Development of Parallel Indirect Methods for Solving Constrained Optimal Control Problems

Chaoyi Yang

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2020

Reading Committee:

Brian C. Fabien, Chair

Santosh Devasia

Duane Storti

Program Authorized to Offer Degree:
Mechanical Engineering

University of Washington

Abstract

Development of Parallel Indirect Methods for Solving Constrained Optimal Control Problems

Chaoyi Yang

Chair of the Supervisory Committee:
Brian C. Fabien

Optimal control is a subject where it is desired to determine the inputs to a dynamical system which optimize a specified performance index while satisfying any constraints on the motion of the system at the same time. Because of the complexity of most applications, optimal control problems (OCPs) are most often solved numerically.

The indirect method for solving OCPs is based on solving the firs-order necessary conditions for the optimum and these necessary conditions are written as a two-point boundary value problem. This dissertation presents indirect methods for solving OCPs including control variable inequality constraints (CVICs), state variable inequality constraints (SVICs), and parameters. The necessary conditions of the optimum for the OCPs are written as a boundary value problem with differential algebraic equations which are proved to be index-1 (BVP-DAEs). The complementarity conditions in the BVP associated with those inequality constraints are approximated using Kanzow's smoothed Fisher-Burmeister formula.

Two numerical methods for solving the BVP-DAEs are developed. The multiple shooting technique is one of the techniques applied. Except solving the DAE using a single step linearly implicit Runge-Kutta method, a novel implementation based on MATLAB built-in DAE solver `ode15s` is provided. The other method used is a collocation method where the DAEs are approximated using Lagrange polynomials within each mesh and required to be satisfied at Lobatto points within each interval. Newton's method is performed to solve the BVP-DAEs systems for both methods and the descent direction is found by solving a

sparse bordered almost block diagonal (BABD) linear system with a structured orthogonal factorization algorithm. For the MATLAB implementation, the efficiency of the embedded parallel computing toolbox is explored.

Moreover, using the graphics processing unit (GPU) to accelerate the numerical algorithm solving process is very promising by using faster hardware. Combining those, this dissertation also presents the GPU based parallel implementation for both numerical methods, which is implemented using Python and CUDA. Numerical examples are presented to illustrate the efficiency of the implementation. The GPU based parallel implementations are shown to be significantly faster than the implementation using Central Processing Unit (CPU) alone or implemented using MATLAB for both methods.

Extending the collocation method presented, a study so called the collocation method with ph adaptive mesh refinement is introduced to further improve the efficient and the robustness of the collocation method presented. First, a novel method to estimate the error of the solution from collocation method is presented which serves as a basis of the ph adaptive mesh refinement method. In the original collocation method, the problem is solved based on a global unified number of collocation points used within each mesh interval without the dynamic mesh of the problem. In the ph adaptive method, not only the mesh varies during the solving process but also the collocation points used within each mesh interval keep changing until a desired numerical tolerance is met. The method is called "ph" because the mesh size of each interval (denoted by h) and the number of collocation points which is also the polynomial degrees (denoted by p) within each mesh interval are updated simultaneously.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGEMENT

I would like to express my gratitude to all of the people who have made this thesis possible. First, I would like to thank my academic advisor Professor Brian C. Fabien for his continuous guidance in the past five years. He has been extremely patient with not only the difficulties I ran into during my research, but also the difficulties I had as an international student and as a non-native English speaker. I am extremely grateful for having a mentor like him, who is incredibly helpful, encouraging and supportive. He has been very flexible with what I would like to explore and has provided many research ideas. Without Professor Fabien, many of the research ideas I would like to study would not be possible. Thank you to professor Duane Storti who not only served as my committee member but also taught me in classes which form the fundamental of my researches. Professor Storti gave me lots of valuable feedbacks on my work from his perspectives as a GPU expert and support me with the machines in his lab at the early stage of my research. I would also like to express my gratitude to Professors Behçet Açikmeşe and Santosh Devasia, for serving on my thesis committee as well as for their encouragement and help that elevates my thesis to a higher level. Thanks to Wanwisa Kisalang for being a knowledgeable and thoughtful counselor throughout my Ph.D. training at University of Washington. I also would like to thank many professors and researchers I have met throughout those years who have influenced my mind, my character and my career decisions. Finally, I would like to thank my family and all of my peers. Thank you to my mother, father, and girlfriend for constant support and love. I sincerely appreciate your patience and for all you have done in bringing me to be the person I am today. Thank you to my colleagues and friends at University of Washington for your help and companion throughout the graduate program.

DEDICATION

to my family

GLOSSARY

OCP: Optimal Control Problem

BVP: Boundary Value Problem

DAE: Differential Algebraic Equation

BVP-DAES: Boundary Value Problem involving Differential Algebraic Equations

CVIC: Control Variable Inequality Constraints

SVIC: State Variable Inequality Constraints

BABD: Bordered Almost Block Diagonal

GPU: Graphics Processing Unit

CPU: Central Processing Unit

CUDA: Compute Unified Device Architecture

SM: Streaming multiprocessor

Chapter 1

INTRODUCTION

Optimal control problem (OCP) is the problem that deals with finding a control law for a dynamical system with certain set of boundary conditions over a time period so that an objective functional is optimized. Numerical methods for solving optimal control problems can be classified into direct methods and indirect methods Betts [1998]. A schematic with the breakdown of the components used by each class of optimal control methods is shown in figure 1.1 Rao [2009].

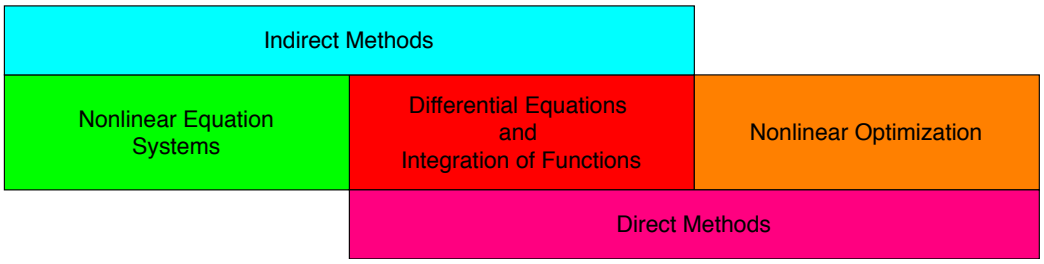


Figure 1.1: Major components of numerical methods for solving optimal control problems.

In a direct method, the states and/or controls are approximated using some appropriate function approximation and the cost functional is approximated at the same time. Then, the parameters of the function approximations are treated as unknown optimization variables and the OCP is transformed to a nonlinear programming (NLP) problem. Fabien [1998, 2008] develops a direct method where the control inputs are approximated using piecewise functions while the differential equations involved are approximated by a linearly implicit Runge-Kutta method. The original optimal control problem is then transformed into a NLP problem. Gaussian quadrature collocation method Elnagar et al. [1995], ELNAGAR and RAZZAGHI [1997], Benson [2005], Rao et al. [2010] is another popular branch for directly solving OCPs. In a Gaussian quadrature collocation method, the problem is discretized

with points associated with the Gaussian quadrature and the states are approximated using Lagrange polynomial. The OCP is then transcribed to a finite-dimensional NLP problem.

In an indirect method, the calculus of variations or the minimum principle is employed to obtain the first-order optimality conditions Hartl et al. [1995], Agrawal and Fabien [2013], Fabien [1995]. These conditions can be transformed into a two-point boundary-value problem (BVP) which normally involves differential algebraic equations (DAEs). Numerical methods for solving BVP-DAEs include collcation methods such as Ascher et al. [1981], Ascher and Spiteri [1994], Kierzenka and Shampine [2001], Fabien [2016a] and multiple shooting methods like Bock and Plitt [1984], Fabien [2014a,b]. Jacobson et al. [1971] uses the exterior penalty functions to transform the problems with state variable inequality constraints (SVICs) and Fabien [2014b] uses a slacked unconstrained penalty function method to approximate the SVICs. The collocation methods solve the problem by approximating the differential equations using Lagrange polynomial on collocation points. The multiple shooting method developed by Fabien [2014a] introduces non-negative slack variables to rewrite the inequality constraints where the DAEs involved are solved using a a single step linearly implicit Runge-Kutta method. The overall time interval is divided into multiple intervals where a residual equation is formed thereby and solved via a damped interior point Newton's method. A very robust parallel method to solve the sparse boarded almost block diagonal (BABD) system from the Newton's method is presented in that paper. Gerdts [2008], Fabien [2016b] also present indirect methods which are able to deal with inequality constraints, where the complementarity conditions associated with the inequality constraints are replaced with with Kanzow's smoothed Fishcher-Burmeister formula (see Kanzow [1996]). Moreover in Fabien [2016b], a noninterior damped Newton's method is presented.

The dissertation shows that the necessary conditions for the optimum of OCPs which contain CVICs, SVICs, and parameters can be written as a BVP-DAEs. These necessary conditions are obtained using the slacked unconstrained penalty function method to relax the inequality constraints. It is proved that the optimal cost functional of the transformed OCP using the slacked unconstrained penalty function method is the same as that of the original OCP. The complementarity conditions are approximated with the smoothed Kan-

zow's smoothed Fishcher-Burmeister formula. The resultant DAEs are proved to be index-1

Two different numerical algorithms are presented for solving the BVP-DAEs, which are based on multiple shooting method and collocation method. For the multiple shooting method, except using the single step linearly implicit Runge-Kutta method in Fabien [2014a], the possibility of using a MATLAB MATLAB [2017] built-in integrator `ode15s` is also explored. The computing capability of MATLAB are taken full advantage of. The embedded parallel computing toolbox is used to try to increase the efficiency of the algorithm. A collocation algorithm is also implemented using MATLAB to better compare the efficiency of the two algorithms in the same MATLAB environment.

In the last decade, the GPU based parallel computing is gaining more and more traction on a global level in various areas. Using the graphics processing unit (GPU) to accelerate the algorithm computation process is very promising by using faster hardware. In this dissertation, the GPU based implementations for the algorithms are also presented, which are shown to work very well on various OCPs. After showing the algorithms and their implementations, numerical evaluations of concrete examples are presented to demonstrate the feasibility and adaptation of the software. Various performance files (Dolan and Moré [2002], Gould and Scott [2016]) are provided to benchmark and quantify the relative performance of the solvers on the example set of OCPs from open literatures.

Chapter 2 presents the problem statement and the essential assumptions. Then, the original OCP with constraints is modified using the slacked unconstrained penalty function method. The resulting first-order necessary conditions are written in the form of a two-point BVP-DAEs where the DAEs are shown to be index-1. The chapter also shows how the complementarity conditions in the differential algebraic equations are approximated using Kanzow's smoothed Fischer-Burmeister formula.

Chapter 3 shows a multiple shooting method to solve the BVP-DAEs. This multiple shooting method leads to a system of residual equation that determines the unknowns on a fixed mesh in the time interval of interest. The residual equation is solved using a damped Newton's continuation method. The DAE integration based on both the single step linearly implicit Runge-Kutta (Rosenbrock-Wanner, ROW) method (Hairer and Wanner [1996]) and `ode15s` using MATLAB are presented. The computation of the search direction for

solving the residual equation requires the solution of a BABD linear system (see Amodio et al. [2000], and Fabien [2014a]). A very robust QR factorization method for solving the BABD system is presented in both sequential and parallel implementations. The numerical evaluation on several nontrivial examples are also provided.

Chapter 4 presents the collocation method used to solve the BVP-DAEs and some analytical aspects of the continuation Newton's method used to solve the residual equation from the collocation formulation. The chapter also gives the collocation algorithm evaluation details. The MATLAB implementation for the algorithm and the comparison with the multiple shooting algorithm using MATLAB are discussed.

Chapter 5 presents new strategies for using the graphics processing unit (GPU) to adapt both algorithm structures and accelerate their computational speed. The chapter explains adequate background to understand the CUDA (Storti and Yurtoglu [2015], Nickolls et al. [2008], Luebke [2008]) programming model and the corresponding GPU based implementations. Concrete examples of the Python (Van Rossum and Drake Jr [1995]) and CUDA implementations are given to show the power of the GPU parallel computing.

Chapter 6 discusses the current the further enhancement of the collocation method introduced in Chapter 4 with an ph adaptive mesh refinement. The chapter first derives a method to estimate the relative error between the numerical collocation solution and the exact solution with mathematical derivations. The relative error estimate is then used to guide the mesh refinement process. In the original collocation method, the problem is solved with a global unified number of collocation points within each mesh interval without the dynamic mesh refinement of the problem. Whereas in the ph adaptive method, not only the mesh varies during the solving process but also the collocation points used within each mesh interval keep changing until a desired numerical tolerance is met. The method is called 'ph' because the mesh size of each interval (denoted by h) and the number of collocation points which is also the polynomial degrees (denoted by p) om each mesh interval are updated simultaneously. The mesh size is increased when the estimated error within a mesh interval is beyond the numerical tolerance by either increasing the order of the approximating polynomial or dividing the interval into multiple subintervals. In the mesh interval where the error tolerance has been met, the mesh size is reduced by either decreasing

the degree of the approximating polynomial or merging adjacent mesh intervals. The chapter presents three examples which show that the approach is more computationally efficient and robust when compared with fixed-order methods The dissertation summarizes in Chapter 8 with all the work accomplished.

Below is a list of the publications conducted during this Ph.D. thesis period.

BIBLIOGRAPHY

Chaoyi Yang and Brian C. Fabien, "Parallel Solution of Optimal Control Problems Using the GPU" submitted to: Optimal Control Applications and Methods; under review; 2020

Chaoyi Yang and Brian C. Fabien, "Accelerating a Collocation Method for Solving Index-1 BVP-DAEs Arising from Optimal Control Problems Using the GPU", submitted to: Applied Mathematics and Computation; under review; 2020

Chaoyi Yang and Brian C. Fabien, "An Adaptive Mesh Refinement Method for Indirectly Solving Optimal Control Problems", submitted to: Numerical Algorithms; under review; 2020

Chapter 2

OPTIMAL CONTROL PROBLEM

2.1 Optimal Control Problem

All the algorithms developed in this dissertation uses the OCP in the following format. Find the state vector $x(t) \in \mathbb{W}^{1,\infty}([t_{\mathrm{i}}, t_{\mathrm{f}}], \mathbb{R}^{n_x})$, where $\mathbb{W}^{1,\infty}([t_{\mathrm{i}}, t_{\mathrm{f}}], \mathbb{R}^{n})$ denotes the Banach space of all absolutely continuous functions $v : [t_{\mathrm{i}}, t_{\mathrm{f}}] \to \mathbb{R}^n$ with norm $\|v\|_{1,\infty} = \max(\|v\|_\infty, \|\dot{v}\|_\infty) < \infty$, the control vector $u(t) \in \mathbb{L}^{\infty}([t_{\mathrm{i}}, t_{\mathrm{f}}], \mathbb{R}^{n_u})$, where $\mathbb{L}^{\infty}([t_{\mathrm{i}}, t_{\mathrm{f}}], \mathbb{R}^{n})$ denotes the Banach space of all measurable functions $v : [t_{\mathrm{i}}, t_{\mathrm{f}}] \to \mathbb{R}^n$ with norm $\|v\|_\infty = \text{ess sup}_{t_{\mathrm{i}} \le t \le t_{\mathrm{f}}} \|v(t)\| < \infty$, and the parameter vector $w \in \mathbb{R}^{n_w}$ that minimizes the cost functional

$$J(x, u, w) = \phi(x(t_{\mathrm{f}}), w) + \int_{t_{\mathrm{i}}}^{t_{\mathrm{f}}} L(x(t), u(t), w) dt \tag{2.1}$$

subject to the constraints

$$\dot{x}(t) = f(x(t), u(t), w), \ t \in [t_{\mathrm{i}}, t_{\mathrm{f}}], \tag{2.2}$$

$$0 = \Gamma(x(t_{\mathrm{i}}), w), \tag{2.3}$$

$$0 \ge d_i(x(t), u(t), w), \ i = 1, 2, \ldots, n_d, \ t \in [t_{\mathrm{i}}, t_{\mathrm{f}}], \tag{2.4}$$

$$0 \ge s_j(x(t)), \ j = 1, 2, \ldots, n_s, \ t \in [t_{\mathrm{i}}, t_{\mathrm{f}}], \tag{2.5}$$

$$0 = \Psi(x(t_{\mathrm{f}}), w). \tag{2.6}$$

The cost functional $J(x, u, w)$ is made up of a scalar terminal penalty term $\phi(x(t_{\mathrm{f}}), w)$, and a scalar integral term with integrand $L(x(t), u(t), w)$. The optimal solution must satisfy the differential equations (2.2) where $f(x(t), u(t), w) \in \mathbb{R}^{n_x}$, the initial time constraints (2.3) where $\Gamma(x(t_{\mathrm{i}}), w) \in \mathbb{R}^{n_\Gamma}$, the mixed control state parameter inequality constraints (2.4), the state variable inequality constraints (2.5), and the final time constraints (2.6) where $\Psi(x(t_{\mathrm{f}}), w) \in \mathbb{R}^{n_\Psi}$.

It should be noted that the dynamic system equations considered (2.2) are autonomous and non-autonomous system can be written as autonomous system by adding a new state

variable to represent the independent time variable. Also, note that problems with unknown initial conditions and minimum time problems with unknown final time $t_{\rm f}$ can be transformed into the form (2.1)-(2.6) by adding parameters to represent the unknown initial conditions and the unknown final time (see Miele [1980]).

The numerical solutions developed in this paper rely on the following assumptions.

Assumption 1. *(Existence of a solution.) There exist a state vector $x(t)$, a control vector $u(t)$, and a parameter vector w that solve the problem defined by (2.1)-(2.6).*

Assumption 2. *(Smoothness of functions.) The functions ϕ, L, f, Γ, d_i, $i = 1, 2, \ldots, n_d$, $s_j, j = 1, 2, \ldots, n_s$ and Ψ are at least twice differentiable with respect to their arguments.*

The i-th control state parameter inequality constraint is said to be active if $d_i(x, u, w) = 0$, while the constraint is said to be inactive if $d_i(x, u, w) < 0$. Let the index set of the active inequality constraints be denoted as $\mathbf{A}(x, u, w) = \{i \mid d_i(x, u, w) = 0\}$, with the cardinality of $\mathbf{A}(x, u, w)$ equal to $\bar{n}_d$. Here, $D_u d_{\mathbf{A}}(x, u, w)$ is the n_u by $\bar{n}_d$ matrix whose columns are $\partial d_i(x, u, w)/\partial u,\ i \in \mathbf{A}(x, u, w)$.

Assumption 3. *(Constraint qualification.) The optimal control of the problem is such that the gradients of the active control state parameter inequality constraints are linearly independent, i.e.,*

$$\text{rank}\left[D_u d_{\mathbf{A}}(x, u, w)\right] = \bar{n}_d$$

2.2 Transformation of the OCP

In order to obtain an approximate solution to OCP (2.1)-(2.6), the following relaxed OCP is considered first. Find $x(t) \in \mathbb{R}^{n_x}$, $u(t) \in \mathbb{R}^{n_u}$, $\nu(t) \in \mathbb{R}^{n_d}$, $\sigma(t) \in \mathbb{R}^{n_s}$, and $w \in \mathbb{R}^{n_w}$ that

minimize the cost functional

$$\begin{aligned}\hat{J}(x,u,w,\nu,\sigma;\alpha) = \phi(x(t_{\mathrm{f}}),w) &+ \int_{t_{\mathrm{i}}}^{t_{\mathrm{f}}} L(x(t),u(t),w) \\ &+ \frac{1}{2\alpha}\sum_{i=1}^{n_d}(d_i(x(t),u(t),w)+\nu_i(t))^2 \\ &+ \frac{1}{2\alpha}\sum_{j=1}^{n_s}(s_j(x(t))+\sigma_j(t))^2 \\ &+ \frac{\alpha}{2}u(t)^T u(t)\,dt \end{aligned} \tag{2.7}$$

subject to the constraints

$$\dot{x}(t) = f(x(t),u(t),w),\ t\in[t_{\mathrm{i}},t_{\mathrm{f}}], \tag{2.8}$$

$$0 = \Gamma(x(t_{\mathrm{i}}),w), \tag{2.9}$$

$$0 \le \nu_i(t),\ i=1,2,\ldots,n_d,\ t\in[t_{\mathrm{i}},t_{\mathrm{f}}], \tag{2.10}$$

$$0 \le \sigma_j(t),\ j=1,2,\ldots,n_s,\ t\in[t_{\mathrm{i}},t_{\mathrm{f}}], \tag{2.11}$$

$$0 = \Psi(x(t_{\mathrm{f}}),w). \tag{2.12}$$

In this problem formulation, $\alpha > 0$ is a penalty parameter. Here, the CVICs and SVICs are treated as penalty terms using the non-negative slack variables $\nu(t)$ and $\sigma(t)$, respectively. These slack variables are considered to be control inputs in this formulation. Also, the term $\frac{\alpha}{2}u(t)^T u(t)$ is added to ensure that the second-order condition (Assumption 4) is satisfied when $\alpha > 0$ is sufficiently large, which is important in proving Theorem 2. Note that this formulation is used to extend the results given in Fabien [2014b].

The usefulness of this slack unconstrained transformation is demonstrated in the following theorem.

Theorem 1. *Let* $\{\alpha^{(k)}\}$ *be an infinite sequence of positive numbers such that* $\alpha^{(k)} > \alpha^{(k+1)} > 0$ *and* $\lim_{k\to\infty}\alpha^{(k)} = 0$. *Let* $(x^{(k)},u^{(k)},w^{(k)},\nu^{(k)},\sigma^{(k)})$ *represents the optimal solution for the transformed optimal control problem (2.7)-(2.12) with the penalty parameter* $\alpha = \alpha^{(k)}$. *Then,* $\lim_{k\to\infty}\hat{J}(x^{(k)},u^{(k)},w^{(k)},\nu^{(k)},\sigma^{(k)};\alpha^{(k)}) = J(x^*,u^*,w^*)$, *where the triple* (x^*,u^*,w^*) *is the optimal solution for the original optimal control problem (2.1)-(2.6), and* $J(x,u,w) = \phi(x(t_{\mathrm{f}}),w) + \int_{t_{\mathrm{i}}}^{t_{\mathrm{f}}} L(x(t),u(t),w)\,dt$.

Proof. To prove this theorem, we need to restate the original and the transformed optimal control problems as follows. First, we define the scalar cost functional $J(x,u,w) = \Phi(x(t_{\rm f}),w)+\int_{t_{\rm i}}^{t_{\rm f}} L(x(t),u(t),w)\,dt$. Define the feasible set of the original problem $\Omega(x,u,w) = \{x(t),u(t),w \mid \dot{x}(t)-f(x,u,w)=0,\ \Gamma(x(t_{\rm i}),w)=0,\ \Psi(x(t_{\rm f}),w)=0,\ d_i(x,u,w)\le 0,\ i=1,2,\ldots,n_d,\ s_j(x)\le 0,\ j=1,2,\ldots,n_s\}$, which is assumed to be compact and non-empty. Then, the original problem becomes $\min_{x,u,w\in\Omega} J(x,u,w)$ and denote the optimal solution to this problem as $(x^*(t),u^*(t),w^*)$. Moreover, we assume $J(x^*,u^*,w^*) > -\infty$. Define $\nu_i^*(t) = \max(-d_i(x^*(t),u^*(t),w^*),0),\ i=1,2,\ldots,n_d$ and $\sigma_j^*(t) = \max(-s_j(x^*(t)),0),\ j=1,2,\cdots,n_s$. Using these definitions, it is easy to show that $D^* = \frac{1}{2}\int_{t_{\rm i}}^{t_{\rm f}} \Sigma_{i=1}^{n_d}$ $(d_i(x^*(t),u^*(t),w^*)+\nu_i^*(t))^2dt = 0$ and $S^* = \frac{1}{2}\int_{t_{\rm i}}^{t_{\rm f}} \Sigma_{j=1}^{n_s}(s_i(x^*(t))+\sigma_j^*(t))^2dt = 0$. Since $u^*(t)$ is bounded, $U^* = \frac{1}{2}\int_{t_{\rm i}}^{t_{\rm f}} u^*(t)^T u^*(t)\,dt < \infty$. So now we can define the scalar cost functional of the transformed optimal control problem as

$$\begin{aligned} K(x,u,w,\nu,\sigma;\alpha) = \Phi(x(t_{\rm f}),w) &+ \int_{t_{\rm i}}^{t_{\rm f}} L(x(t),u(t),w) \\ &+ \frac{1}{2\alpha}\sum_{i=1}^{n_d}(d_i(x(t),u(t),w)+\nu_i(t))^2 \\ &+ \frac{1}{2\alpha}\sum_{j=1}^{n_s}(s_j(x(t))+\sigma_j(t))^2 \\ &+ \frac{\alpha}{2}u(t)^T u(t)\,dt \end{aligned}$$

The feasible set of the problem is $\Lambda(x,u,w,\nu,\sigma) = \{x,u,w,\nu,\sigma \mid \dot{x}(t)-f(x,u,w)$ $= 0,\ \Gamma(x(t_{\rm i}),w) = 0,\ \Psi(x(t_{\rm f}),w) = 0,\ \nu_i(t)\ge 0,\ i=1,2,\ldots,n_d,\ \sigma_j(t)\ge 0,\ j=1,2,\ldots,n_s\}$, which is also assumed to be compact and non-empty. Then, the transformed optimal control problem becomes

$$\min_{x,u,w,\nu,\sigma\in\Lambda} K(x,u,w,\nu,\sigma;\alpha).$$

For every $\alpha^{(k)}$, denote $(x^{(k)}(t),u^{(k)}(t),w^{(k)},\nu^{(k)}(t),\sigma^{(k)}(t))$ as the bounded optimal solution to the problem

$$\min_{x,u,w,\nu,\sigma\in\Lambda} K(x,u,w,\nu,\sigma;\alpha^{(k)}).$$

Define $D^{(k)} = \frac{1}{2}\int_{t_{\rm i}}^{t_{\rm f}} \Sigma_{i=1}^{n_d}(d_i(x^{(k)}(t),u^{(k)}(t),w^{(k)})+\nu_i^{(k)}(t))^2\,dt$ and $S^{(k)} = \frac{1}{2}\int_{t_{\rm i}}^{t_{\rm f}} \Sigma_{j=1}^{n_s}$

$(s_i(x^{(k)}(t)) + \sigma_j^{(k)}(t))^2\,dt$. Therefore, it can be concluded that for all k

$$K(x^{(k)}, u^{(k)}, w^{(k)}, \nu^{(k)}, \sigma^{(k)}; \alpha^{(k)}) \leq K(x^*, u^*, w^*, \nu^*, \sigma^*; \alpha^{(k)})$$

which is

$$\begin{aligned} & K(x^{(k)}, u^{(k)}, w^{(k)}, \nu^{(k)}, \sigma^{(k)}; \alpha^{(k)}) \\ \leq\ & J(x^*, u^*, w^*) + \frac{1}{\alpha^{(k)}} D^* + \frac{1}{\alpha^{(k)}} S^* + \alpha^{(k)} U^* \\ =\ & J(x^*, u^*, w^*) + \alpha^{(k)} U^*. \end{aligned} \tag{2.13}$$

Since the last equation holds for every $\alpha^{(k)}$, we must have $\lim_{k\to\infty} \frac{1}{2} \int_{t_i}^{t_f} \Sigma_{i=1}^{n_d}$ $(d_i(x^{(k)}(t), u^{(k)}(t), w^{(k)}) + \nu_i(t))^2 dt = 0$ and $\lim_{k\to\infty} \frac{1}{2} \int_{t_i}^{t_f} \Sigma_{j=1}^{n_s} (s_i(x^{(k)}(t)) + \sigma_j(t))^2 dt = 0$. Otherwise, the left hand side of the equation becomes unbounded while the right hand side approaches $J(x^*, u^*, w^*)$, which is a contradiction. This implies that $\lim_{k\to\infty}(x^{(k)}, u^{(k)}, w^{(k)},$ $\nu^{(k)}, \sigma^{(k)}) \in \Lambda$.

Let $(\overline{x}, \overline{u}, \overline{w}, \overline{\nu}, \overline{\sigma})$ be the limit of the sequence $(x^{(k)}, u^{(k)}, w^{(k)}, \nu^{(k)}, \sigma^{(k)})$, then the last inequality equation (2.13) implies

$$\lim_{k\to\infty} K(x^{(k)}, u^{(k)}, w^{(k)}, \nu^{(k)}, \sigma^{(k)}; \alpha^{(k)}) = J(\overline{x}, \overline{u}, \overline{w}) \leq J(x^*, u^*, w^*). \tag{2.14}$$

Also, because $(d_i(\overline{x}, \overline{u}, \overline{w}) + \overline{\nu}_i) = 0,\ i = 1, 2, \cdots, n_d$ and $(s_j(\overline{x}) + \overline{\sigma}_j) = 0,\ j = 1, 2, \cdots, n_s$, we know that $(\overline{x}, \overline{u}, \overline{w}) \in \Omega$. Since (x^*, u^*, w^*) minimizes $J(x, u, w)$ on the feasible set Ω, we have

$$J(x^*, u^*, w^*) \leq J(\overline{x}, \overline{u}, \overline{w}). \tag{2.15}$$

Combining equations (2.14) and (2.15), we have $J(\overline{x}, \overline{u}, \overline{w}) = J(x^*, u^*, w^*)$. □

Convex OCP case

In Theorem 1, we successfully proved that the optimal cost functional of the transformed OCP (2.7)-(2.12) converges to the optimal cost functional of the original OCP (2.1)-(2.6) as the penalty parameter α monotonically decreases to 0. If the original OCP is a strictly convex OCP, then the cost functional has only one global minimum. This indicates that the

converged optimal solution of the transformed OCP is the optimal solution of the original OCP.

2.3 Necessary Conditions

The necessary conditions for (x, u, w, ν, σ) to be a minimum of the OCP (2.7)-(2.12) can be derived using the calculus of variations and the Lagrange multiplier rule (see Agrawal and Fabien [2013], Fabien [1995], and Hartl et al. [1995]).

To state the necessary conditions, define the scalar Hamiltonian function as

$$\begin{aligned}\bar{H} = L(x,u,w) &+ \frac{1}{2\alpha}\sum_{i=1}^{n_d}(d_i(x,u,w)+\nu_i)^2 \\ &+ \frac{1}{2\alpha}\sum_{j=1}^{n_s}(s_j(x)+\sigma_j)^2 + \lambda^T f(x,u,w) \\ &- \delta^T\nu - \omega^T\sigma + \frac{\alpha}{2}u(t)^T u(t)\end{aligned}$$

where $\lambda(t) \in \mathbb{R}^{n_x}$ are the costate variables, $\delta(t) \in \mathbb{R}^{n_d}$ are the Lagrange multipliers associated with the non-negative slack variables $\nu(t)$, and $\omega(t) \in \mathbb{R}^{n_s}$ are the Lagrange multipliers associated with the non-negative slack variables $\sigma(t)$.

As we defined $\nu(t)$ and $\sigma(t)$ as the control inputs, if (x, u, w, ν, σ) represents a local

minimum of the OCP, it is necessary that the following conditions be satisfied.

$$\dot{x} = \frac{\partial \bar{H}}{\partial \lambda}, \tag{2.16}$$

$$\dot{\lambda} = -\frac{\partial \bar{H}}{\partial x}, \tag{2.17}$$

$$\dot{\gamma} = \frac{\partial \bar{H}}{\partial w}, \tag{2.18}$$

$$0 = \frac{\partial \bar{H}}{\partial u}, \tag{2.19}$$

$$0 = \frac{\partial \bar{H}}{\partial \nu_i} = \frac{1}{\alpha}(d_i + \nu_i) - \delta_i,\ i = 1, 2, \ldots, n_d, \tag{2.20}$$

$$0 = \frac{\partial \bar{H}}{\partial \sigma_j} = \frac{1}{\alpha}(s_j + \sigma_j) - \omega_j,\ j = 1, 2, \ldots, n_s, \tag{2.21}$$

$$0 = \delta_i \nu_i,\ \delta_i \geq 0,\ \nu_i \geq 0,\ i = 1, 2, \ldots, n_d, \tag{2.22}$$

$$0 = \omega_j \sigma_j,\ \omega_j \geq 0,\ \sigma_j \geq 0,\ j = 1, 2, \ldots, n_s, \tag{2.23}$$

$$0 = \Gamma(x(t_{\mathrm{i}}), w), \tag{2.24}$$

$$0 = \lambda(t_{\mathrm{i}}) + D_x \Gamma(x(t_{\mathrm{i}}), w) K_{\mathrm{i}}, \tag{2.25}$$

$$0 = \gamma(t_{\mathrm{i}}) - D_w \Gamma(x(t_{\mathrm{i}}), w) K_{\mathrm{i}}, \tag{2.26}$$

$$0 = \Psi(x(t_{\mathrm{f}}), w), \tag{2.27}$$

$$0 = \lambda(t_{\mathrm{f}}) - \frac{\partial \phi}{\partial x} - D_x \Psi(x(t_{\mathrm{f}}), w) K_{\mathrm{f}}, \tag{2.28}$$

$$0 = \gamma(t_{\mathrm{f}}) + \frac{\partial \phi}{\partial w} + D_w \Psi(x(t_{\mathrm{f}}), w) K_{\mathrm{f}}. \tag{2.29}$$

In order to write the stationary conditions in a more compact form, define

$$\mu_i(t) = (d_i(x(t), u(t), w) + \nu_i(t))/\alpha,\ i = 1, 2, \ldots, n_d,$$
$$\xi_j(t) = (s_j(x(t)) + \sigma_j(t))/\alpha,\ j = 1, 2, \ldots, n_s.$$

Then, the stationary conditions $\bar{H}_{\nu_i} = 0$ gives $\mu_i = \delta_i,\ i = 1, 2, \ldots, n_d$. Using this, equations (2.20) and (2.22) can be rewritten as

$$0 = d_i(x, u, w) + \nu_i - \alpha \mu_i,\ i = 1, 2, \ldots, n_d, \tag{2.30}$$

$$0 = \mu_i \nu_i,\ \mu_i \geq 0,\ \nu_i \geq 0,\ i = 1, 2, \ldots, n_d. \tag{2.31}$$

Also, the stationary conditions $\bar{H}_{\sigma_j} = 0$ gives $\xi_j = \omega_j,\ j = 1, 2, \ldots, n_s$. Using this,

equations (2.21) and (2.23) can be rewritten as

$$0 = s_j(x) + \sigma_j - \alpha\xi_j,\ j = 1, 2, \ldots, n_s, \tag{2.32}$$

$$0 = \xi_j\sigma_j,\ \xi_j \geq 0,\ \sigma_j \geq 0,\ j = 1, 2, \ldots, n_s. \tag{2.33}$$

2.4 Transformation of the Complementarity Conditions

Fabien [2016b] and Gerdts [2008] approximated the complementarity conditions using the Kanzow's smoothed Fisher-Burmeister formula Kanzow [1996], $\psi : \mathbb{R}^2 \Rightarrow \mathbb{R}$, which is

$$\psi(a, b; \alpha) = a + b - \sqrt{a^2 + b^2 + 2\alpha}. \tag{2.34}$$

It is proved in Kanzow [1996] that formula (2.34) satisfies the following properties which are critical for proving Theorem 2.

1. For any $\alpha > 0$, $\psi(a, b; \alpha) = 0 \ \Leftrightarrow\ ab = \alpha,\ a > 0,\ b > 0$.

2. For all $(a, b) \in \mathbb{R}^2,\ \alpha > 0$,

$$0 < \frac{\partial\psi(a, b; \alpha)}{\partial a} < 2, \quad 0 < \frac{\partial\psi(a, b; \alpha)}{\partial b} < 2.$$

Therefore, the complementarity conditions (2.22)-(2.23) can be approximated by the smooth formula (2.34) and rewritten as

$$0 = \psi(\mu_i, \nu_i; \alpha),\ i = 1, 2, \ldots, n_d, \tag{2.35}$$

$$0 = \psi(\xi_j, \sigma_j; \alpha),\ j = 1, 2, \ldots, n_s. \tag{2.36}$$

where equation (2.35) implies that

$$\mu_i\nu_i = \alpha,\ \mu_i > 0,\ \nu_i > 0,\ i = 1, 2, \ldots, n_d,$$

and equation (2.36) implies that

$$\xi_j\sigma_j = \alpha,\ \xi_j > 0,\ \sigma_j > 0,\ j = 1, 2, \ldots, n_s,$$

which are the same as the complementarity conditions (2.22) and (2.23) from the first-order necessary conditions when α approaches 0.

2.5 Boundary Value Problem

The necessary conditions can be written more compactly by separating the unknowns into: (*i*) differential variables (variables whose derivatives appear explicitly in the equations); (*ii*) algebraic variables (variables whose derivatives appear implicitly in the equations); and (*iii*) parameter variables. Specifically, define

$$y(t) = \begin{bmatrix} x(t) \\ \lambda(t) \\ \gamma(t) \end{bmatrix}, \; z(t) = \begin{bmatrix} u(t) \\ \mu(t) \\ \xi(t) \\ \nu(t) \\ \sigma(t) \end{bmatrix}, \; p = \begin{bmatrix} w \\ K_{\mathrm{i}} \\ K_{\mathrm{f}} \end{bmatrix}, \tag{2.37}$$

where $y(t) \in \mathbb{R}^{n_y}$ are the differential variables, $z(t) \in \mathbb{R}^{n_z}$ are the algebraic variables, and $p \in \mathbb{R}^{n_p}$ are the parameter variables, with $n_y = 2n_x + n_w$, $n_z = n_u + 2n_d + 2n_s$, and $n_p = n_w + n_\Gamma + n_\Psi$. The vector $\gamma(t) \in \mathbb{R}^{n_w}$ is used to write the stationary conditions associated with the parameters w as differential equations instead of integral equations. The Lagrange multipliers $K_{\mathrm{i}} \in \mathbb{R}^{n_\Gamma}$ are associated with the constraints at the initial time and the Lagrange multipliers $K_{\mathrm{f}} \in \mathbb{R}^{n_\Psi}$ are associated with the constraints at the final time.

Using these definitions, the necessary conditions for optimality (2.16)-(2.29) can be written as a two-point BVP-DAEs as

$$\dot{y}(t) = h(y(t), z(t), p), \tag{2.38}$$

$$0 = g(y(t), z(t), p, \alpha), \tag{2.39}$$

$$0 = r(y(t_{\mathrm{i}}), y(t_{\mathrm{f}}), p) \tag{2.40}$$

Here, (2.38) defines a set of differential equations, (2.39) defines a set of algebraic equations, and (2.40) defines a set of boundary conditions. Moreover

$$h(y(t), z(t), p) = \begin{bmatrix} \partial \bar{H}/\partial \lambda \\ -\partial \bar{H}/\partial x \\ \partial \bar{H}/\partial w \end{bmatrix} \in \mathbb{R}^{n_y}, \tag{2.41}$$

$$g(y(t), z(t), p, \alpha) = \begin{bmatrix} \partial \bar{H}/\partial u \\ d(x,u,w) + \mathcal{N}e - \alpha \mathcal{M}e \\ s(x) + \mathcal{S}e - \alpha \mathcal{E}e \\ \psi_d(\mu, \nu; \alpha) \\ \psi_s(\xi, \sigma; \alpha) \end{bmatrix} \in \mathbb{R}^{n_z}, \tag{2.42}$$

$$r(y(t_\mathrm{i}), y(t_\mathrm{f}), p) = \begin{bmatrix} \Gamma(x(t_\mathrm{i}), w) \\ \lambda(t_\mathrm{i}) + D_x\Gamma(x(t_\mathrm{i}), w)K_\mathrm{i} \\ \gamma(t_\mathrm{i}) - D_w\Gamma(x(t_\mathrm{i}), w)K_\mathrm{i} \\ \Psi(x(t_\mathrm{f}), w) \\ \lambda(t_\mathrm{f}) - \frac{\partial \phi}{\partial x} - D_x\Psi(x(t_\mathrm{f}), w)K_\mathrm{f} \\ \gamma(t_\mathrm{f}) + \frac{\partial \phi}{\partial w} + D_w\Psi(x(t_\mathrm{f}), w)K_\mathrm{f} \end{bmatrix} \in \mathbb{R}^{n_y+n_p}, \tag{2.43}$$

$d(x,u,w) = [d_1(x,u,w), d_2(x,u,w), \ldots, d_{n_d}(x,u,w)]^T$, $s(x) = [s_1(x), s_2(x), \ldots, s_{n_s}(x)]^T$, $\mathcal{N} = \mathrm{diag}(\nu_1, \nu_2, \ldots, \nu_{n_d})$, $\mathcal{M} = \mathrm{diag}(\mu_1, \mu_2, \ldots, \mu_{n_d})$, $\mathcal{S} = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_{n_s})$, $\mathcal{E} = \mathrm{diag}(\xi_1, \xi_2, \ldots, \xi_{n_s})$, $\psi_d(\mu, \nu; \alpha) = [\psi(\mu_1, \nu_1; \alpha), \psi(\mu_2, \nu_2; \alpha), \ldots, \psi(\mu_{n_d}, \nu_{n_d}; \alpha)]^T$, $\psi_s(\xi, \sigma; \alpha) = [\psi(\xi_1, \sigma_1; \alpha), \psi(\xi_2, \sigma_2; \alpha), \ldots, \psi(\xi_{n_s}, \sigma_{n_s}; \alpha)]^T$, $D_x\Gamma = [\partial\Gamma_1/\partial x, \partial\Gamma_2/\partial x, \ldots, \partial\Gamma_{n_\Gamma}/\partial x]^T$, $D_w\Gamma = [\partial\Gamma_1/\partial w, \partial\Gamma_2/\partial w, \ldots, \partial\Gamma_{n_\Gamma}/\partial w]^T$, $D_x\Psi = [\partial\Psi_1/\partial x, \partial\Psi_2/\partial x, \ldots, \partial\Psi_{n_\Psi}/\partial x]^T$, $D_w\Psi = [\partial\Psi_1/\partial w, \partial\Psi_2/\partial w, \ldots, \partial\Psi_{n_\Psi}/\partial w]^T$, and $e = [1, \ldots, 1]^T$.

