ISSN 0280-5316
ISRN LUTFD2/TFRT--5892--SE

# Collocation Methods in JModelica.org

Fredrik Magnusson

Department of Automatic Control
Lund University
February 2012

**Lund University**
**Department of Automatic Control**
**Box 118**
**SE-221 00 Lund Sweden**

*Document name*
MASTER THESIS

*Date of issue*
February 2012

*Document Number*
ISRN LUTFD2/TFRT--5892--SE

*Author(s)*
Fredrik Magnusson

*Supervisor*
Claus Führer Numerical Analysis, Lund Sweden
Johan Åkesson Automatic Control, Lund Sweden
Anders Rantzer Automatic Control, Lund Sweden (Examiner)

*Sponsoring organization*

*Title and subtitle*
Collocation methods in JModelica.org (Kollokationsmetoder i JModelica.org)

*Abstract*
In this thesis, we use CasADi to implement a new optimization algorithm in the open-source platform JModelica.org. CasADi is a tool for computing derivatives using automatic differentiation, which is tailored for optimal control. JModelica.org is a platform for simulation and optimization of physical systems created using the Modelica modeling language. The implemented optimization algorithm is based on direct collocation using Radau or Gauss collocation schemes. We provide a thorough presentation of how a dynamic optimization problem described by Modelica and Optimica code is transcribed into a nonlinear programming problem using direct collocation. This nonlinear programming problem is then solved using CasADi's interface to Ipopt, a numerical solver for optimization problems. The implemented algorithm is compared to a similar and already existing optimization algorithm in JModelica.org in five different benchmark problems, including a distillation column and a combined cycle power plant. The new algorithm compares favorably to the other JModelica.org algorithm in a majority of the cases

*Keywords*

*Classification system and/or index terms (if any)*

*Supplementary bibliographical information*

*ISSN and key title*
0280-5316

*ISBN*

*Language*
English

*Number of pages*
104

*Recipient's notes*

*Security classification*

# Preface

This master thesis has been carried out at the company Modelon in Lund in collaboration with the Department of Automatic Control at the Faculty of Engineering of Lund University. Johan Åkesson, who is affiliated with both Modelon and the Department of Automatic Control, has supervised the thesis.

I would like to thank Joel Andersson for developing CasADi and helping me get familiar with it, and also promptly addressing encountered issues with CasADi. Next I want to thank Claus Führer from Lund University for sharing his insights on differential algebraic equation systems and collocation methods. I would also like to thank everyone at Modelon, in particular the JModelica.org team, for being very helpful. Finally I would like to thank Johan Åkesson for everything.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Background

Optimization of large-scale dynamic systems, ranging from power plants to vehicle systems is becoming a standard industrial technology. JModelica.org is a tool partly aimed at large-scale optimization, which is done by formulating a *dynamic optimization problem* (DOP) and then solving it using numerical methods. The DOP is formulated with the use of the Modelica language and its extension Optimica. In this thesis we consider three different generic DOPs.

The first is the *optimal control problem* (OCP), where the aim is to find the control variable(s) that, in some sense, optimize a dynamic system, e.g. by minimizing the amount of resources spent to perform a certain action. The second is the *parameter optimization problem*, where the aim is instead to find optimal parameter values. These two problems are closely related and will receive a common problem formulation. The final DOP is the *parameter estimation problem*, where the problem is to, given some measurement data of a dynamic system, find the values of unknown model parameters that allows the model to most closely mimic the measured data.

Solving dynamic optimization problems is useful, or even necessary, in many different fields and applications. Parameter estimation is used to improve physical models. Parameter optimization aids the design process of a product with some design parameters by finding an optimal design. Optimal control has many different applications, which are often categorized as being on-line or off-line. During on-line applications, the DOP is solved during the actual control of the real system. This is typically done in the form of nonlinear model predictive control, where the system state is sampled and the sample is then used to find an optimal control trajectory for a short future time horizon. After the computed control trajectory has been used to control the system during the short horizon, the state is resampled and the process is repeated iteratively. See [RM09] for more on nonlinear model predictive control.

There are many varieties of off-line applications of optimal control. One example is finding theoretical optimal trajectories for the transition between two stationary operating conditions in a system, which can be used either as a reference during manual control or as a target for an automatic controller if combined with feedback error control. Another example is the determination of how individual components affect the overall performance of a system to identify system bottlenecks.

There are several approaches to solving DOPs. Up until the 1970s, the common way was to use *indirect* approaches, which are based on calculus of variations. However, indirect approaches do not handle inequality constraints easily and the approach is very sensitive to initial guesses of the primal and dual variables, making them ill-suited for developing domain-neutral software [Zav08].

This lead to the introduction of *direct* approaches, which try to find an approximative solution to the DOP by transcribing the infinite-dimensional DOP into a finite-dimensional *nonlinear programming* (NLP) problem. The main difference among direct approaches is how to handle the constraints corresponding to the system dynamics. The three most common direct approaches are *direct single shooting*, *direct multiple shooting* and *direct collocation*. This thesis focuses on direct collocation methods. For a description of the two other variants, see [Bie10, ch. 9] and [Bet10, ch. 3].

## 1.2 An introductory example

An example of a simple problem, albeit non-trivial to solve analytically, of the kind studied in this thesis is optimal control of the *Van der Pol* (VDP) oscillator [Kan07]. The dynamics of the VDP oscillator are given by

$$\dot{x_1}(t) = \left(1 - x_2(t)^2\right) \cdot x_1(t) - x_2(t) + u(t), \tag{1.1a}$$

$$\dot{x_2}(t) = x_1(t), \tag{1.1b}$$

where $x_1$ and $x_2$ are the states, $u$ is the control variable and $t$ is the sole independent variable: time. The considered time interval is given by $[t_0, t_f]$, where $t_0$ is the start time and $t_f$ is the final time. The goal is to find a $u$ that minimizes some cost. An example of a typical cost function for this problem is the integral

$$\int_{t_0}^{t_f} \left(x_1(t)^2 + x_2(t)^2 + u(t)^2\right) \, \mathrm{d}t.$$

The essential problem is then to drive the states $x = (x_1, x_2)$ towards the origin as quickly as possible while maintaining a small value of $u^2$ at all times. To make things a little more interesting, we can also impose the bound

$$u(t) \leq 0.75, \quad \forall t \in [t_0, t_f].$$

To demonstrate the idea of direct collocation, we now solve this problem using a simple collocation method. We discretize the time interval $[t_0, t_f]$ into $n_e = 100$

elements with the constant element length $h = \frac{t_f - t_0}{n_e}$. Let $[a..b]$, where $a, b \in \mathbb{Z}$, denote the integer interval

$$\{k \in \mathbb{Z} : a \leq k \leq b\}.$$

At the mesh points

$$t_i = t_0 + i \cdot h, \quad i \in [1..n_e],$$

we denote the value of the states, their derivatives and the control variable by $x_i = x(t_i)$, $\dot{x}_i = \dot{x}(t_i)$ and $u_i = u(t_i)$ respectively. For the discretized variables, the cost function is approximated by

$$f = \sum_{i=1}^{n_e} h_i \cdot \left( {x_1}_i^2 + {x_2}_i^2 + u_i^2 \right).$$

We require the discretized variables to satisfy the system dynamics given by (1.1), i.e.

$$\dot{x}_{1i} = (1 - {x_2}_i^2) \cdot x_{1i} - x_{2i} + u_i, \quad \forall i \in [1..n_e] \tag{1.2a}$$
$$\dot{x}_{2i} = x_{1i}, \quad \forall i \in [1..n_e] \tag{1.2b}$$

To get the values of $\dot{x}_i$, we use the backward Euler approximation. This gives us the equations

$$\dot{x}_i = \frac{x_i - x_{i-1}}{h}, \quad \forall i \in [1..n_e],$$

where $x_0$ is some specified initial condition. These are called the collocation equations. This results in the following optimization problem for the discretized variables.

$$\min f = \min \sum_{i=1}^{n_e} h_i \cdot ({x_1}_i^2 + {x_2}_i^2 + u_i^2)$$

$$\text{subject to} \begin{cases} \dot{x}_{1i} = (1 - {x_2}_i^2) \cdot x_{1i} - x_{2i} + u_i, \quad \forall i \in [1..n_e], \\ \dot{x}_{2i} = x_{1i}, \quad \forall i \in [1..n_e], \\ x_{10} = x_1(t_0), \\ x_{20} = x_2(t_0), \\ \dot{x}_{1i} = \dfrac{x_{1i} - x_{1i-1}}{h}, \quad \forall i \in [1..n_e], \\ \dot{x}_{2i} = \dfrac{x_{2i} - x_{2i-1}}{h}, \quad \forall i \in [1..n_e], \\ u_i \leq 0.75, \quad \forall i \in [1..n_e]. \end{cases}$$

With the initial conditions

$$(x_1(t_0), x_2(t_0)) = (0, 1)$$

and letting $t_0 = 0$ and $t_f = 10$, this optimization problem has the solution shown in Figure 1.1.

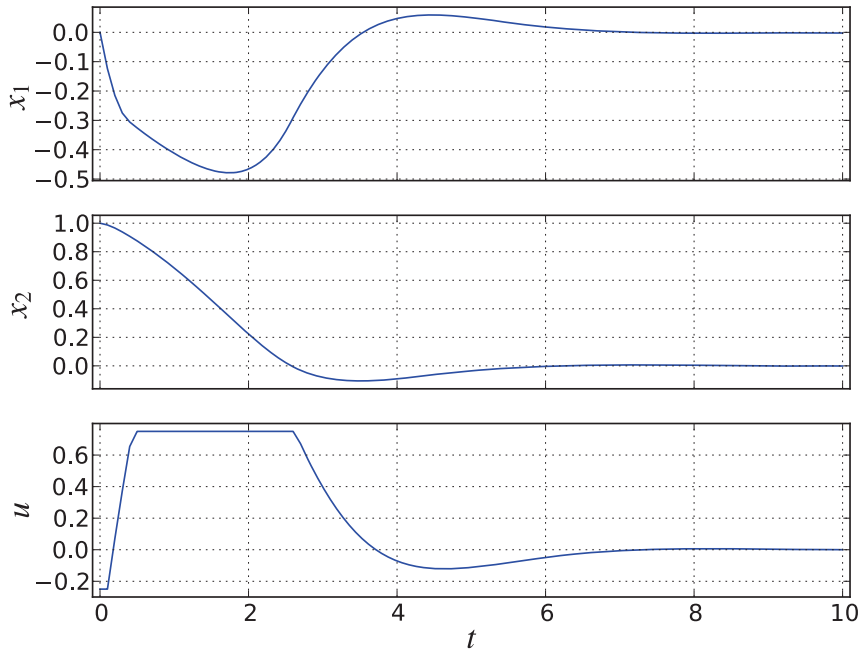


Figure 1.1: Optimal control of a VDP oscillator using backward Euler

The described transcription procedure corresponds to a specific kind of collocation method called Radau collocation (with 1 collocation point), which is described in Chapter 3. More advanced collocation methods are obtained by choosing more advanced finite difference approximations of the derivative than backward Euler.

We will in this thesis study different variants of the VDP OCP, as well as other, more industrially relevant, problems.

## 1.3 Aim of thesis

In this thesis we (further) integrate the JModelica.org platform with the novel automatic differentiation package CasADi with the purpose of implementing collocation algorithms for solution of dynamic optimization problems. The goal is that once the DOP has been formulated using Modelica and Optimica, the algorithms should be able to efficiently and accurately solve the problem with little to no additional effort from the user while still remaining robust.

The above aspiration may be a little too ambitious. In order to solve big and complex problems efficiently (or at all), some tweaking of algorithm parameters may be needed from the user. The aim is to have fully featured algorithms which allow the advanced user to solve a wide variety of advanced problems.

## 1.4 Thesis outline

The thesis starts with introducing some basic concepts related to optimization and defining the generic dynamic optimization problem in Chapter 2. In Chapter 3, we present the relevant theory behind collocation methods. In Chapter 4, the languages and software used for the implementation of the collocation algorithms are presented.

We then move on to describe how the languages and software of Chapter 4 and the theory of Chapter 3 can be combined to implement advanced collocation algorithms in Chapter 5. In Chapter 6, we present a few different problems which we solve using our implemented collocation algorithms and compare the results and performances with a similar JModelica.org optimization algorithm. The thesis is finished by evaluating these results in Chapter 7 and discussing possibilities of further development.

# Chapter 2

# Optimization

At the center of this thesis is the field of optimization of dynamic systems. Before we begin the study of collocation methods, we will introduce some basic concepts related to optimization as well as notation and nomenclature used throughout the thesis. Formal definitions of the DOPs are also given.

## 2.1 Static optimization

The goal of optimization is to minimize an *objective function*

$$f \in C^2(A, \mathbb{R}),$$

where $C^2(A, \mathbb{R})$ denotes the space of twice continuously differentiable function from $A$ into $\mathbb{R}$ and $A$ is some set. For now we assume that $A$ is a subset of $\mathbb{R}^{n_z}$, where $n_z$ is the number of optimization variables, but this assumption will be relaxed when we discuss dynamic optimization in Section 2.2.

An equivalent formulation is to instead maximize $-f$. A simple example of an optimization problem is

$$\min_{z \in \mathbb{R}} f(z) = \min_{z \in \mathbb{R}} z.$$

In this example we have let $A = \mathbb{R}$. While this problem is simple in one sense, it is impossible in another, since it has no solution. In order to make things slightly more interesting and also provide us with a solution, we introduce *variable bounds* on the form

$$A \ni z_L \leq z \leq z_U \in A. \tag{2.1}$$

For example,

$$\begin{aligned} &\min_{z \in \mathbb{R}} z, \\ &\text{subject to} \\ &z \geq z_L = 0. \end{aligned}$$

We denote the solution to the optimization problem by $z^*$, which in this case is $z^* = 0$.

Alas, there are no problems of interest to this thesis that can be formulated with just an objective function and variable bounds. Thus we introduce *constraints*, a more general form of variable bounds. Constraints can be divided into *equality constraints*, which are of the form $g_e(z) = 0$, and *inequality constraints*, which are of the form $g_i(z) \leq 0$, where

$$g_e \in C^2(A, \mathbb{R}^{m_e}) \qquad \text{and} \qquad g_i \in C^2(A, \mathbb{R}^{m_i}),$$

where $m_e$ and $m_i$ are the number of scalar-valued equality and inequality constraints respectively. We further extend the above optimization problem to exemplify this.

$$\min_{z \in \mathbb{R}} z$$
$$\text{subject to } \begin{cases} z \geq 0, \\ g_e(z) = \sin(\pi \cdot z) = 0, \\ g_i(z) = 0.5 - z^2 \leq 0. \end{cases}$$

Our previous solution of $z^* = 0$ is no longer valid, as it violates the inequality constraint. The solution is now the smallest positive root to $g_e$ that satisfies the inequality constraint, which happens to be $z^* = 1$. We will later see that there is a need to further divide constraints into additional subcategories.

We have now arrived at the general *nonlinear programming* (NLP) problem

$$\min_{z \in A} f(z) \tag{2.2a}$$
$$\text{subject to } \begin{cases} z_L \leq z \leq z_U, & \text{(2.2b)} \\ g_e(z) = 0, & \text{(2.2c)} \\ g_i(z) \leq 0. & \text{(2.2d)} \end{cases}$$

An important tool when studying such problems is the *Lagrangian function*, which is defined as

$$\Lambda(z, \lambda, \nu) = f(z) + \lambda \cdot g_e(z) + \nu \cdot g_i(z), \tag{2.3}$$

where $\lambda \in \mathbb{R}^{m_e}$ and $\nu \in \mathbb{R}^{m_i}$ are called the *dual variables*, in contrast to $z$, which is called the *primal variable*, and $\cdot$ denotes the inner product. These are then used to formulate the famous *Karush-Kuhn-Tucker* (KKT) conditions, given by

$$\nabla_z \Lambda(z^*, \lambda^*, \nu^*) = \nabla f(z^*) + \lambda^* \cdot \nabla g_e(z^*) + \nu^* \cdot \nabla g_i(z^*) = 0, \tag{2.4a}$$
$$g_e(z^*) = 0, \quad g_i(z^*) \leq 0, \tag{2.4b}$$
$$\nu^* \cdot g_i(z^*) = 0, \quad \nu^* \geq 0. \tag{2.4c}$$

Under the assumption of a constraint qualification being satisfied, which we discuss later, the existence of a $\lambda^*$ and $\nu^*$ satisfying (2.4) is necessary in order for $z^*$ to be

a local minimum. Condition (2.4a) ensures that the solution is a stationary point. Condition (2.4b) ensures the primal feasibility. Condition (2.4c) ensures that either the dual variable belonging to a scalar inequality constraint is zero, in which case the inequality constraint is said to be *inactive*, or the constraint function is equal to zero, in which case the inequality constraint is said to be *active*.

Another requirement for $z^*$ to be a local minimum is that

$$p \cdot \nabla^2_{zz}\Lambda(z^*, \lambda^*, \nu^*) \cdot p \geq 0, \quad \forall p \in \mathbb{R}^{n_z} \,:\, \nabla g_e(z^*) \cdot p = 0 \wedge p \neq 0, \tag{2.5}$$

which ensures that the solution (whose stationarity is ensured by (2.4a)) is not a local maximum. The reason we required $f, g_e$ and $g_i$ to be twice continuously differentiable is because these derivatives are needed for this condition.

As mentioned above, the KKT conditions require a constraint qualification condition. There are many sufficient constraint qualifications of varying strictness and we do not list them all here. One of the more common (and strict) constraint qualifications is the *linear independence constraint qualification*, which states that the gradients of all the equality constraints and active inequality constraints evaluated at $z^*$ should be linearly independent. This together with (2.4) and (2.5) make up all the conditions *necessary* for a point $(z^*, \lambda^*, \nu^*)$ to be a local minimum. If the inequality in (2.5) is changed to a strict inequality, then the conditions are also *sufficient* for the point to be a local minimum.

These optimality conditions are fundamental to solving NLP problems. However, in this thesis we rely on the third-party software Ipopt to solve the arising NLP problems and thus do not discuss these conditions further. For a more thorough introduction to this topic, see [NW06, ch. 12] and [Bie10, ch. 4].

## 2.2 Dynamic optimization

In this thesis we optimize dynamic systems whose dynamics are described by a *differential algebraic equation* (DAE) system. Before we discuss the DOP further, we introduce DAE systems.

### 2.2.1 Differential algebraic equation systems

A DAE system is a generalization of an *ordinary differential equation* (ODE) system. ODE systems have the general form

$$z^{(n)} = \tilde{F}(t, z(t), z'(t), z''(t), \ldots, z^{(n-1)}(t)), \quad \forall t \in [t_0, t_f],$$

where

$$z \in C^{n-1}(\mathbb{R}, \mathbb{R}^{n_z}),$$
$$\tilde{F} : \mathbb{R} \times \prod_{i=1}^{n} (\mathbb{R}^{n_z}) \to \mathbb{R}^{n_z}$$

and $n$ is called the order of the system. In this thesis we are however not interested in the general form, since all ODE systems of order $n$ can be transformed to a system of order 1 by introducing $n-1$ additional variables without loss of generality. For example, the dynamics of the VDP oscillator are often described by the second-order ODE

$$\ddot{z}(t) - (1 - z(t)^2) \cdot \dot{z}(t) + z(t) - u(t) = 0.$$

By introducing $x_1 = z$ and $x_2 = \dot{z}$, we recover (1.1), which is a first order ODE system. Thus we only consider first order systems in this thesis.

A first order DAE system has the general form

$$F(t, z(t), \dot{z}(t)) = 0, \quad \forall t \in [t_0, t_f].$$

While the difference between an ODE system and a DAE system may seem small at first glance, this is not the case. The most important difference is that purely algebraic equations of the form

$$F_i(t, z(t)) = 0,$$

where $F_i$ is a scalar component of the vector-valued $F$, fall within the definition of a DAE. This mixture of differential and algebraic equations gives rise to several challenges when solving DAE systems compared to solving ODE systems. To facilitate the analysis of DAE systems, we separate the system variables depending on whether we are interested in their derivatives. To this end, we decompose $z$ into the new variables $x$ and $w$. The variable

$$x \in C(\mathbb{R}, \mathbb{R}^{n_x})$$

contains the functions whose derivatives are a part of the DAE system and

$$w \in \mathcal{F}_b(\mathbb{R}, \mathbb{R}^{n_w})$$

contains the functions whose derivatives are *not* a part of the DAE system, where $\mathcal{F}_b(\mathbb{R}, \mathbb{R}^{n_w})$ is the space of bounded functions from $\mathbb{R}$ into $\mathbb{R}^{n_w}$. The variable $x$ is called the *differential variable* and $w$ is called the *algebraic variable*. The boundedness of $w$ is only required to give us problems which we can handle, but it is also possible to consider unbounded functions. A DAE system now instead has the general form

$$F(t, \dot{x}(t), x(t), w(t)) = 0, \quad \forall t \in [t_0, t_f], \tag{2.6}$$

where

$$F : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_w} \to \mathbb{R}^{n_x + n_w}.$$

To demonstrate the difficulties of DAE systems, consider the following very simple example from [Bet10, pg. 105], where $n_x = n_w = 1$.

$$\dot{x}(t) - w(t) = 0, \quad \forall t \in [0, t_f] \tag{2.7a}$$
$$x(t) - t = 0, \qquad \forall t \in [0, t_f]. \tag{2.7b}$$

When dealing with ODE systems, we impose initial conditions in order to obtain a well-posed problem. Let us attempt the same in (2.7) by setting $x(0) = 1$. Now, the solution to the second equation is obviously $x(t) = t$, and thus we get from the first equation $w(t) = 1$. However, our initial condition is clearly not satisfied, as $x(0) = 0 \neq 1$. In other words, the problem has no solution. This is quite different from ODE systems, as according to the Picard-Lindelöf theorem [RR04, th. 1.1 and 1.4], an ODE system together with arbitrary initial conditions will always have exactly one solution locally, given some rather lax conditions on $F$, which are satisfied in this case.

The question is then what properties are required of the initial conditions to the DAE system to get a well-posed problem. At $t_0$, we have $n_x + n_w$ equations from

$$F(t_0, \dot{x}(t_0), x(t_0), w(t_0)) = 0. \tag{2.8}$$

These equations have however $2 \cdot n_x + n_w$ unknowns. So we need an additional $n_x$ initial equations on the form

$$F_0(\dot{x}(t_0), x(t_0), w(t_0)) = 0. \tag{2.9}$$

As noted in the above example, it is important that the initial equations satisfy the DAE system at $t_0$. It is also required that none of the initial equations are equivalent to each other or any of the equations in (2.8). If $F_0$ has all of these properties, we say that the initial conditions (2.9) are *consistent*. The only initial condition consistent with (2.7) is

$$F_0(\dot{x}(0), x(0), w(0)) = x(0) = 0.$$

Trying to impose any other initial conditions will result in an ill-posed problem. So in a sense, the above example actually does not need any initial conditions to be well-posed. The initial conditions are rather imposed by the DAE system itself. This is however not always the case. Some problems have several possible consistent initial conditions, in which case one of them needs to be specified in order to achieve well-posedness.

Most literature on dynamic optimization (e.g. [Bie10], [Bet10] and [Zav08]) focus on a special case of (2.6): equation systems on the form

$$\dot{x}(t) = F_d(t, x(t), w(t)), \tag{2.10a}$$
$$0 = F_a(t, x(t), w(t)), \tag{2.10b}$$

which are called *semi-explicit DAE* systems. The semi-explicit form is useful for discussing the notion of the *index* of a DAE system. We start by differentiating (2.10b) with respect to $t$, yielding

$$0 = F_{at}' + F_{ax}' \cdot \dot{x}(t) + F_{aw}' \cdot \dot{w}(t).$$

If the matrix $F_{aw}'$ is non-singular, we get

$$\dot{w}(t) = -\left(F_{aw}'\right)^{-1} \cdot \left(F_{at}' + F_{ax}' \cdot \dot{x}(t)\right).$$

Together with (2.10a), we now have an ODE system and (2.10) is said to be a DAE system of index one. If however $F'_{a_w}$ is singular, we can differentiate (2.10b) once more. If the second differentiation results in an ODE system, the DAE system is of index two. If not, the differentiation can be repeated. The DAE system is of index $n$ if an ODE is obtained after $n$ differentiations of (2.10b). This is called the *differentiation index* of the DAE system. A DAE system of index zero is thus equivalent to an ODE system. There are other, non-equivalent, definitions of the DAE system index, but this is the only one we consider in this thesis. Note that this definition of index requires all the variables to be sufficiently differentiable (which is not necessarily the case in this thesis). If they are not, the index becomes a local property that may change at discontinuity points. See [HW96] for more on the index of DAE systems and DAE systems in general.

In this thesis we only consider DAE systems of index one or zero. We avoid the difficulties of higher-index systems by relying on the features of JModelica.org to transform the considered models into lower-index systems, essentially allowing us to also work with higher-index problems.

