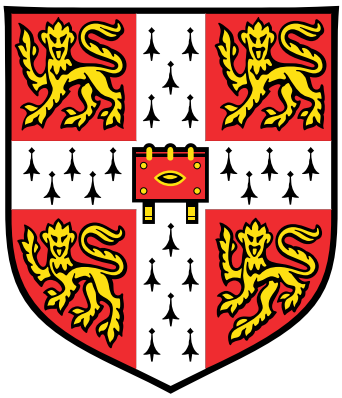
Predictive Simulation of Musculoskeletal Models Using Direct Collocation



Samuel George Brockie

Department of Engineering
University of Cambridge

This thesis is submitted for the degree of
Doctor of Philosophy

Gonville & Caius College September 2021

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 71,500 words including footnotes, tables and equations and has fewer than 150 figures.

Sam Brockie, September 2021

Predictive Simulation of Musculoskeletal Models Using Direct Collocation

Samuel George Brockie

Applications of biomechanical predictive simulation are wide ranging, with the technique used to provide insights into movement disorders, sports performance, and injury prevention. However, current software provision has limitations. Users are restricted from leveraging state-of-the-art methods and algorithms. Alternatively, they are required to develop bespoke implementations of direct collocation, or laboriously manually link multiple software packages. In order to address these limitations, this research aims to develop and critically evaluate a software suite that enables both expert and non-expert users to construct and solve predictive simulation *optimal control problems* (OCPs) involving musculoskeletal models.

Solving OCPs is a critical part of predictive simulation. Algorithms for transcription, scaling, mesh refinement, and derivative generation are presented, along with their implementations in an open-source software package for numerically solving OCPs, *Pycollo*. Benchmarking of *Pycollo* against an industry-standard commercial software package, *GPOPS-II*, by solving five known OCPs from the literature demonstrates comparable convergence and computational performance, with *Pycollo* requiring fewer mesh iterations and sparser discretisation meshes to meet defined error tolerances in four out of five cases.

Biomechanical predictive simulations also require the ability to derive multibody dynamics and implement musculotendon models. Furthermore, these need to be formulated in a way suitable for OCPs. Two software packages, *Pynamics* and *Pyomechanics*, which formulate multibody dynamics and musculoskeletal OCPs respectively, are presented. Comparison of explicit and implicit formulations of multibody dynamics shows that solution accuracies, solve times, convergence rates, and discretisation errors are improved when implicit dynamics are used. Similarly, comparison of multiple musculotendon formulations and their numerical sensitivity finds that implicit musculotendon equations offer the best numerical properties for OCPs and should be preferred. Testing of solution sensitivity to the sigmoidal smoothing coefficient in continuous activation dynamics suggests a value of 100 should be preferred over the previously published recommendation of 10.

Acknowledgements

I would like to express my profound gratitude to my supervisor, Dr David Cole, for his invaluable advice, continuous support, and vital academic guidance. This thesis owes greatly to his insights and encouragement over the past five years, and I am particularly grateful for his caring pastoral support throughout the duration of my postgraduate studies.

I would like to sincerely thank Professor Tony Purnell, whose mentorship and guidance fostered a critical and innovative approach to my research, and has taught me to not shy away from ambitious projects. His consistent and enthusiastic support in both my academic and professional life is greatly appreciated.

I am grateful to my examiners, Professor Garth Wells and Professor Mark King, for their invaluable feedback and suggestions, which helped shape the final version of this thesis into something I could be even more proud of.

I would like to thank Dr Geoff Parks, for setting such an excellent example of a proactive and engaged supervisor throughout our collaboration, from which I could learn.

I owe a great thanks to Dr Digby Symons, for starting me off on my academic journey by asking whether cyclists should pedal backwards.

I would also like to thank Billy Fitton, whose collaboration was not only academically valuable, but showed me that you can love the subject matter of your research.

I am grateful to Dr Paul Barratt and Paul Mullan, for supporting my continued involvement with the Great Britain Cycling Team.

Finally, I owe great thanks to British Cycling, the English Institute of Sport, and the Engineering and Physical Sciences Research Council, for their financial support, which made this thesis possible.

Contents

List of Figures

List of Tables

Nomenclature

Orthogonal Collocation

p	phase
J	objective function
t	time
τ	nondimensionalised time
$\mathbf{y}^{(p)}(t)$	state in phase p
$\mathbf{u}^{(p)}(t)$	control in phase p
$\mathbf{q}^{(p)}$	integral in phase p
$t_0^{(p)}$	initial time of phase p
$t_F^{(p)}$	final time of phase p
$\mathbf{s}$	static parameter
$\boldsymbol{\gamma}^{(p)}$	path constraints in phase p
$\mathbf{b}$	endpoint constraints
$y(t)$	state variable
$u(t)$	control variable
q	integral variable
s	static parameter variable
g	integrand function

$\mathbf{z}$	decision variables
$\mathbf{g}(\mathbf{z})$	equality constraints
$\mathbf{h}(\mathbf{z})$	inequality constraints
$\mathbf{c}(\mathbf{z})$	constraints
$\mathcal{S}$	mesh
$\mathcal{J}$	transcribed NLP objective function
$\mathbf{X}$	transcribed NLP decision variables
$\mathbf{C}$	transcribed NLP constraints
$\mathbf{Y}$	transcribed NLP state variables
$\mathbf{U}$	transcribed NLP control variables
$\boldsymbol{\Delta}$	transcribed NLP defect constraints
$\boldsymbol{\Gamma}$	transcribed NLP path constraints
$\boldsymbol{\Lambda}$	transcribed NLP integral constraints
$\mathbf{w}$	transcribed NLP quadrature weights
$\boldsymbol{\beta}$	transcribed NLP endpoint constraints
$\mathbf{E}$	state difference matrix
$\mathbf{A}$	integration matrix
$\mathbf{g}$	objective gradient
$\mathbf{G}$	constraints Jacobian
$\mathbf{H}$	Lagrangian Hessian
$\mathcal{L}$	Lagrangian
σ	objective factor
λ	Lagrange multiplier
$\tilde{x}$	scaled variable
$\mathbf{V}$	decision variable stretching weights
$\mathbf{r}$	decision variable translation weights

$\mathbf{W}$	constraint scaling weights
$w_{\mathcal{J}}$	objective scaling weight
h	mesh section width
α	stage coefficient
β	step coefficient
ρ	stage fraction
ι	stage collocation start index
κ	stage collocation end index

Derivative Generation

x	function input
y	function output
w	auxiliary (primal) variables
$\dot{w}$	tangent variable
$\bar{w}$	adjoint variable
$\mathbb{T}_i$	tier i
$\mathbf{\Delta}_{\mathbb{T}_i}$	delta matrix for tier i
K	number of mesh sections
$\mathcal{T}$	derivative preprocessing time during OCP setup
$\mathcal{P}$	derivative preprocessing time per mesh iteration
$\mathcal{I}$	NLP function evaluation time per NLP iteration

Multibody Dynamics

$\hat{\mathbf{n}}$	unit vector
$\mathfrak{q}$	generalised coordinate
$\mathfrak{u}$	generalised speed
$\mathfrak{a}$	generalised acceleration
$\mathbf{M}$	mass matrix

$\mathbf{k}$	forcing vector

Musculoskeletal Modelling

l^{MT}	musculotendon length
v^{MT}	musculotendon shortening velocity
l^{T}	tendon length
v^{T}	tendon shorting velocity
l^{M}	muscle fibre length
v^{M}	muscle fibre shortening velocity
F^{T}	tendon force
F^{M}	muscle fibre force
α	pennation angle
β	muscle fibre damping
a	activation
e	excitation
l^{T}_{slack}	tendon slack length
l^{M}_{opt}	optimal muscle fibre length
v^{M}_{max}	muscle fibre maximum shortening velocity
F^{M}_{max}	muscle fibre maximum isometric force
α_{opt}	pennation angle at optimum muscle fibre length
τ_{act}	activation time constant
τ_{deact}	deactivation time constant
$\tilde{l}^{T}$	normalised tendon length
$\tilde{v}^{T}$	normalised tendon shortening velocity
$\tilde{l}^{M}$	normalised muscle fibre length
$\tilde{v}^{M}$	normalised muscle fibre shortening velocity
$\tilde{F}^{T}$	normalised tendon force

$\tilde{F}^M$	normalised muscle fibre force
$\mathrm{fl}^T\left(\tilde{l}^T\right)$	normalised tendon force-length characteristic
$\mathrm{fl}^M_{\mathrm{pas}}\left(\tilde{l}^M\right)$	normalised muscle fibre passive force-length characteristic
$\mathrm{fl}^M_{\mathrm{act}}\left(\tilde{l}^M\right)$	normalised muscle fibre active force-length characteristic
$\mathrm{fv}^M\left(\tilde{v}^M\right)$	normalised muscle fibre velocity-length characteristic
$\mathcal{J}$	optimal cost
$\mathcal{N}$	number of NLP iterations

Acronyms

2D	two-dimensional
3D	three-dimensional
ABC	abstract base class
AD	algorithmic differentiation
API	application programming interface
BFGS	Broyden-Fletcher-Goldfarb-Shanno
BPST	Biomechanics Predictive Simulation Toolkit
BVP	boundary value problem
CAS	computer algebra system
CMG	control moment gyroscope
CPU	central processing unit
CT	computed tomography
DAE	differential-algebraic system of equations
DAG	directed acyclic graph
DFS	depth-first search
DoF	degree of freedom
EoM	equation of motion

ET	evaluation trace
FBD	free body diagram
FD	finite differencing
hSAD	hybrid-symbolic-algorithmic differentiation
IP	interior-point
IVP	initial value problem
JIT	just-in-time
LG	Legendre-Gauss
LGL	Legendre-Gauss-Lobatto
LGR	Legendre-Gauss-Radau
LHS	left hand side
LLOC	logical lines of code
MD	manual differentiation
MRI	magnetic resonance imaging
NaN	not a number
NLP	nonlinear programming problem
NRMSE	normalised root-mean-square error
OCP	optimal control problem
ODE	ordinary differential equation
OO	operator overloading
OOP	object-oriented programming
PCSA	physiological cross-sectional area
PDE	partial differential equation
QP	quadratic programming problem

RAM	random-access memory
RHS	right hand side
RMSE	root-mean-square error
SD	symbolic differentiation
SIMD	single instruction, multiple data
SLOC	source lines of code
SQP	sequential quadratic programming
ST	source transformation
UML	Unified Modelling Language

Chapter 1

Introduction

1.1 Overview of Current Research

1.1.1 Practical Application of Musculoskeletal Simulation

Simulation of musculoskeletal models is becoming an increasingly applied and important tool in the field of biomechanics. Applications are wide-ranging. Musculoskeletal simulation is used to better understand movement in able-bodied humans, showing how the various muscles contribute to propulsion in walking [22, 277] and running [145, 146]. Musculoskeletal simulation also gives insight into movement disorders. Its application discovers ways to reduce knee loading in those with osteoarthritis [117, 119], shows that individuals with cerebral palsy exhibit simplified motor control strategies [311], demonstrates that muscle weakness affects gait [108, 257], and highlights the increased likelihood of injury from walking with a crouch gait [160, 166]. Biomechanical models contribute to medical treatments and injury prevention. The understanding of joint forces from simulations helps to inform surgical interventions such as knee replacements [127]. Further injury prevention applications include minimising the effects of low-gravity environments [118], designing exoskeletons that can assist with the lifting of heavy objects [225], and investigating knee injuries in football players [317].

An extensive range of sports are studied using musculoskeletal simulation, identifying factors that enhance performance. Applications range across gymnastics [334, 337], tennis [195, 196], swimming [207], diving [336], trampolining [335], golf [92], ski jumping [174], cricket [347] and cycling [53, 280].

Musculoskeletal modelling and simulation is not limited to humans. Researchers

in the field of zoology simulate the motion of horses [26], chimpanzees [255], mice [76] and ostriches [181, 281]. Even dinosaur locomotion has been simulated [180, 296], allowing the study of extinct creatures' movement, which would not be possible without these computational methods.

1.1.2 Methods and Software for Musculoskeletal Simulation

In the early years of musculoskeletal modelling, researchers were required to develop bespoke models and simulation frameworks for each individual study [150]. To meet the needs of the biomechanics research community, a number of software packages and frameworks for musculoskeletal modelling and simulation were developed. These include the proprietary software package *AnyBody* [285], the *MATLAB* analysis toolbox *Biomechanics of Bodies* [231], and the open-source software package *OpenSim* [87]. All three deliver high-performance multibody dynamics and simulation capabilities (*OpenSim* uses the open-source, *C++* dynamics engine, *Simbody* [299]). *AnyBody* and *OpenSim* provide extensive, high level of abstraction component libraries of musculoskeletal modelling elements, which allow users to easily develop, save and share their models. *OpenSim* is the most widely-used software package for musculoskeletal simulation due to its permissible open-source license [87]. It also provides a suite of analysis tools, including visualisation capabilities, and scripting *application programming interfaces* (APIs) for *C++*, *MATLAB* and *Python*. APIs are particularly important as they allow researchers to modify or extend the package, interface with other software packages, and run investigations programmatically [87].

Musculoskeletal simulation can involve using data describing the kinematics of a movement to estimate the forces and moments responsible. This is known as *inverse dynamics* [6, 346]. In inverse dynamics, the kinematics data can first be determined by experiments involving real participants [272, 301]. A torque-driven musculoskeletal model can then used to determine the joint moments that produce the prescribed motion [27, 346]. Alternatively, a muscle-driven model can be used to estimate the contributions of individual muscles [301, 346]. Inverse dynamics has also been extended to account for impact forces [46].

An alternative methodology in movement simulation is *forward dynamics* [6, 346]. Here, the motion is determined by a prescribed control strategy and a set of initial conditions. Forward dynamics simulations can be conducted without the need for experimental kinematic data. As they are not biased or constrained by predetermined movement patterns, they can be used as a prediction tool to help determine cause-effect relationships [251, 280] and to design treatment interventions [257, 265].

1.1.3 Musculoskeletal Simulations as Optimal Control Problems

Musculoskeletal simulations are often well suited to being posed as *optimal control problems* (OCPs) [57], in which a system's time-dependent dynamics and control are determined to minimise an *objective function* [36]. Inverse dynamics simulations can be formulated as OCPs. An example is the *muscle redundancy problem*, in which the objective is to estimate the muscle forces that produce a given movement. This problem is difficult because there are more unknown muscle forces than kinematic constraints. Framing the muscle redundancy problem as an OCP and prescribing a cost for minimisation, such as muscular effort, allows it to be solved efficiently [85].

Forward dynamics problems posed as OCPs are termed *predictive simulations* [59]. Kinematics are unspecified in forward dynamics, so in this type of OCP it is the control strategy that is determined in order to minimise the objective function. The kinematic trajectory is a result of the optimised control, and thus motion is predicted. Motion tracking can also be conducted by making the objective function the error between the simulated motion and a motion prescribed by experimental kinematic data [218].

Direct shooting methods have been used to solve predictive simulation OCPs with some success [257, 300, 307]. Using this method, just the control is parameterised as part of the OCP and a forward simulation is conducted at every iteration during solving [36]. The open-source biomechanical optimisation framework *SCONE* [126] allows users to solve predictive simulation OCPs involving *OpenSim* models using direct shooting. This method is well known to exhibit high sensitivity between the variables and the constraints [36]. As such, predictive simulations that employ this method exhibit long solve times (multiple hundreds of hours of compute time [257]) and poor convergence properties [57]. Furthermore, when direct shooting is used, the musculoskeletal models are often simplified by assuming planar *two-dimensional* (2D) motion [225, 300], reducing the number of muscles and simplifying their pathways [84, 333], or using a simplified control scheme, such as on-off excitations [251, 307].

The *direct collocation* method is widely used to solve OCPs involving musculoskeletal models [57, 85, 108, 212, 241, 269]. This method, predominantly developed in the field of aerospace [67, 268], uses polynomial splines to approximate a system's state and control [36, 194]. The system's dynamics are enforced by *collocating* the time derivatives of the state splines with the system's differential dynamic equations at a set of knot points [23, 143]. Direct collocation produces a *nonlinear programming problem* (NLP), with a discretisation of the system's state and control as the

decision variables and the system's dynamics enforced as constraints. This can contain many more decision variables and constraints than an equivalent direct shooting formulation. However, the constraints enforcing the system's dynamics only depend on a small number of the decision variables. The resulting large, sparse NLP is tractable to solve, which can be done by existing methods and software [43, 129].

The advantages of direct collocation has led to its use solving the muscle redundancy problem [85], tracking motions [218], and optimising the design of prostheses [290]. It has also been applied to predict maximal jumping [269], running [243], and walking gait in both healthy [6] and impaired individuals [108]. This research has led to important methodological advances for the application of direct collocation to musculoskeletal simulation. Implicit formulation of dynamic equations has been investigated [57], differentiable algebraic expressions for musculotendon properties have been suggested [85], and *algorithmic differentiation* (AD) has been shown to allow complex models to be simulated more rapidly [108]. Methodological advances accompany discoveries in practical areas, including identifying skipping as the preferred locomotion strategy in low-gravity environments [5], and ways that amputees can improve gait symmetry [203].

Direct collocation is complex and difficult to implement, requiring:

1. the continuous-time OCP to be discretised into a finite-dimensional NLP subproblem using a collocation scheme [36, 121];
2. a number of first- and second-order derivatives to be computed so that gradient-based methods can be used to solve the NLP subproblem [43, 129]; and
3. the implementation of musculoskeletal modelling to meet requirements, such as any functions associated with the OCP be at least second-order continuous [36].

As such, the method is not as widely used as might be expected given its advantages. Predictive simulation studies, where *OpenSim* is used with a bespoke implementation of direct collocation, prove to be laborious for the researchers, do not employe state-of-the-art methods and exhibite slow convergence times [212, 269].

For researchers to solve musculoskeletal OCPs in practice, it is desirable to use an established optimal control software package, which can handle the complexities of direct collocation on behalf of the user. Examples of such packages include the proprietary software *GPOPS-II* [263] and the open-source software *PSOPT* [29]. The primary advantage of such packages is that they abstract the complexity of formulating OCPs using direct collocation away from the user. By not requiring the

user to understand the theory of, or implement, direct collocation, the methodology becomes more widely accessible. Optimal control software packages are also advantageous in that they implement state-of-the-art methods from the field of optimal control. High-order orthogonal collocation methods [38, 122], mesh error calculation [36], and mesh refinement [7, 262] ensure that the OCP is solved to a high degree of accuracy. Performant and accurate derivative-taking algorithms [10, 113, 325] and sparsity exploitation [8, 36, 264] enable OCPs to be solved efficiently. Automatic scaling of the OCP [29, 36] and sensible default solver settings [263] improve ease-of-use for non-expert users.

However, as the general-purpose optimal control software packages described above have not been designed with biomechanical modelling software in mind, they present a number of barriers to the integration of the two. Beyond the inconvenience and challenge of the user having to manually link multiple software packages in potentially multiple programming languages [212], they:

1. do not natively support the implicit formulation of dynamical equations, requiring the user to formulate these manually using additional control variables and path constraints [111];

2. cannot natively differentiate the equations governing a multibody system, requiring inaccurate differencing methods to be used to generate the OCP derivatives [9, 108]; and

3. can struggle to automatically detect the OCP sparsity when linked to external software, resulting in suboptimal performance [263].

GPOPS-II has been used with *OpenSim* to apply direct collocation to musculoskeletal predictive simulations [110], as well as other musculoskeletal OCPs [85, 241]. While these studies brought novel insight, laborious manual integration of the biomechanical and optimal control software packages was required. Furthermore, aspects of *OpenSim* needed to be customised in order to enable *GPOPS-II* to successfully solve the OCPs. These included implementing bespoke musculotendon models with suitable numerical properties [85] and manually approximating musculotendon lengths using smooth polynomials [241].

Moco is a software toolkit, and native extension to *OpenSim*, for the optimisation of the motion and control of musculoskeletal models [91]. It was first released in November 2019 and first published in December 2020 [91]. It is an easy-to-use, customisable and extensible software package that natively formulates musculoskeletal OCPs. Beyond defining their biomechanical model, the user is only required to

supply *Moco* with limited information defining their OCP, such as a choice of objective function and optional bounds on variables. It provides a set of predefined, selectable, biomechanics-specific, objective functions and utilities for imposing constraints on joint angles and periodicity. *Moco* is capable of solving OCPs that involve motion tracking, motion prediction, parameter optimisation, model fitting, and device design [91]. It has been used to predict a squat-to-stand motion while also optimising an assistive device [91], and investigate knee loads and injury risk in single leg jumping [140].

Moco is, however, limited in its scope by a number of design decisions imposed by *Simbody* and *OpenSim*. Firstly, the numerical nature of *Simbody* and *OpenSim* means that *Moco* can only utilise differencing methods to estimate, rather than determine exactly, OCP derivatives [87, 90, 299]. This is well-known to be less accurate and efficient than recent methods based on AD, resulting is worse convergence properties and longer solve times [10, 108]. It also, by default, relies on a quasi-Newton approximation for the second-order OCP derivatives, which is again less accurate and efficient than if this is provided exactly [36, 43]. Secondly, only certain *OpenSim* model components are supported for use in *Moco* OCPs [90]. Users are required to manually replace these with OCP-suitable ones [90], reducing ease-of-use. Finally, limitations of the *C++* architecture mean that it is not possible to provide a number of user-customisable options that would otherwise be desirable. For example, custom objective functions cannot be defined using the scripting APIs [91], meaning that if the provided options do not meet the needs of a user's research problem then *Moco* cannot be used.

Moco also does not support some advantageous features that are available with general-purpose optimal control software. For example, it only implements low-order collocation schemes (trapezoidal and Hermite-Simpson transcription schemes [91]), which are less accurate and efficient than orthogonal collocation [38, 122]. This, in combination with *Moco* not providing mesh error calculation and automatic mesh refinement [90], limits the accuracy of solutions that can be obtained. *Moco* does not support multiphase OCPs. This prohibits users from conducting simulations involving:

1. changes in system dynamics, such as the changes in ground contact at foot-off in jumping (e.g. [269]); and
2. constraints at intermediate times, such as enforcing a joint position or velocity at a specified time (e.g. [212]).

All previous attempts to apply direct collocation to musculoskeletal OCPs have been implemented by attempting to retrofit optimal control functionality to existing

biomechanics software [85, 91, 108, 212, 241, 269]. As outlined above, this prohibits the full application of state-of-the-art methods and algorithms from the field of optimal control, as compatibility with existing musculoskeletal modelling software has to be considered. Furthermore, it has the effect of constraining the way that musculoskeletal models are constructed and analysed.

1.2 Key Conclusions from the Literature

A number of points are clear from the reviewed literature (section 1.1):

1. Tools such as *OpenSim* and *Moco* greatly facilitate the ease with which musculoskeletal models can be constructed and predictive simulations can be formulated and solved. However, the methods and architectures they employ limit the extent to which state-of-the-art and novel algorithms can be leveraged.

2. Where it has been possible to apply direct collocation to biomechanical OCPs, this has lead to significant breakthroughs in the computational performance of predictive simulation.

3. Where it has been possible to utilise exact derivatives (e.g. through AD) when solving biomechanical OCPs, this has lead to significant breakthroughs in the computational performance of predictive simulation.

4. Studies where such progress has been made involve bespoke implementations and the complex integration of multiple software packages, which have often undergone modification, in potentially multiple programming languages. Such labour intensive and specialised approaches do not make them accessible to most practitioners and researchers.

In addition, research among practitioners has identified the following requirements:

1. Modelling tools need to be flexible so that musculoskeletal models can easily be modified or reparameterised. This is to enable the creation of athlete-specific models, including those of para-athletes.

2. Predictive simulation capability needs to be computationally performant so that sensitivity analyses, which require repeated computation of similar problems, can be conducted efficiently.

3. Software tools need to be user-friendly and accessible to biomechanics practitioners and researchers (who may not have expert knowledge in multibody and musculoskeletal dynamics, and optimal control theory).

1.3 Research Aim and Objectives

Based on the literature review and motivation for research in sections 1.1 and 1.2 respectively, the research aim of this thesis is to:

Develop and critically evaluate a comprehensive toolkit, to be known as the *Biomechanics Predictive Simulation Toolkit* (BPST), that enables both expert and non-expert users to construct and solve predictive simulation trajectory optimisation and OCPs involving musculoskeletal models using state-of-the-art theory and methods in the field of optimal control.

To meet this aim, the following research objectives have been identified:

1. Develop and critically evaluate a highly performant, easy-to-use, open-source software package to solve general OCPs, using state-of-the-art theory and methods, which can be used as a foundation element of the BPST.

2. Investigate methods for determining first- and second-order derivative information that reduce computational cost and maximise derivative evaluation speed during an OCP solve, and where practicable implement these as part of the BPST.

3. Develop and critically evaluate a highly-performant, easy-to-use, open-source software package for modelling multibody systems and their dynamics, specifically tailored for use in OCPs, which can be used as a core element of the BPST.

4. Develop and critically evaluate a highly-performant, easy-to-use, open-source software package capable of formulating and solving musculoskeletal predictive OCPs, by adding musculoskeletal modelling functionality and leveraging the capabilities of the other BPST packages.

1.4 Thesis Overview

This thesis contains four main chapters, each addressing one of the four research objectives laid out in section 1.3. Each chapter begins with a section reviewing the related literature and background knowledge relevant to the content. Chapter 2 addresses the first research objective, detailing the development of a number of algorithms related to direct collocation and a general-purpose software package for solving OCPs. In chapter 3, a derivative-taking algorithm suited specifically for application to OCPs is developed and evaluated. Chapter 4 addresses the third research objective by developing a software package for formulating and solving multibody OCPs, along with investigating the performance of a number of OCP formulations. The fourth research objective is addressed in chapter 5, in which the work of chapter 4 is extended to musculoskeletal modelling, the BPST is validated and a number of recommendations about formulating musculoskeletal predictive simulations are made. Chapter 6 summarises the key contributions of this thesis and discusses possible directions for future work.

Chapter 2

Orthogonal Collocation

Solving *optimal control problems* (OCPs) is an important part of predictively simulating musculoskeletal models. The chapter begins with a review of the methods used to numerically solve OCPs and the available software packages. The first objective from section 1.3 is restated, along with a number of sub-objectives, motivated by the review of the literature, that are required to achieve it. A framework and algorithm for treating Gaussian-, Radau- and Lobatto-based collocation as specific cases of a general collocation scheme are presented in section 2.3. Details of an approach to formulating a *nonlinear programming problem* (NLP) subproblem, along with a methodology for algorithmically scaling it, are presented in section 2.4. Section 2.5 describes a mesh refinement algorithm that is designed for use with Lobatto-based collocation. These features, together with state-of-the-art theory, are brought together in a new software package designed for numerically solving OCPs in section 2.6. This is then validated and benchmarked against industry standard software by solving a range of OCPs with known solutions from the literature. This chapter concludes with a discussion of the work presented, within the context of the literature, and recommends areas for further research and development.

2.1 Background, Theory and Review

OCPs arise in many fields including aerospace [32, 67, 267], chemical processing [213], manufacturing [314], medicine [187, 211] and robotics [294]. Solving an OCP (or *trajectory optimisation problem*) involves finding the control for a continuous dynamical system that minimises some objective function, while satisfying a set of constraints. OCPs can contain one or more *phases*, with a phase being a portion of the problem description within which there are consistent dynamics and

constraints. An OCP with more than one phase is a *multiphase problem*.

Trajectory optimisation first appeared in the late 17th century with the introduction of the Brachistochrone problem by Bernoulli [34] in which the objective is to find the trajectory of a point mass that minimises the time spent travelling between two points under the influence of gravity. The mathematical theory underpinning the solving of OCPs was established in the middle of the 20th century [52, 268]. Following the invention of the digital computer, real world OCPs became practical to investigate and solve. As a result, the solving of OCPs has been widely studied in recent years with many approaches and advancements reported in the academic literature [35, 105, 282].

2.1.1 The Multiphase Optimal Control Problem

A multiphase OCP with phases $p \in [1, \ldots, P]$ involves determining the state $\mathbf{y}^{(p)}(t) \in \mathbb{R}^{n_y^{(p)}}$, control $\mathbf{u}^{(p)}(t) \in \mathbb{R}^{n_u^{(p)}}$, integrals $\mathbf{q}^{(p)} \in \mathbb{R}^{n_q^{(p)}}$, phase initial times $t_0^{(p)} \in \mathbb{R}$ and phase final times $t_F^{(p)} \in \mathbb{R}$ in each p along with the global parameters $\mathbf{s} \in \mathbb{R}^{n_s}$ that minimise the objective function

$$J = \Phi\left(\left[\mathbf{e}^{(1)}, \ldots, \mathbf{e}^{(P)}, \mathbf{s}\right]\right) , \tag{2.1}$$

subject to the dynamical constraints

$$\dot{\mathbf{y}}^{(p)} = \mathbf{a}^{(p)}\left(\left[\mathbf{y}^{(p)}(t), \mathbf{u}^{(p)}(t), t, \mathbf{s}\right]\right) , \quad (p = 1, \ldots, P) , \tag{2.2}$$

the path constraints

$$\boldsymbol{\gamma}_{min}^{(p)} \leq \boldsymbol{\gamma}^{(p)}\left(\left[\mathbf{y}^{(p)}(t), \mathbf{u}^{(p)}(t), t, \mathbf{s}\right]\right) \leq \boldsymbol{\gamma}_{max}^{(p)} , \quad (p = 1, \ldots, P) , \tag{2.3}$$

the integral constraints

$$\mathbf{q}_{min}^{(p)} \leq \mathbf{q}^{(p)} \leq \mathbf{q}_{max}^{(p)} , \quad (p = 1, \ldots, P) \tag{2.4}$$

and the endpoint constraints

$$\mathbf{b}_{min} \leq \mathbf{b}\left(\left[\mathbf{e}^{(1)}, \ldots, \mathbf{e}^{(P)}, \mathbf{s}\right]\right) \leq \mathbf{b}_{max} , \tag{2.5}$$

where

$$\mathbf{e}^{(p)} = \left[\mathbf{y}^{(p)}\left(t_0^{(p)}\right), \mathbf{y}^{(p)}\left(t_F^{(p)}\right), \mathbf{q}^{(p)}, t_0^{(p)}, t_F^{(p)}\right] , \quad (p = 1, \ldots, P) \tag{2.6}$$

and with the integrals in p

$$q_i^{(p)} = \int_{t_0^{(p)}}^{t_F^{(p)}} g_i^{(p)}\left(\left[\mathbf{y}^{(p)}(t), \mathbf{u}^{(p)}(t), t, \mathbf{s}\right]\right) dt , \quad \left(i = 1, \ldots, n_q^{(p)}\right) , \quad (p = 1, \ldots, P) . \tag{2.7}$$

J is a *Bolza* objective function [51, 60] as it may be a function of both event variables $\mathbf{e}^{(p)}$ and $\mathbf{s}$ (*Mayer* terms [49, 232]), and may contain integrand terms through also being a function of integral variables $\mathbf{q}^{(p)}$ (*Lagrange* terms [50]). The dynamical constraints (eq. (2.2)) are *ordinary differential equations* (ODEs) of the independent variable t.

Solution of the OCP defined by eqs. (2.1) to (2.7) has historically depended on the application of the *calculus of variations* [268] to derive the first-order *necessary conditions* for optimality (or *Euler-Lagrange equations*) [68, 201, 215]. The necessary conditions specify values for the variables where the objective function is at a turning point and all of the constraints are satisfied [268]. The second-order *sufficiency conditions* can be derived and evaluated to determine whether the solution to the necessary conditions (the *extremal solution*) is a maximum or a minimum [268]. A mathematical description of the optimality conditions for OCPs can be found in [36, 268].

2.1.2 Solving Optimal Control Problems

While the simplest of OCPs can be solved analytically, as problem complexity increases, finding an analytical solution becomes intractable, if not impossible [36]. Therefore, numerical methods must be used to solve most OCPs. The use of numerical methods is caveated by the fact that they must be applied to finite-dimensional problems; an OCP is continuous by nature and therefore infinite-dimensional [23]. One practical numerical method involves recasting the problem in terms of a finite set of variables and constraints, such that it can then be characterised as a NLP [23, 36].

The highest-level categorisation of numerical methods contains two approaches: *direct methods* [215], which find a minimum of the objective function J; and *indirect methods* [68], which solve for a root of the necessary conditions. Direct methods construct a discrete approximation to the state and control, which is then solved numerically [36, 282]. The approach (the *transcription method* [36]) involves:

1. converting the continuous-time OCP to a NLP finite-dimensional approximation (the *transcription step*);

2. solving the NLP (the *optimisation step*); and

3. assessing the accuracy of the finite approximation, and repeating the transcription and optimisation steps if required (the *refinement step*).

Instead, *indirect methods* utilise solutions to the optimality conditions to find optimal solutions [36, 282]. A root of the necessary conditions yields a *boundary value problem* (BVP) that can be solved numerically [23]. Indirect methods present the advantage that the state and *costate* (the necessary conditions relating to the dynamical constraints (eq. (2.2)) and their Lagrange multipliers) are explicitly solved, meaning that the optimal solution can be readily verified [35, 282].

Direct methods are typically preferred as a number of difficulties arise when practically implementing indirect methods [35, 36, 105, 282]:

1. indirect methods require the necessary conditions to be derived, a process which typically cannot be automated and so places additional demands on the user [68, 282];
2. the solution to the BVP generated by an indirect method is very sensitive to the initial guess [35] and ill-conditioned costate equations [68]; and
3. inequality path constraints require accurate initial guesses when they are active for only a portion of the problem's time duration [35, 36].

Due to these practical difficulties with indirect methods, this thesis focuses solely on direct methods.

Numerical Integration

Numerical integration methods are concerned with the numerical solving of differential equations [23, 71]. As OCPs contain differential dynamical constraints (eq. (2.2)) and quadrature constraints (eqs. (2.4) and (2.7)), optimal control and numerical integration are closely tied. Specifically, the multiphase OCP considered in this thesis (section 2.1.1) involves dynamical constraints that are first-order ODEs. Therefore, only numerical methods for solving ODEs are considered.

Two general classes of numerical integration schemes exist: *multistep methods*, where the solution at time t_{k+1} is obtained from a series of adjacent times $t_k, \ldots, t_{k-j}$ with $j \in \mathbb{Z}$; and *multistage methods*, where each time interval $[t_k, t_{k+1}]$ is divided into a number of subintervals [23, 71, 282]. Multistep methods have the general form

$$\mathbf{y}_{i+k} = \sum_{j=0}^{k-1} \alpha_j \mathbf{y}_{i+j} + h \sum_{j=0}^{k} \beta_j \dot{\mathbf{y}}_{i+j} \,, \tag{2.8}$$

where α_j and β_j are constants associated with the specific method [71]. A method is *explicit* if $\beta_k = 0$ such that the value for a timepoint only depends on values

that have been calculated prior [23, 71]. If $\beta_k \neq 0$, the method is *implicit* because it depends on information relating to timepoints after the current time. While explicit and implicit methods have the potential to be equally accurate, the latter are more numerically stable and provide greater accuracy for a given order or size of timestep [124]. However, due to the fact that implicit methods depend on information after the current time, all times must be solved simultaneously (often termed *predictor-corrector*) which necessitates the use of a nontrivial iterative approach to generate their solution [35, 282]. As such they are generally more computationally demanding per timestep than explicit methods [35, 282]. Whether an explicit or implicit method will be more computationally efficient for a whole simulation will therefore depend on a number of factors, including: the stiffness of the ODE, the order of the numerical method, and the size of the timesteps.

