combined-cycle power plant

A relevant and challenging dynamic optimization problem is the minimal time start-up of combined cycle power plants (CCPP). These plants, which are often used for peak-load electricity production, are frequently started up

and shut down causing large energy losses as well as strains in components, shortening their lifetimes and potentially leading to catastrophic failure. By using optimization-based control techniques such as nonlinear model predictive control (NMPC) [77], it is possible to minimize fuel use to (grid) electricity production, while at the same time respecting these hard constraints on strains and pressures.

In the following, we make a brief overview of the optimization problem. It highlights the possibility to import complex dynamic models from physical modeling languages as described in Section 6.2. The results about the tool coupling below were first presented in Andersson et al. [26]. These results were then extended with more focus on the actual optimization problem and presented by Larsson et al. [128]. The condensed presentation in the following follows closely that of Andersson et al. [27].

Dynamic model

To model the plant, we use a first-principle model of the plant written in the physical modeling language Modelica [17] as well as Optimica [20], an extension of Modelica allowing the formulation of dynamic optimization constructs such as cost functions and constraints.

A schematic diagram of the model, which was developed by Casella et al. [58], is shown in Figure 9.4. It consists of a gas turbine and a steam turbine made up of an economizer, an evaporator and a superheater. For details of the model and the optimal control problem formulation, the minimum time start-up under a turbine rotor stress constraint, we refer to [58].

Model import into CasADi and NLP transcription

We use JModelica.org [20] to translate Modelica/Optimica code into an OCP in DAE with 10 differential states and 127 algebraic variables. The objective is an Lagrange-type cost function. This OCP is then exported in a symbolic XML-based format and imported into CasADi as reported in [26]. With the model now completely formulated as symbolic expressions in CasADi, the OCP can be used and manipulated just like the OCP formulated natively in CasADi.

To reformulate the OCP to an NLP, we choose direct collocation approach on 40 finite elements using fourth order Radau points as described in Section 2.6. This implementation is freely available as part of the JModelica.org package. A first prototype of the CasADi-based collocation implementation was presented

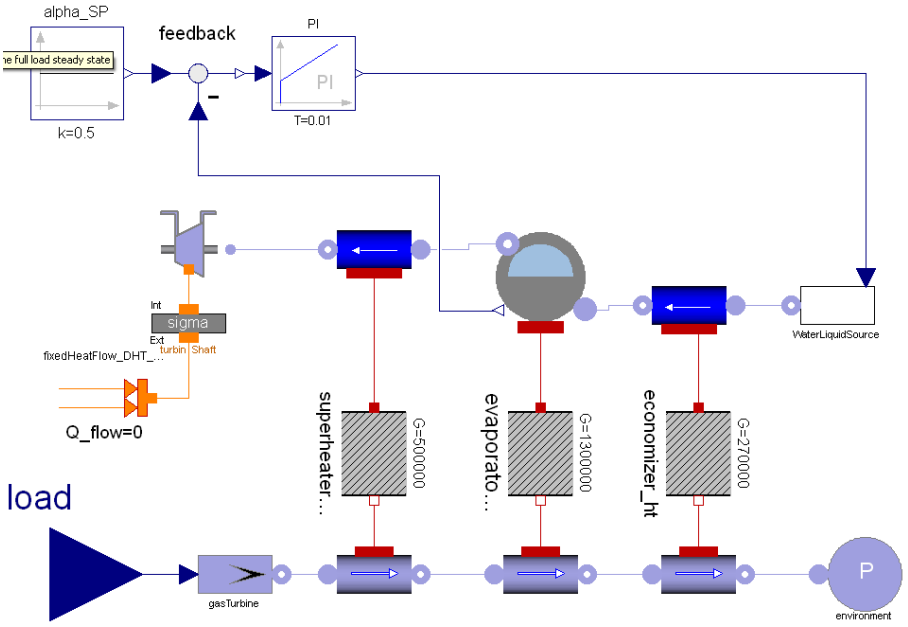

Figure 9.4: A combined cycle power plant

in [26] and was a translation of an earlier C-based implementation [20]. It was later completed and extended in [144].

Solution using CasADi and IPOPT

The NLP was solved in [144] using CasADi's interface to IPOPT (version 3.10) using MA27 as the linear solver. With default options and exact Hessian of the Lagrangian calculation using CasADi's AD framework, IPOPT's primal-dual interior point method needed 79 iterations to converge to the optimal solution. A break-up of the timings into NLP solver initialization as well as the IPOPT internal and CasADi internal part of the NLP solution time, is shown in Table 9.2.

As seen in the table, over 80 % of the NLP solution time was spent in IPOPT internally (mainly in the linear solver). The 0.8 s solution time should compared with 46 s needed by a previous implementation of the same method, based on the general purpose AD tool CppAD [5] applied to generated C code. This in addition to being implemented in a high level language (Python) rather than

Table 9.2: Timings for the CCPP example [144]

Solution step	**Time**
OCP-to-NLP transcription and solver initialization	4.2 s
IPOPT internally and linear solver (MA27)	4.0 s
NLP function and derivative evaluation in CasADi	0.8 s

C. As shown in Chapter 6, the time spent in CasADi can be cut further by up to a factor five by generating C-code for the NLP functions.

For more detail on this optimization problem, we refer to Larsson et al. [128].

Chapter 10

Conclusion

In this thesis, we have presented methods and software aimed at making derivative-based numerical optimization in general and simulation-based optimization in particular more accessible to users in academia and industry. By drastically lowering the effort needed to implement the methods, it becomes easier for researchers to prototype new optimization codes and for advanced users to tailor an optimization algorithm to a particular application.

10.1 Summary with highlights

Chapters 2, 3 and 4 are background chapters aimed at giving readers a brief but relatively self-contained introduction into dynamic optimization, algorithmic differentiation and nonlinear programming. It corresponds essentially to the kind of background needed to be able to use a software package such as CasADi efficiently. The only real contribution in these chapters is the presentation of the lifted Newton method for SQP in Section 4.5, we believe is more accessible than previous presentations of the method.

In Chapter 5, we presented a novel algorithm for QP condensing. This algorithm was shown to be equivalent to the condensing algorithm, used for nearly 30 years in numerous applications of model-based control and estimation, but with quadratic instead of cubic complexity in the prediction horizon. We also pointed out variants of the approach that we believe are useful in the context of first order methods for optimal control.

In Chapter 6, we presented the freely available software framework CasADi,

which is the main contribution of this thesis. We primarily discussed the implementation of the tool, referring to CasADi's user guide [28] as well as a brief tutorial in Appendix A, for a more comprehensive introduction into the syntax and usage. Although the main focus of CasADi is ease-of-use, the chapter also includes a benchmarking section which suggests that CasADi competes favorably against the state-of-the-art algebraic modeling language AMPL in terms of raw numerical efficiency.

In Chapter 7, we proposed a way to efficiently embed solvers of initial-value problems in ordinary or differential-algebraic equations into symbolic expressions. This relieves users of the complex and often error-prone task of formulating and efficiently solving the forward and adjoint sensitivity equations. In particular, this drastically reduces the effort needed to implement shooting-based method for optimal control.

In Chapter 8, we presented work towards a general-purpose structure exploiting and code-generating NLP solver called SCPgen. The main purpose of this tool is to address the solution of the NLPs arising in multiple shooting type methods. These NLPs are often large, but not sparse enough to be handled efficiently by existing general-purpose NLP solvers. We presented preliminary numerical results for the solver that confirm that the solver can indeed be used to solver very large, but structured, NLPs efficiently.

Finally in Chapter 9, we gave an overview of results that have been generated using CasADi to date. This includes results from several different research groups, results that in many cases have been generated with very little interaction with the developers of CasADi. This suggests that CasADi has the potential of establishing itself as a standard tool for numerical optimization. Two applications were highlighted, the control of a chemical reactor using a novel optimization-based control scheme and the startup optimization of a combined cycle power plant using a model defined in the Modelica physical modeling language.

10.2 Future research directions

The algorithms and software presented have already proven useful for real-world applications. By incrementally improving CasADi, we intend to address ever larger and more challenging problems.

New problem classes for CasADi

It is possible to extend CasADi to handle new classes of problems, in particular including *mixed-integer NLPs* (MINLPs). This would in turn greatly facilitate the transcription and solution of *mixed-integer OCPs* (MIOCPs). This would require the addition of solvers or interfaces for MINLPs as well as an extension of the symbolic framework to efficiently handle flow control.

Another interesting extension is support for OCPs in partial-differential equations (PDEs). Such a support is possible either by extending the symbolic framework or by a tight coupling with existing tools for modeling and simulation of PDEs such as FEniCS/DOLFIN/dolfin-adjoint [74, 135, 136].

Structure-exploiting nonlinear programming

As was mentioned in Chapter 8, a possible extension of SCPgen is to implement the rules for *disciplined convex programming* [99]. Rather than to reformulate convex optimization problems in nonstandard form as is done by e.g. CVX [100], it is possible to use these rules to detect and exploit convex substructures in (nonconvex) NLPs. The resulting structured NLP can then be solved using methods for sequential convex programming as described e.g. in the dissertation by Tran-Dinh [169].

Appendix A

Tutorial to optimal control with CasADi

In the following, we provide a short tutorial of how to solve the minimal fuel orbit transfer problem presented in Section 2.1 using CasADi via its Python front-end. The discussion is intended to be self-contained but mainly intended to convey the general solution approach. It is our hope that also readers not proficient with the Python programming language will be able to grasp the main concepts. We also invite users to revisit Section 6.2, before reading the tutorial. For a more detailed discussion, we refer to CasADi's user guide [28]. After encoding the model in CasADi in Section A.1, we show how it can be solved using direct collocation in Section A.2 and direct single shooting in Section A.3. The syntax corresponds to CasADi v1.8.0beta.