The conversion from (2.6) to (2.10) is always possible by introducing the additional algebraic variables

$$\tilde{w} = \dot{x},$$

resulting in

$$\begin{aligned} \dot{x}(t) &= \tilde{w}(t), \\ 0 &= F(t, \tilde{w}(t), x(t), w(t)). \end{aligned}$$

The additional variables increase the size of the model, which is not always desirable. It is often possible to convert a fully implicit DAE to a semi-explicit DAE without the introduction of additional variables. The semi-explicit formulation is particularly useful for shooting methods, but such a conversion is not always preferable for collocation methods. Thus we choose to instead work with the fully implicit DAE form (2.6) for the remainder of this thesis.

### 2.2.2 The dynamic optimization problem

For the *optimal control problem* our optimization variables are the time-dependent control variables. We do not allow the derivative of the control variables to be a part of the DAE system, so they are essentially algebraic variables. However, they are treated very differently, so we need to separate them from the algebraic variables (and the states). Thus, let $u$ denote the control variable and $n_u$ the number of such scalar variables. We introduce the term *free* to describe the variables we wish to optimize, i.e. $u$ is the free variable whereas $x$ and $w$ are the non-free variables. The idea is that once we have freely chosen the free variables, the non-free variables can be uniquely determined from the DAE system.

We now have four different time-dependent and vector-valued functions: $\dot{x}$, $x$, $u$ and $w$. Keeping these separated at all times will lead to increasingly cumbersome notation, so we reintroduce the variable $z$ as the Cartesian product of these functions, i.e.

$$\begin{aligned} z &:= (\dot{x}, x, u, w), \\ n_z &:= 2 \cdot n_x + n_u + n_w. \end{aligned} \tag{2.11}$$

For ease of notation, we do not distinguish between tuples and vectors, e.g. we do not distinguish between $\mathbb{R}^a \times \mathbb{R}^b$ and $\mathbb{R}^{a+b}$ for arbitrary positive integers $a$ and $b$.

For the *parameter optimization problem* and *parameter estimation problem*, the free variables are instead time-independent *parameters*, which we denote by $p$. Note that the DAE system describing the system dynamics may contain parameters which are not free, e.g. physical constants, and are thus not a part of $p$. If we allow both free control variables and parameters (in which case we have a combined optimal control and parameter optimization problem), our DAE systems, together with the initial conditions, now instead have the general form

$$F(t, z(t), p) = 0, \quad \forall t \in [t_0, t_f], \tag{2.12a}$$

$$F_0(z(t_0), p) = 0, \tag{2.12b}$$

where

$$\begin{aligned} F &: \mathbb{R} \times \mathbb{R}^{n_z} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_x + n_w}, \\ F_0 &: \mathbb{R}^{n_z} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_x}, \\ p &\in \mathbb{R}^{n_p}, \end{aligned}$$

where $n_p$ is the number of free parameters. Since the non-free variables ($x$ and $w$) should be determinable from the DAE system given fixed values of the free variables ($u$ and $p$), the DAE system should contain $n_x + n_w$ scalar equations. Henceforth we refer to the differential variable $x$ as the *state*, which makes sense for DAE systems of index one or zero.

With these newly introduced variables, we can formulate the general DOP as follows.

$$\min_{(z,p) \in A} f(z, p), \tag{2.13a}$$

$$\text{subject to} \begin{cases} F(t, z(t), p) = 0, & \text{(2.13b)} \\ F_0(z(t_0), p) = 0, & \text{(2.13c)} \\ z_L \leq z(t) \leq z_U, & \text{(2.13d)} \\ p_L \leq p \leq p_U, & \text{(2.13e)} \\ g_e(t, z(t), p) = 0, & \text{(2.13f)} \\ g_i(t, z(t), p) \leq 0, & \text{(2.13g)} \\ G_e(Z_e) = 0, & \text{(2.13h)} \\ G_i(Z_i) \leq 0, & \text{(2.13i)} \\ \forall t \in [t_0, t_f]. \end{cases}$$

We will now study each part of this problem individually.

The goal of the DOP is the same as any other optimization problem: minimizing some cost function

$$f \in C^2(A, \mathbb{R}).$$

In Section 2.1 we assumed that $A \subset \mathbb{R}^{n_z}$. However, as we are dealing with dynamic optimization, some of our optimization variables are now functions of time, so we now instead assume that

$$A \subset \mathcal{F}_b(\mathbb{R}, \mathbb{R}^{n_z}) \times \mathbb{R}^{n_p}.$$

Note that we now consider all the variables $z$ as optimization variables, and not just its free component $u$. This is because when we add the constraints (2.13d) to (2.13i), we can no longer choose the variables $u$ and $p$ completely arbitrarily. But the fact remains that we seek the optimal values of the free variables and the remaining variables can be uniquely determined by the constraints given values of the free variables. The form of the objective function is dependent on what kind of DOP we are solving (optimal control, parameter optimization or parameter estimation). So for now all we say is that it should be a scalar-valued and twice continuously differentiable function of the optimization variables. We discuss the objective function in more detail in Sections 2.2.3 and 2.2.4.

The rest of (2.13) is however common to all considered DOPs. Equation (2.13b) and (2.13c) are the system dynamics which enter the DOP as equality constraints. Equations (2.13d) and (2.13e) are the variable bounds corresponding to (2.2b) of the NLP problem. Note that our bounds for $z$ are time-independent and must be satisfied for all times $t \in [t_0, t_f]$.

A lot of interesting problems can be formulated with just Equations (2.13a) to (2.13e) (e.g. all the problems considered in Chapter 6). There are however times when you want your solution to have some more properties than just satisfying some variable bounds and system dynamics. This is done by imposing constraints, in addition to the equality constraints corresponding to the system dynamics, as discussed in Section 2.1. In dynamic optimization, two types of constraints appear in addition to the equality and inequality constraints. The first is on the form (2.13f), or its inequality variant (2.13g). These are called *path constraints* and are enforced during all times. The other type of constraints is called *point constraints* and are only enforced at discrete time points, called *constraint points*. These have the form (2.13h), or its inequality variant (2.13i), where $Z_e$ are the values of $z$ at all the points in time where we have a point equality constraint and $Z_i$ is defined analogously. A typical example of a point constraint is a terminal constraint, as discussed at the end of Section 2.2.3.

Note that although (2.13b) and (2.13c) are actually path equality and point equality constraints respectively, we have excluded them from the path and point equality constraints (2.13f) and (2.13h), as (2.13b) and (2.13c) are always present and are treated separately.

In Section 2.1 we required all the constraints to be twice continuously differentiable in order to establish the second-order optimality condition (2.5). It will in Chapter 3

become apparent that we are not interested in the Hessian with respect to time, so we require $g_e$ and $g_i$ to be twice continuously differentiable with respect to their second and third arguments as well as $G_e$ and $G_i$ to be twice continuously differentiable with respect to their only argument. We make however no restrictions on the continuity of $g_e$ and $g_i$ with respect to their first argument, allowing us to have time-variant path constraints, e.g. path constraints which are only enforced during certain times. We also require $F$ and $F_0$ to be twice continuously differentiable (except with respect to time) in order for all the constraints to be twice continuously differentiable.

### 2.2.3 Optimal control and parameter optimization

In the generic DOP (2.13) we allow both control variables and parameters to be optimization variables, thus covering both optimal control problems and parameter optimization problems. The general form of the objective functions for these two problem types are also identical. Thus there is no need to distinguish between the mathematical formulations of OCPs and parameter optimization problems. In this thesis we consider two types of objective functions for these two problems. The first is an objective function of the form

$$f(z, p) = \phi(t_f, z(t_f), p), \tag{2.14}$$

where $\phi \in C^2(\mathbb{R} \times \mathbb{R}^{n_z} \times \mathbb{R}^{n_p}, \mathbb{R})$. The differentiability is needed for the second-order optimality condition (2.5). This objective function is said to be in *Mayer* form. The other type we consider is of the form

$$f(z, p) = \int_{t_0}^{t_f} L(t, z(t), p)\,\mathrm{d}t,$$

where $L \in C^2(\mathbb{R} \times \mathbb{R}^{n_z}, \mathbb{R})$. This is called the *Lagrange* form. The function $L$ is called the *Lagrange integrand*.

As it turns out, these two forms can be interchanged equivalently. Given a DOP with a Lagrange type objective function, the problem can be transformed into a DOP with a Mayer type objective function by introducing the additional state $x_L$ and adding the equation

$$\dot{x}_L(t) = L(t, z(t))$$

to the DAE system and the initial condition $x_L(t_0) = 0$. Integration then yields

$$x_L(t) = x_L(t_0) + \int_{t_0}^{t} L(t, z(t))\,\mathrm{d}t = \int_{t_0}^{t} L(t, z(t))\,\mathrm{d}t.$$

By setting $f(z) = \phi(t_f, z(t_f)) = x_L(t_f)$, the conversion is complete. This conversion often increases the difficulty of solving the DOP, as it introduces a possibly non-linear constraint. The conversion from a Mayer type objective function to a Lagrange objective function is also possible, as discussed in [Udr10], but is not something that we will need.

Even though the two objective forms are equivalent, they have quite different numerical properties, and it is thus interesting to consider both cases. In fact, for some parts of this thesis, we will consider a combination of the two objective function types, i.e. we let

$$f(z,p) = \phi(t_f, z(t_f), p) + \int_{t_0}^{t_f} L(t, z(t), p)\,\mathrm{d}t. \tag{2.15}$$

This objective function is said to be in *Bolza* form.

In some cases, the time interval $[t_0, t_f]$ is not fixed. In this case the endpoints $t_0$ and/or $t_f$ become free optimization variables. The typical example of this is *minimum time problems*, in which you wish to perform some action as quickly as possible while fulfilling some constraints. This is usually combined with point constraints at $t_f$, which are called *terminal constraints*, and the objective function is typically $f = t_f$.

### 2.2.4 Parameter estimation

The basic idea of parameter estimation is that some model of a system needs to be calibrated against the true system by finding the correct values of certain unknown parameters. To do this some measurement data is needed, which is then used to find the parameter values that cause the model to align with the measured data. The measured data is typically given at discrete time points. Let $y \in \mathcal{F}_b(\mathbb{R}, \mathbb{R}^{n_y})$ be the measured variables, which can be either algebraic variables, states or both, where $n_y$ is the number of measured variables. Also let

$$y_m\big(\hat{t}_i\big)\,, \quad i \in [1..n_y],$$

be the values of the measured variables and let $\hat{t}_i$ be the time points where the variables have been measured, which are called *measurement points* and must be shared by all the measured variables.

The objective is then to minimize the deviation of $y\big(\hat{t}_i\big)$ from $y_m\big(\hat{t}_i\big)$ at all the measurement points. We define the deviation using weighted least squares, giving us the cost function

$$f(z,p) = \sum_{i=1}^{n_m} \big(y\big(\hat{t}_i\big) - y_m\big(\hat{t}_i\big)\big) \cdot Q \cdot \big(y\big(\hat{t}_i\big) - y_m\big(\hat{t}_i\big)\big)\,, \tag{2.16}$$

where $Q \in \mathbb{R}^{n_y \times n_y}$ is called the weighting matrix. This is called *discrete parameter estimation*.

Another possible situation is that the measured data $y_m$ is available as a continuous function of time. Such is the case if the data is measured continuously or if the discrete measurements are somehow interpolated. Our cost function is then instead given by

$$f(z,p) = \int_{t_0}^{t_f} (y(t) - y_m(t)) \cdot Q \cdot (y(t) - y_m(t))\,\mathrm{d}t, \tag{2.17}$$

which is called *continuous parameter estimation*. When we implement our algorithms in Chapter 5, we provide the possibility of linearly interpolating discrete measurement data. Linear interpolation is quite rough and there are certainly better methods available. One advantage of interpolating the discrete data, rather than employing discrete parameter estimation, is that the measured variables no longer need to share measurement points. We will in Chapter 3 see additional advantages of continuous parameter estimation. In Section 6.4 we compare the results obtained from discrete and continuous parameter estimation.

# Chapter 3

# Collocation theory

In Chapter 2, we introduced the dynamic optimization problem. The aim of this thesis is to solve this problem, which is done by casting the infinite-dimensional DOP to a finite-dimensional NLP problem using direct collocation methods, which then can be solved numerically. This chapter is dedicated to the theory behind collocation methods, in particular collocation methods with no collocation point at the start of each element. We start by defining Lagrange interpolation polynomials, which we then use to present the fundamental theory behind all of our collocation methods. We finish by describing the specific Radau and Gauss collocation methods.

## 3.1 Lagrange interpolation polynomials

The collocation methods we use are based on *Lagrange interpolation polynomials*. A Lagrange interpolation polynomial has the general form

$$\zeta(t) = \sum_{k=1}^{n_c} \zeta_k \cdot \ell_k(t), \tag{3.1}$$

where $\ell_k$ is the $k$:th *Lagrange basis polynomial*, $\zeta_k \in \mathbb{R}$ is the basis coefficient of basis polynomial $k$ and $n_c$ is the number of *interpolation points*. The degree of $\zeta$ is $n_c - 1$. The basis polynomials are given by

$$\ell_k(t) = \prod_{l \in [1..n_c] \setminus \{k\}} \frac{t - t_l}{t_k - t_l}, \tag{3.2}$$

where $t_k$ is interpolation point $k$. We define the product over an empty set to be equal to 1 in order to handle the case $n_c = 1$. Lagrange interpolation polynomials have the important property

$$\zeta(t_k) = \zeta_k, \quad \forall k \in [1..n_c]. \tag{3.3}$$

This together with their simplicity is the main reason we choose to work with them.

We will also be needing the derivative of a Lagrange interpolation polynomial, which is given by

$$\zeta'(t) = \sum_{k=1}^{n_c} \zeta_k \cdot \ell'_k(t), \tag{3.4}$$

where the derivative of $\ell_k$ is obtained via the product rule as

$$\ell'_k(t) = \sum_{m \in [1..n_c] \setminus \{k\}} \left( \frac{1}{t_k - t_m} \cdot \prod_{l \in [1..n_c] \setminus \{k,m\}} \frac{t - t_l}{t_k - t_l} \right). \tag{3.5}$$

Note that all polynomials actually are Lagrange polynomials; (3.1) is just a specific representation of an arbitrary polynomial. It is thus more semantically correct to speak of the Lagrange *form* of a polynomial. Preserving this semantic correctness can become quite cumbersome, so sometimes we instead simply refer to them as Lagrange polynomials.

## 3.2 Generic collocation theory

### 3.2.1 Collocation polynomial construction

We start the DOP transcription by discretizing the time and variable trajectories into $n_e$ *elements*. We approximate the variable trajectories

$$v(t) := (x(t), u(t), w(t)), \quad \forall t \in [t_0, t_f]$$

in element $i \in [1..n_e]$ by a vector-valued Lagrange interpolation polynomial

$$v_i = (x_i, u_i, w_i) \in C^\infty(\mathbb{R}, \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_w}).$$

These are called *collocation polynomials*. Next, we normalize the time in element $i$ by

$$t(\tau) = t_{i-1} + h_i \cdot \tau, \quad \tau \in [0, 1] \tag{3.6}$$

where $t_i$ is the time at the end of element $i$, which is called the *mesh point* of element $i$, and $h_i$ is the length of element $i$.

The approximation of the variable trajectories $v(t)$ over the entire time interval $[t_0, t_f]$ is then formed by "gluing" the collocation polynomials together at the mesh points. Different structures of the collocation polynomials are needed depending on which variable is considered, in particular on the desired continuity of the considered variable. We introduce new variable compositions to keep track of the variables for which we wish to enforce continuity. Let

$$\begin{aligned} (u^C, u^D) &:= u \quad \text{and} \quad (u_i^C, u_i^D) := u_i, \quad \forall i \in [1..n_e], \\ (w^C, w^D) &:= w \quad \text{and} \quad (w_i^C, w_i^D) := w_i, \quad \forall i \in [1..n_e], \end{aligned}$$

where $u^C$ is the vector of control variables for which we wish to enforce continuity, $u^D$ is the vector of control variables which may be discontinuous, $u_i^C$ and $u_i^D$ are the polynomial approximations of $u^C$ and $u^D$ respectively in element $i$, and $w^C$, $w^D$, $w_i^C$ as well as $w_i^D$ are defined analogously for the algebraic variables. Let $n_u^C$ and $n_u^D$ denote the number of scalar components in $u^C(t)$ and $u^D(t)$ respectively and $n_w^C$ as well as $n_w^D$ be defined analogously for the algebraic variables. Since we want all the states to be continuous, also let

$$\begin{aligned} v^C &:= (x, u^C, w^C) \quad \text{and} \quad v_i^C := (x_i, u_i^C, w_i^C), \quad \forall i \in [1..n_e], \\ v^D &:= (u^D, w^D) \quad \text{and} \quad v_i^D := (u_i^D, w_i^D), \quad \forall i \in [1..n_e]. \end{aligned}$$

Note that since the controls are free, we can freely choose whether to enforce continuity for these. The algebraic variables on the other hand are implicitly defined as functions of $x$ and $u$ by the relation (2.13b), so their continuity depends on the continuity of their dependent variables and can thus not be enforced as arbitrarily as the control variables. Enforcing continuity for algebraic variables which depend on discontinuous controls (the states are always continuous) may result in an infeasible problem (specifically, the transcription will not be degree-preserving, which we define later). So this should only be done when it is known a priori that the algebraic variables in question actually are continuous. And it is not always necessary to enforce continuity for the continuous algebraic variables, since a discontinuous approximation will often suffice.

The approximation of $v^C$ is continuous if and only if

$$v_{i+1}^C(0) = v_i^C(1), \quad \forall i \in [1..n_e - 1]. \tag{3.7}$$

Since we have a condition on the value of the collocation polynomials $v_i^C$ at the start of each element, we place an interpolation there. Since we have no such condition for the collocation polynomials $v_i^D$, we do not place an interpolation there. The collocation polynomials in element $i$ are thus given by

$$\begin{aligned} v_i^C(\tau) &= \sum_{k=0}^{n_c} v_{i,k}^C \cdot \tilde{\ell}_k(\tau), \\ v_i^D(\tau) &= \sum_{k=1}^{n_c} v_{i,k}^D \cdot \ell_k(\tau), \end{aligned}$$

where

$$v_{i,k}^C = \left(x_{i,k}, u_{i,k}^C, w_{i,k}^C\right) \in \mathbb{R}^{n_x} \times \mathbb{R}^{n_u^C} \times \mathbb{R}^{n_w^C}$$

and

$$v_{i,k}^D = \left(u_{i,k}^D, w_{i,k}^D\right) \in \mathbb{R}^{n_u^D} \times \mathbb{R}^{n_w^D}$$

are the basis coefficients,

$$\begin{aligned} \tilde{\ell}_k(\tau) &= \prod_{l \in [0..n_c] \setminus \{k\}} \frac{\tau - \tau_l}{\tau_k - \tau_l}, \\ \ell_k(\tau) &= \prod_{l \in [1..n_c] \setminus \{k\}} \frac{\tau - \tau_l}{\tau_k - \tau_l} \end{aligned}$$

are the Lagrange basis polynomials, where $\tau_k$ is the $k$:th interpolation point normalized according to (3.6). The purpose of normalizing the time is that this lets us use the same interpolation points and basis polynomials in all the elements for all the variables, as long as we separate $v_i^C$ and $v_i^D$. Since we needed an interpolation point at the start of each element for $v_i^C$ in order to achieve continuity, we define

$$\tau_0 := 0.$$

How to actually achieve continuity using this interpolation point depends on which specific collocation method is used and is thus discussed in the sections for the specific methods.

The choice of the remaining interpolation points is what defines a specific collocation method. In this thesis we base this choice on the roots of the shifted Gauss-Jacobi polynomial of degree $K = n_c - \alpha - \beta$, which can be described by

$$P_K^{(\alpha,\beta)}(\tau) = \sum_{j=0}^{K} (-1)^{K-j} \cdot \gamma_j \cdot \tau^j, \tag{3.8}$$

where

$$\begin{cases} \gamma_0 = 1, \\ \gamma_j = \frac{(K+1-j)(K+j+\alpha+\beta)}{j(j+\beta)}, \quad j \in [1..K], \end{cases}$$

where $\alpha$ and $\beta$ are constants of our choosing. The polynomial is shifted in the sense that normally the polynomial is defined with $\tau \in [-1, 1]$, whereas we have chosen the domain $[0, 1]$. In this thesis we consider the following three combinations of $\alpha$ and $\beta$.

$$\begin{aligned} &\alpha = 0 \wedge \beta = 0 \rightarrow \text{ Gauss collocation,} \\ &\alpha = 1 \wedge \beta = 0 \rightarrow \text{ Radau collocation,} \\ &\alpha = 1 \wedge \beta = 1 \rightarrow \text{ Lobatto collocation.} \end{aligned}$$

We discuss Gauss and Radau collocation further in Sections 3.4 and 3.3 respectively. The reason behind these choices of $\tau_k$ is that they yield a high approximation accuracy, which we briefly discuss in Section 3.2.2, and have certain stability properties.

For the approximation of $\dot{x}$ in element $i$ we use the derivative of $x_i$, i.e.

$$\dot{x}_i := \frac{\mathrm{d}x_i}{\mathrm{d}t}$$

With the normalized time, the collocation polynomials representing the state derivatives are given by the chain rule, (3.6) and (3.4) as

$$\dot{x}_i(\tau) = \frac{\mathrm{d}x_i}{\mathrm{d}t}(\tau) = \frac{\mathrm{d}\tau}{\mathrm{d}t} \cdot \frac{\mathrm{d}x_i}{\mathrm{d}\tau}(\tau) = \frac{1}{h_i} \cdot \sum_{k=0}^{n_c} x_{i,k} \cdot \dot{\ell}_k(\tau), \tag{3.9}$$

where $\dot{\ell}_k(\tau)$ is obtained via (3.5).

Another benefit of enforcing continuity for the control variables and algebraic variables is that their derivatives can be defined in the same way as $\dot{x}_i$, i.e.

$$\dot{v}_i^C(\tau) := \frac{\mathrm{d}v_i^C}{\mathrm{d}t}(\tau) = \frac{\mathrm{d}\tau}{\mathrm{d}t} \cdot \frac{\mathrm{d}v_i^C}{\mathrm{d}\tau}(\tau) = \frac{1}{h_i} \cdot \sum_{k=0}^{n_c} v_{i,k}^C \cdot \dot{\ell}_k(\tau). \tag{3.10}$$

This can then be used to define constraints on the derivatives of the continuous control variables and algebraic variables, or include them in the cost function. We did however not consider this possibility in the formulation of the general DOP in Chapter 2, but the possibility of defining $\dot{v}_i^C$ like this is relevant to other ideas discussed later in this thesis.

We now compose our collocation polynomials $\dot{x}_i$, $v_i^C$ and $v_i^D$ into $z_i$ in a manner similar to what we did in (2.11), i.e.

$$z_i := \left(\dot{x}_i, v_i^C, v_i^D\right) = \left(\dot{x}_i, x_i, u_i^C, w_i^C, u_i^D, w_i^D\right). \tag{3.11}$$

Let $\tilde{z}$ denote the approximation of $z$, which is constructed by "gluing" the collocation polynomials $z_i$ together as discussed at the beginning of this section. Let

$$t_{i,k} := t_{i-1} + h_i \cdot \tau_k.$$

When we transcribe the DOP (2.13), we will only be enforcing the constraints at discrete time points. These points are given by the set

$$\left\{t_{i,k} : i \in [1..n_e] \wedge k \in [1..n_c]\right\},$$

and are called the *collocation points*. Note that these coincide with the interpolation points of the collocation polynomials, with the exception of $\tau_0$ in each element. We thus redefine $n_c$ to be the number of collocation points in the collocation polynomials rather than the number of interpolation points. We choose these interpolation points as collocation points because we can easily get all the values of $\tilde{z}$ at these points via (3.3), except those of $\dot{x}_i$. Since $\dot{x}_i$ is obtained by differentiating the Lagrange form of the collocation polynomial $x_i$, the Lagrange form of $\dot{x}_i$ is not easily obtained. We could compute the Lagrange form of $\dot{x}_i$, in order to enable the use of (3.3), but we would have little use of it. All we need is to be able to evaluate $\dot{x}_i$ at arbitrary time points, which we can do via (3.9). So we get the values of $\dot{x}_i$ at the collocation points as

$$\dot{x}_{i,k} := \dot{x}_i(\tau_k) = \frac{1}{h_i} \cdot \sum_{l=0}^{n_c} x_{i,l} \cdot \dot{\ell}_l(\tau_k). \tag{3.12}$$

Note that $\dot{x}_{i,k}$, unlike $v_{i,k}$, are not Lagrange basis coefficients, but rather the value of the polynomial $\dot{x}_i$ at $\tau_k$ (not unlike $v_{i,k}$). The equations (3.12) are called *collocation equations*.

Lagrange interpolation requires all the interpolation points to be distinct. Thus we can not have a collocation point at the start of each element, since we have already chosen $\tau_0 = 0$ as an interpolation point for $v_i^C$. Lobatto collocation, unlike Radau

and Gauss collocation, has a collocation point at the start of each element. For this reason, the approach described in this section can not be used to create a Lobatto collocation method. A different approach is required for Lobatto collocation, which is discussed in Appendix A. Thus we do not discuss Lobatto collocation to the same extent as we discuss Radau and Gauss collocation in this chapter.