Multistep methods also require information at the preceding $k-1$ times which can be problematic at the initial timepoint for a method of order $k \neq 1$ as this suggests the need for timepoints before the initial time [23, 71]. In these cases the order of the method used at the initial timestep must be reduced to a one-step method which may come with a significant penalisation in accuracy [71]. The general multistep method in eq. (2.8) assumes that the timestep h is constant [71]. In cases where h is not constant, careful consideration must be given to the values of the coefficients as they will differ from the case with constant h and may lead to the method being inefficient or ill-conditioned [36, 71]. The commonly used *Adams-Bashforth method* [28] and *Adams-Moulton method* [249] are examples of an explicit and implicit method respectively.

Multistage methods integrate over a single timestep $[t_k, t_{k+1}]$ at a time, but subdivide the interval into K stages

$$t_{i,j} = t_i + h_i \rho_j \,, \quad (j = 1, \ldots, K) \,, \tag{2.9}$$

with $0 \leq \rho_0 \leq \rho_1 \leq \ldots \leq \rho_{K-1} \leq 1$ and $h_i = t_i - t_{i-1}$ [71]. The *K-stage Runge-Kutta* family of methods are defined as

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h_i \sum_{j=1}^{K} \beta_j \mathbf{f}_{i,j} \,, \tag{2.10}$$

where

$$\mathbf{f}_{i,j} = \mathbf{f}\left[\left(\mathbf{y}_i + h_i \sum_{l=1}^{K} \alpha_{j,l} \mathbf{f}_{i,l}\right), (t_i + h_i \rho_j)\right] \tag{2.11}$$

for $1 \leq j \leq K$ [71]. For a specific Runge-Kutta method, the constants $\{\rho_j, \beta_j, \alpha_{j,l}\}$

are known and are usually conveniently described in a *Butcher array* [71]

$$\begin{array}{c|ccc} \rho_0 & \alpha_{0,0} & \cdots & \alpha_{0,K} \\ \vdots & \vdots & & \vdots \\ \rho_K & \alpha_{K,0} & \cdots & \alpha_{K,K} \\ \hline & \beta_0 & \cdots & \beta_K \end{array} .$$

Multistage methods are advantageous because the same order of method can be used across the time domain, even at the boundaries where problems with multistep methods arise [36, 71]. They are, however, more computationally demanding because they require intermediate values at the stages to be computed [36, 71, 282]. Runge-Kutta methods are explicit only if $\alpha_{j,l} = 0$ for $l \geq j$, otherwise they are implicit [71]. There are many well-known explicit numerical integration schemes including the *Euler method* ($K = 1$) and the *classic Runge-Kutta method* ($K = 4$), as well as well known implicit schemes including the *Trapezoidal method* ($K = 2$) and the *Hermite-Simpson method* ($K = 3$) [71].

Runge-Kutta methods can also be motivated by using a polynomial $\tilde{\mathbf{y}}(t)$ to approximate a differential equation $\dot{\mathbf{y}}(t) = \mathbf{a}(\mathbf{y}(t), t)$ [143]. A polynomial of degree K (order $K + 1$) can be used to approximate $\dot{\mathbf{y}}(t)$ over each timestep $t_k \leq t \leq t_{k+1}$ such that

$$\tilde{\mathbf{y}}(t) = \alpha_0 + \alpha_1 (t - t_k) + \ldots + \alpha_K (t - t_k)^K . \tag{2.12}$$

The polynomial coefficients $(\alpha_0, \alpha_1, \ldots, \alpha_K)$ are chosen so that the approximation matches the true function at the initial timepoint

$$\tilde{\mathbf{y}}(t_k) = \mathbf{y}(t_k) \tag{2.13}$$

and the gradient matches at each of the stages (eq. (2.9))

$$\frac{\tilde{\mathbf{y}}(t_{k,i})}{dt} = \mathbf{a}(\tilde{\mathbf{y}}(t_{k,i}), t_{k,i}) . \tag{2.14}$$

Equation (2.14) is the *collocation condition* and resulting methods are *collocation methods* [23].

Collocation methods are useful in trajectory optimisation as the collocation condition ensures that the state and state derivatives are approximated correctly [35, 105, 282]. All Runge-Kutta schemes are collocation methods (although the inverse is not true) [71]. With multistep and multistage methods there is a trade-off between the computational demands of a method, and its accuracy and stability. Explicit multistep methods are the easiest to implement and least computationally demanding to propagate. They are also the least accurate and stable, especially when

dealing with stiff or nonlinear differential equations. Conversely, implicit multistage methods, with their requirements for intermediate values at the stages and use of predictor-correctors, appear the most complex. However, due to their numerical stability and accuracy, particularly in higher-order methods, large timesteps can be used while maintaining accuracy [71]. Consequently, they can actually be less computationally demanding [36]. For these reasons, implicit multistage Runge-Kutta methods are typically preferred for solving OCPs [35, 36, 105, 282].

Single Shooting

Single shooting is the simplest technique for formulating a trajectory optimisation problem [35, 282]. Shooting refers to the fact that the approach is very similar to how one would manually shoot at a target:

1. guess the initial conditions;
2. propagate the state from initial to final time;
3. calculate the error in the final time boundary condition; and
4. use a NLP to find decision variables that satisfy the endpoint boundary constraint, repeating the initial steps as required.

Explicit numerical (typically Runge-Kutta) integration schemes are used to propagate the state forward in time [282]. Single shooting formulations lead to problems with very few decision variables (i.e. small dense problems), which makes them practical to implement. However, the final time defect constraint can be highly nonlinear with respect to the initial guess. It can, therefore, be very difficult to force the endpoint constraint to meet the specified solving tolerance [36, 282]. In cases where the OCP in question is simple and a good initialisation is available, single shooting may perform well. Otherwise, achieving converge can be difficult.

Multiple Shooting

Multiple shooting splits the problem into multiple shorter shooting problems and chains them together [282]. Doing this adds an additional continuity constraint at the boundary between adjacent segments. It also demands an initial guess for the state at the beginning of all segments, not just at the initial time. Multiple shooting increases the size of the resulting NLP; additional variables and constraints are

required to describe the state, and enforce continuity, at the segment boundaries respectively. While multiple shooting NLPs are larger than single shooting ones, the NLP derivative matrices are sparse. Furthermore, sparsity increases as the number of multiple shooting segments increases. Sparsity is important for computational performance because the algorithmic complexity associated with sparse matrix operations is significantly less than that of dense matrix operations [36]. Multiple shooting can result in performant computational implementations thanks to the uncoupling of the different segments. All segments are independent and can, therefore, be forward-propagated in parallel [36].

Direct Collocation

In *direct collocation* the state is approximated using a collocation method [35, 282]. This is powerful as the system's dynamical constraints are automatically enforced by the collocation condition [282]. As such, the majority of research into using direct methods to solve OCP has focussed on direct collocation [36, 39, 105, 282]. When implementing direct collocation, the choice of stages and which nodes to collocate are two important considerations [23, 71]. There is significant prior work discussing the advantages and disadvantages of different collocation methods and their most appropriate applications [71, 105, 282]. Choosing evenly-spaced stages results in the lowest possible accuracy [71]. This is because such schemes are subject to *Runge's phenomenon* whereby large oscillations of the polynomial interpolant are present at the domain edges [71]. The most numerically-stable schemes are based on *Gaussian quadrature*. These involve choosing the stages such that they lie at the roots of the orthogonal *Legendre polynomials* [71, 143]. The Legendre polynomials lie on the interval $[-1, 1]$ and different families of methods exist depending on whether the endpoints of this domain are collocated or not [143]. The nth-order Legendre polynomial is denoted by $P_n^*(x)$. Such schemes are described as *orthogonal collocation*.

The three main families of orthogonal collocation methods are [23, 71, 143]:

1. *Legendre-Gauss* (LG) methods, in which the nodes are given by the roots of $P_n^*(x)$ and where the nodes in $(-1, 1)$ (i.e. only the interior nodes) are collocated;

2. *Legendre-Gauss-Radau* (LGR) methods, in which the nodes are given by the roots of $P_{n-1}^*(x) + P_n^*(x)$ as well as the endpoint 1, and where the nodes in $[-1, 1)$ are collocated; and

3. *Legendre-Gauss-Lobatto* (LGL) methods, in which the nodes are given by the roots of $\frac{dP_{n-1}^*(x)}{dx}$ as well as the endpoints -1 and 1, and where the nodes in

$[-1, 1]$ (i.e. all the nodes) are collocated.

LG [33, 179], LGR [106, 121, 188] and LGL collocation methods [38, 100, 104] have all been extensively studied.

LGL presents the benefit that both endpoints are collocated, which automatically enforces the dynamics for the entire duration of the problem [38]. All three collocation methods lead to predictable formulations of the defect constraints, all of which are sparse [105, 122]. This sparsity information can be exploited when forming and solving the NLP [36, 264].

Optimal Control Software Packages

Recent research has focused on formulating and solving increasingly large OCPs efficiently. Many software packages exist for the solving of OCPs via the transcription method, the overwhelming majority of which implement direct methods [282]. Well known examples include: *ACADO* (Automatic Control and Dynamic Optimization) [172], *DIRCOL* [313], *GPOPS* [283], *GPOPS-II* [263], *ICLOCS* [109], *JModelica* [12], *OTIS* (Optimal Trajectories by Implicit Simulation) [322], *PSOPT* [29], *SOCS* (Sparse Optimal Control Software) [40] and *SOS* (Sparse Optimization Suite) [37].

The most widely-used and performant of these is *GPOPS-II*, which is capable of solving multiphase OCPs and implements an LGR collocation method with algorithmic problem scaling and mesh refinement while exploiting problem sparsity. *CGPOPS*, an updated version of *GPOPS-II* implemented in *C++*, was released in November 2020. Both *GPOPS-II* and *CGPOPS* are proprietary software and require a licence to be used [9, 263]. There is currently no easy-to-use, open-source, high-performance, general-purpose software package for solving OCPs that offers a feature set similar to *GPOPS-II* or *CGPOPS*.

2.1.3 Solving Nonlinear Programming Problems

The NLP arising from a transcribed OCP can be mathematically generalised as: determine the vector of decision variables $\mathbf{z} \in \mathbb{R}^{n_z}$ that minimise the objective function

$$J(\mathbf{z}) , \tag{2.15}$$

subject to the constraints

$$\mathbf{g}(\mathbf{z}) = \mathbf{0} \tag{2.16}$$

$$\mathbf{h}(\mathbf{z}) \leq \mathbf{0} \tag{2.17}$$

where the equality constraints $\mathbf{g} \in \mathbb{R}^{n_g}$ and the inequality constraints $\mathbf{h} \in \mathbb{R}^{n_h}$. The vector of all constraints vertically concatenated is

$$\mathbf{c}(\mathbf{z}) \in \mathbb{R}^{n_c} \tag{2.18}$$

where

$$n_c = n_g + n_h \,. \tag{2.19}$$

The NLP defined by eqs. (2.15) to (2.17) can be solved using nonlinear optimisation [131].

Optimisation algorithms fall in to two categories: *heuristic* methods and *gradient* methods [131]. Heuristic methods take a stochastic approach to probe the search space, while gradient methods make an informed decision about search direction based on information from the NLP about the magnitude of the objective function and violation of the constraints [42]. As such, heuristic methods are *global methods* while gradient methods are *local methods*. While the global methods may appear superior because they should not get stuck in local minima, their unguided stochastic search means that they exhibit slow convergence properties and are, therefore, not practical for application to OCPs [35]. Gradient methods are preferred for the nonlinear optimisation in this application [36, 39, 105, 282].

Gradient Methods

The general approach of a gradient method is to: evaluate the problem at the initial guess; decide on a search direction and step distance; step to a new location in the search space; and repeat the process until the termination conditions are met [36, 131]. Gradient methods are either *sequential quadratic programming (SQP) methods* [129] or *interior-point (IP) methods* [43]. SQP methods approach solving the NLP by treating it as an approximate *quadratic programming problem (QP)* at each iteration to determine the search direction [129, 131]. IP methods use a *merit function* to assist with global convergence [36, 43]. A merit function typically linearly combines the object function and the infeasibility of the constraints with some weighting μ (the *barrier parameter*) [36]. The barrier function is then minimised with μ being gradually reduced to 0 from above as the NLP solver progresses. This assists convergence to a global optimum by allowing the initial guess of the solution to contain infeasible dynamics and other constraint violations. Due to the use of the merit function and barrier parameters, IP methods can generally converge to a solution more reliably than SQP methods, especially when a poor initial guess is provided [36]. This can make IP methods preferable when solving the NLP arising from a transcribed OCP as it places less demand on the user to provide an accurate initial guess [36].

Derivative Generation

To determine their search direction and step size, NLP solvers typically require first- and second-order gradient information [35]. The first-order gradient information includes the partial derivatives of the objective function J with respect to the decision variables $\mathbf{z}$ (the *objective gradient*)

$$\mathbf{g} = \frac{\partial J}{\partial \mathbf{z}}, \tag{2.20}$$

and the partial derivatives of the constraints $\mathbf{c}$ with respect to $\mathbf{z}$ (the *constraints Jacobian*)

$$\mathbf{G} = \frac{\partial \mathbf{c}}{\partial \mathbf{z}}\,. \tag{2.21}$$

Some NLP solvers also require the second-order *Lagrangian Hessian* [43]. The Lagrangian is

$$\mathcal{L} = \sigma J + \boldsymbol{\lambda} \cdot \mathbf{c}\,, \tag{2.22}$$

where σ is the *objective factor* and $\boldsymbol{\lambda}$ is the vector of *Lagrange multipliers* (λ_i corresponds to the ith constraint c_i in $\mathbf{c}$). The Lagrangian Hessian is the matrix of second-order partial derivatives of $\mathcal{L}$ with respect to $\mathbf{z}$

$$\mathbf{H} = \frac{\partial^2 \mathcal{L}}{\partial \mathbf{z}^2}\,. \tag{2.23}$$

The Hessian can be expensive to compute. One commonly used computational approach to reduce this cost is to use a quasi-Newton approximation to the Hessian [36]. This can significantly reduce the computation time spent evaluating derivatives for the NLP [282], however using an exact Hessian rather than a quasi-Newton approximation has been shown to significantly reduce the number of NLP iterations required for convergence [43]. Therefore, exact second derivative information should be obtained when solving OCPs by the transcription approach [36]. As each of $\mathbf{g}$, $\mathbf{G}$ and $\mathbf{H}$ may need to be computed one or more times per NLP iteration it is important that they can be evaluated quickly and efficiently when requested by the NLP solver [131]. One important point for consideration is that the NLP generated by the transcribed OCP is large and sparse and therefore exploitation of this sparsity can be used to significantly improve the performance of the computation of any derivate information [36, 264]. Efficient derivative generation for OCPs is the subject of chapter 3 and so is discussed in detail there.

Nonlinear Programming Problem Solvers

Extensive research has been conducted into how best to solve NLPs via SQP and IP methods [129]. Many highly-performant software implementations of the devel-

oped algorithms exist, including the SQP solver *SNOPT* [130], and the IP solvers *Ipopt* [323] and *KNITRO* [72]. The software package *WORHP* [70] offers both SQP and IP solvers. Of these packages, *Ipopt* is the only one with a permission open-source license.

2.1.4 Scaling

While the general approach to solving OCPs works well in theory, there are a number of intricacies related to the approach that can, in practice, result in difficulties in converging on a feasible optimal solution. One such area relates to the scaling of both the OCP and the resulting NLP [36, 131]. Relative scaling of the variables, objective and constraints influences the convergence rate, termination tests and numerical conditioning of the problem [36]. Good relative scaling is, therefore, important [36, 43].

Constructing a reliable method to automatically scale all problems is complex because a scaling approach that works well for one problem may not perform well for all others [36]. Typical approaches try to automatically scale the variables such that they lie on the same domain or allow users to specify their own scaling [37, 263]. General rules for scaling certain components of OCPs have been suggested [36] and implemented with success [29, 38, 263]. However, no author has published details of a comprehensive approach to scaling all aspects of an OCP.

2.1.5 Mesh Refinement

Mesh refinement is the third step in the transcription approach to solving an OCP. It generally involves:

1. a *mesh error calculation* to determine regions of the discretisation domain where the accuracy of the solution does not meet a specified tolerance; and
2. a rediscretisation step where a new mesh is generated, on which a subsequent NLP will be solved [36].

Historically, h-methods, where h refers to the width of a given mesh sections, have been widely used for mesh refinement [36, 348]. In an h-method, a fixed-order quadrature scheme is used uniformly across the whole problem (both for all mesh sections and for all mesh iterations). Convergence is then achieved by subdividing mesh sections between mesh iterations to increase the density of mesh nodes

where accuracy needs improving. Another family of mesh refinement methods are p-methods, where p refers to the order of the interpolating polynomial within a given mesh section [133]. p-methods use a global approach in which the number and width of mesh sections remains the same over successive NLP iterations, while the order of the interpolating polynomials (and therefore number of collocation nodes in a given mesh section) are adjusted in relation to the accuracy of the solved NLP. Easy to implement, h-methods tend to result in requiring fine meshes with high mesh densities, particularly in nonlinear regions of the solution, for convergence to be achieved. p-methods on the other hand can themselves require impractically high-order interpolating polynomials to achieve convergence, particularly if mesh sections of a fixed width contain both nonlinear and linear artefacts.

hp-methods, which combine both h- and p-methods, are possible with orthogonal collocation due to the state and control being discretised by piecewise polynomials. hp-methods typically perform significantly better than either h-methods or p-methods as they can be preferable to reduce the mesh error by mesh section subdivision over increasing polynomial order and visa versa in different circumstances. This generally results in convergence with fewer total nodes (and therefore fewer total NLP decision variables) in the final NLP iteration when an hp-method is used. Many hp-methods have been presented and analysed [36, 82, 83, 219, 220, 262]. These methods typically apply p- and h-methods successively, prioritising the former as increasing the order of the interpolating polynomial generally gives a greater reduction in mesh error than subdividing the mesh section [262]. Comparisons of [7, 82, 83, 219, 220, 262] have shown that there is no clear best-performing mesh refinement algorithm across a range of problems [9, 263].

One area where hp-methods do not perform well is in scenarios where the state contains a discontinuity [7], as is typically the case when the optimal control contains a discontinuity (*bang-bang control*) [7, 36]. Where a discontinuity appears within a phase, a high density of nodes are required to approximate it with low error. In this case it is best to use a mesh refinement algorithm that is capable of detecting discontinuities and recasting the problem with a different number of phases [7]. Such algorithms have, however, been shown to perform poorly when applied to hypersensitive OCPs (a class of OCP that exhibits highly nonlinear dynamics in small regions of their domain), where they can incorrectly identify severe nonlinearities as discontinuities [9, 284].

2.2 Research Objectives

Section 1.3 laid out the objective of developing and critically evaluating a highly performant, easy-to-use, open-source software package to solve general OCPs. This software package should use state-of-the-art theory and methods, and form a foundational element of the *Biomechanics Predictive Simulation Toolkit* (BPST).

From the analysis and review of past work in section 2.1, a number of limitations and constraints associated with the current software provision in this area were identified. To address these, and meet the overall objective above, the following sub-objectives are laid out:

1. investigate current limitations in the numerical solving of OCPs with the purpose of developing new approaches and solutions that can be incorporated into a software package, including:
   (a) the interchangeability of Gaussian-, Radau- and Lobatto-based collocation methods;
   (b) approaches to scaling OCPs; and
   (c) approaches to mesh refinement;
2. incorporate any resulting findings and current state-of-the-art theory into the development of a highly-performant, open-source software package for numerically solving OCPs;
3. ensure that the software is easy-to-use and extensible, such that it is suitable for use by other researchers and practitioners, and as part of the BPST;
4. validate the performance of the software by solving a range of OCPs from the literature that test different aspects of the package's functionality;
5. benchmark the software against the current industry-standard software and evaluate its overall performance; and
6. identify areas for further development and improvement.

2.3 Implicit K-Stage Runge-Kutta Collocation

A large volume of previous work has shown how an OCP can be transcribed using a specific form of collocation. The most prevalent and performant of these are all based on Legendre polynomials and include the LG [33, 179], LGR [106, 121, 188]

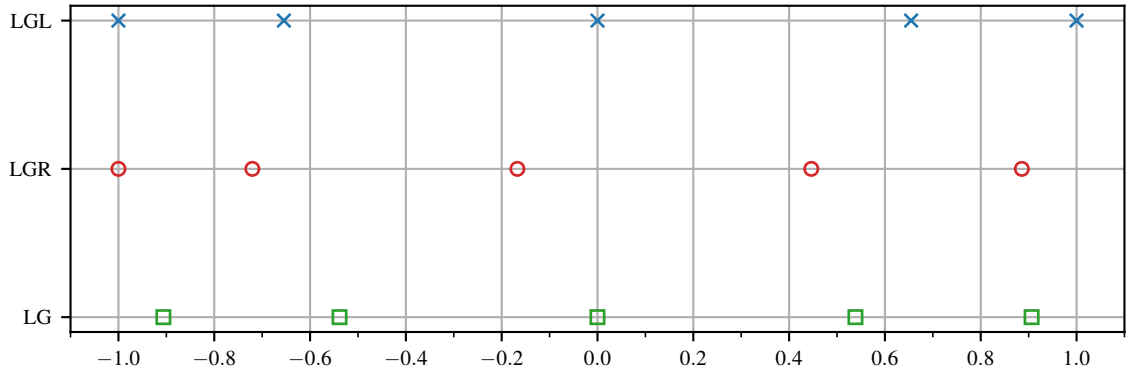


Figure 2.1: Locations of LG (Gaussian), LGR (Radau) and LGL (Lobatto) points.

and LGL [25, 38, 100, 104] collocation methods. These three methods collocate the discretised problem across multiple intervals at the Gaussian, Radau and Lobatto points respectively (fig. 2.1). There are conflicting opinions in the literature about which of LG, LGR and LGL collocation is the most numerically stable and offers the best convergence properties [105, 122]. Therefore, there is a motivation to allow the user of any software package for solving OCPs to select the collocation method that best suits their problem.

A unified framework for formulating for LG, LGR and LGL collocation has been presented before [122]. However, this approach requires the differentiation of a sum of Lagrange polynomials to produce a differentiation matrix, followed by its inversion to produce the constants for integral-form collocation. Three slightly different algorithms are required to formulate the three collocation schemes, which reduces the framework's ease of implementation.

This section develops a method that allows the transcription of the OCP defined by eqs. (2.1) to (2.7) to an NLP by using multiple-interval integral-form variable-order collocation based on K-stage Runge-Kutta methods with a generalised choice of stages. It shows that LG, LGR and LGL collocation methods can all be formulated as K-stage Runge-Kutta methods and, therefore, an appropriate Butcher table can be used to derive the coefficients corresponding to the collocation scheme. The approach allows for any one of LG, LGR and LGL collocation to be used within the same transcription framework, simply by adjusting the function used for computing the quadrature nodes.

2.3.1 Discretising Ordinary Differential Equations

The dynamical equations in an OCP can be characterised as a set of first-order ODEs in terms of the state of the system. Solving the OCP will involve determin-

ing the value of the state such that it is dynamically consistent across the duration of the problem (in addition to meeting any constraints on the state's value at the endpoints). Solution by the transcription method, therefore, involves discretising the problem in a way that the NLP contains constraints that enforce the OCP's dynamics at each of the discretised state values. Each OCP phase may be discretised such that it will consist of a number of mesh sections. However, to simplify the following discussion only a single mesh section (from time t_i to t_{i+1}) will be considered. The following results can then be generalised to a phase consisting of multiple mesh sections.

Consider the simple ODE

$$\dot{y}\left(t\right) = f\left(y\left(t\right), t\right) . \tag{2.24}$$

The value of $y\left(t_{i+1}\right)$ relative to $y\left(t_i\right)$ can be found by integrating along the timestep $h_i = t_{i+1} - t_i$

$$y\left(t_{i+1}\right) = y\left(t_i\right) + \int_{t_i}^{t_{i+1}} \dot{y}\left(t\right) dt . \tag{2.25}$$

The integral in eq. (2.25) can be approximated using a numerical method of integration (section 2.1.2). Further discretisation of the timestep between t_i and t_{i+1} so that is also contains internal stages is useful. If the timestep is discretised such that it contains ν total timepoints ($\nu - 2$ internal stages) then let these timepoints be denoted as

$$t_{ij} = t_i + \rho_j h_i , \quad \left(j = 1, \ldots, \nu\right) , \tag{2.26}$$

where

$$0 = \rho_1 < \rho_2 < \ldots < \rho_{\nu-1} < \rho_\nu = 1 \tag{2.27}$$

and $t_{i1} = t_i$ and $t_{i\nu} = t_{i+1}$. To simplify the nomenclature, the terminology

$$y_{ij} = y\left(t_{ij}\right) , \quad f_{ij} = f\left(y_{ij}, t_{ij}\right) = \dot{y}\left(t_{ij}\right)$$

is also introduced.

2.3.2 Lagrange Polynomials

Integral-form LG [32], LGR [121] and LGL [25] have all previously been derived using Lagrange interpolating polynomials to approximate the integral in eq. (2.25). Lagrange polynomials are defined as

$$L\left(\rho\right) \coloneqq \sum_{j=\iota}^{\kappa} y_{ij} \ell_j\left(\rho\right) , \tag{2.28}$$

where the Lagrange basis polynomials are

$$\ell_j(\rho) = \prod_{m=\iota, m\neq j}^{\kappa} \frac{\rho - \rho_m}{\rho_j - \rho_m}, \quad (\iota \leq j \leq \kappa) \ . \tag{2.29}$$

They are useful for numerical interpolation as they represent the polynomial of lowest degree that can pass through a unique set of points. Some references (e.g. [32]) have presented the basis polynomials in their Barycentric form,

$$\ell_j(\rho) = \frac{\ell(\rho)}{\ell'(\rho_j)(\rho - \rho_j)}, \tag{2.30}$$

where

$$\ell(\rho) = \prod_{m=\iota}^{\kappa} (\rho - \rho_m) \ , \tag{2.31}$$

but these have been converted to the form in eqs. (2.28) and (2.29) for consistency in this thesis.

2.3.3 Legendre Polynomials

Legendre polynomials are a set of orthogonal polynomials with useful mathematical properties that make them highly suited to applications in numerical integration [71, 143]. The Legendre polynomials on the interval $[0, 1]$ are

$$\begin{aligned} P_0^*(x) &= 1, \\ P_1^*(x) &= 2x - 1, \\ P_2^*(x) &= 6x^2 - 6x + 1, \\ P_3^*(x) &= 20x^3 - 30x^2 + 12x - 1, \\ &\vdots \qquad\quad \vdots \end{aligned}$$

and have the properties [71] that

$$\int_0^1 P_m^*(x) P_n^*(x)\, dx = 0, \quad (m \neq n) \ , \tag{2.32}$$

$$P_n^*(1) = 1, \quad (n = 0, 1, 2, \ldots) \ . \tag{2.33}$$

2.3.4 Derivation and Equivalence

Consider approximating the state equation in eq. (2.24) using a Lagrange interpolating polynomial collocated at K points [25, 32, 121]. This yields an approximation to the derivative

$$\dot{y}(t) = f(y(t), t) \approx L(\rho) \tag{2.34}$$

with the property that

$$L\left(\rho_j\right) = f_{ij}\,, \quad (\iota \leq j \leq \kappa)\ . \tag{2.35}$$

From eq. (2.34) and the definition of a Lagrange polynomial (eq. (2.28)), an integral involving $f\left(y\left(t\right),t\right)$ can be approximated as

$$\begin{aligned}\int_{t_i}^{t_{ij}} f\left(y\left(t\right),t\right)dt \approx \int_{t_i}^{t_{ij}} L\left(\rho\right)dt &= h_i \int_0^{\rho_j} L\left(\rho\right)d\rho \\ &= h_i \int_0^{\rho_j} \sum_{j=\iota}^{\kappa} f_{ij}\ell_j\left(\rho\right)d\rho \\ &= h_i \sum_{j=\iota}^{\kappa} f_{ij} \int_0^{\rho_j} \ell_j\left(\rho\right)d\rho\,.\end{aligned} \tag{2.36}$$

By defining

$$\alpha_{ij} = \int_0^{\rho_j} \ell_j\left(\rho\right)d\rho\,, \tag{2.37}$$

estimates for y_{ij} can be made by approximating integrals using the summation from eq. (2.36)

$$y_{ij} \approx y_i + h_i \sum_{\iota}^{\kappa} \alpha_{ij} f_{ij}\,. \tag{2.38}$$

Also, by defining

$$\beta_j = \int_0^{1} \ell_j\left(\rho\right)d\rho\,, \tag{2.39}$$

an estimate for the value y_{i+1} can be similarly approximated as

$$y_{i+1} \approx y_i + h_i \sum_{j=\iota}^{\kappa} \beta_j f_{ij}\,. \tag{2.40}$$

In the NLP, the exact values of y_{ij} and f_{ij} are not known. However, if y_{ij} for $(j = 1,\ldots,\nu)$ are treated as decision variables then values will be available for each and f_{ij} can be directly calculated. Define Y_{ij} as the NLP decision variable at t_{ij} such that

$$\mathbf{Y} = \begin{bmatrix} Y_i & Y_{i2} & \ldots & Y_{i(\nu-1)} & Y_{i+1} \end{bmatrix}^T \tag{2.41}$$

and $F_{ij} = f\left(Y_{ij}, t_{ij}\right)$ such that

$$\mathbf{F} = \begin{bmatrix} F_i & F_{i2} & \ldots & F_{i(\nu-1)} & F_{i+1} \end{bmatrix}^T\,. \tag{2.42}$$

From eq. (2.38) a set of K equality constraints which enforce the system dynamics at the K internal stages can be produced in the form

$$0 = Y_i - Y_{ij} + h_i \sum_{j=\iota}^{\kappa} \alpha_{ij} f_{ij}\,, \quad (j = \iota,\ldots,\kappa)\ . \tag{2.43}$$

Additionally, a further equality constraint, which enforces the system dynamics across the integral of the timestep,

$$0 = Y_i - Y_{i+1} + h_i \sum_{j=\iota}^{\kappa} \beta_j f_{ij} \tag{2.44}$$

can be formed. Combining eqs. (2.43) and (2.44) for all stages and converting the equations in to a linear matrix system yields

$$\mathbf{0} = \mathbf{E}\mathbf{Y} + h_i \mathbf{A}\mathbf{F}\,, \tag{2.45}$$

where $\mathbf{E}$ is a $(\nu - 1) \times \nu$ matrix with ones in the first column and negative ones on the first superdiagonal

$$\mathbf{E} = \begin{bmatrix} 1 & -1 & 0 & \cdots & 0 \\ 1 & 0 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 0 & 0 & \cdots & -1 \end{bmatrix} \tag{2.46}$$

and $\mathbf{A}$ is the $(\nu - 1) \times \nu$ integration matrix populated by the stage and step weights eqs. (2.37) and (2.39).

The previously developed LG [32, 122], LGR [121, 122] and LGL [25, 122] collocation methods populate the integration matrix using integrals of the Lagrange interpolating polynomials. However, the derivation of these methods can be motivated another way due to the fact that collocation methods can also be thought of as Runge-Kutta methods [71].

Consider instead approximating the integral term in eq. (2.25) using an implicit K-stage Runge-Kutta method. The stages ρ_i, quadrature weights β_i and stage weights α_{ij} associated with the method can be derived from analysis of Legendre polynomials such that the error is minimised and the stability is maximised [71]. For Runge-Kutta methods based on Gaussian quadrature, the set of Gaussian points are used for ρ_i. For methods based on Radau and Lobatto quadrature, ρ_i is populated by the set of Radau and Lobatto points respecively.

To maximise the order of the method based on the chosen points, the quadrature weights β_i for $(i = \iota, \ldots, \kappa)$ should be chosen to satisfy the $B(\eta)$ condition [71]

$$B(\eta) : \sum_{j=\iota}^{\kappa} \beta_j \rho_j^{s-1} = \frac{1}{s}\,, \quad (s = 1, \ldots, \eta)\,. \tag{2.47}$$

Additionally, for the stage weights α_{ij} for $(i, j = \iota, \ldots, \kappa)$ should be chosen to satisfy the $C(\zeta)$ condition [71]

$$C(\zeta) : \sum_{j=\iota}^{\kappa} \alpha_{ij} \rho_j^{s-1} = \frac{\rho_i^s}{s}\,, \quad (i = \iota, \ldots, \kappa)\,, \quad (s = 1, \ldots, \zeta)\,. \tag{2.48}$$

Solving the linear system in eq. (2.47) yields the quadrature weights β_i, while solving the K linear systems in eq. (2.48) yields the stage weights (with the ith row given by solving the ith linear system).

The equivalence of a K-stage Runge-Kutta method with the coefficients β_j defined by the $B(\eta)$ condition and the coefficients α_{ij} defined by the $C(\zeta)$ to a collocation method defined using Lagrange polynomials has been proven in [138, 330]. Specifically, for a given set of points, the coefficients α_{ij} defined by eqs. (2.37) and (2.48) are equivalent and β_j defined by eqs. (2.39) and (2.47) are also equivalent. As a result, the defect constraints on the dynamics can be formulated in the form given by eq. (2.45), with $\mathbf{A}$ populated using eqs. (2.47) and (2.48).

2.3.5 Legendre-Gauss Collocation

In LG collocation [32], interpolation is centred around the $K = \nu - 2$ Gaussian points (fig. 2.1). From [4], the Gaussian points are the zeros of

$$P_K^* \,. \tag{2.49}$$

In relation to the discretisation defined in eq. (2.27), the Gaussian points correspond to the internal stages ρ_i for $(i = 2, \ldots, \nu - 1)$, with $\rho_1 = 0$ and $\rho_\nu = 1$ being non-collocated endpoints. As such, $\iota = 2$ and $\kappa = \nu - 1 = K + 1$ in relation to the sums and products involved in the polynomials' definitions in eqs. (2.28) and (2.29).