A.1 Physical modeling

We start by encoding the model equations from Section 2.1:

```
from casadi import *
import numpy as NP

# Physical parameters
mu = 398600.4418e9         # Earth's gravitational parameter
g0 = 9.80665               # Standard gravity [m/s**2]
R = 6378.1e3               # Earth's radius [m]
def g(r): return mu/r**2   # Gravitational field as a function of r

# Spacecraft parameters
T_max = 180e3              # Maximum thrust [N]
```

```
Isp = 450.                  # Specific impulse [s]
beta = 1./(Isp*g0)          # Inverse of the thruster velocity

# State vector with scaling factors and bounds
nx = 5                      # Number of states
r_n = ssym("r");      r_s=1e7;  r=r_s*r_n;       r_lb=0.;      r_ub=inf
phi_n = ssym("phi"); phi_s=1;  phi=phi_s*phi_n; phi_lb=-inf;  phi_ub=inf
v_n = ssym("v");      v_s=1e3;  v=v_s*v_n;       v_lb=0;       v_ub=inf
gam_n = ssym("v_h"); gam_s=1;  gam=gam_s*gam_n; gam_lb=-inf;  gam_ub=inf
m_n = ssym("m");      m_s=1e4;  m=m_s*m_n;       m_lb=10.0;    m_ub=inf
x  = vertcat((   r_n,  phi_n,  v_n,  gam_n,  m_n ))       # State vector
x_s = NP.array( [r_s,  phi_s,  v_s,  gam_s,  m_s])        # Scaling factor
x_lb = NP.array([r_lb, phi_lb, v_lb, gam_lb, m_lb])/x_s   # Lower bound
x_ub = NP.array([r_ub, phi_ub, v_ub, gam_ub, m_ub])/x_s   # Upper bound

# Normalized thrust
nu = 2                      # Number of controls
u = ssym("u",2)             # Control vector

# Spacecraft dynamics
dr    = v*sin(gam)
dphi  = v/r*cos(gam)
dv    = -g(r) * sin(gam) + T_max/m * u[0]
dgam  = (v/r - g(r)/v) * cos(gam) + T_max/(m*v) * u[1]
```

As was mentioned in Section 6.2, an alternative to performing the physical modeling directly in CasADi like this is to import a description of the model formulated in the physical modeling language Modelica [17].

Next, we specify the boundary conditions as described in Equation (2.3):

```
dm    = -beta*T_max*sqrt(u[0]**2 + u[1]**2 + 1e-10)
xdot  = vertcat(( dr, dphi, dv, dgam, dm )) / x_s

# Initial conditions, corresponding to a sun synchronous orbit (SSO)
r_0 = 200e3 + R; phi_0 = 0.; v_0 = 5.5e3; gam_0 = 2*pi/180.; m_0 = 40000.
x_0 = NP.array([r_0,phi_0,v_0,gam_0,m_0]) / x_s

# Terminal conditions, corresponding to a circular orbit
r_f = 800e3 + R; v_f = 7.5e3; gam_f = 0.
x_lbf = NP.array([r_f,-inf,v_f,gam_f,10.]) / x_s
x_ubf = NP.array([r_f, inf,v_f,gam_f,inf]) / x_s
```

We constrain the thrust vector to the positive quadrant, i.e. $u_1(t) \geq 0$ and $u_2(t) \geq 0$). And finally, following the suggestion in Cerf et al. [60], we set the final time to 1483 seconds and use an initial guess for the solution trajectory consisting of two thrust arcs:

```
# Bounds on the controls
def u_lb(t): return NP.array([  -0.,-0.])
def u_ub(t): return NP.array([ inf, inf])

# Final time
tf = 1483. # s

# Optimal trajectory guess
def u0(t):
  if t<500: return NP.array([ .8, .6])
  else:     return NP.array([ .0, .0])
```

This completes the encoding of the model.

A.2 Direct collocation

We we now proceed to the solution using direct collocation. To simplify the presentation, we choose to use a uniform control discretization with $K = 500$ intervals, but with only one finite element per control interval. We use Legendre points of order $d = 1$, which means that Equation (2.22) becomes a mid-point rule, cf. [43]:

$$\tau_1 := 0.5. \tag{A.1}$$

The collocation equations will hence only be enforced once, in the middle of each control interval. From (2.24) and (2.25) we can precalculate $C_{r,1}$ and D_r:

```
K = 500
d = 1
C = NP.array([-2,2])
D = NP.array([-1,2])
```

Using the symbolic expressions for the ODE we now form a function object corresponding to the ODE right hand side:

```
dae_fcn  = SXFunction([x,u],[xdot]);  dae_fcn.init()
```

Next we declare the free variable in the NLP as a symbolic primitive of the matrix-valued expression graph and create references to the section of the variable corresponding to each control interval:

```
nv_k = (d+1)*nx+nu            # Number of NLP variables per control interval
nv = nv_k*K + nx              # Total number of NLP variables
v = msym("v",nv)              # NLP variable vector

# Split NLP variables up by control interval
vk = [v[k*nv_k : min(nv,(k+1)*nv_k)] for k in range(K+1)]
```

We are now ready to form symbolic expressions for the NLP objective functions and constraints:

```
# Construct NLP
g = []                        # NLP constraint vector
lbv = []; ubv = []; v0 = []   # Bounds and initial guess for NLP variables
lbg = []; ubg = []            # Nonlinear bounds
for k in range(K):

  # Time at the beginning of interval
  t = (k*tf)/K

  # State at the beginning of the interval
  xk_0 = vk[k][:nx]
```

```
    v0.append(x_0)
    lbv.append(x_0 if k==0 else x_lb)
    ubv.append(x_0 if k==0 else x_ub)

    # Collocation point
    xk_1 = vk[k][nx:2*nx]
    v0.append(x_0)
    lbv.append(x_lb)
    ubv.append(x_ub)

    # Parametrized control
    uk = vk[k][2*nx:]
    v0.append(u0(t));            lbv.append(u_lb(t));  ubv.append(u_ub(t))

    # Enforce the path constraint
    g.append(inner_prod(uk,uk)); lbg.append([-inf]);   ubg.append([1.0])

    # Expression for the state derivative at the collocation point
    xp = C[0]*xk_0 + C[1]*xk_1

    # Add collocation equations to the NLP
    [fk] = dae_fcn.call([xk_1, uk])
    g.append((tf/K)*fk - xp)
    lbg.append(NP.zeros(nx))
    ubg.append(NP.zeros(nx))

    # Expression for the state at the end of the finite element
    xf = D[0]*xk_0 + D[1]*xk_1

    # Add continuity equation to NLP
    g.append(vk[k+1][:nx] - xf)
    lbg.append(NP.zeros(nx))
    ubg.append(NP.zeros(nx))

# State constraint and initial guess at end time
v0.append(x_0);  lbv.append(x_lbf);  ubv.append(x_ubf)

# Objective function value
f = -vk[K][4]  # -m, i.e. maximize mass
```

Finally, the expressions for the NLP are used to define an NLP solver instance, here using CasADi's interface to IPOPT [179]:

```
# NLP instance
nlp = MXFunction(nlpIn(x=v),nlpOut(f=f,g=vertcat(g)))

# NLP solver instance
solver = IpoptSolver(nlp)
solver.init()

# Pass initial guess and bounds on variables and constraints
solver.setInput(NP.concatenate(v0),"x0")
solver.setInput(NP.concatenate(lbv),"lbx")
solver.setInput(NP.concatenate(ubv),"ubx")
solver.setInput(NP.concatenate(lbg),"lbg")
solver.setInput(NP.concatenate(ubg),"ubg")

# Solve the problem
solver.solve()
```

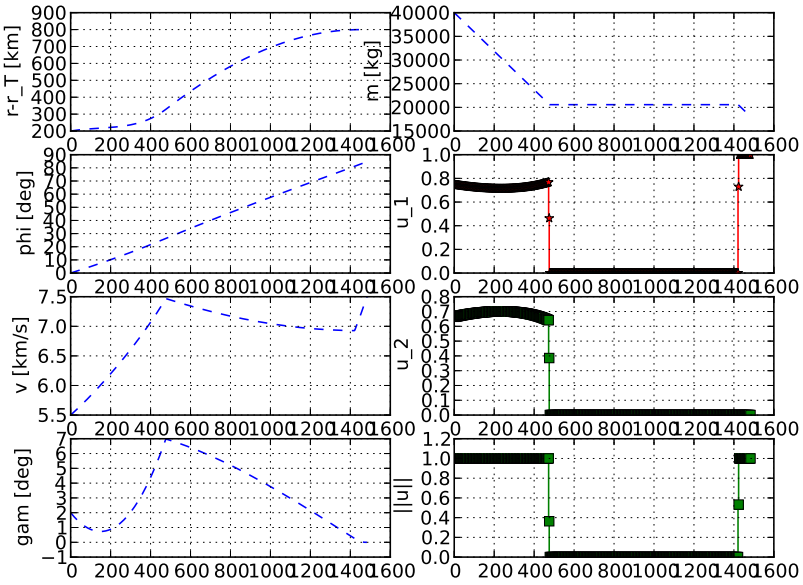

Figure A.1: Solution for the orbital transfer problem using direct collocation

This forms an NLP with 5997 nontrivial variables, 5000 equality constraints and 500 inequality constraints. The script presented in this section converges to the solution shown in Figure A.1 after 40 iterations of IPOPT's primal-dual interior point methods using exact first and second order derivative information provided by CasADi. The total solution time, including symbolic processing is less than 1.5 seconds on 2.5 GHz MacBook Pro, out of which 0.15 seconds are spent to evaluate derivatives and 0.35 seconds are spent internally in IPOPT and in its linear solver, which here was ma27.

A.3 Direct single shooting

Next, we solve the same problem using the direct single shooting method as presented in Section 2.4. It relies on the automatic sensitivity analysis presented in Chapter 7.