Our collocation polynomials are now completed. To summarize, the variables $z$ are approximated in element $i$ by the collocation polynomials $z_i$, given by

$$z_i = \left(\dot{x}_i, x_i, u_i^C, w_i^C, u_i^D, w_i^D\right),$$

where

$$\begin{aligned}
\dot{x}_i(\tau) &= \frac{1}{h_i} \cdot \sum_{k=0}^{n_c} x_{i,k} \cdot \dot{\tilde{\ell}}_k(\tau), & x_i(\tau) &= \sum_{k=0}^{n_c} x_{i,k} \cdot \tilde{\ell}_k(\tau), \\
u_i^C(\tau) &= \sum_{k=0}^{n_c} u_{i,k}^C \cdot \tilde{\ell}_k(\tau), & u_i^D(\tau) &= \sum_{k=1}^{n_c} u_{i,k}^D \cdot \ell_k(\tau), \\
w_i^C(\tau) &= \sum_{k=0}^{n_c} w_{i,k}^C \cdot \tilde{\ell}_k(\tau), & w_i^D(\tau) &= \sum_{k=1}^{n_c} w_{i,k}^D \cdot \ell_k(\tau).
\end{aligned} \tag{3.13}$$

Note that we have not created a polynomial representation of $p$. This is not needed, since the parameters are already discrete. We are now ready to put these to use by transcribing the DOP (2.13) into an NLP problem. But before we proceed to do this, we discuss how the choice of collocation points affects the orders of convergence for our methods.

### 3.2.2 Convergence orders

This section assumes that $u = u^D$ and $w = w^D$.

As shown in [AP91] and [KB08], the uniform global error of the approximated state in element $i$, given by $x_i$, is

$$O\left(h_i^{\min(n_c+1, 2\cdot n_c-\alpha-\beta)}\right).$$

The uniform global error of the approximated algebraic variables and control variables in element $i$ is

$$O\left(h_i^{n_c}\right).$$

If $n_c > 1$, the uniform global error is thus the same for Gauss, Radau and Lobatto collocation.

At the mesh points, the global error of all variables is

$$O\left(h_i^{2\cdot n_c-\alpha-\beta}\right).$$

Thus the convergence order of the solution is up to twice as high on the mesh points than it is on the rest of the element (including the collocation points). This phenomenon is called *superconvergence*. So at the mesh points, Gauss collocation provides the highest accuracy, and Radau collocation provides higher accuracy than Lobatto collocation.

Note that this section is highly reliant on the DAE system being of index 0 or 1. For DAE systems of higher index, the methods may suffer from convergence order reduction, or even become unstable.

### 3.2.3 Transcription of dynamic optimization problems

For ease of notation, we introduce a variable composition similar to the one in (3.11). Let

$$z_{i,k} := z_i(\tau_k) = (\dot{x}_{i,k}, x_{i,k}, u_{i,k}, w_{i,k}), \quad \forall (i,k) \in [1..n_e] \times [0..n_c],$$

where

$$\begin{aligned} u_{i,0} &:= u^C_{i,0}, \quad u_{i,k} := \left(u^C_{i,k}, u^D_{i,k}\right), \quad \forall k \in [1..n_c], \\ w_{i,0} &:= w^C_{i,0}, \quad w_{i,k} := \left(w^C_{i,k}, w^D_{i,k}\right), \quad \forall k \in [1..n_c]. \end{aligned}$$

Let also

$$z^C_{i,k} := v^C_i(\tau_k) = (x_{i,k}, u^C_{i,k}, w^C_{i,k}), \quad \forall (i,k) \in [1..n_e] \times [0..n_c].$$

As optimization variables in the nonlinear programming problem we choose all the values of $\dot{x}_{i,k}$ and the Lagrange basis coefficients, i.e. $z_{i,k}$ for all $k$ greater than 0, as well as the parameters $p$. We also choose the initial values as NLP variables, which we denote by $z_{1,0}$ (which are not Lagrange basis coefficients). We further choose the values of $z^C_{i,0}$ as NLP variables and refer to them as *continuity variables*. We thus let

$$\begin{aligned} Z = (&z_{1,0}, z_{1,1}, z_{1,2}, \ldots, z_{1,n_c}, \\ &z^C_{2,0}, z_{2,1}, z_{2,2}, \ldots, z_{2,n_c}, \\ &z^C_{3,0}, z_{3,1}, z_{3,2}, \ldots, z_{3,n_c}, \\ &\vdots, \\ &z^C_{n_e,0}, z_{n_e,1}, z_{n_e,2}, \ldots, z_{n_e,n_c}, p). \end{aligned}$$

be the vector containing all the NLP variables (slight modifications will be made to this vector for Gauss collocation). The dimension of $Z$, i.e. the number of scalar optimization variables, is given by

$$\begin{aligned} n_Z &= n_z + (n_e - 1) \cdot n^C_z + n_e \cdot n_c \cdot n_z + n_p \\ &= (1 + n_e \cdot n_c) \cdot n_z + (n_e - 1) \cdot n^C_z + n_p. \end{aligned} \tag{3.14}$$

The DOP transcription then gives us the following NLP problem.

$$\min_{Z\in\mathbb{R}^{n_Z}} \tilde{f}(Z) \tag{3.15a}$$

$$\text{s.t.} \begin{cases} F(t_{i,k}, z_{i,k}) = 0, & (3.15b)\\ F_0(z_{1,0}) = 0, & (3.15c)\\ u^D_{1,0} - \sum_{k=1}^{n_c} u^D_{1,k} \cdot \ell_k(0) = 0, & (3.15d)\\ z_L \le z_{i,k} \le z_U, & (3.15e)\\ p_L \le p \le p_U, & (3.15f)\\ g_e(t_{i,k}, z_{i,k}) = 0, & (3.15g)\\ g_i(t_{i,k}, z_{i,k}) \le 0, & (3.15h)\\ G_e(Z_e) = 0, & (3.15i)\\ G_i(Z_i) \le 0, & (3.15j)\\ \dot{x}_{j,l} - \frac{1}{h_j} \cdot \sum_{m=0}^{n_c} x_{j,m} \cdot \dot{\tilde{\ell}}_m(\tau_l) = 0, \quad \forall (j,l) \in [1..n_e] \times [1..n_c] & (3.15k)\\ G(Z) = 0, & (3.15l)\\ \forall (i,k) \in \{(1,0)\} \cup ([1..n_e] \times [1..n_c]). \end{cases}$$

Equation (3.15l) is equality constraints which depend on the choice of collocation points and will thus be defined individually in each section corresponding to each such choice. We will now study the remaining parts of (3.15), which essentially are common to all collocation methods with no collocation point at the start of each element.

Like before, we do not specify the structure of the objective function $\tilde{f}$ in the general formulation, as it depends on what kind of DOP is considered. For optimal control and parameter optimization, we need to transcribe the general Bolza type cost function, see (2.15). How we handle the Mayer part depends on the choice of collocation points and is discussed later. For the Lagrange part, we need to be able to integrate functions of $\tilde{z}$. We accomplish this using *Gaussian quadrature*. We start by using (3.6) to define the Lagrange integrand in each element, given by

$$\tilde{L}_i(\tau, z_i(\tau), p) := L(t(\tau), z_i(\tau), p) = L(t(\tau), \tilde{z}(t(\tau)), p).$$

We then have

$$\begin{aligned} f(\tilde{z}, p) = \int_{t_0}^{t_f} L(t, \tilde{z}(t), p)\,\mathrm{d}t &= \sum_{i=1}^{n_e} \left( h_i \cdot \int_0^1 \tilde{L}_i(\tau, z_i(\tau), p)\,\mathrm{d}\tau \right) \\ &\approx \sum_{i=1}^{n_e} \left( h_i \cdot \sum_{k=1}^{n_c} \omega_k \cdot \tilde{L}_i\left(\tau_k, z_i(\tau_k), p\right) \right) \\ &= \sum_{i=1}^{n_e} \left( h_i \cdot \sum_{k=1}^{n_c} \omega_k \cdot \tilde{L}_i\left(\tau_k, z_{i,k}, p\right) \right) \\ &:= \tilde{f}(Z), \end{aligned} \tag{3.16}$$

where the *quadrature weights* $\omega_k$ are given by

$$\omega_k = \int_0^1 \ell_k(\tau)\,\mathrm{d}\tau.$$

This choice of quadrature weights provides the best integral approximation for the given interpolation points, as shown in [Bie10, ch. 10].

Note that we in the approximation of the integral in (3.16) only used the values of $z_i^C$ at the collocation points, even though the values $z_{i,0}$ are readily available as NLP variables. This might seem wasteful. The reason for this is that the polynomials $z_i^C$ and $z_i^D$ do not share all their respective interpolation points, but since they are both a part of $\tilde{L}_i$, they have to share quadrature weights. There are three possible ways to handle this.

1. Use the quadrature weights belonging to $z_i^D$ and ignore the values of $z_{i,0}^C$. This essentially amounts to approximating the polynomials $z_i^C$ of degree $n_c$ with polynomials of degree $n_c - 1$, using the values of $z_i^C(\tau_k)$ for all $k \in [1..n_c]$.
2. Use the quadrature weights belonging to $z_i^C$ and use (3.13) to evaluate $z_i^D(0)$.
3. If $L$ has a structure that allows it to be decomposed as

   $$L(t, z(t)) = L_1(t, \dot{x}(t), x(t), u^C(t), w^C(t), p) + L_2(t, u^D(t), w^D(t), p),$$

   we can weight the polynomials separately depending on if they have an interpolation at $\tau_0$ or not. If only a part of $L$ can be decomposed like this, then we can separate the corresponding weights and employ option 1 or 2 for part of $L$ that can not be decomposed like this.

The most accurate option is number 3. There is however no easy way to implement this in JModelica.org and the gain in accuracy is believed to be minor, so this option is discarded. Which of the two remaining choices that is the most accurate is beyond the scope of this thesis to investigate. Since option 1 is simpler to implement and also requires less computations, this is the approach chosen for the quadrature in (3.16).

If the considered DOP instead is a continuous parameter estimation problem, see (2.17), we use the same Gaussian quadrature, yielding

$$\begin{aligned}
f(\tilde{z}, p) &= \int_{t_0}^{t_f} (\tilde{y}(t) - y_m(t)) \cdot Q \cdot (\tilde{y}(t) - y_m(t))\,\mathrm{d}t \\
&= \sum_{i=1}^{n_e} \left( h_i \cdot \int_0^1 (y_i(\tau) - y_m(t(\tau))) \cdot Q \cdot (y_i(\tau) - y_m(t(\tau)))\,\mathrm{d}t \right) \\
&\approx \sum_{i=1}^{n_e} \left( h_i \cdot \sum_{k=1}^{n_c} \omega_k \cdot (y_i(\tau_k) - y_m(t(\tau_k))) \cdot Q \cdot (y_i(\tau_k) - y_m(t(\tau_k))) \right) \\
&= \sum_{i=1}^{n_e} \left( h_i \cdot \sum_{k=1}^{n_c} \omega_k \cdot (y_{i,k} - y_m(t(\tau_k))) \cdot Q \cdot (y_{i,k} - y_m(t(\tau_k))) \right) \\
&:= \tilde{f}(Z),
\end{aligned} \tag{3.17}$$

where $\tilde{y}$, $y_i$ and $y_{i,k}$ are the components of $\tilde{z}$, $z_i$ and $z_{i,k}$ respectively corresponding to the measured variables.

If we consider the discrete parameter estimation problem, there are two possible situations. The first is that all the measurement points $\hat{t}_i$ coincide with a collocation point. All the required values to evaluate the cost function are then already available as NLP variables. If some measurement points do *not* coincide with a collocation point, there are two possible approaches. The first is to evaluate the collocation polynomials at the measurement points using (3.13). This is however very expensive and is not a good approach for a large number of measurement points. The alternative is to modify the mesh so that we actually have a collocation point at each measurement point. In this thesis we leave this modification up to the user, and thus assume the first situation, i.e. that all the measurement points $\hat{t}_i$ coincide with a collocation point. The cost function is then straightforward to transcribe as

$$f(\tilde{z}, p) = \sum_{i=1}^{n_m} \left(\tilde{y}(\hat{t}_i) - y_m(\hat{t}_i)\right) \cdot Q \cdot \left(\tilde{y}(\hat{t}_i) - y_m(\hat{t}_i)\right) := \tilde{f}(Z),$$

where $\tilde{y}(\hat{t}_i)$ is the vector of NLP variables that has been determined to correspond to the value of the measured variable $y$ at the measurement point $\hat{t}_i$.

We have now discussed the transcription of all DOP cost functions considered in this thesis and now turn our attention to the constraints. The constraints (3.15b) and (3.15c) are direct transcriptions of (2.13b) and (2.13c) respectively, where we enforce the system dynamics at the collocation points. To get consistent initial values $z_{1,0}$ we also enforce $F$ and $F_0$ at $t_{1,0} = t_0$. To this end we need the value of $z_{1,0}$. The constraints (3.15b) and (3.15c) give us all the parts of $z_{1,0}$ corresponding to the non-free variables, i.e. $\dot{x}_{1,0}$, $x_{1,0}$ and $w_{1,0}$. The value of $u_{1,0}^C$ is a basis coefficient for $u_1^C$ and can thus be chosen freely, since $u$ is free. All that is missing to determine $z_{1,0}$ is the value of $u_{1,0}^D$. Since this is not a basis coefficient, it can not be chosen freely, but instead needs to be governed by a constraint. This constraint uses (3.13) to evaluate $u_1^D$ at $t_0$ and then enforces this value as an equality constraint, which gives us (3.15d).

There are two ways of transcribing the variable bounds and the path constraints, i.e. Equations (2.13d) to (2.13g). We either enforce them at all the collocation points, or at all the mesh points. Using just the mesh points is sensible due to the superconvergence properties of our methods, as discussed in Section 3.2.2. This may however be problematic. Since the DAE system is not necessarily defined for variable values that do not satisfy the variable bounds and path constraints, we may be unable to evaluate the DAE system at the collocation points if the variable bounds and path constraints are not satisfied at these points. In order to avoid this problem, we choose to enforce the variable bounds and path constraints at all the collocation points. We also enforce them at the start time $t_{1,0}$, which is done as a precaution rather than a necessity, since if the user has supplied consistent initial conditions, they should also satisfy the bounds and path constraints.

Point constraints are however less straightforward to transcribe. For these we run into the same problems as we did in the case of discrete parameter estimation. We also use the same solution, i.e. we assume that each constraint point coincides with a collocation point. The values of $Z_e$ and $Z_i$ are thus available as already existing NLP variables, which gives us (3.15i) and (3.15j).

In Section 2.1 we noted that the constraint functions must be twice continuously differentiable with respect to all the primal variables (i.e. $Z$). Since the time is not a primal variable, we do not need to make any restrictions on the differentiability of $g_e$ and $g_i$ with respect to their first arguments, as mentioned in Section 2.2.2.

The last part of the NLP problem is the collocation constraints (3.15k), which are the collocation equations (3.12) enforced at the collocation points. Note that, unlike all the other constraints, we do not enforce the collocation equations at $t_{1,0}$, which is why we do not consider $t_{1,0}$ to be a collocation point. This has the consequence that $\dot{x}_{1,0}$ is not determined by the value of $\frac{\mathrm{d}x_1}{\mathrm{d}t}(0)$, but rather the value that is consistent with the DAE system and its initial conditions. These two values do not necessarily coincide, which is why we can not enforce both of them. This is not necessarily a problem, but if it is, there are two possible solutions. The first is to make the first element sufficiently small, as the continuity of $F$ gives that

$$\frac{\mathrm{d}x_1}{\mathrm{d}t}(0) \to \dot{x}_{1,0}, \quad h_1 \to 0.$$

The other solution is to use a collocation method which has a collocation point at $\tau_0$, such as the Lobatto method.

Finally we note that the general collocation method constructed in this section is an optimization method. Using the ideas of Section 3.2.1, one can also construct simulation methods. These simulation methods are also called collocation methods, and turn out to be a special case of implicit *Runge-Kutta* methods. It is shown in Appendix A how a general simulation collocation method can be equivalently formulated as an implicit Runge-Kutta method. The converse is however not true, e.g. Runge-Kutta methods with non-distinct $c_i$ in their Butcher tableaus can not be formulated as collocation methods. With the exception of Appendix A, we do not focus on the Runge-Kutta formulations of our methods in this thesis. But it is still worth keeping this correspondence in mind, since the specific optimization collocation methods we derive during the rest of this chapter inherit some properties from their Runge-Kutta simulation counterparts. There is also plenty of research on Runge-Kutta methods which is directly applicable to our collocation methods, e.g. some of the points discussed in Section 3.2.2.

In the succeeding sections we discuss the practical differences between Radau and Gauss, e.g. how the transcriptions differ and are implemented. We do not focus on the more theoretical aspects of the different methods, such as stability properties in certain situations. See [HW96, ch. IV] and [Sim10, ch. 3] for discussions on these topics.

## 3.3 Radau collocation

The collocation points $\tau_k$ for the Radau collocation method are obtained as the roots of the shifted Gauss-Jacobi polynomial (see (3.8)) with $\alpha = 1$ and $\beta = 0$. This gives us $n_c - 1$ interpolation points in the open interval $(0, 1)$. The last is chosen as $\tau_{n_c} = 1$. Below is a table with numeric values of the collocation points for all $n_c \in \{1, 2, 3, 4\}$ and a table with numeric values of the corresponding quadrature weights.

| | $\tau_1$ | $\tau_2$ | $\tau_3$ | $\tau_4$ |
|---|---|---|---|---|
| $n_c = 1$ | 1.000000 | N/A | N/A | N/A |
| $n_c = 2$ | 0.333333 | 1.000000 | N/A | N/A |
| $n_c = 3$ | 0.155051 | 0.644949 | 1.000000 | N/A |
| $n_c = 4$ | 0.088588 | 0.409467 | 0.787659 | 1.000000 |

Table 3.1: Radau collocation points

| | $\omega_1$ | $\omega_2$ | $\omega_3$ | $\omega_4$ |
|---|---|---|---|---|
| $n_c = 1$ | 1.000000 | N/A | N/A | N/A |
| $n_c = 2$ | 0.750000 | 0.250000 | N/A | N/A |
| $n_c = 3$ | 0.376403 | 0.512485 | 0.111111 | N/A |
| $n_c = 4$ | 0.220462 | 0.388193 | 0.328844 | 0.062500 |

Table 3.2: Radau quadrature weights

Let us continue by addressing the continuity conditions (3.7). Thanks to the collocation point $\tau_{n_c} = 1$ and the interpolation point $\tau_0 = 0$, the values of $v_i^C(0)$ and $v_i^C(1)$ are easily obtained by (3.3), giving us

$$v_i^C(1) = v_i^C(\tau_{n_c}) = z_{i,n_c}^C,$$
$$v_i^C(0) = v_i^C(\tau_0) = z_{i,0}^C.$$

Since these two values have been chosen as NLP variables, continuity is enforced with the constraints

$$z_{i,n_c}^C - z_{i+1,0}^C = 0, \quad \forall i \in [1..n_e - 1]. \tag{3.18}$$

The handling of Lagrange type cost functions as well as continuous and discrete parameter estimation is fully described in 3.2.3. The only type of cost function we have not yet discussed the transcription of is the Mayer type, see (2.14). This is straightforward to transcribe for Radau collocation, since we have a collocation point at $t_f = t_{n_e,n_c}$. The Mayer type cost function is thus simply transcribed as

$$f(\tilde{z}, p) = \phi(t_f, \tilde{z}(t_f), p) = \phi(t_{n_e,n_c}, z_{n_e,n_c}, p) := \tilde{f}(Z). \tag{3.19}$$

We now have all the information we need to fully transcribe the DOP into an NLP problem using Radau collocation. The full transcription is given by (3.15). All that remains is specifying the function $G$ in (3.15l). For Radau collocation, we let $G = G_R$, where $G_R$ simply is the continuity constraints (3.18). We thus have

$$G_R(Z) := \left(z^C_{1,n_c} - z^C_{2,0}, z^C_{2,n_c} - z^C_{3,0}, \ldots, z^C_{n_e-1,n_c} - z^C_{n_e,0}\right).$$

At the end of Section 3.2.3 we mentioned that our collocation methods are actually implicit Runge-Kutta methods. As it turns out, Radau collocation with $n_c = 1$ corresponds to the arguably most famous implicit numerical method. To see this, we study the collocation equations at the mesh points, i.e.

$$\dot{x}_i(1) = \frac{1}{h_i} \cdot \sum_{k=0}^{n_c} x_{i,k} \cdot \dot{\tilde{\ell}}_k(1).$$

Applying (3.5) gives us

$$\begin{aligned}
\dot{x}_i(1) &= \frac{1}{h_i} \cdot \sum_{k=0}^{1} \left( x_{i,k} \cdot \sum_{m \in \{0,1\}\setminus\{k\}} \left( \frac{1}{\tau_k - \tau_m} \cdot \prod_{l \in \{0,1\}\setminus\{k,m\}} \frac{1-\tau_l}{\tau_k - \tau_l} \right) \right) \\
&= \frac{1}{h_i} \cdot \left( x_{i,0} \cdot \sum_{m=1} \left( \frac{1}{\tau_0 - \tau_m} \cdot \prod_{l \in \{1\}\setminus\{m\}} \frac{1-\tau_l}{\tau_0 - \tau_l} \right) \right. \\
&\qquad \left. + \, x_{i,1} \cdot \sum_{m=0} \left( \frac{1}{\tau_1 - \tau_m} \cdot \prod_{l \in \{0\}\setminus\{m\}} \frac{1-\tau_l}{\tau_1 - \tau_l} \right) \right) \\
&= \frac{1}{h_i} \cdot \left( x_{i,0} \cdot \frac{1}{\tau_0 - \tau_1} + x_{i,1} \cdot \frac{1}{\tau_1 - \tau_0} \right) = \frac{1}{h_i} \cdot \left( \frac{x_{i,0}}{0-1} + \frac{x_{i,1}}{1-0} \right) \\
&= \frac{x_i(1) - x_i(0)}{h_i},
\end{aligned}$$

which we recognize as the derivative approximation used to derive the implicit Euler method.

We finish by counting our degrees of freedom to ensure that we have a sensible transcription formulation, in the sense that given fixed (and feasible) values for all $u_{i,k}$ and $p$, the rest of $Z$ can be uniquely determined from the constraints. This corresponds to having just as many scalar constraint equations as we have NLP variables representing the non-free variables $x$, $\dot{x}$ and $w$. If a transcription has this property, it is said to be *degree-preserving*. This property guarantees that the variables $u_{i,k}$ and $p$ can be chosen as freely as in the continuous DOP, independently of the details of the transcription. A transcription which is not degree-preserving may result in an infeasible NLP problem, even if the original DOP is well-posed.

The total number of NLP variables is given by (3.14). Out of the $n_Z$ NLP variables, we have

$$n_f := n^C_u + n_e \cdot n_c \cdot n_u + n_p$$

NLP variables which represent values of the free DOP variables $u$ and $p$, and we should thus have as many degrees of freedom. Note that the variables $u_{1,0}^D$ and $u_{i,0}^C$ for all $i \in [2..n_e]$ do not provide any additional degrees of freedom, as they are regulated by the constraints (3.15d) and (3.15l). How the constraints (3.15e) to (3.15j) affect the degrees of freedom is dependent on the DOP, so we disregard these for the purpose of determining whether our transcription is degree-preserving. In order for the transcription to be degree-preserving, the equations (3.15b) to (3.15d), (3.15k) and (3.15l) should give us a total of

$$\begin{aligned} n_Z - n_f &= (1 + n_e \cdot n_c) \cdot n_z + (n_e - 1) \cdot n_z^C + n_p \\ &\quad - (n_u^C + n_e \cdot n_c \cdot n_u + n_p) \\ &= n_z - n_u^C + (n_e - 1) \cdot n_z^C + n_e \cdot n_c \cdot (2 \cdot n_x + n_w) \\ &= 2 \cdot n_x + n_w + n_u^D + (n_e - 1) \cdot n_z^C + n_e \cdot n_c \cdot (2 \cdot n_x + n_w) \end{aligned} \tag{3.20}$$

(scalar) equations.

Since the codomain of $F$ is $\mathbb{R}^{n_x+n_w}$ and the codomain of $F_0$ is $\mathbb{R}^{n_x}$, (3.15b) and (3.15c) give us

$$(1 + n_e \cdot n_c) \cdot (n_x + n_w) + n_x$$

equations. The constraints (3.15d) for the initial value of $u^D$ we get

$$n_u^D$$

equations. The collocation constraints (3.15k), enforced at all collocation points, give us

$$(1 + n_e \cdot n_c) \cdot n_x$$

equations. Finally we have the method-dependent constraints (3.15l), which in this case gives us

$$(n_e - 1) \cdot n_z^C$$

equations. Summing them all up and subtracting the number of non-free variables, given by (3.20), we get

$$\begin{aligned} &(1 + n_e \cdot n_c) \cdot (n_x + n_w) + n_u^D + (1 + n_e \cdot n_c) \cdot n_x + (n_e - 1) \cdot n_z^C \\ &-\big(2 \cdot n_x + n_w + n_u^D + (n_e - 1) \cdot n_z^C + n_e \cdot n_c \cdot (2 \cdot n_x + n_w)\big) \\ &=(1 + n_e \cdot n_c) \cdot (2 \cdot n_x + n_w) \\ &-(2 \cdot n_x + n_w + n_e \cdot n_c \cdot (2 \cdot n_x + n_w)) = 0. \end{aligned}$$

In other words, we have the same number of non-free variables as equations and our transcription is thus degree-preserving.