Using eq. (2.45), the set of defect constraints that enforce the system's dynamics when discretised at the Gaussian points can be produced by

$$\mathbf{0} = \mathbf{E}\mathbf{Y} + h_i \mathbf{A}^{\mathrm{LG}} \mathbf{F} \,, \tag{2.50}$$

where $\mathbf{Y}$ (eq. (2.41)), $\mathbf{F}$ (eq. (2.42)) and $\mathbf{E}$ (eq. (2.46)) are as before. As $\iota = 2$ and $\kappa = \nu - 1 = K + 1$, $\mathbf{A}^{\mathrm{LG}}$ is the $(\nu - 1) \times \nu$ integration matrix with $A_{(i-1)j} = \alpha_{(i-1)j}$ for $(i, j = 2, \ldots, \nu - 1)$ and $A_{(\nu-1)j} = \beta_j$ for $(j = 2, \ldots, \nu - 1)$ with remaining elements as zeros

$$\mathbf{A}^{\mathrm{LG}} = \begin{bmatrix} 0 & \alpha_{12} & \alpha_{13} & \cdots & \alpha_{1(\nu-1)} & 0 \\ 0 & \alpha_{22} & \alpha_{23} & \cdots & \alpha_{2(\nu-1)} & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & \alpha_{(\nu-2)2} & \alpha_{(\nu-2)3} & \cdots & \alpha_{(\nu-2)(\nu-1)} & 0 \\ 0 & \beta_2 & \beta_3 & \cdots & \beta_{(\nu-1)} & 0 \end{bmatrix} . \tag{2.51}$$

The elements of $\mathbf{A}^{\mathrm{LG}}$ can be populated using an appropriately sized Gaussian Butcher tableau [32]. Examples for $k = 1, 2, 3$ are given in table 2.1. The integration matrices corresponding to the Butcher tableaus in table 2.1 are shown in table 2.2.

$$\begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array}$$

(a) $k = 1$

$$\begin{array}{c|cc} \frac{1}{2} - \frac{\sqrt{3}}{6} & \frac{1}{4} & \frac{1}{4} - \frac{\sqrt{3}}{6} \\ \frac{1}{2} + \frac{\sqrt{3}}{6} & \frac{1}{4} + \frac{\sqrt{3}}{6} & \frac{1}{4} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

(b) $k = 2$

$$\begin{array}{c|ccc} \frac{1}{2} - \frac{\sqrt{15}}{10} & \frac{5}{36} & \frac{2}{9} - \frac{\sqrt{15}}{15} & \frac{5}{36} - \frac{\sqrt{15}}{30} \\ \frac{1}{2} & \frac{5}{36} + \frac{\sqrt{15}}{24} & \frac{2}{9} & \frac{5}{36} - \frac{\sqrt{15}}{24} \\ \frac{1}{2} + \frac{\sqrt{15}}{10} & \frac{5}{36} + \frac{\sqrt{15}}{30} & \frac{2}{9} + \frac{\sqrt{15}}{15} & \frac{5}{36} \\ \hline & \frac{5}{18} & \frac{4}{9} & \frac{5}{18} \end{array}$$

(c) $k = 3$

Table 2.1: Butcher tableaus for Gaussian methods.

$$\mathbf{A}^{\mathrm{LG}}_{2\times 3} = \begin{bmatrix} 0 & \frac{1}{2} & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

(a) $k = 1$, $\nu = 3$

$$\mathbf{A}^{\mathrm{LG}}_{3\times 4} = \begin{bmatrix} 0 & \frac{1}{4} & \frac{1}{4} - \frac{\sqrt{3}}{6} & 0 \\ 0 & \frac{1}{4} + \frac{\sqrt{3}}{6} & \frac{1}{4} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \end{bmatrix}$$

(b) $k = 2$, $\nu = 4$

$$\mathbf{A}^{\mathrm{LG}}_{4\times 5} = \begin{bmatrix} 0 & \frac{5}{36} & \frac{2}{9} - \frac{\sqrt{15}}{15} & \frac{5}{36} - \frac{\sqrt{15}}{30} & 0 \\ 0 & \frac{5}{36} + \frac{\sqrt{15}}{24} & \frac{2}{9} & \frac{5}{36} - \frac{\sqrt{15}}{24} & 0 \\ 0 & \frac{5}{36} + \frac{\sqrt{15}}{30} & \frac{2}{9} + \frac{\sqrt{15}}{15} & \frac{5}{36} & 0 \\ 0 & \frac{5}{18} & \frac{4}{9} & \frac{5}{18} & 0 \end{bmatrix}$$

(c) $k = 3$, $\nu = 5$

Table 2.2: Integration matrices for Gaussian methods.

2.3.6 Legendre-Gauss-Radau Collocation

In LGR collocation [121], a Lagrange polynomial is used to interpolate the $K = \nu - 1$ Radau points (fig. 2.1). From [4], the set of K Radau points ρ^{LGR} are the zeros of

$$P^*_{K-1}(x) + P^*_K(x) \tag{2.52}$$

and correspond to the stages ρ_i for $(i = 1, \ldots, \nu - 1)$, with ρ_ν again being non-collocated, as defined in eq. (2.27). The set of Radau points, therefore, lie on the half-open domain $[0, 1)$. For LGR collocation, the general definitions for a Lagrange polynomial and the Lagrange basis polynomials (eqs. (2.28) and (2.29)) are parameterised by $\iota = 1$ and $\kappa = \nu - 1 = K$.

Using eq. (2.45), the set of defect constraints that enforce the system's dynamics when discretised at the Radau points can be produced by

$$\mathbf{0} = \mathbf{E}\mathbf{Y} + h_i \mathbf{A}^{\mathrm{LGR}} \mathbf{F}\,, \tag{2.53}$$

where $\mathbf{Y}$ (eq. (2.41)), $\mathbf{F}$ (eq. (2.42)) and $\mathbf{E}$ (eq. (2.46)) are as before. As $\iota = 1$ and $\kappa = \nu - 1 = K + 1$, $\mathbf{A}^{\mathrm{LGR}}$ is the $(\nu - 1) \times \nu$ integration matrix with $A_{ij} = \alpha_{ij}$ for $(i, j = 1, \ldots, \nu - 1)$ and $A_{\nu j} = \beta_j$ for $(j = 1, \ldots, \nu - 1)$ with remaining elements as zeros

$$\mathbf{A}^{\mathrm{LGR}} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1(\nu-1)} & 0 \\ \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2(\nu-1)} & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ \alpha_{(\nu-1)1} & \alpha_{(\nu-1)2} & \cdots & \alpha_{(\nu-1)(\nu-1)} & 0 \\ \beta_1 & \beta_2 & \cdots & \beta_{(\nu-1)} & 0 \end{bmatrix}. \tag{2.54}$$

The elements of $\mathbf{A}^{\mathrm{LGR}}$ can be populated using an appropriately sized Radau I Butcher tableau [32]. Examples for $k = 1, 2, 3$ are given in table 2.3. The integration matrices corresponding to the Butcher tableaus in table 2.3 are shown in table 2.4.

2.3.7 Legendre-Gauss-Lobatto Collocation

In LGL collocation [25] the set of $K = \nu$ Lobatto points (fig. 2.1) are used. From [4], the Lobatto points can be generated as the zeros of

$$\frac{dP^*_{n-1}(x)}{dx} \tag{2.55}$$

(which correspond to the internal stages ρ_i for $(i = 2, \ldots, \nu - 1)$) as well as the endpoints $\rho_1 = 0$ and $\rho_\nu = 1$. In LGL collocation methods, all of the discretisation

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

(a) $k = 1$

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{2}{3} & \frac{1}{3} & \frac{1}{3} \\ \hline & \frac{1}{4} & \frac{3}{4} \end{array}$$

(b) $k = 2$

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{6-\sqrt{6}}{10} & \frac{9+\sqrt{6}}{75} & \frac{24+\sqrt{6}}{120} & \frac{168-73\sqrt{6}}{600} \\ \frac{6+\sqrt{6}}{10} & \frac{9-\sqrt{6}}{75} & \frac{168+73\sqrt{6}}{600} & \frac{24-\sqrt{6}}{120} \\ \hline & \frac{1}{9} & \frac{16+\sqrt{6}}{36} & \frac{16-\sqrt{6}}{36} \end{array}$$

(c) $k = 3$

Table 2.3: Butcher tableaus for Radau I methods.

$$\mathbf{A}^{\mathrm{LGR}}_{1\times 2} = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

(a) $k = 1$, $\nu = 2$

$$\mathbf{A}^{\mathrm{LGR}}_{2\times 3} = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & 0 \\ \frac{1}{4} & \frac{3}{4} & 0 \end{bmatrix}$$

(b) $k = 2$, $\nu = 3$

$$\mathbf{A}^{\mathrm{LGR}}_{3\times 4} = \begin{bmatrix} \frac{9+\sqrt{6}}{75} & \frac{24+\sqrt{6}}{120} & \frac{168-73\sqrt{6}}{600} & 0 \\ \frac{9-\sqrt{6}}{75} & \frac{168+73\sqrt{6}}{600} & \frac{24-\sqrt{6}}{120} & 0 \\ \frac{1}{9} & \frac{16+\sqrt{6}}{36} & \frac{16-\sqrt{6}}{36} & 0 \end{bmatrix}$$

(c) $k = 3$, $\nu = 4$

Table 2.4: Integration matrices for Radau I methods.

0	0	0
1	$\frac{1}{2}$	$\frac{1}{2}$
	$\frac{1}{2}$	$\frac{1}{2}$

(a) $k = 2$

0	0	0	0
$\frac{1}{2}$	$\frac{5}{24}$	$\frac{1}{3}$	$-\frac{1}{24}$
1	$\frac{1}{6}$	$\frac{2}{3}$	$\frac{1}{6}$
	$\frac{1}{6}$	$\frac{2}{3}$	$\frac{1}{6}$

(b) $k = 3$

0	0	0	0	0
$\frac{5-\sqrt{5}}{10}$	$\frac{11+\sqrt{5}}{120}$	$\frac{25-\sqrt{5}}{120}$	$\frac{25-13\sqrt{5}}{120}$	$\frac{-1+\sqrt{5}}{120}$
$\frac{5+\sqrt{5}}{10}$	$\frac{11-\sqrt{5}}{120}$	$\frac{25+13\sqrt{5}}{120}$	$\frac{25+\sqrt{5}}{120}$	$\frac{-1-\sqrt{5}}{120}$
1	$\frac{1}{12}$	$\frac{5}{12}$	$\frac{5}{12}$	$\frac{1}{12}$
	$\frac{1}{12}$	$\frac{5}{12}$	$\frac{5}{12}$	$\frac{1}{12}$

(c) $k = 4$

Table 2.5: Butcher tableaus for Lobatto IIIA methods.

points are collocation points such that the Lobatto points lie on the closed domain $[0, 1]$. Because all stages will be considered collocation points in LGL collocation, the values $\iota = 1$ and $\kappa = \nu = K$ correspond to eqs. (2.28) and (2.29).

Using eq. (2.45), the set of defect constraints that enforce the system's dynamics when discretised at the Lobatto points can be produced by

$$\mathbf{0} = \mathbf{E}\mathbf{Y} + h_i \mathbf{A}^{\mathrm{LGR}} \mathbf{F}\,, \tag{2.56}$$

where $\mathbf{Y}$ (eq. (2.41)), $\mathbf{F}$ (eq. (2.42)) and $\mathbf{E}$ (eq. (2.46)) are as before. As $\iota = 1$ and $\kappa = \nu - 1 = K + 1$, $\mathbf{A}^{\mathrm{LGL}}$ is the $(\nu - 1) \times \nu$ integration matrix with $A_{ij} = \alpha_{(i+1)j}$ for $(i, j = 1, \ldots, \nu)$

$$\mathbf{A}^{\mathrm{LGL}} = \begin{bmatrix} \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2\nu} \\ \alpha_{31} & \alpha_{32} & \cdots & \alpha_{3\nu} \\ \vdots & \vdots & & \vdots \\ \alpha_{(\nu-1)1} & \alpha_{(\nu-1)2} & \cdots & \alpha_{(\nu-1)\nu} \\ \alpha_{\nu 1} & \alpha_{\nu 2} & \cdots & \alpha_{\nu\nu} \end{bmatrix} = \begin{bmatrix} \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2\nu} \\ \alpha_{31} & \alpha_{32} & \cdots & \alpha_{3\nu} \\ \vdots & \vdots & & \vdots \\ \alpha_{(\nu-1)1} & \alpha_{(\nu-1)2} & \cdots & \alpha_{(\nu-1)\nu} \\ \beta_1 & \beta_2 & \cdots & \beta_\nu \end{bmatrix}. \tag{2.57}$$

The bottom row of $\mathbf{A}^{\mathrm{LGL}}$ can be equivalently thought of as $A_{i\nu} = \beta_j$ or $A_{i\nu} = \alpha_{\nu j}$ for $(j = 1, \ldots, \nu)$ (eq. (2.57)).

$$\mathbf{A}^{\mathrm{LGL}}_{1\times 2} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

(a) $k = 2$, $\nu = 2$

$$\mathbf{A}^{\mathrm{LGL}}_{2\times 3} = \begin{bmatrix} \frac{5}{24} & \frac{1}{3} & -\frac{1}{24} \\ \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{bmatrix}$$

(b) $k = 3$, $\nu = 3$

$$\mathbf{A}^{\mathrm{LGL}}_{3\times 4} = \begin{bmatrix} \frac{11+\sqrt{5}}{120} & \frac{25-\sqrt{5}}{120} & \frac{25-13\sqrt{5}}{120} & \frac{-1+\sqrt{5}}{120} \\ \frac{11-\sqrt{5}}{120} & \frac{25+13\sqrt{5}}{120} & \frac{25+\sqrt{5}}{120} & \frac{-1-\sqrt{5}}{120} \\ \frac{1}{12} & \frac{5}{12} & \frac{5}{12} & \frac{1}{12} \end{bmatrix}$$

(c) $k = 4$, $\nu = 4$

Table 2.6: Integration matrices for Lobatto IIIA methods.

The elements of $\mathbf{A}^{\mathrm{LGL}}$ can be populated using an appropriately sized Lobatto IIIA Butcher tableau [32]. Examples for $k = 2, 3, 4$ are given in table 2.5. The integration matrices corresponding to the Butcher tableaus in table 2.5 are shown in table 2.6.

2.3.8 Algorithmic Implementation

Section 2.3.4 showed that LG, LGR and LGL collocation methods can be grouped under a unified framework if implemented using eq. (2.45), with $\mathbf{A}$ being populated appropriately based on the chosen scheme. This unified framework lends itself to algorithmic implementation such that a specific collocation method can be employed just by choosing to use one of the particular sets of quadrature points. Furthermore, the approach of using the Butcher tableaus to populate the integration matrices is powerful as it will be shown that the Butcher tableau for any Runge-Kutta method can be efficiently generated by solving a series linear systems derived from the $B(\eta)$ and $C(\zeta)$ conditions.

In order to construct a generalised algorithm that supports formulation of LG, LGR and LGL collocation methods, the Butcher tableau for a specific method needs to be constructed in a way that is independent of the method. For all three methods it has been shown that the quadrature points can be found as roots of the appropriate Legendre polynomials, the quadrature weights can be found by satisfying the $B(\eta)$ condition and the stage weights can be found by satisfying the $C(\eta)$ condition. This algorithmic implementation consists of the following steps:

1. discretise the ODE at a given set of quadrature points with either no endpoints collocated (Gaussian points), the initial endpoint collocated (Radau points) or both endpoints collocated (Lobatto points);

2. generate the Butcher tableau for the method by solving the set of linear systems resulting from satisfying the $B(\eta)$ and $C(\zeta)$ conditions at the chosen quadrature points;

3. construct the integration matrix $\mathbf{A}$ using the necessary elements from the Butcher tableau; and

4. generate the set of defect constraints using eq. (2.45).

Generation of Quadrature Points

Quadrature points can be determined using any polynomial root finding technique. This is typically done computationally by building the Frobenius companion matrix for the polynomial in question and finding its eigenvalues [312]. All established mathematical computing languages will provide functions or classes to facilitate this with ease. For example, the *Python* package *SciPy* [321] includes a module containing classes for Legendre polynomials which have a method for finding their roots.

Generation of Butcher Tableaus

The quadrature weights are determined by satisfying the $B(\eta)$ condition. Let the row vector of collocation points for a single step be denoted as

$$\boldsymbol{\rho} = \begin{bmatrix} \rho_1 & \rho_2 & \cdots & \rho_K \end{bmatrix}. \tag{2.58}$$

For a method with K collocation points, satisfying the $B(\eta)$ condition will result in a series of K simultaneous equations with the unknowns β_j in terms of the previous calculated ρ_j for $j = 1, \ldots, K$. A linear square matrix system can be formed by constructing the $B(\eta)$ conditions for $j = 1, \ldots, K$, expanding the summation terms and vertically concatenating the resulting equations. In this case, the *right hand side* (RHS) is a column vector with K rows denoted as $\mathbf{b}$ and with its ith entry being the ith value in the Harmonic series

$$\mathbf{b} = \begin{bmatrix} 1 & \frac{1}{2} & \cdots & \frac{1}{K} \end{bmatrix}^T. \tag{2.59}$$

The *left hand side* (LHS) square matrix

$$\mathbf{P} = \begin{bmatrix} \boldsymbol{\rho}^0 \\ \boldsymbol{\rho}^1 \\ \vdots \\ \boldsymbol{\rho}^{K-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ \rho_1 & \rho_2 & \cdots & \rho_K \\ \vdots & \vdots & \ddots & \vdots \\ \rho_1^{K-1} & \rho_2^{K-1} & \cdots & \rho_K^{K-1} \end{bmatrix} . \tag{2.60}$$

The LHS column vector of unknowns corresponding to the K quadrature weights is denoted as

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_1 & \beta_2 & \dots & \beta_K \end{bmatrix}^T . \tag{2.61}$$

This linear system $\mathbf{P}\boldsymbol{\beta} = \mathbf{b}$ can again be solved by established methods such as LU-decomposition [312]. Facilities for this are provided by all mathematical computing languages (e.g. *Python*'s *SciPy*).

Satisfying the $C(\eta)$ condition to evaluate α_{ij} for $(i, j = 1, \dots, K)$ involves finding K^2 unknowns and as such involves solving K linear systems, each with K unknowns. Each row of the stage weights in the Butcher tableau can be denoted by a column vector with the ith being

$$\boldsymbol{\alpha}_i = \begin{bmatrix} \alpha_{i1} & \alpha_{i2} & \cdots & \alpha_{iK} \end{bmatrix}^T \quad (i = 1, \dots, K) . \tag{2.62}$$

Motivated by satisfying the $C(\eta)$ condition, $\boldsymbol{\alpha}_i$ can be found from solving the ith linear system in the form $\mathbf{P}\boldsymbol{\alpha}_i = \mathbf{c}_i$ where the LHS matrix $\mathbf{P}$ is the same as defined in eq. (2.60) and the ith column vector $\mathbf{c}_i$ is defined as

$$\mathbf{c}_i = \begin{bmatrix} \rho_i & \frac{\rho_i}{2} \dots \frac{\rho_i^K}{K} \end{bmatrix}^T \quad (i = 1, \dots, K) . \tag{2.63}$$

Construction of the Integration Matrix

The integration matrix is constructed from the populated Butcher tableau using the following steps:

1. Ignore any rows of stage weights corresponding to the quadrature point 0, as the collocation method does not form a defect constraint between the stage initial point and itself.

2. Ignore any rows of stage weights corresponding to the quadrature point 1. This is only the case for Lobatto IIIA quadrature and is necessary as it has been shown that the stage equation corresponding to the collocated point at 1 and the quadrature equation for this family of schemes are equivalent (eq. (2.57)).

3. Vertically concatenate the remaining stage weights and the quadrature weights to form a $(\nu - 1) \times K$ rectangular matrix of weights.

4. Populate $\mathbf{A}$ using the weights, such that its upper-right entry lies in the first row and the column corresponding to the first stage, which is also a collocation point. For explicit clarity, this means that for the Gaussian-based method the first and last columns are zero-padded, for the Radau-based method the last column in zero-padded, and for the Lobatto-based method no zero padding is necessary.

Formation of Defect Constraints

The defect constraints are formed using the quadrature-specific integration matrix and eq. (2.45).

2.4 Sparse Nonlinear Programming Problem Formulation

Using the generalised framework for implicit integral-form K-stage Runge-Kutta collocation method detailed in section 2.3, a multiphase continuous-time OCP can be transcribed to an NLP that can be solved using a computational NLP solver. This NLP is to determine the vector of decision variables $\mathbf{X}$ that minimises the objective function

$$\mathcal{J} = \Phi(\mathbf{X}) \tag{2.64}$$

subject to the constraints

$$\mathbf{C}_{\min} \leq \mathbf{C}(\mathbf{X}) \leq \mathbf{C}_{\max} \tag{2.65}$$

and where the decision variables are bounded as

$$\mathbf{X}_{\min} \leq \mathbf{X} \leq \mathbf{X}_{\max}. \tag{2.66}$$

This constrained NLP is of a general form such that it can be solved by any general NLP solver.

2.4.1 Decision Variables

The column vector of decision variables is constructed as the concatenation of the phase-specific decision variables $\mathbf{X}^{(p)}$ for $p \in [1, \ldots, P]$ and the phase-independent

static parameter variables $\mathbf{s}$, as

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}^{(1)} \\ \vdots \\ \mathbf{X}^{(P)} \\ \mathbf{s} \end{bmatrix} . \tag{2.67}$$

The phase-specific decision variables for phase p are further constructed as

$$\mathbf{X}^{(p)} = \begin{bmatrix} \mathbf{Y}_1^{(p)} \\ \vdots \\ \mathbf{Y}_{n_y^{(p)}}^{(p)} \\ \mathbf{U}_1^{(p)} \\ \vdots \\ \mathbf{U}_{n_u^{(p)}}^{(p)} \\ \mathbf{q}^{(p)} \\ {t_0}^{(p)} \\ {t_F}^{(p)} \end{bmatrix} . \tag{2.68}$$

$\mathbf{Y}_i^{(p)}$ for $\left(i = 1, \ldots, n_y^{(p)}\right)$ denotes a column vector of the ith state variable from the OCP discretised according to the generalised K-stage Runge-Kutta collocation method outlined in section 2.3. Similarly, $\mathbf{U}_j^{(p)}$ for $\left(j = 1, \ldots, n_u^{(p)}\right)$ denotes a column vector of the j th control variable from the OCP according to the same discretisation. $\mathbf{q}^{(p)}$ is a column vector containing the ${n_q}^{(p)}$ integral variables from phase p in the form

$$\mathbf{q}^{(p)} = \begin{bmatrix} q_1^{(p)} \\ \vdots \\ q_{n_q^{(p)}}^{(p)} \end{bmatrix} , \tag{2.69}$$

while ${t_0}^{(p)}$ and ${t_F}^{(p)}$ are the initial and final times of phase p respectively. The phase-independent decision variables $\mathbf{s}$ directly correspond to the n_s static parameter variables from the OCP and are given as

$$\mathbf{s} = \begin{bmatrix} s_1 \\ \vdots \\ s_{n_s} \end{bmatrix} . \tag{2.70}$$

Note that the size of the NLP with respect to the number of decision variables is dependent on the discretisation mesh used for each phase within the OCP, with denser phase meshes corresponding to larger NLPs. This is because $\mathbf{Y}_i^{(p)}$ and $\mathbf{U}_j^{(p)}$ scale linearly in size with the number of discretisation nodes used within each phase. $\mathbf{X}_{\min}$ and $\mathbf{X}_{\max}$ are the lower and upper bounds on the decision variables and are typically supplied by the user as part of the OCP definition.

2.4.2 Functions

The NLP definition details two functions that need to be evaluated: the objective function $\mathcal{J}$, and the vector of constraints $\mathbf{C}$. Evaluation of $\mathcal{J}$ for a specific value of decision variables $\mathbf{X}$ is trivial from the definition in eq. (2.64). The column vector constraints in eq. (2.65) can be assembled as

$$\mathbf{C} = \begin{bmatrix} \mathbf{C}^{(1)} \\ \vdots \\ \mathbf{C}^{(P)} \\ \boldsymbol{\beta} \end{bmatrix}. \tag{2.71}$$

In the same way as for the decision variables, $\mathbf{C}$ is constructed as a concatenation of the phase-specific constraints $\mathbf{C}^{(p)}$ for $p \in [1, \ldots, P]$ and the phase-independent endpoint constraints $\boldsymbol{\beta}$. The phase-specific constraints for phase p are constructed as

$$\mathbf{C}^{(p)} = \begin{bmatrix} \boldsymbol{\Delta}_1^{(p)} \\ \vdots \\ \boldsymbol{\Delta}_{n_\Delta^{(p)}}^{(p)} \\ \boldsymbol{\Gamma}_1^{(p)} \\ \vdots \\ \boldsymbol{\Gamma}_{n_\Gamma^{(p)}}^{(p)} \\ \boldsymbol{\Lambda}^{(p)} \end{bmatrix}, \tag{2.72}$$

where $\boldsymbol{\Delta}_i^{(p)}$ is a column vector of the defect constraints associated with the ith state variable $\mathbf{Y}_i^{(p)}$ for $\left(i = 1, \ldots, n_y^{(p)}\right)$, $\boldsymbol{\Gamma}_i^{(p)}$ is a column vector of the jth path constraint applied at each of the discretisation nodes for $\left(j = 1, \ldots, n_\gamma^{(p)}\right)$, and $\boldsymbol{\Lambda}^{(p)}$ is a column vector of the $n_q^{(p)}$ integral constraints. The defect constraints $\boldsymbol{\Delta}_i^{(p)}$

can be generated using the generalised implicit integral-form K-stage Runge-Kutta collocation detailed in section 2.3 such that

$$\boldsymbol{\Delta}_i^{(p)} = \mathbf{E}^{(p)}\mathbf{Y}_i^{(p)} + \frac{{t_F}^{(p)} - {t_0}^{(p)}}{2}\mathbf{A}^{(p)}\mathbf{F}_i^{(p)}\,, \quad \left(i = 1, \ldots, n_y^{(p)}\right)\,. \tag{2.73}$$

The matrices $\mathbf{E}^{(p)}$ and $\mathbf{A}^{(p)}$ are the phase-level state-difference and integration matrices respectively.

Both $\mathbf{E}^{(p)}$ and $\mathbf{A}^{(p)}$ are constructed as block matrices from components generated using the generalised implicit integral-form K-stage Runge-Kutta collocation method and have the same block structure (shown in fig. 2.2). The phase in question p is constructed on a mesh $\mathcal{S}^{(p)}$ with K mesh sections. Each mesh section $\mathcal{S}_k^{(p)}$ for $(k = 1, \ldots, K)$ is discretised with $\nu_k^{(p)}$ stages and has an associated $\left(\nu_k^{(p)} - 1\right) \times \nu_k^{(p)}$ state-difference matrix $\mathbf{E}_k^{(p)}$, as well as a $\left(\nu_k^{(p)} - 1\right) \times \nu_k^{(p)}$ integration matrix $\mathbf{A}_k^{(p)}$. $\mathbf{E}_k^{(p)}$ is defined by eq. (2.46) while $\mathbf{A}_k^{(p)}$ is constructed using the algorithm detailed in section 2.3.8.

The integral constraints in phase p, $\boldsymbol{\Lambda}^{(p)}$, are constructed with the ith entry being

$$\Lambda_i^{(p)} = q_i^{(p)} - \frac{{t_F}^{(p)} - {t_0}^{(p)}}{2}\left\langle \mathbf{w}^{(p)}, \mathbf{g}_i^{(p)} \right\rangle\,, \quad \left(i = 1, \ldots, n_q^{(p)}\right)\,, \tag{2.74}$$

where the angle brackets denote the inner vector (dot) product. $\mathbf{w}^{(p)}$ is a column vector of quadrature weights corresponding to the discretisation used for $\mathcal{S}^{(p)}$. Entries is $\mathbf{w}^{(p)}$ are trivial for the internal stages of each mesh section, however the mesh nodes on the interior of the domain can be thought of both as the final node in the LHS mesh section and the initial node in the RHS mesh section. The entries corresponding to the interior mesh nodes are just the sums of the two appropriate quadrature weights from the two mesh sections that the mesh node in question are part of. $\mathbf{g}_i^{(p)}$ is a column vector of the ith integrand function of p evaluated at each of the discretisation nodes of $\mathcal{S}^{(p)}$.

Of the four types of constraints present in the NLP formulation, the defect and integral constraints must be equality constraints and so are bounded as

$$\boldsymbol{\Delta}_i^{(p)} = \mathbf{0}\,, \quad \left(i = 1, \ldots, n_y^{(p)}\right) \tag{2.75}$$

and

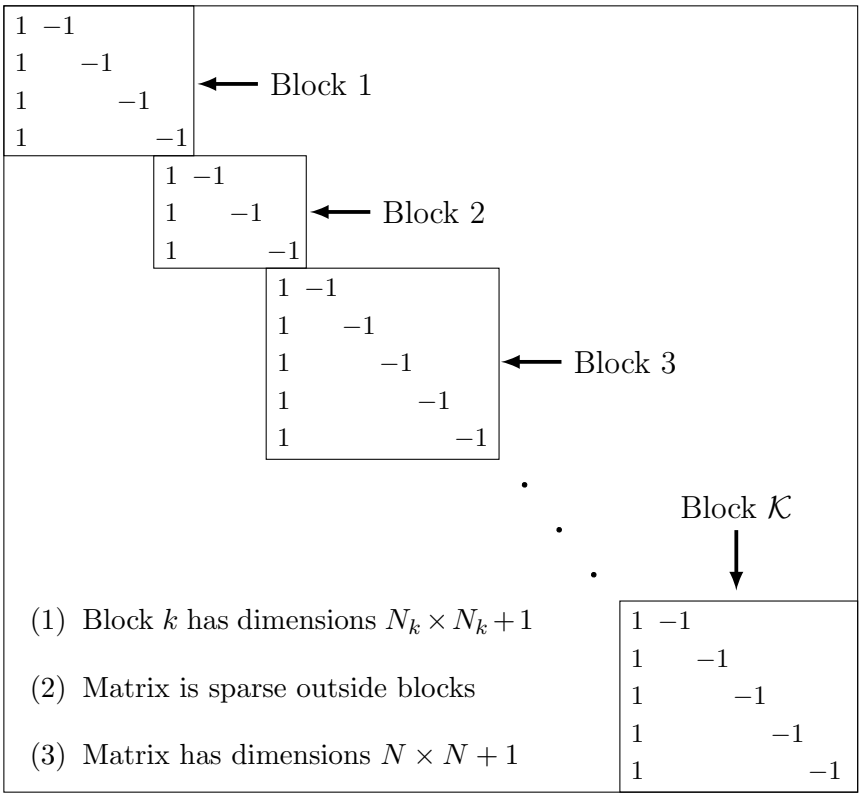
$$\boldsymbol{\Lambda}^{(p)} = \mathbf{0}\,. \tag{2.76}$$

The path and endpoint constraints are inequality constraints (although they may be equality constraints in some problems) and so are bounded as

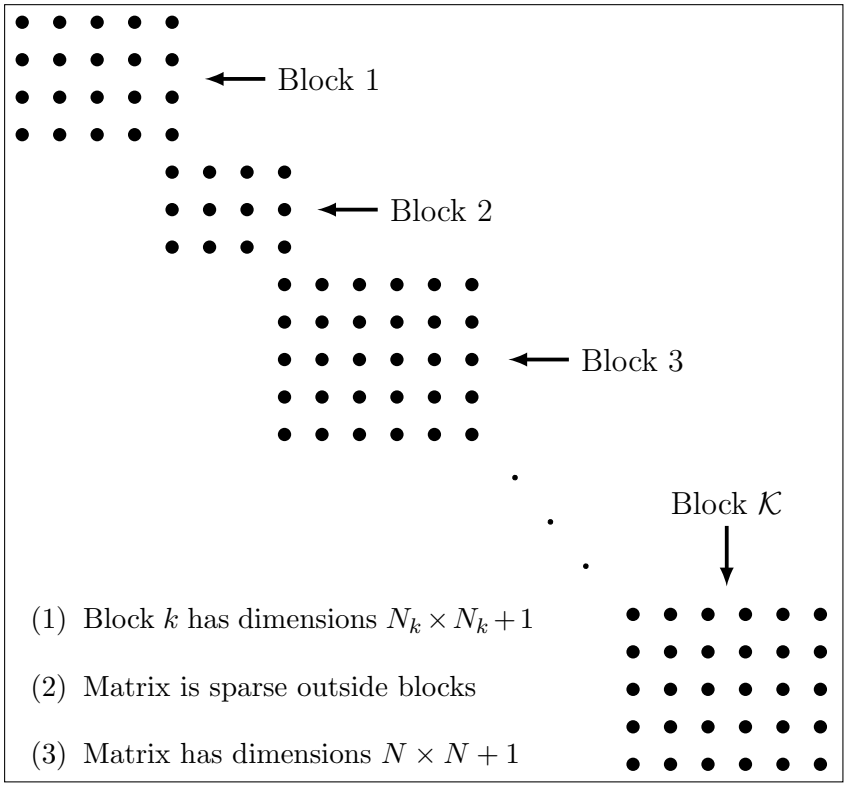
$$\boldsymbol{\Gamma}_{\min}^{(p)} \leq \boldsymbol{\Gamma}_i^{(p)}\left(\mathbf{X}\right) \leq \boldsymbol{\Gamma}_{\max}^{(p)}\,, \quad \left(i = 1, \ldots, n_y^{(p)}\right) \tag{2.77}$$

and

$$\boldsymbol{\beta}_{\min} \leq \boldsymbol{\beta}_i\left(\mathbf{X}\right) \leq \boldsymbol{\beta}_{\max}\,. \tag{2.78}$$



(a) Structure of the phase-level state-difference matrix, $\mathbf{E}^{(p)}$.



(b) Structure of the phase-level integration matrix, $\mathbf{A}^{(p)}$.

Figure 2.2: Block structures of matrices for constructing the defect constraints associated with the phase p.

2.4.3 Derivatives

Computational NLP solvers use derivative information about the NLP to inform their decisions about search direction. These universally use the first-order derivatives of the objective function gradient $\mathbf{g}$ (eq. (2.20)) and the constraints Jacobian $\mathbf{G}$ (eq. (2.21)). The second-order derivative, the Lagrangian Hessian $\mathbf{H}$ (eq. (2.23)), is also sometimes used as inclusion of an exact computation of $\mathbf{H}$ can significantly improve the convergence properties of a NLP [36, 131]. Using *Ipopt* as an example, functions need to be supplied which return evaluations of $\mathbf{g}$, $\mathbf{G}$ and $\mathbf{H}$ at a specific value of $\mathbf{X}$ [43]. Gradient information can be generated in many ways and is the subject of chapter 3.

2.4.4 Scaling

Ensuring that the transcribed NLP subproblem is well-scaled is vitally important because the scaling of the problem affects the convergence rate, termination tests and numerical conditioning [36]. Defining what makes a problem well-scaled is difficult. Furthermore, a problem that is well-scaled at one point in the solution space may be poorly scaled at another point.

Scaled Nonlinear Programming Problem

In relation to the NLP defined in eqs. (2.64) to (2.66), a scaled NLP with decision variables $\widetilde{\mathbf{X}}$ can be defined. The idea is to produce an equivalent NLP which possesses the following characteristics:

1. decision variables $\widetilde{\mathbf{X}}$ that are of the same order of magnitude and are bounded on the same domain [36];
2. an objective function $\widetilde{\mathcal{J}}$ that results in a gradient vector where the majority of entries are within a couple of orders of magnitude of unity [29]; and
3. constraint functions $\widetilde{\mathbf{C}}$ that are of the same order of magnitude, such that the entries within the Jacobian matrix are also of similar orders of magnitude and again within a couple of orders of magnitude of unity [29].

The scaled NLP is to determine the vector of scaled decision variables $\widetilde{\mathbf{X}}$ that minimises the objective function

$$\widetilde{\mathcal{J}} = \widetilde{\Phi}\left(\widetilde{\mathbf{X}}\right), \tag{2.79}$$

subject to the constraints

$$\widetilde{\mathbf{C}}_{\min} \leq \widetilde{\mathbf{C}}\left(\widetilde{\mathbf{X}}\right) \leq \widetilde{\mathbf{C}}_{\max} \tag{2.80}$$

and where the decision variables are bounded as

$$-\mathbf{0.5} \leq \widetilde{\mathbf{X}} \leq \mathbf{0.5}\,. \tag{2.81}$$

Motivation

The only complete description of how an OCP should be scaled is in [36]. However, this approach assumes that the derivative information is determined for the unscaled NLP functions with respect to the unscaled decision variables. This is then scaled by multiple scaling factors to account for the scaling of the decision variables and the scaling of the constraints. When another derivative-taking method is used (chapter 3), it can be desirable to express the NLP functions in terms of $\widetilde{\mathbf{X}}$ rather than $\mathbf{X}$. Section 2.4.4 puts together a framework of recommendations for scaling that: collects scaling recommendations from multiple resources, adapts the recommendations of [36] so that they're suitable for use with non-differencing differentiation methods, and provides some recommendations for how scaling should be approached based on practical experience of automatically scaling OCPs (section 2.7).