Using a control discretization of $K = 20$ and with the model equations already declared exactly as in Section A.2, we start by declaring the NLP variable, which is now only the sought controls:

```
K = 20                                  # Control discretization
v = msym("v",nu*K)                      # NLP variable vector
```

We also declare an instance of the CVODES integrator from the SUNDIALS suite. The integrator integrates over an interval of length `tf/K` rather than 1, assumed in the equations so far:

```
dae = SXFunction(daeIn(x=x,p=u),daeOut(ode=xdot))
F = CVodesIntegrator(dae)
F.setOption("tf",tf/K)                  # Interval length
F.init()
```

Using this function object, we can form a symbolic expression for the state at the end time,

```
xk  = x_0
for k in range(K):
  uk = v[k*nu:(k+1)*nu]                 # Local control
  F_out = F.call(integratorIn(x0=xk,p=uk)) # Call the integrator
  xk, = integratorOut(F_out,"xf")       # Retrieve the state
```

which in turn can be used to define the objective function and constraints of the NLP. The constraint consists of two parts, the terminal constraints on r, v and γ as well the the upper bound on the norm of u:

```
# Objective function value
f = -xk[4]

# NLP constraints
g   =         vertcat((   xk[[0,2,3]],      v[0::2]**2 + v[1::2]**2))
lbg = NP.concatenate((x_lbf[[0,2,3]], -inf*NP.ones(K)))
ubg = NP.concatenate((x_ubf[[0,2,3]],      NP.ones(K)))
```

Finally, we pass this on to the NLP solver, where we again use IPOPT with exact first and second order derivative information generated by CasADi:

```
# Allocate an NLP solver
nlp = MXFunction(nlpIn(x=v),nlpOut(f=f,g=g))
solver = IpoptSolver(nlp)
solver.init()

# Initial guess
v0 = NP.concatenate([u0(k*tf/K) for k in range(K)])

# Set bounds and initial guess
solver.setInput( 0,    "lbx")
solver.setInput( inf,  "ubx")
solver.setInput( v0,   "x0")
solver.setInput( lbg,  "lbg")
solver.setInput( ubg,  "ubg")

# Solve the problem
solver.solve()
```

This optimization converges to the same solution as presented in Figure A.1 after 27 iterations. The solution time is considerably longer, 15 seconds, as can be expected from using a variable-order, variable-step size integrator written to mainly handle very large ODEs.

Appendix B

Derivative propagation rules for embedded integrators

For completeness, and as a service to the reader, we state the derivative rules for the extended integrator formulations proposed in Chapter 7.

B.1 Discrete-time one-step integrators

The derivation follows equivalently with the simplified setting in Section 7.1, while handling implicit dynamics as in Section 3.10.

Forward mode

The forward derivative propagation rule for the function defined by Algorithm 7.5 is shown in Algorithm B.1.

Algorithm B.1 Definition of proposed $\hat{F}$:
$\mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_v} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_u \times m} \times \mathbb{R}^{n_r \times m} \times \mathbb{R}^{n_v \times m}$
$\to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_p} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_q \times m} \times \mathbb{R}^{n_r \times m} \times \mathbb{R}^{n_p \times m}$

input $(x_0, u, r_N, v, \hat{X}_0, \hat{U}, \hat{R}_N, \hat{V})$
$q_0 = 0$
$\hat{Q}_0 = 0$
for $k = 0, \ldots, N-1$ **do**
  $0 = \Theta(x_k, z_k, u)$ ▷ Solve for z_k
  $0 = \hat{\Theta}(x_k, z_k, u, \hat{X}_k, \hat{Z}_k, \hat{U})$ ▷ Solve for $\hat{Z}_k$
  $x_{k+1} = \Phi(x_k, z_k, u)$
  $\hat{X}_{k+1} = \hat{\Phi}(x_k, z_k, u, \hat{X}_k, \hat{Z}_k, \hat{U})$
  $q_{k+1} = \Psi(x_k, z_k, u) + q_k$
  $\hat{Q}_{k+1} = \hat{\Psi}(x_k, z_k, u, \hat{X}_k, \hat{Z}_k, \hat{Q}) + \hat{Q}_k$
end for
$p_N = 0$
$\hat{P}_N = 0$
for $k = N-1, \ldots, 0$ **do**
  $0 = \Theta^*(x_k, z_k, u, r_{k+1}, s_k, v)$ ▷ Solve for s_k
  $0 = \hat{\Theta}^*(x_k, z_k, u, r_{k+1}, s_k, v, \hat{X}_k, \hat{Z}_k, \hat{U}, \hat{R}_{k+1}, \hat{S}_k, \hat{V})$ ▷ Solve for $\hat{S}_k$
  $r_k = \Phi^*(x_k, z_k, u, r_{k+1}, s_k, v)$
  $\hat{R}_k = \hat{\Phi}^*(x_k, z_k, u, r_{k+1}, s_k, v, \hat{X}_k, \hat{Z}_k, \hat{U}, \hat{R}_{k+1}, \hat{S}_k, \hat{V})$
  $p_k = \Psi^*(x_k, z_k, u, r_{k+1}, s_k, v) + p_{k+1}$
  $\hat{P}_k = \hat{\Psi}^*(x_k, z_k, u, r_{k+1}, s_k, v, \hat{X}_k, \hat{Z}_k, \hat{U}, \hat{R}_{k+1}, \hat{S}_k, \hat{V}) + \hat{P}_{k+1}$
end for
return $(x_N, q_N, r_0, p_0, \hat{X}_N, \hat{Q}_N, \hat{R}_0, \hat{P}_0)$

To simplify the notation we have introduced:

$$\hat{\Phi}(x_k, z_k, u, \hat{X}_k, \hat{Z}_k, \hat{U}) := \frac{\partial \Phi}{\partial x}(x_k, z_k, u)\,\hat{X}_k + \frac{\partial \Phi}{\partial z}(x_k, z_k, u)\,\hat{Z}_k + \frac{\partial \Phi}{\partial u}(x_k, z_k, u)\,\hat{U}, \tag{B.1a}$$

$$\begin{aligned}\hat{\Phi}^*(x_k, z_k, u, r_{k+1}, s_k, v, \hat{X}_k, \hat{Z}_k, \hat{U}, \hat{R}_{k+1}, \hat{S}_k, \hat{V}) &:= \frac{\partial \Phi^*}{\partial x}(x_k, z_k, u, r_{k+1}, s_k, v)\,\hat{X}_k + \frac{\partial \Phi^*}{\partial z}(x_k, z_k, u, r_{k+1}, s_k, v)\,\hat{Z}_k \\ &+ \frac{\partial \Phi^*}{\partial u}(x_k, z_k, u, r_{k+1}, s_k, v)\,\hat{U} \\ &+ \frac{\partial \Phi^*}{\partial r}(x_k, z_k, u)\,\hat{R}_{k+1} + \frac{\partial \Phi^*}{\partial s}(x_k, z_k, u)\,\hat{S}_k + \frac{\partial \Phi^*}{\partial v}(x_k, z_k, u)\,\hat{V},\end{aligned} \tag{B.1b}$$

with $\hat{\Theta}$, $\hat{\Psi}$, $\hat{\Theta}^*$ and $\hat{\Psi}^*$ defined analogously.

Reverse mode

Similarly, the adjoint derivative propagation rule is shown in Algorithm B.2.

$$\bar{\Phi}^*(x_k, z_k, u, \bar{R}_k, \bar{S}_k, \bar{P}) := \left[\frac{\partial \Phi^*}{\partial r}(x_k, z_k, u)\right]^\intercal \bar{R}_k + \left[\frac{\partial \Theta^*}{\partial r}(x_k, z_k, u)\right]^\intercal \bar{S}_k + \left[\frac{\partial \Psi^*}{\partial r}(x_k, z_k, u)\right]^\intercal \bar{P}, \tag{B.2a}$$

$$\begin{aligned}\bar{\Phi}(x_k, z_k, u, r_{k+1}, s_k, v, \bar{X}_{k+1}, \bar{Z}_k, \bar{Q}, \bar{R}_k, \bar{S}_k, \bar{P}) &:= \left[\frac{\partial \Phi}{\partial x}(x_k, z_k, u)\right]^\intercal \bar{X}_{k+1} + \left[\frac{\partial \Theta}{\partial x}(x_k, z_k, u)\right]^\intercal \bar{R}_k + \left[\frac{\partial \Psi}{\partial x}(x_k, z_k, u)\right]^\intercal \bar{Q} \\ &+ \left[\frac{\partial \Phi^*}{\partial x}(x_k, z_k, u, r_{k+1}, s_k, v)\right]^\intercal \bar{R}_k + \left[\frac{\partial \Theta^*}{\partial x}(x_k, z_k, u, r_{k+1}, s_k, v)\right]^\intercal \bar{S}_k \\ &+ \left[\frac{\partial \Psi^*}{\partial x}(x_k, z_k, u, r_{k+1}, s_k, v)\right]^\intercal \bar{P},\end{aligned} \tag{B.2b}$$

with $\bar{\Theta}$, $\bar{\Psi}$, $\bar{\Theta}^*$ and $\bar{\Psi}^*$ defined analogously.