## 3.4 Gauss collocation

The collocation points $\tau_k$ for the Gauss collocation method are obtained as the roots of the shifted Gauss-Jacobi polynomial (see (3.8)) with $\alpha = \beta = 0$. This gives us

$n_c$ collocation points in the open interval $(0, 1)$, i.e. we have no collocation points on the element boundaries. Below is a table with numeric values of the collocation points for all $n_c \in \{1, 2, 3, 4\}$ and a table with numeric values of the corresponding quadrature weights. Note how all the collocation points are symmetric around 0.5, which is no coincidence.

| | $\tau_1$ | $\tau_2$ | $\tau_3$ | $\tau_4$ |
|---|---|---|---|---|
| $n_c = 1$ | 0.500000 | N/A | N/A | N/A |
| $n_c = 2$ | 0.211325 | 0.788675 | N/A | N/A |
| $n_c = 3$ | 0.112702 | 0.500000 | 0.887298 | N/A |
| $n_c = 4$ | 0.069432 | 0.330009 | 0.669991 | 0.930568 |

Table 3.3: Gauss collocation points

| | $\omega_1$ | $\omega_2$ | $\omega_3$ | $\omega_4$ |
|---|---|---|---|---|
| $n_c = 1$ | 1.000000 | N/A | N/A | N/A |
| $n_c = 2$ | 0.500000 | 0.500000 | N/A | N/A |
| $n_c = 3$ | 0.277778 | 0.444444 | 0.277777 | N/A |
| $n_c = 4$ | 0.173927 | 0.326072 | 0.326072 | 0.173927 |

Table 3.4: Gauss quadrature weights

To handle the continuity conditions (3.7), we need the values of $v_i^C$ at all the mesh point. These are obtained using (3.13). We introduce these values as the NLP variables $z^C_{i,n_c+1}$ (which are not Lagrange basis coefficients). These values are thus obtained by the equations

$$z^C_{i,n_c+1} - v_i^C(1) = z^C_{i,n_c+1} - \sum_{k=0}^{n_c} z^C_{i,k} \cdot \tilde{\ell}_k(1) = 0, \quad \forall i \in [1..n_e - 1], \tag{3.21}$$

which are called *evaluation constraints*. Our continuity constraints are then given by the conditions

$$z_i^C(1) - z^C_{i+1}(0) = z^C_{i,n_c+1} - z^C_{i+1,0} = 0, \quad \forall i \in [1..n_e - 1].$$

There is an alternative way to determine the values of the continuity variables $z^C_{i,n_c+1}$. Equations (3.6) and (3.10) give

$$\begin{aligned} & h_i \cdot \int_0^1 \dot{v}_i^C(\tau) = v_i^C(1) - v_i^C(0) \\ \iff \quad & v_i^C(1) = v_i^C(0) + h_i \cdot \int_0^1 \dot{v}_i^C(\tau)\,\mathrm{d}\tau. \end{aligned}$$

By applying Gaussian quadrature to the integral (like we did for (3.16) and (3.17)), we get

$$z^C_{i,n_c+1} \approx z^C_{i,0} + h_i \cdot \sum_{k=1}^{n_c} \omega_k \cdot \dot{z}^C_{i,k}, \tag{3.22}$$

where $\dot{z}^C_{i,k}$ is defined in a manner analogous to how we defined $\dot{x}_{i,k}$ in (3.12) by using (3.10). This is called a *quadrature constraint* and can be used instead of (3.21) to get the values of the continuity variables. As shown in [Bie10, th. 10.1], the Gaussian quadrature approximation is exact for polynomials of degrees lesser than $2 \cdot n_c$, and since $\dot{v}^C_i$ is of degree $n_c - 1$, the approximation (3.22) is actually exact. The quadrature constraint is thus equivalent to the evaluation constraint. For now we adopt the evaluation constraint, but in Chapter 5 we implement both alternatives. This is discussed further in Section 5.2.4.12.

Since we do not have any collocation points at the mesh points, the DAE system is not satisfied at these points. This is not a problem in and of itself, but it does degrade the performance of the method when applied to DAE systems, especially those of high index. As discussed in [AP91], it is possible to modify the collocation method so that the approximated solution at least satisfies the algebraic (not the differential) equations at the mesh points. The corresponding Runge-Kutta method is called a *projected implicit Runge-Kutta* method. This is required to get the convergence orders noted in section 3.2.2. We do however not investigate projected Gauss collocation any further in this thesis.

Another consequence of not having a collocation point at the mesh points is that we can not transcribe Mayer type cost functions in the same straightforward manner as we did for Radau collocation in (3.19). We need the value of $z_{n_e}(1)$ to evaluate Mayer type cost functions, and also for terminal constraints in the case of free final time. We denote this value by $z_{n_e,n_c+1}$ and include it as an NLP variable. Its value is obtained by evaluating the collocation polynomials using (3.13). The transcription of Mayer type cost functions is thus given by

$$f(\tilde{z}, p) = \phi(t_f, \tilde{z}(t_f), p) = \phi(t_{n_e,n_c+1}, z_{n_e,n_c+1}, p) := \tilde{f}(Z)$$

and the value of

$$\tilde{z}(t_f) =: z_{n_e,n_c+1} = \left(\dot{x}_{n_e,n_c+1}, z^C_{n_e,n_c+1}, z^D_{n_e,n_c+1}\right),$$

which is called the *terminal value*, is obtained as

$$\begin{aligned}
\dot{x}_{n_e,n_c+1} &= \frac{1}{h_i} \cdot \sum_{l=0}^{n_c} x_{i,l} \cdot \dot{\tilde{\ell}}_l(1), \\
z^C_{n_e,n_c+1} &= \sum_{k=0}^{n_c} z^C_{n_e,k} \cdot \tilde{\ell_k}(1), \\
z^D_{n_e,n_c+1} &= \sum_{k=1}^{n_c} z^D_{n_e,k} \cdot \ell_k(1).
\end{aligned} \tag{3.23}$$

The constraints obtained by enforcing these equations are also called evaluation constraints. Note that the constraint obtained by enforcing equation (3.23) could be replaced by an equivalent quadrature constraint.

These are all the differences between the transcriptions obtained from Radau and Gauss collocation, so we now have all the information we need to fully transcribe the DOP into an NLP problem using Gauss collocation. Compared to the generic transcription in Section 3.2.3, we have added the mesh point values of $v_i^C$ and the terminal values of $z_i$ as NLP variables. We thus get a new expression for the vector containing all the NLP variables, given by

$$\begin{aligned} Z_G = \big(&z_{1,0}, z_{1,1}, z_{1,2}, \dots, z_{1,n_c}, z^C_{1,n_c+1},\\ &z^C_{2,0}, z_{2,1}, z_{2,2}, \dots, z_{2,n_c}, z^C_{2,n_c+1},\\ &z^C_{3,0}, z_{3,1}, z_{3,2}, \dots, z_{3,n_c}, z^C_{3,n_c+1},\\ &\vdots,\\ &z^C_{n_e-1,0}, z_{n_e-1,1}, z_{n_e-1,2}, \dots, z_{n_e-1,n_c}, z^C_{n_e-1,n_c+1},\\ &z^C_{n_e,0}, z_{n_e,1}, z_{n_e,2}, \dots, z_{n_e,n_c+1}, p\big)\,. \end{aligned}$$

Thus we also get a new value for the total number of NLP variables, given by

$$\begin{aligned} n_{Z_G} &= n_z + 2\cdot(n_e-1)\cdot n_z^C + n_e\cdot n_c\cdot n_z + n_z + n_p\\ &= (2 + n_e\cdot n_c)\cdot n_z + 2\cdot(n_e-1)\cdot n_z^C + n_p. \end{aligned}$$

The full transcription is given by (3.15). Like before, we just need to specify the $G$ in equation (3.15l). For Gauss collocation, we let $G = G_G$, where $G_G$ consists of five parts: evaluation constraints, continuity constraints, the terminal value of $\dot{x}$, the terminal value of $v^C$ and the terminal value of $v^D$. We thus have

$$\begin{aligned} G_G(Z_G) = \Bigg(&z^C_{1,n_c+1} - \sum_{k=0}^{n_c} z^C_{1,k}\cdot\tilde{\ell}_k(1), z^C_{2,n_c+1} - \sum_{k=0}^{n_c} z^C_{2,k}\cdot\tilde{\ell}_k(1), \dots,\\ &z^C_{n_e-1,n_c+1} - \sum_{k=0}^{n_c} z^C_{n_e-1,k}\cdot\tilde{\ell}_k(1),\\ &z^C_{1,n_c+1} - z^C_{2,0}, z^C_{2,n_c+1} - z^C_{3,0}, \dots, z^C_{n_e-1,n_c+1} - z^C_{n_e,0},\\ &\dot{x}_{n_e,n_c+1} - \frac{1}{h_i}\cdot\sum_{l=0}^{n_c} x_{i,l}\cdot\dot{\tilde{\ell}}_l(1),\\ &z^C_{n_e,n_c+1} - \sum_{k=0}^{n_c} z^C_{n_e,k}\cdot\tilde{\ell}_k(1), z^D_{n_e,n_c+1} - \sum_{k=1}^{n_c} z^D_{n_e,k}\cdot\ell_k(1)\Bigg)\,. \end{aligned}$$

We finish by counting our degrees of freedom to ensure that our transcription is degree-preserving. Since we already know that the Radau transcription is degree-preserving, see Section 3.3, we do not calculate the degrees of freedom explicitly.

Instead we simply note that compared to the Radau transcription, we have added $(n_e - 1) \cdot n_z^C$ NLP variables for the mesh point values of $v^C$ and $n_z$ variables for all the terminal values. All of these NLP variables are coupled with just as many evaluation (or quadrature) constraints. Thus the Gauss NLP formulation is also degree-preserving.

# Chapter 4

# Languages and software

In this chapter we present the languages and software used to implement the collocation algorithms.

## 4.1 Modelica

*Modelica*[1] is a high-level, object-oriented, domain-neutral and equation-based language designed for graphical and textual modeling of complex physical systems. Development started in 1996 and today Modelica is used in a wide variety of applications, such as chemistry, mechanics and electronics. Its industrial usage is increasing and it is today used extensively by e.g. automotive companies, such as BMW, Ford and Toyota, and power plant providers, such as ABB and Siemens.

Without going into the details of Modelica syntax, the VDP in section 1.2 can be described in Modelica by the following code.

```
model VDP

  Real x1(start=0, fixed=true) "First state";
  Real x2(start=1, fixed=true) "Second state";

  input Real u "Control signal";

equation

  der(x1) = (1 - x2^2) * x1 - x2 + u;
  der(x2) = x1;

end VDP;
```

[1] https://modelica.org/

### 4.1.1 Optimica

Modelica is largely intended for simulation-based analysis. To accommodate the need for conveniently formulating dynamic optimization problems based on Modelica models, the Modelica extension *Optimica* was developed as a part of the Ph.D. thesis [Åk07], as described in [Åk08]. Basic features of Optimica used to formulate an optimization problem includes specifying free optimization variables (control variables and parameters), a Bolza type objective function, variable bounds and general constraints.

The VDP OCP in section 1.2 can be described in Optimica, using the Modelica model from section 4.1, by the following code.

```
optimization VDP_OCP(
    objectiveIntegrand=x1^2 + x2^2 + u^2,
    startTime=0, finalTime=10)

// The VDP Modelica model
extends VDP(u(free=true, max=0.75));

end VDP_OCP;
```

## 4.2 Python

*Python*[2] is a programming language often described by adjectives such as high-level, interactive, interpretive, object-oriented, dynamic, open-source and general-purpose. One of these purposes is scientific computation, which is performed with the use of the Python packages NumPy[3] and SciPy[4] as well as the plotting package matplotlib[5], which is designed to resemble the plotting functionalities of MATLAB[6]. All implementation made as a part of this thesis will be in Python.

## 4.3 JModelica.org

*JModelica.org*[7] is a package for simulation and optimization of Modelica models. It is developed in collaboration between industry and academia, with the purpose of creating an industrially viable platform using state of the art algorithms to analyze complex physical systems. Python is used as a glue language to create a user-friendly interface to all of JModelica.org's components.

[2] http://www.python.org/
[3] http://numpy.scipy.org/
[4] http://www.scipy.org/
[5] http://matplotlib.sourceforge.net/
[6] http://www.mathworks.com/products/matlab/index.html
[7] http://www.jmodelica.org/

Standard Modelica tools, such as Dymola[8] and SimulationX[9], focus on simulation, whereas JModelica.org also puts a lot of focus on optimization. The current main optimization algorithm in JModelica.org is a Radau collocation method implemented in C[10]. The goal of this thesis is to implement new collocation algorithms in JModelica.org with the use of CasADi, resulting in simpler, easier to maintain and extend, and perhaps even more efficient than the existing algorithm. The efficiency of the new algorithms developed as a part of this thesis are compared to JModelica.org's present C-implemented algorithm in Chapter 6.

## 4.4 CasADi

In optimization it is important to efficiently calculate function derivatives. To this end we use *CasADi* (**C**omputer **a**lgebra **s**ystem with **A**utomatic **Di**fferentaion). Once a symbolic representation consisting of CasADi objects of an NLP problem has been created, CasADi provides all the derivative information needed for the numerical solution of the problem with very little effort from the user. From CasADi's official website [A+11]:

"*CasADi is a minimalistic computer algebra system implementing automatic differentiation in forward and adjoint modes by means of a hybrid symbolic/numeric approach. It is designed to be a low-level tool for quick, yet highly efficient implementation of algorithms for numerical optimization.*"

CasADi is developed by mainly Joel Andersson and Joris Gillis at K.U.Leuven, Belgium. It is implemented in C++ and has (nearly) fully-featured Python and Octave interfaces. The first release was made in December 2010.

For a very thorough introduction to *automatic differentiation* (AD), see [GW08]. Conventional AD tools use graphs whose nodes are restricted to scalar-valued unary and binary operations, allowing for very efficient evaluation. This possibility is available in CasADi in the form of *SX* (**S**calar e**X**pression) graphs. CasADi also has a more novel approach, where the nodes are allowed to contain more general operations, such as branching and matrix operations. These are called *MX* (**M**atrix e**X**pression) graphs. The additional generality available in MX graphs sometimes results in slower performance, but if utilized correctly, can in some situations significantly increase the performance.

An important consequence of restricting the nodes to scalar expressions, is that matrix operations are expanded into multiple scalar-valued operations. This can result in very large graphs, which take considerable time to construct and handle while also consuming a lot of memory. Since MX graphs allow matrix expressions, this is avoided. MX nodes can also contain function calls, further reducing the graph size.

[8] http://www.3ds.com/products/catia/portfolio/dymola
[9] http://www.itisim.com/simulationx.html
[10] http://www.open-std.org/JTC1/SC22/WG14/

Another interesting possibility introduced by allowing function calls is the usage of user-provided black-box functions, e.g. numerical integrators.

The SX and MX objects are in many ways similar to each other from a user point of view, but are treated very differently by CasADi. For an introduction to the theory behind CasADi, see [AHD10]. Which of the two that allows for the best performance is dependent on the problem to be solved and it is up to the user to decide which representation to use. It is often wise to use a combination of the two and rarely wise to use MX exclusively. In Section 5.2.2 we briefly discuss what kind of graphs we use for our algorithms.

CasADi also comes with several interfaces to other software useful for numerical optimization. For solving NLP problems, interfaces to Ipopt and KNITRO[11] are available. For numerical integration there are interfaces to CVODES and IDAS from SUNDIALS[12]. CasADi has additional interfaces, some of which are only used internally. The interface to Ipopt is used extensively in this thesis.

## 4.5 Ipopt

We dedicated Chapter 3 to transcribing the DOP into an NLP problem. To actually solve the NLP problem (thus obtaining an approximate solution to the DOP), we use *Ipopt*. From Ipopt's official website[13]:

"*Ipopt (**I**nterior **P**oint **OPT**imizer, pronounced eye-pea-Opt) is a software package for large-scale nonlinear optimization. It is designed to find (local) solutions of mathematical optimization problems of the form*

$$\min_{x \in \mathbb{R}^n} f(x)$$
$$\textit{subject to} \begin{cases} g_L \leq g(x) \leq g_U, \\ x_L \leq x \leq x_U, \end{cases}$$

*where $f(x) : \mathbb{R}^n \to \mathbb{R}$ is the objective function, and $g(x) : \mathbb{R}^n \to \mathbb{R}^m$ are the constraint functions. The vectors $g_L$ and $g_U$ denote the lower and upper bounds on the constraints, and the vectors $x_L$ and $x_U$ are the bounds on the variables $x$. The functions $f(x)$ and $g(x)$ can be nonlinear and nonconvex, but should be twice continuously differentiable.*"

Note that the problem formulation used in Ipopt is slightly different than the NLP problem we defined in (2.2). They are however equivalent, and the transformation from our form to the one used in Ipopt is handled by CasADi's Ipopt interface. Ipopt uses an interior-point algorithm with a filter line-search method. Its underlying theory and implementation is described in [WB06].

[11] http://www.ziena.com/knitro.htm
[12] https://computation.llnl.gov/casc/sundials/main.html
[13] http://www.coin-or.org/Ipopt/

# Chapter 5

# Implementing collocation algorithms in JModelica.org

In this chapter we combine the theory of Chapter 3 with the languages and software of Chapter 4 to implement new optimization algorithms in JModelica.org based on collocation methods. We start by presenting an overview of JModelica.org's optimization framework and how the developed algorithms are integrated into JModelica.org. We then proceed to describe how the algorithms are implemented and also present some new theory regarding the implementation.

## 5.1 Optimization in JModelica.org

Figure 5.1 shows a diagram depicting an overview of the part of JModelica.org's optimization framework that is used for the algorithms developed in this thesis.

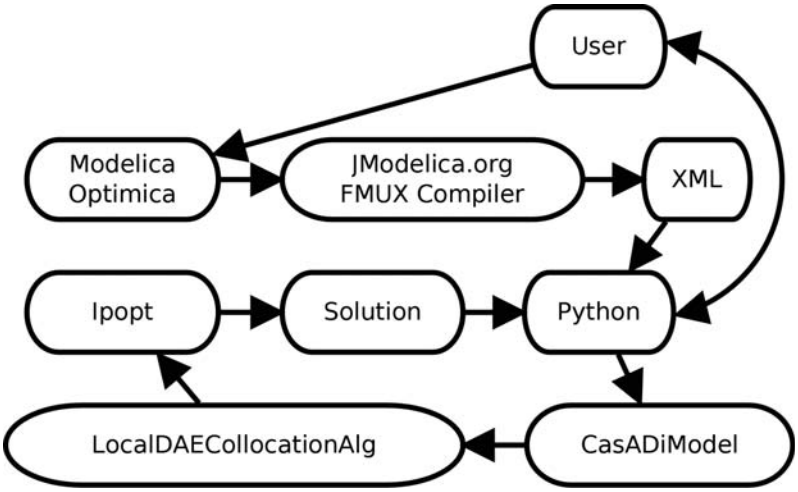


Figure 5.1: Overview of JModelica.org's optimization framework

The user starts by defining the system model in Modelica and the dynamic optimization problem in Optimica. JModelica.org provides an Eclipse[1] plugin for textual editing of Modelica code, but the user is free to create their Modelica and Optimica files in any editor. The user interaction is carried out in Python. The Optimica file is, via Python, sent to JModelica.org's FMUX compiler. The FMUX compiler creates an XML file from the Optimica file, which describes the DOP. The XML format is based on the Functional Mockup Interface[2], which has been extended for additional purposes, in particular dynamic optimization. This XML format is described in [PkC10] and is also used by CasADi.

The XML file is used to create an instance of the Python class `CasADiModel`, which represents the dynamic optimization problem described in the original Optimica file. This class is described in 5.2.1. The `CasADiModel` object is then used by `LocalDAECollocationAlg` to transcribe the DOP into an NLP problem. This NLP problem is then solved by Ipopt. The class `LocalDAECollocationAlg` is a collection of all the algorithms developed as a part of this thesis and is described in the succeeding sections of this chapter. The solution is then written to a result file in a format compliant with Dymola. The solution is also represented by a Python object which is returned to the user, allowing the user to freely analyze the data in Python, e.g. plotting it using matplotlib. How to use JModelica.org's optimization framework is described fully in [AB11].

## 5.2 A comprehensive collocation algorithm

In this section we will describe how the algorithms of this thesis are implemented and integrated into JModelica.org's optimization framework.

### 5.2.1 Implementation overview

The algorithms developed in this thesis are contained in a single class, which is named `LocalDAECollocationAlg`. A diagram depicting an overview of the main Python classes and modules related to this class is shown in Figure 5.2.

The XML document created by JModelica.org's FMUX compiler is used by the `CasADiModel` class to create a representation of the corresponding DOP. This class relies on both JModelica.org's XML parser and CasADi's `FlatOCP` class to do this. The `CasADiModel` class contains an `optimize` method, which takes two parameters: an algorithm and a corresponding algorithm options object. JModelica.org currently has 2 optimization algorithms that are used with `CasADiModel` (other algorithms use other model classes, whose objects can be created in a similar manner given the Modelica/Optimica file): `LocalDAECollocationAlg` (the algorithm of this thesis) and `PseudoSpectralAlg`. The latter algorithm is intended for

[1] http://www.eclipse.org/
[2] http://www.modelisar.com/

*global* (also called *pseudospectral*) rather than *local* collocation. Both these algorithm classes inherit the abstract JModelica.org algorithm class `AlgorithmBase`. This abstract class is inherited by all JModelica.org algorithms, simulation and optimization algorithms alike.

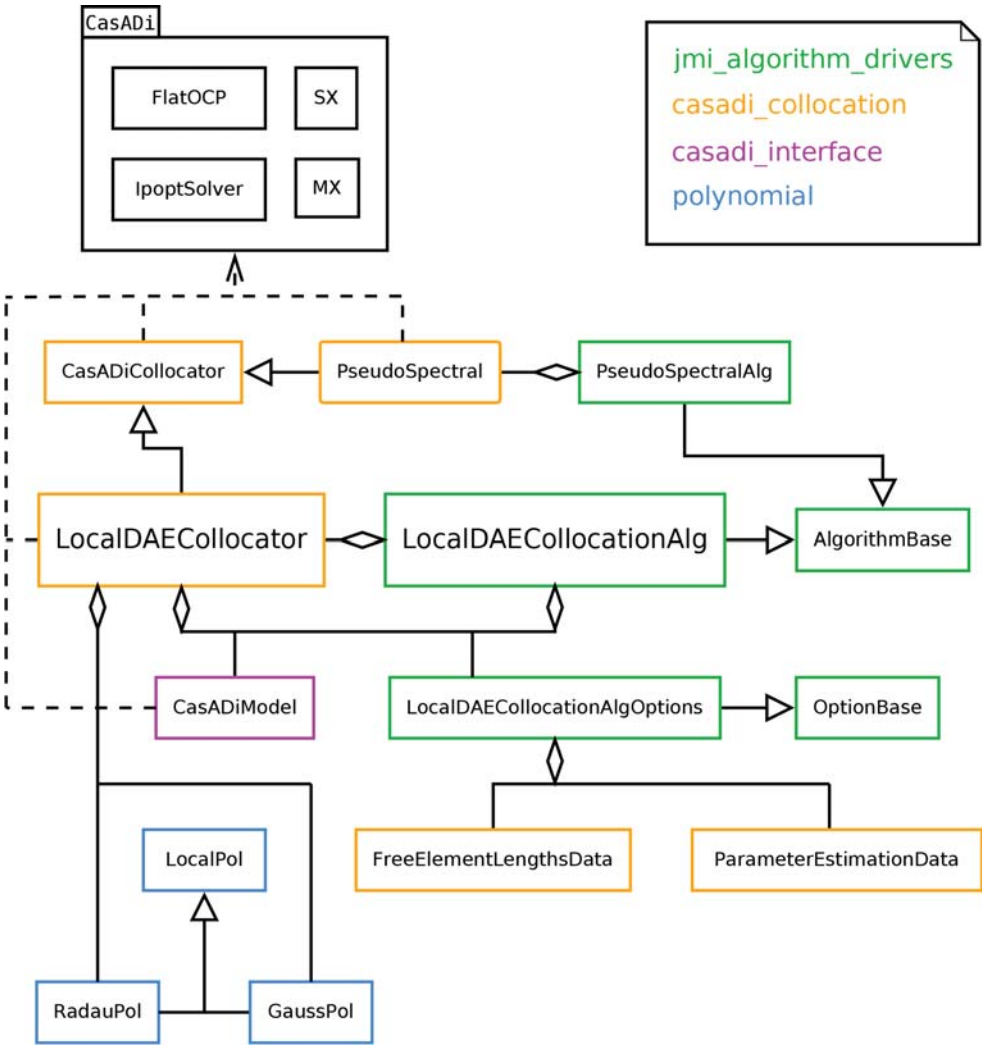


Figure 5.2: Diagram of all Python classes and modules in JModelica.org related to the `LocalDAECollocationAlg` algorithm, color-coded according to their respective modules

Global collocation means that the number of elements are considerably fewer than the number of collocation points in each element, whereas local collocation means that the number of collocation points in each element are considerably fewer than the number of elements. Global collocation is more accurate and efficient than local collocation for problems with relatively smooth solutions, but the less smooth the problem is, the more the performance of global collocation deteriorates, whereas

local collocation does not suffer in the same manner. See [HR07] for a deeper discussion on the differences between local and global collocation.