This framework of recommendations assumes that the OCP is defined in an unscaled form, NLP solver is interfaced with the scaled version of the NLP, and the differentiation method (chapter 3) reformulates the OCP functions in terms of the $\widetilde{\mathbf{X}}$.

Decision Variables

Scaled decision variables should be of the same order of magnitude and should occupy the same domain such that they are bounded on the interval $[-0.5, 0.5]$ as specified by eq. (2.81) [36]. The OCP variables should be scaled separately as this can have physical meaning, making the process easier. This scaling can then be transferred to the NLP [36].

The domains of the decision variables are scaled using the affine transform [36]

$$\mathbf{X} = \mathbf{V}\widetilde{\mathbf{X}} + \mathbf{r}. \tag{2.82}$$

$\mathbf{V}$ is a diagonal matrix of stretching weights and $\mathbf{r}$ is a column vector of translation weights. The entries of $\mathbf{V}$ and $\mathbf{r}$ can be determined using the upper and lower bounds

of the original NLP, $\mathbf{X}_{\min}$ and $\mathbf{X}_{\max}$ from eq. (2.66) where the ith elements are

$$V_{ii} = X_{i,\max} - X_{i,\min} \tag{2.83}$$

and

$$r_i = X_{i,\max} - \frac{X_{i,\max} - X_{i,\min}}{2}, \tag{2.84}$$

with $X_{i,\min}$ and $X_{i,\max}$ being the lower and upper bounds of the ith decision variable respectively. Note that both $\mathbf{V}$ and $\mathbf{r}$ will contain many repeated entries because a single state or control variable in the OCP will result in $\mathbf{X}$ containing many corresponding decision variables due to the discretisation.

The approach in eq. (2.82) is dependent on there being good lower and upper bounds on the decision variables. Application of this approach can be limited in situations where accurate bounds are not available. Software implementations should, therefore, prompt users to provide accurate bounds for each variable. This does, however, assume that the user has a good grasp of what sensible bounds are for their problem. If no bounds are available, a large value floating point number (e.g. 10×10^{20}) should be used in order to avoid computation problems if a numerical infinity is encountered. It is also recommended that a numerical limit (e.g. 10×10^{6}) be placed on the maximum allowable scaling factors to avoid the problem becoming over-scaled due to poor bounding information.

Objective Function

The objective function should be scaled so that the scaled gradient vector $\widetilde{\mathbf{g}}$ has entries within a couple of orders of magnitude of unity [36]. The relative magnitudes of the entries of $\widetilde{\mathbf{g}}$ are governed by the variable scaling of eq. (2.82). The absolute magnitudes can also be scaled by directly scaling the objective function. Using the definitions of eqs. (2.64) and (2.79), the scaled objective function can be defined as

$$\widetilde{\mathcal{J}} = w_{\mathcal{J}}\mathcal{J} = w_{\mathcal{J}}\Phi\left(\mathbf{X}\right) \tag{2.85}$$

where $w_{\mathcal{J}}$ is a linear scaling factor. The magnitudes of the entries in $\widetilde{\mathbf{g}}$ can therefore be controlled by adjusting the objective scaling factor $w_{\mathcal{J}}$. A good approach is to use the rule of thumb that if reciprocal of the Euclidian norm of $\widetilde{\mathbf{g}}$ is 1 then the resulting objective function and gradient will be well-scaled [29, 36]. Mathematically this gives

$$\|\widetilde{\mathbf{g}}\| = 1 \tag{2.86}$$

which can be substituted with eqs. (2.20) and (2.85) and rearranged to give

$$w_{\mathcal{J}} = \frac{1}{\left\|\frac{\partial \mathcal{J}}{\partial \widetilde{\mathbf{X}}}\right\|}. \tag{2.87}$$

The values of $\widetilde{\mathbf{X}}$ that should be used when computing $w_{\mathcal{J}}$ using eq. (2.87) are discussed below.

Constraints

The scaling of the constraints and their Jacobian is to produce a scaled constraints vector in which the entries are of similar orders of magnitude, while also producing a Jacobian whose entries are within a couple of orders of magnitude of unity. As the constraints vector is constructed from defect, path, integral and endpoint constraints, the scaling of each of these should be considered separately.

Defect constraints should be scaled proportionally by the scaling factor used to stretch their associated state variable [36]. To implement this, the scaled defect constraints of phase p, discretised on the mesh $\mathcal{S}^{(p)}$, can be presented mathematically as

$$\widetilde{\boldsymbol{\Delta}}^{(p)} = \mathbf{W}_{\Delta}^{(p)} \boldsymbol{\Delta}^{(p)} = \left[\mathbf{V}_{y}^{(p)}\right]^{-1} \boldsymbol{\Delta}^{(p)}, \tag{2.88}$$

where $\mathbf{W}_{\Delta}^{(p)} = \left[\mathbf{V}_{y}^{(p)}\right]^{-1}$. Similarly, integral constraints should be scaled proportionally by the scaling factor used to stretch their associated integral variable as they rely on numerical quadrature [36]

$$\widetilde{\boldsymbol{\Lambda}}^{(p)} = \mathbf{W}_{\Lambda}^{(p)} \boldsymbol{\Lambda}^{(p)} = \left[\mathbf{V}_{q}^{(p)}\right]^{-1} \boldsymbol{\Lambda}^{(p)} \tag{2.89}$$

where $\mathbf{W}_{\Lambda}^{(p)} = \left[\mathbf{V}_{q}^{(p)}\right]^{-1}$.

Path and endpoint constraints are not linear functions of decision variables like defect and integral constraints. They, therefore, need to be scaled using a different approach. A method similar to that used to scale the objective function and its gradient can be used as this can produce well-sized derivatives [29]. If the path constraints in phase p are to be scaled by $\mathbf{W}_{\Gamma}^{(p)}$, then these weights should be computed as

$$W_{\Gamma,i}^{(p)} = \left\| \frac{\partial \boldsymbol{\Gamma}_{i}^{(p)}}{\partial \widetilde{\mathbf{X}}} \right\|^{-1}, \quad (i = 1, \ldots, n_{\Gamma}), \tag{2.90}$$

where $\boldsymbol{\Gamma}_{i}^{(p)}$ is the ith path constraint evaluated at each of the discretisation nodes within $\mathcal{S}^{(p)}$ and $W_{\Gamma,i}^{(p)}$ is the ith entry in $\mathbf{W}_{\Gamma}^{(p)}$. In the same manner, the endpoint constraints can be scaled by $\mathbf{W}_{\beta}$ with the ith weight $W_{\beta,i}$ corresponding to the ith endpoint constraint being computed as

$$W_{\beta,i} = \left\| \frac{\partial \beta_{i}}{\partial \widetilde{\mathbf{X}}} \right\|^{-1}, \quad (i = 1, \ldots, n_{\beta}). \tag{2.91}$$

Sampling Points

Calculation of the objective and constraints scaling factors involve evaluating first derivatives of the NLP. This requires choosing values for $\mathbf{X}$ at which these functions can be evaluated. A sensible starting point is to calculate the scaling factors using evaluations of the NLP functions at the user-specified initial guess $\mathbf{X}_0$ [36].

It can be a good idea in practice to also compute the scaling factors by randomly sampling the search space because a good initial guess is not always guaranteed. There is no solid mathematical basis for how to best sample the search space, but through trial and error as part of this thesis two rules-of-thumb were found:

1. sample points should be normally distributed where the mean is centrally located between a variable's lower and upper bounds and the standard deviation is one sixth of the bounded range (noting that sampling points produced which lie outside of the variable's range should be shifted to lie on the bound); and

2. the median value of 100 samples for each scaling parameter gave a reliable estimate which do not tend to change dramatically when significantly more sampling points were used.

Updating Scaling Between Mesh Iterations

If mesh refinement is being used to solve a series of NLP subproblems, then for each mesh iteration a good approximation to the OCP's solution should be available after the first mesh iteration. If sampling-based scaling has been used it is usually the case that recomputing the scaling factors using the solution from the previous mesh iterations results in a better-scaled NLP subproblem at the next mesh iteration [263]. However, changing the scaling between successive NLP subproblems can drastically change the convergence properties and even solution. It is typically better, therefore, to gradually adjust the magnitudes of the scaling factors between mesh iterations. If scaling is to be updated, it is suggested to use an exponential moving average such that the generic scaling factor for the next mesh iteration ϕ_{M+1} can be calculated as

$$\phi_{M+1} = \frac{1}{M} \sum_{i=1}^{M} \left[\alpha \left(1 - \alpha\right)^{(M-i)} \phi_i \right] \tag{2.92}$$

where α is a weighting and where there have been M previous mesh iterations which used the scaling factor ϕ_i for $\phi = [1, \ldots, M]$. Again, through trial and error as part of this thesis, a value of $\alpha = 0.8$ is suggested.

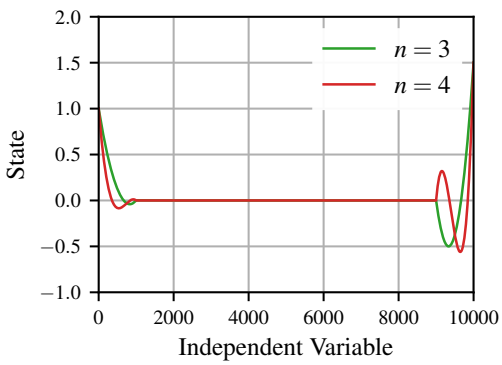
2.5 Mesh Refinement

2.5.1 Motivation

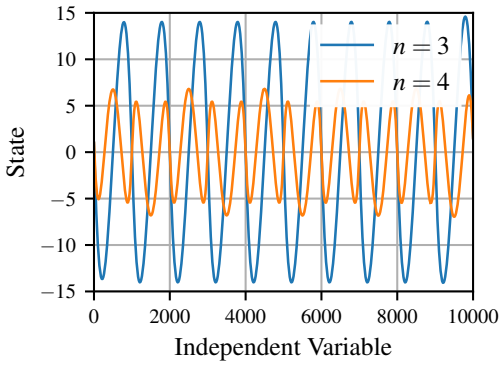
For OCPs where the solution is simple and smooth, conducting mesh refinement is trivial, and may not be required at all. Difficulty arises when the solution to the OCP in question is either highly nonlinear or contains a discontinuity in either its control or its dynamics [36]. In these cases, mesh refinement is needed so that the error between the transcribed, discretised NLP and the real, continuous OCP is less than some desired mesh tolerance [262]. Performant solving of an OCP usually means meeting this desired mesh tolerance in as quick a time as possible [9]. Performant mesh refinement means identifying where additional nodes are needed in the discretisation to decrease the mesh error, whilst also identifying locations where nodes can be removed without the mesh error in that region dropping below the mesh tolerance [36].

One category of OCPs for which mesh refinement for vital in their solving are hypersensitive problems. Numerous examples of hypersensitive problems have been presented in the literature [36, 263, 284]. This class of problem exhibit the features: nonzero endpoint constraints on the state; a solution to the state and control which is zero for the central portion of the time domain; and sharp features in the solution concentrated at the endpoints which appear discontinuous but are in fact smooth. An example optimal state and control for a hypersensitive problem are shown in fig. 2.6 (in section 2.7.4). For mesh refinement to produce an efficient mesh for hypersensitive problems, the collocation nodes need to be highly concentrated in two small regions at either end of the time domain, with the centre of the time domain containing a sparse distribution of collocation nodes.

When an OCP is first solved, it is typical to initialise the first mesh iteration on a mesh with a small number of uniform mesh sections. This is because it is unknown where any important features in the solution are within the time domain. In hypersensitive problems, where the important features lie at the very ends on the time domain, this typically results in the polynomials associated with the lateral mesh sections interpolating through a set of points which are all zero except for one value. This gives rise to highly oscillatory interpolation due to a polynomial not being a suitable function for accurately interpolating a set of values like these. For a mesh refinement algorithm to work well with the K-stage Runge-Kutta discretisation presented earlier in this chapter, the algorithm needs to be universally suitable for LG, LGR and LGL collocation.



(a) Interpolation at the non-symmetric Radau points using LGR collocation.



(b) Interpolation at the symmetric Lobatto points using LGL collocation.

Figure 2.3: Comparison of the oscillatory interpolation of a hypersensitive problem using a symmetric and a non-symmetric collocation scheme

In LGL collocation, the collocation condition is enforced at both ends of each mesh section. This means that, unlike for LG and LGR collocation, the state derivative is continuous at the mesh boundaries and any oscillations present in one mesh section are propagated to the adjacent mesh sections by this two-sided collocation condition. Figure 2.3a illustrates the presence of the oscillations in the endpoint mesh sections when LGR collocation is used, while fig. 2.3b illustrates the propagation of these oscillations across the whole time domain as a result of the additional collocation conditions present in LGL collocation.

These oscillations are unavoidable and are damped out through the mesh refinement process when the concentration of nodes increases at the endpoints. With sufficient nodal density, each polynomial interpolates through a set of points which it is a suitable approximation for. Any mesh refinement algorithm will analyse these oscillations as being inaccuracies in the solution and will therefore increase the density of collocation nodes in these regions. In the case of LGL collocation, this is problematic for the central region of the solution because this apparent inaccuracy is not a true inaccuracy and will reduce naturally as the solution improves at the endpoints. A mesh refinement algorithm will initially introduce additional unnecessary nodes at the interior of the time domain. As shown in 2.3b, the magnitude of the oscillations can exceed the maximum magnitude of the true solution, particularly when a low-order LGL scheme is used. The large magnitude oscillations are interpreted as very large mesh errors and, as such, result in aggressive mesh refinement in which many addition collocation nodes are added to the discretisation mesh. For a mesh refinement algorithm to be performant it needs to be able to accurately detect regions where there are unnecessary densities of nodes and remove these from the discretisation to reduce the size of the NLP for the following mesh iteration.

This section details the development of a novel mesh refinement algorithm suitable for use with LGL collocation. The novel algorithm builds on the *hp*-method presented in [262]. It is extended so that it allows nodal density to be removed in regions where the mesh tolerance is met, while still allowing it to be increased in regions where the solution is deemed inaccurate. The algorithm of [262]was chosen as the basis for the mesh refinement presented in this thesis as it has been shown to exhibit consistent performance on a range of problems [9]. The base algorithm from [262] contains two steps, mesh error assessment and mesh refinement, only the latter of which was modified in this work.

2.5.2 Mesh Error Assessment

Error assessments can be performed using error estimators, which provide objective error measures, or error indicators, which use heuristics to obtain information about the error. A heuristic error indicator based on only the state error is used here. While this is only an estimation of the true error, it has been shown that the following calculation results in an error estimation that is nearly identical to the true error [262].

The error indicator is calculated by comparing a pair of approximations to the NLP solution [262]. A second, denser discretisation mesh $\hat{\mathcal{S}}_i$, with normalised timepoints τ_{ij}, is constructed [262]. The first approximation to the solution $Y(\hat{\tau}_{ij})$ is produced by interpolating the solution at τ_{ij} using Lagrange polynomials [262]. The second approximation $\hat{Y}(\hat{\tau}_{ij})$ is given by

$$\hat{Y}(\hat{\tau}_{ij}) = Y(\tau_{i1}) + h_i \sum_{l=1}^{m_i} \hat{A}_{jl} f(Y(\tau_{ij}), U(\tau_{ij}), \tau_{ij}) , \quad (j = 2, \ldots, m_i) , \tag{2.93}$$

where h_i is the width of the non-normalised mesh section $\mathcal{S}_i$, $\hat{A}_{jl}$ is the appropriate entry from the $(m_i - 1) \times m_i$ integration matrix $\hat{\mathbf{A}}$ corresponding to $\hat{\mathcal{S}}_i$, $Y(\tau_{ij})$ is the Lagrange interpolation of the state solution, and $U(\tau_{ij})$ is the Lagrange interpolation of the control solution.

Comparison of $Y(\hat{\tau}_{ij})$ and $\hat{Y}(\hat{\tau}_{ij})$ can be used to estimate the mesh errors [262]. The *absolute errors* are calculated at each of the nodes of $\hat{\mathcal{S}}$ as

$$E(\hat{\tau}_{ij}) = \left| \hat{Y}(\hat{\tau}_{ij}) - Y(\hat{\tau}_{ij}) \right| , \quad (i = 1, \ldots, K) , \quad (j = 1, \ldots, m_i) . \tag{2.94}$$

The *relative errors* are calculated for each mesh section

$$e(\hat{\tau}_{ij}) = \frac{E(\hat{\tau}_{ij})}{1 + \max_{l \in [1,\ldots,N]} |Y(\hat{\tau}_{ij})|} . \tag{2.95}$$

The *maximum relative error* for each mesh section is calculated by comparing the relative errors for each mesh section across the set of n_y state variables in the OCP

$$e_{\max} = \max_{\substack{i\in[1,\ldots,n_y]\\ l\in[1,\ldots,M]}} e_i^{(k)}\left(\hat{\tau}_l\right) . \tag{2.96}$$

2.5.3 Mesh Refinement Algorithm

Estimation of Required Order of Collocation Method

For the OCP to be successfully solved, a mesh tolerance ϵ_{tol} needs to be met in all mesh sections of $\mathcal{S}$. If ϵ_{tol} is not met in all mesh sections of $\mathcal{S}$ then the mesh will need to be refined, by adjusting the number and width of mesh sections and the order of the collocation scheme within each mesh section, and the new transcribed OCP solved. A method for predicting the number of nodes per mesh section n_i^* that are required in order for ϵ_{tol} to be met is given in [262], based on theory from [142, 171]. This is done by calculating

$$n_i^* = n_i + p_i^* , \tag{2.97}$$

where n_i is the number of nodes in the ith mesh section in $\mathcal{S}$,

$$p_i^* = \begin{cases} p_i & p_i > 0 \\ p_i + \lceil \ln\left(1 - p_i\right) \rceil & p_i \leq 0 \end{cases} \tag{2.98}$$

and

$$p_i = \left\lceil \log_{n_i}\left(\frac{e_{\max}}{\epsilon_{\text{tol}}}\right) \right\rceil . \tag{2.99}$$

Mesh Section Adjustment Type Identification

The novel algorithm now diverges from the one in [262]. Using eq. (2.98) to determine how the number of nodes within each mesh section should be adjusted, mesh sections are then categorised based on one of four options:

1. the mesh section can be merged via an *hp*-method;
2. the mesh section can be subdivided via an *hp*-method;
3. the mesh section should remain *h*-unchanged but may be refined via a *p*-method; or
4. the mesh section is suitable as is and should remain *hp*-unchanged.

If the minimum and maximum allowable number of nodes in a single mesh section are $N_{\min}$ and $N_{\max}$ respectively, the mesh section is suitable for merging (it is *mergeable*) if $n_i^* < N_{\min}$. If $n_i^* > N_{\max}$ then it is estimated that in order to meet the mesh tolerance the mesh section will be required to contain more nodes than are allowed and should, therefore, be subdivided into more than one new mesh section. If neither of these criteria are met then the mesh section in question is predicted to meet the mesh tolerance by changing the number of nodes it contains to n_i^* where $N_{\min} \leq n_i^* \leq N_{\max}$ and can therefore be refined using a simple p-method. Finally, if $n_i^* = n_i$ then the mesh tolerance is met to a degree where the current mesh in that region is suitable and should remain unchanged to the next mesh iteration.

Whole Mesh Analysis

For a merger of mesh sections to occur, a mergeable mesh section must lie adjacent to other mergeable mesh sections. Where two or more mergeable mesh sections lie adjacent, they can be merged to form a suitable number of new mesh sections. The refinement proceeds with a pass across the whole mesh to identify groups of adjacent mergeable mesh sections. All other categories of mesh section can be treated as independent.

Mesh Section Merging

Mergeable mesh sections can either

1. form part of a mergeable group; or
2. not lie adjacent to any other mergeable mesh sections.

If a mesh section does not have mergeable neighbours, then its number of nodes is reduced to $N_{\min}$, the maximum allowable reduction. If a mesh section forms part of a mergeable group, then the different widths of, and numbers of nodes in, all mesh sections need to be considered to best distribute the predicted error.

Merging aims to redistribute as few nodes as possible, located with their density corresponding to the estimated error. Consider a merge group containing the M mesh sections $\mathcal{S}_j$, each containing n_j nodes, for $(j = 1, \ldots, M)$, which is to be refined to a new set of $\hat{M}$ mesh sections $\hat{\mathcal{S}}_k$ for $\left(k = 1, \ldots, \hat{M}\right)$. The merge ratio

$$\hat{m}_j = \frac{n_j}{\hat{n}_j} = \frac{n_j}{N_{\min} - p_j^*}\,, \quad (j = 1, \ldots, M)\,, \tag{2.100}$$

is calculated, motivated by an attempt to achieve the maximum allowable node reduction is each $\mathcal{S}_j$

$$\hat{n}_j = N_{\min} - p_j^* \,, \quad (j = 1, \ldots, M) \tag{2.101}$$

The merge ratio can be thought of as the factor by which $\mathcal{S}_j$ of width h_j and containing n_j^* nodes would need to be increased such that the nodal density would exactly equal a wider mesh section containing $N_{\min}$ nodes. Division by $\hat{n}_j$ (eq. (2.100)) ensures that $0 < \bar{m}_j < 1$ always holds. A smaller value of $\bar{m}_j$ indicates that the portion of the domain currently occupied by $\mathcal{S}_j$ can be more sparsely populated in $\hat{\mathcal{S}}$.

The number of mesh sections $\hat{M}$ required in the merged mesh portion is calculated as

$$\hat{M} = \left\lceil \sum_{j=1}^{M} \hat{m}_j \right\rceil \,, \tag{2.102}$$

with the ceiling rounding ensuring that a conservative integer value is obtained. The scaled mesh section widths

$$\hat{w}_j = \frac{h_j}{\hat{m}_j} \,, \quad (j = 1, \ldots, M) \tag{2.103}$$

use the merge ratio so that all mesh sections in the merge group can be compared relative to one another. The reduction factor

$$\hat{\rho}_j = \phi \hat{w}_j \,, \quad (j = 1, \ldots, M) \,, \tag{2.104}$$

gives the relative required nodal densities across the merge group domain. ϕ (eq. (2.104)) is a normalisation factor

$$\phi = \left[\sum_{j=1}^{M} \bar{h}_j \right]^{-1} \,. \tag{2.105}$$

A new set of mesh points defining $\hat{\mathcal{S}}_j$ for $\left(j = 1, \ldots, \hat{M}\right)$ is computed from the reduction factors. Each of the $\hat{M}$ new mesh section $\hat{\mathcal{S}}_j$ is to contain $N_{\min}$ nodes with $\hat{h}_j$ being chosen so that the cumulative reduction factor is uniformly distributed. A cumulative density function $\Theta(\tau)$ is computed by linearly interpolating the independent vector of mesh points in the merge group against the dependent vector of the cumulative sum of $\bar{\rho}_j$ for $(j = 1, \ldots, M)$. $\Theta(\tau)$ is then evaluated at $\hat{M} + 1$ linearly spaced points, which gives the locations of the $\hat{M} + 1$ new mesh points with the first and $\left(\bar{M} + 1\right)$th mesh points corresponding to the extrema of the merge group's domain.

Mesh Section Subdivision

Mesh section subdivision follows a similar approach to the one presented in [262]. For mesh sections where $n_i^* > N_{\max}$, the subdivision factor

$$\hat{s}_i = \left\lceil \frac{n_i^*}{N_{\min}} \right\rceil \tag{2.106}$$

is computed. $\mathcal{S}_i$ is split in to $\hat{s}_i$ new mesh sections $\bar{\mathcal{S}}_j$ for $(j = 1, \ldots, \bar{s})$, of uniform width, each containing $N_{\min}$ nodes.

Mesh Section Order Reduction

An adjustment to how mesh order reduction is made in the novel algorithm compared to that in [262] is outlined below. The complementary collocation condition in LGL collocation has been shown to cause transfer of oscillations between adjacent mesh sections (fig. 2.3b). This algorithm reduces the sensitivity of mesh section order reduction so that if the mesh tolerance is almost met, a reduction in mesh section order does not risk the mesh tolerance being unmet in the next mesh iteration.

For each mesh section, a reduction tolerance

$$\xi_i^+ = \max \left[\left(1 - \frac{1}{\ln\left(\frac{e_{\max}}{\epsilon_{\text{tol}}}\right)} \right), 0 \right] \tag{2.107}$$

is calculated. ξ_i^+ modifies the allowable order of mesh section order reduction such that the closer $e_{\max}$ is to ϵ_{tol}, the fewer the number of nodes that are removed from a mesh section. From experience in solving hypersensitive problems using LGL collocation (section 2.7.4), it was found that this logarithmic adjustment performed well. When the mesh tolerance is not met, the mesh section $\mathcal{S}_j$ containing n_j nodes and predicted to require an adjustment by $p_j^* \leq 0$ nodes, is refined to contain

$$\hat{n}_j = \left\lceil n_j + \xi_i^+ p_j^* \right\rceil \tag{2.108}$$

nodes.

Algorithmic Flow

1. Set the mesh iteration counter $i \leftarrow 1$; define the initial mesh $\mathcal{S}_1$.
2. Solve the transcribed NLP subproblem (section 2.4) on $\mathcal{S}_i$.
3. Compute the maximum relative error $e_{\max}$ (eq. (2.96)); compare $e_{\max}$ to the mesh error tolerance ϵ_{tol}.

4. If $e_{\max} < \epsilon_{\text{tol}}$, quit; else proceed.
5. Conduct mesh refinement to produce a new mesh $\mathcal{S}_{i+1}$.
   (a) Estimate the required order of mesh refinement for each mesh section.
   (b) Categorise the type of refinement each mesh section needs to undergo (merge, subdivide, reduce order, or remain unchanged).
   (c) Conduct a single pass over the whole mesh to identify groups of mergeable mesh sections.
   (d) Refine the mesh using the category-specific steps outlined in section 2.5.3.
   (e) Increment the mesh counter $i \leftarrow i + 1$; return to step 1.

2.6 Software Implementation: *Pycollo*

In this section, the development of a software package for solving OCPs is detailed. The package is called *Pycollo*, with its name derived from *Python* and collocation. *Python* was chosen as the implementation language due to its high suitability for scientific computing [244, 256] and extensive scientific computing ecosystem [30, 147, 178, 206, 240, 321], which can facilitate rapid and easy development. *Pycollo* implements both state-of-the-art methods from the field of optimal control (sections 1.1 and 2.1), as well as the methods and algorithms described earlier in this chapter and chapter 3. It is an open-source software package, meaning that it and its source code are available for use by anyone. *Pycollo* also forms the foundation of the BPST described in section 1.3.

2.6.1 Overview

Pycollo can solve the general multiphase OCP as defined by eqs. (2.1) to (2.7) in section 2.1.1. It does this numerically via the transcription method. Specifically, *Pycollo* uses the implicit integral-form K-stage Runge-Kutta collocation method described in section 2.3 to form the NLP described in section 2.4.

Pycollo is built using an *object-oriented programming* (OOP) architecture and *application programming interface* (API). This facilitates efficient and robust design and verification [161], while also being intuitive for users [87, 91, 299]. The simplified architecture of *Pycollo* is shown fig. 2.4.

The main method of interaction with *Pycollo* is via the `OptimalControlProblem` class, which acts as the highest level container for all OCP-related information. This

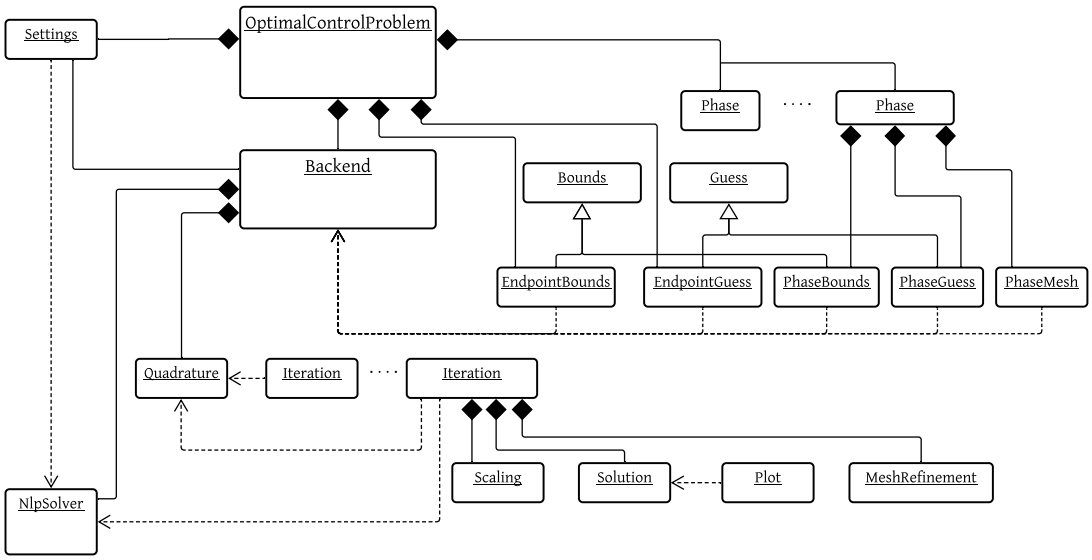


Figure 2.4: A simplified diagram of the architecture of *Pycollo* using the *Unified Modelling Language* (UML).

class is itself used to define the objective function (eq. (2.1)), endpoint constraints (eq. (2.5)) and parameter variables. Each OCP can have one or more phases associated with it. Similarly, each `OptimalControlProblem` instance can have one or more linked `Phase` instances. The `Phase` class is where state, control, integral and time variables are defined for a specific phase. It is also where state equations (eq. (2.2)), path constraints (eq. (2.3)), integrand functions (eqs. (2.4) and (2.7)) and state endpoint constraints are defined.

OCPs require bounds and initial guesses to be prescribed before they can be numerically solved. This is done in the *Pycollo* API by using the `bounds` and `guess` properties of `OptimalControlProblem` and `Phase` instances. For phase-specific bounds and guesses, `Phase` uses the `PhaseBounds` and `PhaseGuess` classes. Endpoint bounds and guesses are similarly provisioned for by the `EndpointBounds` and `EndpointGuess` classes.

A wide range of user-customisable settings are available in *Pycollo*. These are contained within an instance of the `Settings` class, which is accessible via the `settings` property of `OptimalControlProblem`.

Pycollo solves OCPs numerically using the transcription method and orthogonal collocation. As such, it implements functionality for discretising the OCP based on a specific collocation scheme and interfacing with NLP solvers. This functionality is hidden from users and is implemented by the `Backend` class. Pycollo also implements advanced features to improve performance, such as scaling OCPs and mesh refinement.

Figure 2.5 illustrates the simplicity of the *Pycollo* API. It includes the code for a simple example, in which the hypersensitive problem of section 2.7.4 is shown. Some more advanced implementation details of *Pycollo* are discussed in the following sections.

2.6.2 Transcription and Derivative Generation

The `Backend` class is responsible for handling the transcription of the user-defined OCP to a NLP subproblem that can be solved by a NLP solver. *Pycollo* allows any of LG, LGR and LGL collocation to be used, by implementing the approach based on implicit K-stage Runge-Kutta methods detailed in section 2.3.8. The `Quadrature` class is responsible for discretising the OCP at the correct quadrature points and enforcing the collocation condition at the correct discretisation nodes.

An instance of the `Iteration` class is used to define and solve the NLP subproblem associated with a specific mesh iteration. This class formulates the NLP subproblem and interfaces with the NLP solver. As this process is complex and involved, *Pycollo* allows for the `Backend` class to be subclassed, so that other packages can be leveraged to provide some of this functionality. By default, *Pycollo* uses its `CasadiBackend`, which leverages *CasADi*, the open-source tool for nonlinear optimisation and *algorithmic differentiation* (AD). `CasadiBackend` uses *CasADi* to interface with the NLP solver *Ipopt* and compute the NLP derivatives using AD. For each `Iteration`, `CasadiBackend` constructs a new mesh-specific NLP and passes this to *CasADi* for solving. While *CasADi* is useful in that it removes the need for *Pycollo* to generate derivatives itself, it is inefficient as it recalculates the NLP derivatives in a mesh-specific manner for each mesh iteration, even though the OCP functions remain unchanged. A further backend, which addresses some of the limitations of the *CasADi* backend, is developed and detailed in section 3.6.

2.6.3 Scaling

Pycollo supports manual and automatic scaling of the NLP subproblem to improve its numerical conditioning, the latter of which is conducted by following the framework outlined in section 2.4.4. For automatic scaling, *Pycollo* uses the user-supplied bounds and guesses to inform how the NLP variables should be scaled and the NLP derivative functions be sampled, to produce scaling coefficients for the constraints and objective function. *Pycollo* also supports the updating of scaling between successive mesh iterations as more information becomes available, again using the approach outlined in section 2.4.4.

2.6.4 Mesh Refinement

Pycollo implements the mesh refinement algorithm detailed in section 2.5. After a mesh iteration has been successfully solved, with the NLP solver returning a solution to the NLP subproblem, *Pycollo* computes the mesh error and constructs a mesh for the next mesh iteration. *Pycollo* uses the `interpolate` module from *SciPy* to compute the mesh error using the method outlined in [262]. The state solution is interpolated using polynomial splines, while also ensuring that the collocation conditions are met at each of the collocation nodes. *SciPy* is only capable of interpolating, while also meeting the collocation condition, if the Lobatto points are used. Therefore, *Pycollo* is currently only able to compute mesh errors, and conduct mesh refinement, for Lobatto-based collocation.

2.6.5 Settings

Pycollo provides a wide range of user-customisable settings, all of which include a default that has been chosen based either on sound theoretical reasoning or on experience from practically solving OCPs using *Pycollo*. Examples of settings include: the NLP and mesh tolerances; maximum allowable NLP and mesh iterations; scaling method and scaling update properties; collocation scheme; mesh refinement algorithm, and maximum and minimum allowable number of collocation nodes per mesh section; whether exact or approximate second-order derivatives be used; and what information is output to the console during a solve.

2.6.6 Packaging

Pycollo has been released as an open-source package and is available on both *PyPI* [271] and *conda-forge* [78]. The *Pycollo* source code is also made freely available on *GitHub* [63].

2.7 *Pycollo* Benchmarking Investigations

2.7.1 Motivation

In order to test the performance of *Pycollo*, a collection of five test OCPs from the academic literature were solved using the package. Other authors have solved and

published results for the selected test OCPs so their solutions are known, making them suitable for validating and benchmarking against. A varying range of OCPs were selected in order to test many different aspects of *Pycollo*, including its ability to handle problems with: highly nonlinear dynamics (examples 1, 4 and 5); variables scaled across different orders of magnitude (examples 2 and 3); step changes in their optimal control and discontinuities in their dynamics (examples 4 and 5); and multiple phases (example 5).

A number of different OCP software packages were considered for benchmarking *Pycollo* against. *GPOPS-II* [263] was selected due to it being arguably the most advanced currently-available general-purpose collocation software package. It is worth noting that while a more recent iteration of OCP solving software from the developers of *GPOPS-II*, called *CGPOPS* [9], has been published, licenses for this were unavailable at the time this work was conducted. *GPOPS-II* is similar to *Pycollo* in that it implements a form of orthogonal collocation, can handle multiphase problems, implementes mesh refinement, and distributes ready-written implementations of the example problems selected for this thesis. The two software packages can, therefore, be directly compared in many ways. Unlike *Pycollo*, *GPOPS-II* is proprietary software and, therefore, requires a licence to use.