Algorithm B.2 Definition of proposed $\bar{F}$:
$\mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_v} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_q \times m} \times \mathbb{R}^{n_r \times m} \times \mathbb{R}^{n_p \times m}$
$\to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_p} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_u \times m} \times \mathbb{R}^{n_r \times m} \times \mathbb{R}^{n_v \times m}$

input $(x_0, u, r_N, v, \bar{X}_N, \bar{Q}, \bar{R}_0, \bar{P})$
$q_0 = 0$
$\bar{V}_0 = 0$
for $k = 0, \ldots, N-1$ **do**
  $0 = \Theta(x_k, z_k, u)$ ▷ Solve for z_k
  $0 = \bar{\Theta}^*(x_k, z_k, u, \bar{R}_k, \bar{S}_k, \bar{P})$ ▷ Solve for $\bar{S}_k$
  $x_{k+1} = \Phi(x_k, z_k, u)$
  $\bar{R}_{k+1} = \bar{\Phi}^*(x_k, z_k, u, \bar{R}_k, \bar{S}_k, \bar{P})$
  $q_{k+1} = \Psi(x_k, z_k, u) + q_k$
  $\bar{V}_{k+1} = \bar{\Psi}^*(x_k, z_k, u, \bar{R}_k, \bar{S}_k, \bar{P}) + \bar{V}_k$
end for
$p_N = 0$
$\bar{U}_N = 0$
for $k = N-1, \ldots, 0$ **do**
  $0 = \Theta^*(x_k, z_k, u, r_{k+1}, s_k, v)$ ▷ Solve for s_k
  $0 = \bar{\Theta}(x_k, z_k, u, r_{k+1}, s_k, v, \bar{X}_{k+1}, \bar{Z}_k, \bar{Q}, \bar{R}_k, \bar{S}_k, \bar{P})$ ▷ Solve for $\bar{Z}_k$
  $r_k = \Phi^*(x_k, z_k, u, r_{k+1}, s_k, v)$
  $\bar{X}_k = \bar{\Phi}(x_k, z_k, u, r_{k+1}, s_k, v, \bar{X}_{k+1}, \bar{Z}_k, \bar{Q}, \bar{R}_k, \bar{S}_k, \bar{P})$
  $p_k = \Psi^*(x_k, z_k, u, r_{k+1}, s_k, v) + p_{k+1}$
  $\bar{U}_k = \bar{\Psi}(x_k, z_k, u, r_{k+1}, s_k, v, \bar{X}_{k+1}, \bar{Z}_k, \bar{Q}, \bar{R}_k, \bar{S}_k, \bar{P}) + \bar{U}_{k+1}$
end for
return $(x_N, q_N, r_0, p_0, \bar{X}_0, \bar{U}_0, \bar{R}_N, \bar{V}_N)$

B.2 Continuous-time integrators

In continuous time, the rule follows as in Section 7.2, while handling implicit dynamics as in Section 3.11.

Forward mode

The forward derivative propagation rule for the extended continuous-time integrator defined in (7.7) becomes:

$$
\begin{aligned}
&\hat{f} : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_v} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_u \times m} \times \mathbb{R}^{n_r \times m} \times \mathbb{R}^{n_v \times m} \\
&\to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_p} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_q \times m} \times \mathbb{R}^{n_r \times m} \times \mathbb{R}^{n_p \times m}, \\
&\quad (x_0, u, r_1, v, \hat{X}_0, \hat{U}, \hat{R}_1, \hat{V}) \mapsto (x_1, q_1, r_0, p_0, \hat{X}_1, \hat{Q}_1, \hat{R}_0, \hat{P}_0) \\
&\left\{\begin{array}{rcl}
\dot{x}(t) &=& \phi(x(t), z(t), u), \\
\dot{\hat{X}}(t) &=& \hat{\phi}(x(t), z(t), u, \hat{X}(t), \hat{Z}(t), \hat{U}), \\
0 &=& \theta(x(t), z(t), u), \\
0 &=& \hat{\theta}(x(t), z(t), u, \hat{X}(t), \hat{Z}(t), \hat{U}), \\
\dot{q}(t) &=& \psi(x(t), z(t), u), \\
\dot{\hat{Q}}(t) &=& \hat{\psi}(x(t), z(t), u, \hat{X}(t), \hat{Z}(t), \hat{Q}), \\
-\dot{r}(t) &=& \phi^*(x(t), z(t), u, r(t), s(t), v), \\
-\dot{\hat{R}}(t) &=& \hat{\phi}^*(x(t), z(t), u, r(t), s(t), v, \hat{X}(t), \hat{Z}(t), \hat{U}, \hat{R}(t), \hat{S}(t), \hat{V}), \\
0 &=& \phi^*(x(t), z(t), u, r(t), s(t), v), \\
0 &=& \hat{\theta}^*(x(t), z(t), u, r(t), s(t), v, \hat{X}(t), \hat{Z}(t), \hat{U}, \hat{R}(t), \hat{S}(t), \hat{V}), \\
-\dot{p}(t) &=& \phi^*(x(t), z(t), u, r(t), s(t), v), \\
-\dot{\hat{P}}(t) &=& \hat{\psi}^*(x(t), z(t), u, r(t), s(t), v, \hat{X}(t), \hat{Z}(t), \hat{U}, \hat{R}(t), \hat{S}(t), \hat{V}),
\end{array}\right. \\
&\quad t \in [0, 1], \\
&x(0) = x_0, \quad r(1) = r_1, \quad q(0) = 0, \quad p(1) = 0, \\
&\hat{X}(0) = \hat{X}_0, \quad \hat{R}(1) = \hat{R}_1, \quad \hat{Q}(0) = 0, \quad \hat{P}(1) = 0, \\
&x_1 = x(1), \quad q_1 = q(1), \quad r_0 = r(0), \quad p_0 = p(0), \\
&\hat{X}_1 = \hat{X}(1), \quad \hat{Q}_1 = \hat{Q}(1), \quad \hat{R}_0 = \hat{R}(0), \quad \hat{P}_0 = \hat{P}(0),
\end{aligned}
\tag{B.3}
$$

with $\hat{\phi}$, $\hat{\theta}$ and $\hat{\psi}$ defined as in (B.1a) and $\hat{\phi}^*$, $\hat{\theta}^*$ and $\hat{\psi}^*$ defined as in (B.1b).

Reverse mode

Similarly, the adjoint derivative propagation rule for the extended continuous-time integrator defined in (7.7) becomes:

$$
\begin{aligned}
&\bar{f} : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_v} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_q \times m} \times \mathbb{R}^{n_r \times m} \times \mathbb{R}^{n_p \times m}, \\
&\to \mathbb{R}^{n_x} \times \mathbb{R}^{n_q} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_p} \times \mathbb{R}^{n_x \times m} \times \mathbb{R}^{n_u \times m} \times \mathbb{R}^{n_r \times m} \times \mathbb{R}^{n_v \times m} \\
&\quad (x_0, u, r_1, v, \bar{X}_1, \bar{Q}, \bar{R}_0, \bar{P}) \mapsto (x_1, q_1, r_0, p_0, \bar{X}_0, \bar{U}_0, \bar{R}_1, \bar{V}_1) \\
&\left\{ \begin{array}{rcl}
\dot{x}(t) & = & \phi(x(t), z(t), u), \\
\dot{\bar{R}}(t) & = & \bar{\phi}^*(x(t), z(t), u, \bar{R}(t), \bar{S}(t), \bar{P}), \\
0 & = & \theta(x(t), z(t), u), \\
0 & = & \bar{\theta}^*(x(t), z(t), u, \bar{R}(t), \bar{S}(t), \bar{P}), \\
\dot{q}(t) & = & \psi(x(t), z(t), u), \\
\dot{\bar{V}}(t) & = & \bar{\psi}^*(x(t), z(t), u, \bar{R}(t), \bar{S}(t), \bar{P}), \\
-\dot{r}(t) & = & \phi^*(x(t), z(t), u, r(t), s(t), v), \\
-\dot{\bar{X}}(t) & = & \bar{\phi}(x(t), z(t), u, r(t), s(t), v, \bar{X}(t), \bar{Z}(t), \bar{Q}, \bar{R}(t), \bar{S}(t), \bar{P}), \\
0 & = & \phi^*(x(t), z(t), u, r(t), s(t), v), \\
0 & = & \bar{\theta}(x(t), z(t), u, r(t), s(t), v, \bar{X}(t), \bar{Z}(t), \bar{Q}, \bar{R}(t), \bar{S}(t), \bar{P}), \\
-\dot{p}(t) & = & \phi^*(x(t), z(t), u, r(t), s(t), v), \\
-\dot{\bar{U}}(t) & = & \bar{\psi}(x(t), z(t), u, r(t), s(t), v, \bar{X}(t), \bar{Z}(t), \bar{Q}, \bar{R}(t), \bar{S}(t), \bar{P})
\end{array} \right. \\
&\quad t \in [0, 1], \\
&x(0) = x_0, \quad q(0) = 0, \quad r(1) = r_1, \quad p(1) = 0, \\
&\bar{X}(1) = \bar{X}_1, \quad \bar{V}(0) = 0, \quad \bar{R}(0) = \bar{R}_0, \quad \bar{U}(1) = 0, \\
&x_1 = x(1), \quad q_1 = q(1), \quad r_0 = r(0), \quad p_0 = p(0), \\
&\bar{X}_0 = \bar{X}(0), \quad \bar{U}_0 = \bar{U}(0), \quad \bar{R}_1 = \bar{R}(1), \quad \bar{V}_1 = \bar{V}(1),
\end{aligned}
\tag{B.4}
$$

with $\bar{\phi}^*$, $\bar{\theta}^*$ and $\bar{\psi}^*$ defined as in (B.2a) and $\bar{\phi}$, $\bar{\theta}$ and $\bar{\psi}$ defined as in (B.2b).