The two JModelica.org optimization algorithms `PseudoSpectralAlg` as well as `LocalDAECollocationAlg` are essentially very similar, but there are some important differences. One fundamental difference is that `PseudoSpectralAlg` is only implemented for ODE systems, whereas `LocalDAECollocationAlg` handles DAE systems of index zero or one. See [AB11, ch. 8.8] for a description of JModelica.org's pseudospectral algorithm and [Ben05] as well as [Hun07] for an in-depth description of its underlying theory.

The second parameter of `CasADiModel.optimize` is an options object belonging to the algorithm. For the algorithm `LocalDAECollocationAlg` this object should be either an instance of the class `LocalDAECollocationAlgOptions`, which is a class inheriting the abstract JModelica.org class `OptionBase` (similar to `AlgorithmBase` for algorithms), or a Python dictionary. This object lets the user specify certain algorithm behavior, e.g. in our case things such as the parameters $n_e$ and $n_c$ as well as whether to use Radau or Gauss collocation. The available options are described in Section 5.2.4.

In `CasADiModel.optimize`, the supplied algorithm is initialized. The most interesting part in the initialization of the `LocalDAECollocationAlg` algorithm is the creation of a `LocalDAECollocator` object. This is where the collocation is performed. This process is described in Section 5.2.2. After the algorithm has been initialized, its `solve` method is called, where the NLP problem created by `LocalDAECollocator` is solved using Ipopt. So in summary, the classes `LocalDAECollocationAlg` and `LocalDAECollocationAlgOptions` are mainly used as interfaces between the user and the algorithm. The actual collocation is done inside of `LocalDAECollocator`. A Python script demonstrating how to use the optimization algorithm is available in Section 5.2.3.

### 5.2.2 Collocation using CasADi

In this section we briefly describe how some of the information contained in the classes shown in Figure 5.2 is used by the class `LocalDAECollocator` to define the NLP problem using CasADi. The computations are performed in the following steps.

1. The amount of NLP variables is calculated, much in the same manner as we did in Sections 3.3 and 3.4 to ensure that the transcriptions are degree-preserving. The number of variables depend on several of the algorithm options, such as `blocking_factors` and `eliminate_der_var`, which are described in 5.2.4. Once the number of NLP variables are known, the NLP variables are created as either SX or MX objects depending on the chosen graph.

2. The NLP variables are used to construct all the constraints except the variable bounds, since the variable bounds are treated separately by Ipopt. We do this

by defining a function evaluating the left-hand sides of the Equations (3.15b) to (3.15d) and (3.15g) to (3.15l). A fundamental part of creating the constraints is the polynomial operations. This is handled by JModelica.org's `polynomial` module, which performs the required calculations in a robust manner. The cost function is then created in a straightforward manner.

The complete AD graphs for the cost and constraint functions are then finished. If the algorithm option `graph` is set to SX, these will be SX graphs, and if `graph` is set to MX or expanded MX, these will be MX graphs. If an SX graph is used, the graph has been created using only SX-related objects. If an MX graph is used, a suitable mixture of SX and MX objects has been used. For example, the DAE residual function is defined as an SXFunction, which is then used to generate function calls in the MX graph to create the DAE-related constraints.

3. At this point the graphs for MX and expanded MX are identical. If expanded MX is chosen, the created MX graph is expanded into an SX graph. The point of the expanded MX graph is that creating and expanding an MX graph may be quicker than creating the SX graph from scratch, whereas the SX graph may be quicker to evaluate than the original MX graph. In these cases the expanded MX graph will be the most efficient.

4. If the algorithm option `exact_hessian` is enabled, the Hessian of the Lagrangian function defined by (2.3) is computed using CasADi.

5. The initial guesses, variable bounds and scaling factors (see 5.3) are computed using the information in `CasADiModel`. If initial trajectories have been provided in the `init_traj` algorithm option, these are instead used to compute the initial guesses for the provided trajectories.

6. Finally CasADi's interface to Ipopt is used to create an `IpoptSolver` object.

All this is done during the initialization of the `LocalDAECollocationAlg` object (which initializes the `LocalDAECollocator` object). When the `solve` method of the `LocalDAECollocationAlg` object is called, the created NLP problem is solved by the `IpoptSolver` object.

### 5.2.3 Algorithm demonstration

In this section we present a Python script using `LocalDAECollocationAlg` to solve the optimal control problem based on a Van der Pol oscillator, defined by the Modelica and Optimica models shown in 4.1 and used in 1.2. The Modelica and Optimica classes are assumed to be in a single file called `VDP.mop`, located in the same directory as the Python script. We also show how to extract the solution from the result and how to plot it using matplotlib.

```
# Import numerical and plotting libraries
import numpy as N
import matplotlib.pyplot as plt

# Import the required JModelica.org Python classes
from jmodelica import compile_fmux
from pyjmi import CasadiModel

# Compile and load model
fmux_file = compile_fmux("VDP_OCP", "VDP.mop")
model = CasadiModel(fmux_file)

# Create options object with default settings
opts = model.optimize_options(algorithm="LocalDAECollocationAlg")

# Set some options
opts['n_e'] = 40
opts['n_cp'] = 5
opts['graph'] = "MX"
opts['exact_hessian'] = False
opts['discr'] = "LG"
opts['quadrature_constraint'] = False

# Solve the OCP
result = model.optimize(algorithm="LocalDAECollocationAlg", options=opts)

# Extract variable profiles
x1 = result['x1']
x2 = result['x2']
u = result['u']
time = result['time']

# Plot
plt.figure(1)
plt.clf()
plt.subplot(3, 1, 1)
plt.plot(time, x1)
plt.grid()
plt.ylabel('x1')

plt.subplot(3, 1, 2)
plt.plot(time, x2)
plt.grid()
plt.ylabel('x2')

plt.subplot(3, 1, 3)
plt.plot(time, u)
plt.grid()
plt.ylabel('u')
plt.xlabel('t')
plt.show()
```

Running the script produces the plot shown in Figure 5.3, which is very similar to the one shown in Figure 1.1, but slightly different due to the chosen algorithm options and relatively simple plotting.

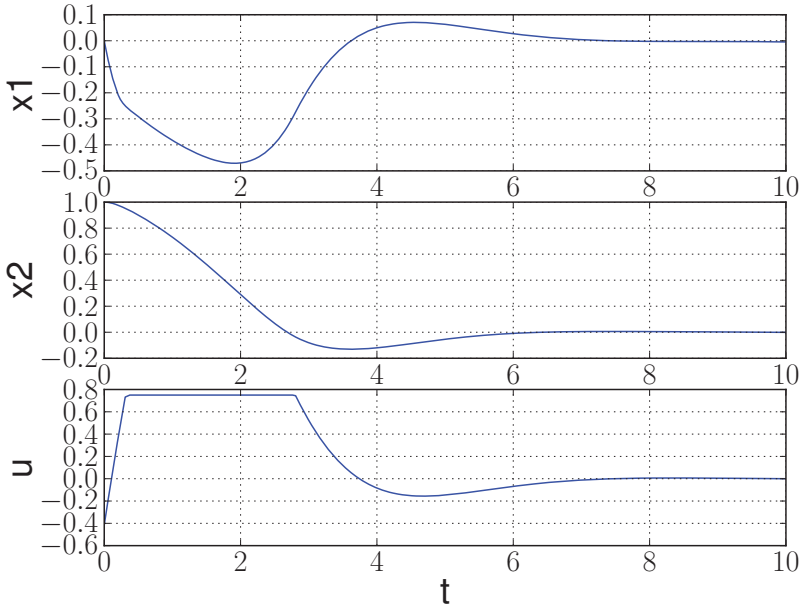


Figure 5.3: Optimal control of a VDP oscillator using tenth-order Gauss collocation

### 5.2.4 Algorithm options

In this section we present the algorithm options for `LocalDAECollocationAlg`. For each option we first present a copy of the part of the Python docstring belonging to `LocalDAECollocationAlgOptions` corresponding to the option, and then some further explanations.

#### 5.2.4.1 `n_e`

```
Number of finite elements.

Type: int
Default: 50
```

This options specifies the value of the variable denoted by $n_e$ in Chapter 3.

#### 5.2.4.2 `hs`

```
Element lengths.

Possible values: None, iterable of floats and "free"

None: The element lengths are uniformly distributed.
```

```
iterable of floats: Component i of the iterable specifies the
length of element i. The lengths must be normalized in the sense
that the sum of all lengths must be equal to 1.

"free": The element lengths become optimization variables and are
optimized according to the algorithm option
free_element_lengths_data.
WARNING: This option is very experimental and will not always give
desirable results.

Type: None, iterable of floats or string
Default: None
```

This option specifies the values of the variables denoted by $h_i$ in Chapter 3. By setting `hs = None`, these variables get the values

$$h_i = \frac{t_f - t_0}{n_e}, \quad \forall i \in [1..n_e].$$

By instead providing a list of floats, the user can freely specify all the element lengths.

The final possibility is `hs = "free"`. This uses all the element lengths as optimization variables, using the class `FreeElementLengthsData`. This option is described more in detail in Section 5.5.

#### 5.2.4.3 free_element_lengths_data

```
Data used for optimizing the element lengths if they are free.
Should be None when hs != "free".

Type: None or
jmodelica.optimization.casadi_collocation.FreeElementLengthsData
Default: None
```

This option is described more in detail in Section 5.5.

#### 5.2.4.4 n_cp

```
Number of collocation points in each element.

Type: int
Default: 3
```

This options specifies the value of the variable denoted by $n_c$ in Chapter 3.

#### 5.2.4.5 discr

```
Determines the collocation scheme used to discretize the problem.

Possible values: "LG" and "LGR"

"LG": Gauss collocation (Legendre-Gauss)

"LGR": Radau collocation (Legendre-Gauss-Radau)

Type: str
Default: "LGR"
```

Whether to use Radau or Gauss collocation, as discussed in Chapter 3.

#### 5.2.4.6 graph

```
CasADi graph type. Possible values are "SX", "MX" and
"expanded_MX".

Type: str
Default: "SX"
```

What kind of CasADi graphs to use, as discussed in Sections 4.4 and 5.2.2.

#### 5.2.4.7 rename_vars

```
Rename NLP variables according to their corresponding
Modelica/Optimica names. This only works if graph == "SX". This is
done in an inefficient manner and should only be used for
investigative purposes.

Type: bool
Default: False
```

The `LocalDAECollocator` instance used to construct the NLP problem is stored as an attribute named `solver` in the result object returned after an optimization. Thus the user has access to all the CasADi objects created during the NLP problem construction. These objects can be printed in Python to get legible expressions for e.g. all the constraints. By default, the NLP variable names are not very descriptive, but by enabling this option, the NLP variables are renamed according to their corresponding name in the Modelica/Optimica code. For example, the NLP variable corresponding to the value of a state named `x` in element 3 and collocation point 2 will be named `x_3_2`.

#### 5.2.4.8 `write_scaled_result`

```
Return the scaled optimization result if set to True, otherwise
return the unscaled optimization result. This option is
only applicable when the CasadiModel has been instantiated with
scale_variables=True. This option is only intended for debugging.

Type: bool
Default: False
```

Scaling is discussed in Section 5.3.

#### 5.2.4.9 `result_mode`

```
Specifies the output format of the optimization result.

Possible values: "collocation_points", "element_interpolation" and
"mesh_points"

"collocation_points": The optimization result is given at the
collocation points.

"element_interpolation": The values of the variable trajectories
are calculated by evaluating the collocation polynomials. The
algorithm option n_evaluation_points is used to specify the
evaluation points within each finite element.

"mesh_points": The optimization result is given at the
mesh points.

Type: str
Default: "collocation_points"
```

If `result_mode = "collocation_points"`, then the values denoted by

$$z_{i,k}, \quad \forall (i,k) \in [1..n_e] \times [1..n_c]$$

in Chapter 3 are given as the result.

If `result_mode = "element_interpolation"`, then the values of the approximative solution, denoted by $\tilde{z}$ in Chapter 3, at specific time points are given as the result. These specific time points are determined by the algorithm option `n_eval_points`. This is called *dense output*.

#### 5.2.4.10 `n_eval_points`

```
The number of evaluation points used in each element when the
algorithm option result_mode is set to "element_interpolation". One
evaluation point is placed at each element end-point (hence the
option value must be at least 2) and the rest are distributed
```

```
uniformly.

Type: int
Default: 20
```

See Section 5.2.4.9.

#### 5.2.4.11 `blocking_factors`

```
The iterable of blocking factors, where each element corresponds to
the number of elements for which all the control profiles should be
constant. For example, if blocking_factors == [2, 1, 5], then
u_0 = u_1 and u_3 = u_4 = u_5 = u_6 = u_7. The sum of all elements
in the iterable must be the same as the number of elements.

If blocking_factors is None, then the usual collocation polynomials
are instead used to represent the controls.

Type: None or iterable of ints
Default: None
```

Control signals are often discrete rather than continuous, in which case they are sometimes better modeled as being piecewise constant. Piecewise constant control signals are also often preferred during model predictive control. This possibility is enabled by this option. Currently all the blocking factors must be shared by all the control variables of the model.

#### 5.2.4.12 `quadrature_constraint`

```
Whether to use quadrature continuity constraints. This option is
only applicable when using Gauss collocation. It is incompatible
with eliminate_der_var set to True.

True: Quadrature is used to get the values of the states at the
mesh points.

False: The Lagrange basis polynomials for the state collocation
polynomials are evaluated to get the values of the states at the
mesh points.

Type: bool
Default: True
```

Quadrature constraints are given by the equations (3.22).

In order to formulate the quadrature constraints, we need the values of $\dot{z}_{i,k}$. The values of $\dot{x}_{i,k}$ are already available as NLP variables, but the remaining values of $\dot{z}^C_{i,k}$ need to be calculated for the quadrature constraint. If all the values of $\dot{z}_{i,k}$ are

already available, quadrature constraints are believed to be more efficient, and perhaps even more robust, than evaluation constraints but this has not been thoroughly investigated.

Calculating the values of $\dot{z}_{i,k}^C$ which are not already available is not a cheap procedure, and it may thus be more efficient to use evaluation constraints if such values do not exist. However, as we do not support continuity for non-state variables in the current implementation (see 5.6), all the needed values are already available, and quadrature constraints are thus believed to be strictly superior in our case. The exception is if the derivative variables have been eliminated, see Section 5.2.4.13. In this case the only supported continuity constraints are evaluation constraints, see (3.21).

#### 5.2.4.13 `eliminate_der_var`

```
True: The variables representing the derivatives are eliminated
via the collocation equations and are thus not a part of the NLP,
with the exception of \dot{x}_{1, 0}, which is not eliminated since
the collocation equations are not enforced at t_0.

False: The variables representing the derivatives are kept as NLP
variables and the collocation equations enter as constraints.

Type: bool
Default: False
```

Elimination of derivative variables is discussed in Section 5.4.2.

#### 5.2.4.14 `eliminate_cont_var`

```
True: Let the same variables represent both the values of the
states at the start of each element and the end of the previous
element.

False:
For Radau collocation, the extra variables x_{i, 0},
representing the states at the start of each element, are created
and then constrained to be equal to the corresponding variable at
the end of the previous element for continuity.

For Gauss collocation, the extra variables x_{i, n_cp + 1},
representing the states at the end of each element, are created
and then constrained to be equal to the corresponding variable at
the start of the succeeding element for continuity.

Type: bool
Default: False
```

Elimination of continuity variables is discussed in Section 5.4.1.

#### 5.2.4.15 init_traj

```
Variable trajectory data used for initialization of the
optimization problem.

Type: None or jmodelica.io.DymolaResultTextual
Default: None
```

When solving the NLP problem, Ipopt needs an initial guess for all the variables. One way of obtaining an initial guess for the NLP variables is by specifying a constant initial guess for each of the DOP variables, and then use each such initial guess for all the corresponding NLP variables. The initial guess for a DOP variable can be set in Optimica using the variable attribute `initialGuess`. If no initial guess is provided, the start value is used as the initial guess. If neither initial guess nor initial value is provided, then 0 is used as an initial guess.

In simple cases, Ipopt is not sensitive to the initial guess and will converge quickly even if the initial guesses for all the DOP variables are 0 (provided that the DAE is defined for these variable values). In more advanced cases, it is important that the initial guesses of the DOP variables are relatively close to the optimal solution to get quick convergence (or to get convergence at all). In even more advanced cases, a constant initial guess for a DOP variable will not suffice. Optimica currently has no support for supplying such an initial guess, which is why we implement this algorithm option. By supplying a result object from a simulation or a previous optimization of the model, the previous results are linearly interpolated to get the variable values at the collocation points, and these are then used as initial guesses. It is not necessary to provide initial trajectories for all of the variables.

#### 5.2.4.16 parameter_estimation_data

```
Parameter estimation data used for solving parameter estimation
problems.

Type: None or
pyjmi.optimization.casadi_collocation.ParameterEstimationData
Default: None
```

Optimica is not flexible enough to fully describe all the kinds of parameter estimation problems which we consider. Thus we implement this algorithm option, which lets us fully describe either a continuous or discrete parameter estimation problem using the `ParameterEstimationData` class, whose documentation is available below.

ParameterEstimationData

```
Data used to define the cost function for parameter estimation problems.

Parameters::

    Q --
        Weighting matrix.
```

```
    Type: rank 2 ndarray

measured_variables --
    List of the names of the measured variables.

    Type: list of strings

data --
    Object containing the measurement data.

    If data is a function, it should take a time point as its argument
    and return an array where element j contains the measured value of
    measured_variables[j] at the given time point.

    If data is a matrix, data[i][0] should contain measurement point i
    and data[i][j] should contain the measured value of
    measured_variables[j] at measurement point i.

    Type: function or rank 2 ndarray

discrete --
    Whether to perform discrete or continuous parameter estimation.

    NOTE: Discrete parameter estimation is not yet supported!

    Continuous parameter estimation uses the cost function

    .. math::

        f = \int_{t_0}^{t_f} (y(t) - y_m(t)) \cdot Q \cdot
        (y(t) - y_m(t))\,\mathrm{d}t,

    where y is a function created by gluing together the collocation
    polynomials for the measured variables at all the mesh points and
    y_m is a function providing the measured values at a given time
    point. If the parameter data is a matrix, the data are linearly
    interpolated to create the y_m function. If data is a function,
    then this function defines y_m.

    Discrete parameter estimation uses the cost function

    .. math::

        f = \sum_{i = 1}^{n_m} (y(t_i) - y_m(t_i)) \cdot
        Q \cdot (y(t_i) - y_m(t_i)),

    where y is the optimized values of the measured variables and y_m
    is the measured values of the measured variables. This option
    requires the parameter data to be a matrix. It is also required
    that each measurement point coincides with either a collocation
    point or a mesh point.

    Type: bool
    Default: False

eps --
    In the case of discrete parameter estimation, the measurement point
    t is considered to coincide with the collocation or mesh point v if
    and only if |t - v| / (t_f - t_0) < eps.

    Type: float
    Default: 1e-5
```

#### 5.2.4.17 exact_hessian

```
True: The Hessian of the Lagrangian function is obtained via CasADi
and supplied to Ipopt.

False: Ipopt uses a quasi-Newton method.

WARNING: exact_hessian is very slow in combination with MX graphs.

Type: bool
Default: True
```

Calculating the Hessian belonging to an MX graph is currently very slow in CasADi, leading to long problem initialization times. The evaluation of the Hessian is however quick.

#### 5.2.4.18 casadi_options

```
In addition, CasADi options can be provided in the options
casadi_options_f, casadi_options_g and casadi_options_l for the NLP
objective, constraint and Lagrangian functions respectively. For a
complete list of CasADi options, please consult the CasADi
documentation.

CasADi options are set using the syntax for dictionaries::

    >>> opts['casadi_options_g']['numeric_jacobian'] = True
```

CasADi also has its own set of options regarding the evaluation of functions. We do not discuss all these possibilities here, but instead refer to CasADi's documentation for `SXFunction`s and `MXFunction`s available in the C++ API docs at [A+11], where all the options are described. The algorithm's default CasADi options are the same as CasADi's default options.

#### 5.2.4.19 IPOPT_options

```
IPOPT options can be provided in the option IPOPT_options. For a
complete list of IPOPT options, please consult the IPOPT
documentation available at
http://www.coin-or.org/Ipopt/documentation/.

IPOPT options are set using the syntax for dictionaries::

    >>> opts['IPOPT_options']['max_iter'] = 200
```

Ipopt *also* has its own set of options regarding the solution of the NLP problem. The algorithm's default Ipopt options are the same as Ipopt's default options, with the exception of `max_iter` being set to 2000.

## 5.3 Scaling

When Ipopt solves the NLP problem, it is important that the NLP variables and equations are well-scaled, meaning that the variables attain values close to 1 and that the rows and columns of the Jacobian of all the constraint functions are of the same magnitude. Equation scaling is done internally in Ipopt. CasADi also offers equation scaling before the call to Ipopt, which is used by setting the option `scale_equations` to `True` when instantiating the `CasADiModel`.

Variable scaling is however left up to the user, as neither Ipopt nor CasADi know a priori what variable values are reasonable. The variable scaling is done by introducing scale factors $z_s$, which are used to create the new variables

$$\bar{z} = z_s \cdot z,$$

based on the old optimization variables $z$. The scaled variables $\bar{z}$ are then used instead of the old variables $z$ as optimization variables, and all occurrences of $z$ in the cost and constraint functions are substituted by $\bar{z}/z_s$.

The scaling factors are entirely determined by the user by setting the Modelica attribute `nominal`. The scaling factors for the NLP variables corresponding to the DOP variable are then set to 1 divided by the value given by the `nominal` attribute. If no nominal value is set, the scaling factor is set to 1, i.e. no scaling is done.

See [NW06] for more on scaling.

## 5.4 Variable elimination

At the beginning of Section 3.2.3 we chose the variables $\dot{x}_{i,k}$, $z_{i,k}$ for all $k \in [1..n_c]$, $p$, $z_{1,0}$ and $z^C_{i,0}$ as NLP variables. For Gauss collocation, we further added the variables $z^C_{i,n_c+1}$ as NLP variables. As it turns out, many of these can be eliminated by substituting them by the expressions in their coupled constraints. This has various benefits and drawbacks.

### 5.4.1 Continuity variables

For Radau collocation, the continuity variables $z^C_{i,0}$ are constrained to be equal to $z^C_{i-1,n_c}$. They can thus be easily eliminated by replacing all occurrences of $z^C_{i,0}$ by $z^C_{i-1,n_c}$. However, this may be unwise, as the resulting system is less sparse (since the variables $z^C_{i-1,n_c}$ will occur in more equations). So this is a trade-off between NLP size (number of variables and constraints) versus sparsity. Experience shows that the additional sparsity usually results in better performance, as discussed in [Bet10, ch. 4].

For Gauss collocation, we have both $z_{i,0}^C$ and $z_{i,n_c+1}^C$ as continuity variables in each element. It would be possible to eliminate both of them, but this would tremendously reduce the NLP sparsity and is thus not implemented. Instead we just provide the option of eliminating $z_{i,n_c+1}^C$.

### 5.4.2 Derivative variables

The state derivative variables $\dot{x}_{i,k}$ obtained by the collocation constraints can also be eliminated. This has even bigger impact on NLP sparsity than the elimination of continuity variables. But rather than just reducing the NLP size, this has an additional benefit. As discussed in Section 5.3, NLP scaling is very important. A particularly difficult part of scaling is the scaling of derivatives. The magnitude of a state derivative is often very different from the magnitude of the state itself. It would thus be desirable to scale them independently of each other. This is possible for collocation methods by scaling the collocation equations. This has however not been implemented in this thesis. However, by eliminating the derivative variables using the collocation equations, there is no longer a need to scale them, and this problem is thus avoided.

## 5.5 Free element lengths

Thus far we have always used constant element lengths, and unless the user provides a mesh, we use an equidistant mesh. There is a lot to be gained in terms of overall accuracy by instead using a mesh that has small elements in regions where the solution is very nonlinear, and larger elements in regions where the solution is less nonlinear. If the user has an idea of what such a mesh would look like, they can provide it. That is however a rare situation. There is thus a need for automatically finding such a mesh.