2.7.2 Performance Metrics

Characterising the performance of a specific approach to solving a particular OCP can involve many aspects. The most important metric should be whether the OCP is solved correctly by the software. For these benchmark investigations, the *solution accuracy* was assessed by comparing the optimal costs quoted in the original references, and those obtained by both *Pycollo* and *GPOPS-II*. Graphical comparisons of the optimal state and control obtained from both *Pycollo* and *GPOPS-II* are also shown for each test OCP.

How quickly an OCP can be solved is another important consideration. Speed can relate to both the time spent formulating and constructing the problem, and the time spent by the software computing the solution once executed by the user. The time spent by the software package solving the problem was used as a measure of *computational performance*. The time for problem initialisation was recorded alongside the time for NLP initialisation, NLP solve and mesh refinement for each mesh iteration. These values were combined to give a total solve time metric for both software packages on each test OCP, with shorter times meaning better performance. When measuring the timing metrics, all OCPs were solved five times with the fastest and slowest solves discounted and the remaining three mean-averaged to negate any

unintended interrupts from the operating system. To compare the efficiency of mesh refinement in both software packages, the number of mesh iterations required I, the number of discretisation nodes used for each NLP subproblem N, and the mesh error at the end of each mesh iteration $e_{\max}$ were also recorded.

As time spent formulating the problem is difficult to quantify, *source lines of code* (SLOC) was included as an additional numerical measure under the justification that fewer SLOC in a software's implementation of an OCP is likely to correspond to one that was quicker and less onerous on the user to construct. While it is acknowledged that this is an imperfect measure, especially when comparing between implementations in different programming languages with different syntax, using SLOC does provide an approximation of the *ease of problem construction* for a user using a specific software package. To present an as unbiased count of SLOC as possible, only *logical lines of code* (LLOC) were counted to ensure that the counts were insensitive to formatting and style conventions. This means only executable statements were counted, and that whitespace lines and comments were omitted in the quoted figures.

2.7.3 Software Settings

To present an as fair comparison between *Pycollo* and *GPOPS-II* as possible, a standard set of solver settings were chosen before the set of test OCPs were solved. These settings were used consistently throughout. *Pycollo* was used without modification so that all of its settings were default. For *GPOPS-II*, default settings were used where possible and were only adjusted where a change was necessary to allow solving of all problems. For example, *GPOPS-II* uses no scaling by default but automatic scaling was used in all cases for benchmarking as some of the test OCPs are poorly scaled due to units varying in orders of magnitude. NLP and mesh tolerances of 10^{-10} and 10^{-7} respectively were used by both software packages so that these were the same.

One major difference between the settings chosen was that *Pycollo*'s Lobatto-based collocation scheme was used, while *GPOPS-II* only supports Radau-based collocation. This was because *Pycollo* can only currently compute mesh errors, and therefore implement mesh refinement, for this collocation scheme. *Pycollo* uses the Lobatto-based scheme by default as it is directly comparable to integral-form trapezoidal collocation when each mesh section contains two collocation nodes [38] and integral-form Hermite-Simpson collocation when each mesh section contains three collocation nodes [38, 94], and these are the most commonly used approaches [36, 38, 91, 194]. Integral form was used to formulate the defect constraints by both

software packages. All problems were initialised on the same mesh per phase, consisting of 10 identical mesh sections, each containing four discretisation nodes. This results in each state and control being discretised to 31 decision variables in the first NLP subproblem.

All computations were performed on a 2019 MacBook Pro running macOS Catalina 10.15.7, with a 2.4 GHz 8-Core Intel Core i9 CPU and 32 GB of 2667 MHz DDR4 RAM. All *Pycollo* scripts were executed by *Python* 3.7.8 in a virtual environment with *Pycollo* 0.3.0 installed. All *GPOPS-II* scripts were executed in *MATLAB* R2020a.

2.7.4 Example 1: Hypersensitive Problem

Problem Definition

The hypersensitive problem (originally in [284] and presented as ex. (4.4) in [36]) was chosen as it is a good exemplar of the importance of mesh refinement. This is due to its solution exhibiting all of its interesting behaviour at the phase endpoints, with the state and control being zero for the vast majority of the problem. The single-phase problem involving the state $\mathbf{y} = [y]$ and control $\mathbf{u} = [u]$ is to minimise

$$J = q \tag{2.109}$$

subject to the dynamical constraint

$$\dot{y} = u - y^3 \tag{2.110}$$

and state endpoint constraints

$$y\left(t_0\right) = 1 \qquad\qquad y\left(t_F\right) = 1.5\,, \tag{2.111}$$

where

$$q = \frac{1}{2}\int_{t_0}^{t_F} y^2 + u^2 dt \tag{2.112}$$

and

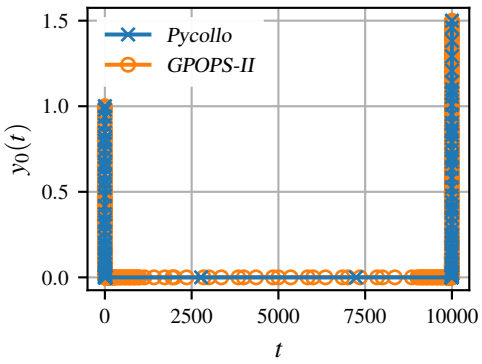
$$t_0 = 0 \qquad\qquad t_F = 10000\,.$$

Results and Discussion

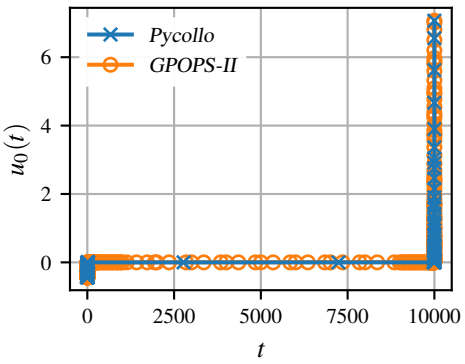
The solutions obtained using *Pycollo* and *GPOPS-II*, 3.3620572 and 3.3620563 respectively, were in very close agreement with one another, as well as with the value

```
import numpy as np
from pycollo import OptimalControlProblem

y, u = symbols("y u")

problem = OptimalControlProblem(name="Hypersensitive problem")
phase = problem.new_phase(name="A")
phase.state_variables = y
phase.control_variables = u
phase.state_equations = {y: -y**3 + u}
phase.integrand_functions = 0.5*(y**2 + u**2)
q = phase.integral_variables[0]

phase.bounds.initial_time = 0.0
phase.bounds.final_time = 10000.0
phase.bounds.state_variables = {y: [-50, 50]}
phase.bounds.control_variables = {u: [-50, 50]}
phase.bounds.integral_variables = {q: [0, 100000]}
phase.bounds.initial_state_constraints = {y: 1.0}
phase.bounds.final_state_constraints = {y: 1.5}

phase.guess.time = [0.0, 10000.0]
phase.guess.state_variables = {y: [1.0, 1.5]}
phase.guess.control_variables = {u: [0.0, 0.0]}
phase.guess.integral_variables = {q: 4}

problem.objective_function = q

problem.solve()
```

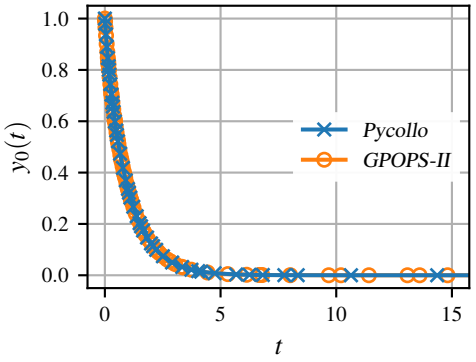
Figure 2.5: Code example of defining and solving a hypersentive problem (section 2.7.4) using the *Pycollo* API.



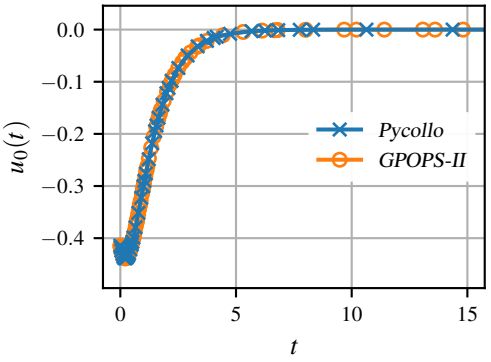
(a) State across time domain



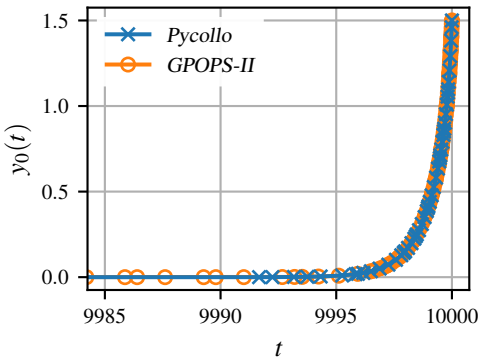
(b) Control across time domain



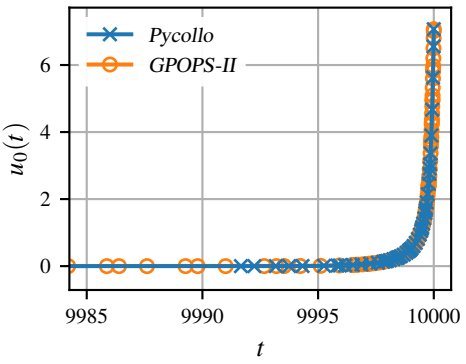
(c) State near initial time



(d) Control near initial time



(e) State near final time



(f) Control near final time

Figure 2.6: Comparison of the optimal states and controls for example 1: hypersensitive problem obtained using *Pycollo* and *GPOPS-II*.

	Pycollo		*GPOPS-II*	
I	$e_{\max}$	N	$e_{\max}$	N
1	9.833×10^{3}	31	2.827×10^{1}	31
2	2.325×10^{1}	181	4.090	69
3	1.320	661	6.066×10^{-1}	111
4	2.336×10^{-1}	243	1.260×10^{-1}	150
5	2.324×10^{-2}	86	6.762×10^{-3}	182
6	5.982×10^{-4}	67	1.256×10^{-4}	247
7	1.875×10^{-5}	90	9.070×10^{-7}	302
8	3.047×10^{-8}	101	7.035×10^{-8}	310

Table 2.7: Mesh refinement performance comparison between *Pycollo* and *GPOPS-II* for example 1: hypersensitive problem. I denotes the mesh iteration count, $e_{\max}$ denotes the maximum relative mesh error, and N denotes the number of discretisation nodes used.

of 3.3620608 quoted in the literature [36]. Figure 2.6 shows the comparison of the optimal solutions obtained for the state and control by *Pycollo* and *GPOPS-II*. As the interesting features of the solution occur close to the time endpoints, the optimal state and control near the initial time (for $t \in [0, 15]$) are shown in figs. 2.6c and 2.6d respectively while the optimal state and control near the final time (for $t \in [9985, 10000]$) are shown in figs. 2.6e and 2.6f respectively. Apparent exact agreement between the solutions obtained by both software packages can be seen. This, in conjunction with the near exact agreement of the optimal objective evaluations, gives certainty that *Pycollo* was able to solve this OCP correctly.

Table 2.7 and fig. 2.7 show that both *Pycollo* and *GPOPS-II* required eight mesh iterations before the mesh tolerance was met, highlighting close algorithmic performance between the two software packages. The distribution of discretisation nodes on the first mesh iteration, shown in fig. 2.7, illustrates the differences between the Lobatto-based collocation scheme employed by *Pycollo* and the Radau-based collocation scheme used by *GPOPS-II*. As the mesh refinement process progresses, the way that the mesh is refined by both software packages differs considerably. *Pycollo* continually refines the mesh in the central portion of the solution at every mesh iteration so that by the final mesh iteration a highly efficient distribution of discretisation nodes is used. *GPOPS-II*, on the other hand, leaves the central 80% of the mesh untouched between the first and last mesh iterations, suggesting that there may be some algorithmic inefficiency here. The main reason that *Pycollo*

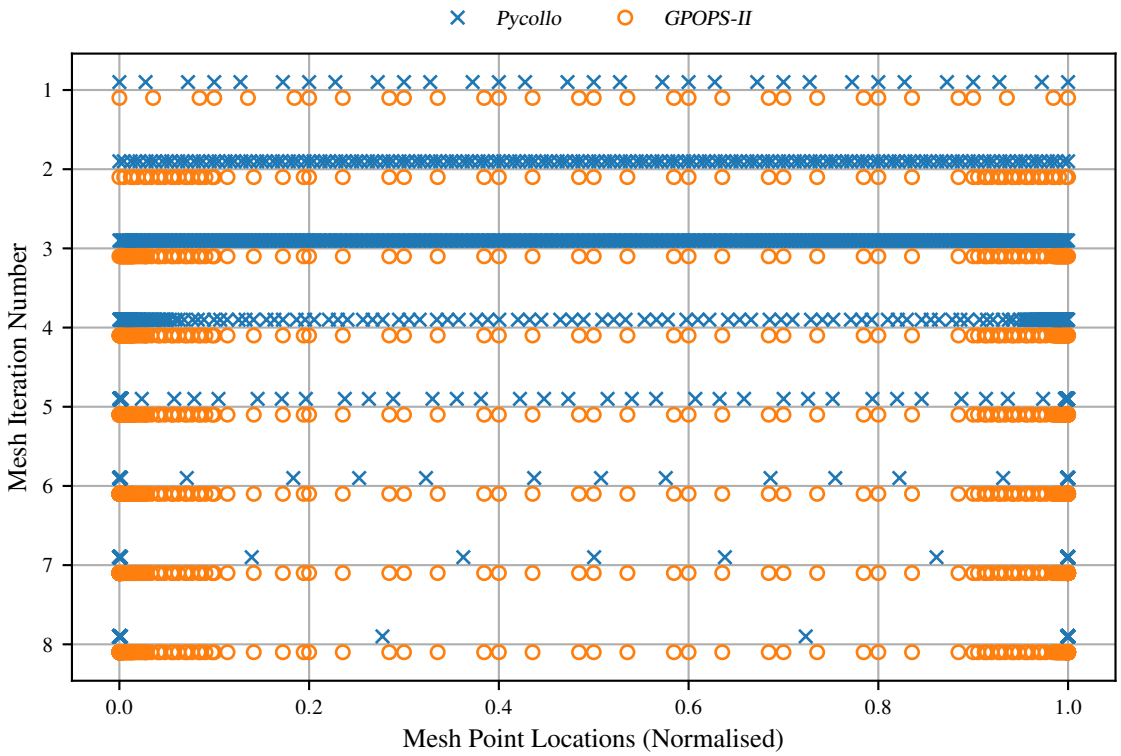


Figure 2.7: Comparison of mesh refinement during the solving of example 1: hyper-sensitive problem by *Pycollo* and *GPOPS-II*.

aggressively increases the mesh density in the central region of the domain during the first three mesh iterations, before continually reducing it across the final five, is due to the Lobatto-based collocation scheme used by *Pycollo*. As described earlier, the Lobatto-based scheme enforces continuity in the state across the entire time domain due to a two-sided collocation condition, which in this case propagates oscillations present at the time endpoints to the centre of the domain on the initial meshes. As *Pycollo*'s mesh refinement algorithm reduces the mesh error at the endpoint regions, the high concentration of central discretisation nodes is no longer needed and so these can be aggressively removed, which *Pycollo* does.

Table 2.7 shows that despite both software packages requiring eight mesh iterations to meet the mesh tolerance, on average *Pycollo* used fewer discretisation points. The exception is on the third mesh iteration where *Pycollo* used twice as many discretisation points than *GPOPS-II* did on its most dense (final) mesh. However, despite this *Pycollo* used significantly fewer discretisation points for each NLP subproblem on average, indicating good performance of its transcription and mesh refinement algorithms. Table 2.7 also shows how the mesh refinement employed by *Pycollo* steadily reduces the estimated mesh error at each mesh iteration until the specified mesh tolerance is met, indicating the algorithms efficient operation across a range of different estimated mesh errors.

Computational performance on this example problem was very comparable between the two software packages. *GPOPS-II* solved the problem in 1.59 s, slightly outperforming *Pycollo*, which took 1.68 s. Finally, this problem was formulated in *Pycollo* using only 23 LLOC while the *GPOPS-II* implementation required 31. As these numbers are small by LLOC standards, this indicates that the demands placed on the user to implement this OCP in both software packages are broadly equivalent.

2.7.5 Example 2: Space Shuttle Reentry Trajectory

Problem Definition

The space shuttle reentry trajectory for maximum crossrange problem (originally in [349] and presented as ex. (6.1) in [36]) was chosen as it is an exemplar of the importance of good scaling. This is because the state involves quantities that differ by six orders of magnitude at the solution. The single-phase problem involving the state

$$\mathbf{y} = \begin{bmatrix} h & \phi & \theta & \nu & \gamma & \psi \end{bmatrix}^T \tag{2.113}$$

and control

$$\mathbf{u} = \begin{bmatrix} \alpha & \beta \end{bmatrix}^T \tag{2.114}$$

is to maximise

$$J = \theta\left(t_F\right) \tag{2.115}$$

subject to the dynamical constraints

$$\dot{h} = \nu \sin\left(\gamma\right) \tag{2.116}$$

$$\dot{\phi} = \frac{\nu \cos\left(\gamma\right) \sin\left(\psi\right)}{r \cos\left(\theta\right)} \tag{2.117}$$

$$\dot{\theta} = \frac{\nu \cos\left(\gamma\right) \cos\left(\psi\right)}{r} \tag{2.118}$$

$$\dot{\nu} = -\frac{D}{m} - g \sin\left(\gamma\right) \tag{2.119}$$

$$\dot{\gamma} = \frac{L \cos\left(\beta\right)}{m\nu} + \cos\left(\gamma\right)\left(\frac{\nu}{r} - \frac{g}{\nu}\right) \tag{2.120}$$

$$\dot{\psi} = \frac{L \sin\left(\beta\right)}{m\nu \cos\left(\gamma\right)} + \frac{\nu \cos\left(\gamma\right) \sin\left(\psi\right) \sin\left(\theta\right)}{r \cos\left(\theta\right)} \tag{2.121}$$

and state endpoint constraints

$$\begin{aligned} h\left(t_0\right) &= 79\,248\,\mathrm{m} & \nu\left(t_0\right) &= 7802.88\,\mathrm{m\,s^{-1}} & h\left(t_F\right) &= 24\,384\,\mathrm{m} \\ \phi\left(t_0\right) &= 0^\circ & \gamma\left(t_0\right) &= -1^\circ & \nu\left(t_F\right) &= 762\,\mathrm{m\,s^{-1}} \\ \theta\left(t_0\right) &= 0^\circ & \psi\left(t_0\right) &= 90^\circ & \gamma\left(t_F\right) &= -5^\circ \end{aligned}$$

where the bounds

$$\begin{aligned} t_0 &= 0 & 0 \leq t_F &\leq 3000\,\mathrm{s} \\ 0 &\leq h & -89^\circ \leq \theta &\leq 89^\circ \\ 1\,\mathrm{m\,s^{-1}} &\leq \nu & -89^\circ \leq \gamma &\leq 89^\circ \\ -90^\circ \leq \alpha &\leq 90^\circ & -89^\circ \leq \beta &\leq 89^\circ \end{aligned}$$

are enforced and the auxiliary substitutions, and aerodynamic and atmospheric constants

$$\begin{aligned} \rho_0 &= 1.2256\,\mathrm{kg\,m^{-3}} & \mu &= 3.9860 \times 10^{14}\,\mathrm{m^3\,s^{-2}} \\ h_r &= 7254.2\,\mathrm{m} & D &= \frac{1}{2} c_D S \rho \nu^2 \\ R_e &= 6\,371\,200\,\mathrm{m} & L &= \frac{1}{2} c_L S \rho \nu^2 \\ S &= 249.91\,\mathrm{m^2} & g &= \frac{\mu}{r^2} \\ c_{L0} &= -0.2070 & r &= R_e + h \\ c_{L1} &= 1.6756 & \rho &= \rho_0 \exp\left(-\frac{h}{h_r}\right) \\ c_{D0} &= 0.07854 & C_L &= c_{L0} + c_{L1}\alpha \\ c_{D1} &= -0.3529 & C_D &= c_{D0} + c_{D1}\alpha + c_{D2}\alpha^2 \\ c_{D2} &= 2.0400 & m &= 92\,079\,\mathrm{kg} \end{aligned}$$

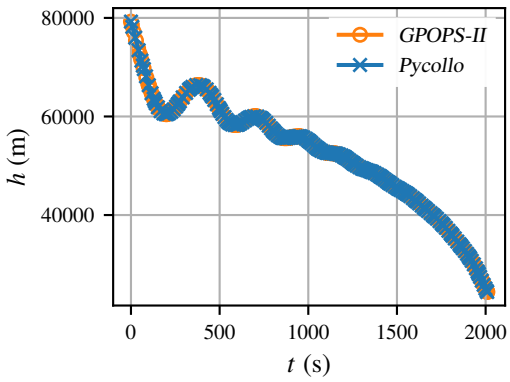
are used.

Results

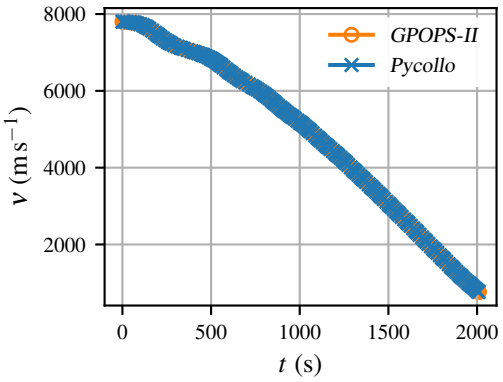
The optimal costs obtained using *Pycollo* and *GPOPS-II* were 34.1501° and 34.1641° respectively. These closely agreed with each other, and with the value of 34.1412° reported in [36]. A comparison of the optimal state and control obtained by the two software packages is shown in fig. 2.8. Close agreement in the optimal costs, states and controls show that *Pycollo* is able to correctly solve this OCP.

The mesh refinement performance of *Pycollo* and *GPOPS-II* is compared in table 2.8. *Pycollo* required three mesh iterations to meet the mesh tolerance, one fewer than *GPOPS-II*. After the first mesh iteration, *Pycollo* produced a lower mesh error using fewer mesh nodes. This suggests that the mesh refinement algorithm described in section 2.5, and implemented in *Pycollo*, is more efficient than the *hp*-method implemented in *GPOPS-II* [262] for this problem.

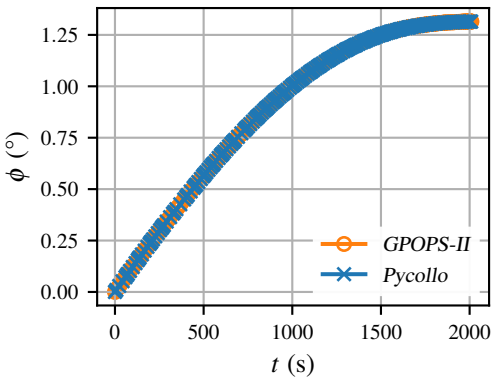
The *Pycollo* formulation of this OCP involved 68 LLOC compared to 135,



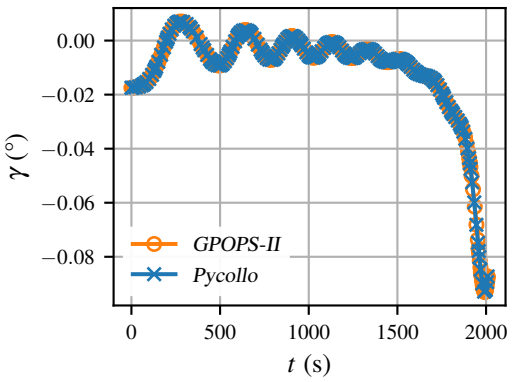
(a) Altitude



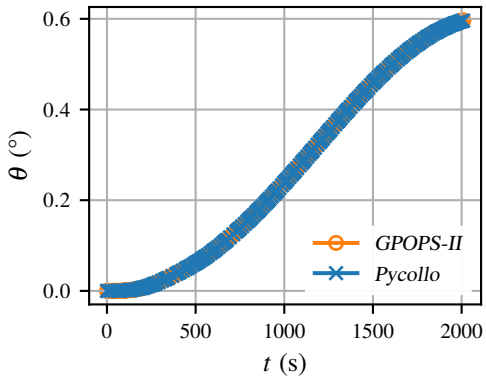
(b) Velocity



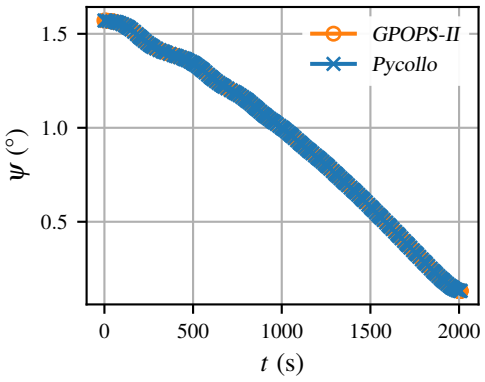
(c) Longitude



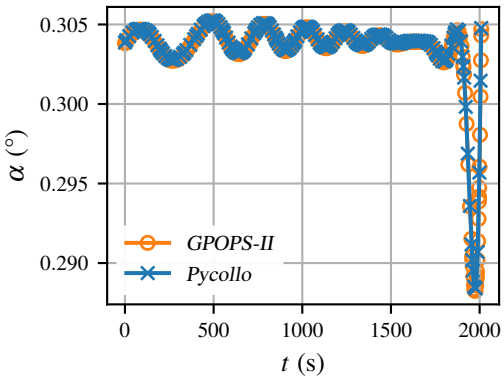
(d) Latitude



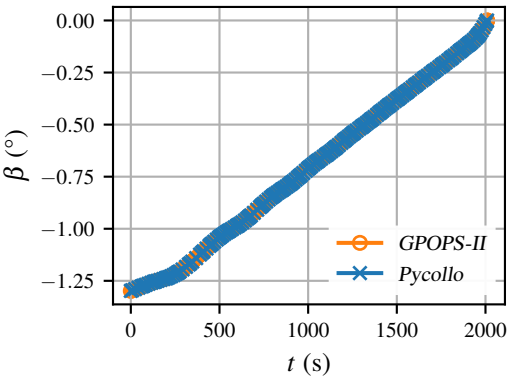
(e) Flight path



(f) Azimuth



(g) Angle of attack



(h) Bank angle

Figure 2.8: Comparison of the optimal states and controls to example 2: space shuttle reentry trajectory obtained using *Pycollo* and *GPOPS-II*.

	Pycollo		*GPOPS-II*	
I	$e_{\max}$	N	$e_{\max}$	N
1	1.026×10^{-1}	31	1.152×10^{-3}	31
2	2.008×10^{-5}	113	2.946×10^{-4}	121
3	4.313×10^{-8}	206	2.742×10^{-6}	282
4			2.496×10^{-8}	295

Table 2.8: Mesh refinement performance comparison between *Pycollo* and *GPOPS-II* for example 2: space shuttle reentry trajectory. I denotes the mesh iteration count, $e_{\max}$ denotes the maximum relative mesh error, and N denotes the number of discretisation nodes used.

almost twice as many, in *GPOPS-II*. *GPOPS-II* solved this OCP faster than *Pycollo*, requiring 2.78 s to converge to the optimal solution, in comparison to 3.64 s.

2.7.6 Example 3: Space Station Attitude Control

Problem Definition

The space station attitude control problem (originally in [267] and presented as ex. (6.7) in [36]) was chosen as it is contains complex *three-dimensional* (3D) dynamics expressed using vector and matrix quantities. The single-phase problem involving the state

$$\mathbf{y} = \begin{bmatrix} \boldsymbol{\omega}^T & \mathbf{r}^T & \mathbf{h}^T \end{bmatrix}^T = \begin{bmatrix} \omega_x & \omega_y & \omega_z & r_x & r_y & r_z & h_x & h_y & h_z \end{bmatrix}^T \tag{2.122}$$

and control

$$\mathbf{u} = \begin{bmatrix} u_x & u_y & u_z \end{bmatrix}^T \tag{2.123}$$

is to maximise

$$J = q \tag{2.124}$$

subject to the dynamical constraints

$$\dot{\boldsymbol{\omega}} = \mathbf{J}^{-1} \left[3\omega_{orb}^2 \mathbf{C}_3^{\otimes} \mathbf{J} \mathbf{C}_3 - \omega^{\otimes} \left[\mathbf{J}\boldsymbol{\omega} + \mathbf{h} \right] - \mathbf{u} \right] \tag{2.125}$$

$$\dot{\mathbf{r}} = \frac{1}{2} \left[\mathbf{r}\mathbf{r}^T + \mathbf{I} + \mathbf{r}^{\otimes} \right] \left[\boldsymbol{\omega} + \omega_{orb} \mathbf{C}_2 \right] \tag{2.126}$$

$$\dot{\mathbf{h}} = \mathbf{u}\,, \tag{2.127}$$

inequality path constraint

$$0 \leq h_x^2 + h_y^2 + h_z^2 \leq h_{max}^2 \tag{2.128}$$

and state endpoint constraints

$$\boldsymbol{\omega}(t_0) = \begin{bmatrix} -9.538 \times 10^{-6} \\ -1.136 \times 10^{-3} \\ 5.347 \times 10^{-6} \end{bmatrix} \qquad \mathbf{r}(t_0) = \begin{bmatrix} 2.996 \times 10^{-3} \\ 1.533 \times 10^{-1} \\ 3.836 \times 10^{-3} \end{bmatrix} \qquad \mathbf{h}(t_0) = \begin{bmatrix} 5000 \\ 5000 \\ 5000 \end{bmatrix}$$

$$\dot{\boldsymbol{\omega}}(t_F) = \mathbf{0}$$
$$\dot{\mathbf{r}}(t_F) = \mathbf{0}$$

where the integral variable

$$q = 10^{-6} \int_{t_0}^{t_F} u_x^2 + u_y^2 + u_z^2 dt\,, \tag{2.129}$$

the times are equality bounded

$$t_0 = 0 \qquad\qquad t_F = 1800\,\mathrm{s}\,,$$

the skew-symmetric cross product operator $\mathbf{a}^{\otimes}$ is defined as

$$\mathbf{a}^{\otimes} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix},$$

$\mathbf{C}_3$ is the third column of the rotation matrix

$$\mathbf{C} = \mathbf{I} + \frac{2\left(\mathbf{r}^{\otimes}\mathbf{r}^{\otimes} - \mathbf{r}^{\otimes}\right)}{1 + \mathbf{r}^T\mathbf{r}}\,,$$

and the constants

$$h_{max} = 10000 \qquad\qquad \omega_{orb} = \frac{0.06511\pi}{180}$$

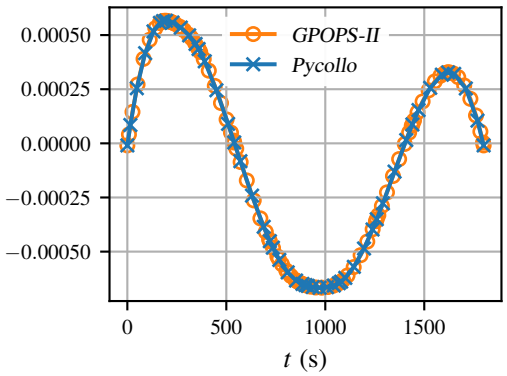
and

$$\mathbf{J} = \begin{bmatrix} 2.807 \times 10^7 & 4.823 \times 10^5 & -1.717 \times 10^7 \\ 4.823 \times 10^5 & 9.514 \times 10^7 & 6.026 \times 10^4 \\ -1.717 \times 10^7 & 6.026 \times 10^4 & 7.659 \times 10^7 \end{bmatrix}.$$

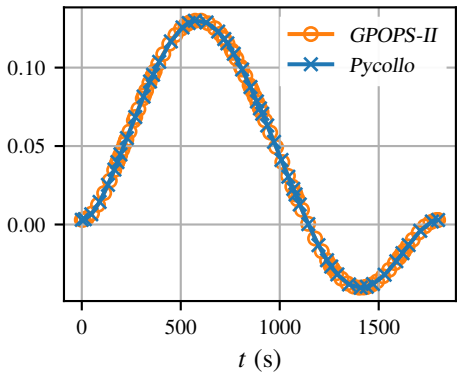
Note that many quantities in the problem definition are without units as specific units are not provided in the original reference [36, 267].

Results

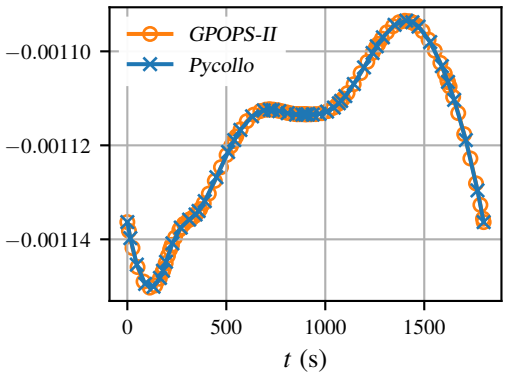
Optimal costs of 3.58679 and 3.58688 were obtained using *Pycollo* and *GPOPS-II* respectively. The optimal cost obtained by *GPOPS-II* agreed exactly with that



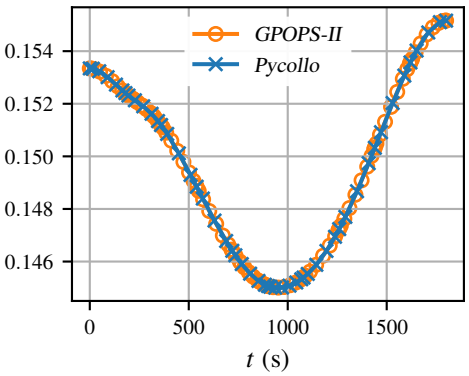
(a) Angular velocity (ω_x)



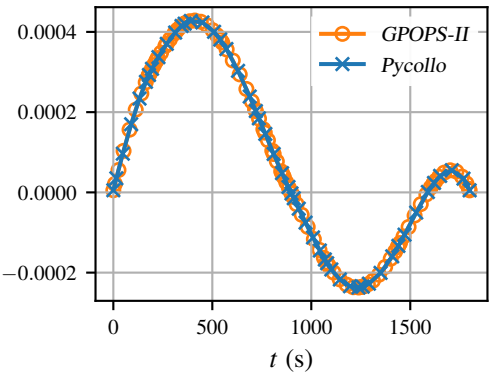
(b) Attitude (r_x)



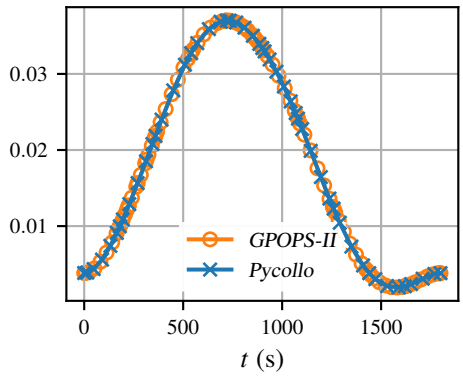
(c) Angular velocity (ω_y)



(d) Attitude (r_y)

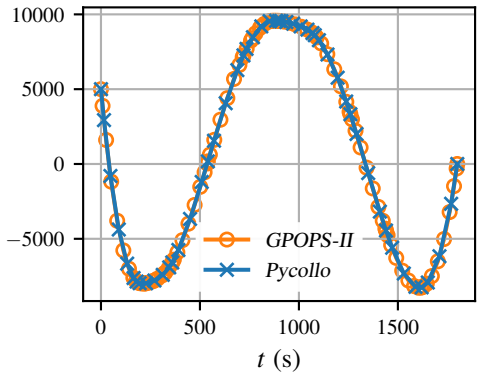


(e) Angular velocity (ω_z)

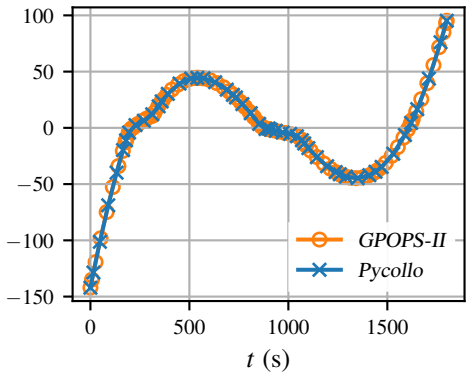


(f) Attitude (r_z)

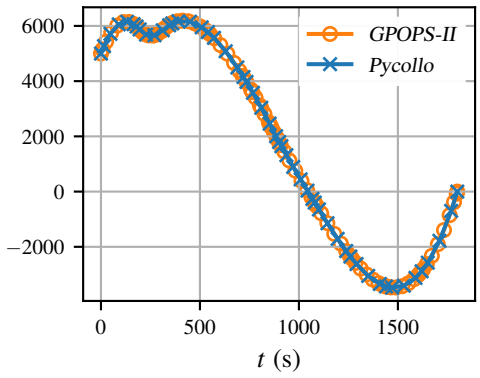
Figure 2.9: Comparison of the optimal spacecraft angular velocity ($\boldsymbol{\omega}$) and attitude ($\mathbf{r}$) states to example 3: space station attitude control obtained using *Pycollo* and *GPOPS-II*.