Bibliography

[1] ADIC. http://www.mcs.anl.gov/research/projects/adic/. [Online; accessed 4-June-2013]. pages 31

[2] ADiMat. http://www.sc.informatik.tu-darmstadt.de/res/adimat/. [Online; accessed 4-June-2013]. pages 31

[3] APMonitor. http://apmonitor.com/. [Online; accessed 10-October-2013]. pages 4

[4] ASCEND. http://ascend4.org. [Online; accessed 10-October-2013]. pages 4

[5] CppAD. http://www.coin-or.org/CppAD. [Online; accessed 11-May-2012]. pages 31, 76, 123

[6] Dymola. http://www.dynasim.se/. [Online; accessed 10-October-2013]. pages 4

[7] Eigen. http://eigen.tuxfamily.org/. [Online; accessed 4-June-2013]. pages 76

[8] FADBAD++. http://www.fadbad.com/fadbad.html. [Online; accessed 4-June-2013]. pages 31

[9] FMI. https://www.fmi-standard.org/. [Online; accessed 29-May-2013]. pages 75

[10] GAMS. http://www.gams.com. [Online; accessed 1-September-2011]. pages 35

[11] μBLAS. http://www.boost.org/. [Online; accessed 12-January-2010]. pages 76

[12] OpenAD. http://www.mcs.anl.gov/OpenAD/. [Online; accessed 4-June-2013]. pages 31

[13] PROPT. http://tomdyn.com. [Online; accessed 10-October-2013]. pages 3

[14] RAWESOME. https://github.com/ghorn/rawesome. [Online; accessed 10-October-2013]. pages 113

[15] SOCS:. http://www.boeing.com/boeing/phantom/socs/. [Online; accessed 10-October-2013]. pages 3

[16] TAPENADE. http://www-sop.inria.fr/tropics/. [Online; accessed 4-June-2013]. pages 31

[17] The Modelica Association. http://www.modelica.org. pages 75, 122, 130

[18] CasADi. http://casadi.org, 2013. [Online; accessed 10-October-2013]. pages 72

[19] Sacado. http://trilinos.sandia.gov/packages/, 2013. [Online; accessed 4-June-2013]. pages 31

[20] Åkesson, J., Årzén, K.-E., Gäfvert, M., Bergdahl, T., and Tummescheit, H. Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem. *Computers and Chemical Engineering 34* (2010), 1737–1749. pages 4, 75, 113, 122, 123

[21] Albersmeyer, J. *Adjoint-based algorithms and numerical methods for sensitivity generation and optimization of large scale dynamic systems*. PhD thesis, Ruprecht-Karls-Universitität Heidelberg, 2010. pages 44

[22] Albersmeyer, J., and Bock, H. Sensitivity Generation in an Adaptive BDF-Method. In *Modeling, Simulation and Optimization of Complex Processes: Proceedings of the International Conference on High Performance Scientific Computing, March 6-10, 2006, Hanoi, Vietnam* (2008), H. Bock, E. Kostina, X. Phu, and R. Rannacher, Eds., Springer, pp. 15–24. pages 44

[23] Albersmeyer, J., and Diehl, M. The Lifted Newton Method and its Application in Optimization. *SIAM Journal on Optimization 20*, 3 (2010), 1655–1684. pages 4, 11, 53, 54, 58, 103, 106, 107, 108, 109

[24] Alexandrescu, A. *Modern C++ Design*. Addison-Wesley, 2001. pages 74

[25] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., and Sorensen, D. *Lapack Users' Guide*, 2nd edition ed. SIAM, Philadelphia, 1995. pages 29, 73

[26] Andersson, J., Åkesson, J., Casella, F., and Diehl, M. Integration of CasADi and JModelica.org. In *8th International Modelica Conference* (2011). pages 75, 122, 123

[27] Andersson, J., Åkesson, J., and Diehl, M. CasADi – A symbolic package for automatic differentiation and optimal control. In *Recent Advances in Algorithmic Differentiation* (Berlin, 2012), S. Forth, P. Hovland, E. Phipps, J. Utke, and A. Walther, Eds., Lecture Notes in Computational Science and Engineering, Springer. pages 122

[28] Andersson, J., Kozma, A., Gillis, J., and Diehl, M. *CasADi User Guide, version 1.7*. Electrical Engineering Department (ESAT-SCD) and Optimization in Engineering Center (OPTEC), K.U. Leuven, 2012. pages 71, 99, 126, 129

[29] Andreani, R., Birgin, E. G., Martínez, J. M., and Schuverdt, M. L. On Augmented Lagrangian Methods with General Lower-Level Constraints. *SIAM Journal on Optimization 18*, 4 (2007), 1286–1309. pages 58

[30] Azmy, Y. Y. Post-convergence automatic differentiation of iterative schemes. *Nuclear Science and Engineering 125* (1997), 12–18,. pages 35

[31] Bauer, I. *Numerische Verfahren zur Lösung von Anfangswertaufgaben und zur Generierung von ersten und zweiten Ableitungen mit Anwendungen bei Optimierungsaufgaben in Chemie und Verfahrenstechnik*. PhD thesis, Universität Heidelberg, 1999. pages 44

[32] Beazley, D. M. Automated scientific software scripting with SWIG. *Future Gener. Comput. Syst. 19* (2003), 599–609. pages 72

[33] Bell, B. M., and Burke, J. V. Algorithmic Differentiation of Implicit Functions and Optimal Values. In *Advances in Automatic Differentiation* (2008), C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, and J. Utke, Eds., pp. 67–77. pages 35

[34] Bellman, R. *Dynamic programming*. Princeton University Press, 1957. pages 2

[35] Benson, D. *A Gauss pseudospectral transcription for optimal control*. PhD thesis, Massachusetts Institute of Technology. Dept. of Aeronautics and Astronautics, 2005. pages 22

[36] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)* (2010). pages 31

[37] Bertsekas, D. *Nonlinear Programming*, 2nd ed. Athena Scientific, 1999. pages 47

[38] Bertsekas, D. *Dynamic Programming and Optimal Control*, 3rd ed., vol. 2. Athena Scientific, 2007. pages 2

[39] Betts, J. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*, 2nd ed. SIAM, 2010. pages 19

[40] Biegler, L. Solution of dynamic optimization problems by successive quadratic programming and orthogonal collocation. *Computers and Chemical Engineering 8* (1984), 243–248. pages 2

[41] Biegler, L. Advances in nonlinear programming concepts for process control. *Journal of Process Control 8*, 5 (1998), 301–311. pages 60

[42] Biegler, L. T. Simultaneous modular simulation and optimization. Tech. rep., Department of Chemical Engineering, 1983. Paper 76. pages 2

[43] Biegler, L. T. *Nonlinear Programming*. MOS-SIAM Series on Optimization. SIAM, 2010. pages 16, 19, 20, 47, 131

[44] Biegler, L. T., Nocedal, J., and Schmid, C. A reduced Hessian method for large-scale constrained optimization. *SIAM Journal of Optimization 5* (1995), 314–347. pages 61, 63

[45] Bischof, C., Carle, A., Corliss, G., Griewank, A., and Hovland, P. ADIFOR Generating derivative codes from Fortran programs. *Scientific Programming 1* (1992), 11–29. pages 31

[46] Bischof, C. H., and Bücker, H. M. Computing derivatives of computer programs. In *Modern Methods and Algorithms of Quantum Chemistry: Proceedings, Second Edition*, J. Grotendorst, Ed., vol. 3 of *NIC Series*. NIC-Directors, Jülich, 2000, pp. 315–327. pages 30

[47] Bischof, C. H., Bücker, H. M., Lang, B., Rasch, A., and Vehreschild, A. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In *Proceedings of the Second IEEE International Workshop on Source*

Code Analysis and Manipulation (SCAM 2002) (2002), pp. 65–72. pages 31

[48] Bock, H. Numerical Solution of Nonlinear Multipoint Boundary Value Problems with Applications to Optimal Control. *Zeitschrift für Angewandte Mathematik und Mechanik 58* (1978), 407. pages 11

[49] Bock, H. Recent advances in parameter identification techniques for ODE. In *Numerical Treatment of Inverse Problems in Differential and Integral Equations*, P. Deuflhard and E. Hairer, Eds. Birkhäuser, Boston, 1983. pages 44, 60

[50] Bock, H., Eich, E., and Schlöder, J. Numerical Solution of Constrained Least Squares Boundary Value Problems in Differential-Algebraic Equations. In *Numerical Treatment of Differential Equations*, K. Strehmel, Ed. Teubner, Leipzig, 1988. pages 3

[51] Bock, H., and Plitt, K. A multiple shooting algorithm for direct solution of optimal control problems. In *Proceedings 9th IFAC World Congress Budapest* (1984), Pergamon Press, pp. 242–247. pages 3, 14, 53, 60

[52] Boyd, S., and Vandenberghe, L. *Convex Optimization*. University Press, Cambridge, 2004. pages 48, 51

[53] Bryson, A., and Ho, Y. *Applied optimal control: optimization, estimation, and control*. Blaisdell, Waltham, MA, 1969. pages 10, 19, 20

[54] Bulirsch, R. Die Mehrzielmethode zur numerischen Lösung von nichtlinearen Randwertproblemen und Aufgaben der optimalen Steuerung. Tech. rep., Carl-Cranz-Gesellschaft, Oberpfaffenhofen, 1971. pages 11

[55] Büskens, C., and Wassel, D. *Modeling and Optimization in Space Engineering*. Springer Verlag, 2012, ch. The ESA NLP Solver WORHP. pages 58, 75

[56] Byrd, R. H., Nocedal, J., and Waltz, R. A. KNITRO: An integrated package for nonlinear optimization. In *Large Scale Nonlinear Optimization* (2006), G. Pillo and M. Roma, Eds., Springer Verlag, pp. 35–59. pages 58, 75

[57] Cao, Y., Li, S., Petzold, L., and Serban, R. Adjoint Sensitivity Analysis for Differential-Algebraic Equations: The Adjoint DAE System and its Numerical Solution. *SIAM Journal on Scientific Computing 24*, 3 (2003), 1076–1089. pages 20, 44, 101

[58] Casella, F., Donida, F., Åkesson, J., and Donida, F. Object-oriented modeling and optimal control: A case study in power plant start-up. In *Proc. 18th IFAC World Congress* (2011). pages 122

[59] Cellier, F., and Kofman, E. *Continuous system simulation*. Springer, 2010. pages 54, 101

[60] Cerf, M., Haberkorn, T., and Trelat, E. Continuation from a flat to a round earth model in the coplanar orbit transfer problem. *Optimal Control Applications and Methods 33*, 6 (November/December 2012), 654–675. pages 8, 9, 10, 130

[61] Cervantes, A. M., Wächter, A., Tutuncu, R., and Biegler, L. T. A reduced space interior point strategy for optimization of differential algebraic systems. *Computers and Chemical Engineering 24* (2000), 39–51. pages 61, 63

[62] Christianson, B. Reverse Accumulation and Implicit Functions. *Optimization Methods and Software 9*, 4 (1998), 307–322. pages 35

[63] Coleman, T. F., and Verma, A. ADMAT: An Automatic Differentiation Toolbox for MATLAB. Tech. rep., Computer Science Department, Cornell University, 1998. pages 31

[64] Conn, A., Gould, N., and Toint, P. *Trust-Region Methods*. MPS/SIAM Series on Optimization. SIAM, Philadelphia, USA, 2000. pages 53, 58