One approach to this is to let the element lengths $h_i$ enter as NLP variables. The element lengths essentially affect two things: the discretization error and the cost. Ipopt just minimizes the cost and has no knowledge of the discretization error, so when a decrease in the cost results in an increase in the discretization error, we get the opposite of the desired result. We thus need to somehow incorporate a measure of the discretization error into the cost function.

A naive approach, which is the one mentioned in Section 5.2.4.3, is to assume that the nonlinearity of the solution is proportional to the derivative. We thus want the element lengths to be small where the derivative is big, and vice versa. This idea inspires the augmented cost function

$$\hat{f} = f + c \cdot \sum_{i=1}^{n_e} \left( h_i^a \cdot \int_{t_i}^{t_{i+1}} \dot{v}_i^C(t) \cdot R \cdot \dot{v}_i^C(t) \, \mathrm{d}t \right),$$

where $f$ is the original cost function, $R$ weights the various derivatives and $c$ as well as $a$ are method parameters. The integral is computed using Gaussian quadrature. With appropriate choices of $c$, $a$ and element length bounds, this approach works in cases where the nonlinearity actually is proportional to the derivative (to the power of two). However, this is rarely the case in practice. A common example of the nonlinearity not being proportional to the derivative, is when the solution is non-constant and (locally) linear, or at least close to being linear. Such is the case for e.g. the continuously stirred tank reactor discussed in Section 6.3 and the combined cycle power plant discussed in Section 6.6. It is possible to improve the augmented cost function so that it works in a more general setting, as discussed in Section 7.2, where we also discuss alternative applications and implementations of free element lengths.

Below is the documentation of the class `FreeElementLengthsData`, which is used to control the free element lengths in the implemented algorithm.

FreeElementLengthsData

```
Data used to control the element lengths when they are free.

The objective function f is adjusted to penalize large element
lengths for elements with high state derivatives, resulting in the
augmented objective function \hat{f} defined as follows:

.. math::

    \hat{f} = f + c \cdot \sum_{i = 1}^{n_e} \left(h_i^a \cdot
    \int_{t_i}^{t_{i+1}} \dot{x}(t) \cdot Q \cdot
    \dot{x}(t)\,\mathrm{d}t\right).

Parameters::

    c --
        The coefficient for the newly introduced cost term.

        Type: float

    Q --
        The coefficient matrix for weighting the various state
        derivatives.

        Type: ndarray with shape (n_x, n_x)

    bounds --
        Element length bounds. The bounds are given as a tuple
        (l, u), where the bounds are used in the following way:

        .. math::
            l / n_e \leq h_i \leq u / n_e,
            \quad \forall i \in [1, n_e],

        where h_i is the normalized length of element i.

        Type: tuple
```

```
        Default: (0.7, 1.3)

    a --
        The exponent of the element length.

        Type: float
        Default: 1.
```

## 5.6 Unsupported features

In Chapters 2 and 3 we discussed certain problems and how to solve them which, for various reasons, are yet to supported by the `LocalDAECollocationAlg` algorithm. These are summarized here.

- The time interval $[t_0, t_f]$ must be fixed, i.e. $t_0$ and $t_f$ may not be free. This was discussed in Section 2.2.3.
- Continuous parameter estimation is fully supported, but no forms of discrete parameter estimation are implemented. This was discussed in Section 2.2.4.
- Since the time interval must be fixed, point constraints have not been implemented either, as the main application of point constraints is terminal constraints for minimum time problems. This was discussed in Sections 2.2.2 and 2.2.3.
- Time-variant path constraints are not supported. This was discussed in Section 2.2.2.
- Variables bounds on state derivatives are not supported. These can however be equivalently formulated as path inequality constraints, with the effect of being handled slightly differently by Ipopt. This was discussed in Section 2.2.2.
- Enforcing control variables and algebraic variables to be continuous is not supported. This was discussed in Section 3.2.1.

# Chapter 6

# Benchmarks

In this chapter we solve four optimal control problems and one parameter estimation problem using the developed algorithm and compare the results with those obtained with the old C-implemented collocation method in JModelica.org. The considered problems in this chapter are based on a VDP oscillator, a continuously stirred tank reactor, a quadruple-tank process, a distillation column and a combined cycle power plant.

## 6.1 Benchmark premises

The old collocation algorithm implemented in C in JModelica.org, which is named `CollocationLagrangePolynomials` in Python and henceforth referred to as *the old algorithm*, and the algorithm implemented in this thesis, which is named `LocalDAECollocationAlg` in Python and henceforth referred to as *the new algorithm*, are in many ways similar. The old algorithm supports Radau collocation with 1-10 collocation points. Since the purpose of this chapter is to compare the old and the new algorithm, and not to compare Gauss and Radau or local and global collocation, we will also be using Radau collocation with the same number of collocation points for the new algorithm in the benchmarks.

The two algorithms are based on the same theory and the constructed NLP problems are nearly identical, so the obtained solutions can also be expected to be nearly identical. There are however a few key differences. The old algorithm uses CppAD[1] to construct and evaluate AD graphs for the DOP functions, which are then used to construct the NLP problem. The new algorithm uses CasADi to construct AD graphs for the entire NLP problem. This leads to longer initialization times, but faster evaluation times for the new algorithm.

Since the new algorithm constructs AD graphs for the entire NLP problem, the computation of the Hessian of the Lagrangian function, which can be used by Ipopt, is

[1] http://www.coin-or.org/CppAD/

easy and efficient. Obtaining this information for the old algorithm using CppAD, although possible, would require a tremendous effort to implement, which has not been done. Thus Ipopt employs a quasi-Newton method for the old algorithm, in which the Hessian instead is approximated.

The other important difference between the old and new algorithm is how parameter estimation is performed. The old algorithm uses discrete parameter estimation and obtains the values of the variables at the measurement points by evaluating the collocation polynomials, whereas the new algorithm uses continuous parameter estimation. The measurement data is also handled differently.

When doing parameter estimation with the old algorithm, a parameter is created in Modelica for each measured value at each measurement point. These parameters are then used to create the cost function (2.16). The measurement data for the control signals are inlined as non-discrete functions, eliminating the control variables from the model. The values of the parameters belonging to the measurement data are set post-compilation by the user in Python. For large amounts of measurement data, this process creates a lot of Modelica parameters, which leads to long compilation times. When doing parameter estimation with the new algorithm, the control variables are kept in the model as free control variables, but are also treated as measured variables, which is similar to constraining the control variables using path equality constraints. The cost function (2.17) is created using the Python class `ParameterEstimationData`, documented in Section 5.2.4.16.

A final difference worth noting is how blocking factors are handled. The old algorithm introduces an NLP variable in each collocation point and then constrains them to be equal according to the provided blocking factors. The new algorithm only introduces an NLP variable for each distinct value of the control variable, i.e. for each element in the blocking factor list. How this difference affects Ipopt has not been investigated

For each benchmark we will provide the following run-time statistics:

- Compile: The CPU time it takes to compile the Modelica/Optimica code using JModelica.org's compilers and then create a model out of it. For the new algorithm, the FMUX compiler is used as discussed in Chapter 5 to create a `CasADiModel`. For the old algorithm, the JMU compiler is used to create a `JMUModel`. Since the FMUX compiler just creates an XML representation of the DOP, whereas the JMU compiler also generates and compiles the C code used for CppAD, the compilation part will in general be a lot quicker for the new algorithm.
- Initialize: The CPU time it takes to initialize the algorithm. For the new algorithm, this is where the AD graphs for the NLP problem are created using CasADi, which can be a time-consuming process for big problems. For the old algorithm, AD graphs are only created for the DOP functions, which is done during the model instantiation. This step will thus be a lot slower for the new algorithm.

- Ipopt: The CPU time spent internally in Ipopt. Since the NLP problems constructed with the old and new algorithms are nearly identical from a mathematical point of view, the main reason this time varies between the old and new algorithm is that the new algorithm computes the Lagrangian Hessian using AD, as discussed earlier. However, Ipopt is very sensitive to seemingly insignificant differences in the NLP problem, so the minor transcription differences between the old and new algorithm may in certain cases result in very different Ipopt iterations, in a manner which can be considered random, even though the same solution will be found (up to user-provided tolerances).
- Evaluate: The CPU time spent evaluating the NLP functions when solving the NLP problem in Ipopt. For the new algorithm, this is carried out by CasADi, whereas for the old algorithm, this is carried out by CppAD.
- Total: The total CPU time from the start of the compilation until the optimization result is returned. Note that this is not the sum of the rest, as there are a few additional minor computations.
- Iterations: The number of iterations required by Ipopt to solve the problem. This is related to the Ipopt statistic discussed above and varies for the same reasons.

All of the stated times are measured in seconds. Which of these times are interesting depends on the application. Roughly, it's the sum of the Ipopt and Evaluation times which are important for on-line applications, whereas for off-line applications it's the Total time that is of interest.

For each benchmark, we also provide the number of NLP variables $n_Z$, as given by (3.14), the number of equality constraints $m_e$ (none of the benchmark problems have any inequality constraints except variable bounds) and the number of non-zero elements in the Jacobian of the equality constraints (whose size is $m_e \times n_Z$). The numbers are only given for the NLP problem constructed by the new algorithm, since the corresponding numbers for the NLP problem constructed by the old algorithm are often very similar.

The Modelica and Optimica code used for the benchmarks can be found in Appendix B, with the exception of the combined cycle power plant model, as it is proprietary. For each model, we scale variables to the extent that nominal values are given in Modelica, but let Ipopt handle the equation scaling. All the benchmarks are run on a Windows 7 computer with an Intel Core i7-950 Quad processor @ 3.07 GHz and 6 GB of Crucial DDR3 BallistiX-1600 RAM. Revisions [2993] and [2194] of JModelica.org and of CasADi respectively are used, together with version 3.10.0 of Ipopt with the MA27 linear solver. Unless otherwise stated, the default options of the two algorithms are used, e.g. the meshes are equidistant and the NLP variables are initialized according to the initial guesses given in the Modelica and Optimica code.

For the new algorithm we always use pure SX graphs, for the following reasons:

- MX graphs in the current CasADi version have proven to be considerably slower for our collocation algorithms, both to construct and evaluate.
- The memory usage of the pure SX graphs is not a hindrance for the considered benchmarks.
- None of the additional generality of MX graphs is necessary for the considered benchmarks.

## 6.2 Van der Pol oscillator

The VDP oscillator was introduced in Section 1.2 and a Modelica/Optimica implementation was provided in Section 4.1. However, for this benchmark we make a slight modification by transforming the cost function from Lagrange to Mayer form, as described in Section 2.2.3, in order to increase the difficulty of the OCP. With $n_e = 100$ and $n_c = 5$, the following result is obtained.

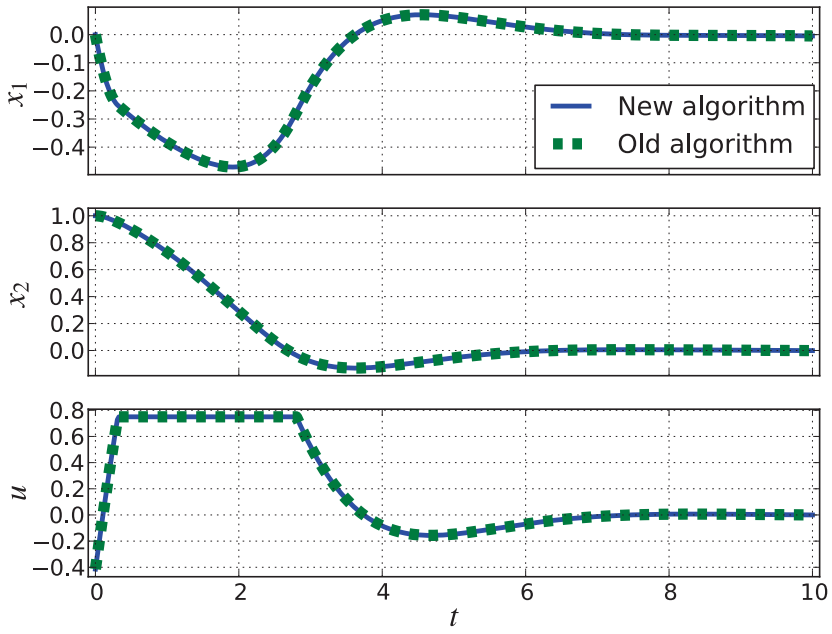


Figure 6.1: Comparison of the old and new algorithm applied to a VDP oscillator

| | Compile | Initialize | Ipopt | Evaluate | Total | Iterations |
|---|---|---|---|---|---|---|
| New algorithm | 0.1 | 0.7 | 0.2 | 0.0 | 1.1 | 24 |
| Old algorithm | 3.5 | 0.0 | 0.5 | 0.2 | 4.3 | 30 |

Table 6.1: Run-time statistics for the Van der Pol oscillator benchmark

| $n_Z$ | $m_e$ | Non-zero Jacobian elements |
|---|---|---|
| 3804 | 3304 | 16113 |

Table 6.2: NLP problem statistics for the Van der Pol oscillator benchmark

## 6.3 Continuously stirred tank reactor

The continuously stirred tank reactor (CSTR) model used for this benchmark was developed in [HR71]. The system contains a highly nonlinear exothermic reaction and has two states: reactant concentration $c$ [mol/m$^3$] and reactor temperature $T$ [K]. The rate $F_0$ [m$^3$/s], concentration $c_0$ [mol/m$^3$] and temperature $T_0$ [K] of the reactant inflow are assumed to be constant. The reactor has a liquid cooling system, whose temperature $T_c$ [K] is the sole control variable.

The dynamics of the system are modelled by

$$\begin{aligned}\dot{c}(t) &= F_0 \cdot \frac{c_0 - c(t)}{V} - k_0 \cdot e^{-\frac{E_a}{T(t)}} \cdot c(t),\\ \dot{T}(t) &= F_0 \cdot \frac{T_0 - T(t)}{V} - \frac{H}{\rho \cdot C_p} \cdot k_0 \cdot e^{-\frac{E_a}{T(t)}} \cdot c(t) + \\ &\quad \frac{2 \cdot U}{r \cdot \rho \cdot C_P} \cdot (T_c(t) - T(t)),\end{aligned}$$

where $V, k_0, E_A, H, \rho, C_p, U$ and $r$ are physical parameters and constants. The task is to find the control signal that moves the system from the stationary operation point

$$\begin{aligned}c(t_0) &\approx 956.3,\\ T(t_0) &\approx 250.1,\\ T_c(t_0) &= 370\end{aligned}$$

at $t_0 = 0$, to another stationary operation point

$$\begin{aligned}c^{\text{ref}} &\approx 338.8,\\ T^{\text{ref}} &\approx 280.1,\\ T_c^{\text{ref}} &= 280,\end{aligned}$$

while minimizing the deviation from the final stationary operation point. We thus formulate the Lagrange cost function

$$f(z) = \int_0^{t_f} \left( \left(c(t) - c^{\text{ref}}\right)^2 + \left(T(t) - T^{\text{ref}}\right)^2 + \left(T_c(t) - T_c^{\text{ref}}\right)^2 \right) \mathrm{d}t$$

and then transform it to Mayer form, for the same reason and in the same way that we did in Section 6.2. In order to avoid too high temperatures, we also impose the bounds

$$\begin{aligned} T(t) &\leq 350, \quad \forall t \in [t_0, t_f], \\ T_c(t) &\leq 370, \quad \forall t \in [t_0, t_f]. \end{aligned}$$

With $t_f = 200$ s, $n_e = 70$ and $n_c = 5$, we get the following result.

| | Compile | Initialize | Ipopt | Evaluate | Total | Iterations |
|---|---|---|---|---|---|---|
| New algorithm | 0.5 | 0.6 | 0.5 | 0.1 | 1.8 | 88 |
| Old algorithm | 4.4 | 0.0 | 0.9 | 0.9 | 6.2 | 82 |

Table 6.3: Run-time statistics for the CSTR benchmark

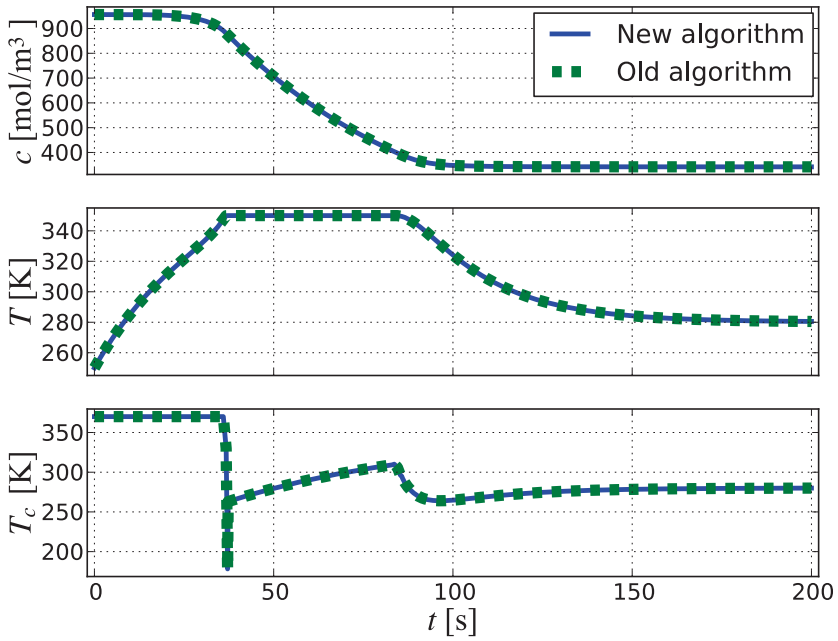


Figure 6.2: Comparison of the old and new algorithm applied to a CSTR

| $n_Z$ | $m_e$ | Non-zero Jacobian elements |
|---|---|---|
| 2664 | 2314 | 11634 |

Table 6.4: NLP problem statistics for the CSTR oscillator benchmark

### 6.3.1 CSTR remarks

In this section we discuss two aspects of the CSTR problem, which are not directly relevant to the benchmark, but nonetheless interesting.

#### 6.3.1.1 Control variable discontinuity

In Figure 6.2, we see that something peculiar happens to the control variable around $t = 37$ for both the optimization algorithms. Figure 6.3 depicts an enlarged picture of the control variable around that point in time. This kind of trajectory, where it quickly jumps down up and down for no apparent reason, is obtained when the optimal solution is discontinuous, and this discontinuity does not occur at a mesh point. The algorithm is then forced to approximate a discontinuous function with a polynomial, so we end up with something like this.

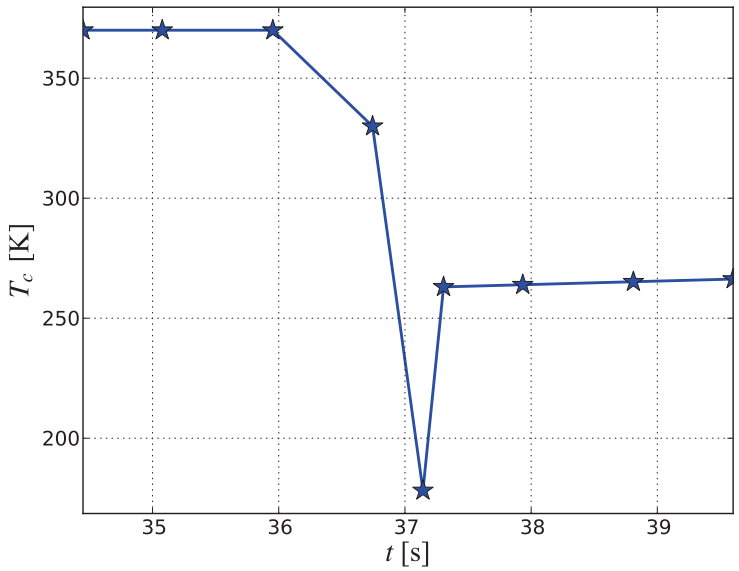


Figure 6.3: Part of the optimal control variable for the CSTR, where the collocation points are marked by stars

If this kind of behavior is a problem, which it may be depending on the application, there are two ways to solve it. The first is to locate the discontinuity a priori and then construct a mesh such that the discontinuity coincides with a mesh point. This approach is not feasible in on-line applications. The approach then is to let the element lengths be free (or at least those elements close to the discontinuity), as discussed in Section 5.5, in which case the NLP solver will find the discontinuity and make sure that it coincides with a mesh point.

#### 6.3.1.2 Optimization result verification

We have now seen some optimization results obtained with our optimization algorithm, but how accurate are they? In Section 3.2.2 we discussed the convergence orders of our methods, but unless we are successively decreasing our element lengths, we do not know how far away the approximative solution is from the limit.

This is however not the most appropriate way of verifying the optimization result. For large models, increasing the number of elements will rapidly increase the solution time. A more efficient approach is to instead simulate the model, using the control variable trajectories obtained from optimization.

We simulate the CSTR model with the control variable trajectory obtained by the new algorithm in Section 6.3 (which is virtually the same trajectory as the one obtained by the old algorithm), using the IDA solver from JModelica.org's interface to SUNDIALS. In the figure below, we compare the simulation and optimization result.

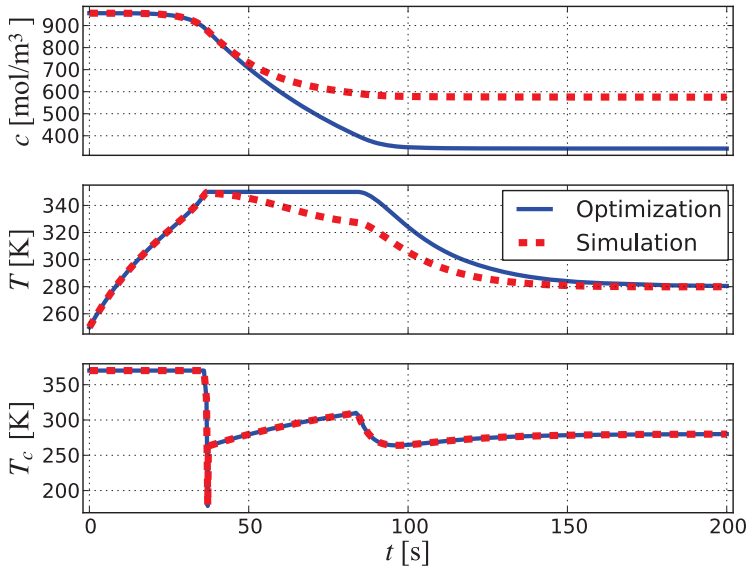


Figure 6.4: Comparison between CSTR optimization with $n_e = 70$ and simulation

The optimization result certainly leaves something to be desired in terms of accuracy. This kind of discrepancy can be expected for systems as nonlinear as this CSTR. However, since we are using Radau collocation with 5 collocation points, and thus have a method of order 9, we can expect quite the improvement if we halve all the element lengths. We proceed to do so, and the result is shown below. The error is no longer visible to the naked eye, and so we are satisfied.

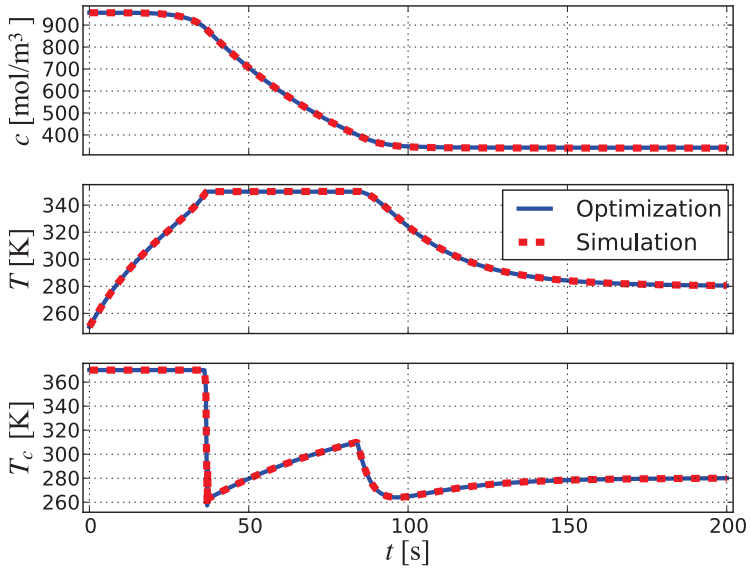


Figure 6.5: Comparison between CSTR optimization with $n_e = 140$ and simulation

## 6.4 Quadruple-tank process

The quadruple-tank process model used in this benchmark was developed in [Joh00]. The process consists of four tanks, each having an inlet flow of water regulated by two pumps and a water outlet at the bottom. The model has four states, which are the water heights $x_i$ [m] in each of the tanks, two control variables, which are the pump voltages $u_1$ [V] and $u_2$ [V], and four free parameters, which are the water outlet areas $a_i$ [m$^2$] in each of the tanks. The dynamics of the system are modelled by

$$\begin{aligned}
\dot{x}_1 &= -\frac{a_1}{A_1} \cdot \sqrt{2 \cdot g \cdot x_1} + \frac{\gamma_1 \cdot k_1}{A_1} \cdot u_1 + \frac{a_3}{A_1} \cdot \sqrt{2 \cdot g \cdot x_3}, \\
\dot{x}_2 &= -\frac{a_2}{A_2} \cdot \sqrt{2 \cdot g \cdot x_2} + \frac{\gamma_2 \cdot k_2}{A_2} \cdot u_2 + \frac{a_4}{A_2} \cdot \sqrt{2 \cdot g \cdot x_4}, \\
\dot{x}_3 &= -\frac{a_3}{A_3} \cdot \sqrt{2 \cdot g \cdot x_3} + \frac{(1-\gamma_2) \cdot k_2}{A_3} \cdot u_2, \\
\dot{x}_3 &= -\frac{a_4}{A_4} \cdot \sqrt{2 \cdot g \cdot x_4} + \frac{(1-\gamma_1) \cdot k_1}{A_4} \cdot u_1,
\end{aligned}$$

where $A_1$, $A_2$, $A_3$, $A_4$, $\gamma_1$, $\gamma_2$, $k_1$, $k_2$ and $g$ are physical parameters and constants.