Next, it can be shown that the BVP-DAEs (2.38)-(2.40) are index-1 along the optimal trajectory. To do so, the second order necessary condition is required. Hence, the following assumption is made.

Assumption 4. *(Second order necessary condition.) The matrix $\bar{H}_{uu} = \partial^2\bar{H}/\partial u^2 \in \mathbb{R}^{n_u \times n_u}$ is positive definite along the optimal trajectory. That is, for each bounded vector function $\hat{u}(t) \in \mathbb{R}^{n_u}$, in the interval $[t_\mathrm{i}, t_\mathrm{f}]$, it is assumed that there is a constant $c > 0$, such that $\hat{u}(t)^T\bar{H}_{uu}\hat{u}(t) \geq c\|\hat{u}(t)\|^2$.*

Theorem 2. *For $\alpha > 0$, if (x, u, w, ν, σ) solves the OCP (2.7)-(2.12), then there exists vector (y, z, p) that solves the BVP-DAEs (2.38)-(2.40) and $\partial g/\partial z$ is non-singular; that is, the BVP-DAEs are index-1.*

Proof. Equations (2.38)-(2.40) are the restatement of the necessary conditions (2.16)-(2.29). Therefore, the proof should show that the BVP-DAEs satisfy the index-1 condition which

is $\partial g/\partial z$ is non-singular along the optimal trajectory for $\alpha > 0$. The proof is done via contradiction. Suppose that $\partial g/\partial z$ is singular, then there should exist $\Delta u \in \mathbb{R}^{n_u}$, $\Delta\mu \in \mathbb{R}^{n_d}$, $\Delta\xi \in \mathbb{R}^{n_s}$, $\Delta\nu \in \mathbb{R}^{n_d}$, and $\Delta\sigma \in \mathbb{R}^{n_s}$ not being all zero such that

$$\left[\frac{\partial g}{\partial z}\right] \begin{bmatrix} \Delta u \\ \Delta\mu \\ \Delta\xi \\ \Delta\nu \\ \Delta\sigma \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.44}$$

where

$$\left[\frac{\partial g}{\partial z}\right] = \begin{bmatrix} \bar{H}_{uu} & D_u d & 0 & 0 & 0 \\ D_u d^T & -\alpha I & 0 & I & 0 \\ 0 & 0 & -\alpha I & 0 & I \\ 0 & D_\mu \psi_d & 0 & D_\nu \psi_d & 0 \\ 0 & 0 & D_\xi \psi_s & 0 & D_\sigma \psi_s \end{bmatrix}$$

In this expression, $\bar{H}_{uu} = \partial^2 \bar{H}/\partial u^2$, $D_u d = [\partial d_1/\partial u, \ldots, \partial d_{n_d}/\partial u]$, $D_\mu \psi_d = \mathrm{diag}(\partial \psi_{d_i}/\partial \mu_i)$, $D_\nu \psi_d = \mathrm{diag}(\partial \psi_{d_i}/\partial \nu_i)$, $i = 1, 2, \ldots, n_d$, $D_\xi \psi_s = \mathrm{diag}(\partial \psi_{s_j}/\partial \xi_j)$, $D_\sigma \psi_s = \mathrm{diag}(\partial \psi_{s_j}/\partial \sigma_j)$, $j = 1, 2, \ldots, n_s$. In section 2.4, it is implied that for $\alpha > 0$, $D_\mu \psi_d$, $D_\nu \psi_d$, $D_\xi \psi_s$, and $D_\sigma \psi_s$ are all positive definite.

First, it is easy to show that the third and fifth block rows in (2.44) yield $\Delta\xi = 0$ and $\Delta\sigma = 0$. The fourth row shows $\Delta\nu = -(D_\nu \psi_d)^{-1} D_\mu \psi_d \Delta\mu$. Using this result in the second row yields $\Delta\mu = [\alpha + (D_\nu \psi_d)^{-1} D_\mu \psi_d]^{-1} D_u d^T \Delta u$. Substitute this result in the first row and multiply the row by Δu^T gives

$$\Delta u^T \left\{ \bar{H}_{uu} + [\alpha + (D_\nu \psi_d)^{-1} D_\mu \psi_d]^{-1} D_u d D_u d^T \right\} \Delta u = 0. \tag{2.45}$$

Since $\bar{H}_{uu}$, $D_\nu \psi_d$, and $D_\mu \psi_d$ are all positive definite and $\alpha > 0$, Equation (2.45) yields $\Delta u = 0$. It then gives the result that $\Delta u = \Delta\mu = \Delta\nu = \Delta\xi = \Delta\sigma = 0$, which contradicts the assumption that $\partial g/\partial z$ is singular.

Therefore, this proof provides sufficient conditions for the BVP-DAEs to be index-1. □

2.6 Conclusion

This chapter first presents the optimal control problem of interest of this thesis which is equipped with control variable inequality constraints, state variables inequality constraints, and parameters. Then, a penalty function method is introduced to transfer problems with variable inequality constraints to a sequence of problems with inequality constraints. By decreasing the penalty term gradually to zero, it is also shown that the solution given by the transformed problem converges to the real solution of the original optimal control problem. The necessary conditions of the optimum of the transformed optimal control problem are the presented where the associated complementarity conditions are approximated by using the Kanzow's smoothed Fisher-Burmeister formula to eliminate the explicit inequalities. The necessary conditions are then represented in compact form as a boundary value problem involving differential-algebraic equations which are guaranteed to be index-1.

Chapter 3

MULTIPLE SHOOTING ALGORITHM

This chapter introduces a multiple shooting method to solve the BVP-DAEs (2.38)-(2.40). The implementation shown in this chapter are all using MATLAB computing environment.

3.1 Multiple Shooting Method

First, the multiple shooting method request the overall time interval $[t_{\mathrm{i}}, t_{\mathrm{f}}]$ be discretized into a mesh with N time nodes such that $t_{\mathrm{i}} = t_1 < t_2 < ... < t_N = t_{\mathrm{f}}$. Note that the mesh does not need to be uniform.

At each node $t_j,\ j = 1, 2, ..., N$, let $s_j^y \in \mathbb{R}^{n_y}$ represent the differential variables, and $s_j^z \in \mathbb{R}^{n_z}$ represent the algebraic variables at t_j. Let p denote the parameter variables of the system. Starting with the initial conditions (s_j^y, s_j^z, p) at time t_j, let $y(t; s_j^y, s_j^z, p)$ denote the solution to the DAEs (2.38)-(2.39) at time t in the interval of interest $[t_j, t_{j+1}]$. Then, we require the following conditions be satisfied:

1. At each time node t_j, the following algebraic equations must be satisfied.

$$g(s_j^y, s_j^z, p, \alpha) = 0,\ j = 1, 2, \ldots, N. \tag{3.1}$$

2. At each time node t_j except the last one, the following continuity conditions must be satisfied.

$$y(t_{j+1}; s_j^y, s_j^z, p) - s_{j+1}^y = 0,\ j = 1, 2, \ldots, N-1. \tag{3.2}$$

3. The following boundary conditions must be satisfied.

$$r(s_1^y, s_N^y, p) = 0. \tag{3.3}$$

Following this setup, the schematic of the multiple shooting method is shown in figure 3.1.

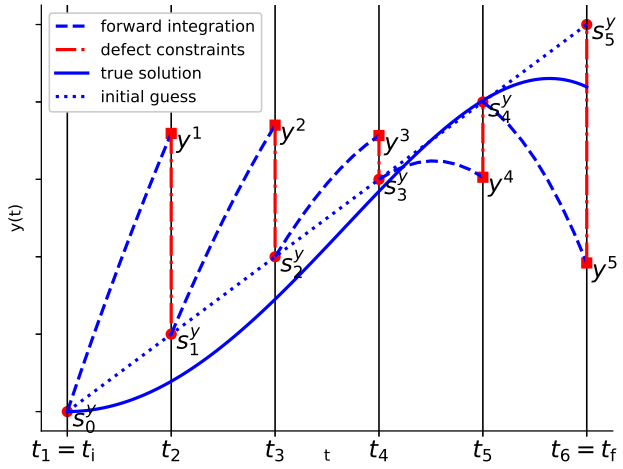


Figure 3.1: Schematic of the multiple shooting Method.

These conditions lead to a residual system as

$$F(s,p,\alpha) = \begin{bmatrix} g(s_1^y, s_1^z, p, \alpha) \\ y(t_2; s_1^y, s_1^z, p) - s_2^y \\ g(s_2^y, s_2^z, p, \alpha) \\ y(t_3; s_2^y, s_2^z, p) - s_3^y \\ \vdots \\ y(t_N; s_{N-1}^y, s_{N-1}^z, p) - s_N^y \\ g(s_N^y, s_N^z, p, \alpha) \\ r(s_1^y, s_N^z, p) \end{bmatrix} = 0 \tag{3.4}$$

where $s = [s_1^{yT}, s_1^{zT}, s_2^{yT}, s_2^{zT}, \ldots, s_N^{y\,T}, s_N^{z\,T}]^T \in \mathbb{R}^{n_s}$, $F(s,p,\alpha) \in \mathbb{R}^{n_s+n_p}$, and $n_s = N(n_y + n_z)$.

Thus, an approximate solution to the BVP-DAEs (2.38)-(2.40) can be obtained by solving the residual equation $F(s,p,\alpha) = 0$.

3.1.1 Solution of the residual equation

A damped Newton's method is used to solve the residual equation (3.4). For each $\alpha > 0$, given an initial estimate (s^k, p^k) of the solution to the residual equation (3.4), an improved solution (s^{k+1}, p^{k+1}) is obtained from the damped Newton's increment

$$\begin{bmatrix} s^{k+1} \\ p^{k+1} \end{bmatrix} = \begin{bmatrix} s^k \\ p^k \end{bmatrix} + \tau^k \begin{bmatrix} \Delta s^k \\ \Delta p^k \end{bmatrix}, \ k = 0, 1, \ldots, \tag{3.5}$$

$$DF(s^k, p^k, \alpha) \begin{bmatrix} \Delta s^k \\ \Delta p^k \end{bmatrix} = -F(s^k, p^k, \alpha), \tag{3.6}$$

where the Jacobian is

$$DF(s, p, \alpha) = \begin{bmatrix} A_1 & C_1 & & & & H_1 \\ & A_2 & C_2 & & & H_2 \\ & & \ddots & \ddots & & \vdots \\ & & & A_{N-1} & C_{N-1} & H_{N-1} \\ B_1 & & & & B_N & H_N \end{bmatrix}, \tag{3.7}$$

with

$$A_j = \begin{bmatrix} \partial g(s_j^y, s_j^z, p, \alpha)/\partial s_j^y & \partial g(s_j^y, s_j^z, p, \alpha)/\partial s_j^z \\ \partial y(t_{j+1})/\partial s_j^y & \partial y(t_{j+1})/\partial s_j^z \end{bmatrix},$$

$$C_j = \begin{bmatrix} 0 & 0 \\ -I & 0 \end{bmatrix},$$

$$H_j = \begin{bmatrix} \partial g(s_j^y, s_j^z, p, \alpha)/\partial p \\ \partial y(t_{j+1})/\partial p \end{bmatrix}, \ j = 1, 2, \ldots, N-1,$$

$$B_1 = \begin{bmatrix} 0 & 0 \\ \partial r(s_1^y, s_N^y, p)/\partial s_1^y & 0 \end{bmatrix},$$

$$B_N = \begin{bmatrix} \partial g(s_N^y, s_N^z, p, \alpha)/\partial s_N^y & \partial g(s_N^y, s_N^z, p, \alpha)/\partial s_N^z \\ \partial r(s_1^y, s_N^y, p)/\partial s_N^y & 0 \end{bmatrix},$$

$$H_N = \begin{bmatrix} \partial g(s_N^y, s_N^z, p, \alpha)/\partial p \\ \partial r(s_1^y, s_N^y, p)/\partial p \end{bmatrix}.$$

After obtaining $(\Delta s^k, \Delta p^k)$, we need to find the stepsize τ^k to guarantee that $\left\|F(s^{k+1}, p^{k+1}, \alpha)\right\| < \left\|F(s^k, p^k, \alpha)\right\|$, where $s^{k+1} = s^k + \tau^k \Delta s^k$ and $p^{k+1} = p^k + \tau^k \Delta p^k$. The stepsize τ^k is obtained by using the classic line-search method. Starting at $\tau^k = 1$ and evaluating the $\left\|F(s^{k+1}, p^{k+1}, \alpha)\right\|$, if $\left\|F(s^{k+1}, p^{k+1}, \alpha)\right\| < \left\|F(s^k, p^k, \alpha)\right\|$, we stop the line-search immediately, or we obtain a new τ^k by dividing it by two $\tau^k_{new} = \tau^k_{old}/2$ and reevaluate the residual term. The maximum number of line-search time is limited. If a damped term τ^k is failed to be found after the maximum number of line-search, a remesh of the problem is performed which is described below.

3.1.2 Integration of the DAEs and Sensitivity Equations

This section describes the techniques used to compute the residual and the building blocks of the Jacobian (3.7). The construction of the residual equation (3.4) requires $y(t_{j+1}; s^y_j, s^z_j, p) - s^y_{j+1}$, $j = 1, 2, \ldots, N-1$, where $y(t_{j+1}; s^y_j, s^z_j, p)$ represents the solution to the DAEs (2.38)-(2.39) at time node t_{j+1} using the initial conditions at time node t_j where $y(t_j) = s^y_j$ and $z(t_j) = s^z_j$. Whereas, the construction of the Jacobian matrix $DF(s, p, \alpha)$ (3.7) requires the trajectory sensitivities $\partial y(t_{j+1})/\partial s^y_j$, $\partial y(t_{j+1})/\partial s^z_j$, and $\partial y(t_{j+1})/\partial p$, $j = 1, 2, \ldots, N-1$. These trajectory sensitivities are obtained by integrating the variational equations associated with the DAEs (2.38)-(2.39) in the interval $[t_j, t_{j+1}]$. Here, two different implementations based on different integrators are presented, which are the integrator using a single step linearly implicit Runge-Kutta (Rosenbrock-Wanner, ROW) method (Hairer and Wanner [1996]) and the MATLAB built-in integrator `ode15s`.

3.1.3 Integration using ROW method

The integration is performed as follows.

Define

$$\bar{x}(t) = \begin{bmatrix} y(t) \\ z(t) \end{bmatrix}, \ \bar{f}(x, p, \alpha) = \begin{bmatrix} h(y(t), z(t), p) \\ g(y(t), z(t), p, \alpha) \end{bmatrix},$$

$$\bar{M} = \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} \in \mathbb{R}^{(n_y+n_z)\times(n_y+n_z)}, \ I \in \mathbb{R}^{n_y \times n_y}.$$

Then the index-1 DAEs (2.38)-(2.39) is written in the form of $\bar{M}\dot{\bar{x}} = \bar{f}(\bar{x}, p, \alpha)$.

Let $\bar{x}_j = \bar{x}(t_j) = [s_j^{yT}, s_j^{zT}]^T = s_j$ be the initial condition for the DAEs at time t_j. The accurate integration of the DAEs requires consistent initial conditions, which should be $g(y(t_j), z(t_j), p, \alpha) = 0$ at the initial time t_j. However, this condition is not guaranteed during the Newton's iteration process.

To ensure that the initial conditions are consistent, a relaxation term is introduced. For example in the interval $[t_j, t_{j+1}]$ the DAEs (2.38)-(2.39) is substituted by

$$\bar{M}\dot{\bar{x}} = \bar{f}(\bar{x}, p, \alpha) - \bar{v}_j, \tag{3.8}$$

where $\bar{v}_j = [0^T, g(s_j^y, s_j^z, p, \alpha)^T]^T \in \mathbb{R}^{n_y+n_z}$ is the relaxation vector. Hence, the initial conditions (s_j^y, s_j^z, p) are consistent for the DAE equation (3.8). One major advantage of using this technique is that as the iterations of the Newton's method converge to the solution of the boundary value problem, the relaxation term $\bar{v}_j \to 0$ and the relaxed DAEs becomes the original DAEs.

The Jacobian of $\bar{f}(\bar{x}, p, \alpha)$ with respect to $\bar{x}$ is needed during the DAEs integration. This Jacobian is defined as

$$\begin{aligned} \bar{W}(\bar{x}, p, \alpha) &= \partial \bar{f}(\bar{x}, p, \alpha)/\partial \bar{x} \\ &= \begin{bmatrix} \partial h(y,z,p)/\partial y & \partial h(y,z,p)/\partial z \\ \partial g(y,z,p,\alpha)/\partial y & \partial g(y,z,p,\alpha)/\partial z \end{bmatrix} \in \mathbb{R}^{(n_y+n_z)\times(n_y+n_z)}. \end{aligned} \tag{3.9}$$

The inputs to the integration of the DAEs are the relaxed DAEs equation (3.8), the time interval of interest $[t_j, t_{j+1}]$, the initial condition $\bar{x}_j$, the parameter p, the mass matrix $\bar{M}$, the Jacobian (3.9) of the DAEs and the relaxation term $\bar{v}_j$.

Then, the approximate solution of the DAEs at time t_{j+1} in time interval $[t_j, t_{j+1}]$,

$j = 1, 2, \ldots, N-1$ is obtained by using Algorithm 3.1.

Algorithm 3.1: Single step integration of the DAE from t_j to t_{j+1} using ROW method.

Input: Initial time t_j, terminal time t_{j+1}, initial condition of the DAEs $\bar{x}_j = [s_j^{yT}, s_j^{zT}]^T$, parameters p, Jacobian of the DAE $\bar{W}(\bar{x}_j, p, \alpha)$, DAE relaxations $\bar{v}_j$.

Output: The approximate solution of the DAEs (3.8) $\bar{x}_{j+1}$.

Set $\bar{W}_j = \bar{W}(\bar{x}_j, p, \alpha)$ and $\delta_j = t_{j+1} - t_j$.

for $i = 1, 2, \ldots, m$ **do**

Set the value for the DAE variables at stage i as

$$\hat{x}_{j,i} = \bar{x}_j + \delta_j \sum_{l=1}^{i-1} \bar{\alpha}_{i,l} \bar{k}_l, (\hat{x}_{j,1} = \bar{x}_j) \tag{3.10}$$

Solve the linear system by LU decomposition

$$(\bar{M} - \bar{\gamma}\delta_j \bar{W}_j)\bar{k}_i = \bar{f}(\hat{x}_{j,i}, p, \alpha) + \delta_j \bar{W}_j \sum_{l=1}^{i-1} \bar{\gamma}_{i,l} \bar{k}_l - \bar{v}_j. \tag{3.11}$$

end

Set the output values

$$\bar{x}_{j+1} = \bar{x}_j + \delta_j \sum_{i=1}^{m} \bar{b}_i \bar{k}_i. \tag{3.12}$$

In equations (3.10)-(3.12), the constant coefficients $\bar{\gamma}$, $\bar{\alpha}_{i,l}$, $\bar{\gamma}_{i,l}$, and $\bar{b}_i$ are the coefficients of an m-stage linearly implicit Runge-Kutta method. The set of coefficients used in this paper corresponds to the 4-th order ROW method developed by Novati [2008] and can be found in Fabien [2014a] Appendix A.

The ROW method also has an embedded formula which is used to estimate the local truncation error of the solution. Define $\bar{x}(t_{j+1})$ as the true solution to the DAE (3.8) with initial condition $\bar{x}_j$, p. The estimated error of the approximated solution $\bar{x}_{j+1}$ using the ROW method can be obtained by the formula

$$\bar{\epsilon}_{j+1} = \left\| (t_{j+1} - t_j) \sum_{i=1}^{m} \bar{e}_i \bar{k}_i \right\| \approx \left\| \bar{x}(t_{j+1}) - \bar{x}_{j+1} \right\|. \tag{3.13}$$

The coefficients $\bar{e}_i$ can be found in Fabien [2014a] Appendix A. This local truncation error estimation is used in the mesh refinement part described in the latter part of the chapter.

The construction of the Jacobian matrix $DF(s,p,\alpha)$ (3.7) requires the trajectory sensitivities $\partial y(t_{j+1})/\partial s_j^y$, $\partial y(t_{j+1})/\partial s_j^z$, and $\partial y(t_{j+1})/\partial p$, $j = 1, 2, \ldots, N-1$. These trajectory sensitivities are obtained by integrating the variational equations associated with (3.8) in the interval $[t_j, t_{j+1}]$. The approach used is described as follows.

Let $\bar{X}(t) = [(\partial\bar{x}/\partial s_j^y)^T, (\partial\bar{x}/\partial s_j^z)^T, (\partial\bar{x}/\partial p)^T]^T \in \mathbb{R}^{(n_y+n_z)\times(n_y+n_z+n_p)}$ represent the sensitivities of $\bar{x}$ with respect to s_j^y, s_j^z, and p. Then using equation (3.8), it can be seen that in each interval $[t_j, t_{j+1}]$, $j = 1, 2, \ldots, N-1$, the trajectory sensitivities must satisfy the linear DAEs

$$\bar{M}\dot{\bar{X}} = \bar{W}(\bar{x}, p, \alpha)\bar{X}(t) + \bar{F}_p(\bar{x}, p, \alpha) - \bar{V}_j, \tag{3.14}$$

where

$$\bar{F}_p(\bar{x}, p, \alpha) = \begin{bmatrix} 0 & \partial h(y,z,p)/\partial p \\ 0 & \partial g(y,z,p,\alpha)/\partial p \end{bmatrix}, \tag{3.15}$$

$$\bar{V}_j = \begin{bmatrix} 0 & 0 & 0 \\ \partial g(s_j^y, s_j^z, p, \alpha)/\partial y & \partial g(s_j^y, s_j^z, p, \alpha)/\partial z & \partial g(s_j^y, s_j^z, p, \alpha)/\partial p \end{bmatrix}, \tag{3.16}$$

and $\bar{W}(\bar{x}, p, \alpha)$ is given in (3.9). The DAE sensitivities have the initial conditions as

$$\bar{X}_j = \bar{X}(t_j) = \begin{bmatrix} I_1 & 0 & 0 \\ 0 & I_2 & 0 \end{bmatrix},$$

where $I_1 \in \mathbb{R}^{n_y \times n_y}$ and $I_2 \in \mathbb{R}^{n_z \times n_z}$ are both the identity matrices. The initial conditions here are also consistent.

The inputs to the integration of the DAE sensitivities are the relaxed linear DAEs equation (3.14), the time interval $[t_j, t_{j+1}]$, the initial condition $\bar{x}_j$, the parameter p, the matrix $\bar{M}$, the Jacobian (3.9) of the DAEs and the relaxation term $\bar{V}_j$. The integration of the DAE sensitivities needs to be done together with the DAE integration in Algorithm 3.1 where the stage vectors $\hat{x}_{j,i}$, the Jacobian $\bar{W}_j$, the coefficient matrix $\bar{M} - \bar{\gamma}\delta_j\bar{W}_j$ are reused.

Then the approximate solution of the DAE sensitivities at time t_{j+1} in time interval

$[t_j, t_{j+1}]$, $j = 1, 2, \ldots, N-1$ is obtained by using Algorithm 3.2.

Algorithm 3.2: Single step integration of the DAE sensitivities from t_j to t_{j+1} using ROW method.

Input: Initial time t_j, terminal time t_{j+1}, initial condition of the DAEs $\bar{x}_j = [s_j^{yT}, s_j^{zT}]^T$, parameters p, DAE sensitivity relaxations $\bar{V}_j$ and intermediate variables from Algorithm 3.1.

Output: The approximate solution of the DAE sensitivities of equation (3.14) $\bar{X}_{j+1}$ at time t_{j+1}.

Set $\bar{W}_j = \bar{W}(\bar{x}_j, p, \alpha)$, $\delta_j = t_{j+1} - t_j$, and

$$\bar{X}_j = \bar{X}(t_j) = \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \end{bmatrix}.$$

for $i = 1, 2, \ldots, m$ **do**

Set the values for the DAE sensitivities at stage i as

$$\hat{X}_{j,i} = \bar{X}_j + \delta_j \sum_{l=1}^{i-1} \bar{\alpha}_{i,l} \bar{K}_l, (\hat{X}_{j,1} = \bar{X}_j) \tag{3.17}$$

Solve the linear system by LU decomposition

$$(\bar{M} - \bar{\gamma}\delta_j \bar{W}_j)\bar{K}_i = \bar{W}(\hat{x}_{j,i}, p, \alpha)\hat{X}_{j,i} + \bar{F}_p(\hat{x}_{j,i}, p, \alpha) + \delta_j \bar{W}_j \sum_{l=1}^{i-1} \bar{\gamma}_{i,l} \bar{K}_l - \bar{V}_j. \tag{3.18}$$

end

Set the output values

$$\bar{X}_{j+1} = \bar{X}_j + \delta_j \sum_{i=1}^{m} \bar{b}_i \bar{K}_i. \tag{3.19}$$

3.1.4 Integration using `ode15s`

This section presents the integration using the MATLAB built-in DAE solver `ode15s` which implements the backward differentiation formulas (BDFs, also known as Gear's method, see Shampine et al. [1999]) to solve the index-1 DAEs. The inputs to this solver are the

DAE equation (3.8), the time interval $[t_j, t_{j+1}]$, the initial condition $\bar{x}_j$ and the parameter p. We use the functional options of the `ode15s` to set the mass matrix $\bar{M}$, and the Jacobian matrix of the DAEs to improve the efficiency. We also want the solution at the mid-point of the time interval to estimate the residual error of the DAE solution. So we specify the time interval as $[t_j, \frac{t_j+t_{j+1}}{2}, t_{j+1}]$ to get the dense output solution at the mid-point of the time interval as $[s^y_{midpoint}, s^z_{midpoint}]$. Then an estimate of the error in the residual can be obtained using the formula

$$\bar{e}_j = g(s^y_{midpoint}, s^z_{midpoint}, p, \alpha),\ j = 1, 2, \ldots, N-1. \tag{3.20}$$

This residual error estimate $\bar{e}_j$ is used in the mesh refinement strategy described in the latter part of the chapter.

The linear DAE (3.14) for trajectory sensitivities is also integrated by using `ode15s`. However, while using `ode15s`, the input to the function must be written in the form of a column vector. This leads to a reshape from the matrix form to the vector form of the DAE sensitivities $\bar{X}_j$ to meet the requirement of the input format of `ode15s`. First, for the sensitivity variable $\bar{X}(t)$,

$$\begin{aligned}
\bar{X}(t) &= \begin{bmatrix} \partial\bar{x}/\partial s^y_j & \partial\bar{x}/\partial s^z_j & \partial\bar{x}/\partial p \end{bmatrix} \\
&= \begin{bmatrix} \partial y/\partial s^y_j & \partial y/\partial s^z_j & \partial y/\partial p \\ \partial z/\partial s^y_j & \partial z/\partial s^z_j & \partial z/\partial p \end{bmatrix} \\
&= \begin{bmatrix}
\frac{\partial y_1}{\partial s^y_1} & \cdots & \frac{\partial y_1}{\partial s^y_{n_y}} & \frac{\partial y_1}{\partial s^z_1} & \cdots & \frac{\partial y_1}{\partial s^z_{n_z}} & \frac{\partial y_1}{p_1} & \cdots & \frac{\partial y_1}{\partial p_{n_p}} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
\frac{\partial y_{n_y}}{\partial s^y_1} & \cdots & \frac{\partial y_{n_y}}{\partial s^y_{n_y}} & \frac{\partial y_{n_y}}{\partial s^z_1} & \cdots & \frac{\partial y_{n_y}}{\partial s^z_{n_z}} & \frac{\partial y_{n_y}}{\partial p_1} & \cdots & \frac{\partial y_{n_y}}{\partial p_{n_p}} \\
\frac{\partial z_1}{\partial s^y_1} & \cdots & \frac{\partial z_1}{\partial s^y_{n_y}} & \frac{\partial z_1}{\partial s^z_1} & \cdots & \frac{\partial z_1}{\partial s^z_{n_z}} & \frac{\partial z_1}{p_1} & \cdots & \frac{\partial z_1}{\partial p_{n_p}} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
\frac{\partial z_{n_z}}{\partial s^y_1} & \cdots & \frac{\partial z_{n_z}}{\partial s^y_{n_y}} & \frac{\partial z_{n_z}}{\partial s^z_1} & \cdots & \frac{\partial z_{n_z}}{\partial s^z_{n_z}} & \frac{\partial z_{n_z}}{\partial p_1} & \cdots & \frac{\partial z_{n_z}}{\partial p_{n_p}}
\end{bmatrix} \in \mathbb{R}^{(n_y+n_z)\times(n_y+n_z+n_p)},
\end{aligned}$$

MATLAB has the built-in function `reshape`(MATLAB [2017]) which can reshape the array into the desired shape. Using the `reshape` as $\tilde{X}(t) = \texttt{reshape}(\bar{X}(t))$, the result column vector $\tilde{X}(t)$ becomes $\tilde{X}(t) = [\frac{\partial y_1}{\partial s^y_1}, \ldots, \frac{\partial y_{n_y}}{\partial s^y_1}, \frac{\partial z_1}{\partial s^y_1}, \ldots, \frac{\partial z_{n_z}}{\partial s^y_1}, \frac{\partial y_1}{\partial s^y_2}, \ldots, \frac{\partial y_{n_y}}{\partial s^y_2}, \frac{\partial z_1}{\partial s^y_2}, \ldots, \frac{\partial z_{n_z}}{\partial s^y_2}, \ldots, \frac{\partial y_1}{\partial s^y_{n_y}}, \ldots, \frac{\partial y_{n_y}}{\partial s^y_{n_y}}, \frac{\partial z_1}{\partial s^y_{n_y}}, \ldots, \frac{\partial z_{n_z}}{\partial s^y_{n_y}}]^T \in \mathbb{R}^{(n_y+n_z)(n_y+n_z+n_p)}$.

As the sensitivities are in the column vector form, the transformed Jacobian matrix becomes a large block diagonal matrix with the number of $(n_y + n_z + n_p)$ Jacobian matrices $\bar{W}(\bar{x}, p, \alpha)$ on the diagonal as

$$\tilde{W}(\bar{x}, p, \alpha) = \mathrm{diag}(\bar{W}(\bar{x}, p, \alpha)) = \begin{pmatrix} \bar{W} & & \\ & \ddots & \\ & & \bar{W} \end{pmatrix}, \in \mathbb{R}^{(n_y+n_z)(n_y+n_z+n_p)\times(n_y+n_z)(n_y+n_z+n_p)} \tag{3.21}$$

$\bar{F}_p$ and $\bar{V}_j$ are reshaped in the same manner to retain the same linear DAEs as equation (3.14) as

$$\tilde{F}_p(\bar{x}, p, \alpha) = \begin{bmatrix} 0 \\ \texttt{reshape}(\partial h(y, z, p)/\partial p) \\ \texttt{reshape}(\partial g(y, z, p, \alpha)/\partial p) \end{bmatrix} \in \mathbb{R}^{(n_y+n_z)(n_y+n_z+n_p)},$$

$$\tilde{V}_j(\bar{x}, p, \alpha) = \begin{bmatrix} 0 \\ \texttt{reshape}(g(s_j^y, s_j^z, p, \alpha)/\partial y) \\ 0 \\ \texttt{reshape}(g(s_j^y, s_j^z, p, \alpha)/\partial z) \\ 0 \\ \texttt{reshape}(g(s_j^y, s_j^z, p, \alpha)/\partial p) \end{bmatrix} \in \mathbb{R}^{(n_y+n_z)(n_y+n_z+n_p)}.$$

As the Jacobian matrix (3.9) of the DAE sensitivities depends on the DAE variables, the integration of the DAEs and sensitivities need to be done together. Therefore, two DAE systems are combined together and a large DAE System is obtained as

$$\tilde{M} \begin{bmatrix} \dot{\bar{x}} \\ \dot{\tilde{X}} \end{bmatrix} = \begin{bmatrix} \bar{f}(\bar{x}, p, \alpha) \\ \tilde{W}(\bar{x}, p, \alpha)\tilde{X} \end{bmatrix} + \begin{bmatrix} 0 \\ \tilde{F}_p(\bar{x}, p, \alpha) \end{bmatrix} - \begin{bmatrix} \bar{v}_j \\ \tilde{V}_j \end{bmatrix}. \tag{3.22}$$

where $\tilde{M}$ is a large block diagonal matrix with $(n_y + n_z + n_p + 1)$ mass matrices $\bar{M}$ on the diagonal as

$$\tilde{M} = \mathrm{diag}(\bar{M}) = \begin{pmatrix} \bar{M} & & \\ & \ddots & \\ & & \bar{M} \end{pmatrix} \in \mathbb{R}^{(n_y+n_z)(n_y+n_z+n_p+1)\times(n_y+n_z)(n_y+n_z+n_p+1)} \tag{3.23}$$

Then, the approximate solution of the DAEs and DAE sensitivities at time t_{j+1} in time interval $[t_j, t_{j+1}]$, $j = 1, 2, \ldots, N-1$ can be obtained by using Algorithm 3.3.

Algorithm 3.3: Integration of the DAEs and the DAE sensitivities from t_j to t_{j+1} using `ode15s`

Input: Initial time t_j, terminal time t_{j+1}, initial condition of the DAEs $\bar{x}_j = [s_j^{yT}, s_j^{zT}]^T$, parameters p, DAE relaxations $\bar{v}_j$, and DAE sensitivity relaxations $\bar{V}_j$.

Output: The approximate solution of the DAEs (3.8) $\bar{x}_{j+1}$, and the approximate solution of the DAE sensitivities (3.14) $\bar{X}_{j+1}$ at time t_{j+1}.

Set $\bar{W}_j = \bar{W}(\bar{x}_j, p, \alpha)$ and

$$\bar{X}_j = \bar{X}(t_j) = \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \end{bmatrix}.$$

Reshape the DAE sensitivities and the other variables into the vector form $\tilde{X}_j = \texttt{reshape}(\bar{X}_j)$, $\tilde{F}_p(\bar{x}, p, \alpha) = \texttt{reshape}(\bar{F}_p(\bar{x}, p, \alpha))$, and $\tilde{V}_j = \texttt{reshape}(\bar{V}_j)$.

Generate the large block diagonal Jacobian matrix $\tilde{W}_j$

$$\tilde{W}_j = \text{diag}(\bar{W}_j). \tag{3.24}$$

Define the DAEs to be integrated as

$$\tilde{M} \begin{bmatrix} \dot{\bar{x}} \\ \dot{\tilde{X}} \end{bmatrix} = \begin{bmatrix} \bar{f}(\bar{x}, p, \alpha) \\ \tilde{W}(\bar{x}, p, \alpha)\tilde{X} \end{bmatrix} + \begin{bmatrix} 0 \\ \tilde{F}_p(\bar{x}, p, \alpha) \end{bmatrix} - \begin{bmatrix} \bar{v}_j \\ \tilde{V}_j \end{bmatrix}.$$

Use `ode15s` to solve the DAEs, where the API is

$$\texttt{options} = \texttt{odeset}(\text{'Mass'}, \tilde{M}, \text{'Jacobian'}, \tilde{W}_j),$$
$$[t, y] = \texttt{ode15s}(\texttt{odefun}, \texttt{tspan}, \texttt{y0}, \texttt{options}),$$

where `odefun` is the DAEs defined (3.24), `tspan` is $[t_j, \frac{t_j + t_{j+1}}{2}, t_{j+1}]$, `y0` is $[\bar{x}_j^T, \tilde{X}_j^T]^T$, and `options` indicate the mass matrix and the Jacobian matrix of the DAEs.

The last row of the output y is the concatenation of the approximate solution of the DAEs $\bar{x}_{j+1}$ and the DAE sensitivities $\tilde{X}_{j+1}$.

Those solutions are retrieved as

$$\bar{x}_{j+1} = y[\text{end}, 1 : n_y],$$
$$\tilde{X}_{j+1} = y[\text{end}, n_y + 1 : \text{end}],$$
$$\bar{X}_{j+1} = \texttt{reshape}(\tilde{X}_{j+1}).$$

3.1.5 Sequential Evaluation of the Residual Equation and the Jacobian

In section 3.1.2, we have established methods for computing the trajectories $\bar{x}_j$, and the trajectory sensitivities $\bar{X}_j$, $j = 2, 3, \ldots, N$. We first present the sequential methods for evaluating the residual equation (3.4) and the Jacobian (3.7). For the computation in each interval $[t_j, t_{j+1}]$, $j = 1, 2, 3, \ldots, N-1$, we only need the initial condition s_j at time t_j,the parameters p and the relaxations $\bar{v}_j$ and $\bar{V}_j$ to integrate the DAEs. It's easy to come up with the idea to perform the integration from the first time interval $[t_1, t_2]$ sequentially to the last time interval $[t_{N-1}, t_N]$. Therefore, we can obtain the trajectories $\bar{x}_j$, and the trajectory sensitivities $\bar{X}_j$ at every time node t_j, $j = 2, 3, ..., N$.

3.1.6 Parallel evaluation of the residual equation and the Jacobian

After establishing the sequential methods for computing the trajectories $\bar{x}_j$, and the trajectory sensitivities $\bar{X}_j$, $j = 2, 3, \ldots, N$, we can now describe the method for the parallel construction of the residual equation (3.4) and the Jacobian (3.7).

It is easy to notice that in section 3.1.2, the integration done at any time interval $[t_j, t_{j+1}]$ is independent of the integration done in any other time interval $[t_i, t_{i+1}]$, $i \neq j$. Therefore, the integration in $[t_j, t_{j+1}$ can be performed simultaneously with the integration in t_i, t_{i+1}, $i \neq j$ with no need for communication of sharing data.

In this chapter, we develop a parallel multiple shooting method using MATLAB parallel computing toolbox which assumes that the availability of the multiple processors $P > 1$ and it shares the whole work required to integrate the DAEs (3.8) and the sensitivities (3.14). One approach that can be used to achieve this is to divide the $N-1$ time intervals into P partitions. Normally, P depends on the number of processors available in MATLAB local parallel pool. The MATLAB parallel computing toolbox has a function `parfor` (MATLAB [2017]) which executes for-loop iterations in parallel on workers in parallel pool and loop iterations are executed in parallel in a nondeterministic order. Notice that the integration within each interval in the algorithm 3.3 can be executed independently and in a nondeterministic order. So the parallel construction of the integration elements in the residual equation (3.4) and the sensitivity elements in the Jacobian matrix (3.7) can be realized

simply using the `parfor` in MATLAB parallel computing toolbox.

3.2 BABD Linear System Solution

As shown in section 3.1, the realization of the damped Newton's method to the residual equation (3.6) requires the solution of a BABD system of the form as

$$\begin{bmatrix} A_1 & C_1 & & & & H_1 \\ & A_2 & C_2 & & & H_2 \\ & & \ddots & \ddots & & \vdots \\ & & & A_{N-1} & C_{N-1} & H_{N-1} \\ B_1 & & & & B_N & H_N \end{bmatrix} \begin{bmatrix} \Delta s_1 \\ \Delta s_2 \\ \vdots \\ \Delta s_{N-1} \\ \Delta s_N \\ \Delta p \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{N-1} \\ b_N \\ b_p \end{bmatrix}, \tag{3.25}$$

where $A_j, C_j \in \mathbb{R}^{n_{\Delta s} \times n_{\Delta s}}$, $H_j \in \mathbb{R}^{n_{\Delta s} \times n_p}$, $j = 1, 2, ..., N-1$; $b_j, \Delta s_j \in \mathbb{R}^{n_{\Delta s}}$, $j = 1, 2, ..., N$; $\Delta p, b_p \in \mathbb{R}^{n_p}$, $B_1, B_N \in \mathbb{R}^{(n_{\Delta s}+n_p) \times (n_{\Delta s})}$, and $H_N \in \mathbb{R}^{(n_{\Delta s}+n_p) \times n_p}$, $\Delta s = n_y + n_z$. One thing to notice is that equation (3.25) is just an expansion of the equation (3.6).

Fabien [2014a] introduces a very robust and accurate method of solving this BABD system involves the reduction of the coefficient matrix into a sparse upper triangular matrix using orthogonal factorization. The description of the implementation used is presented below.