[65] Conn, A. R., Gould, N. I. M., and Toint, P. L. A globally convergent Augmented Lagrangian algorithm for optimization with general constraints and simple bounds. *SIAM Journal on Numerical Analysis 28* (1991), 545–572. pages 58

[66] Curtis, A., Powell, M., and Reid, J. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl. 13* (1974), 117–119. pages 32

[67] Cuthrell, J., and Biegler, L. Simultaneous optimization and solution methods for batch reactor profiles. *Computers and Chemical Engineering 13*, 1/2 (1989), 49–62. pages 2

[68] Davis, T. *Direct Methods for Sparse Linear Systems*. SIAM, 2006. pages 51, 98

[69] Demmel, J. W. *Applied Numerical Linear Algebra*. SIAM, 1997. pages 76

[70] Diehl, M. *Real-Time Optimization for Large Scale Nonlinear Processes*. PhD thesis, Universität Heidelberg, 2001. http://www.ub.uni-heidelberg.de/archiv/1659/. pages 112

[71] Diehl, M., Bock, H., Schlöder, J., Findeisen, R., Nagy, Z., and Allgöwer, F. Real-time optimization and Nonlinear Model Predictive Control of Processes governed by differential-algebraic equations. *Journal of Process Control 12*, 4 (2002), 577–585. pages 60

[72] Domahidi, A., Zgraggen, A., Zeilinger, M., Morari, M., and Jones, C. Efficient Interior Point Methods for Multistage Problems Arising in Receding Horizon Control. In *IEEE Conference on Decision and Control (CDC)* (Maui, HI, USA, Dec. 2012), pp. 668 – 674. pages 51

[73] Fabien, B. dsoa: The implementation of a dynamic system optimization algorithm. *Optimal Control Applications and Methods 31* (2010), 231–247. pages 3

[74] Farrell, P. E., David, Ham, A., Funke, S. W., and Rognes, M. E. Automated derivation of the adjoint of high-level transient finite element programs. *SIAM Journal on Scientific Computing 35* (2013), C369–C393. pages 127

[75] Ferreau, H. qpOASES – An Open-Source Implementation of the Online Active Set Strategy for Fast Model Predictive Control. In *Proceedings of the Workshop on Nonlinear Model Based Control – Software and Applications, Loughborough* (2007), pp. 29–30. pages 51, 75

[76] Fertin, G., Raspaud, A., and Reed, B. On star coloring of graphs. In *Graph-Theoretic Concepts in Computer Science, 27th International Workshop, Lecture Notes in Comput. Sci. 2204,*. Springer-Verlag, Berlin, 2001, p. 140âĂŞ153. pages 33

[77] Findeisen, R., and Allgöwer, F. An Introduction to Nonlinear Model Predictive Control. In *Proc. of 21st Benelux Meeting on Systems and Control* (2002). pages 122

[78] Fletcher, R., and Leyffer, S. Nonlinear programming without a penalty function. *Mathematical Programming 91* (2002), 239–269. pages 53, 58

[79] Forth, S. A. An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in MATLAB. *ACM Transactions on Mathematical Software (TOMS) 32* (2006), 195–222. pages 31

[80] Frasch, J., Wirsching, L., Sager, S., and Bock, H. Mixed-Level Iteration Schemes for Nonlinear Model Predictive Control. In *Proceedings of the IFAC Conference on Nonlinear Model Predictive Control* (2012). pages 61, 63

[81] Frasch, J. V., Vukov, M., Ferreau, H. J., and Diehl, M. A dual Newton strategy for the efficient solution of sparse quadratic programs arising in SQP-based nonlinear MPC. In *Proceedings of the 52nd Conference on Decision and Control (CDC)* (2013). (submitted). pages 51

[82] Fritzson, P. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley Interscience, 2006. pages 54, 101

[83] Galassi, M., et al. *GNU Scientific Library Reference Manual (3rd Ed.).* pages 43

[84] Gay, D. M. Automatic Differentiation of Nonlinear AMPL Models. In *In Automatic Differentiation of Algorithms: Theory, implementation and application* (1991), SIAM, pp. 61–73. pages 4, 35, 75

[85] Gay, D. M. Writing .nl files. Tech. rep., Optimization and Uncertainty Estimation, Sandia National Laboratories, 2005. pages 75

[86] Gebremedhin, A. H., Manne, F., and Pothen, A. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review 47* (2005), 629–705. pages 33, 82

[87] Gertz, E., and Wright, S. Object-Oriented Software for Quadratic Programming. *ACM Transactions on Mathematical Software 29*, 1 (2003), 58–81. pages 50, 75

[88] Giering, R., and Kaminski, T. Automatic sparsity detection implemented as a source-to-source transformation. In *Lecture Notes in Computer Science*, vol. 3994. Springer Berlin Heidelberg, 2006, pp. 591–598. pages 33

[89] Giles, M. Collected matrix derivative results for forward and reverse mode algorithmic differentiation. In *Advances in Automatic Differentiation*. Springer, 2008, pp. 35–44. pages xv, 29, 30, 74, 80

[90] Giles, M. B. An extended collection of matrix derivative results for forward and reverse mode automatic differentiation. Tech. rep., Oxford University Computing Laboratory, 2008. pages 29, 80

[91] Gill, P., Murray, W., and Saunders, M. SNOPT: An SQP algorithm for large-scale constrained optimization. Tech. rep., Numerical Analysis Report 97-2, Department of Mathematics, University of California, San Diego, La Jolla, CA, 1997. pages 58

[92] Gill, P., Murray, W., and Saunders, M. SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization. *SIAM Review 47*, 1 (2005), 99–131. pages 58

[93] Gill, P., Murray, W., Saunders, M., and Wright, M. User's guide for NPSOL 5.0: A fortran package for nonlinear programming. Technical report sol 86-6, Stanford University, 2001. pages 58

[94] Gillis, J., and Diehl, M. A Positive Definiteness Preserving Discretization Method for nonlinear Lyapunov Differential Equations. In *Proceedings of the 52nd IEEE Conference on Decision and Control* (2013). pages 113

[95] Gillis, J., Geebelen, K., Sternberg-Kaletta, J., Gros, S., Houska, B., and Diehl, M. Lyapunov based design of robust linear-feedback for time-optimal periodic quadcopter motions. In *Proc. Benelux Meeting on Systems and Control 2012* (2012). pages 113

[96] Gillis, J., Goos, J., Geebelen, K., Swevers, J., and Diehl, M. Optimal periodic control of power harvesting tethered airplanes: how to fly fast without wind and without propellor? *American Control Conference* (2012). pages 113

[97] Gould, N., Orban, D., and Toint, P. GALAHAD, a library of thread-safe Fortran 90 packages for large-scale nonlinear optimization. *ACM Transactions on Mathematical Software 29*, 4 (2004), 353–372. pages 58

[98] Grabner, M., Pock, T., Gross, T., and Kainz, B. Automatic Differentiation for GPU-Accelerated 2D/3D Registration. In *Recent Advances in Algorithmic Differentiation*. Springer-Verlag Berlin Heidelberg, 2008. pages 85

[99] Grant, M. *Disciplined Convex Programming*. PhD thesis, Stanford University, 2004. pages 112, 127

[100] Grant, M., and Boyd, S. Graph implementations for nonsmooth convex programs, Recent Advances in Learning and Control. *Lecture Notes in Control and Information Sciences, Springer* (2008), 95–110. pages 112, 127

[101] Griewank, A. *Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation*. No. 19 in Frontiers in Appl. Math. SIAM, Philadelphia, 2000. pages 33

[102] Griewank, A., Juedes, D., Mitev, H., Utke, J., Vogel, O., and Walther, A. ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++. Tech. rep., Technical University of Dresden, Institute of Scientific Computing and Institute of Geometry, 1999. Updated version of the paper published in *ACM Trans. Math. Software* 22, 1996, 131–167. pages 31, 76

[103] Griewank, A., and Mitev, C. Detecting Jacobian sparsity patterns by Bayesian probing. *Mathematical Programming 93*, 1 (June 2002), 1–25. pages 82

[104] Griewank, A., and Walther, A. *Evaluating Derivatives*, 2 ed. SIAM, 2008. pages 23, 26, 28, 29, 32, 74, 77

[105] Gros, S., and Diehl, M. *Airborne Wind Energy*. Springer, 2013, ch. Modeling of Airborne Wind Energy Systems in Natural Coordinates. pages 113

[106] Grover, P., and Andersson, C. Optimized Three-Body Gravity Assists and Manifold Transfers in End-to-End Lunar Mission Design. In *In Proc. AAS/AIAA Space Flight Mechanics Meeting* (January 2012). pages 113

[107] Hairer, E., Nørsett, S., and Wanner, G. *Solving Ordinary Differential Equations I*, 2nd ed. Springer Series in Computational Mathematics. Springer, Berlin, 1993. pages 20, 43

[108] Hairer, E., Nørsett, S., and Wanner, G. *Solving Ordinary Differential Equations II – Stiff and Differential-Algebraic Problems*, 2nd ed. Springer Series in Computational Mathematics. Springer, Berlin, 1996. pages 43

[109] Han, S. Superlinearly Convergent Variable-Metric Algorithms for General Nonlinear Programming Problems. *Mathematical Programming 11* (1976), 263–282. pages 50

[110] Han, S. P. A Globally Convergent Method for Nonlinear Programming. *JOTA 22* (1977), 297–310. pages 2, 50

[111] Hart, W., Watson, J.-P., and Woodruff, D. Pyomo: Modeling and Solving Mathematical Programs in Python. *Mathematical Programming Computation 3*, 3 (August 2011), 1–42. pages 75

[112] Haugen, J., and Imsland, L. Optimization-Based Autonomous Remote Sensing of Surface Objects Using an Unmanned Aerial Vehicle. In *Proc. 2013 European Control Conference (ECC)* (2013). pages 113

[113] Hindmarsh, A., Brown, P., Grant, K., Lee, S., Serban, R., Shumaker, D., and Woodward, C. SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers. *ACM Transactions on Mathematical Software 31* (2005), 363–396. pages 12, 37, 43, 45, 46, 75, 97, 98, 120