The task is to, given some measurement data from the real system, determine the values of the free parameters $a_1$, $a_2$, $a_3$ and $a_4$ which minimize the discrepancy between the model behavior and the measurement data. The used measurement data span 60 seconds with a sampling frequency of 2 Hz, giving us 121 measurement points. The control variables used during the measurements can be seen in the optimization result in Figure 6.6.

The used weighting matrix $Q$ is a diagonal matrix, where each of the parameter weights is 1. The new algorithm also needs weights for the two control variables, as they are treated as measured variables. We choose 5 as the weight for both the control variables. With $n_e = 100$ and $n_c = 5$, the following result is obtained.

| | $a_1$ [mm$^2$] | $a_2$ [mm$^2$] | $a_3$ [mm$^2$] | $a_4$ [mm$^2$] |
|---|---|---|---|---|
| New algorithm | 2.6590 | 2.7056 | 3.0065 | 2.9348 |
| Old algorithm | 2.6592 | 2.7057 | 3.0067 | 2.9346 |

Table 6.5: Estimated parameter values for the quadruple-tank process

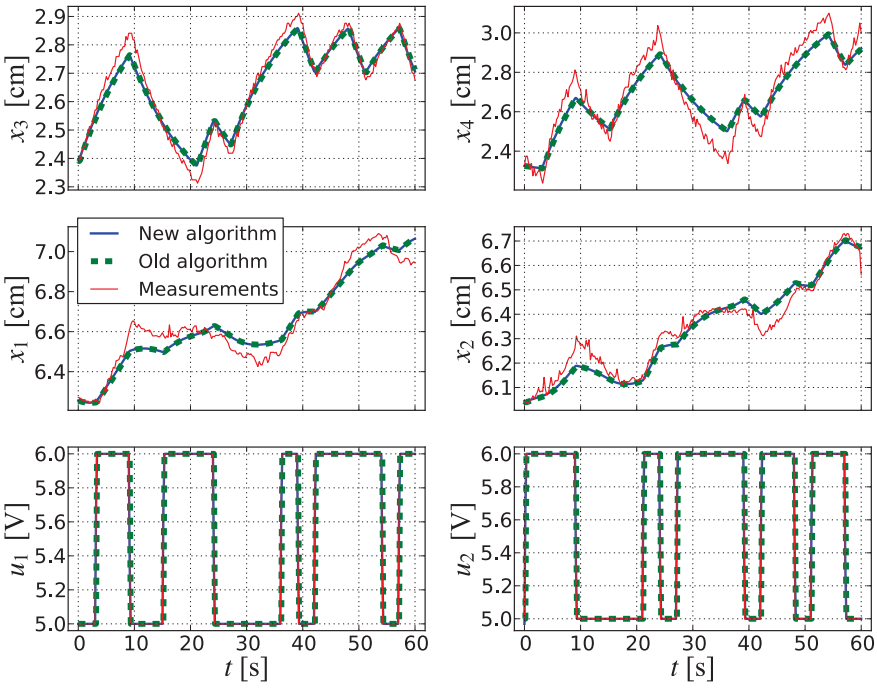


Figure 6.6: Comparison of the old and new algorithm applied to the quadruple-tank process

| | Compile | Initialize | Ipopt | Evaluate | Total | Iterations |
|---|---|---|---|---|---|---|
| New algorithm | 0.3 | 9.3 | 0.3 | 0.1 | 10.0 | 20 |
| Old algorithm | 44.2 | 0.1 | 1.2 | 9.9 | 55.4 | 10 |

Table 6.6: Run-time statistics for the quadruple-tank process benchmark, using 121 measurement points

| $n_Z$ | $m_e$ | Non-zero Jacobian elements |
|---|---|---|
| 5410 | 4406 | 24828 |

Table 6.7: NLP problem statistics for the quadruple-tank process benchmark

The compilation and evaluation times of the old algorithm is highly dependent on the amount of measurement data, whereas the execution times of the new algorithm are largely unaffected. The additional measurement points only affect the new algorithm during the interpolation of the measurement data, which is a relatively quick procedure. Below is a table of statistics showing how the execution times vary between the old and new algorithm as the number of measurement points vary.

| | Compile | Evaluate | Total |
|---|---|---|---|
| New algorithm, $n_y = 31$ | 0.3 | 0.1 | 10.0 |
| Old algorithm, $n_y = 31$ | 9.2 | 2.2 | 12.0 |
| New algorithm, $n_y = 61$ | 0.3 | 0.1 | 10.0 |
| Old algorithm, $n_y = 61$ | 17.8 | 3.9 | 22.5 |
| New algorithm, $n_y = 121$ | 0.3 | 0.1 | 10.0 |
| Old algorithm, $n_y = 121$ | 44.2 | 9.9 | 55.4 |
| New algorithm, $n_y = 241$ | 0.4 | 0.1 | 10.1 |
| Old algorithm, $n_y = 241$ | 158.8 | 41.0 | 203.6 |

Table 6.8: Select execution times for the quadruple-tank process benchmark with a varying amount of measurement points

## 6.5 Distillation column

The distillation column model used for this benchmark is described in [HE02]. The column has 32 trays, indexed from top to bottom, and the distillate has two chemical components. The feed stream is introduced in tray 17. The mole fraction in the vapor of the first component in tray $i$ is $y_i$ [1] and is an algebraic variable. The mole fraction in the liquid of the first component in tray $i$ is $x_i$ [1] and is a state. Additional algebraic variables are the liquid flowrate in the rectification section $F_R$

[mol/s], the vapor flowrate in the column $V$ [mol/s] and the liquid flowrate in the stripping section $F_S$ [mol/s]. The control variable is the reflux ratio $u$ [1]. We thus have a total of 32 states, 35 algebraic variables and 1 control variable.

The dynamics of the system are modelled by

$$\begin{aligned}
\dot{x}_1 &= \frac{V \cdot (y_2 - x_1)}{A_C}, \\
\dot{x}_i &= \frac{F_R \cdot (x_{i-1} - x_i) - V \cdot (y_i - y_{i+1})}{A_T}, \quad \forall i \in [2, 16], \\
\dot{x}_{17} &= \frac{D + F_R \cdot x_{16} - F_S \cdot x_{17} - V \cdot (y_{17} - y_{18})}{A_T}, \\
\dot{x}_i &= \frac{F_S \cdot (x_{i-1} - x_i) - V \cdot (y_i - y_{i+1})}{A_T}, \quad \forall i \in [18, 31], \\
\dot{x}_{32} &= \frac{F_S \cdot x_{31} - (F - D) \cdot x_{32} - V \cdot y_{32}}{A_R}, \\
y_i &= \frac{\alpha \cdot x_i}{1 + (\alpha - 1) \cdot x_i}, \quad \forall i \in [1, 32], \\
F_R &= D \cdot u, \\
V &= D + F_R, \\
F_S &= F + F_R,
\end{aligned}$$

where $A_C$, $A_T$, $A_R$, $D$, $F$ and $\alpha$ are physical parameters and constants.

The task is to drive the system from the stationary point at $u = 3$ to the stationary point at $u = 2$, which is done by penalizing the deviation of $u$ and any one of the states or algebraic variables from their values at the second stationary point. We choose to penalize the deviation of $y_1$ from $y_1^{ref} \approx 0.896$. We thus construct the Lagrange cost function

$$f(z) = \int_{t_0}^{t_f} \left( 1000 \cdot \left( y_1(t) - y_1^{\mathrm{ref}} \right)^2 + (u(t) - 2)^2 \right) \mathrm{d}t,$$

and then transform it to Mayer form for the same reason and in the same way that we did in Section 6.2. The reflux ratio is not allowed to go below 1, so we also impose the bound

$$u(t) \geq 1, \quad \forall t \in [t_0, t_f].$$

We also impose blocking factors on the reflux ratio, forcing it to change only every 2 seconds.

With $t_0 = 0$ s, $t_f = 50$ s, $n_e = 100$ and $n_c = 4$, the following result is obtained.

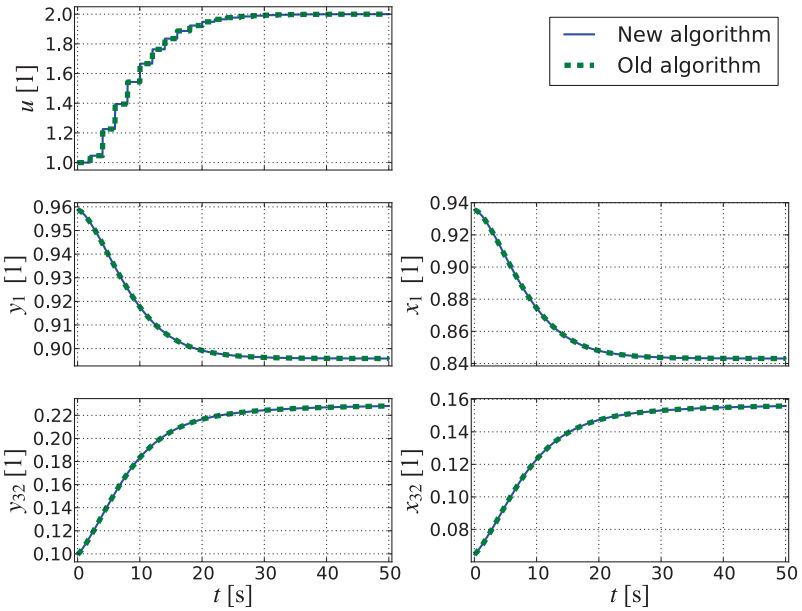


Figure 6.7: Comparison of the old and new algorithm applied to a binary distillation column

| | Compile | Initialize | Ipopt | Evaluate | Total | Iterations |
|---|---|---|---|---|---|---|
| New algorithm | 3.4 | 12.7 | 9.2 | 0.5 | 26.0 | 29 |
| Old algorithm | 13.4 | 0.2 | 8.0 | 3.0 | 24.9 | 19 |

Table 6.9: Run-time statistics for the distillation column benchmark

| $n_Z$ | $m_e$ | Non-zero Jacobian elements |
|---|---|---|
| 43793 | 43769 | 203662 |

Table 6.10: NLP problem statistics for the distillation column benchmark

A noteworthy and unexpected result is that the old algorithm evaluates its NLP functions almost as quickly as the new algorithm in this case, even though the problem is fairly large (which is what causes the long initialization time for the new algorithm). So for off-line applications, this is a case where the old algorithm actually is more efficient than the new algorithm, especially if the Compile time is unimportant. The reason for the quick evaluation by the old algorithm has not been investigated.

## 6.6 Combined cycle power plant

The combined cycle power plant (CCPP) model used for this benchmark is described in [CDk11]. The model has 9 states, 128 algebraic variables and 1 control variable. The task is to minimize the time required to start up the power plant. The startup process is considered finished when the normalized load input signal $u$ [1] to the steam turbine, starting at 15 %, has reached 100 % and the evaporator pressure $p$ [Pa], which is a state with an initial value of approximately 3.47 MPa, has reached approximately 8.35 MPa.

In order to reduce the wear and tear on the steam turbine, which is one of the most expensive parts of the power plant, the thermal stress in the turbine $\sigma$ [Pa], which is an algebraic variable, may not exceed 260 MPa. This is the main limiting factor in the startup process. Another imposed constraint is that the derivative of the load input signal $u$ may not be negative and may not exceed $0.1/60$ s$^{-1}$. Since some of the bounds are on the derivative of the control variable, which is not supported by neither the old nor the new algorithm, we introduce the control variable $\dot{u}$ and add the equation

$$\frac{\mathrm{d}u}{\mathrm{d}t} = \dot{u},$$

to the DAE system. This converts the previous control variable $u$ into a state, giving us a total of 10 states, and the sole control variable is now instead $\dot{u}$, which we can impose the mentioned bounds on.

We formulate a Lagrange cost function which penalizes the deviation of the load input signal and the evaporator pressure from their respectively desired values, given by

$$f(z) = \int_{t_0}^{t_f} \left(10^{-12} \cdot \left(p(t) - 8.35 \cdot 10^6\right)^2 + 0.5 \cdot \left(u(t) - 1\right)^2\right) \mathrm{d}t.$$

All the NLP variables, except those corresponding to $\dot{u}$, are initialized based on a simulation of the startup process with

$$u(t) = 0.15 + 0.85 \cdot \frac{t}{T \cdot \left(1 + \left(\frac{t}{T}\right)^6\right)^{\frac{1}{6}}},$$

where $T = 10000$ s is the simulation duration. With $t_0 = 0$ s, $t_f = 4000$ s, $n_e = 40$ and $n_c = 4$, the following optimization result is obtained.

| | Compile | Initialize | Ipopt | Evaluate | Total | Iterations |
|---|---|---|---|---|---|---|
| New algorithm | 4.2 | 4.2 | 4.0 | 0.8 | 13.4 | 79 |
| Old algorithm | 22.0 | 0.1 | 6.0 | 46.3 | 74.5 | 69 |

Table 6.11: Run-time statistics for the CCPP benchmark

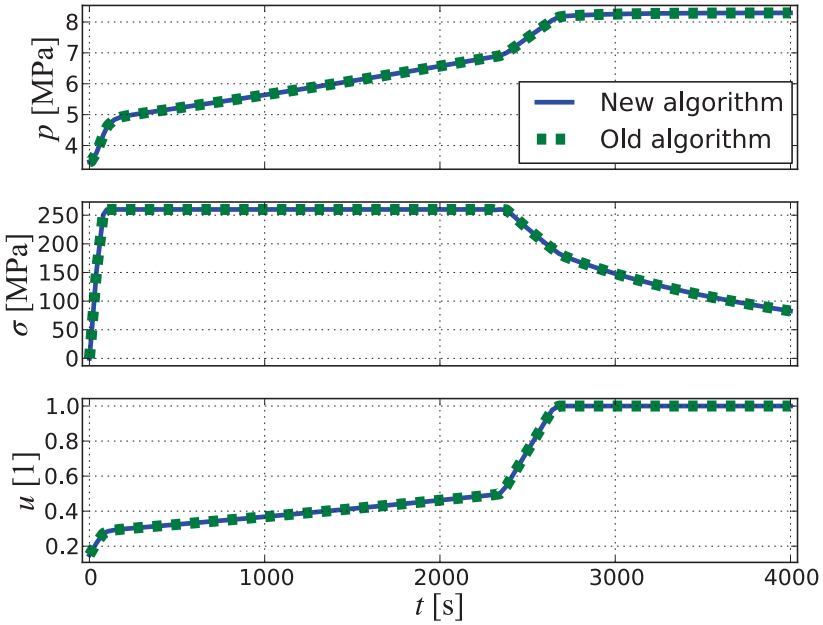


Figure 6.8: Comparison of the old and new algorithm applied to a CCPP

| $n_Z$ | $m_e$ | Non-zero Jacobian elements |
|---|---|---|
| 24379 | 24219 | 74151 |

Table 6.12: NLP problem statistics for the CCPP benchmark

In this case we clearly see the benefits of constructing AD graphs for the entire NLP problem using CasADi for large-scale problems, which is what allows for the exceptionally quick NLP function evaluations.

# Chapter 7

# Concluding remarks

## 7.1 Conclusions

We have successfully derived and implemented an optimization algorithm based on Radau and Gauss collocation and discussed why Lobatto collocation requires a different approach than the one employed in this thesis. The algorithm has been compared to an old algorithm in JModelica.org, which employs Radau collocation. The solutions found by the two algorithms have shown to be as identical as can be expected, i.e. up to Ipopt tolerances.

The performance of the new algorithm compared to the old algorithm, in terms of speed, has varied slightly, but the new algorithm usually outperforms the old one. During the NLP solution, the new algorithm clearly evaluates its functions faster, and thanks to the possibility provided by the new algorithm to compute the Hessian of the Lagrangian, the time spent internally in Ipopt is also most often shorter. The new algorithm is thus clearly superior for on-line applications.

However, the time required to initialize the new algorithm is considerably longer than the one required by the old algorithm. So in an off-line application where the model is only compiled and instantiated a few times, but each model instance is optimized several times but with different parameter values, such as variable bounds, it is not clear which of the two algorithms is superior. For such an application, the old algorithm was about twice as efficient for the distillation column, whereas the new algorithm was about six times as efficient for the CCPP.

In off-line applications where you often recompile the model between each optimization, the fast compilation provided by the FMUX compiler becomes useful. For such applications the new algorithm should be superior in almost every case, although the distillation column is an example of when the old algorithm actually is slightly faster.

Since the old and new algorithm construct nearly the same NLP problem, they should have a similar amount of robustness. The robustness of the algorithms is directly related to the robustness of Ipopt. Depending on the problem, Ipopt may be very

sensitive to scaling and initial guesses. Finding good nominal values and initial guesses may require a lot of work from the user, but such is the nature of large-scale optimization.

CasADi is a tool still under heavy development, so the overall performance of the new algorithm can be expected to improve even further in the future.

## 7.2 Future work

In terms of being fully-featured, there are still a few important features missing for the new algorithm. CasADi combined with Python is however very flexible, so adding new collocation-related features is often straightforward, which is not the case for the old algorithm implemented in C. In Section 5.6 we listed some missing features. Some other important missing features are related to what we discussed in 5.5. The optimization of element lengths can be improved by analyzing the actual discretization error and instead use this to augment the cost function. An alternative approach is to instead constrain the discretization error to be smaller than some tolerance. This is discussed in [Bie10, ch. 10].

An altogether different approach to finding an optimal mesh is called *mesh refinement*. This approach instead solves the NLP problem with a fixed element mesh, assesses the discretization error, constructs a new mesh based on this and then iteratively repeats this process. The idea behind this approach is that solving the problem repeatedly with a small mesh can be both faster and more accurate than solving it only once using a large equidistant grid. Mesh refinement is discussed in [Bet10, ch. 4.7].

When applying collocation methods to DAE systems, the discretization error analysis required for these approaches is made more difficult. This is because collocation methods often suffer from order reduction when applied to DAE systems, and the reduction depends on the index of the DAE system, which is generally not known. However, since JModelica.org transforms the DAE system into an index one (or zero) system, and the orders of our methods are well-known in these cases, as discussed in Section 3.2.2, the algorithms implemented in this thesis should be well-suited for being combined with either a mesh refinement algorithm or an improved formulation with free element lengths.

An application of having free element lengths other than improving the solution accuracy, is the introduction of *phases*, as discussed in [Bet10, ch. 3]. In a single phase the two following requirements are usually made:

1. The DAE system must remain the same.
2. The states must be continuous.

By allowing multiple phases, more complex systems can be modeled and optimized. The phase boundaries are regulated by *events*, which are either *time events* or *state*

*events*. Time events trigger a phase change at a specific point in time. Since these time points are known a priori, there is no need to have free element lengths to handle these if an appropriate mesh has been chosen. State events however trigger a phase change when the states (or algebraic variables) satisfy some conditions. Since it is generally not known a priori at what points in time these events occur, at least some of the element lengths need to be free in order to handle this.

More future work includes modifying the Gauss collocation implementation into a projected Runge-Kutta method to improve its stability and accuracy properties, as discussed in Section 3.4. It would also be interesting to implement collocation methods based on integration rather than differentiation, as discussed in Appendix A. This would also allow the implementation of a Lobatto collocation method.

Another topic the implemented algorithm can be used for is investigating how certain transcription details affect the performance. For example, how does the elimination of derivative variables affect the solution of the NLP problem, and how do evaluation constraints compare to quadrature constraints? Does the choice of Radau versus Gauss collocation points affect overall Ipopt robustness?

# Appendix A

# Collocation methods as Runge-Kutta methods

In this appendix we present two different derivations of Runge-Kutta methods based on collocation. We start by describing the problem to be solved and briefly introduce Runge-Kutta methods. Next we present the first derivation, which uses the Lagrange form of the collocation polynomial's derivative and then integrates it to obtain the collocation polynomial. The second derivation uses the Lagrange form of the collocation polynomial itself as the starting point and then differentiates it to get the collocation equations. The second approach is the one used in this thesis, as discussed in Chapter 3, and we show why this approach can not be used to construct a Lobatto collocation method.

Collocation-related notation from previous chapters is not used unless explicitly reintroduced.

## A.1 Introduction

In this Appendix we only consider scalar-valued ODEs on the form

$$\dot{x}(t) = f(t, x(t)).$$

The generalization from scalar-valued ODEs to ODE systems is trivial and skipped for ease of notation. The generalization from ODE systems to DAE systems is nontrivial, as discussed in Chapter 2 and 3. The ODE case is however sufficient for the purposes of this appendix.

The goal is to take a single integration step from $t_n$ to $t_{n+1} = t_n + h$, where $h$ is the step size, given the value $x_n = x(t_n)$, thus finding the next value $x_{n+1}$. The value of $x_n$ is either given by the initial values or the previous integration step. An $s$-stage

Runge-Kutta method doing this has the general form

$$\begin{aligned} X_i &= x_n + h \cdot \sum_{j=1}^{s} a_{i,j} \cdot \dot{X}_i, \quad \forall i \in [1..s], \\ x_{n+1} &= x_n + h \cdot \sum_{i=1}^{s} b_i \cdot \dot{X}_i, \end{aligned} \tag{A.1}$$

where $X_i$ are called the *stage values*,

$$\dot{X}_i := f(t_n + c_i \cdot h, X_i)$$

are called the *stage derivatives* and the coefficients

$$A = [a_{i,j}] \in \mathbb{R}^{s \times s}, \qquad b = [b_i] \in \mathbb{R}^s, \qquad c = [c_i] \in \mathbb{R}^s$$

define the Runge-Kutta method. Runge-Kutta methods are conveniently represented by their respective *Butcher tableaus*, given by

$$\begin{array}{c|c} c & A \\ \hline & b \end{array}.$$

A collocation method is constructed by finding the collocation polynomial $u$ of degree $s$ that satisfies the initial condition at the start of the integration step and the ODE at the collocation points, i.e.

$$u(t_n) = x_n, \tag{A.2}$$

$$\dot{u}(t_n + c_i \cdot h) = f(t_n + c_i \cdot h, X_i), \quad \forall i \in [1..s], \tag{A.3}$$

where $c_i$ are the collocation points. The value $x_{n+1}$ is then found by evaluating the collocation polynomial at the mesh point, i.e.

$$x_{n+1} := u(t_{n+1}). \tag{A.4}$$

Note that the coefficients $c$ of the Runge-Kutta method are the collocation points chosen for the collocation method. By defining

$$X_i := u(t_n + c_i \cdot h) \tag{A.5}$$

and noting that the stage derivatives are given by (A.3), a Runge-Kutta method is obtained, which we show in two ways.

## A.2 Derivation by integration

Let $\tau \in [0, 1]$ denote the localized and normalized time in an integration step. The Lagrange form of $\dot{u}$ is given by

$$\dot{u}(t_n + \tau \cdot h) = \sum_{i=1}^{s} \dot{u}(t_n + c_i \cdot h) \cdot \ell_i(\tau), \tag{A.6}$$

where $\ell_i$ is the Lagrange basis polynomial given by

$$\ell_i(\tau) = \prod_{j \in [1..s] \setminus \{i\}} \frac{\tau - c_j}{c_i - c_j}.$$

Applying (A.3) to (A.6) and integrating yields

$$\int_0^{c_i} \dot{u}(t_n + \tau \cdot h)\,\mathrm{d}\tau = \int_0^{c_i} \sum_{j=1}^{s} \dot{X}_j \cdot \ell_j(\tau)\,\mathrm{d}\tau, \quad \forall i \in [1..s].$$

Applying (A.2) and some calculus gives

$$\frac{u(t_n + c_i \cdot h) - x_n}{h} = \sum_{j=1}^{s} \dot{X}_j \cdot \int_0^{c_i} \ell_j(\tau)\,\mathrm{d}\tau, \quad \forall i \in [1..s].$$

By defining

$$a_{i,j} = \int_0^{c_i} \ell_j(\tau)\,\mathrm{d}\tau. \tag{A.7}$$

and applying (A.5), we obtain

$$X_i = x_n + h \cdot \sum_{j=1}^{s} a_{i,j} \cdot \dot{X}_j, \quad \forall i \in [1..s]. \tag{A.8}$$

By setting $\tau = 1$ in (A.6) and repeating the above procedure, we obtain

$$u(t_n + h) = x_n + h \cdot \sum_{i=1}^{s} b_i \cdot \dot{X}_i, \tag{A.9}$$

where

$$b_i = \int_0^1 \ell_i(\tau)\,\mathrm{d}\tau.$$

By (A.4), (A.8) together with (A.9) gives us a Runge-Kutta method on the form given by (A.1).