3.2.1 Sequential QR Factorization

Here, the notations of "block-row" and "block-column" are used when specifying the location of certain sub-matrix in the BABD system. For example, in the system (3.25), we say that A_1 is in the first block-row, and first block-column. Similarly, H_2 is in the second block-row and the $N+1$ block-column.

To describe the QR factorization algorithm, it's necessary to rearrange the equations where the A_1 and B_1 blocks appear in the penultimate block-column so that we can form

a upper-triangular matrix later. Then, equation (3.25) is rewritten as

$$\begin{bmatrix} C_1 & & & & & A_1 & H_1 \\ A_2 & & & & & & H_2 \\ & A_3 & C_3 & & & & H_3 \\ & & \ddots & \ddots & & & \vdots \\ & & & A_{N-1} & C_{N-1} & & H_{N-1} \\ & & & & B_N & B_1 & H_N \end{bmatrix} \begin{bmatrix} \Delta s_2 \\ \Delta s_3 \\ \Delta s_4 \\ \vdots \\ \Delta s_N \\ \Delta s_1 \\ \Delta p \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{N-1} \\ b_N \\ b_p \end{bmatrix}, \tag{3.26}$$

Notice that Δs_1 is also moved from the first block-row to the penultimate block-row to meet the consistency.

The serial QR factorization is presented below. Let $Q_1 \in \mathbb{R}^{2n_{\Delta s} \times 2n_{\Delta s}}$ be an orthogonal matrix such that

$$Q_1^T \begin{bmatrix} C_1 \\ A_2 \end{bmatrix} = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}, \tag{3.27}$$

where $R_1 \in \mathbb{R}^{n_{\Delta s} \times n_{\Delta s}}$ is a nonsingular upper triangular matrix and the matrix Q and R are obtained by performing a QR decomposition on matrix $[C_1^T \ A_2^T]^T$. Then, multiplying both sides of (3.26) by $(Q_1^T, \ I)$ where $I \in \mathbb{R}^{(N-1)n_{\Delta s}+n_p \times (N-1)n_{\Delta s}+n_p}$ yields

$$\begin{bmatrix} R_1 & E_1 & & & & G_1 & J_1 \\ & \tilde{C}_2 & & & & \tilde{G}_2 & \tilde{H}_2 \\ & A_3 & C_3 & & & & H_3 \\ & & \ddots & \ddots & & & \vdots \\ & & & A_{N-1} & C_{N-1} & & H_{N-1} \\ & & & & B_N & B_1 & H_N \end{bmatrix} \begin{bmatrix} \Delta s_2 \\ \Delta s_3 \\ \Delta s_4 \\ \vdots \\ \Delta s_N \\ \Delta s_1 \\ \Delta p \end{bmatrix} = \begin{bmatrix} d_1 \\ \tilde{b}_2 \\ b_3 \\ \vdots \\ b_{N-1} \\ b_N \\ b_p \end{bmatrix},$$

where

$$Q_1^T \begin{bmatrix} C_1 & 0 & A_1 & H_1 & b_1 \\ A_2 & C_2 & 0 & H_2 & b_2 \end{bmatrix} = \begin{bmatrix} R_1 & E_1 & G_1 & J_1 & d_1 \\ 0 & \tilde{C}_2 & \tilde{G}_2 & \tilde{H}_2 & \tilde{b}_2 \end{bmatrix}. \tag{3.28}$$

Hence, the first block-row of (3.26) is reduce to an upper triangular form.

The second block-row can also be reduced to upper triangular form by using the orthogonal factorization

$$Q_2^T \begin{bmatrix} \tilde{C}_2 \\ A_2 \end{bmatrix} = \begin{bmatrix} R_2 \\ 0 \end{bmatrix}. \tag{3.29}$$

Again, Q_2 is an orthogonal matrix and R_2 is upper triangular.

After applying this factorization to the first $N-2$ block-rows of (3.26), the system becomes

$$\begin{bmatrix} R_1 & E_1 & & & & G_1 & J_1 \\ & R_2 & E_2 & & & G_2 & J_2 \\ & & R_3 & E_3 & & G_3 & J_3 \\ & & & \ddots & \ddots & & \vdots \\ & & & & \tilde{C}_{N-1} & \tilde{G}_{N-1} & \tilde{H}_{N-1} \\ & & & & B_N & B_1 & H_N \end{bmatrix} \begin{bmatrix} \Delta s_2 \\ \Delta s_3 \\ \Delta s_4 \\ \vdots \\ \Delta s_N \\ \Delta s_1 \\ \Delta p \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ \tilde{b}_{N-1} \\ b_N \\ b_p \end{bmatrix},$$

Apply the factorization to the last two block-row yields the strictly upper triangular system

$$\begin{bmatrix} R_1 & E_1 & & & & G_1 & J_1 \\ & R_2 & E_2 & & & G_2 & J_2 \\ & & R_3 & E_3 & & G_3 & J_3 \\ & & & \ddots & \ddots & & \vdots \\ & & & & R_{N-1} & G_{N-1} & J_{N-1} \\ & & & & & R_N & J_N \\ & & & & & & R_p \end{bmatrix} \begin{bmatrix} \Delta s_2 \\ \Delta s_3 \\ \Delta s_4 \\ \vdots \\ \Delta s_N \\ \Delta s_1 \\ \Delta p \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ \tilde{b}_{N-1} \\ b_N \\ b_p \end{bmatrix}.$$

Here, $R_N \in \mathbb{R}^{n_{\Delta s} \times n_{\Delta s}}$, and $R_p \in \mathbb{R}^{n_p \times n_p}$, are upper triangular. The unknowns Δs_j, $j = 1, 2, ..., N$ and Δp can then be found by from the last block-row to the first block-row backwardly using the backward substitution by using the "mldivide" function "\" in MATLAB.

3.2.2 Parallel QR Factorization

The parallel QR factorization method for solving the BABD system is developed to make use of the fact that modern computers are equipped with more than 1 processors to execute the computations. The parallel factorization method takes advantage of the special structure of the BABD system that the first $N-1$ block rows of (3.25) can be divided into M ($1 \leq M \leq (N-1)/2$) partitions while each smaller partition can be simultaneously factored into a bordered upper triangular form on an individual CPU core. The resultant BABD system is composed of $M+1$ block-rows and can be solved using the sequential QR factorization described in section 3.2.1.

The parallel method is implemented in the way described as follows. Define $M+1$ indices r_j, $j = 0, 1, \ldots, M$ s.t. $0 = r_0 < r_1 < \cdots < r_M = N-1$, and $r_{j+1} \geq r_j + 2$. The block-rows associated with each partition are defined using those indices, where block-rows starting from $r_0 + 1 = 1$ to r_1 belong to the first partition and block-rows starting from $r_1 + 1$ to r_2 belong to the second partition.

Take the first partition as an example first as

$$\begin{bmatrix} A_1 & C_1 & & & & H_1 \\ & A_2 & C_2 & & & H_2 \\ & & \ddots & \ddots & & \vdots \\ & & & A_{r_1} & C_{r_1} & H_{r_1} \end{bmatrix} \begin{bmatrix} \Delta s_1 \\ \Delta s_2 \\ \vdots \\ \Delta s_{r_1+1} \\ \Delta p \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{r_1} \end{bmatrix}, \tag{3.30}$$

All the other partitions have the similar structure, where partition j starts at block-row $r_{\text{start}} = r_{j-1} + 1$ and ends at block-row $r_{\text{end}} = r_j$. The procedure to factor those partitions

into a boardered upper triangular form is presented in Algorithm 3.4.

Algorithm 3.4: Partition Factorization.

Input: Block-rows from $r_{\text{start}} = r_{j-1} + 1$ to $r_{\text{end}} = r_j$.

Output: Block matrices during the factorization, $G_{r_{\text{start}}}, \ldots, G_{r_{\text{end}}-1}$, $R_{r_{\text{start}}}, \ldots, R_{r_{\text{end}}-1}$, $E_{r_{\text{start}}}, \ldots, E_{r_{\text{end}}-1}$, $J_{r_{\text{start}}}, \ldots, J_{r_{\text{end}}-1}$, $d_{r_{\text{start}}}, \ldots, d_{r_{\text{end}}-1}$, $\tilde{A}_{r_{\text{end}}}, \tilde{C}_{r_{\text{end}}}, \tilde{H}_{\text{end}}$.

Set $\tilde{G}_{r_{\text{start}}} = A_{r_{\text{start}}}$, $\tilde{C}_{r_{\text{start}}} = C_{r_{\text{start}}}$, $\tilde{H}_{r_{\text{start}}} = H_{r_{\text{start}}}$, $\tilde{b}_{r_{\text{start}}} = b_{r_{\text{start}}}$. **for** $i = r_{start}, \ldots, r_{end} - 1$ **do**

Using householder QR factorization to solve

$$\begin{bmatrix} \tilde{C}_i \\ A_{i+1} \end{bmatrix} = Q_i \begin{bmatrix} R_i \\ 0 \end{bmatrix},$$

Compute

$$\begin{bmatrix} E_i & G_i & J_i & d_i \\ \tilde{C}_{i+1} & \tilde{G}_{i+1} & \tilde{H}_{i+1} & \tilde{b}_{i+1} \end{bmatrix} = Q_i^T \begin{bmatrix} 0 & \tilde{G}_i & \tilde{H}_i & \tilde{b}_i \\ C_{i+1} & 0 & H_{i+1} & b_{i+1} \end{bmatrix}.$$

end

Set $\tilde{A}_{r_{\text{end}}} = \tilde{G}_{r_{\text{end}}}$.

After applying Algorithm 3.4 to the each partition like (3.30), the following more compact bordered upper triangular system is obtained as

$$\begin{bmatrix} G_1 & R_1 & E_1 & & & J_1 \\ G_2 & & R_2 & E_2 & & J_2 \\ \vdots & & & \ddots & & \vdots \\ \tilde{A}_{r_1} & & & & \tilde{C}_{r_1} & \tilde{H}_{r_1} \end{bmatrix} \begin{bmatrix} \Delta s_1 \\ \Delta s_2 \\ \vdots \\ \Delta s_{r_1+1} \\ \Delta p \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ \tilde{b}_{r_1} \end{bmatrix}, \tag{3.31}$$

Then, the variables $\Delta s_2, \ldots, \Delta s_{r_1}$ can be obtained after the variables Δs_1, Δs_{r_1+1}, and Δp are solved, with the following formula,

$$\Delta s_j = R_{j-1}^{-1}(d_j - G_{j-1}\Delta s_{r_{\text{start}}} - E_{j-1}\Delta s_{j+1} - J_{j-1}\Delta p)$$

In this system, the variables Δs_1, Δs_{r_1+1}, and Δp are so called *boundary variables*, while the variables $\Delta s_2, \ldots, \Delta s_{r_1}$ are called *internal variables.* The procedure to recover the solution

to the partiton like (3.31) is presented in Algorithm 3.5.

Algorithm 3.5: Partition Substitution.

Input: Indices r_{start} and r_{end}, $\Delta s_{r_{\text{start}}}$, $\Delta s_{r_{\text{end}}+1}$, and Δp, $G_{r_{\text{start}}}, \ldots, G_{r_{\text{end}}-1}$, $R_{r_{\text{start}}}, \ldots, R_{r_{\text{end}}-1}$, $E_{r_{\text{start}}}, \ldots, E_{r_{\text{end}}-1}$, $J_{r_{\text{start}}}, \ldots, J_{r_{\text{end}}-1}$, $d_{r_{\text{start}}}, \ldots, d_{r_{\text{end}}-1}$.

output: $\Delta s_{r_{\text{start}}+1}, \ldots, \Delta s_{r_{\text{end}}}$.

for $i = r_{end}, r_{end} - 1, \ldots, r_{start} + 1$ **do**

$$\Delta s_j = R_{j-1}^{-1}(d_j - G_{j-1}\Delta s_{r_{\text{start}}} - E_{j-1}\Delta s_{j+1} - J_{j-1}\Delta p)$$

end

Apply Algorithm 3.4 to all the M partitions gives M reduced systems as (3.31). Expand the last block-row of the reduced partition, we have the equation of

$$\tilde{A}_{r_{\text{start}}}\Delta s_{r_{\text{start}}} + \tilde{C}_{r_{\text{start}}}\Delta s_{r_{\text{end}+1}} + \tilde{H}_{r_{\text{start}}}\Delta p = \tilde{b}_{r_{\text{start}}}. \tag{3.32}$$

Collect the M equations like (3.32) and assemble them together gives the reduced BABD system as

$$\begin{bmatrix} \tilde{A}_{r_1} & \tilde{C}_{r_1} & & & & \tilde{H}_{r_1} \\ & \tilde{A}_{r_2} & \tilde{C}_{r_2} & & & \tilde{H}_{r_2} \\ & & \ddots & \ddots & & \\ & & & & & \\ B_1 & & & & B_N & H_N \end{bmatrix} \begin{bmatrix} \Delta s_1 \\ \Delta s_{r_1+1} \\ \vdots \\ \Delta s_{r_M+1} \\ \Delta p \end{bmatrix} = \begin{bmatrix} \tilde{b}_{r_1} \\ \tilde{b}_{r_2} \\ \vdots \\ \tilde{b}_{r_M} \\ b_N \\ b_p \end{bmatrix}, \tag{3.33}$$

The system (3.33) contains the boundary variables $\Delta s_1, \Delta s_{r_1+1}, \Delta s_{r_M+1}, \Delta p$ for all the M partitions. Once those boundary variables are obtained by solving the system, all the interval variables $\Delta s_2, \ldots, \Delta s_{r_1}, \Delta s_{r_1+2}, \ldots, \Delta s_{r_2}, \ldots$ can be computed by using Algorithm 3.5.

We can notice that the reduced BABD system (3.33) is of the exact same form as the original BABD system (3.25). However, the reduced BABD system is only equipped with $M + 1$ block-rows instead of N block-rows. Therefore, we can solve the reduced BABD system (3.33) using the sequential QR solver described in section 3.2.1 with much less computation effort.

For the parallel QR solver to solve the BABD system (3.25), we can first apply Algorithm 3.4 simultaneously to the M partitions. Then, solve the reduced BABD system (3.33) sequentially and recover all the necessary internal variables via Algorithm 3.5 for all M partitions at the same time.

3.2.3 Mesh Refinement

The BVP-DAEs (2.38)-(2.40) is solved primarily on a fixed mesh. However, in some cases we may need to remesh the problem. The remesh criteria is based on the local truncation error (3.13) or the residual error (3.20) computed with the corresponding integrator in each time interval. The error is denoted as $\bar{\epsilon}_j$, $j = 2, 3, \cdots, N$. One remesh criteria is when $\bar{\epsilon}_j$ exceeds the desired numerical tolerance after the convergence of the Newton's iteration. Another criteria is when Newton's iteration fails to find a descent direction during the line-search.

The remesh policy is that if $\bar{\epsilon}_j > \epsilon$, where ϵ is the desired numerical tolerance, the mesh interval $[t_j, t_{j+1}]$ is subdivided by adding one time node to the middle of the time interval or if $\bar{\epsilon}_j > 100\epsilon$, the mesh interval is subdivided by adding three uniformly spaced nodes to the time interval. Linear interpolation is used to estimate the solution at the newly created nodes. Also, if $\bar{\epsilon}_j < \frac{\epsilon}{100}$, we evaluate the residual errors of the 4 subsequent nodes $\bar{\epsilon}_{j+i}$, $i = 1, 2, 3, 4$ together and if $\bar{\epsilon}_{j+i} < \frac{\epsilon}{100}$, $i = 1, 2, 3, 4$, we delete the time nodes t_{j+1} and t_{j+3}. The remesh policy is sequentially applied on each time interval. The implementation limits the number of nodes that can be added, the minimum number of nodes allowed, and the number of mesh refinements allowed.

3.3 Continuation Method

The continuation method used to solve the BVP-DAEs (2.38)-(2.40) is given as follows. The algorithm described above is used to solve the residual equation (3.4) for a sequence of decreasing parameters α. Starting with an initial estimate $(s^{(0)}, p^{(0)})$ and continuation parameter $\alpha^0 > 0$, the algorithm solves (3.4) to obtain $(\tilde{s}, \tilde{p})$ that satisfies $\left\|F(\tilde{s}, \tilde{p}, \alpha^0)\right\| \leq \epsilon$, where ϵ is the desired convergence tolerance. The solution $(\tilde{s}, \tilde{p})$ is then used as the initial estimate for the next continuation iteration where α is decreased by a factor β which is

usually between 0.6 and 0.9. The algorithm terminates when $\alpha \leq \alpha_m$, where α_m is the desired continuation parameter termination value.

3.4 Implementation and Evaluation

Algorithms for solving the optimal control problems OCP (2.7)-(2.12) are implemented using MATLAB. The main functionality provided by these codes is a set of routes that solve the BVP-DAEs problems of the form (2.38)-(2.40). These codes are directly implemented from all the algorithms described above in this chapter. All the implementations of the parallel codes are standalone. The usability of the solver is enhanced by the symbolic functionality in MATLAB to translate the problem of OCPs into MATLAB functions that can be used by the solver.

The solver is composed of two main parts as: construction of the residual with DAEs integration and the Jacobian matrix with the sensitivity integration in section 3.1.2, and BABD linear system solving in section 3.2. Both the sequential and parallel versions of these two parts are implemented to compare the efficiency and adaptation to the solver. The following terminologies are used in the forthcoming analysis. $Solver_1$ denotes the solver implemented with the sequential implementation of the construction of the residual with DAEs integration and the Jacobian matrix with the sensitivity integration, and sequential implementation of the BABD linear system solving; $Solver_2$ denotes the solver with the parallel implementation of the construction of the residual with DAEs integration and the Jacobian matrix with the sensitivity integration, and sequential implementation of the BABD linear system solving; $Solver_3$ denotes the solver implemented with both algorithms in parallel. $N_{residual}$ and $T_{residual}$ denote the overall calling times and the running time of the construction of the residual with DAEs integration; $N_{sensitivity}$ and $T_{sensitivity}$ denote the overall calling times and the running time of the construction of the Jacobian matrix with the DAE sensitivities integration; N_{BABD} and T_{BABD} denote the overall calling times and the running time of the construction of the BABD linear system solving. T_{all} denotes the overall computation time of the whole problem and P denotes the number of processors used.

The solver is tested over 120 examples, while the four examples presented are the typical

and complicated ones to show the robustness and efficiency of the solver. All the examples are tested with both solvers implemented with the `ode15s` integrator the and integrator with ROW method.

Parallelism. The parallel code in algorithms are implemented using the MATLAB parallel computing toolbox with `parfor` (parallel for loop) and `spmd` (single program multiple data) MATLAB [2017]. The MATLAB parallel pool instantiates the thread for each parallel process and these threads are executed concurrently. Execution of statements following the parallel workers are blocked until all threads are completed.

Computing environment. All the numerical results given below are performed on a desktop with the following hardware and software characteristics. The CPU is an Intel(R) Xeon(R) CPU E5-2640 @ 2.60GHz and the system has 64.0 GB memory. The operating system is Windows 7. The MATLAB version is 2017b. All the codes of the solver can be obtained at `https://github.com/UW-OCP/mps_solver_MATLAB`.

Example 3.4.1. This is a problem of a continuous stirred-tank chemical reactor from Kirk [2012] (pp. 405 and 406). The state equations for a continuous stirred-tank chemical reactor are given below. The flow of a coolant through a coil inserted in the reactor is to control the first-order, irreversible exothermic reaction taking place in the reactor.

$$\min_{x(t)\in\mathbb{R}^2} \int_0^{t_\mathrm{f}} [x_1(t)^2 + x_2(t)^2]\, dt,$$

subject to

$$\begin{aligned} \dot{x_1}(t) &= -2.0a_1(t) + a_4(t) - a_1(t)u(t), \\ \dot{x_2}(t) &= 0.5 - x_2(t) - a_4(t). \end{aligned}$$

where $a_1(t) = x_1(t) + 0.25$, $a_2(t) = x_2(t) + 0.5$, $a_3(t) = x_1(t) + 2.0$ and
$a_4(t) = a_2(t)\exp[\frac{25.0x_1(t)}{a_3(t)}]$.

The desired objective is to maintain the temperature and concentration close to their steady-state values. The initial constraint is that $\Gamma = [x_1(0) - 0.05, x_2(0)] = 0$ and the final constraint is that $\Psi = [x_1(t_\mathrm{f}), x_2(t_\mathrm{f})] = 0$. The control variable inequality constraints are bounded as $|u(t)| \leq 1$, $t \in [0, t_f]$. Also the final time is $t_\mathrm{f} = 0.78$.

The initial estimate for the two state variables are straight lines with boundary conditions at both endpoints. The control is estimating with a quadratic function with initial and final zero value. The estimate is obtained on a uniform mesh with $N = 101$ nodes.

Using this initial estimate, we obtain the solutions for the differential and algebraic variables that are shown in figure 3.2, figure 3.3, and figure 3.4. The convergence tolerance for this problem is $\epsilon = 10^{-6}$. Both `ode15s` and ROW integrators are abole to solve this example with the desired tolerance. The final mesh of this problem has $N = 101$ nodes with `ode15s` and $N = 430$ nodes with ROW integrator.

Table 3.1 and 3.2 show the calling times and running time of each part using both `ode15s` and ROW method as the integrator of the algorithm. It also shows the speedup factor which is defined as $S = \frac{T_1}{T_P}$.

The results show that for the construction of the residual with DAEs integration and the Jacobian matrix with the sensitivity integration with integrator `ode15s` is much slower than ROW method in sequential. However, there is a significant improvement in speed of the parallel implementation of the algorithm using `ode15s`. As shown in the table 3.2, there is a decrease in speed of the parallel implementation of the algorithm using ROW method. That's because the overhead in setting up the parallel worker takes too much time compared with the computation time used in integration for each function call in MATLAB. Also, it can be concluded from the results of $solver_3$ of the two tables that the parallel implementation of the BABD linear system solving using the `spmd` decreases the speed a lot which is because the data distribution between the parallel workers of MATLAB takes too much time compared with the sequential linear system solving process.

Example 3.4.2. This problem considers an underwater vehicle problem from aquanautics, where a model for the control of an underwater vehicle is given with ten state variables and four control variables from Büskens and Maurer [2000] (pp. 99 to pp. 106). The problem is tested by the algorithms without using the penalty function method (see (2.1)-(2.6)) on the CVICs, and algorithm using the the penalty function method (see (2.7)-(2.12)). The algorithm without the penalty function was stuck and unable to converge with the warning from MATLAB saying that the condition number of the Jacobian matrix of the DAEs is

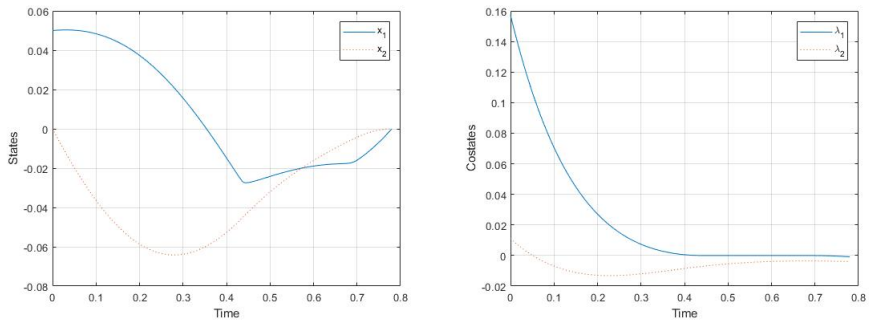


Figure 3.2: Optimal solution of states and costates for Example 3.4.1.

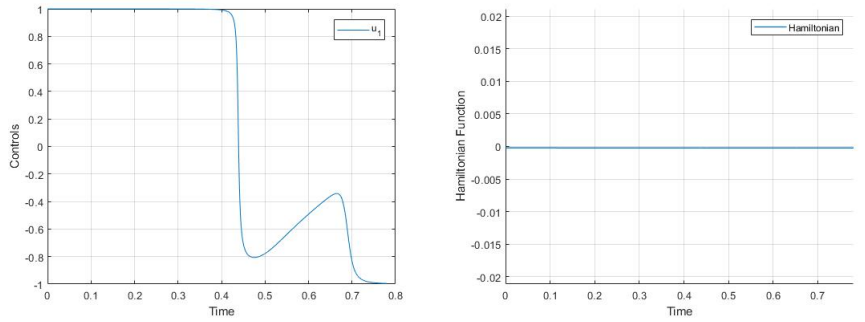


Figure 3.3: Optimal solution of control and Hamiltonian function for Example 3.4.1.

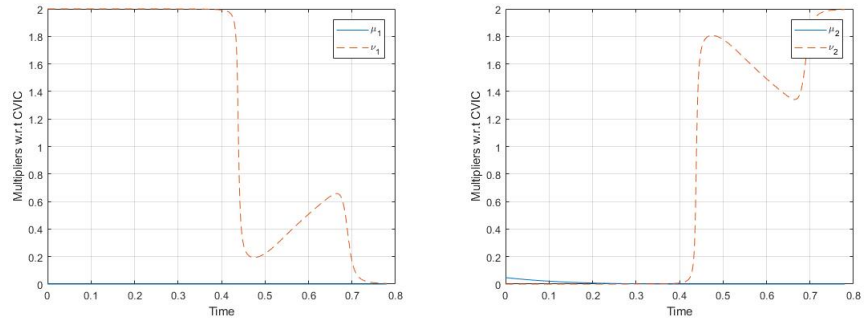


Figure 3.4: Optimal solution of multipliers for Example 3.4.1.

	$Solver_1$	$Solver_2$		$Solver_3$	
P	1	4	16	4	16
$N_{residual}$	1147	1147	1147	1147	1147
$N_{sensitivity}$	627	627	627	627	627
N_{BABD}	494	494	494	494	494
$T_{residual}(s)$	256.8	150.1	107.7	160.8	110.2
$T_{sensitivity}(s)$	1079.6	301.2	165.0	300.5	167.7
$T_{BABD}(s)$	4.6	5.9	4.9	247.3	652.4
$T_{total}(s)$	1343.4	479.8	316.3	732.1	971.7
Speedup		2.8	4.2	1.8	1.4

Table 3.1: Example 3.4.1, Evaluating times, running time and speedup factor with `ode15s`

	$Solver_1$	$Solver_2$		$Solver_3$	
P	1	4	16	4	16
$N_{residual}$	749	749	749	749	749
$N_{sensitivity}$	403	403	403	403	403
N_{BABD}	270	270	270	270	270
$T_{residual}(s)$	27.3	80.4	81.1	80.3	86.4
$T_{sensitivity}(s)$	53.8	90.4	88.0	97.4	95.5
$T_{BABD}(s)$	6.9	6.9	6.4	257.3	385.8
$T_{total}(s)$	89.3	205.0	205.8	466.9	600.0
Speedup		0.44	0.43	0.2	0.15

Table 3.2: Example 3.4.1, Evaluating times, running time and speedup factor with ROW integrator

too large. The problem wants to determine the minimum energy control $u(t)$, $t \in [0,1]$ that minimizes the energy functional

$$\min_{u_i \in \mathbb{R}} \int_0^1 \sum_{i=1}^{4} u_i(t)^2 \, dt,$$

subject to

$$\begin{aligned}
\dot{x_1} &= \cos(x_6)\cos(x_5)x_7 + r_x, \\
\dot{x_2} &= \sin(x_6)\cos(x_5)x_7, \\
\dot{x_3} &= -\sin(x_5)x_7 + r_z, \\
\dot{x_4} &= x_8 + \sin(x_4)\tan(x_5)x_9 + \cos(x_4)\tan(x_5)x_{10}, \\
\dot{x_5} &= \cos(x_4)x_9 - \sin(x_4)x_{10}, \\
\dot{x_6} &= \frac{\sin(x_4)}{\cos(x_5)}x_9 + \frac{\cos(x_4)}{\cos(x_5)}x_{10}, \\
\dot{x_7} &= u_1, \\
\dot{x_8} &= u_2, \\
\dot{x_9} &= u_3, \\
\dot{x_{10}} &= u_4.
\end{aligned}$$

The variables $x_1(t), x_2(t), x_3(t)$ specify the position of the center of the vehicle, while $x_4(t), x_5(t), x_6(t)$ describe the orientation of the mass by Euler angles. The vehicle is assumed to be moving with velocity $x_7(t)$ and angular velocities $x_8(t) - x_{10}(t)$. Hence, the control variable $u_1(t)$ represents the acceleration of the underwater vehicle, while $u_2(t) - u_4(t)$ describe the angular accelerations. The nonlinear currents are modeled by

$$\begin{aligned}
r_x &= -u_{x_{max}} e^{-((x_1 - c_x)/r_x)^2}(x_1 - c_x)(\frac{x_3 - c_z}{c_z})^2, \\
r_z &= -u_{z_{max}} e^{-((x_1 - c_x)/r_x)^2}(\frac{x_3 - c_z}{c_z})^2,
\end{aligned}$$

with constants $u_{x_{max}} = 2$ representing the maximal horizontal current, $u_{z_{max}} = 1$ representing the maximal vertical current, $c_x = 0.5$ representing the center of current (center of underwater ditch), $c_z = 0.1$ representing the depth of zero current, $r_x = 0.1$ representing the factor for the expansion of the current. The objective is to find the control

inputs $u_i(t)$, $i = 1, 2, 3, 4$ that are required to move the vehicle from the initial constraint $\Gamma = [x_1(0), x_2(0), x_3(0) - 0.2, x_4(0) - \frac{\pi}{2}, x_5(0) - 0.1, x_6(0) + \frac{\pi}{4}, x_7(0) - 1, x_8(0), x_9(0) - 0.5, x_{10}(0) - 0.1] = 0$ to the final constraint $\Psi = [x_1(1) - 1, x_2(1) - 0.5, x_3(1), x_4(1) - \frac{\pi}{2}, x_5(1), x_6(1), x_7(1), x_8(1), x_9(1), x_{10}(1)] = 0$, while minimizing the cost functional. There are also eight CVICs bounded as follows, $|u_i(t)| \leq 15$, $i = 1, 2, 3, 4$, $t \in [0, 1]$.

The initial estimate for this problem is obtained by solving an unconstrained version of the problem and with no bounds on the control inputs. The estimate is obtained on a uniform mesh with $N = 101$ nodes.

Using this initial estimate, both algorithms with `ode15s` and ROW integrators are performed. However, the algorithm with `ode15s` fails to converge to a solution. With ROW method, we obtain the solutions for the differential and algebraic variables that are shown in figure 3.5, figure 3.6 and figure 3.7. The final mesh of this problem has $N = 407$ nodes. The convergence tolerance for this problem is $\epsilon = 10^{-6}$.

From table 3.3, we can see that the parallel implementation of the construction of the residual with DAEs integration and the Jacobian matrix with the sensitivity integration using ROW method is faster than the sequential solver with a speedup factor 1.94 (when $P = 4$) and 1.75 (when $P = 16$). The reason why there is speedup in parallel compared with example 1 is that this problem has more state variables and the system is much bigger. Therefore, the time for performing the computation of the integration is in a bigger portion compared with the overhead on setting up the parallel thread. When the number of workers are too many (like when $P = 16$), this overhead still delays the computation process as we see there's a decrease in the speedup factor with more processors. And the parallel implementation of the BABD linear system solving is still decreasing the speed in this example.

3.5 Conclusion

This chapter presents a multiple shooting method to solve the boundary value problem involving index-1 differential-algebraic equations which are resultant from the necessary conditions of the optimum for the optimal control problem. A damped Newton's method is used to solve the residual equation by collecting the local residual on each time node after

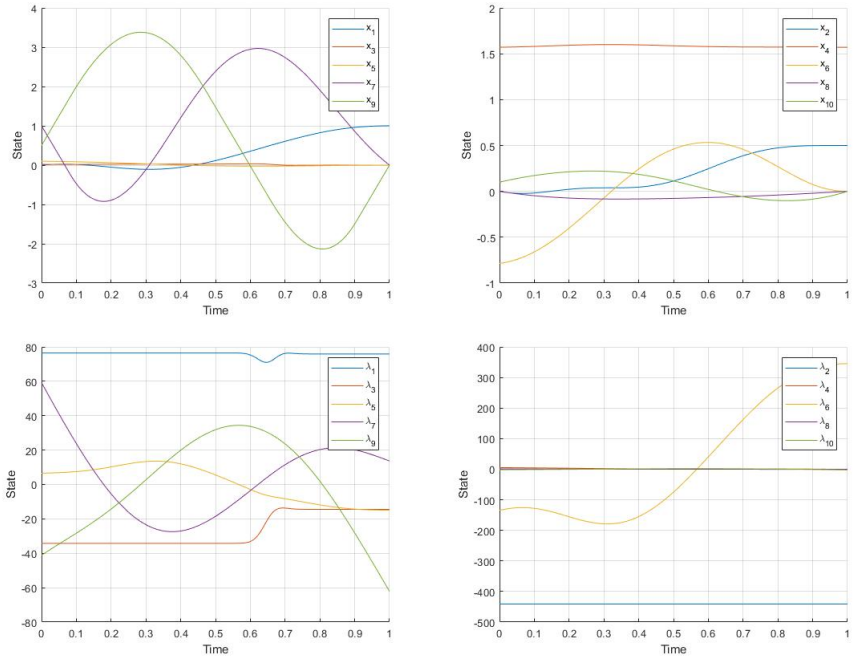


Figure 3.5: Optimal solution of states and costates for Example 3.4.2.

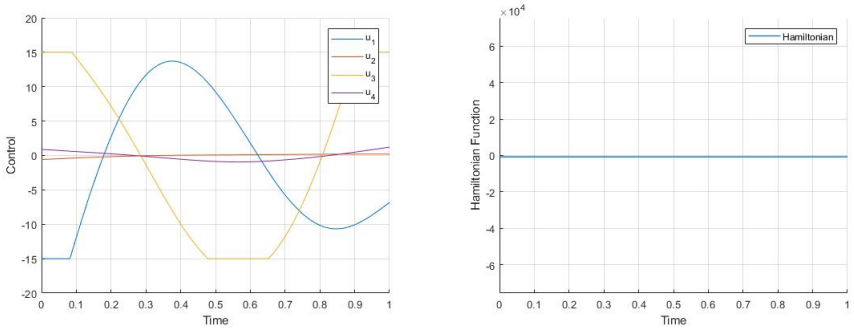


Figure 3.6: Optimal solution of controls and Hamiltonian function for Example 3.4.2.

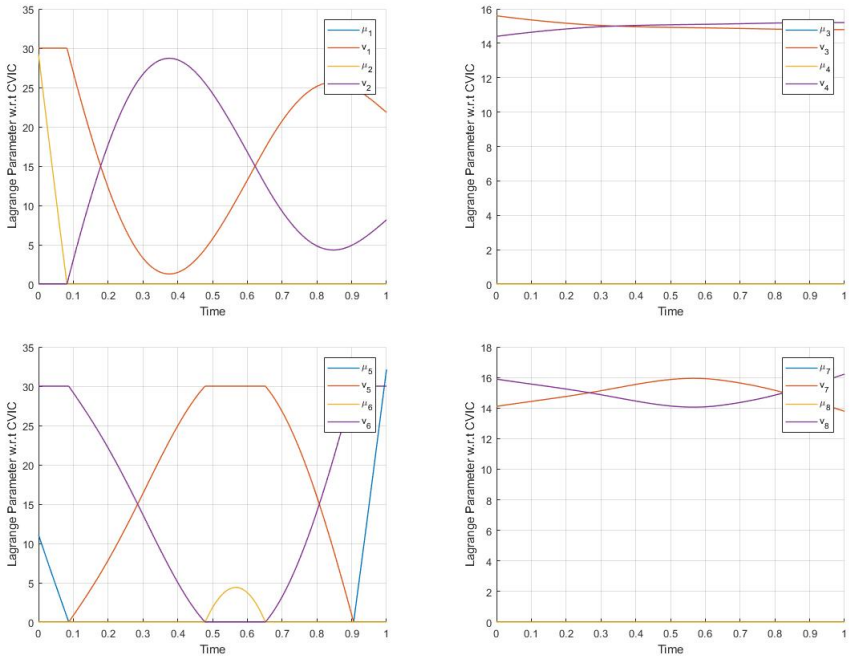


Figure 3.7: Optimal solution of multipliers for Example 3.4.2.

	$Solver_1$	$Solver_2$		$Solver_3$	
P	1	4	16	4	16
$N_{residual}$	845	845	845	845	845
$N_{sensitivity}$	451	451	451	451	451
N_{BABD}	318	318	318	318	318
$T_{residual}(s)$	93.7	78.0	105.9	79.4	106.1
$T_{sensitivity}(s)$	554.3	177.8	140.6	180.6	143.5
$T_{BABD}(s)$	112.5	112.4	115.2	348.1	524.5
$T_{total}(s)$	764.8	393.5	409.1	634.6	820.8
Speedup		1.94	1.75	1.21	0.93

Table 3.3: Example 3.4.2, Calling times,running time and speedup factor with ROW integrator

discretization and a continuation method is used to deal with the inequality constraints approximated by the Kanzow's smoothed Fisher-Burmeister formula.

A numerical solver is implemented using MATLAB which provides a built-in DAE integrator `ode15s` also with the use of its parallel computing capability. The algorithm implements the part that can be processed independently in parallel and decrease the running time for solving the problem significantly. A comparison of the efficiency and robustness between `ode15s` and the integrator implemented using Runge-Kutta (ROW) method is also shown by the examples in this chapter. With the increase of the computing ability of the computer and the increase of the number of processors used to solve the problem, a further reduced running time is also possible.

As shown in section 3.4, with a lot of tests performed by both solvers with `ode15s` and ROW integrator, it can be concluded that the integrator of `ode15s` is not suitable nor efficient to the multiple shooting algorithm developed. Comparing the performance of the sequential and parallel implementation of different solvers, it can be seen there is a big speedup in the parallel implementation with `parfor` of the construction of the the Jacobian matrix with the sensitivity integration. The integration with `ode15s` can give a better precision but is more time consuming, so the speedup is more remarkable.

The integration with ROW method is faster both in sequential and parallel solvers. However, there is some overhead in the set-up with `parfor` that slows down the solving process as seen from Example 3.4.1. For some complicated problems with lots of variables and inequality constraints, the parallel algorithm is more efficient. However, the performance of the parallel implementation using `spmd` of the BABD linear system solving is not satisfactory. Because `spmd` is a form of distributed memory computing, the overhead in setting up the workers and transferring data from local client to different workers in MATLAB is too costly compared with doing all the work on the local client using shared memory. So, we suggest to use the combination of the parallel implementation of the construction of the residual with DAEs integration and the Jacobian matrix with the sensitivity integration in section 3.1.2, and a sequential implementation of the BABD linear system solving in section 3.2 with the implemented code in MATLAB.

Chapter 4

COLLOCATION ALGORITHM

This chapter introduces another method called collocation method (Alexander [1990]) to solve the BVP-DAEs (2.38)-(2.40).

4.1 Collocation Method

With the algebraic equations, the continuity conditions of the differential variables, and the boundary conditions, the BVP-DAEs defined by (2.38)-(2.40) can be represented in the form of a residual equation as

$$\tilde{F}(q,\alpha) = \begin{bmatrix} h(y(t),z(t),p) - \dot{y}(t) \\ g(y(t),z(t),p,\alpha) \\ y(t) - y(0) - \int_{t_\mathrm{i}}^{t} \dot{y}(t)dt \\ r(y(t_\mathrm{i}),y(t_\mathrm{f}),p) \end{bmatrix} = 0, \tag{4.1}$$

where $q = [y(t)^T,\ \dot{y}(t)^T,\ z(t)^T,\ p^T]^T \in \mathbb{Q}$ with $\mathbb{Q} = \mathbb{W}^{1,\infty}([t_\mathrm{i},t_\mathrm{f}],\mathbb{R}^{n_y}) \times \mathbb{L}^{\infty}([t_\mathrm{i},t_\mathrm{f}],\mathbb{R}^{n_y}) \times \mathbb{L}^{\infty}([t_\mathrm{i},t_\mathrm{f}],\mathbb{R}^{n_z}) \times \mathbb{R}^{n_p}$, and $F : \mathbb{Q} \times \mathbb{R} \to \mathbb{S}$, with $\mathbb{S} = \mathbb{L}^{\infty}([t_\mathrm{i},t_\mathrm{f}],\mathbb{R}^{n_y}) \times \mathbb{L}^{\infty}([t_\mathrm{i},t_\mathrm{f}],\mathbb{R}^{n_z}) \times \mathbb{W}^{1,\infty}([t_\mathrm{i},t_\mathrm{f}],\mathbb{R}^{n_y}) \times \mathbb{R}^{n_y+n_p}$.