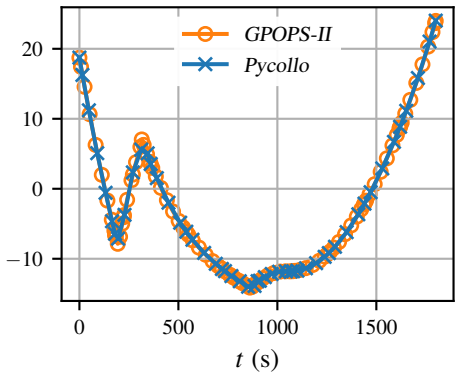
(a) CMG angular momentum (h_x)



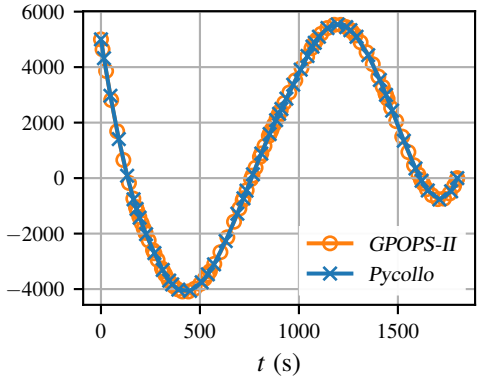
(b) Torque (u_x)



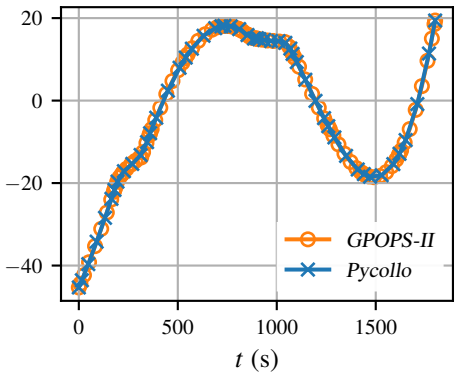
(c) CMG angular momentum (h_y)



(d) Torque (u_y)



(e) CMG angular momentum (h_z)



(f) Torque (u_z)

Figure 2.10: Comparison of the optimal spacecraft *control moment gyroscopes* (CMGs) angular momentum (**h**) states and control torque (**u**) control to example 3: space station attitude control obtained using *Pycollo* and *GPOPS-II*.

	Pycollo		*GPOPS-II*	
I	$e_{\max}$	N	$e_{\max}$	N
1	2.718×10^{-6}	31	3.362×10^{-5}	31
2	1.769×10^{-7}	48	3.919×10^{-7}	86
3	6.796×10^{-8}	51	1.007×10^{-6}	91
4			1.816×10^{-7}	96
5			2.971×10^{-7}	99
6			1.018×10^{-7}	100
7			8.854×10^{-8}	101

Table 2.9: Mesh refinement performance comparison between *Pycollo* and *GPOPS-II* for example 3: space station attitude control. I denotes the mesh iteration count, $e_{\max}$ denotes the maximum relative mesh error, and N denotes the number of discretisation nodes used.

reported in [36], with that obtained by *Pycollo* also being close. A comparison of the optimal state and control obtained by the two software packages is shown in figs. 2.9 and 2.10. Close agreement in the optimal costs, states and controls shows that *Pycollo* is able to correctly solve this OCP.

The mesh refinement performance of *Pycollo* and *GPOPS-II* is compared in table 2.9. *Pycollo* required three mesh iterations to meet the mesh tolerance, while *GPOPS-II* required seven. It was also able to meet the mesh tolerance using half as many mesh nodes as required by *GPOPS-II*. *GPOPS-II* was close to meeting the mesh tolerance after the second mesh iteration, but then required five further mesh refinements, with limited mesh adjustment, to converge fully. In contrast, *Pycollo* converged successfully on the next mesh iteration after getting within an order of magnitude of the mesh tolerance.

Again, formulating the OCP required fewer LLOC in *Pycollo* (98) than in *GPOPS-II* (179). *GPOPS-II* was also marginally faster at solving the OCP than *Pycollo*, requiring 6.09 s compared to 6.79 s.

2.7.7 Example 4: Free-Flying Robot

Problem Definition

The free-flying robot problem (originally in [294] and presented as ex. (6.13) in [36]) was chosen as the optimal solution exhibits bang-bang control. As a consequence of this, the optimal state involves discontinuities. This is, therefore, a good test for mesh refinement algorithms, which must place a high density of nodes at the locations where the optimal control switches. The single-phase problem involving the state

$$\mathbf{y} = \begin{bmatrix} x_1 & x_2 & \theta & v_1 & v_2 & \omega \end{bmatrix}^T \tag{2.130}$$

and control

$$\mathbf{u} = \begin{bmatrix} u_1 & u_2 & u_3 & u_4 \end{bmatrix}^T \tag{2.131}$$

is to maximise

$$J = q \tag{2.132}$$

subject to the dynamical constraints

$$\dot{x}_1 = v_1 \tag{2.133}$$
$$\dot{x}_2 = v_2 \tag{2.134}$$
$$\dot{\theta} = \omega \tag{2.135}$$
$$\dot{v}_1 = (u_1 - u_2 + u_3 - u_4)\cos(\theta) \tag{2.136}$$
$$\dot{v}_2 = (u_1 - u_2 + u_3 - u_4)\sin(\theta) \tag{2.137}$$
$$\dot{\omega} = \alpha(u_1 - u_2) - \beta(u_3 - u_4)\ , \tag{2.138}$$

inequality path constraints

$$u_1 + u_2 \leq 1 \qquad\qquad u_3 + u_4 \leq 1 \tag{2.139}$$

and state endpoint constraints

$$\begin{aligned} x_1(t_0) &= -10 & x_1(t_F) &= 0 \\ x_2(t_0) &= -10 & x_2(t_F) &= 0 \\ \theta(t_0) &= \frac{\pi}{2} & \theta(t_F) &= 0 \\ v_1(t_0) &= 0 & v_1(t_F) &= 0 \\ v_2(t_0) &= 0 & v_2(t_F) &= 0 \\ \omega(t_0) &= 0 & \omega(t_F) &= 0\,, \end{aligned}$$

where the integral variable

$$q = \int_{t_0}^{t_F} u_1 + u_2 + u_3 + u_4 dt \tag{2.140}$$

and the constants

$$t_0 = 0 \qquad t_F = 12.0\,\mathrm{s}$$
$$\alpha = 0.2 \qquad \beta = 0.2$$

are used.

Results

The optimal costs obtained using *Pycollo* and *GPOPS-II* were 7.90960 and 7.90997 respectively. These both agreed closely with each other, and with the value of 7.9102 reported in [36]. A comparison of the optimal state and control obtained by the two software packages is shown in figs. 2.11 and 2.12. Close agreement in the optimal cost, state and control show that *Pycollo* is able to correctly solve this OCP.

The mesh refinement performance of *Pycollo* and *GPOPS-II* is compared in table 2.10. *GPOPS-II* required 16 mesh iterations, fewer than the 22 required by *Pycollo*. Both software packages required a similar number of nodes to meet the mesh tolerance, with *Pycollo* requiring 1404 and *GPOPS-II* requiring 1326. Mesh refinement was gradual in both cases, with neither software package performing well due to the fact that many mesh iterations were still required once getting within an order of magnitude of the mesh tolerance (eight in *GPOPS-II*'s case and 14 in *Pycollo*'s case).

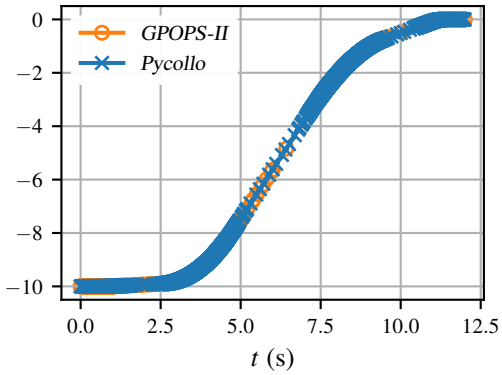
The *Pycollo* formulation of this OCP involved 36 LLOC compared to 104, almost three times as many, in *GPOPS-II*. Neither software package was able to solve this OCP particularly quickly due to the excessive number of mesh refinement iterations required. Solve times of 89.2 s for *GPOPS-II* and 532.3 s for *Pycollo* were recorded. *GPOPS-II* was significantly faster than *Pycollo*. This was due in majority to the large and increasing mesh initialisation times recorded by *Pycollo* as the number of mesh nodes increased in later mesh iterations.

2.7.8 Example 5: Tumour Anti-Angiogenesis

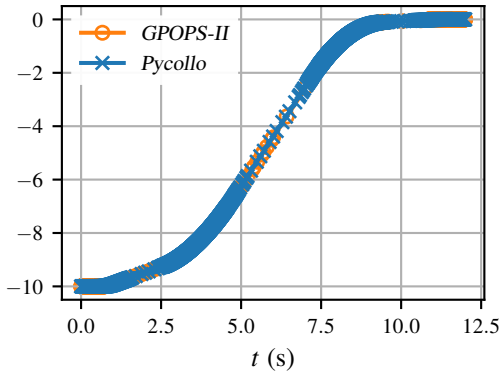
Problem Definition

The tumour anti-angiogenesis problem (originally in [211] and presented as ex. (6.17) in [36]) was chosen as it is an example of a multiphase OCP involving the state

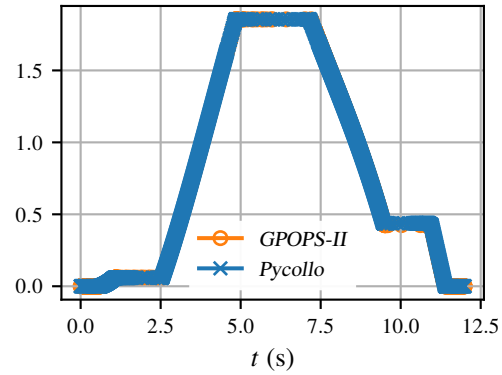
$$\mathbf{y}^{(p)} = \begin{bmatrix} v^{(p)} & c^{(p)} \end{bmatrix}^T \tag{2.141}$$



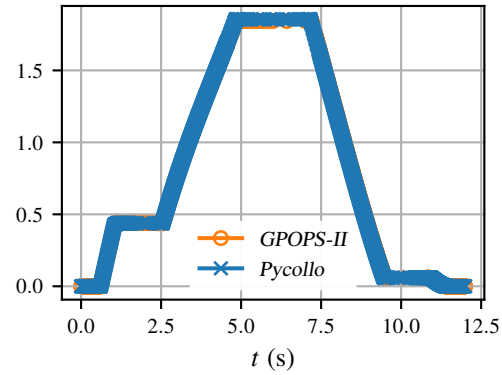
(a) Position (x_1)



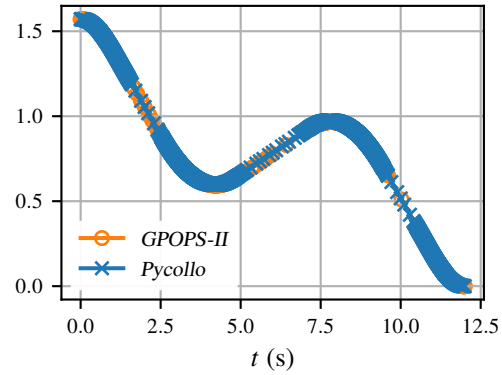
(b) Position (x_2)



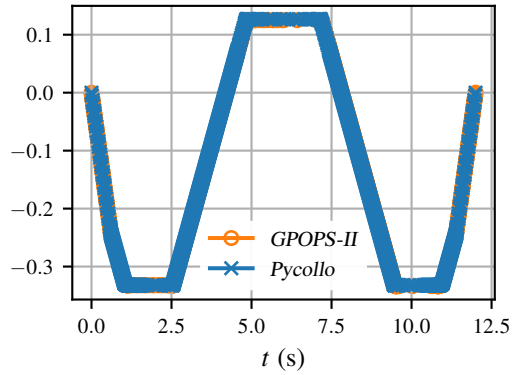
(c) Velocity (v_1)



(d) Velocity (v_2)

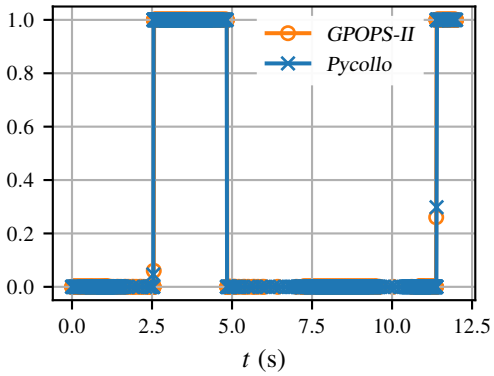


(e) Angle (θ)

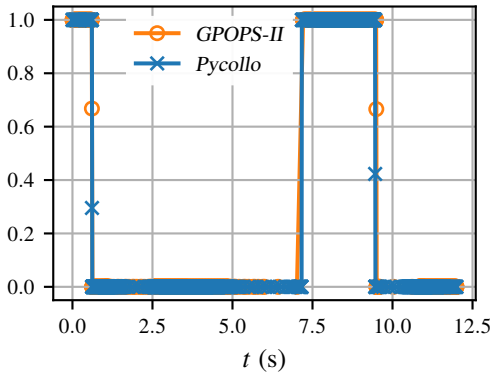


(f) Angular velocity (ω)

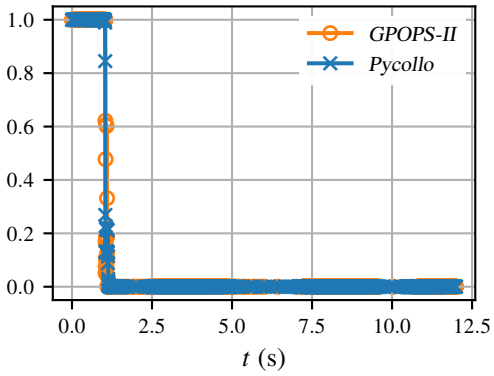
Figure 2.11: Comparison of the optimal state to example 4: free-flying robot obtained using *Pycollo* and *GPOPS-II*.



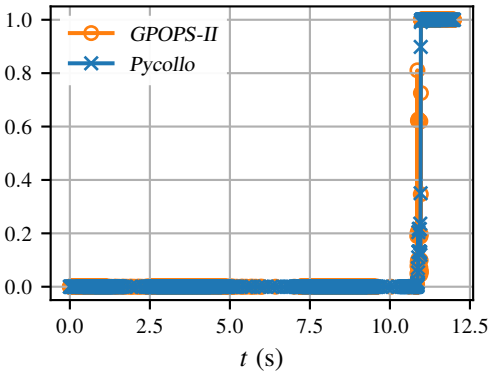
(a) Control (u_1)



(b) Control (u_2)



(c) Control (u_3)



(d) Control (u_4)

Figure 2.12: Comparison of the optimal control to example 4: free-flying robot obtained using *Pycollo* and *GPOPS-II*.

	Pycollo		*GPOPS-II*	
I	$e_{\max}$	N	$e_{\max}$	N
1	2.646×10^{-3}	31	2.524×10^{-3}	31
2	1.845×10^{-4}	84	1.098×10^{-3}	97
3	2.915×10^{-5}	156	1.129×10^{-4}	172
4	1.424×10^{-5}	222	1.043×10^{-5}	244
5	7.015×10^{-6}	279	8.462×10^{-6}	311
6	2.995×10^{-6}	355	1.696×10^{-6}	387
7	1.317×10^{-6}	429	1.057×10^{-6}	451
8	6.674×10^{-7}	501	5.330×10^{-7}	539
9	5.250×10^{-7}	581	3.497×10^{-7}	656
10	5.250×10^{-7}	663	5.440×10^{-7}	765
11	5.244×10^{-7}	755	4.909×10^{-7}	868
12	5.238×10^{-7}	875	2.621×10^{-7}	971
13	5.232×10^{-7}	983	2.249×10^{-7}	1085
14	4.398×10^{-7}	1090	2.249×10^{-7}	1161
15	2.362×10^{-7}	1188	2.178×10^{-7}	1254
16	1.739×10^{-7}	1282	9.370×10^{-8}	1326
17	1.735×10^{-7}	1358		
18	1.730×10^{-7}	1372		
19	1.726×10^{-7}	1386		
20	1.715×10^{-7}	1397		
21	1.699×10^{-7}	1402		
22	9.919×10^{-8}	1404		

Table 2.10: Mesh refinement performance comparison between *Pycollo* and *GPOPS-II* for example 4: free-flying robot. I denotes the mesh iteration count, $e_{\max}$ denotes the maximum relative mesh error, and N denotes the number of discretisation nodes used.

in phases $p = 1$ and $p = 2$, and control

$$\mathbf{u}^{(1)} = \left[u^{(1)}\right]^T \tag{2.142}$$

in phase $p = 1$ only. The objective is to maximise

$$J = q^{(1)} \tag{2.143}$$

subject to the dynamical constraints

$$\dot{v}^{(p)} = -\xi v^{(p)} \log\left(\frac{v^{(p)}}{c^{(p)}}\right) \tag{2.144}$$

$$\dot{c}^{(1)} = c^{(1)}\left[b - \left(\mu + d\left(v^{(1)}\right)^{\frac{2}{3}} + Gu^{(1)}\right)\right] \tag{2.145}$$

$$\dot{c}^{(2)} = c^{(2)}\left[b - \left(\mu + d\left(v^{(2)}\right)^{\frac{2}{3}}\right)\right], \tag{2.146}$$

and state endpoint constraints

$$v^{(1)}\left(t_0^{(1)}\right) = \frac{v_{max}}{2} \qquad\qquad c^{(1)}\left(t_0^{(1)}\right) = \frac{c_{max}}{4},$$

where the integral variable

$$q^{(1)} = \int_{t_0^{(1)}}^{t_F^{(1)}} u^{(1)} dt \tag{2.147}$$

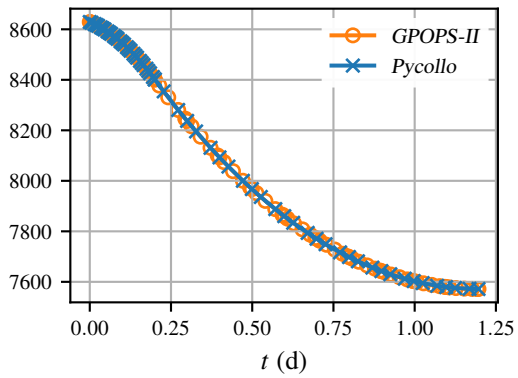
and the constants

$$\begin{aligned}
t_0 &= 0 & t_F &= 12.0\,\mathrm{s} \\
\alpha &= 0.2 & \beta &= 0.2 \\
\xi &= 0.084\,\mathrm{d}^{-1} & b &= 5.85\,\mathrm{d}^{-1} \\
d &= 0.008\,73\,\mathrm{mm}^{-2}\,\mathrm{d}^{-1} & G &= 0.15\,\mathrm{kg\,mg}^{-1}\,\mathrm{d}^{-1} \\
\mu &= 0.02\,\mathrm{d}^{-1} & v_{max} = c_{max} &= \left(\frac{b-\mu}{d}\right)^{\frac{3}{2}} \\
a &= 75 & A &= 15
\end{aligned}$$

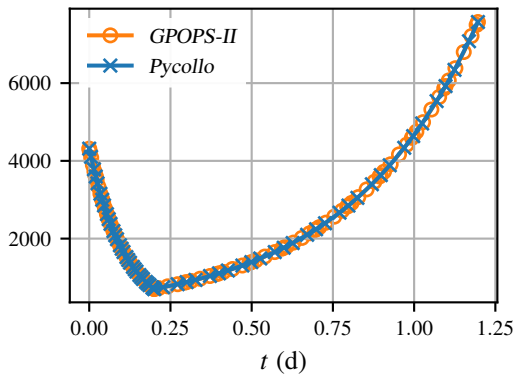
are used.

Results

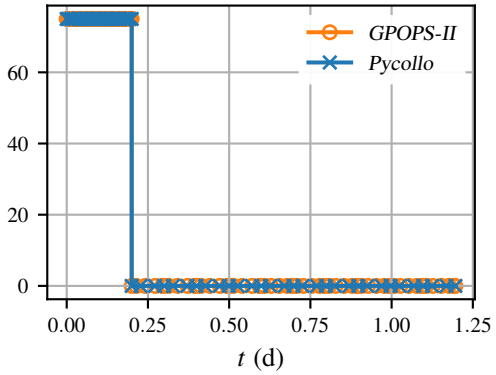
The optimal costs obtained using *Pycollo* and *GPOPS-II* agreed almost exactly, with values of 7571.6701 and 7571.6700 respectively. The optimal cost achieved using *Pycollo* was identical to the two-phase solution reported in [36] of 7571.6701. A comparison of the optimal state and control obtained by the two software packages is shown in fig. 2.13. Close agreement in the optimal costs, states and controls show that *Pycollo* is able to correctly solve this OCP.



(a) Tumour volume (v)



(b) Vascular capacity (c)



(c) Angiogenic dose rate (u)

Figure 2.13: Comparison of the optimal states and controls to example 5: tumour anti-angiogenesis obtained using *Pycollo* and *GPOPS-II*.

	Pycollo		*GPOPS-II*	
I	$e_{\max}$	N	$e_{\max}$	N
1	8.268×10^{-7}	62	2.694×10^{-6}	62
2	5.531×10^{-8}	78	2.675×10^{-9}	108

Table 2.11: Mesh refinement performance comparison between *Pycollo* and *GPOPS-II* for example 5: tumour anti-angiogenesis. I denotes the mesh iteration count, $e_{\max}$ denotes the maximum relative mesh error, and N denotes the number of discretisation nodes used.

The mesh refinement performance of *Pycollo* and *GPOPS-II* is compared in table 2.11. Both software packages required two mesh iterations to meet the mesh tolerance, with *Pycollo* using fewer nodes in the final mesh (78 compared to 108).

This OCP required 42 LLOC to be formulated in *Pycollo* compared to 69 in *GPOPS-II*. Solve time was also slightly faster in *Pycollo* at 0.89 s compared to 1.04 s in *GPOPS-II*.

2.8 Discussion

The literature review (section 2.1) highlighted a number of major limitations of current software for solving OCPs. These included: limitations to the choice of collocation scheme available to the user; no published framework for automatic problem scaling that can be further developed and enhanced by other researchers; and no suitable mesh refinement algorithm for use with LGL collocation.

In addition, at present, there is no easy-to-use, highly-performant, open-source software package for solving OCPs. The closest software package to meeting these requirements is *GPOPS-II*. However, as proprietary software, it does not lend itself to forming part of a wider software framework (like the BPST). The implementation of *Pycollo* sought to address these requirements and current limitations, and demonstrate its functionality through robust validation.

2.8.1 Performance

Optimal costs, states and controls obtained using *Pycollo* and *GPOPS-II* were in close agreement for all five of the test OCPs. In addition, close agreement to the previously published solutions [36, 211, 267, 284, 294, 349] was also demonstrated. This confirms that *Pycollo* is able to accurately and reliably solve a range of different OCPs, and validates the software package.

The ability of *Pycollo* to reliably solve the five test OCPs, including two that involve states spanning multiple orders of magnitude (examples 2 and 3), indicates robust performance of the automatic problem scaling approach outlined in section 2.4.4 and implemented in *Pycollo*. This approach has been shown to deliver results at least equivalent to those generated by *GPOPS-II* using its proprietary scaling algorithm. This open-source scaling framework can now be used by researchers, and further investigated and enhanced as required.

The solve times and mesh refinement counts recorded demonstrate that the computational performance of *Pycollo* is broadly comparable to that of *GPOPS-II*. The only example where this was not the case was for the free-flying robot problem (example 4), where *GPOPS-II* was able to solve the OCP significantly faster. *Pycollo*'s longer solve time can be attributed to mesh preprocessing. These preprocessing times are incurred by the *CasADi* backend and involve recomputing the NLP derivatives for each mesh-specific discretisation. This process is inefficient and is addressed as part of chapter 3, where an algorithm is proposed to negate this.

The number of LLOC required to formulate an OCP using *Pycollo* has been shown to be materially less than required by *GPOPS-II*. This is particularly so for more complex OCPs. This has been achieved while meeting the objectives of an easy-to-use and extensive API, as illustrated in fig. 2.5.

2.8.2 Mesh Refinement

The mesh refinement algorithm presented in section 2.5 and implemented in *Pycollo* performed well on the hypersensitive problem (example 1), allowing the mesh tolerance to be met using significantly fewer discretisation nodes than was possible using *GPOPS-II*. This was primarily because *Pycollo*'s algorithm is able to reduce the number of discretisation nodes in regions where the mesh error is sufficiently met. This aspect of the algorithm makes it the first directly suited for use alongside LGL collocation. It will, however, need to be tested with LGR and LG collocation, once these are fully supported by *Pycollo*, to investigate its efficiency alongside these other collocation schemes.

The mesh refinement performance of *Pycollo* was poor when applied to problems with bang-bang control, as highlighted by the free-flying robot test OCP (example 4). A recent publication has presented a modified mesh refinement algorithm in which discontinuities in the dynamics are detected and new phases are introduced with their boundaries falling at these locations [7]. This approach has been shown to be highly-performant on OCPs with bang-bang control. Future work should, therefore, investigate whether the ideas in [7] can be combined with the mesh refinement algorithm developed in section 2.5. This would potentially lead to a mesh refinement algorithm that is performant when solving OCPs with both nonlinearities and discontinuities when used alongside LGL collocation.

2.8.3 Limitations of *Pycollo*

The presentation, analysis and discussion of the benchmark problems highlighted some limitations of *Pycollo*. These include:

1. The solution obtained by *Pycollo* is a local optimum. If the OCP in question has multiple solutions then there is no guarantee that the solution found is the global optimum. Which local optimum *Pycollo* converges to is typically dependent on the choice of the initial guess. Therefore, it is suggested that the problem is solved multiple times using different initial guesses, especially if it is suspected that the solution space is multimodal.

2. *Pycollo*'s automatic scaling algorithm works best when the user supplies appropriate bounds for all of the OCP's variables. If the user does not supply bounds, or supplies bounds that are many orders of magnitude different from the optimal solution, then the resulting problem scaling may be poor, and inefficient convergence or incorrect solutions may result.

3. The NLP solver employed by *Pycollo*, *Ipopt*, expects all functions to be second-order continuous, such that accurate derivatives can be computed to determine the search direction. If the OCP is formulated in a way that the functions are discontinuous then convergence of the NLP subproblem can be adversely affected. This is somewhat addressed by the work of section 3.6, where a second derivative backend to *Pycollo*, which only supports appropriately continuous functions.

4. Mesh refinement is only supported when LGL collocation is used. *Pycollo* uses a *SciPy* module to interpolate the NLP solution as part of the mesh error computation. However, this module does not currently support interpolation involving Radau or Gaussian collocation conditions. In order for this to be addressed, a custom interpolation module will need to be developed for *Pycollo*. This module would replace *SciPy* for this purpose and should be able to compute the mesh error independent of the collocation scheme used.

5. *Pycollo* is intended to be used to solve OCPs whose optimal solutions are continuous for both the control and the first derivative of the state. Problems which exhibit bang-bang control, such as the free-flying robot problem (example 4), do not meet these criteria. While *Pycollo* was able to converge on the correct optimal solution in this case, for other bang-bang OCPs, this may well not be the case. This limitation can be addressed by reformulating the OCP, adding additional phases with their boundaries lying at the control discontinuities. This approach was demonstrated for the tumour anti-angiogenesis

problem (example 5). How the functionality of *Pycollo* could also be extended in the future to support the accurate and efficient solving of this category of OCP is discussed above.

6. *Pycollo* is not currently able to solve OCPs with constraints that are explicit functions of time (e.g. the kinetic batch reactor problem [36]). This motivates the need for the package to be extended to support this type of OCP.

2.9 Conclusions

In this chapter a number of algorithms have been developed to aid the computational solution of OCPs, including:

1. a generalised framework that allows the transcription of an OCP to a NLP using any of Gaussian, Radau or Lobatto collocation schemes, and a description of how this can be implemented algorithmically;
2. a framework for automatically scaling an OCP's transcribed NLP based on user-supplied variable bounds, random sampling of the search space and analysis of the problem's function's derivatives; and
3. an adapted *hp* mesh refinement algorithm that supports decreasing mesh sparsity in regions of the domain where the mesh tolerance is exceeded and is suitable for use alongside Lobatto collocation.

Pycollo, an open-source *Python* package for the computational solution of OCPs using state-of-the-art theory and methods, and the algorithms detailed in this chapter, has been developed. Once an OCP has been formulated using *Pycollo*'s syntax, *Pycollo* handles all onerous tasks on behalf of the user including:

- transcribing the OCP to a NLP subproblem;
- automatically formulating any derivative information required by the NLP solver and compiling these as callable functions;
- interfacing with the NLP solver *Ipopt*;
- automatically scaling the NLP subproblem to improve its numerical conditioning; and
- conducting adaptive mesh refinement to ensure that the OCP is solved sufficiently accurately via the transcription method.

Pycollo also provides additional facilities to aid analysis such as modules for timing and visualisation.

Pycollo was comprehensively benchmarked against the industry-standard commercial software package *GPOPS-II*. This:

- demonstrated *Pycollo*'s ability to accurately solve a range of different OCPs;
- showed the comparable performance of the open-source software package to the established commercial software;
- highlighted key areas where the algorithms and frameworks presented earlier in this chapter provide algorithmic improvements, specifically that the adapted *hp* mesh refinement algorithm offers improved performance over established algorithms when used with LGL collocation;
- showed that LGL collocation and the presented mesh refinement algorithm are not performant on OCPs with bang-bang control; and
- identified some key areas for future work where *Pycollo* can be further developed and improved.

The open-source provision of *Pycollo* should provide researchers in this field with a base platform to continue developing algorithms for the computational solution of OCPs and allow the investigation and solution of novel OCPs easily, efficiently and reliably. In particular, it is recommended that future research investigates:

- applying the presented mesh refinement algorithm to LGR and LG collocation;
- incorporating the latest ideas and developments from bang-bang mesh refinement into the adapted *hp* mesh refinement algorithm; and
- developing mesh error calculation functionality for LGR and LG collocation in *Pycollo*, so that the relative performance of LGL, LGR and LG collocation can be directly compared.

Chapter 3

Derivative Generation

As outlined in chapter 2, the calculation of derivatives, particularly gradients, Jacobians and Hessians, are of significant importance when solving an *optimal control problem* (OCP) via transcription to a *nonlinear programming problem* (NLP). This chapter begins with a review of the main methods currently used in the research field for derivative production, describing where appropriate fundamental underlying concepts, and highlighting the areas in which certain approaches currently excel and where their disadvantages lie. An algorithm named *hybrid-symbolic-algorithmic differentiation* (hSAD), which has been developed as part of this thesis, is presented. The development of hSAD has been motivated by the need for a derivative-taking method that can exploit the known sparsity structure of Jacobians and Hessians produced by the transcription method presented in chapter 2. hSAD and its implementation in *Pycollo* are then benchmarked.

3.1 Background, Theory and Review

3.1.1 Derivatives for Optimal Control Problems

As outlined in chapter 2, modern OCPs are typically solved numerically rather than analytically due to their complexity. In direct collocation, the predominant approach, the continuous-time OCP is transcribed to a finite NLP by approximating the state and control by a set of smooth basis functions, and enforcing the constraints at a set of discrete points [36]. The resulting NLP, which is large and sparse, can then be solved using established software packages [43, 129].

Gradient-based NLP solvers are preferred as they offer superior performance

for this application in comparison to stochastic or global solvers [35, 282]. There are two general categories of gradient-based methods on which NLP solvers are founded: quasi-Newton methods and Newton-methods [105, 282]. The transcribed NLP is defined in terms of an objective function (eq. (2.64)), a set of inequality constraints (eq. (2.65)) and a set of bounded decision variables (eq. (2.66)) [131]. In addition to this, in a quasi-Newton method, the first-order derivatives with respect to the decision variables, the objective gradient and the constraints Jacobian, are also required [36, 131]. A quasi-Newton approximation to the inverse of the Lagrangian Hessian is also required, typically computed using the *Broyden-Fletcher-Goldfarb-Shanno* (BFGS) algorithm [66, 114, 132, 298]. In a Newton method, the true Lagrangian Hessian is instead required [36, 131]. *SNOPT* [129] and *Ipopt* [43] are software packages for solving NLPs that implement a quasi-Newton and a Newton method respectively.

Gradient information is important when solving an NLP using a gradient method, as is the case for solving OCPs using *Pycollo* (section 2.6). This is for two main reasons:

1. Convergence of the NLP solver is highly dependent on the accuracy of the supplied derivatives because these determine the search direction. Inaccurate derivatives can result in slow convergence or nonconvergence in some cases [36].
2. The majority of *central processing unit* (CPU) time spent by a NLP solver is inside function calls to evaluate derivatives [10]. The more cheaply these derivatives can be evaluated, the faster the NLP can be solved.

Because of these two considerations, to tractably solve an OCP via the transcription method using a gradient-based NLP solver, first- and second-order derivatives must be supplied, such that they are both highly accurate and efficient to evaluate.