## A.3 Derivation by differentiation

The Lagrange form of $u$ is given by

$$u(t_n + \tau \cdot h) = \sum_{i=0}^{s} u(t_n + c_i \cdot h) \cdot \tilde{\ell}_i(\tau), \tag{A.10}$$

where $c_0 = 0$ and $\tilde{\ell}_i$ is the Lagrange basis polynomial given by

$$\tilde{\ell}_i(\tau) = \prod_{j \in [0..s] \setminus \{i\}} \frac{\tau - c_j}{c_i - c_j}.$$

Note that $c_0$ is not a collocation point.

Differentiating (A.10) with respect to $\tau$ gives

$$h \cdot \dot{u}(t_n + \tau \cdot h) = \sum_{i=0}^{s} u(t_n + c_i \cdot h) \cdot \dot{\tilde{\ell}}_i(\tau).$$

At the collocation points, this equation together with (A.3) gives us the collocation equations

$$h \cdot \dot{X}_i = \sum_{j=0}^{s} u(t_n + c_j \cdot h) \cdot \dot{\tilde{\ell}}_j(c_i), \quad \forall i \in [1..s].$$

A slight reformulation using (A.5) gives us

$$\sum_{j=0}^{s} \dot{\tilde{\ell}}_j(c_i) \cdot X_j = h \cdot \dot{X}_i, \quad \forall i \in [1..s].$$

Together with (A.2), this gives us the equation system

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ \dot{\tilde{\ell}}_0(c_1) & \dot{\tilde{\ell}}_1(c_1) & \cdots & \dot{\tilde{\ell}}_s(c_1) \\ \vdots & \vdots & \ddots & \vdots \\ \dot{\tilde{\ell}}_0(c_s) & \dot{\tilde{\ell}}_1(c_s) & \cdots & \dot{\tilde{\ell}}_s(c_s) \end{bmatrix} \cdot \begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ X_s \end{bmatrix} = \begin{bmatrix} x_n \\ h \cdot \dot{X}_1 \\ \vdots \\ h \cdot \dot{X}_s \end{bmatrix}.$$

Solving the first trivial equation

$$X_0 = x_n \tag{A.11}$$

gives us

$$\underbrace{\begin{bmatrix} \dot{\tilde{\ell}}_1(c_1) & \dot{\tilde{\ell}}_2(c_1) & \cdots & \dot{\tilde{\ell}}_s(c_1) \\ \dot{\tilde{\ell}}_1(c_2) & \dot{\tilde{\ell}}_2(c_2) & \cdots & \dot{\tilde{\ell}}_s(c_2) \\ \vdots & \vdots & \ddots & \vdots \\ \dot{\tilde{\ell}}_1(c_s) & \dot{\tilde{\ell}}_2(c_s) & \cdots & \dot{\tilde{\ell}}_s(c_s) \end{bmatrix}}_{L} \cdot \underbrace{\begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_s \end{bmatrix}}_{X} = h \cdot \underbrace{\begin{bmatrix} \dot{X}_1 \\ \dot{X}_2 \\ \vdots \\ \dot{X}_s \end{bmatrix}}_{\dot{X}} - x_n \cdot \underbrace{\begin{bmatrix} \dot{\tilde{\ell}}_0(c_1) \\ \dot{\tilde{\ell}}_0(c_2) \\ \vdots \\ \dot{\tilde{\ell}}_0(c_s) \end{bmatrix}}_{Q}.$$

If $L$ is non-singular, we get

$$X = -x_n \cdot L^{-1} \cdot Q + h \cdot L^{-1} \cdot \dot{X}. \tag{A.12}$$

If

$$L^{-1} \cdot Q = (-1, -1, \ldots, -1) =: -U, \tag{A.13}$$

we get the equations defining the stage values for a Runge-Kutta method, where the matrix $A$ is given by $L^{-1}$.

The coefficients $b$ are found by evaluating

$$x_{n+1} = u(t_n + h) = \sum_{i=0}^{s} \tilde{\ell}_i(1) \cdot X_i.$$

Equation (A.11) yields that

$$x_{n+1} = x_n \cdot \tilde{\ell}_0(1) + \sum_{i=1}^{s} \tilde{\ell}_i(1) \cdot X_i.$$

Under the assumption that $L$ is non-singular and that (A.13) holds, (A.12) gives us

$$\begin{aligned} x_{n+1} &= x_n \cdot \tilde{\ell}_0(1) + \sum_{i=1}^{s} \tilde{\ell}_i(1) \cdot \left( x_n + h \cdot L_i^{-1} \cdot \dot{X} \right) \\ &= x_n \cdot \sum_{i=0}^{s} \tilde{\ell}_i(1) + \sum_{i=1}^{s} \tilde{\ell}_i(1) \cdot h \cdot L_i^{-1} \cdot \dot{X}, \end{aligned}$$

where $L_i^{-1}$ is the $i$:th row of $L^{-1}$. Since

$$\sum_{i=0}^{s} \tilde{\ell}_i = 1,$$

which we do not show, we get

$$x_{n+1} = x_n + h \cdot \sum_{i=1}^{s} \tilde{\ell}_i(1) \cdot L_i^{-1} \cdot \dot{X} = x_n + h \cdot \sum_{i=1}^{s} \tilde{\ell}_i(1) \cdot \sum_{j=1}^{s} L_{i,j}^{-1} \cdot \dot{X}_j$$

By noting that

$$b_i = \sum_{j=1}^{s} \tilde{\ell}_j(1) \cdot L_{j,i}^{-1},$$

we get

$$x_{n+1} = x_n + h \cdot \sum_{i=1}^{s} b_i \cdot \dot{X}_i.$$

Together with (A.12), we have thus obtained a Runge-Kutta method. In order to find explicit values for $A$ and $b$, $L$ needs to be inverted, which is not something we do for the general case.

## A.4 Conclusions

Determining the singularity of $L$ in the general case (for an arbitrary choice of collocation points) is beyond the scope of this thesis. However, experiments (not documented in this report) for specific choices of $c$ indicate that $L$ is non-singular if and only if $c_1 \neq 0$, which is the case for Radau and Gauss, but not Lobatto. It is actually easy to realize that derivation by differentiation as described in Section A.3 breaks down for $c_1 = 0$, since we introduce $c_0 = 0$ as an interpolation point for the collocation polynomial, and the construction of a Lagrange interpolation polynomial is dependent on all of the interpolation points being distinct.

Experiments also indicate that $L^{-1} = A$, where $A$ is the coefficient matrix derived in A.2, in the case that $L$ is non-singular. Assumption (A.13) also seems to hold if $L$ is non-singular. This means that the two methods derived by either integration or differentiation are in fact equivalent. This can be motivated by how derivation by integration constructs the equation system

$$X = x_n \cdot U + h \cdot A \cdot \dot{X},$$

whereas derivation by differentiation constructs the equation system

$$L \cdot X = -x_n \cdot Q + h \cdot \dot{X}$$

and then solves it for $X$ to get the exact same structure as derivation by integration. Once again, determining the singularity of $L^{-1} = A$ is non-trivial in the general case. It is, however, trivial to realize that $A$ is singular in the case of $c_1 = 0$. From (A.7), we get that the first row of $A$ is given by

$$A_1 = \left( \int_0^0 \ell_1(\tau)\,\mathrm{d}\tau, \int_0^0 \ell_2(\tau)\,\mathrm{d}\tau, \ldots, \int_0^0 \ell_s(\tau)\,\mathrm{d}\tau \right) = (0, 0, \ldots, 0),$$

and thus $A$ is singular, further showing that derivation by differentiation does not work for $c_1 = 0$.

So why bother with derivation by differentiation? As it turns out, an implementation based on derivation by differentiation has better numerical properties with regards to round-off and iteration errors, as discussed in [HW96, ch. IV.8].

# Appendix B

# Benchmark models

In this chapter we provide the Modelica and Optimica source code used for the benchmarks in Chapter 6, with the exception of the combined cycle power plant model, which is proprietary.

## B.1 Van der Pol oscillator

```
model VDP

  Real x1(start=0, fixed=true);
  Real x2(start=1, fixed=true);

  input Real u;

equation

  der(x1) = (1 - x2^2) * x1 - x2 + u;
  der(x2) = x1;

end VDP;

optimization VDP_OCP(objective=cost(finalTime), startTime=0, finalTime=10)

  Real cost(start=0, fixed=true);

  extends VDP(u(free=true, max=0.75));

equation

  der(cost) = x1^2 + x2^2 + u^2;

end VDP_OCP;
```

## B.2 Continuously stirred tank reactor

```
model CSTR

  parameter Modelica.SIunits.VolumeFlowRate F0 = 100/1000/60 "Inflow";
  parameter Modelica.SIunits.Concentration c0 =
      1000 "Concentration of inflow";
  parameter Modelica.SIunits.Temp_K T0 = 350;
  parameter Modelica.SIunits.Length r = 0.219;
  parameter Real k0 = 7.2e10/60;
  parameter Real EdivR = 8750;
  parameter Real U = 915.6;
  parameter Real rho = 1000;
  parameter Real Cp = 0.239*1000;
  parameter Real dH = -5e4;
  parameter Modelica.SIunits.Volume V = 100 "Reactor Volume";
  parameter Modelica.SIunits.Concentration c_init = 956.271065;
  parameter Modelica.SIunits.Temp_K T_init = 250.051971;

  Real c(start=c_init, fixed=true, nominal=c0);
  Real T(start=T_init, fixed=true, nominal=T0);

  Modelica.Blocks.Interfaces.RealInput Tc "Cooling temperature";

equation

  der(c) = F0 * (c0 - c) / V - k0 * c * exp(-EdivR / T);
  der(T) = F0 * (T0 - T) / V -
           dH / (rho * Cp) * k0 * c * exp(-EdivR / T) +
           2 * U / (r * rho * Cp) * (Tc - T);

end CSTR;

optimization CSTR_OCP(objective=cost(finalTime), startTime=0, finalTime=200)

  CSTR cstr(c(initialGuess=600), T(initialGuess=300, max=350));
  input Real u(initialGuess=320, min=100, max=370) = cstr.Tc;

  Real cost(start=0, fixed=true, nominal=1e7);
  parameter Real c_ref = 338.775766;
  parameter Real T_ref = 280.099198;
  parameter Real Tc_ref = 280;

equation

  der(cost) = (cstr.c - c_ref)^2 + (cstr.T - T_ref)^2 +
              (cstr.Tc - Tc_ref)^2;

end CSTR_OCP;
```

## B.3 Quadruple-tank process

```
model QuadTank

  // Process parameters
  parameter Modelica.SIunits.Area A1 = 4.9e-4, A2 = 4.9e-4,
                                  A3 = 4.9e-4, A4 = 4.9e-4;
  parameter Modelica.SIunits.Area a1(start=0.03e-4), a2(start=0.03e-4),
                                  a3(start=0.03e-4), a4(start=0.03e-4);
  parameter Modelica.SIunits.Acceleration g = 9.81;
  parameter Real k1_nmp(unit="m^3/s/V") = 0.56e-6,
                 k2_nmp(unit="m^3/s/V") = 0.56e-6;
  parameter Real g1_nmp = 0.30, g2_nmp = 0.30;

  // Initial tank levels
  parameter Modelica.SIunits.Length x1_0 = 0.06255;
  parameter Modelica.SIunits.Length x2_0 = 0.06045;
  parameter Modelica.SIunits.Length x3_0 = 0.02395;
  parameter Modelica.SIunits.Length x4_0 = 0.02325;

  // Tank levels
  Modelica.SIunits.Length x1(fixed=true, start=x1_0, min=0.0001);
  Modelica.SIunits.Length x2(fixed=true, start=x2_0, min=0.0001);
  Modelica.SIunits.Length x3(fixed=true, start=x3_0, min=0.0001);
  Modelica.SIunits.Length x4(fixed=true, start=x4_0, min=0.0001);

  // Inputs
  input Modelica.SIunits.Voltage u1;
  input Modelica.SIunits.Voltage u2;

equation

  der(x1) = -a1/A1*sqrt(2*g*x1) + a3/A1*sqrt(2*g*x3) + g1_nmp*k1_nmp/A1*u1;
  der(x2) = -a2/A2*sqrt(2*g*x2) + a4/A2*sqrt(2*g*x4) + g2_nmp*k2_nmp/A2*u2;
  der(x3) = -a3/A3*sqrt(2*g*x3) + (1-g2_nmp)*k2_nmp/A3*u2;
  der(x4) = -a4/A4*sqrt(2*g*x4) + (1-g1_nmp)*k1_nmp/A4*u1;

end QuadTank;

model PRBS1

  Modelica.Blocks.Interfaces.RealOutput y;
  parameter Integer N = 10;
  parameter Real ts[N] =
      {0., 3.3, 9.3, 15.3, 24.3, 36.3, 39.3, 42.3, 54.3, 57.3};
  parameter Real ys[N] =
      {5., 6.,  5.,  6.,   5.,   6.,   5.,   6.,   5.,   6.};

equation

  y = noEvent(if time <= ts[2] then ys[1] else
              if time <= ts[3] then ys[2] else
              if time <= ts[4] then ys[3] else
              if time <= ts[5] then ys[4] else
              if time <= ts[6] then ys[5] else
              if time <= ts[7] then ys[6] else
              if time <= ts[8] then ys[7] else
              if time <= ts[9] then ys[8] else
              if time <= ts[10] then ys[9] else ys[10]);

end PRBS1;
```

```
model PRBS2

  Modelica.Blocks.Interfaces.RealOutput y;
  parameter Integer N = 11;
  parameter Real ts[N] =
      {0., 0.3, 9.3, 21.3, 24.3, 27.3, 39.3, 42.3, 48.3, 51.3, 57.3};
  parameter Real ys[N] =
      {5., 6.,  5.,  6.,   5.,   6.,   5.,   6.,   5.,   6.,   5.};

equation

  y = noEvent(if time <= ts[2] then ys[1] else
              if time <= ts[3] then ys[2] else
              if time <= ts[4] then ys[3] else
              if time <= ts[5] then ys[4] else
              if time <= ts[6] then ys[5] else
              if time <= ts[7] then ys[6] else
              if time <= ts[8] then ys[7] else
              if time <= ts[9] then ys[8] else
              if time <= ts[10] then ys[9] else
              if time <= ts[11] then ys[10] else ys[11]);

end PRBS2;

optimization QuadTank_PE_Old(
    objective=sum((x1_meas[i] - qt.x1(t_meas[i]))^2 +
                  (x2_meas[i] - qt.x2(t_meas[i]))^2 +
                  (x3_meas[i] - qt.x3(t_meas[i]))^2 +
                  (x4_meas[i] - qt.x4(t_meas[i]))^2 for i in 1:N_meas),
    startTime=0, finalTime=60)

  QuadTank qt(a1(free=true, initialGuess=0.03e-4, nominal=0.03e-4,
                 min=0, max=0.1e-4),
              a2(free=true, initialGuess=0.03e-4, nominal=0.03e-4,
                 min=0, max=0.1e-4),
              a3(free=true, initialGuess=0.03e-4, nominal=0.03e-4,
                 min=0, max=0.1e-4),
              a4(free=true, initialGuess=0.03e-4, nominal=0.03e-4,
                 min=0, max=0.1e-4));

  parameter Integer N_meas = 121;
  parameter Real t_meas[N_meas] = 0:60.0/(N_meas-1):60;
  parameter Real x1_meas[N_meas] = ones(N_meas);
  parameter Real x2_meas[N_meas] = ones(N_meas);
  parameter Real x3_meas[N_meas] = ones(N_meas);
  parameter Real x4_meas[N_meas] = ones(N_meas);

  PRBS1 prbs1;
  PRBS2 prbs2;

equation

  connect(prbs1.y, qt.u1);
  connect(prbs2.y, qt.u2);

end QuadTank_PE_Old;

optimization QuadTank_PE_New(objective=1, startTime=0, finalTime=60)

  QuadTank qt(a1(free=true, initialGuess=0.03e-4, nominal=0.03e-4,
                 min=0, max=0.1e-4),
              a2(free=true, initialGuess=0.03e-4, nominal=0.03e-4,
                 min=0, max=0.1e-4),
              a3(free=true, initialGuess=0.03e-4, nominal=0.03e-4,
```

```
                min=0, max=0.1e-4),
              a4(free=true, initialGuess=0.03e-4, nominal=0.03e-4,
                min=0, max=0.1e-4));

  input Real u1(free=true) = qt.u1;
  input Real u2(free=true) = qt.u2;

end QuadTank_PE_New;
```

## B.4 Distillation column

```
model Dist_Col

  // Import Modelica SI unit library
  import SI = Modelica.SIunits;
  type MolarFlowRate = Real(quantity="Molar Flow Rate", unit="mol/s");
  type Moles = Real(quantity="Mols", unit="mol", displayUnit="mols");

  // The model has to be rescaled in order to enable
  // use of SI units.

  // Model parameters
  parameter MolarFlowRate Feed = 24/60 "Feed Flowrate [mol/s]";
  parameter SI.MassFraction x_Feed = 0.5 "Mole Fraction of Feed";
  parameter MolarFlowRate D = x_Feed * Feed "Distillate Flowrate [mol/s]";
  parameter Real vol = 1.6 "Relative Volatility = (yA/xA) / (yB/xB)";
  parameter Moles atray = 0.25 "Molar Holdup in the Condenser [mol]";
  parameter Moles acond = 0.5 "Molar Holdup on each Tray [mol]";
  parameter Moles areb = 1.0 "Molar Holdup in the Reboiler [mol]";

  // Algebraic variables
  Real rr "Reflux Ratio";
  MolarFlowRate L "Liquid Flowrate in the Rectification Section [mol/s]";
  MolarFlowRate V "Vapor Flowrate in the Column [mol/s]";
  MolarFlowRate FL "Liquid Flowrate in the Stripping Section [mol/s]";

  parameter Integer N = 32 "Number of trays";

  SI.MoleFraction y[N](each min=0) "Vapor Mole Fraction of Component A";

  // Initial values for the states
  parameter Real x_0[N] =
      {0.93541941,  0.90052553,  0.86229644,  0.82169939,  0.77999079,
       0.73857167,  0.6988049 ,  0.66184252,  0.62850776,  0.59925269,
       0.57418567,  0.55314422,  0.53578454,  0.5216655 ,  0.51031495,
       0.50127506,  0.49412898,  0.48544973,  0.47420289,  0.45980255,
       0.44164493,  0.41918698,  0.39206347,  0.36023105,  0.32410541,
       0.28463656,  0.24326615,  0.20174466,  0.16184802,  0.12508549,
       0.09249569,  0.06458059};

  SI.MoleFraction x[N](start=x_0, each fixed=true, each min=0)
      "Reflux Drum Liquid Mole Fraction of Component A";

  Modelica.Blocks.Interfaces.RealInput u;

equation

  rr = u;
  L = rr*D;
  V = L+D;
  FL = Feed + L;

  // Vapor Mole Fractions of Componenent A
  // From the equilibrium assumption and mole balances
  // 1) vol = (yA/xA) / (yB/xB)
  // 2) xA + xB = 1
  // 3) yA + yB = 1
  for i in 1:N loop
     y[i] = (x[i]*vol) / (1 + (vol-1)*x[i]);
  end for;
```

```
  // ODEs
  der(x[1]) = (V * (y[2]-x[1])) / acond;
  for i in 2:16 loop
    der(x[i]) = (L*(x[i-1]-x[i]) - V*(y[i]-y[i+1])) / atray;
  end for;

  der(x[17]) = (D + L*x[16] - FL*x[17] - V*(y[17]-y[18])) / atray;

  for i in 18:31 loop
    der(x[i]) = (FL*(x[i-1]-x[i]) - V*(y[i]-y[i+1])) / atray;
  end for;

  der(x[32]) = (FL*x[31] - (Feed-D)*x[32] - V*y[32]) / areb;

end Dist_Col;

optimization Dist_Col_OCP(objective=cost(finalTime),
                          startTime=0., finalTime=50.)

  extends Dist_Col(u(min=1, initialGuess=1.5),
                   x(each initialGuess=0.5), y(each initialGuess=0.5));

  Real cost(start=0, fixed=true);
  parameter Real gamma = 1000;
  parameter Real rho = 1;

  parameter Real u_ref = 2.0;
  parameter Real y1_ref = 0.89581418893128228;

equation

  der(cost)  = gamma * (y[1] - y1_ref)^2 + rho*(u - u_ref)^2;

end Dist_Col_OCP;
```

# Index

# Bibliography


[A+11] Joel Andersson et al. CasADi. http://www.casadi.org/, Sep 2011.

[AB11] Modelon AB. JModelica.org User Guide 1.6.0. http://www.jmodelica.org/page/236, Oct 2011.

[AHD10] Joel Andersson, Boris Houska, and Moritz 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*. Department of Electrical Engineering and Optimization in Engineering Center (OPTEC), K.U.Leuven, Belgium, Oct 2010.

[AP91] Uri M. Ascher and Linda R. Petzold. Projected implicit runge-kutta methods for differential-algebraic equations. *SIAM Journal on Numerical Analysis*, 28(4):1097–1120, 1991.

[Ben05] David Benson. *A Gauss Pseudospectral Transcription for Optimal Control*. PhD thesis, Massachusetts Institute of Technology, 2005.

[Bet10] John T. Betts. *Practical Methods for Optimal Control and Estimation using Nonlinear Programming*. SIAM's Advances in Design and Control. Society for Industrial and Applied Mathematics, 2nd edition, 2010.

[Bie10] Lorenz T. Biegler. *Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes*. MOS-SIAM Series on Optimization. Mathematical Optimization Society and the Society for Industrial and Applied Mathematics, 2010.

[CDk11] Francesco Casella, Filippo Donida, and Johan Åkesson. Object-oriented modeling and optimal control: A case study in power plant start-up. In *18th IFAC World Congress*, Milano, Italy, August 2011.

[GW08] Andreas Griewank and Andrea Walther. *Evaluating Derivatives – Principles and Techniques of Algorithmic Differentiation*. MOS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, 2008.

[HE02] Juergen Hahn and Thomas F. Edgar. An improved method for nonlinear model reduction using balancing of empirical gramians. *Computers & Chemical Engineering*, 26(10):1379 – 1397, 2002.

[HR71] G. A. Hicks and W. H. Ray. Approximation methods for optimal control synthesis. *The Canadian Journal of Chemical Engineering*, 49(4):522–528, 1971.

[HR07] Geoffrey T. Huntington and Anil V. Rao. A comparison between global and local orthogonal collocation methods for solving optimal control problems. *2007 American Control Conference*, pages 1950–1957, 2007.

[Hun07] Geoffrey T. Huntington. *Advancement and Analysis of Gauss Pseudospectral Transcription for Optimal Control Problems*. PhD thesis, Massachusetts Institute of Technology, 2007.

[HW96] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and differential-algebraic problems*. Springer series in computational mathematics. Springer-Verlag, 2nd edition, 1996.

[Joh00] Karl Henrik Johansson. The quadruple-tank process: a multivariable laboratory process with an adjustable zero. *Control Systems Technology, IEEE Transactions on*, 8(3):456–465, may 2000.

[Åk07] Johan Åkesson. *Languages and Tools for Optimization of Large-Scale Systems*. PhD thesis, Faculty of Engineering at Lund University, 2007.

[Åk08] Johan Åkesson. Optimica—an extension of modelica supporting dynamic optimization. In *In 6th International Modelica Conference 2008*. Modelica Association, March 2008.

[Kan07] Takashi Kanamaru. Van der Pol oscillator. *Scholarpedia*, 2(1):2202, 2007.

[KB08] Shivakumar Kameswaran and Lorenz T. Biegler. Convergence rates for direct transcription of optimal control problems using collocation at radau points. *Computational Optimization and Applications*, 41:81–126, September 2008.

[NW06] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, 2nd edition, 2006.

[PkC10] Roberto Parrotto, Johan Åkesson, and Francesco Casella. An XML representation of DAE systems obtained from continuous-time Modelica models. In *Third International Workshop on Equation-based Object-oriented Modeling Languages and Tools*, September 2010.

[RM09] J.B. Rawlings and D.Q. Mayne. *Model Predictive Control Theory and Design*. Nob Hill Pub., 2009.

[RR04] Michael Renardy and Robert C. Rogers. *An Introduction to Partial Differential Equations*. Texts in Applied Mathematics. Springer, 2nd edition, 2004.

[Sim10] Theodore E. Simos. *Recent Advances in Computational and Applied Mathematics*. Springer, 2010.

[Udr10] Constantin Udriste. Equivalence of multitime optimal control problems. *Balkan Journal of Geometry and Its Applications*, 15(1):155–162, 2010.

[WB06] Andreas Wächter and Lorenz T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.

[Zav08] Victor M. Zavala. *Computational Strategies for the Optimal Operation of Large-Scale Chemical Processes*. PhD thesis, Carnegie Mellon University, 2008.