The collocation method used in this thesis requests the overall time interval $[t_\mathrm{i},t_\mathrm{f}]$ be discretized into a mesh with N time nodes such that $t_\mathrm{i} = t_1 < t_2 < \ldots < t_N = t_\mathrm{f}$. Each divided time interval $\mathcal{S}_j = [t_j, t_{j+1}]$, $j = 1,2,\ldots,N-1$ is further divided by m_j collocation points, where each collocation point is obtained by $\tau_{j,l} = t_j + c_l^{(j)}\delta_j$, $l = 1,2,\ldots,m_j$, $\delta_j = t_{j+1} - t_j$ and $0 \le c_1^{(j)} < c_2^{(j)} < \cdots < c_{m_{j-1}}^{(j)} < c_{m_j}^{(j)} \le 1$. A global unified number of collocation points m_j is used in this chapter where $m_1 = m_2 = \ldots = m_j$, while in a collocation with adaptive mesh refinement ability, this condition is not necessary. The coefficients c_i used in the implementation of this thesis are from the Lobatto IIIA implicit Runge-Kutta method (Alexander [1990] pp. 211-214).

In each time interval $\mathcal{S}_j = [t_j, t_{j+1}]$, using the collocation method, the derivative at any time in this interval is approximated using the Lagrange polynomial and derivatives at the m_j collocation points as

$$\dot{\tilde{y}}^{(j)}(t) = \sum_{l=1}^{m_j} L_l^{(j)}\left(\frac{t-t_j}{\delta_j}\right)\ \dot{\tilde{y}}_l^{(j)},\ t \in [t_j, t_{j+1}], \tag{4.2}$$

where

$$L_l^{(j)}(\varsigma) = \prod_{k=1, k\neq l}^{m_j} \frac{\varsigma - c_k^{(j)}}{c_l^{(j)} - c_k^{(j)}} \tag{4.3}$$

is a basis of Lagrange polynomials and $\dot{\tilde{y}}_l^{(j)} = \dot{\tilde{y}}^{(j)}(\tau_{j,l}) \in \mathbb{R}^{n_y}$.

From equation (4.2), within each time interval $\mathcal{S}_j$, the differential variables $\tilde{y}^{(j)}(t)$ can be approximated with

$$\tilde{y}^{(j)}(t) = \tilde{y}_j + \delta_j \sum_{l=1}^{m_j} I_l^{(j)}\left(\frac{t-t_j}{\delta_j}\right)\ \dot{\tilde{y}}_l^{(j)},\ t \in [t_j, t_{j+1}]. \tag{4.4}$$

where

$$I_l^{(j)}(\varsigma) = \int_0^{\varsigma} L_l^{(j)}(\zeta)\, d\zeta \tag{4.5}$$

and $\tilde{y}_j = \tilde{y}(t_j) \in \mathbb{R}^{n_y}$ represent the differential variables at t_j.

The differential variables at certain collocation points in $\mathcal{S}_j$ can be obtained as

$$\tilde{y}_l^{(j)} = \tilde{y}^{(j)}(\tau_{j,l}) = \tilde{y}_j + \delta_j \sum_{k=1}^{m_j} a_{lk}\, \dot{\tilde{y}}_k^{(j)},\ l = 1, \ldots, m_j, \tag{4.6}$$

$$\tilde{y}_{j+1} = \tilde{y}^{(j)}(t_{j+1}) = \tilde{y}_j + \delta_j \sum_{l=1}^{m_j} b_l\, \dot{\tilde{y}}_l^{(j)}, \tag{4.7}$$

where

$$a_{lk} = \int_0^{c_l^{(j)}} L_k^{(j)}(\zeta)\, d\zeta,\ b_l = \int_0^1 L_l^{(j)}(\zeta)\, d\zeta, \tag{4.8}$$

are the coefficients associated with the specific m_j-stage Lobatto IIIA implicit Runge-Kutta method used and can be pre-computed before the method is applied to a certain problem.

The algebraic variables $\tilde{z}(t)$ in time interval $\mathcal{S}_j$ can also be approximated with the $m-1$ degree Lagrange polynomial as

$$\tilde{z}^{(j)}(t) = \sum_{l=1}^{m_j} L_l^{(j)}\left(\frac{t-t_j}{\delta_j}\right)\ \tilde{z}_l^{(j)},\ t \in [t_j, t_{j+1}]. \tag{4.9}$$

where $\tilde{z}_l^{(j)} = \tilde{z}(\tau_{j,l}) \in \mathbb{R}^{n_z}$ represent the algebraic variables at each collocation point in $\mathcal{S}_j$.

By requiring the collocation and algebraic conditions be satisfied for all the mesh intervals together using those approximation equations, a residual equation can be formulated as

$$\tilde{F}(\tilde{q},\alpha) = \begin{bmatrix} h(\tilde{y}_1^{(1)}, \tilde{z}_1^{(1)}, \tilde{p}) - \dot{\tilde{y}}_1^{(1)} \\ g(\tilde{y}_1^{(1)}, \tilde{z}_1^{(1)}, \tilde{p}, \alpha) \\ \vdots \\ h(\tilde{y}_{m_1}^{(1)}, \tilde{z}_{m_1}^{(1)}, \tilde{p}) - \dot{\tilde{y}}_{m_1}^{(1)} \\ g(\tilde{y}_{m_1}^{(1)}, \tilde{z}_{m_1}^{(1)}, \tilde{p}, \alpha) \\ y_2 - y_1 - \delta_1 \sum_{l=1}^{m_1} b_l \ \dot{\tilde{y}}_l^{(1)} \\ \vdots \\ h(\tilde{y}_i^{(j)}, \tilde{z}_i^{(j)}, \tilde{p}) - \dot{\tilde{y}}_i^{(j)} \\ g(\tilde{y}_i^{(j)}, \tilde{z}_i^{(j)}, \tilde{p}, \alpha) \\ y_{j+1} - y_j - \delta_j \sum_{l=1}^{m_j} b_l \ \dot{\tilde{y}}_l^{(j)} \\ \vdots \\ r(\tilde{y}_1, \tilde{y}_N, \tilde{p}) \end{bmatrix} = 0, \quad \tilde{q} = \begin{bmatrix} \tilde{y}_1 \\ \vdots \\ \tilde{y}_j \\ \dot{\tilde{y}}_i^{(j)} \\ \tilde{z}_i^{(j)} \\ \vdots \\ \tilde{y}_N \\ \tilde{p} \end{bmatrix}, \tag{4.10}$$

where $j = 1, 2, \ldots, N-1$, $i = 1, 2, \ldots, m_j$. Moreover, $\tilde{q} \in \tilde{\mathbb{Q}} = \mathbb{R}^{n_{\tilde{q}}}$ with $n_{\tilde{q}} = Nn_y + \sum_{j=1}^{N-1} m_j(n_y + n_z) + n_p$ and $F(\tilde{q},\alpha) \in \tilde{\mathbb{S}} = \tilde{\mathbb{Q}}$.

Therefore, the BVP-DAEs is transformed into the problem of solving the residual equation $\tilde{F}(\tilde{q},\alpha) = 0$, respectively, where $\tilde{q}$ yields the approximate solution of the OCP. It is shown in [Ascher, 1989, Theorem 18] [Fabien, 2016a, Theorem 2] that let $y(t)$, $z(t)$, and p be the true solution to the BVP-DAEs (2.38)-(2.40), then the collocation solution has consistency of order m_j as $\left\|y^{(j)}(t) - \tilde{y}^{(j)}(t)\right\| = \mathcal{O}(\delta_j^m)$, $\left\|z^{(j)}(t) - \tilde{z}^{(j)}(t)\right\| = \mathcal{O}(\delta_j^m)$, $j = 1, \ldots, N-1$.

4.1.1 Solution of the residual equation

A damped Newton's method (see Fabien [2016b]) is used here to solve the residual equation (4.10), which is similar to the one developed in Section 3.1.1.

Suppose an initial guess $\tilde{q}^k$ which is sufficiently close to the solution of the residual

equation (4.10) is provided, a closer solution $\tilde{q}^{k+1}$ can be obtained from

$$\tilde{q}^{k+1} = \tilde{q}^k + \xi \Delta \tilde{q}^k, \ k = 0, 1, \ldots, \tag{4.11}$$

$$D\tilde{F}(\tilde{q}^k, \alpha)\ \Delta \tilde{q}^k = -\tilde{F}(\tilde{q}^k, \alpha), \tag{4.12}$$

where the stepsize factor ξ is obtained by using the classic line-search method and the Jacobian is

$$D\tilde{F}(\tilde{q}^k, \alpha) = \begin{bmatrix} J_1 & W_1 & 0 & & & & V_1 \\ -I & -D_1 & I & & & & 0 \\ & J_2 & W_2 & 0 & & & V_2 \\ & -I & -D_2 & I & & & 0 \\ & & \ddots & & & & \vdots \\ & & & J_{N-1} & W_{N-1} & 0 & V_{N-1} \\ & & & -I & -D_{N-1} & I & 0 \\ B_1 & & & & & B_N & V_N \end{bmatrix}, \tag{4.13}$$

with

$$\Delta\tilde{q} = \begin{bmatrix} \Delta\tilde{y}_1 \\ \Delta\tilde{k}_1 \\ \vdots \\ \Delta\tilde{y}_j \\ \Delta\tilde{k}_j \\ \vdots \\ \Delta\tilde{y}_{N-1} \\ \Delta\tilde{k}_{N-1} \\ \Delta\tilde{y}_N \\ \Delta\tilde{p} \end{bmatrix}, \ \Delta\tilde{k}_j = \begin{bmatrix} \Delta\dot{\tilde{y}}_1^{(j)} \\ \Delta\tilde{z}_1^{(j)} \\ \vdots \\ \Delta\dot{\tilde{y}}_i^{(j)} \\ \Delta\tilde{z}_i^{(j)} \\ \vdots \\ \Delta\dot{\tilde{y}}_{m_j}^{(j)} \\ \Delta\tilde{z}_{m_j}^{(j)} \end{bmatrix}, \ \tilde{F}(\tilde{q}^k, \alpha) = \begin{bmatrix} \tilde{f}_1^a \\ \tilde{f}_1^b \\ \vdots \\ \tilde{f}_j^a \\ \tilde{f}_j^b \\ \vdots \\ \tilde{f}_{N-1}^a \\ \tilde{f}_{N-1}^b \\ \tilde{f}_N \end{bmatrix}, \quad \tilde{f}_j^a = \begin{bmatrix} \tilde{h}(\tilde{y}_1^{(j)}, \tilde{z}_1^{(j)}, \tilde{p}) - \dot{\tilde{y}}_1^{(j)} \\ \tilde{g}(\tilde{y}_1^{(j)}, \tilde{z}_1^{(j)}, \tilde{p}, \alpha) \\ \vdots \\ \tilde{h}(\tilde{y}_i^{(j)}, \tilde{z}_i^{(j)}, \tilde{p}) - \dot{\tilde{y}}_i^{(j)} \\ \tilde{g}(\tilde{y}_i^{(j)}, \tilde{z}_i^{(j)}, \tilde{p}, \alpha) \\ \vdots \\ \tilde{h}(\tilde{y}_{m_j}^{(j)}, \tilde{z}_m^{(j)}, \tilde{p}) - \dot{\tilde{y}}_m^{(j)} \\ \tilde{g}(\tilde{y}_{m_j}^{(j)}, \tilde{z}_m^{(j)}, \tilde{p}, \alpha) \end{bmatrix}, \quad \tilde{f}_j^b = \left[\tilde{y}_{j+1} - \tilde{y}_j - \delta_j \sum_{l=1}^{m} b_l \dot{\tilde{y}}_l^{(j)} \right], \tag{4.14}$$

$$J_j = \begin{bmatrix} \tilde{h}^y_{j1} \\ \tilde{g}^y_{j1} \\ \vdots \\ \tilde{h}^y_{ji} \\ \tilde{g}^y_{ji} \\ \vdots \\ \tilde{h}^y_{jm} \\ \tilde{g}^y_{jm} \end{bmatrix},\ V_j = \begin{bmatrix} \tilde{h}^p_{j1} \\ \tilde{g}^p_{j1} \\ \vdots \\ \tilde{h}^p_{ji} \\ \tilde{g}^p_{ji} \\ \vdots \\ \tilde{h}^p_{jm} \\ \tilde{g}^p_{jm} \end{bmatrix}, \begin{aligned} \tilde{h}^y_{ji} &= \partial\tilde{h}(\tilde{y}^{(j)}_i, \tilde{z}^{(j)}_i, \tilde{p})/\partial y, \\ \tilde{h}^z_{ji} &= \partial\tilde{h}(\tilde{y}^{(j)}_i, \tilde{z}^{(j)}_i, \tilde{p})/\partial z, \\ \tilde{h}^p_{ji} &= \partial\tilde{h}(\tilde{y}^{(j)}_i, \tilde{z}^{(j)}_i, \tilde{p})/\partial p, \\ \tilde{g}^y_{ji} &= \partial\tilde{g}(\tilde{y}^{(j)}_i, \tilde{z}^{(j)}_i, \tilde{p})/\partial y, \\ \tilde{g}^z_{ji} &= \partial\tilde{g}(\tilde{y}^{(j)}_i, \tilde{z}^{(j)}_i, \tilde{p})/\partial z, \\ \tilde{g}^p_{ji} &= \partial\tilde{g}(\tilde{y}^{(j)}_i, \tilde{z}^{(j)}_i, \tilde{p})/\partial p, \end{aligned}$$

$$W_j = \begin{bmatrix} -I + \delta_j a_{11}\tilde{h}^y_{j1} & \tilde{h}^z_{j1} & \delta_j a_{12}\tilde{h}^y_{j1} & 0 & \cdots & \delta_j a_{1m}\tilde{h}^y_{j1} & 0 \\ \delta_j a_{11}\tilde{g}^y_{j1} & \tilde{g}^z_{j1} & \delta_j a_{12}\tilde{g}^y_{j1} & 0 & \cdots & \delta_j a_{1m}\tilde{g}^y_{j1} & 0 \\ \delta_j a_{21}\tilde{h}^y_{j2} & 0 & -I + \delta_j a_{22}\tilde{h}^y_{j2} & \tilde{h}^z_{j2} & \cdots & \delta_j a_{2m}\tilde{h}^y_{j2} & 0 \\ \delta_j a_{21}\tilde{g}^y_{j2} & 0 & \delta_j a_{22}\tilde{g}^y_{j2} & \tilde{g}^z_{j2} & \cdots & \delta_j a_{1m}\tilde{g}^y_{j1} & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \delta_j a_{m1}\tilde{h}^y_{jm} & 0 & \delta_j a_{m2}\tilde{h}^y_{jm} & 0 & \cdots & -I + \delta_j a_{mm}\tilde{h}^y_{jm} & \tilde{h}^z_{jm} \\ \delta_j a_{m1}\tilde{g}^y_{jm} & 0 & \delta_j a_{m2}\tilde{g}^y_{jm} & 0 & \cdots & \delta_j a_{mm}\tilde{g}^y_{jm} & \tilde{g}^z_{jm} \end{bmatrix},$$

$$D_j = \delta_j \begin{bmatrix} b_1 I & 0 & b_2 I & 0 & \cdots & b_m I & 0 \end{bmatrix},$$

$$B_1 = \partial\tilde{r}(\tilde{y}_1, \tilde{y}_N, \tilde{p})/\partial\tilde{y}_1,\ B_N = \partial\tilde{r}(\tilde{y}_1, \tilde{y}_N, \tilde{p})/\partial\tilde{y}_N,\ V_N = \partial\tilde{r}(\tilde{y}_1, \tilde{y}_N, \tilde{p})/\partial\tilde{p},$$

$$\Delta\tilde{k}_j, \tilde{f}^a_j \in \mathbb{R}^{m(n_y+n_z)},\ \tilde{f}^b_j \in \mathbb{R}^{n_y}, J_j \in \mathbb{R}^{m(n_y+n_z)\times n_y},\ V_j \in \mathbb{R}^{m(n_y+n_z)\times n_p},$$

$$W_j \in \mathbb{R}^{m(n_y+n_z)\times m(n_y+n_z)},\ D_j \in \mathbb{R}^{n_y\times m(n_y+n_z)},$$

$$j = 1, 2, \ldots, N-1,\ i = 1, 2, \ldots, m,\ \tilde{f}_N = r(\tilde{y}_1, \tilde{y}_N, \tilde{p}) \in \mathbb{R}^{(n_y+n_p)}.$$

Expanding equation (4.12) gives that for $j = 1, 2, \ldots, N-1$ we have

$$J_j \Delta\tilde{y}_j + W_j \Delta\tilde{k}_j + V_j \Delta\tilde{p} = -\tilde{f}_j^a, \tag{4.15}$$

$$-\Delta\tilde{y}_j - D_j \Delta\tilde{k}_j + \tilde{y}_{j+1} = -\tilde{f}_j^b. \tag{4.16}$$

Since the DAEs are index-1, which implies that g_{ji}^z is nonsingular so that for sufficiently small δ_j, the matrix W_j is nonsingular also. Using equation (4.15) to eliminate $\tilde{k}_j$, we get

$$\Delta\tilde{k}_j = W_j^{-1}(-\tilde{f}_j^a - J_j \Delta\tilde{y}_j - V_j \Delta\tilde{p}). \tag{4.17}$$

Substitute this result into equation (4.16) gives that

$$(-I + D_j W_j^{-1} J_j)\Delta\tilde{y}_j + \Delta\tilde{y}_{j+1} + D_j W_j^{-1} V_j \Delta\tilde{p} = -\tilde{f}_j^b - D_j W_j^{-1} \tilde{f}_j^a.$$

Using this elimination approach, the original BABD system (4.13) can be reduced to a compact upper triangular BABD system shown below as

$$\begin{bmatrix} A_1 & C_1 & & & & H_1 \\ & A_2 & C_2 & & & H_2 \\ & & \ddots & \ddots & & \vdots \\ & & & A_{N-1} & C_{N-1} & H_{N-1} \\ B_1 & & & & B_N & H_N \end{bmatrix} \begin{bmatrix} \Delta\tilde{y}_1 \\ \Delta\tilde{y}_2 \\ \vdots \\ \Delta\tilde{y}_{N-1} \\ \Delta\tilde{y}_N \\ \Delta\tilde{p} \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \vdots \\ \tilde{b}_{N-1} \\ \tilde{b}_N \end{bmatrix}, \tag{4.18}$$

where $A_j = -I + D_j W_j^{-1} J_j \in \mathbb{R}^{n_y \times n_y}$, $C_j = I \in \mathbb{R}^{n_y \times n_y}$, $H_j = D_j W_j^{-1} V_j \in \mathbb{R}^{n_y \times n_p}$, $\tilde{b}_j = -\tilde{f}_j^b - D_j W_j^{-1} \tilde{f}_j^a \in \mathbb{R}^{n_y}$, $H_N = V_N$, $\tilde{b}_N = -\tilde{f}_N$, $j = 1, 2, \ldots, N-1$.

Once a solution to (4.18) is found, the updates $\Delta\tilde{k}_j$ can be obtained via equation (4.17). The updates $\Delta\dot{\tilde{y}}_i^{(j)}$ and $\Delta\tilde{z}_i^{(j)}$ can be determined from $\Delta\tilde{k}_j$, $j = 1, 2, \ldots, N-1$ and $i = 1, 2, \ldots, m_j$. This reduced BABD system (3.33) can be solved by the parallel QR reduction algorithm presented in section 3.2.

4.2 Collocation Algorithm Evaluation

4.2.1 Generation of the initial input

Normally for the BVP-DAEs (2.38)-(2.40), the necessary variables are the values of the differential variables $\tilde{y}_j$, algebraic variables $\tilde{z}_j$, and parameter variables $\tilde{p}$ except the values

of the derivatives of the differential variables $\dot{\tilde{y}}_{ji}$. Therefore, the input to the BVP-DAEs is usually in the form of $s_0 = [y_1^T, z_1^T, \ldots, y_j^T, z_j^T, \ldots, y_N^T, z_N^T]^T,\ j = 1, 2, \ldots, N$. In each time interval $\mathcal{S}_j$, the derivatives of the differential variables $\dot{\tilde{y}}_i^{(j)}$ for the initial guess of the algorithm at each collocation point $\tau_{j,i}, i = 1, 2, \ldots, m_j$ are generated using equation (2.38). The inputs to the equation are the differential variables and algebraic variables at each collocation point which are obtained using the linear interpolation with the variables at time nodes t_j and t_{j+1}. The approach to generating the initial input $\tilde{q}^0$ is given in Algorithm 4.1.

Algorithm 4.1: Generation of the initial input $\tilde{q}^0$

Input: $y_j, z_j, p,\ j = 1, 2, \ldots, N$.

Output: $\tilde{y}_j, \dot{\tilde{y}}_i^{(j)}, \tilde{z}_i^{(j)}, \tilde{y}_N, \tilde{p},\ j = 1, 2, \ldots, N-1;\ i = 1, 2, \ldots, m_j$.

for $j = 1, 2, \ldots, N-1$ **do**

  $\tilde{y}_j = y_j$

  for $i = 1, 2, \ldots, m_j$ **do**

    $y_i^{(j)} = (1 - c_i)y_j + c_i y_{j+1}$,

    $z_i^{(j)} = (1 - c_i)z_j + c_i z_{j+1}$,

    Using equation (2.38), $\dot{\tilde{y}}_i^{(j)} = h(y_i^{(j)}, z_i^{(j)}, p)$,

    $\tilde{z}_i^{(j)} = z_i^{(j)}$.

  end

end

$\tilde{y}_N = y_N$,

$\tilde{p} = p$.

4.2.2 Evaluation of the residual equation and the Jacobian

The residual $\tilde{F}(\tilde{q}^k, \alpha)$ (4.14) is made up by the residual terms $[\tilde{f}_j^{aT}\ \tilde{f}_j^{bT}]^T, j = 1, 2, \ldots, N-1$ in each time interval $[t_j, t_{j+1}]$. So is the Jacobian $D\tilde{F}(\tilde{q}^k, \alpha)$ (3.7) which is made up by the matrix elements J_j, V_j, D_j and W_j from each time interval. The computation of these terms require the differential variables $\tilde{y}_i^{(j)}$ at each collocation point which can be obtained by using the equation (4.6). After obtaining $\tilde{y}_i^{(j)}$, the construction of the residual equation and the Jacobian can be easily realized by performing the necessary computations with all the necessary inputs computed.

The approaches to constructing the residual $\tilde{F}(\tilde{q},\alpha)$ and its Jacobian $D\tilde{F}(\tilde{q},\alpha)$ can be achieved as described in Algorithm 4.2 and Algorithm 4.3.

Algorithm 4.2: Construction of $\tilde{F}(\tilde{q},\alpha)$

Input: $\tilde{y}_j, \dot{\tilde{y}}_i^{(j)}, \tilde{z}_i^{(j)}, \tilde{y}_N, \tilde{p}, \alpha$, $j = 1,2,\ldots,N-1$, $i = 1,2,\ldots,m_j$.

Output: The elements $\tilde{f}_j^a, \tilde{f}_j^b$, and $\tilde{f}_N$, in the residual $\tilde{F}(\tilde{q},\alpha)$ and the differential variables $\tilde{y}_i^{(j)}$ at the collocation points $\tau_{j,i}$.

for $j = 1,2,\ldots,N-1$ **do**

Evaluate

$$\tilde{f}_j^b = y_{j+1} - y_j - \delta_j \sum_{l=1}^{m} b_l \dot{\tilde{y}}_l^{(j)}$$

for $i = 1,2,\ldots,m_j$ **do**

Using equation (4.6), evaluate

$$\tilde{y}_i^{(j)} = \tilde{y}_j + \delta_j \sum_{l=1}^{m_j} a_{il}\ \dot{\tilde{y}}_l^{(j)},$$

Evaluate $h(\tilde{y}_i^{(j)}, \tilde{z}_i^{(j)}, \tilde{p}), g(\tilde{y}_i^{(j)}, \tilde{z}_i^{(j)}, \tilde{p}, \alpha)$ and construct the corresponding ith block-row term in $\tilde{f}_j^a$.

end

end

Compute $\tilde{f}_N$.

Algorithm 4.3: Construction of $D\tilde{F}(\tilde{q},\alpha)$

Input: $\tilde{y}_i^{(j)}, \tilde{z}_i^{(j)}, \tilde{y}_1, \tilde{y}_N, \tilde{p}, \alpha$, $j = 1,2,\ldots,N-1$, $i = 1,2,\ldots,m_j$.

Output: The elements J_j, V_j, D_j, W_j, $j = 1,2,\ldots,N-1$ and B_1, B_N, V_N in the Jacobian $D\tilde{F}(\tilde{q},\alpha)$.

for $j = 1,2,\ldots,N-1$ **do**

for $i = 1,2,\ldots,m_j$ **do**

Evaluate $\tilde{h}_{ji}^y, \tilde{h}_{ji}^z, \tilde{h}_{ji}^p, \tilde{g}_{ji}^y, \tilde{g}_{ji}^z, \tilde{g}_{ji}^p$ and construct the corresponding ith row or column block in J_j, V_j, D_j and W_j.

end

end

Compute B_1, B_N and V_N.

4.2.3 Parallel evaluation of the residual Equation and the Jacobian

From the evaluation of the residual equation (4.14) in Algorithm 4.2, we can easily notice that the evaluation of the residual in each time interval $[\tilde{f}_j^{aT}\ \tilde{f}_j^{bT}]^T, j = 1, 2, \ldots, N-1$ is independent of the evaluation done in any other time interval. So the residual in the interval $[t_j, t_{j+1}]$ can be computed with the residual in any other interval $[t_i, t_{i+1}], i \neq j$ at the same time with no need for communication of data between them.

The same is true for computing elements J_j, V_j, D_j and $W_j,\ j = 1, 2, \ldots, N-1$ of the Jacobian $D\tilde{F}(\tilde{q}, \alpha)$ (3.7) in Algorithm 4.3. For the computation of the elements in the interval $[t_j, t_{j+1}]$, they can be performed independently and simultaneously without sharing data.

In this chapter, a parallel collocation method taking advantage of MATLAB parallel toolbox (MATLAB [2017]) is developed which assumes the computer is equipped with multiple computing cores. To illustrate these parallel computing algorithms, we let $P > 1$ represent the number of available cores that can share the whole computation work required. Normally, P depends on the number of cores available in MATLAB local parallel worker pool. The MATLAB parallel computing toolbox has a function `parfor` (parallel for loop) MATLAB [2017] which executes for-loop iterations in parallel and loop iterations are executed in parallel in a nondeterministic order. Since there is no data communication between the computation performed in each time interval, the order of executing the computation in each time interval is trivial. Then, the parallel execution for (i) the computation of the block residual equation $\tilde{F}(\tilde{q}, \alpha)$ in Algorithm 4.2; (ii) the computation of the Jacobian $D\tilde{F}(\tilde{q}, \alpha)$ in Algorithm 4.3 can be realized simply using the `parfor` in MATLAB which partitions the time intervals into P groups as uniform as possible and execute these work in an nondeterministic order.

4.2.4 Reduction and recover of the linear system

The original BABD system (3.7) is first reduced into a compact upper triangular BABD system (3.33) and the resultant system is solved by the QR factorization discussed in details in section 3.2.

The elements $A_j, C_j, H_j, \tilde{b}_j,\ j = 1, 2, \ldots, N-1$, and $H_N, \tilde{b}_N$ are computed using LU factorization. The results from LU factorization of W_j are saved for later use when recovering the updates $\Delta\tilde{k}_j$ of the original system using forward and backward substitution. The details of the reduction and the recover of the linear system are shown in Algorithm 4.4 and

Algorithm 4.5.

Algorithm 4.4: Reduction of the Jacobian

Input: $\tilde{f}_j^a, \tilde{f}_j^b, J_j, V_j, D_j, W_j$, $j = 1, 2, \ldots, N-1$ and $\tilde{f}_N, B_1, B_N, V_N$, $j = 1, 2, \ldots, N-1$.

Output: $A_j, C_j, H_j, \tilde{b}_j$, $j = 1, 2, \ldots, N-1$, and $H_N, \tilde{b}_N$.

for $j = 1, 2, \ldots, N-1$ **do**

Factor W_j using LU factorization and save the factorization results. Compute the following elements using forward and backward substitution using the LU factorization results,

$$A_j = -I + D_j W_j^{-1} J_j,$$
$$C_j = I,$$
$$H_j = D_j W_j^{-1} V_j,$$
$$\tilde{b}_j = -\tilde{f}_j^b - D_j W_j^{-1} \tilde{f}_j^a.$$

end

Compute $H_N = V_N$, $\tilde{b}_N = -\tilde{f}_N$,.

Algorithm 4.5: Recover of the updates $\Delta\tilde{k}_j$

Input: Saved LU factorization results of W_j, $\tilde{f}_j^a, J_j, V_j$, and the solution from the reduced BABD system $\Delta\tilde{y}_j, \Delta\tilde{p}$, $j = 1, 2, \ldots, N-1$.

Output: Solution to the original BABD system $\Delta\tilde{k}_j$.

for $j = 1, 2, \ldots, N-1$ **do**

Use the forward and backward substitution with the saved LU factorization results of W_j to solve the equation

$$\Delta\tilde{k}_j = W_j^{-1}(-\tilde{f}_j^a - J_j\Delta\tilde{y}_j - V_j\Delta\tilde{p}).$$

end

4.2.5 Continuation method

The continuation method used in this chapter is the same as the one developed in Section 3.3. The residual equation (4.10) is first solved from an initial guess $\tilde{q}^{(0)}$ and continuation

parameter α_0. The algorithm then uses Newton's method to solve the residual (4.10) to obtain $\tilde{q}$ which satisfies $\|F(\tilde{q}, \alpha)\| \leq \epsilon$, where ϵ is the desired convergence tolerance for the numerical iteration. After the converged Newton's iteration, the continuation parameter α is decreased by a scale factor β which is less than 1 and the converged solution $\tilde{q}$ is used as the initial guess for the next problem with the decreased continuation parameter. The algorithm shall stop when the continuation parameter is below the desired continuation termination tolerance α_m. The part is given below in Algorithm 4.6.

Algorithm 4.6: Damped Continuation Newton's Method.

Input: An initial guess s_0, an initial continuation parameter $\alpha > 0$, a scale factor $0 < \beta < 1$, a convergence tolerance $\epsilon > 0$, a continuation termination tolerance $\alpha_m > 0$ and a maximum iteration number $\max_{\text{iter}}$.

Output: $\tilde{q}^* \in \mathbb{R}^{n_q}$ where $\|\tilde{F}(\tilde{q}^*, \alpha^*)\| \leq \epsilon$ and $\alpha^* \leq \alpha_m$.

Use Algorithm 4.1 to generate the initial input $\tilde{q}$ to the collocation method.

while $\alpha > \alpha_m$ **do**

  for $i = 0, 1, \ldots, max_{iter}$ **do**

    Use Algorithm 4.2 to construct $\tilde{F}(\tilde{q}, \alpha)$.

    if $\|\tilde{F}(\tilde{q}, \alpha)\| \leq \epsilon$ **then**

      $\tilde{q}^* = \tilde{q}$.

      break

    end

    Use Algorithm 4.3 to construct $D\tilde{F}(\tilde{q}, \alpha)$. /* Compute a descent direction */

    Solve the linear system (4.12) for $\Delta\tilde{q}^{(k)}$, i.e.,

$$D\tilde{F}(\tilde{q}, \alpha)\Delta\tilde{q} = -\tilde{F}(\tilde{q}, \alpha).$$

    Use line-search algorithm to find the biggest $\xi \in (0, 1]$ such that $\|\tilde{F}(\tilde{q} + \xi\Delta\tilde{q}, \alpha)\| \leq \|\tilde{F}(\tilde{q}, \alpha)\|$. Then Set $\tilde{q} = \tilde{q} + \xi\Delta\tilde{q}$.

  end

  $\alpha = \beta\alpha$.

  $\tilde{q} = \tilde{q}^*$.

end

Output $\tilde{q}^*$.

4.3 Implementation and Evaluation

The solver is directly implemented from the algorithms described in the previous sections using MATLAB. The solver provides the functionality of solving the BVP-DAE problems of the form (2.38)-(2.40). The usability of the solver is also enhanced by the symbolic functionality from MATLAB to translate the OCP into BVP-DAEs of MATLAB functions that can be used by the solver.

The solver implemented is composed of three main parts as: construction of the residual equation, construction of the Jacobian matrix in section 4.2.3, and BABD linear system solving in section 4.1.1. The results from Chapter 3 point out that the parallel implementation of the BABD solving in MATLAB gives no increase in the efficiency of the solver. Therefore, the BABD solving algorithm is implemented in sequential and the other two parts are implemented both in sequential and parallel as shown in Algorithm 4.2 and 4.3 to explore the efficiency of the parallel implementation in MATLAB.

The feasibility of the solver is tested by over 120 example problems from open literatures and the 2 examples chosen are the examples studied in chapter 3, which can directly give the comparison between those two solvers. A performance profile is generated over the chosen examples from the example set to compare the performance between the solver from this chapter and the one from chapter 3 where a multiple shooting method is used to solve the BVP-DAEs.

Parallelism. The parallel code from Algorithm 4.2 and 4.3 are implemented with the MATLAB parllel computing toolbox functionality `parfor`. The MATLAB parallel worker pool instantiates the thread needed for each parallel process which is very convenient and all the work is executed concurrently.

Computing environment. All the numerical results given below were performed on a desktop with the following hardware and software characteristics. The CPU is an Intel(R) Xeon(R) CPU E5-2640 @ 2.60GHz and the system has 64.0 GB memory. The operating system is Windows 7. The MATLAB version is 2017b.

As the plots of the solution are provided in that chapter already, we do not provide plots in this chapter to save space. For the following evaluations, the initial continuation

parameter is $\alpha = 1$, the scale factor is $\beta = 0.8$, the continuation termination tolerance is $\alpha_m = 1e^{-6}$, and the numerical convergence tolerance is $\epsilon = 1e^{-6}$.

Example 4.3.1. This problem considers a continuous stirred-tank chemical reactor (see Kirk [2012] pp. 338 and pp. 406).

$$\min_{x(t)\in\mathbb{R}^2} \int_0^{t_\mathrm{f}} [x_1(t)^2 + x_2(t)^2]dt$$

subject to

$$\dot{x_1}(t) = -2.0a_1(t) + a_4(t) - a_1(t)u(t),$$
$$\dot{x_2}(t) = 0.5 - x_2(t) - a_4(t).$$

where $a_1(t) = x_1(t) + 0.25, a_2(t) = x_2(t) + 0.5, a_3(t) = x_1(t) + 2.0$ and $a_4(t) = a_2(t)exp[\frac{25.0x_1(t)}{a_3(t)}]$.

The desired objective is to maintain the temperature and concentration close to their steady-state values.The initial constraint is that $\Gamma = [x_1(0) - 0.05, x_2(0)0] = 0$ and the final constraint is that $\Psi = [x_1(t_\mathrm{f}), x_2(t_\mathrm{f})] = 0$. The control variable inequality constraints are bounded as follows, $|u| \leq 1$. Also with the final time $t_\mathrm{f} = 0.78$.

The initial estimate for the states, costates and control is obtained by solving the unconstrained problem where the CVICs are ignored. The initial values of the multipliers are all set to 1. The initial mesh of the problem has $N = 101$ nodes.

The solver converged to the solution with desired numerical tolerance using this initial estimate. The solution from the solver in this paper has $N = 60$ nodes while the solution of the solver using the multiple shooting method from chapter 3 uses $N = 430$ nodes.

Example 4.3.2. This problem is taken from Büskens and Maurer [2000] (pp. 99 to pp. 106) and considers an underwater vehicle problem from aquanautics. The problem is stated as follows

$$\min_{u_i\in\mathbb{R}} \int_0^1 \sum_{i=1}^{4} u_i(t)^2$$

subject to

$$
\begin{aligned}
\dot{x_1} &= \cos(x_6)\cos(x_5)x_7 + r_x,\\
\dot{x_2} &= \sin(x_6)\cos(x_5)x_7,\\
\dot{x_3} &= -\sin(x_5)x_7 + r_z,\\
\dot{x_4} &= x_8 + \sin(x_4)\tan(x_5)x_9 + \cos(x_4)\tan(x_5)x_{10},\\
\dot{x_5} &= \cos(x_4)x_9 - \sin(x_4)x_{10},\\
\dot{x_6} &= \frac{\sin(x_4)}{\cos(x_5)}x_9 + \frac{\cos(x_4)}{\cos(x_5)}x_{10},\\
\dot{x_7} &= u_1,\\
\dot{x_8} &= u_2,\\
\dot{x_9} &= u_3,\\
\dot{x_{10}} &= u_4.
\end{aligned}
$$

The problem has 10 state variables and 4 control variables. The variable $x_1(t)-x_3(t)$ specify the position of the center of the vehicle, while $x_4(t) - x_6(t)$ describe the orientation of the mass by Euler angles. The vehicle is assumed to be moving with velocity $x_7(t)$ and angular velocities $x_4(t) - x_6(t)$. Hence, the control variable $u_1(t)$ represents the acceleration of the underwater vehicle, while $u_2(t) - u_4(t)$ describe the angular accelerations. The nonlinear current is modeled by

$$
\begin{aligned}
r_x &= -u_{x_{max}} e^{-((x_1-c_x)/r_x)^2}(x_1 - c_x)(\frac{x_3 - c_z}{c_z})^2,\\
r_z &= -u_{z_{max}} e^{-((x_1-c_x)/r_x)^2}(\frac{x_3 - c_z}{c_z})^2.
\end{aligned}
$$

with constants $u_{x_{max}} = 2$ representing the maximal horizontal current, $u_{z_{max}} = 1$ representing the maximal vertical current, $c_x = 0.5$ representing the center of current (center of underwater ditch), $c_z = 0.1$ representing the depth of zero current, $r_x = 0.1$ representing the factor for the expansion of the current. The objective is to find the control inputs u that are required to move the vehicle from the initial constraint $\Gamma = [x_1(0), x_2(0), x_3(0)-0.2, x_4(0)-\frac{\pi}{2}, x_5(0)-0.1, x_6(0)+\frac{\pi}{4}, x_7(0)-1, x_8(0), x_9(0)-0.5, x_{10}(0)-0.1] = 0$ to the final constraint $\Psi = [x_1(1) - 1, x_2(1) - 0.5, x_3(1), x_4(1) - \frac{\pi}{2}, x_5(1), x_6(1), x_7(1), x_8(1), x_9(1), x_{10}(1)] = 0$,

while minimizing the cost function. There are also eight CVICs bounded as follows, $|u_i| \leq 15, t \in [0, 1]$.

The initial estimate for the states, costates, and controls is obtained by solving an unconstrained version of the problem where there is no bounds on the control inputs. All the other unknowns (multiplier and parameter variables) are set to 1. The initial estimate uses a uniform mesh with $N = 101$ nodes.

Using this initial estimate, two solvers successfully solve the problem, where the collocation solver uses $N = 171$ nodes and the multiple shooting solver uses $N = 407$ nodes.

Table 4.1 compares the performance of the collocation solver implemented from this chapter and the multiple shooting solver implemented from chapter 3 when applied to the 2 example problems. For each problem, the table shows CPU computation time required for the collocation method and the multiple shooting method all using MATLAB. In addition, the table gives the performance for both solvers ran in sequential or parallel with 4 CPU cores. As can be seen from the table, the parallel implementation using MATLAB of the collocation solver can hardly offer any increase in the performance and even slow the computing process.

Problem	Collocation Method-time(s)		Multiple Shooting Method-time(s)	
	Sequential	Parallel (4 cores)	Sequential	Parallel (4 cores)
Ex. 1	61.8	109.2	89.3	205.0
Ex. 2	500.8	418	764.8	393.5

Table 4.1: Performance comparison between solvers implemented using MATLAB

4.3.1 Performance Profile

A performance profile (Dolan and Moré [2002] and Gould and Scott [2016]) is generated to better quantify the performance of two solvers and benchamark the solver with large problem

set. There are 99 OCPs in the example set. The collocation solver can successfully all of the problems whereas the multiple shooting solver can only solve 85 of them. The performance profile is shown in figure 4.1. From the figure, we can also see that the collocation solver is not only more robust but also more time efficient and we can conclude that the collocation solver is more robust and efficient when solving the BVP-DAEs of the form (2.38)-(2.40).

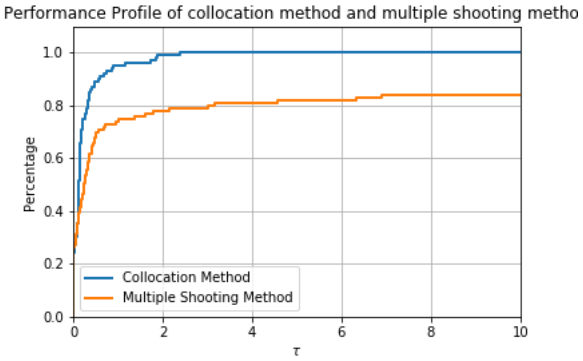


Figure 4.1: Performance profile between solvers implemented using MATLAB.

4.4 Conclusion

In this chapter, another indirect method for solving OCPs with CVICs, SVICs and parameters is presented based on the work done in Chapter 2 which gives the transformation from OCPs to index-1 BVP-DAEs. The main work of this chapter is that it develops a collocation method of solving the index-1 BVP-DAEs and a continuation Newton's method to solve the residual equation from the collocation formulation.

A numerical solver is programmed using MATLAB along with the parallel computing capability. The parts that can be executed independently are implemented in parallel. However, the performance of the parallel solver is not satisfying. That is mainly because the overhead on setting up the parallel worker in MATLAB parallel computing pool is too much than just computing the whole work in sequential in MATLAB.