[114] Hirota, T., et al. Identification of Small Molecule Activators of Cryptochrome. *Science 337*, 6098 (2012), 1094–1097. See also Supplementary Materials to the article. pages 113

[115] Houska, B. *Robust Optimization of Dynamic Systems*. PhD thesis, Katholieke Universiteit Leuven, 2011. (ISBN: 978-94-6018-394-2). pages 3

[116] Houska, B., and Ferreau, H. ACADO Toolkit User's Manual. http://www.acadotoolkit.org, 2009–2011. pages 3

[117] Houska, B., Ferreau, H., and Diehl, M. An Auto-Generated Real-Time Iteration Algorithm for Nonlinear MPC in the Microsecond Range. *Automatica 47*, 10 (2011), 2279–2285. pages 3, 68

[118] IBM Corp. *IBM ILOG CPLEX V12.1, User's Manual for CPLEX*, 2009. pages 50, 75

[119] Karush, W. Minima of Functions of Several Variables with Inequalities as Side Conditions. Master's thesis, Department of Mathematics, University of Chicago, 1939. pages 49

[120] Kharche, R. V., and Forth, S. A. Source Transformation for MATLAB Automatic Differentiation. *International Conference on Computational Science - ICCS* (2006), 558–565. pages 31

[121] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.2*, 2012. pages 85, 90

[122] Kirches, C. *Fast Numerical Methods for Mixed-Integer Nonlinear Model-Predictive Control*. PhD thesis, Ruprecht-Karls-Universität Heidelberg, 2010. pages 3, 51, 65

[123] Kirches, C., Wirsching, L., Bock, H., and Schlöder, J. Efficient Direct Multiple Shooting for Nonlinear Model Predictive Control on Long Horizons. *Journal of Process Control 22*, 3 (2012), 540–550. pages 61, 63

[124] Kögel, M., and Findeisen, R. Fast predictive control of linear systems combining Nesterov's gradient method and the method of multipliers. In *50th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC)* (2011). pages 68

[125] Kozma, A., Andersson, J., Savorgnan, C., and Diehl, M. Distributed Multiple Shooting for Optimal Control of Large Interconnected Systems. In *Proceedings of the International Symposium on Advanced Control of Chemical Processes* (2012). pages 113

[126] Kühl, P., Diehl, M., Kraus, T., Schlöder, J. P., and Bock, H. G. A real-time algorithm for moving horizon state and parameter estimation. *Computers & Chemical Engineering 35*, 1 (2011), 71–83. pages 60

[127] Kuhn, H., and Tucker, A. Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability* (Berkeley, 1951), J. Neyman, Ed., University of California Press. pages 49

[128] Larsson, P.-O., Casella, F., Magnusson, F., Andersson, J., Diehl, M., and Åkesson, J. A Framework for Nonlinear Model-Predictive Control Using Object-Oriented Modeling with a Case Study in Power Plant Start-Up. In *Proc. 2013 IEEE Multi-conference on Systems and Control* (2013). pages 113, 122, 124

[129] Lattner, C., and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization* (2004). pages 84

[130] Lee, E., and Markus, L. *Foundations of Optimal Control Theory*. Wiley, New York, 1968. pages 2

[131] Leineweber, D. *Efficient reduced SQP methods for the optimization of chemical processes described by large sparse DAE models*, vol. 613 of *Fortschritt-Berichte VDI Reihe 3, Verfahrenstechnik*. VDI Verlag, Düsseldorf, 1999. pages 3, 19, 20, 21, 22, 51, 53, 61, 63

[132] Leineweber, D., Bauer, I., Schäfer, A., Bock, H., and Schlöder, J. An Efficient Multiple Shooting Based Reduced SQP Strategy for Large-Scale Dynamic Process Optimization (Parts I and II). *Computers and Chemical Engineering 27* (2003), 157–174. pages 61, 63

[133] Lennernäs, B. A CasADi Based Toolchain For JModelica.org. Master's thesis, Department of Automatic Control, Lund University, 2013. pages 113

[134] Lin, Y. L., and Skiena, S. S. Algorithms for square roots of graphs. *SIAM J. Discrete Math. 8* (1995), 99–118. pages 33

[135] Logg, A., Mardal, K.-A., Wells, G. N., et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. pages 127

[136] Logg, A., Wells, G. N., and Hake, J. *DOLFIN: a C++/Python Finite Element Library, Automated Solution of Differential Equations by the Finite Element Methods*, vol. 84 of *Lecture Notes in Computational Science and Engineering*. Springer, 2012, ch. 10. pages 127

[137] Lucia, S., Andersson, J., Brandt, H., Schild, A., Diehl, M., and Engell, S. Efficient and Non-conservative Robust Economic Nonlinear Model Predictive Control of an Industrial Batch Reactor. Tech. rep., Process Dynamics and Operations Group, Technische Universität Dortmund, 2013. pages 113, 114, 120, 121

[138] Lucia, S., and Engell, S. Stability Properties of Multi-Stage Robust Nonlinear Model Predictive Control. Tech. rep., Process Dynamics and Operations Group, Technische Universität Dortmund, 2012. pages 118

[139] Lucia, S., Finkler, T., Basak, B., and Engell, S. A new Robust NMPC Scheme and its Application to a Semi-batch Reactor Example. In *Proc. of the International Symposium on Advanced Control of Chemical Processes* (2012), pp. 69–74. pages 114, 118

[140] Lucia, S., Finkler, T., and Engell, S. Multi-stage Nonlinear Model Predictive Control Applied to a Semi-batch Polymerization Reactor under Uncertainty. Tech. rep., Process Dynamics and Operations Group, Technische Universität Dortmund, 2013. pages 114, 118, 120

[141] Lundahl, K., Berntorp, K., Olofsson, B., Åslund, J., and Nielsen, L. Studying the Influence of Roll and Pitch Dynamics in Optimal Road-Vehicle Maneuvers. In *Proc. 23rd International Symposium on Dynamics of Vehicles on Roads and Tracks, Qingdao, China* (August 2013). pages 113

[142] Lynn, L. L., Parkin, E. S., and Zahradnik, R. L. Near-optimal control by trajectory approximations. *I&EC Fundamentals 9*, 1 (1970), 58–63. pages 2

[143] Lynn, L. L., and Zahradnik, R. L. The use of orthogonal polynomials in the near-optimal control of distributed systems by trajectory approximation. *Int. J. Control 12*, 6 (1970), 1079–1087. pages 2

[144] Magnusson, F. Collocation Methods in JModelica.org. Master's thesis, Department of Automatic Control, Lund University, Sweden, 2012. pages xv, 113, 123, 124

[145] Maly, T., and Petzold, L. R. Numerical methods and software for sensitivity analysis of differential-algebraic systems. *Applied Numerical Mathematics 20*, 1–2 (February 1996), 57–79. pages 44

[146] Maree, J. P., , and Imsland, L. Multi-objective Predictive Control for Non Steady-State Operation. In *Proc. 2013 European Control Conference (ECC)* (2013). pages 113

[147] Monagan, M. B., and Neuenschwander, W. M. GRADIENT: algorithmic differentiation in Maple. In *ISSAC '93: Proceedings of the 1993 international symposium on Symbolic and algebraic computation* (1993). pages 34

[148] Morrison, D. D., Riley, J. D., and Zancanaro, J. F. Multiple shooting method for two-point boundary value problems. *Communications of the ACM 5*, 12 (1962), 613–614. pages 11

[149] Nalum, K. Modeling and Dynamic Optimization in Oil Production. Master's thesis, Department of Engineering Cybernetics, Norwegian University of Science and Technology, 2013. pages 113

[150] Narayanan, S. H. K., Norris, B., Hovland, P., Nguyen, D. C., and Gebremedhin, A. H. Sparse Jacobian Computation Using ADIC2 and ColPack. *Procedia Computer Science 4* (2011), 2115 – 2123. pages 33

[151] Nocedal, J., and Wright, S. *Numerical Optimization*, 2 ed. Springer Series in Operations Research and Financial Engineering. Springer, 2006. pages 47, 49, 51, 53

[152] Özyurt, D. B., and Barton, P. I. Cheap Second Order Directional Derivatives of Stiff ODE Embedded Functionals. *SIAM Journal on Scientific Computing 26* (2005), 1725–1743. pages 99

[153] Pantelides, C. The Consistent Initialization of Differential-Algebraic Systems. *SIAM J. Sci. and Stat. Comput. 9*, 2 (1988), 213–231. pages 101

[154] Petzold, L. A Description of DASSL: A Differential-Algebraic System Solver. In *Proc. 10th IMACS World Congress* (Montreal, 1982). pages 12, 43

[155] Pontryagin, L., Boltyanski, V., Gamkrelidze, R., and Miscenko, E. *The Mathematical Theory of Optimal Processes*. Wiley, Chichester, 1962. pages 2, 10

[156] Powell, M. A fast algorithm for nonlinearly constrained optimization calculations. In *Numerical Analysis Dundee* (Berlin, 1977), G. e. Watson, Ed., vol. 3 of *Springer Verlag*, pp. 144–157. pages 2, 50

[157] PSE. *gPROMS Advanced User's Manual*. London, 2000. pages 4

[158] Rao, C., Rawlings, J., and Lee, J. Constrained linear state estimation - a moving horizon approach. *Automatica 37*, 2 (2001), 1619–1628. pages 60

[159] Rao, C., Wright, S., and Rawlings, J. Application of Interior-Point Methods to Model Predictive Control. *Journal of Optimization Theory and Applications 99* (1998), 723–757. pages 60

[160] Richter, S., Jones, C., and Morari, M. Real-time input-constrained MPC using fast gradient methods. In *Proceedings of the IEEE Conference on Decision and Control, Shanghai, China* (2009). pages 67, 68