3.1.2 Transcribed Nonlinear Programming Problem Sparsity Structure

A general NLP in the form of eqs. (2.64) to (2.66) does not typically possess any form of structure [131]. However, the first- and second-order derivatives of a direct collocation NLP subproblem have well-defined and predictable structures [8, 41, 264]. Specifically, the sparse first- and second-order derivatives of the NLP subproblem can be constructed from the dense first- and second-order derivatives of the OCP functions evaluated at the discretisation nodes [8, 41, 264]. The derivatives of the OCP functions have significantly fewer dimensions than the NLP subproblem

derivatives [36]. These derivatives need to be evaluated at points associated with the discretisation mesh [36]. Each state and control in the OCP has an associated vector of discretised decision variables in the NLP [36]. Furthermore, the OCP functions (and their derivatives) form groups of similar equations where a single equation corresponds to a single node in the discretisation mesh [38, 122]. Each of these equations is only a function of the discretised state and control at the node to which it corresponds and is independent of decision variables associated with other nodes [8, 264]. Consequently, significant portions of these derivative calculations can be vectorised [264]. By exploiting the known sparsity pattern of the derivatives of the NLP and their relationship to the OCP, increased efficiency in derivative calculation can be achieved [8, 41, 264]. Therefore, when solving an OCP via transcription to an NLP, this knowledge of the sparsity structure should be exploited to improve efficiency of derivative computations [36].

3.1.3 Categories of Derivative Methods

Approximation and Analytic Methods

A number of different approaches have been developed for evaluating derivatives for computer programs. These can be broadly categorised as either approximation or analytic methods. Approximation methods use a truncated Taylor series expansion of the function in question to provide an estimate for the derivative with a defined order of error [131]. The function is typically sampled at the point of interest alongside one-or-more neighbouring points to evaluate the estimate. The simplest family of approximation methods, *finite differencing* (FD) methods, are well-known and extensively used across many fields [139, 227]. They do, however, have many limitations and recently new methods have been developed to address these [10, 113, 208]. These more recent methods include ones based on complex arithmetic, such as the bicomplex step method [208], as well as methods based on hyper-dimensional extensions of generalised complex numbers, such as hyper dual numbers [113].

Analytic methods provide exact derivatives by decomposing the target function into a sequence of elementary arithmetic operations and applying the rules of differential calculus at each [136]. This family includes *manual differentiation* (MD), *symbolic differentiation* (SD) and *algorithmic differentiation* (AD). Analytic methods provide the major advantage of providing exact (or as-exact-as-possible once converted to a callable function based on floating point arithmetic) derivative information, as well as the potential to be highly computationally efficient if implemented correctly [139, 227]. These advantages do, however, typically come at the expense

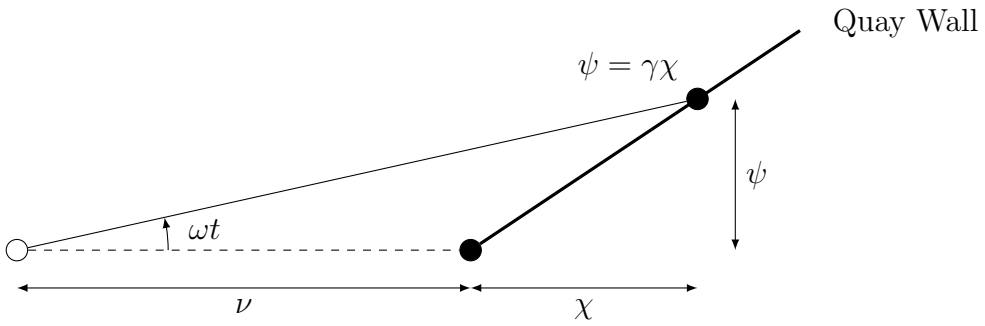


Figure 3.1: Lighthouse geometry.

of conceptual and implementational complexity [10, 136].

The Lighthouse Example (Griewank and Walther, 2008)

Certain derivative-taking methods are conceptually difficult, especially when described in their pure mathematical or algorithmic form. To assist the reader, a simple example (the lighthouse example from [136]) is presented. This example will be used to aid with the explanation of each method and to facilitate discussion of their advantages and disadvantages.

The lighthouse example models the position on a quay wall of the spot of light emitted from a lighthouse (fig. 3.1). The wall has slope γ and is a horizontal distance ν from the lighthouse, while the lighthouse rotates at an angular velocity ω in time t. The plan coordinates of the spot of light, χ and ψ, are given as

$$\chi = \frac{\nu \tan \omega t}{\gamma - \tan \omega t} \tag{3.1}$$

and

$$\psi = \frac{\gamma\nu \tan \omega t}{\gamma - \tan \omega t}\,. \tag{3.2}$$

With the independent variables $\mathbf{x} = [\nu, \gamma, \omega, t]$ and the dependent variables $\mathbf{y} = [\chi, \psi]$, the function $F : \mathbb{R}^4 \mapsto \mathbb{R}^2$ describes the position $\mathbf{y} = F(\mathbf{x})$. Note that the dependent variables differ only by the relationship $\psi = \gamma\chi$ so in this instance it would be uneconomical to manipulate and evaluate χ and ψ independently. It is often the case when dealing with vector functions in many variables that many of the scalar dependent variables will be comprised of like-subexpressions, especially when modelling a real system.

As both the independent variables $\mathbf{x}$ and dependent variables $\mathbf{y}$ are vectors, if directional derivatives are taken then this will result in a Jacobian matrix

$$\mathbf{G} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} G_{1,1} & G_{1,2} & G_{1,3} & G_{1,4} \\ G_{2,1} & G_{2,2} & G_{2,3} & G_{2,4} \end{bmatrix}, \tag{3.3}$$

where the entries in $\mathbf{G}$, $G_{i,j}$ for $(i = 1, 2; j = 1, 2, 3, 4)$, which have been derived analytically by hand, are given by

$$G_{1,1} = \frac{\tan(\omega t)}{\gamma - \tan(\omega t)} \tag{3.4}$$

$$G_{1,2} = -\frac{\nu \tan(\omega t)}{(\gamma - \tan(\omega t))^2} \tag{3.5}$$

$$G_{1,3} = \frac{\nu t \sec^2(\omega t)}{\gamma - \tan(\omega t)} + \frac{\nu t \sec^2(\omega t) \tan(\omega t)}{(\gamma - \tan(\omega t))^2} \tag{3.6}$$

$$G_{1,4} = \frac{\nu \omega \sec^2(\omega t)}{\gamma - \tan(\omega t)} + \frac{\nu \omega \sec^2(\omega t) \tan(\omega t)}{(\gamma - \tan(\omega t))^2} \tag{3.7}$$

$$G_{2,1} = \frac{\gamma \tan(\omega t)}{\gamma - \tan(\omega t)} \tag{3.8}$$

$$G_{2,2} = \frac{\nu \tan(\omega t)}{\gamma - \tan(\omega t)} - \frac{\nu \gamma \tan(\omega t)}{(\gamma - \tan(\omega t))^2} \tag{3.9}$$

$$G_{2,3} = \frac{\nu \gamma t \sec^2(\omega t)}{\gamma - \tan(\omega t)} + \frac{\nu \gamma t \sec^2(\omega t) \tan(\omega t)}{(\gamma - \tan(\omega t))^2} \tag{3.10}$$

$$G_{2,4} = \frac{\nu \gamma \omega \sec^2(\omega t)}{\gamma - \tan(\omega t)} + \frac{\nu \gamma \omega \sec^2(\omega t) \tan(\omega t)}{(\gamma - \tan(\omega t))^2}. \tag{3.11}$$

Evaluation Trace

To understand the computational cost associated with a method, it is important to think about how the calculation would be carried out when executed by a CPU. This can be done by analysing the *evaluation trace* (ET) of a function. An ET can be produced by decomposing a function down to a series of atomic operations and, resultantly, it details the dependency graph between the variables of the function. Each auxiliary variable is computed using only a simple (typically unary or binary) operation and only depends on variables that have been previously calculated. As a result the ET is a *directed acyclic graph* (DAG).

Decomposing eqs. (3.1) and (3.2) to produce an ET results in the series of computations seen in table 3.1. The computation of y_1 and y_2, which correspond to χ and ϕ respectively, in the ET is interesting because common subexpressions are automatically reused. This is best illustrated by the case in which $\psi = \gamma\chi$. Rather than recomputing ψ from scratch using its symbolic primitives (ν, γ, ω and t), it can be evaluated cheaply using only a binary multiplication of the already-computed χ and the symbolic primitive γ. Exploitation of this reusable subexpressions is of high importance when ensuring that an algorithm is efficient.

Independent Variables
$x_1 = \nu$
$x_2 = \gamma$
$x_3 = \omega$
$x_4 = t$
Auxiliary Variables
$w_1 = \omega t = x_3 x_4 = \text{Mul}(x_3, x_4)$
$w_2 = \tan(\omega t) = \tan(w_1) = \text{Tan}(w_1)$
$w_3 = \gamma - \tan(\omega t) = x_2 - w_2 = \text{Sub}(x_2, w_2)$
$w_4 = \nu \tan(\omega t) = x_1 w_2 = \text{Mul}(x_1, w_2)$
$w_5 = \frac{\nu \tan(\omega t)}{\gamma - \tan(\omega t)} = \frac{w_4}{w_3} = \text{Div}(w_4, w_3)$
$w_6 = \frac{\gamma\nu \tan(\omega t)}{\gamma - \tan(\omega t)} = x_2 w_5 = \text{Mul}(x_2, w_5)$
Dependent Variables
$y_1 = \chi = \frac{\nu \tan(\omega t)}{\gamma - \tan(\omega t)} = w_5$
$y_2 = \psi = \frac{\gamma\nu \tan(\omega t)}{\gamma - \tan(\omega t)} = w_6$

Table 3.1: ET for the lighthouse example.

3.1.4 Finite Differencing

The most prevalent method for derivative generation in optimisation, due to its ease of implementation, is FD [36, 131]. As FD is a numerical method, it results in only approximate derivatives, the accuracy of which depend on the choice of perturbation size h [131]. Two conflicting error sources, *truncation error* and *subtractive cancellation* (or *roundoff*) *error*, exist.

FD formulas with higher-order errors and for higher-order derivatives exist [36, 131, 229]. These all require increasing numbers of perturbations, raising a tradeoff between accuracy and computational cost [10, 113]. Recent benchmarking of derivative estimation methods for solving OCPs using direct collocation has shown that FD, while easy to implement stably, typically results in the largest runtimes [10]. This is because its associated inaccuracies typically result in requiring more NLP iterations to reach convergence [10]. For this reason, FD should be avoided for OCP applications.

3.1.5 Complex Arithmetic

The use of complex arithmetic to approximate derivatives has been thoroughly investigated [3, 115, 205, 208, 223, 309]. Methods based on hyper-dual numbers are the most useful practically as they allow derivatives of any order to be computed,

while being immune to both subtractive cancellation and truncation errors [113]. They are therefore accurate to machine precision and functionally equivalent to forward-mode algorithmic differentiation [113].

Hyper-dual methods are up to 3.5 times more expensive than central FD per derivative when all factors are taken into account [113]; they typically require fewer perturbations than FD [36, 113, 208], but a hyper-dual operation typically takes between four and 14 elementary operations per each real-valued operation [113]. Computational benchmarking of central FD and a hyper-dual method in the context of solving OCPs has been published [10]. The hyper-dual method outperformed central FD overall due to its accuracy, despite being more computationally expensive per NLP iteration [10]. Therefore, methods based on hyper-dual numbers are preferred for OCP derivative approximations.

3.1.6 Manual Differentiation

MD of functions by hand almost universally results in the development of derivative code with the fastest runtimes [310]. This is because the differentiated functions can usually be successfully reduced down to their most simplified form with all common subexpressions factorised out, reducing the number of intermediate calculations required in a single function evaluation [136]. MD is, however, very time consuming, labour intensive and error prone, and is therefore seldom utilised for OCP applications [139, 227, 310].

3.1.7 Symbolic Differentiation

SD, is the mathematical differentiation of functions using a computer. SD relies upon the use of a *computer algebra system* (CAS), which can represent and manipulate variables and expressions symbolically. CASs are capable of handling a wide range of mathematics, most important of which to this application is analytic calculus. Examples of CASs include the *MATLAB*'s *MuPAD*-based Symbolic Math Toolbox and *Python*'s *SymPy*.

All expressions are represented within a CAS by their decomposition down to a DAG. Each intermediate node in the DAG represents a single mathematical operator that operates on a sequence of one or more operands. CASs are capable of taking both total and partial derivatives. The CAS is coded to contain the derivative rules for all of its contained operations. The total and partial derivatives of more complex expressions can be taken by the application of differentiation rules such as the chain

rule.

SD is capable of efficient low-level code generation by conducting expression simplification and utilising common subexpression elimination before compilation [310]. However, it generally requires the representation of entire functions as a single expression and is therefore memory intensive, particularly for large functions [136]. Furthermore, for large expressions, expression simplification and common subexpression elimination can result in high upfront computational costs. Therefore, due to its memory inefficiency and slow runtimes, SD is typically not utilised for derivative generation [136, 139, 227, 310], especially within the field of optimal control [10].

3.1.8 Algorithmic Differentiation

AD generates analytic derivatives by applying the rules of calculus numerically to a computer program [136]. Derivatives are produced by carrying numerical partial derivatives through every step in the calculation of the original function and using these to apply to rules of differential calculus [136]. For this reason AD is sometimes also referred to as *automatic differentiation* [136, 139, 227, 310]. As the derivative function is never explicitly generated, but is instead evaluated through an augmented evaluation of the original functions, AD has a computational cost of only a small constant number times the original function call [136]. Also, because the derivative is being evaluated numerically, AD supports the use of program flow control with conditional statements, loops and recursion [136].

AD requires the function of interest, which is to be differentiated, to be expressed as a computational graph [136]. Any computer function can be decomposed to a set of elementary operations, for each of which the derivative is known. AD can then be implemented by interpreting this computational graph and augmenting it calculations of various derivatives. This is done by representing each variable as a value-derivative pair and applying the chain rule at each elementary operation, carrying the derivative terms through. When done for the entire computation, the overall derivative is yielded [136]. AD comes in two modes, forward-mode and reverse-mode, depending on whether the derivative matrix is to be found by propagating the chain rule from the input or the output of the function respectively.

Forward-Mode

Forward-mode AD allows the partial derivatives of a function to be evaluated numerically with respect to a single input variable at a time. Consider the function

$F : \mathbb{R}^n \mapsto \mathbb{R}^m$ with the n input variables x_i for $i \in [1, \ldots, n]$ and the m output variables y_j for $j \in [1, \ldots, m]$. Each input variable x_i is associated with a corresponding derivative term $\dot{x}_i$. Each intermediate variable w_k (*primal*) is also given a corresponding derivative term $\dot{w}_k$ (*tangent*), where

$$\dot{w}_k = \frac{\partial w_k}{\partial x} \tag{3.12}$$

and is an evaluation of the intermediate variable's sensitivity to the function's inputs. Note that each tangent variable, as defined in eq. (3.12), is a sensitivity with respect to a generic input variable. It is therefore important that only a single input variable at a time is assigned a nonzero tangent.

A tangent can be evaluated by applying the chain rule to the elementary operation of its corresponding primal. Conducting this process for all elements in the computation graph results in both the evaluation of all of the output variables y_j as well as their partial derivatives with respect to the specified input variable $\dot{y}_j$. Construction of this augmented forward-mode trace for the lighthouse example is shown in table 3.2.

In the forward-mode derivative trace (table 3.2), each auxiliary variable in the original ET has between one and three additional lines associated with it, each corresponding to an additional unary or binary expression. The computational cost of the forward-mode evaluation is, therefore, a small multiple of the cost of the original function.

In order to compute the full Jacobian matrix of a vector function, a structured series of successive forward-mode passes can be used. Each forward-mode pass numerically evaluates a single column of the Jacobian matrix. This is done algorithmically by successively setting a single input variable's tangent to 1.0 with all other input tangents set to 0.0. As F is a function of n input variables, computation of the full Jacobian via this manner therefore requires n forward-mode passes. Therefore, computational expense increases linearly with n.

Reverse Mode

In contrast to forward-mode AD, which propagated tangents through the original function from its inputs, reverse-mode AD operates in the opposite direction [136]. In reverse-mode, derivatives are accumulated backwards through the function's ET starting at its output [136]. Each intermediate variable w_k is associated with an *adjoint* $\bar{w}_k$, where

$$\bar{w_k} = \frac{\partial y}{\partial w_k} \tag{3.13}$$

Independent Variables
$x_1 = \nu$
$\dot{x}_1 = \dot{\nu}$
$x_2 = \gamma$
$\dot{x}_2 = \dot{\gamma}$
$x_3 = \omega$
$\dot{x}_3 = \dot{\omega}$
$x_4 = t$
$\dot{x}_4 = \dot{t}$
Auxiliary Variables
$w_1 = x_3 x_4 = \text{Mul}\,(x_3, x_4)$
$\dot{w}_{1a} = x_3 \dot{x}_4 = \text{Mul}\,(x_3, \dot{x}_4)$
$\dot{w}_{1b} = \dot{x}_3 x_4 = \text{Mul}\,(\dot{x}_3, x_4)$
$\dot{w}_1 = x_3 \dot{x}_4 + \dot{x}_3 x_4 = \text{Add}\,(\dot{w}_{1a}, \dot{w}_{1b})$
$w_2 = \tan(w_1) = \text{Tan}\,(w_1)$
$\dot{w}_{2a} = \sec(w_1) = \text{Sec}\,(w_1)$
$\dot{w}_{2b} = \sec^2(w_1) = \text{Pow}\,(w_{2a}, 2)$
$\dot{w}_2 = \dot{w}_1 \sec^2(w_1) = \text{Mul}\,(\dot{w}_1, \dot{w}_{2b})$
$w_3 = x_2 - w_2 = \text{Sub}\,(x_2, w_2)$
$\dot{w}_3 = \dot{x}_2 - \dot{w}_2 = \text{Sub}\,(\dot{x}_2, \dot{w}_2)$
$w_4 = x_1 w_2 = \text{Mul}\,(x_1, w_2)$
$\dot{w}_{4a} = x_1 \dot{w}_2 = \text{Mul}\,(x_1, \dot{w}_2)$
$\dot{w}_{4b} = \dot{x}_1 w_2 = \text{Mul}\,(\dot{x}_1, w_2)$
$\dot{w}_4 = x_1 \dot{w}_2 + \dot{x}_1 w_2 = \text{Add}\,(\dot{w}_{4a}, \dot{w}_{4b})$
$w_5 = \frac{w_4}{w_3} = \text{Div}\,(w_4, w_3)$
$\dot{w}_{5a} = w_5 \dot{w}_3 = \text{Mul}\,(w_5, \dot{w}_3)$
$\dot{w}_{5b} = \dot{w}_4 - w_5 \dot{w}_3 = \text{Sub}\,(\dot{w}_4, \dot{w}_{5a})$
$\dot{w}_5 = \frac{\dot{w}_4}{w_3} - \frac{\dot{w}_3 w_4}{w_3^2} = \text{Div}\,(\dot{w}_{5b}, w_3)$
$w_6 = x_2 w_5 = \text{Mul}\,(x_2, w_5)$
$\dot{w}_{6a} = \dot{x}_2 w_5 = \text{Mul}\,(\dot{x}_2, w_5)$
$\dot{w}_{6b} = x_2 \dot{w}_5 = \text{Mul}\,(x_2, \dot{w}_5)$
$\dot{w}_6 = \dot{x}_2 w_5 + x_2 \dot{w}_5 = \text{Add}\,(\dot{w}_{6a}, \dot{w}_{6b})$
Dependent Variables
$y_1 = w_5$
$\dot{y}_1 = \dot{w}_5$
$y_2 = w_6$
$\dot{y}_2 = \dot{w}_6$

Table 3.2: Jacobian ET for the lighthouse example using forward-mode AD.

and represents the sensitivity of the function's output with respect to changes in the intermediate variable in question. Note again that eq. (3.13) defines the adjoints as sensitivities with respect to a generic output variable. Therefore, to produce a meaningful result, only one output adjoint can be seeded to 1.0 per single reverse-mode pass.

Reverse-mode AD is a two stage process. In the first stage, the function is evaluated by running forward through the ET. The numerical values of the intermediate variables are stored in memory and their dependency relationships recorded to be used in the second stage. This is followed by the second stage in which the adjoints $\bar{w}_k$ are propagated backwards through the ET, from the last intermediate variable to the first.

Table 3.3 details reverse-mode AD for the lighthouse example. It can be seen from this example that a single reverse-mode pass generates a row of the Jacobian matrix. Therefore, for a function $F : \mathbb{R}^n \mapsto \mathbb{R}^m$, the full Jacobian can be evaluated with m reverse-mode passes. In contrast to forward-mode AD, the complexity of full Jacobian evaluation using reverse-mode AD grows linearly with the number of output variables. It has been shown in the literature that if the function F can be evaluated using σ operations, then a single reverse-mode pass should take at most 6σ operations to complete [136]. In reality the complexity of a single reverse-mode pass is typically only two or three times greater than the original function [136].

Discussion

Forward-mode AD is suited to situations where the number of input variables is small in comparison to the number of output variables. Conversely, reverse-mode AD is suited to situations where the number of output variables is small compared to the number of input variables. Therefore, for the function $F : \mathbb{R}^n \mapsto \mathbb{R}^m$, forward-mode AD is preferred if $n \ll m$ while reverse-mode is preferred if $n \gg m$ [136, 139].

Reverse-mode AD requires that a single pass must be conducted in two parts, with the first part involving computing and storing the entire ET of the original function such that it can be used in the second part. For simple functions with a small number of intermediate variables this is not a problem. However, if the ET is so large that the amount of data exceeds the available *random-access memory* (RAM), significant performance penalties are incurred [227]. Techniques, such as *retaping* and *checkpointing*, have been developed to improve the memory properties of reverse-mode AD, however these significantly increase the complexity of imple-

Independent Variables
$x_1 = \nu$
$x_2 = \gamma$
$x_3 = \omega$
$x_4 = t$
Auxiliary Variables
$w_1 = x_3 x_4 = \text{Mul}(x_3, x_4)$
$w_2 = \tan(w_1) = \text{Tan}(w_1)$
$w_3 = x_2 - w_2 = \text{Sub}(x_2, w_2)$
$w_4 = x_1 w_2 = \text{Mul}(x_1, w_2)$
$w_5 = \frac{w_4}{w_3} = \text{Div}(w_4, w_3)$
$w_6 = x_2 w_5 = \text{Mul}(x_2, w_5)$
Dependent Variables
$y_1 = w_5$
$y_2 = w_6$

Forward Sweep

Independent Variables
$x_1 = \nu$
$x_2 = \gamma$
$x_3 = \omega$
$x_4 = t$
$\bar{y}_1$
$\bar{y}_2$
Auxiliary Variables
$\bar{w}_6 = 0 = \text{Init}(0)$
$\bar{w}_6 \mathrel{+}= \bar{y}_2$
$\bar{w}_5 = 0 = \text{Init}(0)$
$\bar{w}_5 \mathrel{+}= \bar{y}_1$
$\bar{w}_5 \mathrel{+}= \text{Mul}(x_2, \bar{w}_6)$
$\bar{x}_2 = 0 = \text{Init}(0)$
$\bar{x}_2 \mathrel{+}= \text{Mul}(w_5, \bar{w}_6)$
$\bar{w}_3 = 0 = \text{Init}(0)$
$w_{3a} = w_3^{-1} = \text{Pow}(w_3, -1)$
$w_{3b} = \frac{w_4}{w_3^2} = \text{Mul}(w_5, w_{3a})$
$w_{3c} = -\frac{w_4}{w_3^2} = \text{Neg}(w_{3b})$
$\bar{w}_3 \mathrel{+}= \text{Mul}(w_{3c}, \bar{w}_5)$
$\bar{w}_4 = 0 = \text{Init}(0)$
$\bar{w}_4 \mathrel{+}= \text{Mul}(w_{3a}, \bar{w}_5)$
$\bar{x}_1 = 0 = \text{Init}(0)$
$\bar{x}_1 \mathrel{+}= \text{Mul}(w_2, \bar{w}_4)$
$\bar{w}_2 = 0 = \text{Init}(0)$
$\bar{w}_2 \mathrel{+}= \text{Mul}(x_1, \bar{w}_4)$
$\bar{x}_2 \mathrel{+}= \bar{w}_3$
$w_{2a} = -w_3 = \text{Neg}(w_3)$
$\bar{w}_2 \mathrel{+}= \text{Mul}(w_{2a}, \bar{w}_3)$
$\bar{w}_1 = 0 = \text{Init}(0)$
$w_{1a} = \sec(w_1) = \text{Sec}(w_1)$
$w_{1b} = \sec^2(w_1) = \text{Mul}(w_{1a}, w_{1a})$
$\bar{w}_1 \mathrel{+}= \text{Mul}(w_{1b}, \bar{w}_2)$
$\bar{x}_3 = 0 = \text{Init}(0)$
$\bar{x}_3 \mathrel{+}= \text{Mul}(x_4, \bar{w}_1)$
$\bar{x}_4 = 0 = \text{Init}(0)$
$\bar{x}_4 \mathrel{+}= \text{Mul}(x_3, \bar{w}_1)$
Dependent Variables
$\dot{x}_1 = \bar{x}_1$
$\dot{x}_2 = \bar{x}_2$
$\dot{x}_3 = \bar{x}_3$
$\dot{x}_4 = \bar{x}_4$

Reverse Sweep

Table 3.3: Jacobian ET for the lighthouse example using reverse-mode AD.

mentation [136]. For OCP applications where the number of decision variables and constraints in the NLP are typically of the same order, forward-mode AD would be preferred due to its easier implementation. Note, however, that reverse-mode AD would be required to facilitate efficient evaluation of second-order derivative information if this were required [136].

One of AD's main advantages is that it can be applied to standard computer code without the need for substantial modification, unlike SD which requires all expressions to be given in closed form [139]. This means that AD can be applied to code containing conditional branching, loops and recursion [136, 139]. However, as the NLP solvers employed when solving an OCP computationally require smooth second derivatives [43, 129], it is generally easier to ensure a well-conditioned smooth problem formulation by avoiding the use of conditional branching, loops and recursion.

OCP applications also require numerical evaluation of second-order derivatives in the form of the Hessian of the Lagrangian. AD software packages almost universally compute second-order derivatives by applying AD twice [136, 139]. In the case of OCPs, the first-order Lagrangian gradient can be computed in a single reverse-mode pass [139]. The full Hessian matrix can then be evaluated by applying an additional n passes, either forward- or reverse-mode, where n is the number of variables in the OCP. Efficient evaluation of Hessian matrices is, therefore, dependent on reverse-mode AD and can be done with $n + 1$ AD passes. If only the forward-mode is used then n^2 passes will be required. Computational benchmarking of numerically solving OCPs in conjunction with AD has shown poor performance in comparison to other approaches, such as hyper-dual approximations, potentially because of the complexity involved in computing the second-order derivatives this way [10].

3.1.9 Computational Implementation

For a derivative-taking algorithm to be of practical use it requires a computational implementation. This implementation must be both efficient and easy to use. Naive implementations can result in prohibitively slow code and excessive memory usage [139, 227].

Source Transformation

Source transformation (ST) works by starting with the source implementation of a function and then augmenting it in the manner required by the AD algorithm, pro-

ducing new source code for the modified function. The elementary operations in the target function are analysed, each is augmented with the differential rules of calculus, and new source code is generated which can be used to evaluate derivatives alongside the original function. If ST is being used to generate executables that consist solely of mathematical functions and do not take advantage of advanced *object-oriented programming* (OOP) features, then ST is considered by the AD community as the preferred approach [227]. Examples of software packages that implement AD via ST are *ADIFOR* [45], *Tapenade* [148] and *ADiGator* [325].

Operator Overloading

Operator overloading (OO) works by introducing a new class of variable which contains both the numerical real component alongside a numerical derivative component. This class overloads all arithmetic operators and mathematical functions such that they function alongside this new class, carrying through the numerical derivatives of each variable as the original function is evaluated. Implementing forward-mode AD using OO is conceptually simple. While it is still possible to implement reverse-mode AD using OO, this requires considerably more labour [227]. Using hyper-dual numbers with operator overloading is functionally equivalent to forward-mode AD [113]. Examples of software packages that implement AD via OO are *ADOL-C* [324], *Adept* [168], *JAX* [61] and *ForwardDiff* [287].

Discussion

There is no universally preferred approach for computational implementations of AD [227]. Whether ST or OO should be used will depend on the specifics of the algorithm in question. For example, OO is best suited to approximation methods based on hyper-dual numbers and bicomplex steps [113, 208]. This is because these methods, by definition, carry numerical values of the derivative terms in tandem with the intermediate variables through the ET. Therefore, the derivative terms are automatically calculated when the rules of the number system's arithmetic are obeyed. In this scenario OO leads to a trivial implementation of AD provided types are declared for the number system and its rules of arithmetic are correctly and fully defined. Conversely, ST may be considered the appropriate approach for implementations of reverse-model AD [227]. This is because construction of the backwards pass is conceptually involved. It can be most easily done by analysing the dependency relationships from the forward pass and using this information to implement the appropriate adjoint accumulations in the reverse pass. Therefore, when implementing computer code for an AD algorithm, the cohesion between ST

and OO, and the specifics of the algorithm should be analysed and considered in detail before a decision is made.

3.2 Research Objectives

Section 1.3 laid out the objective of investigating methods for determining first- and second-order derivative information that reduce computational cost and maximise derivative evaluation speed during an OCP solve. The findings and developments should, where practicable, be implemented as part of the *Biomechanics Predictive Simulation Toolkit* (BPST).

From the analysis and review of past work in section 3.1, a number of limitations and constraints associated with the current methods available in this area were identified. To address these, and meet the overall objective above, the following sub-objectives are laid out:

1. investigate ways in which SD and AD can be combined such that the benefits of each method are used to offset the limitations of the other;
2. investigate ways in which the efficiency of derivative-taking methods can be improved in the context of OCPs, particularly in the area of exact evaluation of Lagrangian Hessians;
3. investigate ways in which the known sparsity of the NLP subproblem can be exploited when combined with SD- and AD-based methods;
4. incorporate any resulting findings into the development of a supplementary derivative backend for *Pycollo* (section 2.6);
5. validate the performance of the supplementary derivative backend by solving a range of OCPs from the literature, and benchmark performance against *Pycollo*'s *CasADi* backend;
6. identify areas for further development and improvement.

3.3 Hybrid-Symbolic-Algorithmic Differentiation

This section details the development of a derivative-taking algorithm motivated by the requirements of computing the first- and second-order derivative information specifically required by Newton and quasi-Newton NLP solvers such as *Ipopt*.

3.3.1 Motivation

Section 3.1 describes all of the derivative-taking methods that are currently practically used in the field of optimal control, namely AD and approximation methods based on hyper-dual numbers. A significant downside to all of these methods is that the derivatives are somewhat evaluated numerically based on the original function. This is done by using seed values (of 0.0 and 1.0) to propagate sensitivities to either the input or output variables through the target function based on the target function's original ET. For this to result in correct partial derivatives, only one input or output variable can be seeded at a time. Therefore, a derivative matrix must be evaluated on a column-by-column or row-by-row basis.

Column-by-column evaluation can lead to inefficiencies in computation in two ways. Firstly, it cannot be known at runtime which seeds have been set to 0.0 and 1.0. This can result in the execution of many trivial floating point operations in which the seed values are multiplied by other variables in order to ensure sensitivities with respect to only a single variable are propagated at a time. For example, if the auxiliary variable w_1 is denoted by the function $w_1 = x_1 x_2$ then in forward-mode AD the sensitivity of w_1 with respect to the functions input, $\dot{w}_1$, can be computed as

$$\dot{w}_1 = x_1 \dot{x}_2 + \dot{x}_1 x_2 \,, \tag{3.14}$$

where $\dot{x}_1$ and $\dot{x}_2$ are input seeds. On the first pass with $\dot{x}_1 = 1.0$ and $\dot{x}_2 = 0.0$, eq. (3.14) will evaluate to $\dot{w}_1 = x_2$. With $\dot{x}_1 = 0.0$ and $\dot{x}_2 = 1.0$ on the second pass, eq. (3.14) will instead evaluate to $\dot{w}_1 = x_1$. Combined, this is four unnecessary multiplications and two unnecessary additions that could be replaced with assignments. If this could be analysed symbolically, these six wasted operations could be avoided.

The second inefficiency arises from the fact that the column-by-column evaluation requires the same function to be evaluated with different seeds numerous times. While this column-by-column calculation is not necessarily poorly performing, because all columns need to be populated in order to evaluate a full derivative matrix, it can result in many repeated calculations being done on each pass. This is because the function being evaluated will likely contain many intermediate expressions that are not functions of the sensitivities and so do not change in value between passes. While these could be precomputed and cached, neither AD nor approximation methods based on hyper-dual numbers allow for any preprocessing of the dependency relationships between intermediate variables due to their numerical nature. If the dependency relationships between these intermediate variables could be easily determined and shared between columns, then significant computation could likely be saved when evaluating a full derivative matrix.

For large problems, the number of function passes required when using AD or hyper-dual approximation scales with the size of the NLP subproblem, even if its sparsity is being exploited due to the underlying structure of the OCP. If the OCP is to be solved on a very dense mesh, as may be needed in order to meet a required mesh accuracy, then the derivative computations required by the NLP solver may become prohibitively expensive. It is, therefore, highly desirable to develop a derivative-taking algorithm for OCP applications in which these wasted and repeated calculations are avoided. This will be done by applying symbolic methods to the theory underpinning AD, with the algorithm termed *hybrid-symbolic-algorithmic differentiation* (hSAD).