Two concrete examples are tested with both solvers shown in this chapter and the one in chapter 3 to directly compare the performance between them. A performance profile tested

over 95 examples from open literatures is also shown to better quantify the efficiency and robustness between the solvers. It can be seen from the performance profile that the solver using collocation method is more robust and efficient on a large set of OCP examples than the solver using multiple shooting method.

Next chapter will present the work focusing on implementing the algorithm on more advanced computing hardwares as graphics processing units (GPUs) using Python (Van Rossum and Drake Jr [1995]). Although the the computing ability of single GPU computing core is not better that on single CPU computing core, the number of computing cores on a GPU is much more than that on a CPU and it can increase the efficiency of the solver to a large extent. Python also has lots of packages that are efficient and convenient for scientific computing such as sympy, numpy, and scipy.

Chapter 5

GPU BASED IMPLEMENTATION

5.1 Motivation

In chapters 3 and 4, two numerical methods to solve the BVP-DAEs (2.38)-(2.40) are introduced. The implementation using MATLAB for both algorithms are presented and the parallel computing toolbox in MATLAB is explored. However, the evaluation results of the implementation using MATLAB are not satisfactory. For the multiple shooting method in chapter 3, the implementation tries to make use of the built-in integrator `ode15s`, where the test results showed it was not suitable for the multiple shooting algorithm developed than the integrator implemented using ROW method. Moreover, due to the heavy overhead in the MATLAB parallel computing toolbox, the parallel implementation can not provide too much performance boost on the multiple shooting algorithm and even decrease the performance on the collocation algorithm.

In the past decade, the increase of computing power of CPUs has slowed down due to the breakdown of Moore's law, which states that the number of transistors in a dense integrated circuit doubles approximately every two years Brock and Moore [2006]. In contrast, leveraging the ability of Graphics Processing Units (GPUs) to do massively paralleled work has become increasingly popular in the community of computational science and engineering. Using the GPU to accelerate the algorithm solving process is very promising by using faster hardware and more computing cores. GPU is a special purpose processor that is built specifically for performing large volumes of computations. Modern GPU contains thousands of cores which are capable of performing trillions of floating point operations per second (FLOPs), accelerating the rate at which the computations can be completed. For large computations GPU can perform the same operation over and over again on them at a much faster rate than a CPU alone. Many researchers from different disciplines such as molecular dynamics Anderson et al. [2008], Liu et al. [2007], computational biology Schatz

et al. [2007], weather forecasting Michalakes and Vachharajani [2008], linear algebra Barrachina et al. [2008], computational fluid dynamics Schive et al. [2010], astrophysics Wang et al. [2010], tsunami modeling Qin et al. [2019] have developed numerical models that run on the GPUs to make use of the computing power.

The GPU is initially widely used in the computer graphics community, while before the appearance of libraries for general-purpose GPU computing like CUDA, researchers had to write their own programs in a way such that they could take use of the GPUs' capability of computing color for a large number of pixels in parallel Owens et al. [2007]. The appearance of programming models for general purpose GPU computing such as CUDA (Nickolls et al. [2008], Luebke [2008]) makes developing numerical algorithms on the GPU much easier.

Python (Van Rossum and Drake Jr [1995]) is becoming one of the most popular programming languages recently as it has many well-developed tools for scientific computing which allows developers to better focus on the hurdle on the path to performance.

In the optimal control community, the author does not know any research on employing the computing power of the GPU yet. In this chapter, we present the GPU based implementation for the multiple shooting method from Chapter 3 and the collocation method from Chapter 4 using CUDA and Python.

5.2 CUDA Programming Model

The CUDA programming used is supported by a high performance Python compiler Numba (Lam et al. [2015]), which directly compiles a restricted subset of Python code into CUDA kernels and device functions following the CUDA execution model. The detailed instructions of the CUDA programming model can be found in the Nvidia CUDA C Programming Guide (Nvidia [2019]) and Storti and Yurtoglu [2015]. Here, we provide adequate introduction about CUDA programming to help understand the implementation shown in this chapter.

CUDA is a hardware and software combined system for parallel computing on the CUDA-enabled GPUs. The functions executed on the GPU is called CUDA kernel functions. CUDA follows the Single Instruction, Multiple Threads (SIMT) programming model where the kernel function specifies the instructions which are then executed by certain threads using the GPU. Threads are grouped into blocks, which form a grid then. On the hardware

side, a single CUDA-enabled GPU contains thousands of compute cores which are capable of performing the specified instructions. Those compute cores are grouped together and called Streaming Multiprocessors (SMs). CUDA blocks are independent of each other and every SM can be assigned with multiple blocks. Every CUDA block is futher divided as warps where each warp contains 32 threads and execute the same instructions concurrently. Each active thread is uniquely identified based on the block and thread index information and is able to access its corresponding portion of the data.

5.3 Multiple Shooting Method

5.3.1 CUDA kernel implementation

The CUDA implementation of the multiple shooting method uses the integration evaluation from Section 3.1.3, where a single step linearly implicit Runge-Kutta (ROW) method is used for integrating the DAEs.

The core idea for CUDA implementation is to design the implementation of the kernel function. When implementing the CUDA kernel function, the primary goal is to divide the overall algorithm into parallel tasks that can be assigned to CUDA threads. From Algorithm 3.1 and Algorithm 3.2 of the whole method, it is intuitive to conclude that the integration of the DAEs (3.8) and linear DAEs (3.14) performed in each time interval $[t_j, t_{j+1}]$, $j = 1, 2, \ldots, N-1$ can be assigned to each individual CUDA thread.

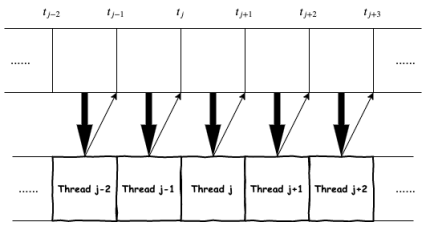


Figure 5.1: CUDA kernel layout for multiple shooting method.

Figure 5.1 shows how the CUDA threads are first assigned to each time interval for integrating DAEs, and how the approximated solutions are returned. The solid arrows

denote initial set up for each thread by passing the corresponding input while the the thin arrows show the return of the approximate solutions after the computation of the integration.

5.3.2 Numerical evaluation

Algorithms for solving the OCP (2.7)-(2.12) are implemented using Python and CUDA. The main functionality provided by this implementation is a set of routines that solve BVP-DAE problems of the form (2.38)-(2.40). The codes are directly implemented from all the algorithms described above which can be found at web site `https://github.com/UW-OCP/Multiple_Shooting_Solver_CUDA`, and all the implementations of the codes are standalone.

The solver is composed of two major parts: (i) construction of the residual with DAEs integration and construction of the Jacobian with the DAE sensitivities integration described in section 3.1.3; (ii) the solver for the BABD linear system described in section 3.2. Both the sequential and parallel versions of these two parts are implemented to compare the efficiency of the implementation.

As shown in the tables below, Solver_1 denotes the solver implemented with the sequential implementation of the construction of the residual with DAEs integration and the construction of the Jacobian with the DAE sensitivities integration, and sequential implementation of the BABD linear system solving; Solver_2 denotes the solver with the parallel implementation of the construction of the residual with DAEs integration and the construction of the Jacobian with the DAE sensitivities integration, and parallel implementation of the BABD linear system solving.

The result tables use the following terminologies. N_{solution} denotes the number of times nodes in the final solution; T_{all} denotes the overall computation time of the whole problem; M denotes the number of partitions used in solving the BABD system in parallel; N_{residual} denotes the number of evaluating times of the construction of the residual (3.4) with DAEs integration; T_{residual} denotes the the running time of the construction of the residual (3.4) with DAEs integration; N_{Jacobian} denotes the number of evaluating times of the construction of the Jacobian (3.7) with the DAE sensitivities integration; T_{Jacobian} denotes the running

time of the construction of the Jacobian (3.7) with the DAE sensitivities integration; N_{BABD} denotes the number of evaluating times of the BABD linear system solving (3.25); and T_{all} denotes the running time of the BABD linear system solving (3.25).

The solver is evaluated using more than 140 examples from opern literatures which can be found at website `https://github.com/UW-OCP/ocp_test_problems`. Here, we present two examples from this problem set to illustrate the robustness and efficiency of the solver.

Computing environment. All the numerical results given below are performed on a computer with the following hardware and software characteristics. The computer is equipped with an Intel® Core™ i9-9900K CPU @ 3.60GHz × 16 and a GeForce RTX 2070 SUPER/PCIe/SSE2 GPU. The operating system is Ubuntu 19.10.

Example 5.3.1. This is a self-excited oscillation system described by van der Pol's equation with two state variables, one control variable, and one state variable inequality constraint from Shimizu and Ito [1994].

$$\min_{x(t)\in\mathbb{R}^2, u(t)\in\mathbb{R}} \int_0^{t_{\text{f}}} 0.5[x_1(t)^2 + x_2(t)^2 + u(t)^2]\, dt,$$

subject to

$$\begin{aligned} \dot{x_1}(t) &= x_2(t), \\ \dot{x_2}(t) &= -x_1(t) + (1.0 - x_1(t)^2)x_2(t) + u(t). \end{aligned}$$

The initial condition is that $\Gamma = [x_1(0) - 1.0, x_2(0)] = 0$ and there is no final time constraint. The state variable inequality constraint is $-(x_2(t) + 0.25) \leq 0$. Also the final time is $t_{\text{f}} = 5$. The initial estimates for the state variables and control variable are all ones and obtained on a uniform mesh with $N = 101$ nodes.

Using those initial estimates, we obtain the solutions for the differential and algebraic variables that are shown in figure 5.2 and figure 5.3. The convergence tolerance for this problem is $\epsilon = 10^{-6}$.

Table 5.1 shows the number of calling times of each algorithm and the computational running time of each algorithm. It also shows the speedup factor which is defined as $S = \frac{T_{\text{sequential}}}{T_M}$, where $T_{\text{sequential}}$ is the running time of the sequential solver and T_M is the running time of the parallel solver with M partitions in BABD system solving.

The results show that there is a significant boost in speed of the GPU based parallel implementation in every part of the algorithm and an overall speedup factor between 24 to 71 are obtained with different number of partitions M used in the BABD system solving. It can be seen that the running time for evaluating residual equation and Jacobian are almost the same for parallel solvers with different partitions M which only affects the BABD linear system solving. As the number of partitions M is increased from 1 to 16, a very significant decrease in the BABD system solving time is realized. It also can be seen that the number of evaluating times for each part of the algorithm and the number of time nodes of the final solution N_{solution} are exactly the same for sequential and parallel solvers. This indicates that although the instructions of the sequential solver happens on CPU which is different from the GPU based parallel solver, their numerical computation are still identical.

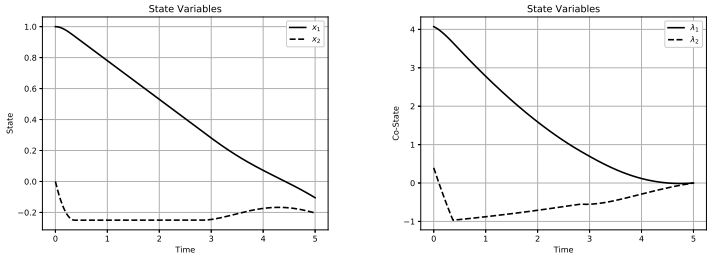


Figure 5.2: Optimal solution of states and costates for Example 5.3.1.

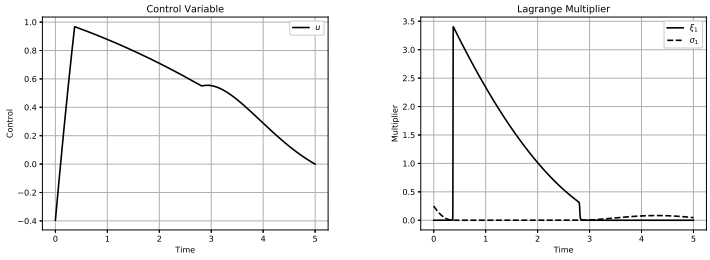


Figure 5.3: Optimal solution of control and multipliers for Example 5.3.1.

	Solver_1	Solver_2				
M	1	1	4	8	16	32
N_{residual}	512	512	512	512	512	512
N_{Jacobian}	254	254	254	254	254	254
N_{BABD}	191	191	191	191	191	191
$T_{\text{residual}}(\text{s})$	367.5	9.4	9.7	9.5	9.5	9.5
$T_{\text{Jacobian}}(\text{s})$	1777.6	14.4	14.8	14.7	14.7	14.2
$T_{\text{BABD}}(\text{s})$	403.2	81.7	25.7	16.3	12.9	11.8
N_{solution}	2440	2440	2440	2440	2440	2440
$T_{\text{total}}(\text{s})$	2548.6	105.8	50.5	40.9	37.4	35.8
Speedup		24.1	50.5	62.3	68.1	71.2

Table 5.1: Example 5.3.1, Evaluating times, running time and speedup factor

Example 5.3.2. This problem considers a system with 6 state variables, 4 control variables, 1 parameter variable, and 4 control variable inequality constraints. The problem is tested by the algorithms without using the penalty function method (see (2.1)-(2.6)) on the CVICs, and algorithm using the the penalty function method (see (2.7)-(2.12)). (Note that the algorithm without the penalty function was unable to converge.) The problem is to minimize the cost functional

$$\min_{u_i \in \mathbb{R}, p \in \mathbb{R}} \int_0^1 (p+10)(\rho + u_1(t)^2 + u_2(t)^2 + u_3(t)^2 + u_4(t)^2)\, dt,$$

subject to

$$
\begin{aligned}
\dot{x}_1(t) &= (p+10)x_2(t), \\
\dot{x}_2(t) &= (p+10)((u_1(t)+u_3(t))\cos(x_5(t)) - (u_2(t)+u_4(t))\sin(x_5(t))/m, \\
\dot{x}_3(t) &= (p+10)x_4(t), \\
\dot{x}_4(t) &= (p+10)((u_1(t)+u_3(t))\sin(x_5(t)) - (u_2(t)+u_4(t))\cos(x_5(t))/m, \\
\dot{x}_5(t) &= (p+10)x_6(t), \\
\dot{x}_6(t) &= (p+10)((u_1(t)+u_3(t))d - (u_2(t)+u_4(t))l)/I,
\end{aligned}
$$

with constants $m = 10.0$, $d = 5.0$, $l = 5.0$, $I = 12.0$, and $\rho = 10.0$. The objective is to find the control inputs u_i, $i = 1, 2, 3, 4$ that move the system from the initial time constraint $\Gamma = [x_1(0), x_2(0), x_3(0), x_4(0), x_5(0), x_6(0)] = 0$ to the final time constraint $\Psi = [x_1(1) - 4.0, x_2(1), x_3(1) - 4.0, x_4(1), x_5(1) - \frac{\pi}{4.0}, x_6(1)] = 0$, while minimizing the cost functional. There are also four inequality constraints that bound the control variables as follows, $-(u_i(t) - 5.0)(-5.0 - u_i(t)) \leq 0, i = 1, 2, 3, 4$.

The initial estimate for this problem is obtained by solving an unconstrained version of the problem and with no bounds on the control inputs. The estimate is obtained on a uniform mesh with $N = 101$ nodes.

Using this initial estimate, we obtain the solutions for the differential and algebraic variables that are shown in figure 5.4, figure 5.5 and figure 5.6. The convergence tolerance for this problem is $\epsilon = 10^{-6}$.

From table 5.2, we can see that both solvers converge to the same solution with a mesh of $N_{\text{solution}} = 287$ nodes and every part of the algorithm has a significant boost in performance with an overall speedup factor from 27 to 54. This problem contains a large number of unknown variables and as a result, the computation for sensitivity integration and the solution of the BABD system are time consuming where the performance improvement is significant. However, more variables in the system mean that the memory transfer between the CPU and the GPU is slower. Also, the number of time nodes in this problem is much fewer than the previous example so that the speedup for evaluating the residual and the Jacobian is smaller.

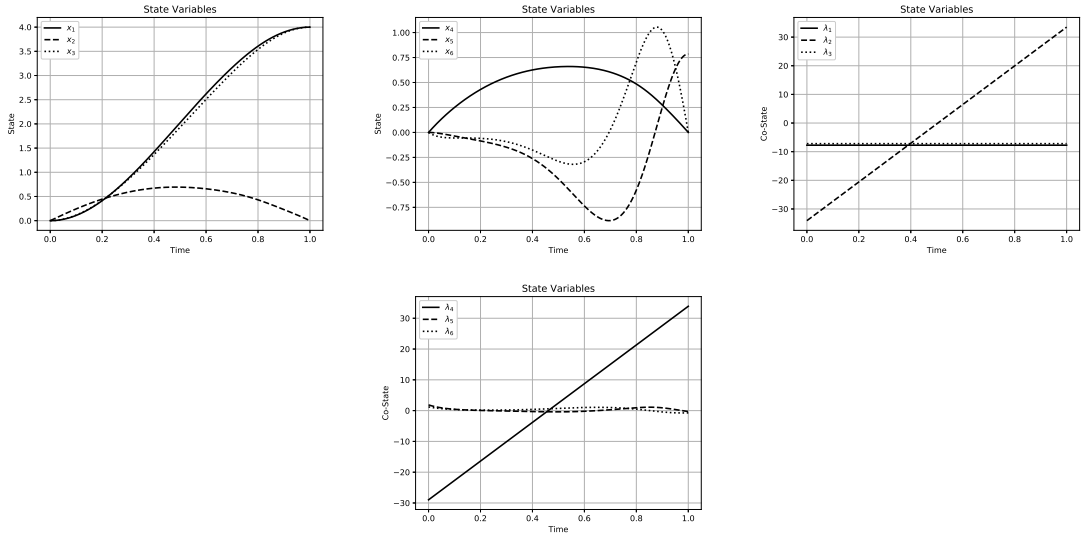


Figure 5.4: Optimal solution of states and costates for Example 5.3.2.

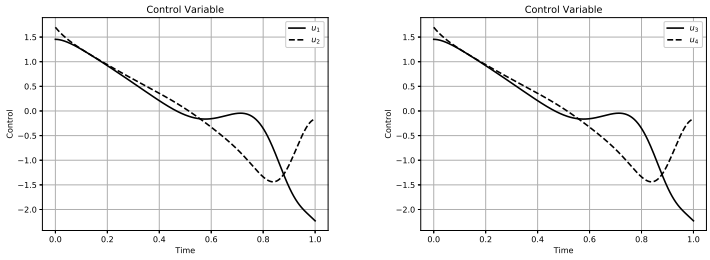


Figure 5.5: Optimal solution of controls for Example 5.3.2.

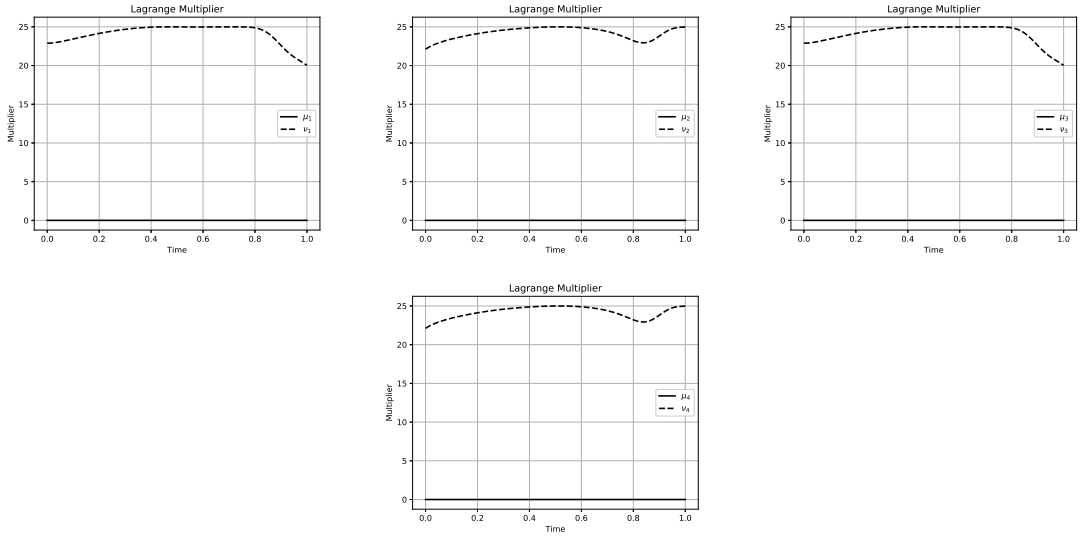


Figure 5.6: Optimal solution of multipliers for Example 5.3.2.

	Solver_1	Solver_2				
M	1	1	4	8	16	32
N_{residual}	187	187	187	187	187	187
N_{Jacobian}	123	123	123	123	123	123
N_{BABD}	60	60	60	60	60	60
T_{residual}(s)	460.6	16.4	16.6	16.4	16.4	16.4
T_{Jacobian}(s)	10661.2	163.9	164.6	167.6	165.7	167.0
T_{BABD}(s)	1650.6	285.2	89.9	71.5	53.3	52.4
N_{solution}	287	287	287	287	287	287
T_{total}(s)	12772.4	465.6	271.1	255.6	235.4	235.8
Speedup		27.4	47.1	50.0	54.3	54.2

Table 5.2: Example 5.3.2, Evaluating times,running time and speedup factor

5.4 Collocation Method

5.4.1 Overview

Figure 5.7: Work flow of the Collocation algorithm.

Figure 5.7 shows the major workflow procedures of the collocation algorithm introduced in chapter 4. The two colors presented in the figure represent the two major types of hardware resources that are involved in the execution of the code, CPU and GPU. The red block indicates the code of the procedure that is executed on the CPU, whereas the green block indicates the execution on the GPU. The arrow from procedure 1 to procedure 2 indicates that the procedure 2 can be executed only after procedure 1 is finished. The memory transfer between CPU and GPU is kept at the least amount possible and can be neglected.

5.4.2 Design of CUDA Kernel

The implementation of the multiple shooting algorithm in section 5.3 employs a 1D CUDA kernel. However, the CUDA kernel function can be launched by multi-dimensional block where up to 3-dimensional is supported. The thread can then be indexed with the corresponding multi-dimensional identifier.

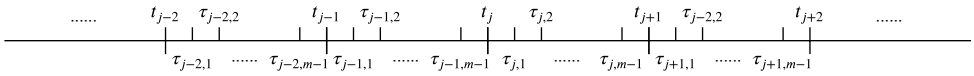


Figure 5.8: The 1-dimensional layout of the mesh of the time interval from the collocation algorithm.

Figure 5.8 shows the 1-dimensional layout of the mesh of the overall time interval resultant from collocation method shown in section 4.1. The overall time interval $[t_{\mathrm{i}}, t_{\mathrm{f}}]$ is divided into $N-1$ sub-intervals $[t_j, t_{j+1}]$, $j = 1, 2, \ldots, N-1$ and each sub-interval $[t_j, t_{j+1}]$ is further divided by m_j collocation points $\tau_{j,i}$, $i = 1, 2, \ldots, m_j$. A global unified number of collocation points m_j is used here. From Algorithms 4.1, 4.2, 4.3, 4.4, and 4.5, it is easy to notice that all the computations performed in the interval $[t_j, t_{j+1}]$ can be done independently and concurrently with any other interval $[t_i, t_{i+1}]$, $i \neq j$ without any data racing possibility. Therefore, the computation in each time interval can be done by easily assigning each CUDA thread to finish the corresponding computation using 1-dimensional CUDA grid.

However, notice that Algorithms 4.1, 4.2, and 4.3 are computed with a nested for loop where all the computation in the nested loop can be done independently. The only difference between those algorithms is that for Algorithm 4.2, some computation needs to be done at the first level of the loop which has no effect on the inner loop. Figure 5.9 shows according to the 2D nested loop how we are able to transform the mesh of the time interval from the original 1D mesh to the 2D mesh.

Current implementation makes use of the above feature and a 2-dimensional CUDA grid is used in implementing Algorithms 4.1, 4.2, and 4.3. Figure 5.10 shows how CUDA

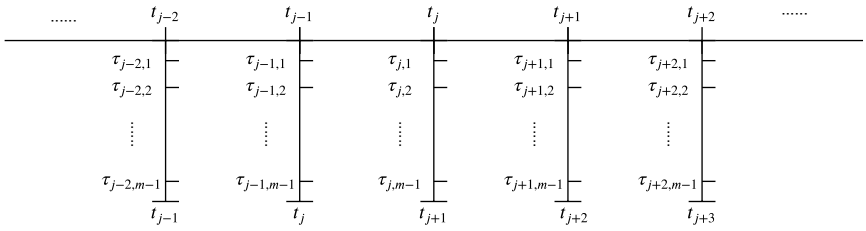


Figure 5.9: The 2-dimensional layout of the mesh of the time interval from the collocation algorithm.

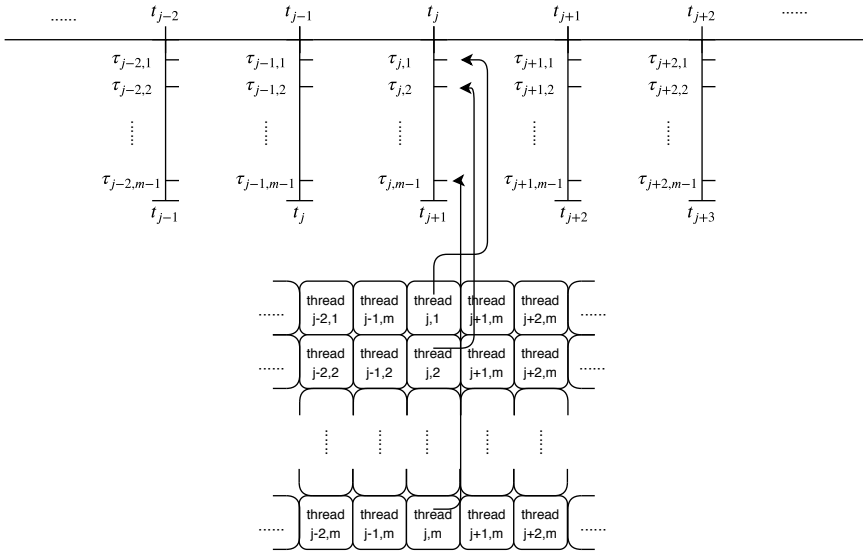


Figure 5.10: Assignment of 2D CUDA threads to the mesh.

threads are assigned to the 2D mesh to solve the whole problem. Each arrow shows the correspondence between the collocation points and the 2D CUDA threads where the index can be uniquely identified with CUDA API. For the extra work in the first level of the loop mentioned above in Algorithm 4.2, current implementation uses the first thread in the second dimension (thread j, 1 shown in Figure 5.10) to finish the work. The implementation for Algorithms 4.4 and 4.5 uses a simpler 1D CUDA grid to solve the problem where each thread solves the work of the corresponding time interval.

5.4.3 Data and memory Layout

When executing the kernel function, CUDA blocks are broken up into warps where each warp consists of 32 threads and run the same instructions including memory loading and writing simultaneously. Warps are switched to execute on the SM where only one warp can be executed on the SM at the same time. GPU is able to execute the memory loading operations in a warp in the most efficient way if the memory requested by the threads in the warp are contiguous on memory, which is called coalesced memory access in CUDA. Due to this property of CUDA, we use the Structures of Arrays as the data layout in the implementation. Using this data layout format, all the variables, such as $\tilde{y}_j, \dot{\tilde{y}}_i^{(j)}, \tilde{z}_{ji}, \tilde{f}_j^a, \tilde{f}_j^b, J_j, V_j, D_j, W_j$, $j = 1, 2, \ldots, N-1$, $i = 1, 2, \ldots, m_j$ are stored together continuously as arrays on the memory during the execution. All the memory are pre-allocated before the execution of the kernel.

The memory types most critical to the performance in GPU parallel computing are register memory, shared memory, and global memory. Global memory lies furthest from the compute cores and has the longest data access latency. However, global memory has the largest device memory and can be accessed by all the threads in the grid. Shared memory lies locally with each SM and is allocated for every CUDA block where all the threads in the same block can access it. It has a much faster latency than global memory, while the amount on each SM is limited. Register memory provides the fastest access, but is handled directly by the device. Each thread has its own register memory but is extremely limited.

Using shared memory is very advantageous in computations which access the same mem-

ory locations multiple times. When computing the differential variables $\tilde{y}_i^{(j)}$ in Algorithm 4.2, the variables $\tilde{y}_j$, $\dot{\tilde{y}}_l^{(j)}$ and the collocation coefficients a_{il} are accessed m_j times for the threads. Therefore, the implementation can benefit from loading those variables from the global memory to the shared memory first and then execute all the computations. However, notice that it is not practical to load all the memory from the global memory to the shared memory, perform all the computations using the shared memory to load data, and write the results back to the global memory. The reason is that the shared memory on each SM is limited and if the CUDA block uses too much shared memory, then only few blocks are able to reside on the SM simultaneously. This is called low occupancy and the CUDA kernel executes very inefficiently.

5.4.4 Numerical evaluation

This section aims to evaluate the performance and correctness of the GPU implementation of the collocation method introduced. The solver is directly implemented from the algorithms and evaluation details shown using Python and CUDA. All the codes are standalone and can be obtained at `https://github.com/UW-OCP/Collocation-Solver-CUDA`. The solver aims to solve the BVP-DAEs problem of the form (2.38)-(2.40) respectively. The usability of the solver is enhanced by the symbolic functionality to translate the OCP into BVP-DAEs of the desired format. All computation results are obtained on a machine equipped with Intel® Core™ i9-9900K CPU @ 3.60GHz × 16 and GeForce RTX 2070 SUPER/PCIe/SSE2 GPU running Ubuntu 19.10 Operating System.

The feasibility of the solver is tested by over 140 examples which are taken from the open literatures. The solver is applied to four specific examples in this section to show the robustness and efficiency. Performance profiles Dolan and Moré [2002], Gould and Scott [2016] are generated over the chosen example sets to compare the performance between the solver here and in section 5.3 where a GPU based multiple shooting method is implemented, and between solvers with different number of collocation points used.

Following terminologies are used in the upcoming example evaluations. Col denotes the collocation algorithm; Ms denotes the multiple shooting algorithm; $T_{Alg\,4.2}$ denotes the

computation time of Algorithm 4.2; $T_{Alg\,4.3}$ denotes the computation time of Algorithm 4.3; $T_{Alg\,4.4}$ denotes the computation time of Algorithm 4.4; $T_{Alg\,4.5}$ denotes the computation time of Algorithm 4.5; and T denotes the overall computation time for solving the example. All the examples are tested with the numerical tolerance $\epsilon = 10^{-6}$. The solver is executed with double-precision floating point formats. To keep the order of the solution accuracy the same between the collocation method and the multiple shooting method, all the examples are executed with $m_j = 4$ collocation points. Details of the formulation and plots of the solutions of examples 5.4.1-5.4.4 can refer to `https://github.com/UW-OCP/Collocation-Solver-CUDA/blob/master/Notebook/examples_presentation.ipynb`.

Example evaluation

Example 5.4.1. Container transfer problem

Consider a problem of transferring container from a ship to a cargo truck adapted from Teo [1996]. The problem is to minimize the swing during the transfer where the container crane is driven by a hoist motor and a trolley drive motor. The problem can be modelled as follows:

$$J = \int_0^1 p(1 + 0.01(u_1(t)^2 + u_2(t)^2)\ dt,$$

subject to the dynamic equations

$$\begin{aligned}
\dot{x}_1(t) &= px_4(t),\\
\dot{x}_2(t) &= px_5(t),\\
\dot{x}_3(t) &= px_6(t),\\
\dot{x}_4(t) &= p(u_1(t) + 17.2656x_3(t)),\\
\dot{x}_5(t) &= pu_2(t),\\
\dot{x}_6(t) &= -\frac{p}{x_2(t)}[u_1(t) + 27.0756x_3(t) + 2x_5(t)x_6(t)],
\end{aligned}$$

and the boundary conditions

$$\begin{aligned}
x(0) &= [0, 22, 0, 0, -1, 0]^T,\\
x(1) &= [10, 14, 0, 2.5, 0, 0]^T,
\end{aligned}$$

Table 5.3: Example 5.4.1, computation time and speedup factor

	Sequential	Parallel	Speedup
$T_{Alg\,4.2}$ (s)	10.86	3.68	2.95
$T_{Alg\,4.3}$ (s)	56.61	2.65	21.36
$T_{Alg\,4.4}$ (s)	5073.96	37.30	136.03
$T_{Alg\,4.5}$ (s)	1617.51	10.17	159.05
T (s)	7002.13	71.63	97.75

and the inequality constraints

$$|u_1(t)| \leq 2.83374, \forall t \in [0,1],$$
$$-0.80865 \leq u_2(t) \leq 0.71265, \forall t \in [0,1].$$

The initial guesses are obtained by solving the unconstrained problem with all the unknown variables being constants with $N = 101$ nodes. The final converged solution has a mesh of $N = 108$ nodes. The computation time for the sequential and parallel implementation of each algorithm in section 4.2 is shown in table 5.3. Sequential code is executed on a single CPU alone. The table also shows the speedup factor for each algorithm, which is defined as $S = T_{\text{sequential}}/T_{\text{parallel}}$.

Algorithm 4.2 presents relative lower speedup compared with Algorithms 4.3, 4.4, and 4.5. It is because the overhead to launch GPU kernels takes nearly a fixed amount of time no matter how long the actual execution time of the kernel takes. To execute a tiny GPU kernel function, the overhead of launching the kernel can take a large portion of the whole execution time. Compared with the total work costs of the execution of the kernels in Algorithms 4.3, 4.4, and 4.5, the total work cost in Algorithm 4.2 is much less so that the speedup is not as great. However, the overall performance boost for solving the example is very promising with a speedup factor around 97.

Example 5.4.2. Underwater vehicle problem

This is a high dimensional and highly nonlinear optimal control problem taken from Büskens and Maurer [2000]. The problem is to control an underwater vehicle with ten state variables and four control variables respectively. The objective is to minimize the energy control corresponding to:

$$J = \int_0^1 \sum_{i=1}^{4} u_i^2 \, dt,$$

subject to the dynamic equations

$$\begin{aligned}
\dot{x}_1 &= \cos(x_6)\cos(x_5)x_7 + r_x, \\
\dot{x}_2 &= \sin(x_6)\cos(x_5)x_7, \\
\dot{x}_3 &= -\sin(x_5)x_7 + r_z, \\
\dot{x}_4 &= x_8 + \sin(x_4)\tan(x_5)x_9 + \cos(x_4)\tan(x_5)x_{10}, \\
\dot{x}_5 &= \cos(x_4)x_9 - \sin(x_4)x_{10}, \\
\dot{x}_6 &= \frac{\sin(x_4)}{\cos(x_5)}x_9 + \frac{\cos(x_4)}{\cos(x_5)}x_{10}, \\
\dot{x}_7 &= u_1, \\
\dot{x}_8 &= u_2, \\
\dot{x}_9 &= u_3, \\
\dot{x}_{10} &= u_4,
\end{aligned}$$

and the boundary conditions

$$\begin{aligned}
x(0) &= [0, 0, 0.02, \frac{\pi}{2}, 0.1, -\frac{\pi}{4}, 1.0, 0, 0.5, 0.1]^T, \\
x(1) &= [1.0, 0.5, 0, \frac{\pi}{2}, 0, 0, 0, 0, 0, 0]^T,
\end{aligned}$$

and eight control variable inequality constraints

$$-15 \leq u_i(t) \leq 15, \forall t \in [0, 1], i = 1, \ldots, 4.$$

Here, x_1-x_3 represent the position of the center of the mass of the vehicle and x_4-x_6 denote the angular orientation of the vehicle. x_4 specifies the roll motion, while x_5 and x_6

Table 5.4: Example 5.4.2, computation time and speedup factor

	Sequential	Parallel	Speedup
$T_{Alg\,4.2}$ (s)	104.42	15.53	6.72
$T_{Alg\,4.3}$ (s)	378.93	7.54	50.26
$T_{Alg\,4.4}$ (s)	58004.93	265.57	218.42
$T_{Alg\,4.5}$ (s)	19536.56	68.79	284.00
T (s)	80167.24	459.71	174.39

describe the pitch and the yaw motions. The model assumes the vehicle moves with velocity x_7 and angular velocities x_8-x_{10}. u_1 denotes the vehicle acceleration and u_2-u_4 refer to the angular accelerations. The nonlinear current is modeled by

$$r_x = -u_{x_{max}} e^{-((x_1-c_x)/r_x)^2}(x_1 - c_x)(\frac{x_3 - c_z}{c_z})^2,$$
$$r_z = -u_{z_{max}} e^{-((x_1-c_x)/r_x)^2}(\frac{x_3 - c_z}{c_z})^2.$$

with constants $u_{x_{max}} = 2$ representing the maximal horizontal current, $u_{z_{max}} = 1$ representing the maximal vertical current, $c_x = 0.5$ representing the center of current (center of underwater ditch), $c_z = 0.1$ representing the depth of zero current, $r_x = 0.1$ representing the factor for the expansion of the current.

The initial estimates for the states, costates, and controls are obtained by solving an unconstrained version of the problem where there is no bound on the control inputs. All the other unknown variables are set as constants. The initial estimate uses a uniform mesh with $N = 101$ nodes.

Using the initial estimates, the solver successfully solves the problem with $N = 295$ nodes. The computation time for the sequential and parallel implementation is shown in table 5.4.

Table 5.4 shows a similar result as in Table 5.3, while a higher overall speedup factor

around 175 is achieved here. The reason is that this example has more number of variables with 10 state variables, 4 control variables, and 8 control variable inequality constraints. As the overhead to launch GPU kernels takes a nearly fixed amount of time, kernel function which takes longer computation time can benefit more and thus has a higher speedup.

Example 5.4.3. Freespace rocket problem

This is problem about a freespace rocket model with three states and one control from Houska et al. [2011]. The aim is to fly in minimum time from the initial location to the final location where the final time is represented with the parameter variable p. The problem is formulated as follows:

$$J = \int_0^1 p(1 + 0.1u(t)^2)\ dt,$$

subject to the dynamic equations

$$\begin{aligned}
\dot{y_1}(t) &= py_2(t), \\
\dot{y_2}(t) &= p\frac{u(t) - 0.2y_2(t)^2}{y_3(t)}, \\
\dot{y_3}(t) &= -0.01pu(t)^2,
\end{aligned}$$

and the boundary conditions

$$\begin{aligned}
x(0) &= [0, 0, 1.0]^T, \\
x(1) &= [10.0, 0]^T,
\end{aligned}$$

and two control variable inequality constraints

$$-1.1 \leq u(t) \leq 1.1, \forall t \in [0, 1],$$

and two state variable inequality constraints

$$-0.1 \leq y_2(t) \leq 1.7, \forall t \in [0, 1].$$

The three states denote the distance, velocity, and acceleration of the rocket.

The initial guesses are obtained by solving a relaxed problem with a bigger cost functional. The final solution obtained has $N = 83$ time nodes with the optimal time $t_\mathrm{f} = p =$

Table 5.5: Example 5.4.3, computation time and speedup factor

	Sequential	Parallel	Speedup
$T_{Alg\,4.2}$ (s)	4.11	4.23	0.97
$T_{Alg\,4.3}$ (s)	16.08	2.44	6.59
$T_{Alg\,4.4}$ (s)	897.46	15.64	57.38
$T_{Alg\,4.5}$ (s)	374.58	5.56	67.37
T (s)	1315.62	43.62	30.16

7.43. The computation time for the sequential and parallel implementation of the algorithm evaluation is shown in table 5.5.

It can be seen from the table that except Algorithm 4.2 every other part has a performance boost with parallel implementation. The reason for no speedup in Algorithm 4.2 is because the parallel implementation uses a just-in-time (jit) compilation where the GPU kernel functions are compiled when they get called for the first time which takes a huge amount of time. As the Algorithms get evaluated very few times and need less time to finish, compilation overhead is much slower compared with direct CPU execution. This amortized time complexity may decrease the performance however it is suggested in the compiler Numba as it can generate optimized code based on the jit compilation. Also, it can be seen that the speedup of this example is smaller than the two preceding examples where the overall speedup factor is around 30. This is mainly due to the problem has relatively less number of variables and is less complicated. The total computation time is shorter so that the parallel implementation can not benefit too much due to the fixed overhead.