[161] RWTH Aachen, Germany. *DyOS User Manual*, 2.1 ed., 2002. pages 3

[162] Savorgnan, C., Kozma, A., Andersson, J., and Diehl, M. Adjoint-Based Distributed Multiple Shooting for Large-Scale Systems. In *18th IFAC World Congress* (2011), vol. 18. pages 113

[163] Savorgnan, C., Romani, C., Kozma, A., and Diehl, M. Multiple shooting for distributed systems with applications in hydro electricity production. *Journal of Process Control 21* (2011), 738–745. pages 100

[164] Schlöder, J. *Numerische Methoden zur Behandlung hochdimensionaler Aufgaben der Parameteridentifizierung*, vol. 187 of *Bonner Mathematische Schriften*. Universität Bonn, Bonn, 1988. pages 53

[165] Scokaert, P. O. M., and Mayne, D. Q. Min-max feedback model predictive control for constrained linear systems. *IEEE Transactions on Automatic Control 43* (1998), 1136–1142. pages 118

[166] Shitahun, A., Ruge, V., Gebremedhin, M., Bachmann, B., Eriksson, L., Andersson, J., Diehl, M., and Fritzson, P. Model-Based Dynamic Optimization with OpenModelica and CasADi. In *7th IFAC Symposium on Advances in Automotive Control* (2013). pages 113

[167] Sontag, E. *Mathematical Control Theory: Deterministic Finite Dimensional Systems*, second edition ed. No. 6 in Textbooks in Applied Mathematics. Springer-Verlag, New York, 1998. pages 2

[168] St John, P. C., and Doyle III, F. J. Estimating confidence intervals in predicted responses for oscillatory biological models. *BMC systems biology 7* (2013), 71. pages 113

[169] Tran-Dinh, Q. *Sequential Convex Programming and Decomposition Approaches for Nonlinear Optimization*. Phd thesis, Arenberg Doctoral School, KU Leuven, Department of Electrical Engineering (ESAT/SCD) and Optimization in Engineering Center, Kasteelpark Arenberg 10, 3001-Heverlee, Belgium, November 2012. pages 52, 127

[170] Van Loock, W., Bellens, S., Pipeleers, G., Schutter, J. D., and Swevers, J. Time-optimal parking and flying: Solving path following problems efficiently. In *Proceedings of the IEEE International Conference on Mechatronics* (2013). pages 113

[171] Van Loock, W., Pipeleers, G., and Swevers, J. Time-optimal path planning for differentially flat systems with application to a wheeled mobile robot. In *Proc. International Workshop on Robot Motion and Control* (2013). pages 113

[172] Van Loock, W., Pipeleers, G., and Swevers, J. Time-optimal quadrotor fight. In *Proc. 2013 European Control Conference (ECC)* (2013). pages 113

[173] Vanderbei, R. J. LOQO: An interior point code for quadratic programming. *Optimization Methods and Software 11* (1999), 451–484. pages 58

[174] Verma, A. *Structured automatic differentiation*. PhD thesis, Cornell University, 1998. pages 29, 31

[175] Villard, D., and Monagan, M. B. Automatic Differentiation: An Implementation in Maple, 1998. pages 34

[176] Volckaert, M., and Swevers, J. A flexible, efficient software package for nonlinear ILC. In *31st Benelux Meeting on Systems and Control* (2012). pages 113

[177] von Stryk, O. *User's Guide for DIRCOL*. Fachgebiet Simulation und Systemoptimierung (SIM), Technische Universität Darmstadt, 2000. pages 3

[178] Vukov, M., Domahidi, A., Ferreau, H. J., Morari, M., and Diehl, M. Auto-generated Algorithms for Nonlinear Model Predicitive Control on Long and on Short Horizons. In *Proceedings of the 52nd Conference on Decision and Control (CDC)* (2013). (accepted). pages 61

[179] Wächter, A., and Biegler, L. On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming. *Mathematical Programming 106*, 1 (2006), 25–57. pages 4, 52, 58, 75, 89, 112, 120, 132

[180] Wächter, A., and Biegler, L. T. Line Search Filter Methods for Nonlinear Programming: Local Convergence. *SIAM Journal on Optimization 16* (2006), 32–48. pages 53

[181] Wächter, A., and Biegler, L. T. Line Search Filter Methods for Nonlinear Programming: Motivation and Global Convergence. *SIAM Journal on Optimization 16* (2006), 1–31. pages 53

[182] Walraven, D., Laenen, B., and D'haeseleer, W. Optimum configuration of plate-type heat exchangers for the use in orcs for low-temperature geothermal heat sources. In *Proceedings of the European Geothermal Congress 2013* (2013). pages 113

[183] Welsh, D. J. A., and Powell, M. B. An upper bound for the chromatic number of a graph and its application to timetabling problems. *Comput. J. 10* (1967), 85–86. pages 82

[184] Wikipedia. Fold (higher-order function) — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Fold_(higher-order_function)`. [Online; accessed 5-June-2013]. pages 90

[185] Wikipedia. Functional programming — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Functional_programming`, 2013. [Online; accessed 5-June-2013]. pages 90

[186] Wikipedia. Monad (functional programming) — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Monad_(functional_programming)`, 2013. [Online; accessed 5-June-2013]. pages 90

[187] Wikipedia. Operator overloading — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Operator_overloading`, 2013. [Online; accessed 5-June-2013]. pages 76

[188] Wikipedia. Shallow water equations — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Shallow_water_equations`, 2013. [Online; accessed 5-June-2013]. pages 107

[189] Wikipedia. Turing completeness — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Turing_completeness`, 2013. [Online; accessed 5-June-2013]. pages 90

[190] Wilson, R. *A simplicial algorithm for concave programming*. PhD thesis, Harvard University, 1963. pages 2, 50

[191] Wirsching, L., Bock, H. G., and Diehl, M. Fast NMPC of a chain of masses connected by springs. In *Proceedings of the IEEE International Conference on Control Applications, Munich* (2006), pp. 591–596. pages xiii, 68, 69

[192] Zanon, M., Gros, S., Andersson, J., and Diehl, M. Airborne Wind Energy Based on Dual Airfoils. *IEEE Transactions on Control Systems Technology 21* (July 2013). pages 113

Curriculum vitae

Joel Arvid Emanuel Andersson was born in Kungälv, Sweden in 1982. He went to Chalmers University of Technology (Gothenburg, Sweden) where he graduated in 2008 with degrees in Engineering Physics and Engineering Mathematics. His master's thesis was on simulation of chromatographic columns at the Forschungszentrum Jülich in Germany. From October 2008 until September 2013 he pursued a PhD degree in Engineering at the Katholieke Universiteit Leuven (Belgium) under the main supervision of prof. Moritz Diehl. The results of this research are summarized in this dissertation.

List of publications

Journal papers

1. M. Zanon, S. Gros, J. Andersson and M. Diehl, *Airborne Wind Energy Based on Dual Airfoils. IEEE Transactions on Control Systems Technology 21* (2013)

2. N. Ahlbrink, J. Andersson, M. Diehl and R. Pitz-Paal, *Optimization of the Mass Flow Rate Distribution of an Open Volumetric Air Receiver.* J. Sol. Energy Eng. 135(4) (2013)

3. E. von Lieres, J. Andersson, A fast and accurate solver for the general rate model of column liquid chromatography. *Computers & chemical engineering 34*, (2010), 1180–1191

Book chapters

1. J. Andersson, J. Åkesson and M. Diehl, *CasADi – A symbolic package for automatic differentiation and optimal control.* In *Recent Advances in Algorithmic Differentiation*, S. Forth et al. Eds. Berlin, 2012.

Conference papers

1. P-O. Larsson, F. Casella, F. Magnusson, J. Andersson, M. Diehl, J. Åkesson, *A Framework for Nonlinear Model-Predictive Control Using Object-Oriented Modeling with a Case Study in Power Plant Start-Up.* In *2013 IEEE Multi-conference on Systems and Control* (2013)

2. A. Shitahun, V. Ruge, M. Gebremedhin, B. Bachmann, L. Eriksson, J. Andersson, M. Diehl, P. Fritzson, *Tool Demonstration Abstract: OpenModelica and CasADi for Model-Based Dynamic Optimization.* In

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (2013)

3. J. Andersson, J. Åkesson and M. Diehl, *Dynamic optimization with CasADi*. In *Proceedings of the 51st IEEE Conference on Decision and Control*, Maui (2012)

4. A. Kozma, J. Andersson, C. Savorgnan and M. Diehl, *Distributed Multiple Shooting for Optimal Control of Large Interconnected Systems*. In Proceedings of the International Symposium on Advanced Control of Chemical Processes (2012)

5. J. Andersson, J. Åkesson, F. Casella and M. Diehl, *Integration of CasADi and JModelica.org*. In *8th International Modelica Conference* (2011)

6. C. Savorgnan, A. Kozma, J. Andersson and M. Diehl, *Adjoint-Based Distributed Multiple Shooting for Large-Scale Systems*. In *18th IFAC World Congress* (2011)

7. T. Hirsch, N. Ahlbrink, R. Pitz-Paal, C. Teixeira Boura, B. Hoffschmidt, J. Gall, D. Abel, V. Nolte, M. Wirsum, J. Andersson, M. Diehl, *Dynamic Simulationof a solar tower system with open volumetic receiver – a review on the vICERP project*. In *Proceeding of SolarPACES* (2011)

8. J. Andersson and B. Houska and M. Diehl, *Towards a Computer Algebra System with Automatic Differentiation for use with Object-Oriented modelling languages*, In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools* (2010)

9. J. Gall, D. Abel, N. Ahlbrink, R. Pitz-Paal, J. Andersson, M. Diehl, C. Teixeira Boura, M. Schmitz and B. Hoffschmidt, *Simulation and Control of Solar Thermal Power Plants*, In *International Conference on Renewable Energies and Power Quality* (2010)

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF ELECTRICAL ENGINEERING
SISTA RESEARCH GROUP AND OPTIMIZATION IN ENGINEERING CENTER (OPTEC)
Kasteelpark Arenberg 10
B-3001 Heverlee
first.name@dept.kuleuven.be
www.website.kuleuven.be