Example 5.4.4. Trolley problem

This is a problem [Chen and Gerdts, 2012] considering a trolley of mass m_1 moving in a high rack storage area with a load of mass m_2 attached to the trolley by a rigid cable of

lengh l. The problem is formulated as follows: for $t_\mathrm{f} = 2.7$, minimize

$$J = \int_0^{t_\mathrm{f}} u^2 + 5x_4^2 \; dt,$$

subject to the dynamic equations

$$\begin{aligned}
\dot{x}_1 &= x_3, \\
\dot{x}_2 &= x_4, \\
\dot{x}_3 &= \frac{m_2^2 l^3 \sin(x_2) x_4^2 - m_2 l^2 u + m_2 I_y l x_4^2 \sin(x_2) - I_y u + m_2^2 l^2 g \cos(x_2) \sin(x_2)}{-m_1 m_2 l^2 - m_1 I_y - m_2^2 l^2 - m_2 I_y + m_2^2 l^2 \cos(x_2)^2}, \\
\dot{x}_4 &= \frac{m_2 l (m_2 l \cos(x_2) x_4^2 \sin(x_2) - \cos(x_2) u + g \sin x_2 (m_1 + m_2))}{-m_1 m_2 l^2 - m_1 I_y - m_2^2 l^2 - m_2 I_y + m_2^2 l^2 \cos(x_2)^2},
\end{aligned}$$

and the boundary conditions

$$\begin{aligned}
x(0) &= [0, 0, 0, 0]^T, \\
x(t_\mathrm{f}) &= [1.0, 0, 0, 0]^T,
\end{aligned}$$

and two control variable inequality constraints

$$-0.5 \leq u(t) \leq 0.5, \forall t \in [0, 1],$$

where $g = 9.81$, $m_1 = 0.3$, $m_2 = 0.5$, $l = 0.75$, and $I_y = 0.002$. Here, x_1 and x_3 represent the coordinate of the trolley and its velocity in x direction, while x_2 and x_4 denote the angle between the cable and the vertical axis and the corresponding velocity.

The initial estimates are obtained by solving the problem without inequality constraint with $N = 101$ time nodes. The computation time for the sequential and parallel implementation is shown in table 5.6.

A speedup factor of around 110 is achieved for this example. This is mainly because the example is highly nonlinear and complicated which can be seen from the dynamical equations of the problem. The nonlinearity and complexity results in longer computation time needed and a higher speedup in parallel evaluation.

5.4.5 Results discussion

Table 5.7 compares the performance of the GPU based collocation solver in section 5.4 and the GPU based multiple shooting solver in section 5.3 when applied to examples 5.4.1-5.4.4.

Table 5.6: Example 5.4.4, computation time and speedup factor

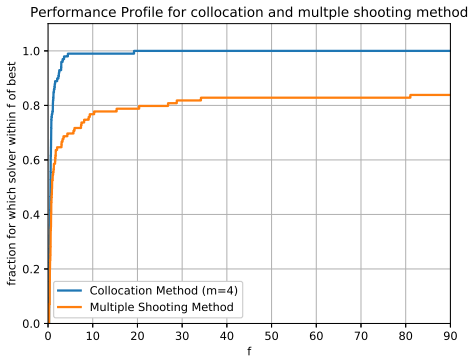
	Sequential	Parallel	Speedup
$T_{Alg\,4.2}$ (s)	31.95	4.56	7.01
$T_{Alg\,4.3}$ (s)	85.31	4.88	17.48
$T_{Alg\,4.4}$ (s)	3476.31	10.22	340.15
$T_{Alg\,4.5}$ (s)	1116.54	3.74	298.54
T (s)	4953.44	45.08	109.88

For each example, the table shows the computation time required for the sequential and parallel collocation solver and the parallel multiple shooting solver. Both parallel solvers are implemented using one single GPU. It can be seen that the parallel collocation solver has a much better computation efficiency than the parallel multiple shooting solver on those example OCPs.

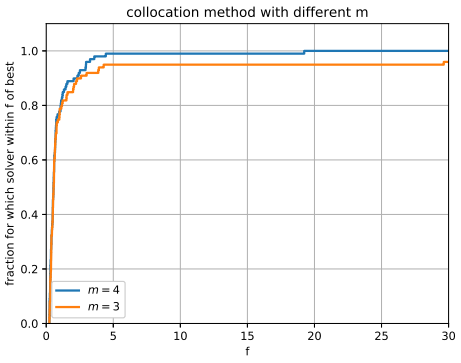
5.4.6 Performance Profiles

Four performance profiles shown in figure 5.11 are generated to better quantify and benchmark the performance of solvers on large problem set. The files compare the collocation solver and the multiple shooting solver both in parallel, and collocation solvers with different number of collocation points m_j used. The set of optimal control problems can be found at `https://github.com/UW-OCP/ocp_test_problems`. Gould and Scott [2016] points out that the performance can be better evaluated with only two solvers in single performance profile.

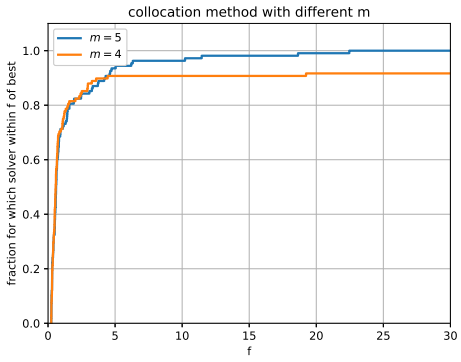
From figure 5.11(a), it can be seen that the parallel collocation solver is more robust and efficient compared with the parallel multiple shooting solver, where the collocation solver can solve more problems in shorter time. Figure 5.11(b) compares the performance between collocation solvers using $m_j = 3$ and $m_j = 4$ collocation points. This profile shows that



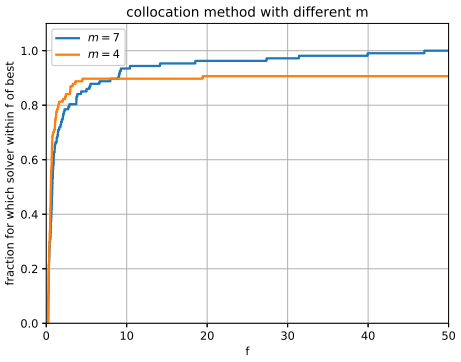
(a) Col vsMs



(b) $m = 3$ vs $m = 4$



(c) $m = 4$ vs $m = 5$



(d) $m = 4$ vs $m = 7$

Figure 5.11: Performance profiles between solvers

Table 5.7: Performance comparison between solvers

	Col, time (s)		Ms, time (s)
Example	Sequential	Parallel	Parallel
Ex. 5.4.1	7002.13	71.63	463.0
Ex. 5.4.2	80167.24	459.71	4019.4
Ex. 5.4.3	1315.62	43.62	214.2
Ex. 5.4.4	953.44	45.08	1428.3

although $m_j = 3$ collocation solver is more time efficient, while $m_j = 4$ solver is capable of solving more problems. Performance profile between $m_j = 4$ and $m_j = 5$ is presented in figure 5.11(c) and profile between $m_j = 4$ and $m_j = 7$ is exhibited in figure 5.11(d). The two profiles also give the result that solver using less collocation points ($m_j = 4$) saves computation time while solvers using more collocation points ($m_j = 5, 7$) are more robust on the example set. This motivates the work presented in chapter 6.

5.5 Conclusion

This chapter presents two fast and accurate GPU-based version of the solvers introduced in the chapter 3 and chapter 4. The numerical solvers are implemented using Python and CUDA with the use of GPU parallel computing capability. The algorithm implements the part that can be processed independently in parallel and decrease the running time for solving the problem significantly. Several concrete optimal control problem examples from open literatures are presented to show the accuracy and efficiency of the solvers. The solvers are able to solve high dimensional and highly nonlinear problems both accurately and efficiently.

Comparing the performance of the sequential and parallel implementation of the multiple shooting method, it can be seen there is a significant speedup in the parallel implementation.

For some complicated problems with many variables and complicated inequality constraints, the parallel multiple shooting algorithm is very efficient.

The special structure of the collocation algorithm are made full use of by designing adaptable CUDA kernels to employ the power of the GPU hardware resources and the use of the shared memory to further improve the performance. Significant and consistent speedups in the performance of the GPU based parallel collocation implementation are noticed through those examples using a single GeForce RTX 2070 SUPER GPU. The comparison of the efficiency and robustness between the solvers is also shown by the performance profiles in this chapter which show that the GPU based collocation solver in section 5.4 is more robust and efficient than the GPU based multiple shooting solver in section 5.3. With the increase of the computing ability of the hardware, a further reduction of computation time is also possible.

The result from the performance profiles conducted on the parallel collocation solver with different number of collocation points used showed that the solver more collocation points are more robust but less time efficient. The implementation developed here uses a global unified number of the collocation points. This result motivates the development of a collocation method with the ability to adaptively refine the number of collocation points used within each interval which is presented in the next chapter.

Chapter 6

COLLOCATION METHOD WITH ADAPTIVE MESH REFINEMENT

6.1 Introduction

In chapter 4, we introduced a collocation method to solve the BVP-DAEs. The method discretize the overall time interval $[t_{\mathrm{i}}, t_{\mathrm{f}}]$ into a mesh with N time nodes such that $t_{\mathrm{i}} = t_1 < t_2 < \ldots < t_N = t_{\mathrm{f}}$. Each divided time interval $[t_j, t_{j+1}],\ j = 1, 2, \ldots, N-1$ is further divided by m_j collocation points and a global unified number of collocation points m_j is used within each mesh interval. From the performance profiles shown in 5.4.6, we can see that solver with more collocation points are more robust than fewer points. The number of collocation points used is indeed controlling the degree of the underlying approximation polynomial of the problem. However, the increase in the number of collocation points used can increase the computations cubically or even more. The dominant part when using the collocation method are the Algorithms 4.4 and 4.5, where the LU decomposition of the matrix W_j is the most time consuming part. As the size complexity of the element W_j in the Jacobian (4.13) is $\sim \mathcal{O}(m_j)$, the time complexity of the LU decomposition of the matrix is $\sim \mathcal{O}(m^3)$. Therefore, we may waste lots of computation resources if we try to only increase the number of collocation points m_j globally. This leads to the need to developed a collocation method with the ability to adaptively refine the mesh of the problem where not only the mesh size can be varied but also the number of collocation points within each interval is not the same. The work first appeared in the domain of finite elements in mechanics and spectral methods in fluid dynamics. Babuška and Suri [1990, 1994], Gui and Babuška [1986] introduced the mathematical properties of h, p, hp methods for finite elements. Galvão et al. [2008] introduced an *hp* adaptive least-squares spectral element method (LS-SEM) for solving hyperbolic partial differential equations. Dorao and Jakobsen [2008] solved the population balance equation by a *hp* adaptive LS-SEM. There have been some existing research work in the optimal control community also and we give an overview of the previous related work

in the next section.

6.2 Overview

Numerical methods for solving optimal control problems (OCPs) are divided into direct and indirect methods [Betts, 1998, Rao, 2009].

In a direct method, the state and/or control of the OCP is discretized in some manner and the problem is transcribed into a nonlinear optimization problem or nonlinear programming problem (NLP)[Fabien, 1998, 2008, Goh and Teo, 1988, Von Stryk and Bulirsch, 1992]. In direct collocation methods, the overall time interval of the problem is divided into a number of sub mesh intervals [Herman and Conway, 1996, Herman, 1995]. The methods are called h methods as the degree of the polynomial used to approximate the variable within each mesh interval is fixed and the number of mesh intervals is varied in order to solve the problem. Recently, Gaussian quadrature collocation methods become popular when solving OCPs directly, where the state is approximated using a Lagrange polynomial at support points associated with the Gaussian quadrature formula [Elnagar et al., 1995, ELNAGAR and RAZZAGHI, 1997, Benson, 2005, Rao et al., 2010]. Originally, the direct Gaussian quadrature collocation methods were implemented as p methods using a single interval. For problems with smooth and well-behaved solutions, the direct Gaussian quadrature collocation methods have relative simple structures and can converge rapidly [Fornberg, 1998, Trefethen, 2000].

There are some mesh refinement strategies that use h methods or p methods alone when solving OCPs directly. For example, Gong et al. [2008] used a differentiation matrix to identify the discontinuities while using the p method to solve the problem. Grüne [1997], Munos and Moore [2002] employed a local splitting scheme on certain sub mesh intervals according to some splitting criteria. Although h methods have a long history and p methods have became prominent while solving certain types of problems, they both have their drawbacks. When trying to solve highly nonlinear and complicated problems with high accuracy, h methods may require particularly fine meshes while p methods may require some extremely high-degree polynomial approximations and either drawback can lead to a large computational burden in terms of both computation time and memory.

In order to reduce the computational burden, a great deal of research has been focusing on developing the combination of both methods which are so called hp direct collocation methods. In hp direct collocation methods, both the number of mesh intervals and the degree of the approximating polynomial within each mesh interval are allowed to vary simultaneously [Darby et al., 2011a,b, Patterson et al., 2015, Liu et al., 2015, Zhao and Li, 2020]. Darby et al. [2011a,b] presented hp adaptive methods where the error estimate is based on the difference between an approximation of the time derivative of the state and the right-hand side of the dynamics midway between the collocation points. It should be noted that the method from Darby et al. [2011a,b] created a great deal of noise in the error estimate, thereby making these approaches computationally intractable when a high-accuracy solution is desired. In Betts [2010], an error estimate was developed considering only the state in the differential algebraic equations (DAEs) by integrating the difference between an interpolation of the time derivative of the state and the right-hand side of the dynamics using the Romberg method. The error estimate developed in Betts [2010] was predicated on the use of a fixed-order method (trapezoid, Hermite–Simpson, Runge–Kutta) and computed a low-order approximation of the integral of the aforementioned difference. Patterson et al. [2015] developed an error estimate considering only the state variables based on the difference between the state interpolated on an increased number of Legendre–Gauss–Radau (LGR) points in each mesh interval and the state obtained by integrating the dynamics on the solution using the interpolated state and control. The paper also developed a ph mesh refinement method which can only increase the size of the mesh. Liu et al. [2015] introduced an adaptive mesh refinement method that had the ability to both increase and decrease the mesh size based on the error estimate from Patterson et al. [2015]. The mesh size was decreased by either reducing the degree of the approximating polynomial within a mesh interval or combining neighboring mesh intervals based on the power series representation. Zhao and Li [2020] presented an error estimation considering only the control variables based on the interpolation obtained from all the neighboring points from the mesh and developed an adaptive mesh method based on the multiresolution techniques (MRTs).

In an indirect method, the calculus of variations or the minimum principle is used to determine the first-order optimality conditions of the original OCP. These conditions are

then converted into a two-point boundary value problem (BVP) which normally involves DAEs. Numerical methods for solving BVP-DAEs include collocation methods [Ascher and Spiteri, 1994, Kierzenka and Shampine, 2001, Fabien, 2016a] and multiple shooting methods [Fabien, 2014a,b, Stoer and Bulirsch, 2013]. Those methods also request the overall time interval be discretized into multiple mesh intervals. In an indirect collocation method, the variables are parameterized using Lagrange polynomials where the degree of the polynomials are associated with the number of collocation points used within each mesh interval. While in an indirect multiple shooting method, initial guesses are made in each mesh interval where the variables are then integrated using some appropriate approach and the continuity conditions are requested at the inner mesh points. In chapter 2, we developed a method based on the slacked unconstrained penalty function approach for converting the OCP containing mixed control-state variable inequality constraints and parameters to a BVP involving index-1 DAEs. A multiple shooting method was developed to solve the BVP-DAEs where a fourth-order Rosenbrock-Wanner method is used to integrate the DAEs in chapter 3. Chapter 4 introduced an indirect collocation method to solve the same index-1 BVP-DAEs and in chapter 5, both the multiple shooting method and collocation method introduced were successfully accelerated by the GPU and Fabien [2016a] developed an indirect collocation method to solve the index-1 BVP-DAEs that arises directly from constrained OCPs. All those methods are h methods as only the number of the mesh intervals are allowed to vary and a global unified number of collocation points is used.

The main contribution of this chapter is the development of an adaptive mesh refinement method based on the indirect collocation method introduced in chapter 4 for solving the index-1 BVP-DAEs introduced in chapter 2. The chapter develops a novel error estimation for the collocation solution obtained, where the related errors for both differential and algebraic variables are interconnected by a boarded almost block diagonal (BABD) [Amodio et al., 2000] linear system. By solving the BABD system, the upper bound of the error for the variables can be obtained, which is then used for the mesh refinement process. The adaptive mesh refinement scheme is based on the power series representation of the variables [Liu et al., 2015], that allows the number of mesh intervals, the width of each interval, and the number of collocation points used in each interval to vary simultaneously, An adapted

GPU-accelerated implementation of the algorithm using Python [Van Rossum and Drake Jr, 1995] and CUDA [Nickolls et al., 2008, Luebke, 2008, Stoer and Bulirsch, 2013] is contributed and the parallel code is shown to work very well on various OCPs. Three concrete OCP examples are shown to prove that the adaptive mesh refinement method developed is more computationally efficient and robust and is able to produce smaller meshes for a given numerical accuracy tolerance when compared with the original fixed-order h method.

6.3 Motivation

This section provides an example to better motivate the development of an ph adaptive collocation method. Consider the following two first-order differential equations on the interval $\tau \in [-1, 1]$:

$$\frac{dy_1}{d\tau} = \pi \cos(\pi\tau), -1 \leq \tau \leq +1, y_1(-1) = y_{10}. \tag{6.1}$$

$$\frac{dy_2}{d\tau} = \begin{cases} 0, & -1 \leq \tau < -1/2, \\ \pi \cos(\pi\tau), & -1/2 \leq \tau \leq +1/2, \\ 0, & +1/2 < \tau \leq 1, \end{cases} \quad y_2(-1) = y_{20}. \tag{6.2}$$

The solution to the differential equations (6.1) and (6.2) are

$$y_1(\tau) = y_{10} + \sin(\pi\tau), -1 \leq \tau \leq +1. \tag{6.3}$$

$$y_2(\tau) = \begin{cases} y_{20}, & -1 \leq \tau < -1/2, \\ y_{20} + \sin(\pi\tau), & -1/2 \leq \tau \leq +1/2, \\ y_{20}, & +1/2 < \tau \leq 1. \end{cases} \tag{6.4}$$

Suppose we want to approximate the solutions to the differential equations (6.1) and (6.2) using the three different methods based on the collocation method described in chapter 4: (*i*) an h method using N equally spaced mesh intervals where N is allowed to vary and 4 collocation points are used within each time interval; (*ii*) a p method where the function is approximated using m collocation points on the interval $[-1, 1]$ and m is allowed to vary; and (*iii*) an ph method where both the number of mesh intervals N and the number of collocation points m within each interval are allowed to vary simultaneously. The

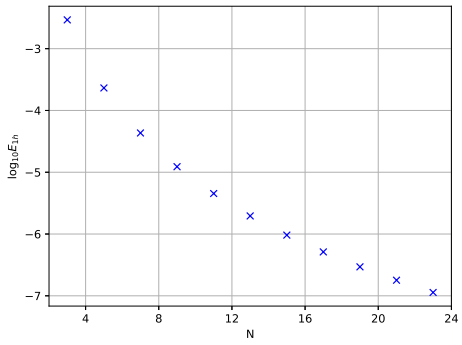
functions $y_k(\tau)$, $(k = 1, 2)$ are approximated using the Lagrange polynomials as equation (4.6). Following the format in the collocation method developed in section 4.1, $\tau_{j,i}$ denotes the ith collocation point within the interval $[t_j, t_{j+1}]$. Let $Y^k_{j,i}$ denote the analytic solution of y_k at $\tau_{j,i}$.

Define the maximum absolute error in the solution of the differential equation as

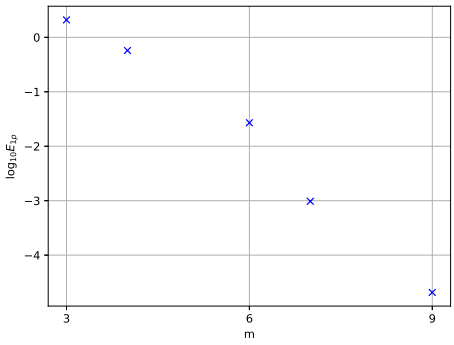
$$E_k = \max_{j\in[1,\ldots,N],i\in[1,\ldots,m]} \left| Y^k_{j,i} - y_k(\tau_{j,i}) \right|, \ (k = 1, 2).$$

Figure 6.1(a) and 6.1(b) show the result of the base 10 logarithm of E_1 as a function of N for the h method and as a function of m for the p method. First, it is seen that because $y_1(\tau)$ is a smooth function, both h and p methods converge fast to the analytic solution. Figure 6.1(c) and 6.1(d) show the result of the base 10 logarithm of E_2. Unlike function $y_1(\tau)$, function $y_2(\tau)$ is continuous but not smooth. As a result, the h method converges faster than the p method, because no single polynomial on the domain $[-1, 1]$ is able to approximate the solution to equation (6.2) as accurately as a piecewise polynomial. However, while the h method converges faster than the p method when approximating the solution of equation (6.2), it is seen that the h method does not converge as accurate as it does when approximating the solution to equation (6.1).

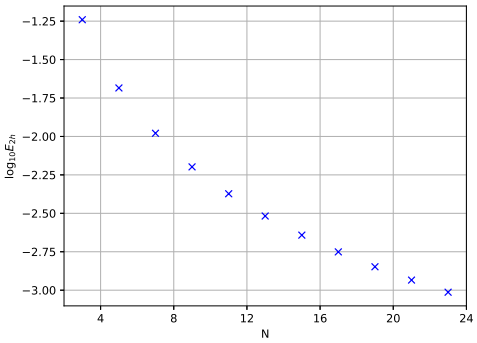
Given the analysis above, suppose now that the solution to equation (6.2) is approximated using the ph collocation method. Assume further that the ph method is constructed such that the time interval $[-1, 1]$ is divided into three mesh intervals as $[-1, -1/2]$, $[-1/2, +1/2]$, and$[+1/2, 1]$, and m_1, m_2, and m_3 collocation points, respectively, are used within each mesh interval. Suppose $m_1 = m_2 = 3$, and m_2 is allowed to vary. This is because solution $y_2(\tau)$ is a constant in the first and third mesh intervals, so E_2 only depends on m_2. Figure 6.1(e) shows the error E^{hp}_2 in $y_2(\tau)$. It can be seen that the result is much better than using only h or p method alone. Therefore, while an h method may outperform a p method on a problem whose solution is not smooth, it is possible to improve the convergence rate by using a ph adaptive method. This further motivates the work to develop an adaptive collocation method and the more motivations can be referred to Darby et al. [2011a], Patterson et al. [2015].



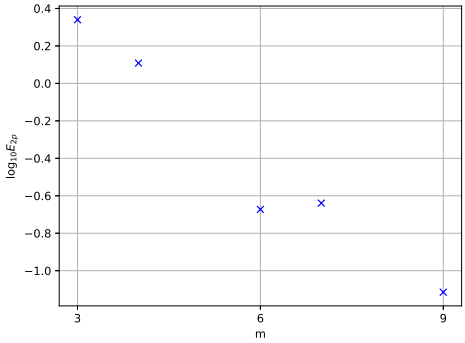
(a) E_{1h} v.s. N using h method.



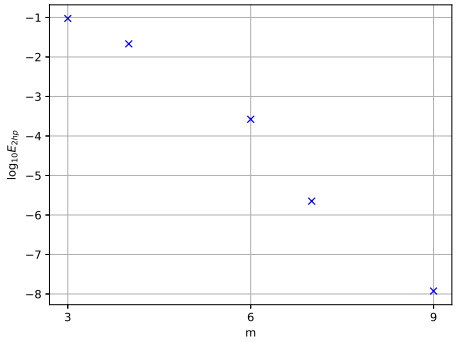
(b) E_{1p} v.s. m using p method.



(c) E_{2h} v.s. N using h method.



(d) E_{2p} v.s. m using p method.



(e) E_{2hp} v.s. m using hp method.

Figure 6.1: Absolute errors in solutions of equation (6.2) using h, p, hp method.

6.4 Collocation Method as the Basis

As the indirect collocation method to solve the BVP-DAEs (2.38)-(2.40) is already introduced in chapter 4, we omit the re-introduction here. The only thing to notice is that a global unified number of collocation points is used before and here we do not have that requirement. This may introduce some non-uniformity of the dimension in the block term of the residual (4.10) and the block matrix term in the Jacobian (4.13). However, notice that the block terms in the reduced BABD system (4.18) has the uniform size which does not depend on the number of collocation points used. Due to this feature, we only need to modify the CUDA implementation a little bit to incorporate the non-uniformity in the residual and Jacobian construction. And due to the overhead of the CUDA programming model, 1D CUDA grids are used for those implementations instead of the 2D grids introduced in section 5.4.2.

6.5 Adaptive Mesh Refinement Method

In this section, we present the developed adaptive mesh refinement for using the indirect collocation method. Section 6.5.1 introduces the method to estimate the error in the computed collocation solution, while section 6.5.2 presents the detailed mesh scheme based on the estimated error obtained in section 6.5.1. The adaptive mesh refinement collocation algorithm is summarized in section 6.5.3.

6.5.1 Error estimation

In this section, the estimate of the relative error in the collocation solution is derived. Betts [2010] developed a method to estimate the error of the problem containing DAEs. However, in that approach, only the differential variables are considered and the related error is obtained using the Romberg method. Patterson et al. [2015] introduced an approach by integrating the dynamics considering only differential variables using an increased number of LGR points and the integration is done by using the Gaussian quadrature. Both methods ignore the error related to the algebraic variables.

The error estimate developed in this thesis considers the error both in differential and

algebraic variables as both variables are interconnected in the DAEs. The chapter uses the same setup as Patterson et al. [2015] where an increased number of the collocation points is used. The key idea is that the error related to the variables can be obtained by solving a related BABD (boarded almost block diagonal) linear system. Then, the associated error can be estimated using the Lobatto quadrature method.

Assume the residual equation (4.10) has been solved on the fixed mesh $\mathcal{S}_j$, $j = 1, 2, \ldots, N-1$ with m_j collocation points. And notice that m_j does not to be equal to m_i where $j \neq i$ here. The error of the solution in each mesh interval $\mathcal{S}_j$ is estimated by using a set of $n_j = m_j + 1$ Lobatto collocation points in the interval, where $\bar{\tau}_{jl} = t_j + \bar{c}_l^{(j)}\delta_j$, $l = 1, \ldots, n_j$ and $0 \leq \bar{c}_l^{(j)} \leq 1$ are the associated collocation points.

The differential variables at the mesh points t_j, $j = 1, \ldots, N$ are obtained via equation (4.4) which are denoted as $\bar{y}_j$. The derivatives of the differential variables and algebraic variables at the new collocation points $\bar{\tau}_{jl}$, $l = 1, \ldots, n_j$ are obtained using the Lagrange polynomial approximation via equations (4.2) and (4.9) which are denoted as $(\dot{\bar{y}}_1^{(j)}, \ldots, \dot{\bar{y}}_{n_j}^{(j)})$ and $(\bar{z}_1^{(j)}, \ldots, \bar{z}_{n_j}^{(j)})$. By using the Lagrange Polynomial approximations with the n_j collocation points, the differential variables can be obtained as

$$\bar{y}_l^{(j)} = \bar{y}^{(j)}(\bar{\tau}_{j,l}) = \bar{y}_j + \delta_j \sum_{k=1}^{n_j} a_{lk}\, \dot{\bar{y}}_k^{(j)},\ l = 1, \ldots, n_j \tag{6.5}$$

$$\bar{y}_{j+1} = \bar{y}^{(j)}(t_{j+1}) = \bar{y}_j + \delta_j \sum_{l=1}^{m_j} b_l\, \dot{\bar{y}}_l^{(j)}. \tag{6.6}$$

Let $y^{(j)}(t)$, $z^{(j)}(t)$, p be the true solution to the BVP-DAEs, the values of the differential equation (2.38) at the newly allocated collocation points of the true solution can be expanded using Taylor series with respect to the approximated solution as

$$\begin{aligned} h(y_l^{(j)}, z_l^{(j)}, p) =& h(\bar{y}_l^{(j)}, \bar{z}_l^{(j)}, \bar{p}) + \frac{\partial h}{\partial y}(y_j - \bar{y}_j) + \frac{\partial h}{\partial y}\delta_j \sum_{k=1}^{n_j} a_{ik}\,(\dot{y}_k^{(j)} - \dot{\bar{y}}_k^{(j)}) \\ &+ \frac{\partial h}{\partial z}(z_l^{(j)} - \bar{z}_l^{(j)}) + \frac{\partial h}{\partial p}(p - \bar{p}) + h.o.t. \end{aligned}$$

where $h.o.t.$ denotes the second order and higher order terms.

The derivative of the differential variables at the collocation points can be represented

as

$$\dot{y}_l^{(j)} = \dot{\bar{y}}_l^{(j)} + (\dot{y}_l^{(j)} - \dot{\bar{y}}_l^{(j)}).$$

Because the true solution satisfies the DAEs at the collocation points as $\dot{y}_l^{(j)} = h(y_l^{(j)}, z_l^{(j)}, p)$, by applying the Taylor series expansion and ignoring the higher order terms, we can obtain the fact that

$$\begin{aligned}
\dot{\bar{y}}_l^{(j)} - h(\bar{y}_l^{(j)}, \bar{z}_l^{(j)}, \bar{p}) =& \frac{\partial h}{\partial y}(y_j - \bar{y}_j) + \frac{\partial h}{\partial y}\delta_j \sum_{k=1}^{n_j} a_{lk}\,(\dot{y}_k^{(j)} - \dot{\bar{y}}_k^{(j)}) \\
&+ \frac{\partial h}{\partial z}(z_l^{(j)} - \bar{z}_l^{(j)}) + \frac{\partial h}{\partial p}(p - \bar{p}) - (\dot{y}_l^{(j)} - \dot{\bar{y}}_l^{(j)}) \\
=& \frac{\partial h}{\partial y}\Delta y_j + (\frac{\partial h}{\partial y}\delta_j a_{ll} - I)\Delta\dot{y}_l^{(j)} + \frac{\partial h}{\partial y}\delta_j \sum_{\substack{k=1 \\ k\neq l}}^{n_j} a_{lk}\,\Delta\dot{y}_k^{(j)} \\
&+ \frac{\partial h}{\partial z}\Delta z_l^{(j)} + \frac{\partial h}{\partial p}\Delta p.
\end{aligned}$$

Then, considering the continuity condition of the differential variables, we have that

$$\begin{aligned}
y_{j+1} &= y_j + \delta_j \sum_{l=1}^{n_j} b_l\,\dot{y}_{jl} \\
\bar{y}_{j+1} + (y_{j+1} - \bar{y}_{j+1}) &= \bar{y}_j + (y_j - \bar{y}_j) + \delta_j \sum_{l=1}^{n_j} b_l\ \dot{\bar{y}}_l^{(j)} + \delta_j \sum_{l=1}^{n_j} b_l\,(\dot{y}_l^{(j)} - \dot{\bar{y}}_l^{(j)}) \\
\bar{y}_j + \delta_j \sum_{l=1}^{n_j} b_l\,\dot{\bar{y}}_l^{(j)} - \bar{y}_{j+1} &= (y_{j+1} - \bar{y}_{j+1}) - \delta_j \sum_{l=1}^{n_j} b_l\,(\dot{y}_l^{(j)} - \dot{\bar{y}}_l^{(j)}) - (y_j - \bar{y}_j) \\
&= \Delta y_{j+1} - \delta_j \sum_{l=1}^{n_j} b_l\,\Delta\dot{y}_l^{(j)} - \Delta y_j.
\end{aligned}$$

Next, by expanding the algebraic equation using the Taylor series expansion with respect to the approximate solution at the collocation points, and neglecting higher-order terms, we get

$$\begin{aligned}
g(y_l^{(j)}, z_l^{(j)}, p) =& g(\bar{y}_l^{(j)}, \bar{z}_l^{(j)}, \bar{p}) + \frac{\partial g}{\partial y}(y_j - \bar{y}_j) + \frac{\partial g}{\partial y}\delta_j \sum_{k=1}^{n_j} a_{lk}\,(\dot{y}_k^{(j)} - \dot{\bar{y}}_k^{(j)}) \\
&+ \frac{\partial g}{\partial z}(z_l^{(j)} - \bar{z}_l^{(j)}) + \frac{\partial g}{\partial p}(p - \bar{p}).
\end{aligned}$$

Taking into account that $g(y_l^{(j)}, z_l^{(j)}, p) = 0$, the equation can be simplified as

$$-g(\bar{y}_l^{(j)}, \bar{z}_l^{(j)}, \bar{p}) = \frac{\partial g}{\partial y}\Delta y_j + \frac{\partial g}{\partial y}\delta_j \sum_{k=1}^{n_j} a_{lk}\, \Delta \dot{y}_k^{(j)} + \frac{\partial g}{\partial z}\Delta z_l^{(j)} + \frac{\partial g}{\partial p}\Delta p.$$

By expanding the boundary condition in the same manner, we can get

$$-r(\bar{y}_1, \bar{y}_N, \bar{p}) = \frac{\partial r}{\partial y_1}\Delta y_1 + \frac{\partial r}{\partial y_N}\Delta y_N + \frac{\partial r}{\partial p}\Delta p.$$

By forming all those equations at all the n_j new allocated collocation points in $\mathcal{S}_j$, $j = 1, \ldots, N-1$, we can get a BABD linear system in the same form as equation (4.12). By solving the BABD system, we can obtain the error with the increased number of collocation points as $[\Delta y_j, \Delta \dot{y}_l^{(j)}, \Delta z_l^{(j)}, \Delta y_j, \Delta p]$, $j = 1, \ldots, N-1$, $l = 1, \ldots, n_j$.

Consider the error of the differential variables within $\mathcal{S}_j$ first. By integrating the differential variables directly we get

$$\begin{aligned} y^{(j)}(t_{j+1}) - \tilde{y}^{(j)}(t_{j+1}) &= y_j + \int_{t_j}^{t_{j+1}} \dot{y}^{(j)}(t)\, dt - \bar{y}_j + \int_{t_j}^{t_{j+1}} \dot{\tilde{y}}^{(j)}(t)\, dt \\ &= \Delta y_j + \int_{t_j}^{t_{j+1}} \Delta \dot{y}^{(j)}(t)\, dt. \end{aligned}$$

Taking absolute values of each component, we then obtain the bound as

$$\begin{aligned} |y^{(j)}(t_{j+1}) - \tilde{y}^{(j)}(t_{j+1})| &= |\Delta y_j + \int_{t_j}^{t_{j+1}} \Delta \dot{y}^{(j)}(t)\, dt| \\ &\le |\Delta y_j| + |\int_{t_j}^{t_{j+1}} \Delta \dot{y}^{(j)}(t)\, dt| \\ &\le |\Delta y_j| + \int_{t_j}^{t_{j+1}} |\Delta \dot{y}^{(j)}(t)|\, dt. \end{aligned}$$

As the error term of the derivatives of the differential variables are computed at the Lobatto collocation points, the integral $\int_{t_j}^{t_{j+1}} |\Delta \dot{y}(t)|\, dt$ can be estimated using the Lobatto quadrature. So the error can be approximated and bounded as

$$|y^{(j)}(t_{j+1}) - \tilde{y}^{(j)}(t_{j+1})| \le |\Delta y_j| + \delta_j \sum_{l=1}^{n_j} w_l |\Delta \dot{y}_l^{(j)}| = E^{(j,y)},$$

where w_j is the associated Lobatto quadrature weight.

Denote $E_i^{(j,y)}$ as the ith element of $E^{(j,y)}$. The error can be scaled as

$$\bar{E}_i^{(j,y)} = \frac{E_i^{(j,y)}}{1+\omega_i^{(y)}}, i = 1, \ldots, n_y, \tag{6.7}$$

where the scale weight is

$$\omega_i^{(y)} = \max_{\substack{j=1,\ldots,N-1,\\ l=1,\ldots,m_j}} [|\dot{\tilde{y}}_{l,i}^{(j)}|, |\tilde{y}_{j,i}|], \tag{6.8}$$

and $\dot{\tilde{y}}_{l,i}^{(j)}$, $\tilde{y}_{j,i}$ denote the ith element of $\dot{\tilde{y}}_l^{(j)}$ and $\tilde{y}_j$.

Then, the error associated with the algebraic variables can be obtained by taking the maximum of the error on the collocation points as

$$E_i^{(j,z)} = \max_{l=1,\ldots,n_j} |\Delta z_{l,i}^{(j)}|, i = 1, \ldots, n_z, \tag{6.9}$$

By normalizing the absolute error, we can obtain the relative error of the algebraic variables as

$$\bar{E}_i^{(j,z)} = \frac{E_i^{(j,z)}}{1+\omega_i^{(z)}}, \tag{6.10}$$

where the scale weight is defined as

$$\omega_i^{(z)} = \max_{\substack{j=1,\ldots,N-1,\\ l=1,\ldots,m_j}} |\tilde{z}_{l,i}^{(j)}|. \tag{6.11}$$

The maximum relative error in $\mathcal{S}_j$ is then defined as

$$\bar{E}^{(j)} = \max\{\max_{i=1,\ldots,n_y} \bar{E}_i^{(j,y)}, \max_{i=1,\ldots,n_z} \bar{E}_i^{(j,z)}\}. \tag{6.12}$$

6.5.2 Mesh refinement scheme

When the residual equation (4.10) is solved with a fixed mesh, the estimated maximum relative error of the current mesh $\mathcal{S}_j$, $j = 1, 2, \ldots, N-1$ is computed using equation (6.12). If the estimated maximum relative error $\bar{E}^{(j)}$ in $\mathcal{S}_j$ is above the desired mesh refinement threshold tolerance ϵ_a, then the current mesh interval is updated by first trying to increase the number of collocation points used in the interval. If the increased number of collocation points exceeds the maximum allowable number of collocation points in a single interval,

the mesh interval is divided into multiple subintervals. This is a so called "p-then-h" mesh refinement scheme when dealing with intervals having errors beyond the threshold tolerance [Patterson et al., 2015]. Moreover, if $\bar{E}^{(j)}$ is below the desired mesh refinement threshold tolerance ϵ_r, the current mesh interval $\mathcal{S}_j$ is updated by either decreasing the number of collocation points used or trying to merge with the adjacent mesh intervals [Liu et al., 2015]. If the estimated maximum relative error from two adjacent mesh intervals are all below ϵ_r and the number of collocation points in both intervals reaches the minimum allowable number of collocation points simultaneously, an attempt to merge the two mesh intervals is performed.

Method for increasing the number of collocation points

Assume that the estimated error $\bar{E}_j$ in a certain mesh interval $\mathcal{S}_j$ with m_j collocation points does not meet the desired mesh refinement threshold tolerance ϵ_a. Then, the number of collocation points used is increased to meet the tolerance ϵ_a. Denote $m_{\min}$ and $m_{\max}$ as the minimum and maximum allowable number of collocation points used within any mesh interval. It is shown in chapter 4 that the error of the collocation solution is of the order $\mathcal{O}(\delta_j^m)$. If the width of the mesh interval remains the same between the two meshes, the additional required number of collocation points M_j can be computed by

$$\begin{aligned} \bar{E}_j &= \mathcal{O}(\delta_j^{m_j}), \\ \epsilon_a &= \mathcal{O}(\delta_j^{m_j+M_j}), \end{aligned}$$

which implies that

$$M_j = \log_{\delta_j}(\frac{\epsilon_a}{\bar{E}_j}). \tag{6.13}$$

Because the number of collocation points in a mesh interval must be an integer, we need to round up the result obtained from equation (6.13) to get a valid number as

$$M_j = \left\lceil \log_{\delta}(\frac{\epsilon_a}{\bar{E}_j}) \right\rceil. \tag{6.14}$$

Notice that $M_j \geq 1$ because we only employ the method when $\bar{E}_j > \epsilon_a$. Thus the updated number of collocation points of the mesh interval is $\tilde{m}_j = m_j + M_j$. However, if the number

of the updated collocation points exceeds $m_{\max}$, the mesh interval is divided into multiple subintervals by the method introduced in section 6.5.2.

Method for dividing the mesh interval

When the estimated error $\bar{E}_j$ in a mesh interval $\mathcal{S}_j = [t_j, t_{j+1}]$ does not meet the desired threshold tolerance and the computed increased number of collocation points $\tilde{m}_j = m_j + M_j$ exceeds the maximum allowable collocation points, which is $\tilde{m}_j > m_{\max}$, our method divides the mesh interval into multiple smaller intervals.

The method follows the following scheme. The needed collocation points $\tilde{m}_j$ of the original mesh interval are divided into multiple subintervals equipped with the minimum allowable collocation points $m_{\min}$. The number of subintervals A_j is obtained as

$$A_j = \frac{\tilde{m}_j}{m_{\min}}. \tag{6.15}$$

Note that the number of the divided subintervals must also be an integer. We use the same roundup technique in equation (6.15) to get a valid number as

$$A_j = \left\lceil \frac{\tilde{m}_j}{m_{\min}} \right\rceil. \tag{6.16}$$

Using this scheme, it is ensured that the number of collocation points in the interval $[t_j, t_{j+1}]$ remains the same no matter whether the mesh interval uses more collocation points or is divided into multiple subintervals. The scheme is so called "p-then-h" mesh refinement scheme as the method initially tries to increase the number of collocation points in the interval which actually increases the order of the underlying approximation polynomial ("p" scheme). After the number of collocation points reaches the maximum allowable, the original mesh interval is divided into multiple subintervals where the width of each subinterval is smaller ("h" scheme). The "h" scheme is only applied when the "p" scheme reaches its limit whereas afterwards the "p" scheme starts again.

Method for decreasing the number of collocation points

When the estimated error $\bar{E}_j$ in $\mathcal{S}_j$ is below the desired mesh refinement threshold ϵ_r, we try to decrease the mesh size by either decreasing the number of collocation points in the

mesh interval or merging the two adjacent mesh intervals. The two schemes are introduced in section 6.5.2 and 6.5.2. The method used here is similar to Liu et al. [2015], while in that paper only the state variables are considered and here we consider a more complex case.

When the number of collocation points m_j is above $m_{\min}$, we first try to decrease the number of collocation points in the mesh interval while still keeping the estimated error below the threshold ϵ_r. The decision is made based on using the power series representation of the Lagrange polynomial approximation of the differential and algebraic variables. The Lagrange polynomial from equation (4.3) is a polynomial of order m_j-1 and can be rewritten as a power series in the form as

$$L_l^{(j)}(\varsigma) = \prod_{k=1,k\neq l}^{m_j} \frac{\varsigma - c_k}{c_l - c_k} = \sum_{k=0}^{m_j-1} \tilde{a}_{kl}^{(j)} \varsigma^k. \tag{6.17}$$

Similarly, the integration of the Lagrange polynomial from equation (4.5) can be written as a power series of order m_j as

$$I_l^{(j)}(\varsigma) = \int_0^{\varsigma} L_l^{(j)}(\eta)\, d\eta = \sum_{k=1}^{m_j} \tilde{b}_{kl}^{(j)} \varsigma^k. \tag{6.18}$$

Coefficients the in power series representation in equation (6.17) and (6.18) only depend on the number of Lobatto IIIA collocation points used in each mesh interval and can be pre-computed and saved before the algorithm is applied to the specific problem.

Applying equation (6.18) to equation (4.4), we obtain that

$$\begin{aligned} \tilde{y}^{(j)}(t) &= \tilde{y}_j + \delta_j \sum_{l=1}^{m_j} I_l^{(j)}(\frac{t-t_j}{\delta_j})\, \dot{\tilde{y}}_l^{(j)} \\ &= \tilde{y}_j + \sum_{k=1}^{m_j} \tilde{c}_k^{(j,y)} (\frac{t-t_j}{\delta_j})^k,\ t \in [t_j, t_{j+1}], \end{aligned} \tag{6.19}$$

$$\tilde{c}_k^{(j,y)} = \delta_j \sum_{l=1}^{m_j} \tilde{b}_{kl}^{(j)} \dot{\tilde{y}}_l^{(j)}, \tag{6.20}$$

where $\frac{t-t_j}{\delta_j} \leq 1$, $t \in \mathcal{S}_j = [t_j, t_{j+1}]$, and $(\frac{t-t_j}{\delta_j})^k \leq 1$, $k = 1, \ldots, m_j$. As we decrease the number of collocation points in the mesh interval, we are actually dropping the power series terms in descending order starting from order m_j in equation (6.19). Let $\tilde{y}_i^{(j)}(t)$ denote the ith element in $\tilde{y}^{(j)}(t)$ and $\tilde{c}_{k,i}^{(j,y)}$ denote the ith element in $\tilde{c}_k^{(j,y)}$, $i = 1, \ldots, n_y$. Therefore,

as we drop the kth order term, the pointwise absolute error in $\tilde{y}_i^{(j)}(t)$ is at most $|\tilde{c}_{k,i}^{(j,y)}|$. As the estimated error from equation (6.7) is in a relative scale, normalizing the coefficients $\tilde{c}_{jk}^{y}$ is also necessary for comparison with the mesh refinement threshold ϵ_r. The normalizing quantity used is the same as equation (6.8).

Thus, in terms of the differential variables, starting from the highest order term in the power series in descending order, the term is allowed to be removed as long as $\frac{|\tilde{c}_{k,i}^{(j,y)}|}{1+\omega_i^{(y)}} < \epsilon_r$, $i = 1, \ldots, n_y$.

Next, consider the estimated error related to the algebraic variables. Applying equation (6.17) to equation (4.9), the approximation of the algebraic variables can be rewritten as the power series of order $m_j - 1$ as

$$\begin{aligned}
\tilde{z}^{(j)}(t) &= \sum_{l=1}^{m_j} L_l^{(j)}(\frac{t-t_j}{\delta_j})\ \tilde{z}_l^{(j)}, \\
&= \sum_{k=0}^{m_j-1} \tilde{c}_k^{(j,z)}(\frac{t-t_j}{\delta_j})^k,\ t \in [t_j, t_{j+1}], \qquad (6.21) \\
\tilde{c}_k^{(j,z)} &= \sum_{l=1}^{m_j} \tilde{a}_{kl}^{(j)} \tilde{z}_l^{(j)}. \qquad (6.22)
\end{aligned}$$

The decrease of the number of collocation points drops the power series term in equation (6.21) in the same way as the differential variables. The normalizing quantity used is the same as equation (6.11). Then, so as long $\frac{|\tilde{c}_{k,i}^{(j,z)}|}{1+\omega_i^{(z)}} < \epsilon_r$, $i = 1, \ldots, n_z$, we are allowed to drop the kth order term in the power series expansion without violating the threshold tolerance for the estimated error in terms of the algebraic variables.

Therefore, for $\mathcal{S}_j$ originally equipped with $m_j > m_{\min}$ collocation points, when the estimated error is below the threshold ϵ_r, we start from the highest order in the power series expansion and check whether dropping the term keeps the upper bound of the pointwise error still below the threshold tolerance for both differential variables and algebraic variables. If the bounded pointwise error is below the threshold tolerance for both variables, we keep dropping the power series term, which is equal to decrease the number of collocation points in $\mathcal{S}_j$ until the minimum allowable number is reached or the upper bound of the pointwise error is beyond the threshold tolerance.

Method for merging adjacent mesh intervals

When the number of collocation points in a mesh interval reaches the minimum allowable number $m_{\min}$, the way to reduce the mesh size is by trying to merge the mesh interval $\mathcal{S}_j$ with the adjacent mesh interval $\mathcal{S}_{j+1}$ if both estimated errors are below the threshold tolerance ϵ_r. The intervals are merged based on that if the polynomial representation from the bigger interval is extended to the smaller interval, the pointwise difference of the estimated error resulting from the polynomial extension is below the tolerance still. Notice that if the number of collocation points in two intervals are not the same, we can not perform the merge as the polynomial degrees in the two intervals are not the same. Due to the continuity condition, the variables are the same at t_{j+1} for the two intervals. Intuitively, if we extend the polynomial from one interval to the other, the biggest difference should happen at the end points, either t_j or t_{j+2}. Here, we present a method to derive an upper bound of the pointwise difference of the estimated error due to the substitution of the extension polynomial for the original polynomial which is valid through the entire smaller mesh interval.

Consider two adjacent mesh intervals $\mathcal{S}_j = [t_j, t_{j+1}]$ and $\mathcal{S}_{j+1} = [t_{j+1}, t_{j+2}]$ whose estimated errors are both below the desired tolerance ϵ_r and the number of collocation points used are both the minimum allowable as $m_j = m_{j+1} = m_{\min}$. As the two intervals are connected at t_{j+1}, we first unify the representations of the variables in both intervals in terms of t_{j+1} which helps bound the difference between the extended and the original polynomials in the smaller interval. This requests the expansion of the the Lagrange polynomial representation in $\mathcal{S}_j$ using the end point t_{j+1} instead of the start point t_j as illustrated before. The Lagrange polynomials representation (4.3) can be expanded in terms of the end point as

$$L_l^{(j)}(\varsigma) = \sum_{k=0}^{m_j-1} \hat{a}_{kl}^{(j)}(\varsigma - 1)^k, \tag{6.23}$$

Then, the integration of the Lagrange polynomial from equation (4.5) can be expanded in terms of the end point as

$$I_l^{(j)}(\varsigma) = \sum_{k=1}^{m_j} \hat{b}_{kl}^{(j)}(\varsigma - 1)^k. \tag{6.24}$$

Apply the power series expansion of the Lagrange polynomial in terms of the end point using equation (6.24) to equation (4.4) in $\mathcal{S}_j$ and the power series expansion in terms of the start point as equation (6.19) in $\mathcal{S}_{j+1}$, we get

$$\tilde{y}^{(j)}(t) = \tilde{y}_{j+1} + \sum_{k=1}^{m_j} \bar{c}_k^{(j,y)} (\frac{t - t_{j+1}}{\delta_j})^k,\ t \in [t_j, t_{j+1}], \tag{6.25}$$

$$\bar{c}_k^{(j,y)} = \delta_j \sum_{l=1}^{m_j} \hat{b}_{kl}^{(j)} \dot{\tilde{y}}_l^{(j)}, \tag{6.26}$$

$$\tilde{y}^{(j+1)}(t) = \tilde{y}_{j+1} + \sum_{k=1}^{m_j} \bar{c}_k^{(j+1,y)} (\frac{t - t_{j+1}}{\delta_{j+1}})^k,\ t \in [t_{j+1}, t_{j+2}], \tag{6.27}$$

$$\bar{c}_k^{(j+1,y)} = \delta_{j+1} \sum_{l=1}^{m_j} \tilde{b}_{kl}^{(j+1)} \dot{\tilde{y}}_l^{(j+1)}. \tag{6.28}$$

The power series representations of the differential variables in two intervals now only differ with the denominator in the series as δ_j is used in equation (6.25) while δ_{j+1} is used in equation (6.27). The difficulty can be overcame by defining $\bar{\delta}_j = \min\{\delta_j, \delta_{j+1}\}$ to unify the representations as

$$\tilde{y}^{(j)}(t) = \tilde{y}_{j+1} + \sum_{k=1}^{m_j} \bar{d}_k^{(j,y)} (\frac{t - t_{j+1}}{\bar{\delta}_j})^k,\ t \in [t_j, t_{j+1}], \tag{6.29}$$

$$\bar{d}_k^{(j,y)} = \bar{c}_k^{(j,y)} (\frac{\bar{\delta}_j}{\delta_j})^k, \tag{6.30}$$

$$\tilde{y}^{(j+1)}(t) = \tilde{y}_{j+1} + \sum_{k=1}^{m_j} \bar{d}_k^{(j+1,y)} (\frac{t - t_{j+1}}{\bar{\delta}_j})^k,\ t \in [t_{j+1}, t_{j+2}], \tag{6.31}$$

$$\bar{d}_k^{(j+1,y)} = \bar{c}_k^{(j+1,y)} (\frac{\bar{\delta}_j}{\delta_{j+1}})^k. \tag{6.32}$$

As the power terms are the same in two intervals, the difference between the differential variables between the two representations can be obtained as

$$\tilde{y}^{(j)}(t) - \tilde{y}^{(j+1)}(t) = \sum_{k=1}^{m_j} (\bar{d}_k^{(j,y)} - \tilde{d}_k^{(j+1,y)}) (\frac{t - t_{j+1}}{\bar{\delta}_j})^k. \tag{6.33}$$

Because $\frac{t - t_{j+1}}{\bar{\delta}_j} \leq 1$ is always true in the smaller interval, the difference can be bounded as

$$\tilde{y}^{(j)}(t) - \tilde{y}^{(j+1)}(t) \leq \sum_{k=1}^{m_j} (\bar{d}_k^{(j,y)} - \tilde{d}_k^{(j+1,y)}), t \in \bar{\mathcal{S}}_j, \tag{6.34}$$

where $\bar{\mathcal{S}}_j$ is the smaller interval of the two. The pointwise difference can be bounded as

$$|\tilde{y}_i^{(j)}(t) - \tilde{y}_i^{(j+1)}(t)| \leq \sum_{k=1}^{m_j} |\tilde{d}_{k,i}^{(j,y)} - \tilde{d}_{k,i}^{(j+1,y)}|,\ i = 1, \ldots, n_y,\ t \in \bar{\mathcal{S}}_j. \tag{6.35}$$

Equation (6.35) provides the upper bound for the pointwise difference in the differential variables in the smaller interval. To compare with the threshold tolerance ϵ_r, it is necessary to normalize the absolute difference to get a upper bound of the relative difference between the differential variables. Scaling equation (6.35) with the normalizing quantity defined in equation (6.8) gives

$$\frac{|\tilde{y}_i^{(j)}(t) - \tilde{y}_i^{(j+1)}(t)}{1 + \omega_i^{(y)}}| \leq \frac{\sum_{k=1}^{m_j} |\bar{d}_{k,i}^{(j,y)} - \bar{d}_{k,i}^{(j+1,y)}|}{1 + \omega_i^{(y)}},\ i = 1, \ldots, n_y,\ t \in \bar{\mathcal{S}}_j. \tag{6.36}$$

If the relative upper bound of the pointwise difference of the differential variables is below the threshold tolerance ϵ_r for all components as

$$\frac{\sum_{k=1}^{m_j} |\bar{d}_{k,i}^{(j,y)} - \bar{d}_{k,i}^{(j+1,y)}|}{1 + \omega_i^{(y)}} < \epsilon_r,\ i = 1, \ldots, n_y, \tag{6.37}$$

this indicates that by extending the Lagrange polynomial from the bigger interval into the smaller interval, the relative difference of the differential variables is kept below the threshold tolerance still.

Then, consider the estimated error change of the algebraic variables if two intervals are merged. The algebraic variables from two intervals are expanded using power series where algebraic variables from $\mathcal{S}_j$ are expanded in terms of the end point t_{j+1} and algebraic variables from $\mathcal{S}_{j+1}$ are expanded in terms of the start point t_{j+1} in the similar manner as equations (6.29) and (6.31) as

$$\tilde{z}^{(j)}(t) = \sum_{k=0}^{m_j - 1} \bar{d}_k^{(j,z)} (\frac{t - t_{j+1}}{\bar{\delta}_j})^k,\ t \in [t_j, t_{j+1}], \tag{6.38}$$

$$\bar{d}_k^{(j,z)} = (\frac{\bar{\delta}_j}{\delta_j})^k \sum_{l=1}^{m_j - 1} \hat{a}_{kl}^{(j)} \tilde{z}_l^{(j)}, \tag{6.39}$$

$$\tilde{z}^{(j+1)}(t) = \sum_{k=0}^{m_j - 1} \bar{d}_k^{(j+1,z)} (\frac{t - t_{j+1}}{\bar{\delta}_j})^k,\ t \in [t_{j+1}, t_{j+2}], \tag{6.40}$$

$$\bar{d}_k^{(j+1,z)} = (\frac{\bar{\delta}_j}{\delta_{j+1}})^k \sum_{l=1}^{m_j - 1} \tilde{a}_{kl}^{(j+1)} \tilde{z}_l^{(j+1)}. \tag{6.41}$$

The absolute difference of the algebraic variables and the corresponding upper bound can be obtained as

$$\tilde{z}^{(j)}(t) - \tilde{z}^{(j+1)}(t) = \sum_{k=0}^{m_j-1} (\bar{d}_k^{(j,z)} - \bar{d}_k^{(j+1,z)})(\frac{t - t_{j+1}}{\bar{\delta}_j})^k \tag{6.42}$$

$$\leq \sum_{k=0}^{m_j-1} (\bar{d}_k^{(j,z)} - \bar{d}_k^{(j+1,z)}),\ t \in \bar{\mathcal{S}}_j. \tag{6.43}$$

Then, normalize the quantity using the scale weight defined in equation (6.11). If the relative upper bound of the pointwise difference of the algebraic variables is below the threshold tolerance ϵ_r for all components as

$$\frac{\sum_{k=1}^{m_j} |\bar{d}_{k,i}^{(j,z)} - \bar{d}_{k,i}^{(j+1,z)}|}{1 + \omega_i^{(z)}} < \epsilon_r,\ i = 1, \ldots, n_z, \tag{6.44}$$

this indicates that the relative difference of the algebraic variables by extending the Lagrange polynomial from the bigger interval into the smaller interval is kept below the threshold tolerance also.

So long as the conditions from equations (6.37) and (6.44) are met simultaneously, the two intervals $\mathcal{S}_j$ and $\mathcal{S}_{j+1}$ are possible to be merged into one single mesh interval $\hat{\mathcal{S}}_j = [t_j, t_{j+2}]$ with m_j collocation points without violating the mesh refinement threshold tolerance.

6.5.3 Mesh adaptive collocation method

The mesh adaptive collocation method for solving the BVP-DAEs (2.38)-(2.40) is concluded in the algorithm 6.1, where the details about the collocation method for solving the residual

equation (4.10) can be found in Chapter 4.

Algorithm 6.1: Adaptive mesh collocation method

Input: initial continuation parameter α_0, termination continuation parameter α_m, continuation parameter scaler β, numerical tolerance ϵ, $\epsilon_a = \epsilon$ and ϵ_r, $m_{\min}$, $m_{\max}$.

Initialize the continuation parameter $\alpha = \alpha_0$.

for $i = 1, 2, \ldots, iter_{max}$ **do**

  Solve the residual equation (4.10) with ϵ.

  Evaluate the estimated error in every mesh interval $\mathcal{S}_j$ using the method given in section 6.5.1.

  if $\max_{j=1,\ldots,N-1} \bar{E}^{(j)} < \epsilon_a$ **then**

    if $\alpha < \alpha_m$ **then**

      Break.

    end

    Decrease the continuation parameter as $\alpha = \beta\alpha$.

  end

  else

    Refine the mesh using the methods given in sections 6.5.2 to 6.5.2 with ϵ_a, ϵ_r, $m_{\min}$, $m_{\max}$.

  end

end

Output the solution.

6.6 Implementation and Evaluation

A GPU-accelerated implementation of the algorithm presented in this chapter is contributed using Python and CUDA which can be obtained at `https://github.com/UW-OCP/`. The implementation details is almost the same as shown in chapter 5 which uses Python and CUDA to implement the collocation algorithm for solving the OCPs while as mentioned the implementation here uses 1D grid for all the CUDA kernels.

The code is applied to three typical optimal control problems and the performance of the algorithm is discussed with a normal mesh refinement method, where a global unified number of collocation points is used and when the estimated error in a mesh interval is beyond

the threshold tolerance ϵ_a, the mesh interval is divided into two subintervals. Following terminologies are used in the upcoming example evaluations. First, m_{init} denotes the initial number of collocation points used in each interval; N denotes the total number of mesh nodes in the final mesh; m_{total} denotes the total number of collocation points in the final mesh; while T denotes the computation time for solving the problem. In all the examples, the numerical tolerance used are $\epsilon = \epsilon_a = \epsilon_r = 1 \times 10^{-6}$; the initial continuation parameter is $\alpha_0 = 0.1$; the termination continuation parameter is $\alpha_m = 1 \times 10^{-6}$; the scale factor is $\beta = 0.9$. It should be noted that by adjusting ϵ_r, the behavior of the solver can be adjusted towards either keeping a dense mesh or evolving to a sparse mesh. All the results shown are the five runs average result. All computation results are obtained on a machine equipped with Intel® Core™ i9-9900K CPU @ 3.60GHz × 16 and GeForce RTX 2070 SUPER/PCIe/SSE2 GPU running Ubuntu 19.10 Operating System.

Example 6.6.1. Hyper-sensitive problem

This is a hyper-sensitive optimal control problem from Rao and Mease [2000]. Minimize the cost functional

$$J = \int_0^1 \frac{t_f}{2}(x^2 + u^2)\, dt,$$

subject to

$$\dot{x} = t_f(-x + u),$$
$$x(0) = 1.5,\ x(1) = 1.0,$$

where t_f is fixed. It is known that for sufficiently large values of t_f, the solution of the hyper-sensitive problem presents a so-called "take-off", "cruise", and "landing" structure. While during the "cruise" period, the state and control remain as constants and during the "take-off" and "landing" periods, the state and control have rapid decaying and growing phenomenon. The "cruise" period takes most percentage of the solution and the the "take-off" and "landing" periods take place in a very short amount of time.

The analytic optimal solution of the problem is given as

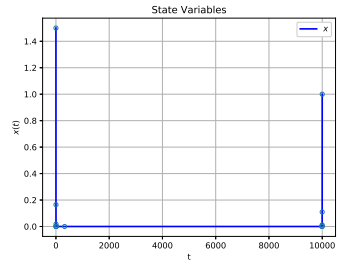
$$x^*(t) = c_1 \exp(t\sqrt{2}) + c_2 \exp(-t\sqrt{2}),$$
$$\lambda^*(t) = -\dot{x}^*(t) - x^*(t),$$
$$u^*(t) = \dot{x}^*(t) + x^*(t),$$

where

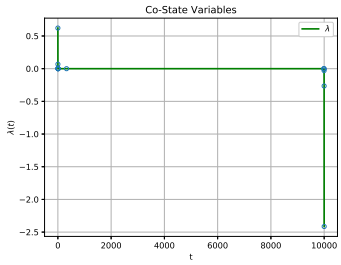
$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \frac{1}{\exp(-t_f\sqrt{2}) - \exp(t_f\sqrt{2})} \begin{bmatrix} 1.5\exp(-t_f\sqrt{2}) - 1 \\ 1 - 1.5\exp(t_f\sqrt{2}) \end{bmatrix}.$$

Figure 6.2(a), 6.2(b), and 6.2(c) show state, costate, and control variables of the analytical optimal solution and the numerical solution using the ph collocation method with $t_f = 10000$, $m_{\min} = 3$, $m_{\max} = 9$, $m_{\text{init}} = 3$. In terms of the state variable, the "take-off" and "landing" segments correspond to the initial rapid decay segment $t \in [0, 50]$ and the terminal rapid growth segment $t \in [0 = 9950, 10000]$ while the "cruise" segment corresponds to the long constant middle segment. The problem is solved using an initial mesh with 101 mesh points. Figure 6.2(d) presents the mesh points distribution history of every mesh refinements and figure 6.2(e) and 6.2(f) show the local mesh points distribution of the final mesh for the "take-off", "landing" segments. From the mesh points distribution history, it can be seen that the ph method successfully figures out the solution is pretty smooth in the "cruise" segment where it keeps reducing the mesh size in that segment by merging adjacent intervals. Also, the rapid decay and growth segments around the initial and final segments are identified successfully and more mesh points are placed.

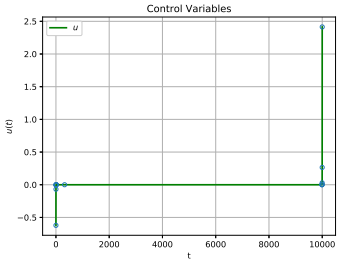
The ph mesh refinement method is compared against the normal h method and tested with different settings. The results with various $m_{\min}$, $m_{\max}$, m_{init}, t_f are presented in table 6.1. It should be noted by setting $m_{\min} = m_{\max} = m_{\text{init}}$, the ph method is the same as the h method with the extra ability to reduce the mesh size than the normal h method. For $t_f = (100, 1000, 10000)$, it can be seen from table 6.1 that under this h method setting, the mesh size and the total number of collocation points using the ph method are much smaller than the h method. More importantly, as we increase $m_{\max}$, the mesh size and the total number of collocation points are even smaller, which is because by using more collocation points, the underlying polynomial degree is increased and can approximate the



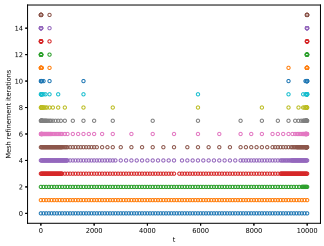
(a) state vs time.



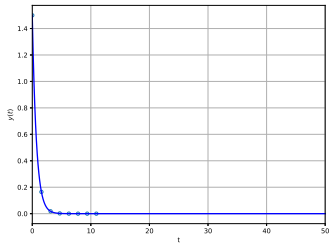
(b) costate vs time.



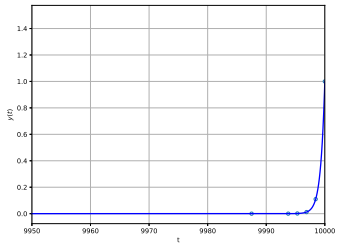
(c) control vs time.



(d) Mesh points distribution history.



(e) Mesh points local distribution near 0.



(f) Mesh points local distribution near τ.

Figure 6.2: Optimal solution of state, costate, and control with $t_f = 10000$ for example 6.6.1

solution with a bigger mesh interval. Also, the mesh size by using the normal h method increases a lot each time because the problem can only be solved with a mesh that is dense enough as the method can not automatically identify the smoothness of the segments and can only increase the mesh size. While using the ph method, the method can identify the segment where the solution is smooth and uses less nodes in the smooth segment and concentrates the mesh nodes in the segment where the solution changes rapidly. However, the computation time does not benefit a lot from using the ph method. This is due to the overhead in using CUDA programming model, as the computation in each interval is done by each thread, the performance is limited by the longest computation time in the interval using the most collocation points.

Table 6.1: Total number of nodes of the final mesh for example 6.6.1 with different t_f, various mesh size, and initial number of collocation points.

t_f	method	$m_{\min}$	$m_{\max}$	m_{init}	N	m_{total}	T
	normal	-	-	3	212	633	15.08
	normal	-	-	4	127	504	14.87
	normal	-	-	5	108	535	14.78
	p-h mesh	3	3	3	136	405	15.18
	p-h mesh	3	5	3	33	145	15.05
	p-h mesh	3	7	3	19	94	15.00
	p-h mesh	3	9	3	16	75	14.91
100	p-h mesh	4	4	4	50	196	14.96
	p-h mesh	3	5	4	26	116	15.04
	p-h mesh	3	7	4	17	82	14.96
	p-h mesh	3	9	4	12	65	15.15
	p-h mesh	5	5	5	30	145	14.72
	p-h mesh	3	5	5	30	136	14.95
	p-h mesh	3	7	5	15	83	15.01

	p-h mesh	3	9	5	12	72	14.98
	normal	-	-	3	273	816	15.49
	normal	-	-	4	164	652	15.16
	normal	-	-	5	133	660	15.01
	p-h mesh	3	3	3	134	399	15.35
	p-h mesh	3	5	3	31	130	15.44
	p-h mesh	3	7	3	21	95	15.36
	p-h mesh	3	9	3	14	69	15.12
1000	p-h mesh	4	4	4	51	200	15.10
	p-h mesh	3	5	4	28	125	15.26
	p-h mesh	3	7	4	16	77	15.45
	p-h mesh	3	9	4	12	68	15.43
	p-h mesh	5	5	5	35	170	14.98
	p-h mesh	3	5	5	36	154	15.12
	p-h mesh	3	7	5	15	82	15.29
	p-h mesh	3	9	5	10	65	15.30
	normal	-	-	3	617	1848	15.81
	normal	-	-	4	385	1536	15.55
	normal	-	-	5	298	1485	15.36
	p-h mesh	3	3	3	143	426	15.72
	p-h mesh	3	5	3	42	165	15.98
	p-h mesh	3	7	3	19	90	15.97
	p-h mesh	3	9	3	15	74	16.08
10000	p-h mesh	4	4	4	58	228	15.60
	p-h mesh	3	5	4	29	126	15.72
	p-h mesh	3	7	4	19	93	16.06
	p-h mesh	3	9	4	14	81	16.05
	p-h mesh	5	5	5	34	165	15.20
	p-h mesh	3	5	5	38	161	15.31

p-h mesh	3	7	5	14	75	15.75
p-h mesh	3	9	5	11	69	15.91

Next, we analyze the quality of the error estimate developed in section 6.5.1. Table 6.2 shows the estimated and exact relative error in the differential and algebraic variables with different mesh size and number of collocation points used. First, it is seen that as the number of mesh nodes or the collocation points is increased, the error in the computed solution decreases accordingly which meets the error analysis of the collocation method. The relationship of the base-10 logarithm of the exact relative error for both variables with different mesh size for the same number of collocation points used are plotted in figure 6.3(a) and 6.3(b), which follows a linear relationship meeting the order consistency of the collocation method given. Next, the consistency in the estimated relative error and the exact relative error demonstrates the accuracy of the error estimation derived in section 6.5.1. Therefore, the error estimate method proposed in this chapter can give a conservative estimate and is able to reflect the locations where the error of the computed solution is large correctly. The ph mesh refinement method can successfully construct new meshes by making the mesh dense in segment where the error is big and reduce the unnecessary mesh in segment where the error is small.

Example 6.6.2. This is an optimal control problem with non-negative control variable inequality constraint from Dontchev [1996]. Minimize the cost functional

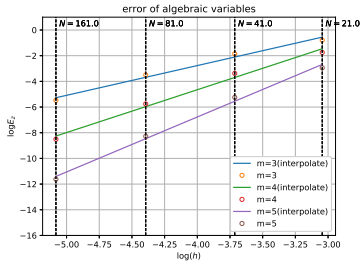
$$J = 0.5 \int_0^2 u(t)^2 \, dt,$$

subject to

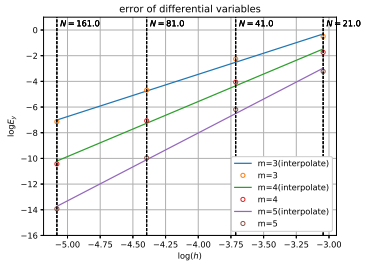
$$\begin{aligned} \dot{x_1}(t) &= x_2, \\ \dot{x_2}(t) &= u, \\ u(t) &\geq 0, \forall t \in [0, 2], \\ x_1(0) &= 0.0, \ x_2(0) = 0.0, \\ x_1(2) &= 5/6, x_2(2) = 1/2. \end{aligned}$$

Table 6.2: Estimated relative error and exact relative error of the solution for example 6.6.1 with $t_f = 100$ with various mesh size and number of collocation points

N	m_{init}	$\bar{E}^y_{\text{estimate}}$	$\bar{E}^y_{\text{exact}}$	$\bar{E}^z_{\text{estimate}}$	$\bar{E}^z_{\text{exact}}$
21	3	9.23e-01	1.80e-01	1.57e-01	1.37e-01
	4	3.09e-01	5.33e-02	4.43e-02	5.14e-02
	5	8.96e-02	1.21e-02	1.69e-02	1.56e-02
41	3	1.77e-01	3.88e-02	6.01e-02	5.55e-02
	4	3.22e-02	6.26e-03	1.06e-02	1.12e-02
	5	5.09e-03	7.80e-04	1.56e-03	1.92e-03
81	3	2.20e-02	5.70e-03	1.49e-02	1.46e-02
	4	2.03e-03	4.26e-04	1.49e-03	1.51e-03
	5	1.68e-04	2.77e-05	1.13e-04	1.31e-04
161	3	1.97e-03	6.04e-04	2.67e-03	2.71e-03
	4	9.32e-05	1.98e-05	1.44e-04	1.45e-04
	5	3.96e-06	6.71e-07	5.61e-06	6.15e-06



(a) differential variables vs nodes.



(b) algebraic variables vs nodes.

Figure 6.3: Error in log scale of example 6.6.1

The optimal control is a nonsmooth function in time, that is,

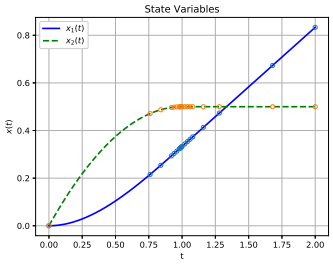
$$u^*(t) = \begin{cases} 1 - t, & \text{for } t \in [0, 1), \\ 0, & \text{for } t \in [1, 2]; \end{cases}$$

where the optimal control is discontinuous at $t = 1$.

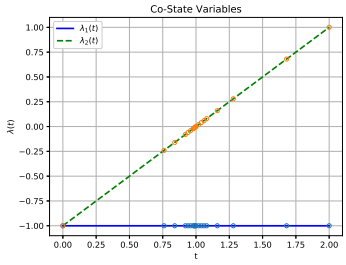
We apply the ph collocation method with $m_{\min} = 3$, $m_{\max} = 9$, $m_{\text{init}} = 3$ to the problem and solved the problem with the optimal solution of the states, costates, control, multiplier shown in figure 6.4(a)-6.4(d). From the plots of the solution of the problem, it can be seen that the optimal control obtained using the ph method captured the discontinuity accurately around $t = 1$. Thus, to have an accurate solution with the existence of the discontinuity, it is necessary that more mesh points are put near the discontinuity of the control and less mesh points are put around the area that the control is constant and the state is smooth.

The problem is solved using an initial mesh with 101 mesh points. The final solution has a mesh of $N = 19$ time nodes and the node distribution of the final mesh is presented in figure 6.4(e). Interesting, it can be seen that the ph method progresses to a final mesh distribution that the mesh points concentrate densely near the control discontinuity point and are very sparse in other parts. The figures of the states and costates show that the nonsmoothness of the solution are accurately captured by the ph method.

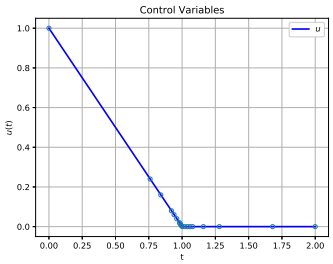
The problem can also be successfully solve by the collocation method with the normal h mesh refinement method. Table 6.3 summarizes the computation time and mesh sizes for various methods with different settings. It is seen that the mesh size using the ph method is much smaller than using the normal mesh refinement method not only in the total number of collocation points used but also the total number of mesh intervals. This is due to the flexible feature of the ph method such that for the solution segment that is smooth enough, it chooses to use a small number of collocation points in the segment and when the solution is not that smooth, it chooses to increase the underlying polynomial degree or divide the interval into small subintervals with less collocation points. For the segment that is smooth, the mesh intervals are merged together to further reduce the mesh size which saves lots of computational memory used. It should be noted that the computation time using the ph method is more than that of the normal h method. This is due to the essence of the CUDA



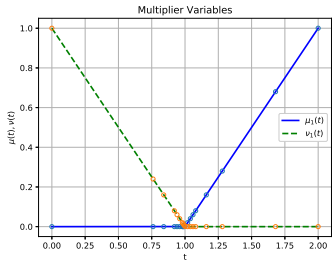
(a) state vs time.



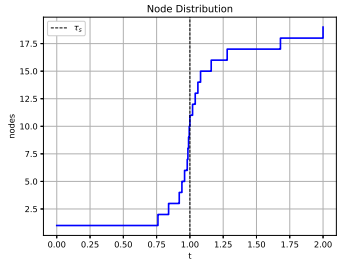
(b) costate vs time.



(c) control vs time.



(d) multiplier vs time.



(e) node distribution

Figure 6.4: Optimal solution of states, costates, control and multipliers for example 6.6.2

Table 6.3: Total number of nodes of the final mesh and computation time using various collocation methods for example 6.6.2.

Method	$m_{\min}$	$m_{\max}$	m_{init}	N	m_{total}	T
normal	-	-	3	129	384	31.64
normal	-	-	4	108	428	31.28
normal	-	-	5	105	520	32.74
p-h mesh	3	3	3	30	87	30.33
p-h mesh	3	5	3	19	67	32.24
p-h mesh	3	7	3	18	64	32.72
p-h mesh	3	9	3	19	68	33.65
p-h mesh	4	4	4	20	76	30.20
p-h mesh	3	5	4	21	73	30.76
p-h mesh	3	7	4	16	60	32.75
p-h mesh	3	9	4	15	59	32.80
p-h mesh	5	5	5	21	100	31.79
p-h mesh	3	5	5	29	106	31.80
p-h mesh	3	7	5	16	62	33.92
p-h mesh	3	9	5	15	59	34.49

programming model where the performance is bottlenecked by the execution of the thread taking the longest time which is also mentioned in example 6.6.1.

Example 6.6.3. Consider a time optimal low-thrust trajectory problem, where a satellite is transferred from a circular low-Earth orbit to a geosynchronous orbit from Büskens and Maurer [2000] Find a thrust direction control $u(t)$, $0 \leq t \leq 1$, that minimizes

$$J = p_1,$$

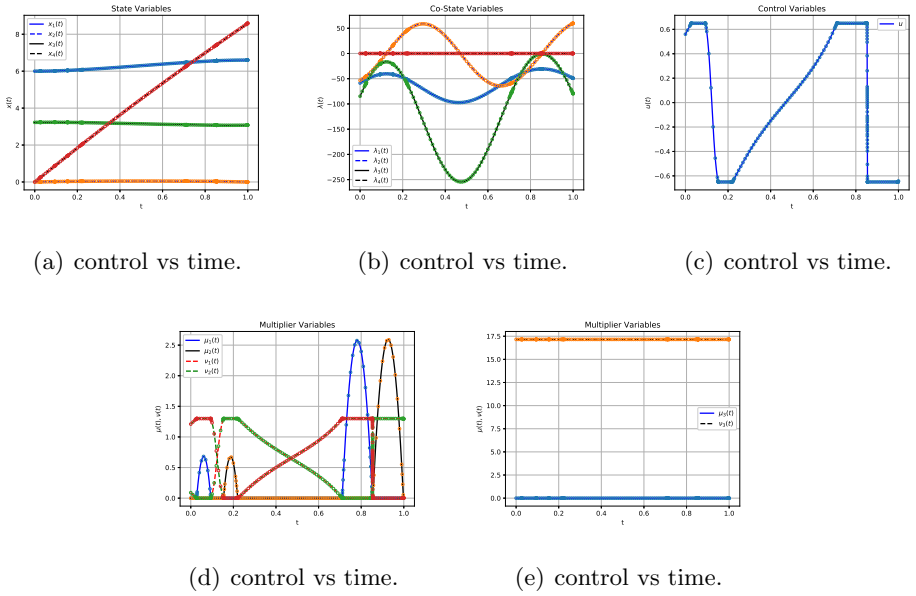
subject to

$$
\begin{aligned}
& p_1 = 20 + t_f, \\
& a(t,p) = 0.01 + p, \\
& \dot{x_1} = p_1 x_2, \\
& \dot{x_2} = p_1(\frac{x_3^2}{x_1} - \frac{r_\mu}{x_1^2} + a(t,p)\sin(u)), \\
& \dot{x_3} = p_1(-\frac{x_2 x_3}{x_1} + a(t,p)\cos u), \\
& \dot{x_4} = p_1(\frac{x_3}{x_1}), \\
& |u(t)| \leq 0.65, \forall t \in [0,1], \\
& -p_1 \leq 0, \\
& x(0) = [6.0, 0.0, \sqrt{\frac{r_\mu}{x_1(0)}}, 0.0], \\
& x(1) = [6.6, 0.0, \sqrt{\frac{r_\mu}{x_1(1)}}].
\end{aligned}
$$

where $x_1(t)$ represents the radial position, $x_2(t)$ the radial velocity, $x_3(t)$ the circumferential velocity and $x_4(t)$ the polar angle, with $r_\mu = 62.5$ representing the gravitational parameter for the earth. In the original model, $a(t,p)$ is a function of time describing the thrust acceleration of the spacecraft and depending on the reduced mass. For simplicity we consider the function $a(t,p) = 0.01+p$ with perturbation parameter p to illustrate sensitivity analysis for this problem. The nominal parameter is $p_0 = 0$. The free final time t_f is treated as an additional optimization parameter and this is a minimum time optimal control problem.

This is a highly nonlinear and complicated optimal control problem with 4 state variables, 1 control variables, and 3 control variable inequality constraints. Figures 6.5(a)-6.5(e) show the optimal solution obtained by using the ph collocation method with $m_{\min} = 3$, $m_{\max} = 9$, $m_{\text{init}} = 3$ with an initial mesh of 101 mesh points.

It can be seen that the optimal controls of the problem have several discontinuities and the states and costates are pretty nonsmooth. The normal mesh method failed to solve the problem with a limited total of 4000 mesh points to the maximum with various number of collocation points ($m_{\text{init}} = 3, 4, 5$).



(a) control vs time. (b) control vs time. (c) control vs time.

(d) control vs time. (e) control vs time.

Figure 6.5: Optimal solution of states and costates, control, and multupliers for example 6.6.3

Table 6.4 summarizes the computation time and mesh sizes for the ph mesh method with different settings. It is seen as the initial number of collocation points used is increased $m_{\text{init}} = (3, 4, 5)$, the total number of mesh points and collocation points all decrease. As using more collocation points is the same as using a polynomial with higher degree to approximate the solution, and the same error accuracy can be obtained with a bigger mesh interval. As the normal h mesh method is not able to solve the problem with a maximum allowable number of mesh points, this means the ph mesh method is more flexible and can put the mesh points at necessary locations to save the computational workload.

6.7 Discussion

All the examples shown in section 6.6 present various features of the adaptive ph mesh refinement method introduced in this chapter. Example 6.6.1 presents that the ph mesh refinement method is able to dramatically decrease the mesh size by eliminating mesh

Table 6.4: Total number of nodes of the final mesh and computation time for example 6.6.3

Method	$m_{\min}$	$m_{\max}$	m_{init}	N	m_{total}	T
p-h mesh	3	9	3	214	999	390.01
p-h mesh	3	9	4	200	997	395.83
p-h mesh	3	9	5	169	936	444.73

points and collocation points and merging adjacent mesh intervals in the segment where the solution does not change significantly and put mesh points in the segment where the solution changes rapidly. The result from example 6.6.2 exhibits how the ph mesh refinement method can effectively detect the discontinuity in the variable and therefore concentrates the mesh points in the vicinity of the discontinuity while reduces the mesh size in the segment where the solution is smooth at the same time. Moreover, similar to the result obtained in example 6.6.1, the result shows the mesh size is dramatically smaller when solving the problem using the ph mesh refinement method compared with the normal h mesh refinement method. Example 6.6.3 shows the ability of the ph method to economically solve the complicated problem where the solution is highly nonsmooth and contains several discontinuities.

The adaptive ph mesh refinement method introduced in this chapter has the potential advantage that the final mesh is smaller in size when compared with that of a normal h method, requiring less memory than might be required to achieve the same accuracy using an h method. Also, the method is more robust when the number of maximum allowable mesh points is limited. Moreover, the ph mesh refinement method introduced in this chapter is based on the novel error estimate of both the differential and algebraic variables.

6.8 Conclusion

An adaptive mesh refinement method for solving optimal control problems based on the indirect collocation method has been introduced in this chapter. The method has the ability to both increase and decrease the mesh size. The mesh refinement is guided by a method to estimate the error of both the differential and algebraic variables. Using this error estimate, a mesh refinement method is developed that is able to iteratively reduce the error estimate

when necessary either by increasing the degree of the polynomial approximation in a mesh interval or by increasing the number of mesh intervals. Furthermore, the size of the mesh can be decreased either by dropping the negligible terms in the power series representation of the variables or by combining mesh intervals that share the similar polynomial approximation. The method is described in detail and applied successfully to three examples from the open literatures. The results obtained in this research show that the method outperforms fixed-order methods a lot and can reduce the computational burden to a big extent.

Chapter 7

AUTO DETECTION OF ACTIVE SVICS

7.1 Backgroud

When solving the Newton's iteration (4.12) of collocation method, the original BABD system (4.13) is reduced into a smaller BABD system (4.18) via equation (4.17). Whereas, equation (4.17) is based on the assumption that matrix W_j is nonsingular for sufficiently small δ_j. However, during the empirical testing to the OCPs, we found that for some OCPs equipped with SVICs, the assumption is not always true, which lead us to further investigate what happened behind the scene.

Originally, the index of the BVP-DAEs considered (2.38)-(2.40) is directly affected by $\frac{\partial g}{\partial z}$ which is

$$\left[\frac{\partial g}{\partial z}\right] = \begin{bmatrix} \bar{H}_{uu} & D_u d & 0 & 0 & 0 \\ D_u d^T & -\alpha I & 0 & I & 0 \\ 0 & 0 & -\alpha I & 0 & I \\ 0 & D_\mu \psi_d & 0 & D_\nu \psi_d & 0 \\ 0 & 0 & D_\xi \psi_s & 0 & D_\sigma \psi_s \end{bmatrix}.$$

The DAEs is index-1 when $\frac{\partial g}{\partial z}$ is nonsingular which is $\det(\frac{\partial g}{\partial z}) \neq 0$.

Consider problems equipped with only SVICs where $\frac{\partial g}{\partial z}$ can be further simplified as

$$\left[\frac{\partial g}{\partial z}\right] = \begin{bmatrix} \bar{H}_{uu} & 0 & 0 \\ 0 & -\alpha I & I \\ 0 & D_\xi \psi_s & D_\sigma \psi_s \end{bmatrix}.$$

As the elements in $\frac{\partial g}{\partial z}$ are all diagonal, which does not affect the behavior when com-

puting the determinant. Consider a problem with only one SVIC, $\frac{\partial g}{\partial z}$ becomes

$$\left[\frac{\partial g}{\partial z}\right] = \begin{bmatrix} \bar{H}_{uu} & 0 & 0 \\ 0 & -\alpha & 1 \\ 0 & D_\xi \psi_s & D_\sigma \psi_s \end{bmatrix}.$$

where

$$D_\xi \psi_s = 1 - \frac{\xi}{\sqrt{\sigma^2 + \xi^2 + 2\alpha}},$$
$$D_\sigma \psi_s = 1 - \frac{\sigma}{\sqrt{\sigma^2 + \xi^2 + 2\alpha}},$$

can be computed using the Kanzow's smoothed Fisher-Burmeister formula (2.34).

The determinant $\det(\frac{\partial g}{\partial z})$ can be obtained simply as

$$\begin{aligned} \det(\frac{\partial g}{\partial z}) &= \bar{H}_{uu}(-\alpha D_\sigma \psi_s - D_\xi \psi_s) \\ &= -\bar{H}_{uu}(\alpha D_\sigma \psi_s + D_\xi \psi_s). \end{aligned}$$

By the second order necessary condition assumption 4, it is ensured that $\bar{H}_{uu} > 0$. By the properties of the Kanzow's smoothed Fisher-Burmeister formula (2.34), it is ensured that $0 < D_\sigma \psi_s < 2$, $0 < D_\xi \psi_s > 2$. With $\alpha > 0$, it is ensured that

$$\begin{aligned} \det(\frac{\partial g}{\partial z}) &= -\bar{H}_{uu}(\alpha D_\sigma \psi_s + D_\xi \psi_s) < 0 \\ &= -\bar{H}_{uu}[\alpha(1 - \frac{\sigma}{\sqrt{\sigma^2 + \xi^2 + 2\alpha}}) + (1 - \frac{\xi}{\sqrt{\sigma^2 + \xi^2 + 2\alpha}})], \end{aligned}$$

which proves that $\frac{\partial g}{\partial z}$ shall not be singular for $\alpha > 0$.

Then, we investigate under what condition $\frac{\partial g}{\partial z}$ can be singular which is when

$$[\alpha(1-\frac{\sigma}{\sqrt{\sigma^2+\xi^2+2\alpha}})+(1-\frac{\xi}{\sqrt{\sigma^2+\xi^2+2\alpha}})]=0$$

$$\frac{\xi+\alpha\sigma}{\sqrt{\sigma^2+\xi^2+2\alpha}}=1+\alpha$$

$$(1+\alpha)(\sqrt{\sigma^2+\xi^2+2\alpha})=\xi+\alpha\sigma$$

$$(1+2\alpha+\alpha^2)(\sigma^2+\xi^2+2\alpha)=(\xi^2+2\alpha\sigma\xi+\alpha^2\sigma^2)$$

$$\sigma^2+\xi^2+2\alpha+2\alpha\sigma^2+2\alpha\xi^2+4\alpha^2+\alpha^2\sigma^2+\alpha^2\xi^2+2\alpha^3=\xi^2+2\alpha\sigma\xi+\alpha^2\sigma^2$$

$$\sigma^2+2\alpha+2\alpha\sigma^2+2\alpha\xi^2+4\alpha^2+\alpha^2\xi^2+2\alpha^3=2\alpha\sigma\xi$$

$$\sigma^2-2\alpha\sigma\xi+\alpha^2\xi^2+2\alpha+2\alpha\sigma^2+2\alpha\xi^2+4\alpha^2+2\alpha^3=0$$

$$(\sigma-\alpha\xi)^2+2\alpha+2\alpha\sigma^2+2\alpha\xi^2+4\alpha^2+2\alpha^3=0$$

$$(\sigma-\alpha\xi)^2+2\alpha(1+\sigma^2+\xi^2+2\alpha+\alpha^2)=0$$

This can be discussed under two cases as

1. When the SVIC is active, $\sigma-\alpha\xi=0$, $2\alpha(1+\sigma^2+\xi^2+2\alpha+\alpha^2)>0\approx 0$.

2. When the SVIC is inactive, $(\sigma-\alpha\xi)^2+2\alpha(1+\sigma^2+\xi^2+2\alpha+\alpha^2)>0$.

So, the discussion meets the empirical results as when the SVIC is active, $\det(\frac{\partial g}{\partial z})\approx 0$ resulting in a large condition number of the matrix W_i.

7.2 Methodology

From the above discussion, we draw the conclusion that the large $cond(W)$ can indirectly indicate the active SVICs. Solving equations (2.32) and (2.33) with the active SVIC can draw the result that $\sigma=\alpha$, $\xi=1$. Thus, we no longer need the Fischer-Burmeister formula of the complementarity condition associated with the SVICs and can eliminate one of the σ or ξ. To avoid the diminishing determinant problem of $\frac{\partial g}{\partial z}$, we choose to keep the σ variable and eliminate the ξ variable which reduces the complexity of the system and $\frac{\partial g}{\partial z}$

are simplified as

$$\left[\frac{\partial g}{\partial z}\right] = \begin{bmatrix} \bar{H}_{uu} & 0 \\ 0 & 1 \end{bmatrix}$$

for problems with SVICs only.

Equations (4.15) and (4.16) become

$$J'_j \Delta\tilde{y}_j + W'_j \Delta\tilde{k}'_j + V'_j \Delta\tilde{p} = -\tilde{f}_j^{a'}, \tag{7.1}$$

$$-\Delta\tilde{y}_j - D'_j \Delta\tilde{k}'_j + \tilde{y}_{j+1} = -\tilde{f}_j^{b}. \tag{7.2}$$

where $\tilde{f}_j^{a'} \in \mathbb{R}^{m(n_y+n_z-n_s)}$, $\tilde{f}_j^{b} \in \mathbb{R}^{n_y}$, $J'_j \in \mathbb{R}^{m(n_y+n_z-n_s)\times n_y}$, $D'_j \in \mathbb{R}^{n_y \times m(n_y+n_z-n_s)}$, $V'_j \in \mathbb{R}^{m(n_y+n_z-n_s)\times n_p}$, $W'_j \in \mathbb{R}^{m(n_y+n_z-n_s)\times m(n_y+n_z-n_s)}$, $\tilde{k}'_j \in \mathbb{R}^{m(n_y+n_z-n_s)}$ and n_s is the number of SVICs of the problem. $\tilde{f}_j^{a'}$, J'_j, D', V', W'_j, $\tilde{k}'_j$ are obtained by eliminating the rows corresponding to the Fischer-Burmeister formula associated with the SVICs and the columns corresponding to the variables ξ.

Equation (4.17) are substituted as

$$\Delta\tilde{k}'_j = W_j^{'-1}(-\tilde{f}_j^{a'} - J'_j \Delta\tilde{y}_j - V'_j \Delta\tilde{p}). \tag{7.3}$$

where the eliminated variables ξ are set automatically to α. The other parts are kept the same as shown in Chapter 4.

7.3 Numerical Evaluation

A hybrid CPU-GPU implementation of the algorithm presented in this chapter is contributed using Python and CUDA. The implementation details is almost the same as shown in chapter 6 and the calculation of the condition number of W_i is done by using Numpy Harris et al. [2020].

The code is applied to two typical optimal control problems and the performance of the algorithm is discussed. Following terminologies are used in the upcoming example evaluations. N_{iter} denotes the number of continuation iterations used to solve the problem, while T denotes the computation time for solving the problem. In all the examples, the numerical tolerance used is $\epsilon = 1 \times 10^{-5}$; the initial continuation parameter is $\alpha_0 = 0.1$; the termination continuation parameter is $\alpha_m = 1 \times 10^{-6}$; the threshold for deciding the activeness of

the SVICs by checking the condition number of W_i is $\sqrt{\frac{1}{\epsilon_{machine}}}$. All computation results are obtained on a machine equipped with Intel® Core™ i9-9900K CPU @ 3.60GHz × 16 and GeForce RTX 2070 SUPER/PCIe/SSE2 GPU running Ubuntu 19.10 Operating System.

Example 7.3.1. This is a problem from Jacobson and Lele [1969] with three state variables and one control variable which is to minimize

$$J = \int_0^1 x_1^2(t) + x_2^2(t) + 0.005u^2(t)\, dt,$$

subject to

$$\begin{aligned} \dot{x_1}(t) &= x_2(t), \\ \dot{x_2}(t) &= -x_2(t) + u(t), \\ \dot{x_3}(t) &= 1.0, \end{aligned}$$

and the boundary conditions

$$x_1(0) = 0.0, x_2 + 1.0 = 0, x_3 = 0.0,$$

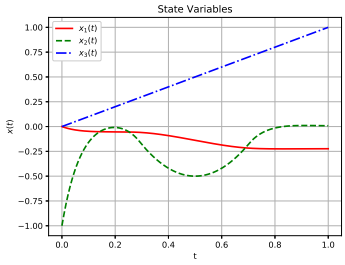
with the CVICs

$$|u(t)| \leq 20.0, \forall t \in [0, 1],$$

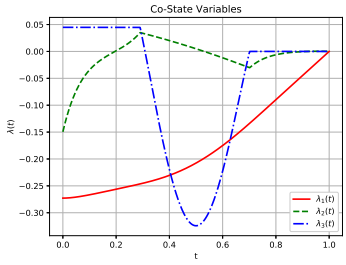
and the SVIC

$$-(8.0(x_3(t) - 0.5)^2 - 0.5 - x_2(t)) \leq 0, \forall t \in [0, 1].$$

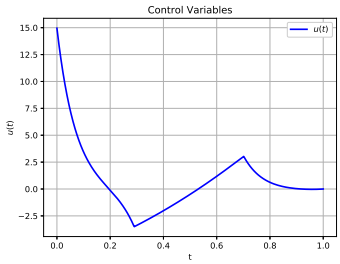
The initial guesses for solving this problem are using straight lines between the initial and final conditions for those variables with them and constants for those without. Both methods with or without automatically adjusting the problem solving formulation by detecting the activeness of the SVICs through the condition number of W_i are able to solve the problem. The plots of the numerical solution from the proposed method are shown in figure 7.1(a)-7.1(e) and the computation results for both methods are shown in table 7.1.



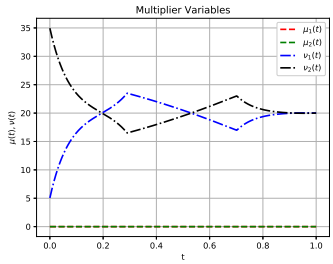
(a) state vs time.



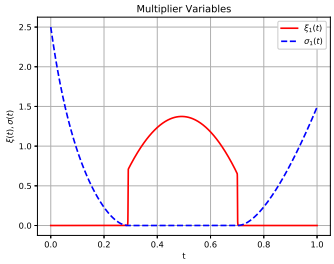
(b) costate vs time.



(c) control vs time.



(d) multiplier vs time.



(e) multiplier vs time.

Figure 7.1: Optimal solution of states, costates, control and multipliers for example 7.3.1

Table 7.1: Computation results for example 7.3.1 using both methods

method	cost function	N_{iter}	T
auto	0.16981926	160	74.89
normal	0.16981929	177	57.90

It can be seen from table 7.1 that the proposed method is able to solve the problem with much less continuation iterations (around 10%). As the analytical solution of the problem is unknown, the cost function values computed for both methods are also listed for a comparison. It is worth noting that the cost function values from both methods are consistent up to seven decimal places, while the proposed method can give a slightly better cost function value. Also, the computation time of the proposed method is longer than the normal method. This is mainly due to the fact that the proposed method needs to compute the condition number at every iteration and when it detects the SVIC is active, it needs to automatically set the values for the lagrange multipliers and transform the original problem into another reduced form problem which introduces some overhead. Although the improvement in the cost function value is small, this example demonstrates the ability of the proposed method to converge faster to the solution of the problem efficiently and fast.

Example 7.3.2. This is problem with three states and one control from Houska et al. [2011]. The final time of the problem is represented with the parameter variable p. The problem is formulated as follows:

$$J = \int_0^1 p(1 + 0.1u(t)^2)\ dt,$$

Table 7.2: Computation results for example 7.3.2 using both methods

method	cost function	N_{iter}	T
auto	7.47173740	197	138.66
normal	7.47173741	224	117.68

subject to the dynamic equations

$$\begin{aligned}
\dot{y}_1(t) &= p y_2(t), \\
\dot{y}_2(t) &= p\frac{u(t) - 0.2y_2(t)^2}{y_3(t)}, \\
\dot{y}_3(t) &= -0.01pu(t)^2,
\end{aligned}$$

and the boundary conditions

$$\begin{aligned}
x(0) &= [0, 0, 1.0]^T, \\
x(1) &= [10.0, 0]^T,
\end{aligned}$$

and two control variable inequality constraints

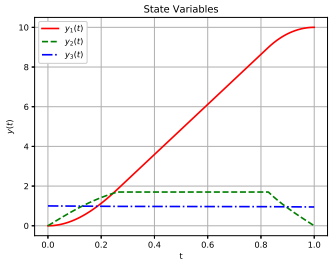
$$-1.1 \leq u(t) \leq 1.1, \forall t \in [0, 1],$$

and two state variable inequality constraints

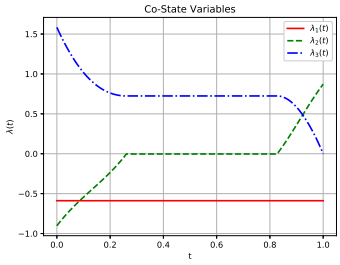
$$-0.1 \leq y_2(t) \leq 1.7, \forall t \in [0, 1].$$

The initial guesses are obtained by solving a relaxed problem with a relaxed cost functional. Both methods are able to solve the problem. The plots of the numerical solution from the proposed method are shown in figure 7.2(a)-7.2(e) and the computation results for both methods are shown in table 7.2.

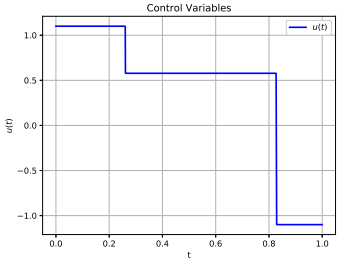
The computation results for this example exhibit the same effect with example 7.3.1. The proposed method is able to solve the problem with much less continuation iterations



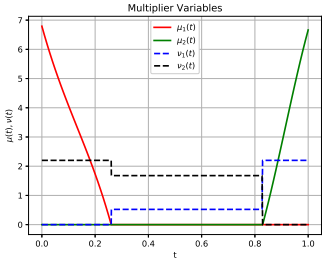
(a) state vs time.



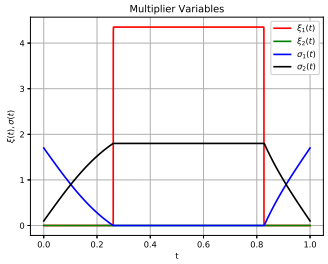
(b) costate vs time.



(c) control vs time.



(d) multiplier vs time.



(e) multiplier vs time.

Figure 7.2: Optimal solution of states, costates, control and multipliers for example 7.3.2

with longer computation time due to the overhead it introduced. The cost function value computed for both methods are consistent up to seven decimal places and the proposed method gives a slightly better cost function value. This again proves the robustness and effectiveness of the proposed method compared with the normal method.

7.4 Conclusion

A method which can automatically detect the activeness of the SVICs of the OCPs based on the indirect collocation method is introduced in this chapter. The chapter first investigates the problems from the empirical results using the original indirect collocation method and gives the analysis on the reason of the problem. Then, the chapter proposes a method to solve the problem due to the active SVICs which makes the original system unstable to solve. By reducing the original system into a more compact system when know the SVIC is active, the unstable numerical solving problem can be resolved where the values of the lagrange multipliers are automatically set. The method is described in detail and applied successfully to two examples from the open literatures. The results obtained in this research show that the method outperforms the original indirect collocation method by solving the problem with much less continuation iterations and can give a solution at least as accurate as the original method.

Chapter 8
SUMMARY AND FUTURE WORK

8.1 Summary

The primary focus of this work is on developing robust and efficient parallel indirect methods for solving optimal control problems equipped with inequality constraints and parameters. The goal also includes implementing a general-purpose software based on the indirect methods introduced which can overcome several difficulties

- Deriving the analytical optimality conditions in the software program.
- Incorporating various inequality constraints without a-prior knowledge of the sequence of constrained and unconstrained segments.
- Time consuming numerical computations.

A slacked unconstrained penalty function method is used to incorporate the control variable inequality constraints and state variable inequality constraints of the optimal control problems. It is proved that the optimal cost functional of the transformed problem by using the penalty function method converges to the optimal cost functional of the original problem by decreasing the penalty parameter term gradually to zero. Then, the first-order necessary optimality conditions are obtained by using the calculus of variation and can be represented in a compact form as a boundary value problem involving differential-algebraic equations. The complementarity conditions associated with the inequality constraints in the necessary conditions are approximated in a more compact way using the Kanzow's smoothed Fisher-Burmeister formula. It is proved that the involved differential-algebraic equations are guaranteed to be index-1, which enables the use of various numerical methods to solve the problem.

A multiple shooting method is first introduced to solve the derived BVP-DAEs. Except an integrator based on a single step linearly implicit Runge-Kutta (Rosenbrock-Wanner, ROW) method, the use of the MATLAB (MATLAB [2017]) built-in integrator `ode15s` is explored to integrate the associate differential equations. Adapted differential equation formats are designed to adapt to the API of the `ode15s` integrator. A damped Newton's method is used to solve the residual equation resulted from the multiple shooting formulation where the Jacobian is in a boarded almost block diagonal structure and a huge linear system is to be solved. A very robust parallel QR decomposition method is used to solve the associated linear system and the classic line-search is employed to obtain the descent direction from using the Newton's method. To improve the computational efficiency, the MATLAB parallel computing toolbox is employed to implement the parallel version of the multiple shooting algorithm. The performance of the implemented multiple shooting solvers with the two different integrators are compared by testing over 140 optimal control problems from open literatures and it is shown that the MATLAB (MATLAB [2017]) built-in integrator `ode15s` is not suitable for the multiple shooting algorithm developed here. From the test performances, it is also concluded that the algorithm can not benefit from the MATLAB parallel computing toolbox due to the heavy overhead from it.

A collocation method is also presented to solve the BVP-DAEs. where a global unified number of collocation points is used and the derivatives of the differential variables and the algebraic variables are approximated using the Lagrange polynomials at the collocation points associated with the specific m-stage Lobatto IIIA implicit Runge-Kutta method. The same damped Newton's method is employed to solve the residual system from using the collocation method. The associated Jacobian is able to be reduced into the same BABD form as the multiples shooting method by some reduction techniques and the same parallel QR decomposition method is used to solve the reduced BABD system. A parallel MATLAB implementation of the collocation algorithm is conducted and compared with the MATLAB implementation of the multiple shooting algorithm. Although the MALTAB implementation of the collocation algorithm is more efficient and robust than that of the multiple shooting algorithm, the collocation method still can not benefit from the MATLAB parallel computing technique.

After figuring out the inefficiency of the MATLAB parallel computing toolbox, the multiple shooting algorithm and the collocation algorithm are accelerated with the Graphics Processing Units (GPUs) by using the CUDA programming model. Adequate background knowledge about the CUDA programming model is presented to help understanding the implementation. The adapted multiple shooting algorithm structure is first described to help explain the design and implementation of the GPU code. The GPU parallel implementation boosts the performance a lot and several concrete numerical examples are shown to prove the efficiency and robustness of the implementation. Next, the GPU based parallel collocation algorithm is presented. Ideas about how to design the specialized CUDA kernels to fully take advantage of the collocation algorithm structures are described and a correspondence graph between the collocation time nodes and 2D CUDA threads is constructed. Custom memory pool and data structure are developed to reduce the overhead of frequent allocation of the GPU memories. Several non-trivial examples are shown to explore the performance of the GPU parallel implementation of the collocation algorithm. Also, the performance profile between the GPU implementations of the multiple shooting algorithm and collocation algorithm is construct to better quantify the robustness and efficiency of the two. Several other performance profiles between using different number of collocation points for the collocation algorithm are constructed. It is shown that the GPU implementation of the collocation algorithm is much more efficient and robust than the multiple shooting algorithm. And collocation algorithm with more collocation points can solve more problems in a longer computation time.

Based on the results from performance profiles between the collocation algorithms using different number of collocation points, a research of developing a collocation algorithm with the ability to dynamically refining the mesh is conducted to best improve the efficiency and robustness. Originally, the collocation method uses a global unified number of collocation points in all the mesh intervals and the width of the mesh interval is allowed to vary, whereas in the developed ph adaptive mesh refinement method, both the width of the mesh interval and the corresponding number of collocation points used with the interval are allowed to vary simultaneously. A novel method to estimate the relative error between the numerical solution and the optimal solution is developed first to be used as the basis for the adaptive

mesh refinement process. Based on this error estimate, a ph mesh refinement method was developed which is able to iteratively reduce the error of the interval when needed by either increasing the number of collocation points used in a mesh interval or decreasing the interval width. Moreover, the method can reduced the mesh size by either decreasing the number of collocation points used or by merging adjacent mesh intervals. The method is applied successfully to three examples from the open literature which demonstrates different features of the method and the results show that the developed method outperforms fixed-order methods a lot and can reduce the computational burden to a big extent.

8.2 Future Work

This dissertation contributes to the development of the parallel indirect methods for solving optimal control problems. The general-purpose software is developed based on the methods described in this dissertation and the power of the faster hardware as GPU is exploited.

A drawback for using the indirect method to solve the optimal control problem is that the initial guess of the solution is very important where the convergence space is pretty small Rao [2009]. Moreover, Newton's method can only offer a fast convergence only when the solution is pretty close to the optimum. This leads to the need to develop some systematic method to generate good initial guess before applying those methods which can lead to a guaranteed and faster convergence to the optimal solution.

Future extension of the presented work could also focus on further exploiting the power of the GPUs. In the current implementation, only one GPU is used when running the algorithm and for highly nonlinear and complicated problems, the algorithm may be able to benefit from the use of multiple GPUs. There are some existing framework which is able to incorporate the use of multiple GPUs like OpenMP which could be easily applied to the current work. Also, current CUDA implementation uses a high level programming language Python and CUDA also supports lower programming languages as C and C++. Implementations using C/C++ and CUDA can offer more flexibility and further increase the efficiency.

The scheme of the current adaptive mesh refinement method is that when reducing the error within a mesh interval, the method first try to increase the number of collocation

points used and then divide the interval after reaching the maximum number of collocation points allowed. However, in the segment where the solution is not smooth, the best way to refine the mesh is to directly divide the bigger interval into multiple smaller intervals. So, a method to detect the smoothness of the solution in every interval can be a good directly to work on.

BIBLIOGRAPHY

John T Betts. Survey of numerical methods for trajectory optimization. *Journal of Guidance, Control, and Dynamics*, 21(2):193–207, 1998.

Anil V Rao. A survey of numerical methods for optimal control. *Advances in the Astronautical Sciences*, 135(1):497–528, 2009.

Brian C Fabien. Some tools for the direct solution of optimal control problems. *Advances in Engineering Software*, 29(1):45–61, 1998.

Brian C. Fabien. Direct optimization of dynamic systems described by differential-algebraic equations. *Optimal Control Applications and Methods*, 29(6):445–466, 2008. doi: 10.1002/oca.838. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/oca.838`.

G. Elnagar, M. A. Kazemi, and M. Razzaghi. The pseudospectral legendre method for discretizing optimal control problems. *IEEE Transactions on Automatic Control*, 40(10):1793–1796, Oct 1995. ISSN 2334-3303. doi: 10.1109/9.467672.

GAMAL N. ELNAGAR and MOHSEN RAZZAGHI. Short communication: A collocation-type method for linear quadratic optimal control problems. *Optimal Control Applications and Methods*, 18(3):227–235, 1997. doi: 10.1002/(SICI)1099-1514(199705/06)18:3⟨227::AID-OCA598⟩3.0.CO;2-A.

David Benson. *A Gauss pseudospectral transcription for optimal control*. PhD thesis, Massachusetts Institute of Technology, 2005.

Anil V. Rao, David A. Benson, Christopher Darby, Michael A. Patterson, Camila Francolin, Ilyssa Sanders, and Geoffrey T. Huntington. Algorithm 902: Gpops, a matlab software for solving multiple-phase optimal control problems using the gauss pseudospectral method. *ACM Trans. Math. Softw.*, 37(2), April 2010. ISSN 0098-3500. doi: 10.1145/1731022.1731032. URL `https://doi.org/10.1145/1731022.1731032`.

Richard F Hartl, Suresh P Sethi, and Raymond G Vickson. A survey of the maximum principles for optimal control problems with state constraints. *SIAM review*, 37(2):181–218, 1995.

Sunil Kumar Agrawal and Brian C Fabien. *Optimization of dynamic systems*, volume 70. Springer Science & Business Media, 2013.

Brian C Fabien. Indirect numerical solution of constrained optimal control problems with parameters. In *American Control Conference, Proceedings of the 1995*, volume 3, pages 2075–2076. IEEE, 1995.

U. Ascher, J. Christiansen, and R. D. Russell. Collocation software for boundary-value odes. *ACM Trans. Math. Softw.*, 7(2):209–222, June 1981. ISSN 0098-3500. doi: 10.1145/355945.355950. URL `https://doi-org.offcampus.lib.washington.edu/10.1145/355945.355950`.

Uri M. Ascher and Raymond J. Spiteri. Collocation software for boundary value differential-algebraic equations. *SIAM Journal on Scientific Computing*, 15(4):938–952, 1994. doi: 10.1137/0915056. URL `https://doi.org/10.1137/0915056`.

Jacek Kierzenka and Lawrence F Shampine. A bvp solver based on residual control and the maltab pse. *ACM Transactions on Mathematical Software (TOMS)*, 27(3):299–316, 2001.

Brian C Fabien. Parallel collocation solution of index-1 bvp-daes arising from constrained optimal control problems. *Numerical Algorithms*, 71(2):311–335, 2016a.

Hans Georg Bock and Karl-Josef Plitt. A multiple shooting algorithm for direct solution of optimal control problems. *IFAC Proceedings Volumes*, 17(2):1603–1608, 1984.

Brian C Fabien. Parallel indirect solution of optimal control problems. *Optimal Control Applications and Methods*, 35(2):204–230, 2014a.

Brian C Fabien. Indirect solution of inequality constrained and singular optimal control

problems via a simple continuation method. *Journal of Dynamic Systems, Measurement, and Control*, 136(2):021003, 2014b.

David H Jacobson, Milind M Lele, and Jason L Speyer. New necessary conditions of optimality for control problems with state-variable inequality constraints. *Journal of mathematical analysis and applications*, 35(2):255–284, 1971.

Matthias Gerdts. Global convergence of a nonsmooth newton method for control-state constrained optimal control problems. *SIAM Journal on Optimization*, 19(1):326–350, 2008.

Brian C Fabien. A noninterior continuation method for constrained optimal control problems. In *Control Conference (ECC), 2016 European*, pages 1598–1603. IEEE, 2016b.

Christian Kanzow. Some noninterior continuation methods for linear complementarity problems. *SIAM Journal on Matrix Analysis and Applications*, 17(4):851–868, 1996.

MATLAB. *version 7.10.0 (R2017b)*. The MathWorks Inc., Natick, Massachusetts, 2017.

Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

Nicholas Gould and Jennifer Scott. A note on performance profiles for benchmarking software. *ACM Transactions on Mathematical Software (TOMS)*, 43(2):15, 2016.

Ernst Hairer and Gerhard Wanner. Solving ordinary differential equations ii: Stiff and differential-algebraic problems second revised edition with 137 figures. *Springer Series in Computational Mathematics*, 14, 1996.

Pierluigi Amodio, JR Cash, G Roussos, RW Wright, Graeme Fairweather, Ian Gladwell, GL Kraut, and Marcin Paprzycki. Almost block diagonal linear systems: sequential and parallel solution techniques, and applications. *Numerical linear algebra with applications*, 7(5):275–317, 2000.

Duane Storti and Mete Yurtoglu. *CUDA for engineers: an introduction to high-performance parallel computing*. Addison-Wesley Professional, 2015.

John Nickolls, Ian Buck, and Michael Garland. Scalable parallel programming. In *2008 IEEE Hot Chips 20 Symposium (HCS)*, pages 40–53. IEEE, 2008.

David Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*, pages 836–838. IEEE, 2008.

Guido Van Rossum and Fred L Drake Jr. *Python tutorial*, volume 620. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

Angelo Miele. Gradient algorithms for the optimization of dynamic systems. In *control and Dynamic systems*, volume 16, pages 1–52. Elsevier, 1980.

Paolo Novati. Some secant approximations for rosenbrock w-methods. *Applied Numerical Mathematics*, 58(3):195–211, 2008.

Lawrence F Shampine, Mark W Reichelt, and Jacek A Kierzenka. Solving index-1 daes in matlab and simulink. *SIAM review*, 41(3):538–552, 1999.

Donald E Kirk. *Optimal control theory: an introduction*. Courier Corporation, 2012.

Christof Büskens and Helmut Maurer. Sqp-methods for solving optimal control problems with control and state constraints: adjoint variables, sensitivity analysis and real-time control. *Journal of computational and applied mathematics*, 120(1-2):85–108, 2000.

Roger Alexander. Solving ordinary differential equations i: Nonstiff problems (e. hairer, sp norsett, and g. wanner). *Siam Review*, 32(3):485, 1990.

Uri Ascher. On numerical differential algebraic problems with application to semiconductor device simulation. *SIAM journal on numerical analysis*, 26(3):517–538, 1989.

David C Brock and Gordon E Moore. *Understanding Moore's law: four decades of innovation*. Chemical Heritage Foundation, 2006.

Joshua A Anderson, Chris D Lorenz, and Alex Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of computational physics*, 227(10):5342–5359, 2008.

Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Müller-Wittig. Molecular dynamics simulations on commodity gpus with cuda. In *International Conference on High-Performance Computing*, pages 185–196. Springer, 2007.

Michael C Schatz, Cole Trapnell, Arthur L Delcher, and Amitabh Varshney. High-throughput sequence alignment using graphics processing units. *BMC bioinformatics*, 8(1):1–10, 2007.

John Michalakes and Manish Vachharajani. Gpu acceleration of numerical weather prediction. *Parallel Processing Letters*, 18(04):531–548, 2008.

Sergio Barrachina, Maribel Castillo, Francisco D Igual, Rafael Mayo, and Enrique S Quintana-Ortí. Solving dense linear systems on graphics processors. In *European Conference on Parallel Processing*, pages 739–748. Springer, 2008.

Hsi-Yu Schive, Yu-Chih Tsai, and Tzihong Chiueh. Gamer: a graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics. *The Astrophysical Journal Supplement Series*, 186(2):457, 2010.

Peng Wang, Tom Abel, and Ralf Kaehler. Adaptive mesh fluid simulations on gpu. *New Astronomy*, 15(7):581–589, 2010.

Xinsheng Qin, Randall J LeVeque, and Michael R Motley. Accelerating an adaptive mesh refinement code for depth-averaged flows using gpus. *Journal of Advances in Modeling Earth Systems*, 11(8):2606–2628, 2019.

John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, page 7. ACM, 2015.

Nvidia. *CUDA C programming guide*. Nvidia Corporation, Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2019.

Kiyotaka Shimizu and S Ito. Constrained optimization in hilbert space and a generalized dual quasi-newton algorithm for state-constrained optimal control problems. *IEEE transactions on automatic control*, 39(5):982–986, 1994.

K. L. Teo. A unified computational approach to optimal control problems. In *Proceedings of the First World Congress on World Congress of Nonlinear Analysts, Volume III*, WCNA '92, page 2763–2774, USA, 1996. Walter de Gruyter & Co. ISBN 311013215X.

Boris Houska, Hans Joachim Ferreau, and Moritz Diehl. Acado toolkit—an open-source framework for automatic control and dynamic optimization. *Optimal Control Applications and Methods*, 32(3):298–312, 2011. doi: 10.1002/oca.939. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/oca.939`.

Jinhai Chen and Matthias Gerdts. Smoothing technique of nonsmooth newton methods for control-state constrained optimal control problems. *SIAM Journal on Numerical Analysis*, 50(4):1982–2011, 2012. doi: 10.1137/110822177. URL `https://doi.org/10.1137/110822177`.

Ivo Babuška and Manil Suri. The p- and h-p versions of the finite element method, an overview. *Computer Methods in Applied Mechanics and Engineering*, 80(1):5 – 26, 1990. ISSN 0045-7825. doi: https://doi.org/10.1016/0045-7825(90)90011-A. URL `http://www.sciencedirect.com/science/article/pii/004578259090011A`.

Ivo Babuška and Manil Suri. The p and h-p versions of the finite element method, basic principles and properties. *SIAM Review*, 36(4):578–632, 1994. doi: 10.1137/1036141. URL `https://doi.org/10.1137/1036141`.

Wen-zhuang Gui and Ivo Babuška. Theh, p andh-p versions of the finite element method in 1 dimension. *Numerische Mathematik*, 49(6):613–657, 1986.

Árpád Galvão, Marc Gerritsma, and Bart De Maerschalck. hp-adaptive least squares spectral element method for hyperbolic partial differential equations. *Journal of Computational and Applied Mathematics*, 215(2):409–418, 2008.

CA Dorao and HA Jakobsen. hp-adaptive least squares spectral element method for population balance equations. *Applied Numerical Mathematics*, 58(5):563–576, 2008.

CJ Goh and KL Teo. Miser: a fortran program for solving optimal control problems. *Advances in Engineering Software (1978)*, 10(2):90–99, 1988.

Oskar Von Stryk and Roland Bulirsch. Direct and indirect methods for trajectory optimization. *Annals of operations research*, 37(1):357–373, 1992.

Albert L Herman and Bruce A Conway. Direct optimization using collocation based on high-order gauss-lobatto quadrature rules. *Journal of Guidance, Control, and Dynamics*, 19(3):592–599, 1996.

Albert Lee Herman. *Improved collocation methods with application to direct trajectory optimization.* PhD thesis, University of Illinois at Urbana-Champaign, 1995.

Bengt Fornberg. *A practical guide to pseudospectral methods*, volume 1. Cambridge university press, 1998.

Lloyd N Trefethen. *Spectral methods in MATLAB*, volume 10. Siam, 2000.

Qi Gong, Fariba Fahroo, and I Michael Ross. Spectral algorithm for pseudospectral methods in optimal control. *Journal of Guidance, Control, and Dynamics*, 31(3):460–471, 2008.

Lars Grüne. An adaptive grid scheme for the discrete hamilton-jacobi-bellman equation. *Numerische Mathematik*, 75(3):319–337, 1997.

Rémi Munos and Andrew Moore. Variable resolution discretization in optimal control. *Machine learning*, 49(2-3):291–323, 2002.

Christopher L Darby, William W Hager, and Anil V Rao. An hp-adaptive pseudospectral method for solving optimal control problems. *Optimal Control Applications and Methods*, 32(4):476–502, 2011a.

Christopher L Darby, William W Hager, and Anil V Rao. Direct trajectory optimization using a variable low-order adaptive pseudospectral method. *Journal of Spacecraft and Rockets*, 48(3):433–445, 2011b.

Michael A Patterson, William W Hager, and Anil V Rao. A ph mesh refinement method for optimal control. *Optimal Control Applications and Methods*, 36(4):398–421, 2015.

Fengjin Liu, William W Hager, and Anil V Rao. Adaptive mesh refinement method for optimal control using nonsmoothness detection and mesh size reduction. *Journal of the Franklin Institute*, 352(10):4081–4106, 2015.

Jisong Zhao and Shuang Li. Adaptive mesh refinement method for solving optimal control problems using interpolation error analysis and improved data compression. *Journal of the Franklin Institute*, 357(3):1603–1627, 2020.

John T Betts. *Practical methods for optimal control and estimation using nonlinear programming*. SIAM, 2010.

Josef Stoer and Roland Bulirsch. *Introduction to numerical analysis*, volume 12. Springer Science & Business Media, 2013.

Anil V Rao and Kenneth D Mease. Eigenvector approximate dichotomic basis method for solving hyper-sensitive optimal control problems. *Optimal Control Applications and Methods*, 21(1):1–19, 2000.

Asen L Dontchev. An a priori estimate for discrete approximations in nonlinear optimal control. *SIAM journal on control and optimization*, 34(4):1315–1328, 1996.

Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi,

Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL `https://doi.org/10.1038/s41586-020-2649-2`.

D Jacobson and M Lele. A transformation technique for optimal control problems with a state variable inequality constraint. *IEEE Transactions on Automatic Control*, 14(5): 457–464, 1969.

Milton Abramowitz and Irene A Stegun. *Handbook of mathematical functions: with formulas, graphs, and mathematical tables*, volume 55. Courier Corporation, 1965.

Andrew P Sage and Chelsea C White. *Optimum systems control*. Prentice Hall, 1977.

Wilhelm Heinrichs. An adaptive spectral least-squares scheme for the burgers equation. *Numerical Algorithms*, 44(1):1–10, 2007.

Fengjin Liu, William W Hager, and Anil V Rao. An hp mesh refinement method for optimal control using discontinuity detection and mesh size reduction. In *53rd IEEE Conference on Decision and Control*, pages 5868–5873. IEEE, 2014.

Claudio Canuto, M Yousuff Hussaini, Alfio Quarteroni, A Thomas Jr, et al. *Spectral methods in fluid dynamics*. Springer Science & Business Media, 2012.

James E Cuthrell and Lorenz T Biegler. On the optimization of differential-algebraic process systems. *AIChE Journal*, 33(8):1257–1270, 1987.

James E Cuthrell and Lorenz T Biegler. Simultaneous optimization and solution methods for batch reactor control profiles. *Computers & Chemical Engineering*, 13(1-2):49–62, 1989.

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(1):81–126, 2008.

L Lasdon, A Waren, and R Rice. An interior penalty method for inequality constrained optimal control problems. *IEEE Transactions on Automatic Control*, 12(4):388–395, 1967.

Audrey Hermant. Homotopy algorithm for optimal control problems with a second-order state constraint. *Applied Mathematics and Optimization*, 61(1):85, 2010.