3.3.2 Development

Let $\mathbf{f}(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$ be a generic vector function, and its Jacobian matrix

$$\mathbf{G} = \frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}} . \tag{3.15}$$

Evaluating $\mathbf{G}$ using a computational evaluation procedure of atomic operations becomes nontrivial when $\mathbf{f}(\mathbf{x})$ is a composite function. If $\mathbf{f}(\mathbf{x})$ is a composite expression of L functions then

$$\mathbf{f}(\mathbf{x}) = \mathbf{f}_L \circ \mathbf{f}_{L-1} \circ \ldots \circ \mathbf{f}_2 \circ \mathbf{f}_1(\mathbf{x}) \tag{3.16}$$

where

$$\mathbf{f}_2 \circ \mathbf{f}_1(\mathbf{x}) = \mathbf{f}_2(\mathbf{f}_1(\mathbf{x})) . \tag{3.17}$$

The chain rule is defined as

$$\frac{\partial \mathbf{f}_b \circ \mathbf{f}_a(\mathbf{x})}{\partial \mathbf{x}} = \frac{\partial \mathbf{f}_b \circ \mathbf{f}_a(\mathbf{x})}{\partial \mathbf{f}_a(\mathbf{x})} \frac{\partial \mathbf{f}_a(\mathbf{x})}{\partial \mathbf{x}} . \tag{3.18}$$

Using the nomenclature $\mathbf{G}_a = \frac{\partial \mathbf{f}_a(\mathbf{x})}{\partial \mathbf{x}}$ and $\mathbf{G}_b = \frac{\partial \mathbf{f}_b \circ \mathbf{f}_a(\mathbf{x})}{\partial \mathbf{f}_a(\mathbf{x})}$, the matrix of first-order partial derivatives of the composite function can be generated by applying the chain rule L times, giving

$$\mathbf{G} = \mathbf{G}_L \mathbf{G}_{L-1} \ldots \mathbf{G}_2 \mathbf{G}_1 . \tag{3.19}$$

Tier Partitioning

When a function's ET of atomic operations is produced, a number of auxiliary variables, which define the procedure of operations between the input and output, are defined. The auxiliary variables in the ET form an ordered serial list with each auxiliary variable defining an operation with operands consisting of only previous

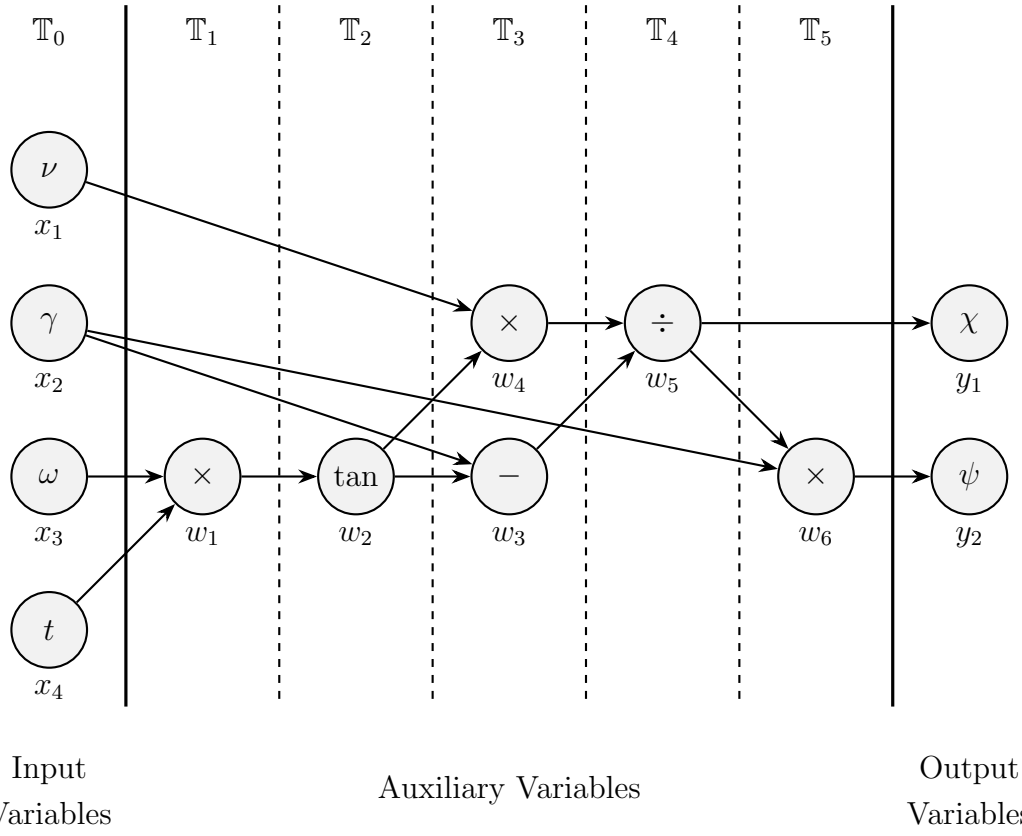


Figure 3.2: DAG for the lighthouse example.

auxiliary variables. The ET can, therefore, be represented as a DAG between the function's input and output.

Unlike a linear evaluation procedure, a DAG is not strictly serial and may contain parallel nodes. In the context of auxiliary variables, this means that certain nodes in the DAG are of equal precedence and their order of evaluation can be rearranged without impacting the validity of the evaluation procedure. This is illustrated in the lighthouse example (section 3.1.3), where the order in which the two auxiliary variables w_3 and w_4 are evaluated in a linear evaluation procedure is inconsequential. This is because both w_3 and w_4 only depend on auxiliary variables in the set $\{x_1, x_2, x_3, x_4, w_1, w_2\}$, which have been previously evaluated. The parallel nature of w_3 and w_4 is illustrated by fig. 3.2, the DAG for the ET of the lighthouse example shown in table 3.1, in which the w_3 and w_4 nodes are aligned vertically. Both nodes are also placed to the right of the w_2 node and to the left of the w_5. w_2 must be computed before w_3 and w_4 as both nodes' operations have w_2 as an operand. Similarly, w_5 must be computed after both w_3 and w_4 as both are operands in the operation associated with w_5.

hSAD makes use of the fact that certain auxiliary variables are of equal precedence in order to allow the whole expression graph to be processed in fewer stages

using matrix operations. In the context of the hSAD algorithm, this step of grouping auxiliary variables by their ET precedence is termed *tier partitioning*. Conducting tier partitioning in full for the lighthouse example yields the five auxiliary tiers alongside the zeroth tier, $\mathbb{T}_0$, containing the independent variables

$$x_{\mathbb{T}_0} = \{x_1, x_2, x_3, x_4\} \tag{3.20}$$
$$x_{\mathbb{T}_1} = \{w_1\} \tag{3.21}$$
$$x_{\mathbb{T}_2} = \{w_2\} \tag{3.22}$$
$$x_{\mathbb{T}_3} = \{w_3, w_4\} \tag{3.23}$$
$$x_{\mathbb{T}_4} = \{w_5\} \tag{3.24}$$
$$x_{\mathbb{T}_5} = \{w_6\} \ . \tag{3.25}$$

The tiers are denoted in set-notation in eqs. (3.20) to (3.25), but can also be denoted as row vectors, as will be required when used in the matrix notation of partial derivatives to follow

$$\mathbf{x}_{\mathbb{T}_0} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix} \tag{3.26}$$
$$\mathbf{x}_{\mathbb{T}_1} = \begin{bmatrix} w_1 \end{bmatrix} \tag{3.27}$$
$$\mathbf{x}_{\mathbb{T}_2} = \begin{bmatrix} w_2 \end{bmatrix} \tag{3.28}$$
$$\mathbf{x}_{\mathbb{T}_3} = \begin{bmatrix} w_3 & w_4 \end{bmatrix} \tag{3.29}$$
$$\mathbf{x}_{\mathbb{T}_4} = \begin{bmatrix} w_5 \end{bmatrix} \tag{3.30}$$
$$\mathbf{x}_{\mathbb{T}_5} = \begin{bmatrix} w_6 \end{bmatrix} \ . \tag{3.31}$$

Row vectors are required when used in the matrix notation as the order of variables in each tier must be consistent between operations. As expected from the rationale above, examining eqs. (3.23) and (3.29) shows that $\mathbf{x}_{\mathbb{T}_3}$ contains both w_3 and w_4.

For a simple problem like the lighthouse example, determining the number of tiers and the tier associated with each node is trivial by inspection. However, for larger more complex problems, and when a computational implementation is required, this approach is not feasible. In order to algorithmically determine the tier of a node in the DAG, a *depth-first search* (DFS) algorithm [80] can be used. In this approach, each node is assigned a tier variable, which will be used to store the node's associated tier. The tier level of a node of interest is determined by inspecting its parent nodes (i.e. the node or nodes which are associated with the operands of its operation). If all parent nodes already have a defined tier level then the tier level of the node of interest is set to one more than the maximum tier level of its parent nodes. If a parent node does not have a defined tier level yet then it becomes a new node of interest with greater priority. If the node is a root node, ascertainable by the

fact that it has no associated parent nodes, then it is assigned a tier level of 0. The recursion can be unwound and the root node's child node assigned a tier level of 1. The maximum tier can be determined trivially by finding the maximum tier value in the whole set of nodes associated with the DAG. Conducting tier partitioning using this DFS approach results in both linear time complexity and linear memory complexity provided that a node's tier is cached once determined.

To explain this approach based on DFS more explicitly, take the node w_3 from fig. 3.2 as an example. w_3 has two parent nodes, x_2 and w_2. First, x_2 is investigated. x_2 is a root node and so is assigned to $\mathbb{T}_0$, $x_2 \mapsto \mathbb{T}_0$. w_2 is currently unassigned a tier and so becomes a new node of interest with the new highest priority. w_2 has the sole parent node w_1 which is itself current unassigned a tier, becoming a further new node of interest, again with the new highest priority. w_1 has two parent nodes, x_3 and x_4, both of which are root nodes and so are assigned to $\mathbb{T}_0$, $x_3 \mapsto \mathbb{T}_0$ and $x_4 \mapsto \mathbb{T}_0$. All parent nodes of w_1 now have an assigned tier, so the tier of w_1 can be assigned as

$$w_1 \mapsto \max\left[x_3 \mapsto \mathbb{T}_0, x_4 \mapsto \mathbb{T}_0\right] \oplus 1 = \mathbb{T}_0 \oplus 1 = \mathbb{T}_1 \tag{3.32}$$

with $\oplus 1$ denoting the incrementation of the tier level by one. With its sole parent, w_1, assigned to $\mathbb{T}_1$, w_2 can now be assigned its tier as

$$w_2 \mapsto \max\left[w_1 \mapsto \mathbb{T}_1\right] \oplus 1 = \mathbb{T}_1 \oplus 1 = \mathbb{T}_2\,. \tag{3.33}$$

Finally, with both parents of w_3, x_2 and w_2, having been assigned tiers, w_3 can at last be assigned its tier as

$$w_3 \mapsto \max\left[x_2 \mapsto \mathbb{T}_0, w_2 \mapsto \mathbb{T}_2\right] \oplus 1 = \mathbb{T}_2 \oplus 1 = \mathbb{T}_3\,. \tag{3.34}$$

If further nodes need to be tier-partitioned then this can be done using the same procedure. Indeed, in this case w_4 can now be tier-partitioned with ease as both of its parent nodes, x_1 and w_2, are either trivially assigned to $\mathbb{T}_0$, due to being a root node, or have already been tier-partitioned themselves.

Delta Matrices

Applying the chain rule of eq. (3.18) to an ET, such as that in table 3.1, in a way that results in an exact expression for the ET's first-order partial derivatives is difficult. This is because the operands of each operation, and, therefore, each sub-Jacobian in the product, are almost certainly different. As a result, algorithmically determining sensitivities to the input variables requires tracking the sensitivity of every auxiliary variable to every other lower precedence auxiliary variable. For example, if $\mathbf{y} = \mathbf{f}(\mathbf{x})$ is a composite expression of $L = 2$ functions, then its Jacobian matrix of first-order

partial derivatives given by the chain rule is

$$\frac{\partial \mathbf{f}_2\left(\mathbf{f}_1\left(\mathbf{x}\right)\right)}{\partial \mathbf{x}} = \frac{\partial \mathbf{f}_2\left(\mathbf{f}_1\left(\mathbf{x}\right)\right)}{\partial \mathbf{f}_1\left(\mathbf{x}\right)} \frac{\partial \mathbf{f}_1\left(\mathbf{x}\right)}{\partial \mathbf{x}} . \tag{3.35}$$

Determining the $\frac{\partial \mathbf{f}_1(\mathbf{x})}{\partial \mathbf{x}}$ term is easy as it can only be a function of $\mathbf{x}$. However, the $\frac{\partial \mathbf{f}_2(\mathbf{f}_1(\mathbf{x}))}{\partial \mathbf{f}_1(\mathbf{x})}$ is already significantly more complicated as it can potentially be a function of not only $\mathbf{x}$ but also of $\mathbf{f}_1\left(\mathbf{x}\right)$. This complexity only increases as L increases.

The tier partitioning conducted in section 3.3.2 simplifies the application of the chain rule by:

1. tracking the dependencies between auxiliary variable operations and their operands; and
2. reducing the depth of the partial derivative ET as some tiers contain more than one variable and can be processed in parallel.

Consider the vector expression $\mathbf{y} = \mathbf{f}\left(\mathbf{x}\right)$. Firstly, consider the trivial situation where $\mathbf{f}\left(\mathbf{x}\right)$ simply maps the input variables to the output variables. While this scenario is trivially simple, it lays important foundations for the incrementally more complicated scenarios that will follow. Note that by definition the zeroth tier contains only the function's input variables, that is $\mathbf{x} = \mathbf{x}_{\mathbb{T}_0}$. Therefore, when tier-partitioned, $\mathbf{y}$ only requires a single tier, $\mathbb{T}_0$. As there are no auxiliary variables in this scenario, the partial derivatives of all tier variables with respect to the input variables can be expressed as

$$\frac{\partial \mathbf{x}_{\mathbb{T}_0}}{\partial \mathbf{x}} = \frac{\partial \mathbf{x}_{\mathbb{T}_0}}{\partial \mathbf{x}_{\mathbb{T}_0}} = \mathbf{I} \tag{3.36}$$

where $\mathbf{I}$ denotes the identity matrix

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} . \tag{3.37}$$

The Jacobian of $\mathbf{y}$ with respect to $\mathbf{x}$ can then be expressed as

$$\mathbf{G} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_0}} \frac{\partial \mathbf{x}_{\mathbb{T}_0}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_0}} \frac{\partial \mathbf{x}_{\mathbb{T}_0}}{\partial \mathbf{x}_{\mathbb{T}_0}} \tag{3.38}$$

where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_0}}$ will, in this scenario only, trivially be the identity matrix due to the fact that the output can only be constructed from the $\mathbb{T}_0$ variables.

Secondly, consider the next simplest situation in which $\mathbf{y} = \mathbf{f}\left(\mathbf{x}\right)$, which requires auxiliary variables in order to be expressed, has been tier-partitioned and found

to require $\{\mathbb{T}_0, \mathbb{T}_1\}$. Auxiliary variables are required to correspond to an atomic operation, with these atomic operations typically being unary or binary operations. The $\mathbb{T}_1$ auxiliary variables, $\mathbf{x}_{\mathbb{T}_1}$, are therefore exclusively simple operations, usually with one or two $\mathbb{T}_0$ variables as their operands. The partial derivatives of $\mathbf{x}_{\mathbb{T}_1}$ with respect to $\mathbf{x}$ can be computed as

$$\frac{\partial \mathbf{x}_{\mathbb{T}_1}}{\partial \mathbf{x}} = \frac{\partial \mathbf{x}_{\mathbb{T}_1}}{\partial \mathbf{x}_{\mathbb{T}_0}}, \tag{3.39}$$

where each entry can be a new auxiliary variable denoting the partial derivative of the $\mathbb{T}_1$ variable in question with respect to a single one of its operands. These partial derivatives are trivial to ascertain due to each auxiliary variable being an atomic operation. The partial derivatives of $\mathbf{x}_{\mathbb{T}_0}$ with respect to $\mathbf{x}$ will remain as per eq. (3.36) due to the fact that tier variables may only have operands from lesser tiers. They are, therefore, completely unchanged by the addition of any greater tiers. $\mathbf{y}$ can be constructed using variables from $\mathbb{T}_0$ and $\mathbb{T}_1$ as its entries. Therefore, the Jacobian matrix of $\mathbf{y}$ with respect to $\mathbf{x}$ can be expressed as

$$\mathbf{G} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_0}} \frac{\partial \mathbf{x}_{\mathbb{T}_0}}{\partial \mathbf{x}} + \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_1}} \frac{\partial \mathbf{x}_{\mathbb{T}_1}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_0}} \frac{\partial \mathbf{x}_{\mathbb{T}_0}}{\partial \mathbf{x}_{\mathbb{T}_0}} + \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_1}} \frac{\partial \mathbf{x}_{\mathbb{T}_1}}{\partial \mathbf{x}_{\mathbb{T}_0}}, \tag{3.40}$$

where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_0}}$ and $\frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_1}}$ are matrices of ones and zeros with the ones corresponding to the locations in $\mathbf{y}$ that are populated by the required auxiliary variables from that tier. Note, therefore, that each row of $\frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_i}}$ can contain only a single one. Each column, however, may contain multiple ones as the same auxiliary variable may be used to populate more than one entry in $\mathbf{y}$, if such entries are like-expressions.

Thirdly, consider $\mathbf{y} = \mathbf{f}(\mathbf{x})$ now requiring $\{\mathbb{T}_0, \mathbb{T}_1, \mathbb{T}_2\}$ after having been tier-partitioned. The $\mathbb{T}_0$ and $\mathbb{T}_1$ partial derivative matrices remain defined by eq. (3.36) and eq. (3.39) respectively. The $\mathbb{T}_2$ partial derivative matrix can be expressed as

$$\frac{\partial \mathbf{x}_{\mathbb{T}_2}}{\partial \mathbf{x}} = \frac{\partial \mathbf{x}_{\mathbb{T}_2}}{\partial \mathbf{x}_{\mathbb{T}_0}} + \frac{\partial \mathbf{x}_{\mathbb{T}_2}}{\partial \mathbf{x}_{\mathbb{T}_1}} \frac{\partial \mathbf{x}_{\mathbb{T}_1}}{\partial \mathbf{x}_{\mathbb{T}_0}}, \tag{3.41}$$

which allows for all $\mathbb{T}_2$ variables to be functions of both $\mathbb{T}_0$ and $\mathbb{T}_1$ variables. Note that eq. (3.41) contains within it the *right hand side* (RHS) term of the $\mathbb{T}_1$ partial derivative matrix from eq. (3.39). The Jacobian matrix can this time be expressed as

$$\begin{aligned} \mathbf{G} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} &= \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_0}} \frac{\partial \mathbf{x}_{\mathbb{T}_0}}{\partial \mathbf{x}} + \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_1}} \frac{\partial \mathbf{x}_{\mathbb{T}_1}}{\partial \mathbf{x}} + \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_2}} \frac{\partial \mathbf{x}_{\mathbb{T}_2}}{\partial \mathbf{x}} \\ &= \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_0}} \frac{\partial \mathbf{x}_{\mathbb{T}_0}}{\partial \mathbf{x}_{\mathbb{T}_0}} + \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_1}} \frac{\partial \mathbf{x}_{\mathbb{T}_1}}{\partial \mathbf{x}_{\mathbb{T}_0}} + \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_2}} \left(\frac{\partial \mathbf{x}_{\mathbb{T}_2}}{\partial \mathbf{x}_{\mathbb{T}_0}} + \frac{\partial \mathbf{x}_{\mathbb{T}_2}}{\partial \mathbf{x}_{\mathbb{T}_1}} \frac{\partial \mathbf{x}_{\mathbb{T}_1}}{\partial \mathbf{x}_{\mathbb{T}_0}} \right). \end{aligned} \tag{3.42}$$

Note that the $\frac{\partial \mathbf{x}_{\mathbb{T}_0}}{\partial \mathbf{x}}$, $\frac{\partial \mathbf{x}_{\mathbb{T}_1}}{\partial \mathbf{x}}$ and $\frac{\partial \mathbf{x}_{\mathbb{T}_2}}{\partial \mathbf{x}}$ terms come from eqs. (3.36), (3.39) and (3.41) and may contain common terms, e.g. $\frac{\partial \mathbf{x}_{\mathbb{T}_1}}{\partial \mathbf{x}_{\mathbb{T}_0}}$.

Finally, consider scenarios where $\mathbf{y} = \mathbf{f}(\mathbf{x})$ requires further tiers following tier partitioning. For example, if $\mathbb{T}_3$ auxiliary variables were to be introduced, then the partial derivatives of these with respect to the input variables would be computed as

$$\frac{\partial \mathbf{x}_{\mathbb{T}_3}}{\partial \mathbf{x}} = \frac{\partial \mathbf{x}_{\mathbb{T}_3}}{\partial \mathbf{x}_{\mathbb{T}_0}} + \frac{\partial \mathbf{x}_{\mathbb{T}_3}}{\partial \mathbf{x}_{\mathbb{T}_1}} \frac{\partial \mathbf{x}_{\mathbb{T}_1}}{\partial \mathbf{x}_{\mathbb{T}_0}} + \frac{\partial \mathbf{x}_{\mathbb{T}_3}}{\partial \mathbf{x}_{\mathbb{T}_2}} \left(\frac{\partial \mathbf{x}_{\mathbb{T}_2}}{\partial \mathbf{x}_{\mathbb{T}_0}} + \frac{\partial \mathbf{x}_{\mathbb{T}_2}}{\partial \mathbf{x}_{\mathbb{T}_1}} \frac{\partial \mathbf{x}_{\mathbb{T}_1}}{\partial \mathbf{x}_{\mathbb{T}_0}} \right) . \tag{3.43}$$

The Jacobian matrix would now be expressed as

$$\mathbf{G} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_0}} \frac{\partial \mathbf{x}_{\mathbb{T}_0}}{\partial \mathbf{x}} + \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_1}} \frac{\partial \mathbf{x}_{\mathbb{T}_1}}{\partial \mathbf{x}} + \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_2}} \frac{\partial \mathbf{x}_{\mathbb{T}_2}}{\partial \mathbf{x}} + \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_3}} \frac{\partial \mathbf{x}_{\mathbb{T}_3}}{\partial \mathbf{x}} , \tag{3.44}$$

with $\frac{\partial \mathbf{x}_{\mathbb{T}_0}}{\partial \mathbf{x}}$, $\frac{\partial \mathbf{x}_{\mathbb{T}_1}}{\partial \mathbf{x}}$, $\frac{\partial \mathbf{x}_{\mathbb{T}_2}}{\partial \mathbf{x}}$ and $\frac{\partial \mathbf{x}_{\mathbb{T}_3}}{\partial \mathbf{x}}$ coming from eqs. (3.36), (3.39), (3.41) and (3.43).

It should now be clear that every time an additional tier is introduced, an additional partial derivative matrix must be produced and that this partial derivative matrix depends on all previous partial derivative matrices in a structured form. Furthermore, these partial derivative matrices contain common subexpressions. This is particularly true as the maximum tier number increases. This warrants the introduction of a new concept, termed *delta matrices*, which will assist with the algorithmic construction of the partial derivative matrices of tier variables and will allow for efficient reuse of common subexpressions. Consider replacing the terms $\frac{\partial \mathbf{x}_{\mathbb{T}_0}}{\partial \mathbf{x}}$ and $\frac{\partial \mathbf{x}_{\mathbb{T}_1}}{\partial \mathbf{x}}$ of eqs. (3.36) and (3.39) with the compact notations $\mathbf{\Delta}_{\mathbb{T}_0}$ and $\mathbf{\Delta}_{\mathbb{T}_1}$. These are the zeroth and first delta matrices. The $\frac{\partial \mathbf{x}_{\mathbb{T}_2}}{\partial \mathbf{x}}$ term of eq. (3.41), written more compactly using the new delta matrix notation as $\mathbf{\Delta}_{\mathbb{T}_2}$, can be defined using other delta matrices as

$$\mathbf{\Delta}_{\mathbb{T}_2} = \frac{\partial \mathbf{x}_{\mathbb{T}_2}}{\partial \mathbf{x}_{\mathbb{T}_0}} \mathbf{\Delta}_{\mathbb{T}_0} + \frac{\partial \mathbf{x}_{\mathbb{T}_2}}{\partial \mathbf{x}_{\mathbb{T}_1}} \mathbf{\Delta}_{\mathbb{T}_1} . \tag{3.45}$$

From this point on, a matrix in the form $\frac{\partial \mathbf{x}_{\mathbb{T}_j}}{\partial \mathbf{x}_{\mathbb{T}_i}}$ for $i, j \in \mathbb{Z}^+$ and $i < j$ will be termed a *partial derivative matrix*. Similarly, the $\frac{\partial \mathbf{x}_{\mathbb{T}_3}}{\partial \mathbf{x}}$ term of eq. (3.43), equivalent to $\mathbf{\Delta}_{\mathbb{T}_3}$, can be defined as

$$\mathbf{\Delta}_{\mathbb{T}_3} = \frac{\partial \mathbf{x}_{\mathbb{T}_3}}{\partial \mathbf{x}_{\mathbb{T}_0}} \mathbf{\Delta}_{\mathbb{T}_0} + \frac{\partial \mathbf{x}_{\mathbb{T}_3}}{\partial \mathbf{x}_{\mathbb{T}_1}} \mathbf{\Delta}_{\mathbb{T}_1} + \frac{\partial \mathbf{x}_{\mathbb{T}_3}}{\partial \mathbf{x}_{\mathbb{T}_2}} \mathbf{\Delta}_{\mathbb{T}_2} . \tag{3.46}$$

The pattern by which further delta matrices could be defined from previous delta matrices should now be apparent.

If the delta matrices are computed in increasing sequential order, starting with $\mathbf{\Delta}_{\mathbb{T}_1}$ (trivially, $\mathbf{\Delta}_{\mathbb{T}_0} = \mathbf{I}$), then later delta matrices can be computed using a summation of at most t matrix products where t is the delta matrix's tier number. One obvious potential inefficiency is that as the number of tiers increases, so does both the computational complexity and the memory complexity of producing each delta matrix. This point will be addressed later in section 3.4.

3.3.3 hSAD Equations

Collating the information described in section 3.3.2 and presenting it in concise mathematical form gives the hSAD equation

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \sum_{i=1}^{\tau} \frac{\partial \mathbf{f}}{\partial \mathbf{x}_{\mathbb{T}_i}} \mathbf{\Delta}_{\mathbb{T}_i} , \tag{3.47}$$

where τ is the number of tiers required,

$$\mathbf{\Delta}_{\mathbb{T}_0} = \mathbf{I} \tag{3.48}$$

and

$$\mathbf{\Delta}_{\mathbb{T}_i} = \sum_{j=0}^{i-1} \frac{\partial \mathbf{x}_{\mathbb{T}_i}}{\partial \mathbf{x}_{\mathbb{T}_j}} \mathbf{\Delta}_{\mathbb{T}_j} , \quad (i = 1, \ldots, \tau) \tag{3.49}$$

is the delta matrix equation. Equations (3.47) to (3.49), along with tier partitioning described in section 3.3.2 fully define the hSAD algorithm.

3.3.4 Worked Example

To illustrate the practical application of hSAD, a worked example using the lighthouse example is be presented. The first step in the hSAD algorithm is to ensure that the function to be differentiated, termed the *target function*, has been presented as a DAG, either directly or by construction from an ET. Both an ET and its corresponding DAG have already been presented for the lighthouse example, in table 3.1 and fig. 3.2 respectively. The second step involves the tier partitioning of the auxiliary variables. Again, this has already been done in section 3.3.2 with the partitioned tiers stated by eqs. (3.26) to (3.31).

The next step is to sequentially produce the delta matrices. As defined by eq. (3.48), the zeroth delta matrix, $\mathbf{\Delta}_{\mathbb{T}_0}$, is the 4×4 identity matrix $\mathbf{I}_{4\times4}$, with its dimension due to the cardinality of $x_{\mathbb{T}_0}$, $|x_{\mathbb{T}_0}| = 4$. The first delta matrix, $\mathbf{\Delta}_{\mathbb{T}_1}$, is produced trivially as it can only be a function of the $\mathbb{T}_0$ variables

$$\mathbf{\Delta}_{\mathbb{T}_1} = \frac{\partial \mathbf{x}_{\mathbb{T}_1}}{\partial \mathbf{x}_{\mathbb{T}_0}} \mathbf{\Delta}_{\mathbb{T}_0} = \begin{bmatrix} 0 & 0 & x_4 & x_3 \end{bmatrix} \mathbf{I}_{4\times4} = \begin{bmatrix} 0 & 0 & x_4 & x_3 \end{bmatrix} . \tag{3.50}$$

The second delta matrix, $\mathbf{\Delta}_{\mathbb{T}_2}$, is calculated as

$$\begin{aligned} \mathbf{\Delta}_{\mathbb{T}_2} &= \cancelto{\mathbf{0}_{1\times4}}{\frac{\partial \mathbf{x}_{\mathbb{T}_2}}{\partial \mathbf{x}_{\mathbb{T}_0}} \mathbf{\Delta}_{\mathbb{T}_0}} + \frac{\partial \mathbf{x}_{\mathbb{T}_2}}{\partial \mathbf{x}_{\mathbb{T}_1}} \mathbf{\Delta}_{\mathbb{T}_1} \\ &= \begin{bmatrix} \sec^2(w_1) \end{bmatrix} \begin{bmatrix} 0 & 0 & x_4 & x_3 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 & w_9 & w_{10} \end{bmatrix} , \end{aligned} \tag{3.51}$$

where the additional auxiliary substitutions are created

$$\begin{aligned}
w_7 &= \sec(\omega t) = \sec(w_1) = \mathrm{Sec}(w_1) \\
w_8 &= \sec^2(\omega t) = (w_7)^2 = \mathrm{Sqr}(w_7) \\
w_9 &= t\sec^2(\omega t) = x_4 w_8 = \mathrm{Mul}(x_4, w_8) \\
w_{10} &= \omega\sec^2(\omega t) = x_3 w_8 = \mathrm{Mul}(x_3, w_8) \ .
\end{aligned}$$

The third delta matrix, $\boldsymbol{\Delta}_{\mathbb{T}_3}$, is calculated as

$$\begin{aligned}
\boldsymbol{\Delta}_{\mathbb{T}_3} &= \frac{\partial \mathbf{x}_{\mathbb{T}_3}}{\partial \mathbf{x}_{\mathbb{T}_0}}\boldsymbol{\Delta}_{\mathbb{T}_0} + \cancelto{\mathbf{0}_{2\times 1}}{\frac{\partial \mathbf{x}_{\mathbb{T}_3}}{\partial \mathbf{x}_{\mathbb{T}_1}}\boldsymbol{\Delta}_{\mathbb{T}_1}} + \frac{\partial \mathbf{x}_{\mathbb{T}_3}}{\partial \mathbf{x}_{\mathbb{T}_2}}\boldsymbol{\Delta}_{\mathbb{T}_2} \\
&= \begin{bmatrix} 0 & 1 & 0 & 0 \\ w_2 & 0 & 0 & 0 \end{bmatrix} \mathbf{I}_{4\times 4} + \begin{bmatrix} -1 \\ x_1 \end{bmatrix} \begin{bmatrix} 0 & 0 & w_9 & w_{10} \end{bmatrix} \\
&= \begin{bmatrix} 0 & 1 & 0 & 0 \\ w_2 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & -w_9 & -w_{10} \\ 0 & 0 & x_1 w_9 & x_1 w_{10} \end{bmatrix} \\
&= \begin{bmatrix} 0 & 1 & w_{11} & w_{12} \\ w_2 & 0 & w_{13} & w_{14} \end{bmatrix} ,
\end{aligned} \tag{3.52}$$

where the additional auxiliary substitutions are created

$$\begin{aligned}
w_{11} &= -t\sec^2(\omega t) = -w_9 = \mathrm{Neg}(w_9) \\
w_{12} &= -\omega\sec^2(\omega t) = -w_{10} = \mathrm{Neg}(w_{10}) \\
w_{13} &= \nu t\sec^2(\omega t) = x_1 w_9 = \mathrm{Mul}(x_1, w_9) \\
w_{14} &= \nu\omega\sec^2(\omega t) = x_1 w_{10} = \mathrm{Mul}(x_1, w_{10}) \ .
\end{aligned}$$

The fourth delta matrix, $\boldsymbol{\Delta}_{\mathbb{T}_4}$, is calculated as

$$\begin{aligned}
\boldsymbol{\Delta}_{\mathbb{T}_4} &= \cancelto{\mathbf{0}_{1\times 4}}{\frac{\partial \mathbf{x}_{\mathbb{T}_4}}{\partial \mathbf{x}_{\mathbb{T}_0}}\boldsymbol{\Delta}_{\mathbb{T}_0}} + \cancelto{\mathbf{0}_{1\times 1}}{\frac{\partial \mathbf{x}_{\mathbb{T}_4}}{\partial \mathbf{x}_{\mathbb{T}_1}}\boldsymbol{\Delta}_{\mathbb{T}_1}} + \cancelto{\mathbf{0}_{1\times 1}}{\frac{\partial \mathbf{x}_{\mathbb{T}_4}}{\partial \mathbf{x}_{\mathbb{T}_2}}\boldsymbol{\Delta}_{\mathbb{T}_2}} + \frac{\partial \mathbf{x}_{\mathbb{T}_4}}{\partial \mathbf{x}_{\mathbb{T}_3}}\boldsymbol{\Delta}_{\mathbb{T}_3} \\
&= \begin{bmatrix} -\frac{w_4}{w_3^2} & \frac{1}{w_3} \end{bmatrix} \begin{bmatrix} 0 & 1 & w_{11} & w_{12} \\ w_2 & 0 & w_{13} & w_{14} \end{bmatrix} \\
&= \begin{bmatrix} w_{17} & w_{15} \end{bmatrix} \begin{bmatrix} 0 & 1 & w_{11} & w_{12} \\ w_2 & 0 & w_{13} & w_{14} \end{bmatrix} \\
&= \begin{bmatrix} w_2 w_{15} & w_{17} & w_{11} w_{17} + w_{13} w_{15} & w_{12} w_{17} + w_{14} w_{15} \end{bmatrix} \\
&= \begin{bmatrix} w_{18} & w_{17} & w_{19} + w_{20} & w_{21} + w_{22} \end{bmatrix} \\
&= \begin{bmatrix} w_{18} & w_{17} & w_{23} & w_{24} \end{bmatrix} ,
\end{aligned} \tag{3.53}$$

where the additional auxiliary substitutions are created

$$
\begin{aligned}
w_{15} &= \frac{1}{\gamma - \tan(\omega t)} = \frac{1}{w_3} = \text{Recip}(w_3) \\
w_{16} &= \frac{\nu \tan(\omega t)}{(\gamma - \tan(\omega t))^2} = w_5 w_{15} = \text{Mul}(w_5, w_{15}) \\
w_{17} &= -\frac{\nu \tan(\omega t)}{(\gamma - \tan(\omega t))^2} = -w_{16} = \text{Neg}(w_{16}) \\
w_{18} &= \frac{\tan(\omega t)}{\gamma - \tan(\omega t)} = w_2 w_{15} = \text{Mul}(w_2, w_{15}) \\
w_{19} &= -\frac{\nu t \tan(\omega t) \sec^2(\omega t)}{(\gamma - \tan(\omega t))^2} = w_{11} w_{17} = \text{Mul}(w_{11}, w_{17}) \\
w_{20} &= \frac{\nu t \sec^2(\omega t)}{\gamma - \tan(\omega t)} = w_{13} w_{15} = \text{Mul}(w_{13}, w_{15}) \\
w_{21} &= \frac{\nu \omega \tan(\omega t) \sec^2(\omega t)}{(\gamma - \tan(\omega t))^2} = w_{12} w_{17} = \text{Mul}(w_{12}, w_{17}) \\
w_{22} &= \frac{\nu \omega \sec^2(\omega t)}{\gamma - \tan(\omega t)} = w_{14} w_{15} = \text{Mul}(w_{14}, w_{15}) \\
w_{23} &= -\frac{\nu t \tan(\omega t) \sec^2(\omega t)}{(\gamma - \tan(\omega t))^2} + \frac{\nu t \sec^2(\omega t)}{\gamma - \tan(\omega t)} = w_{19} + w_{20} = \text{Add}(w_{19}, w_{20}) \\
w_{24} &= \frac{\nu \omega \tan(\omega t) \sec^2(\omega t)}{(\gamma - \tan(\omega t))^2} + \frac{\nu \omega \sec^2(\omega t)}{\gamma - \tan(\omega t)} = w_{21} + w_{22} = \text{Add}(w_{21}, w_{22}) \ .
\end{aligned}
$$

The fifth and final delta matrix, $\boldsymbol{\Delta}_{\mathbb{T}_5}$, is calculated as

$$
\begin{aligned}
\boldsymbol{\Delta}_{\mathbb{T}_5} &= \frac{\partial \mathbf{x}_{\mathbb{T}_5}}{\partial \mathbf{x}_{\mathbb{T}_0}} \boldsymbol{\Delta}_{\mathbb{T}_0} + \cancelto{\mathbf{0}_{1\times 1}}{\frac{\partial \mathbf{x}_{\mathbb{T}_5}}{\partial \mathbf{x}_{\mathbb{T}_1}}} \boldsymbol{\Delta}_{\mathbb{T}_1} + \cancelto{\mathbf{0}_{1\times 1}}{\frac{\partial \mathbf{x}_{\mathbb{T}_5}}{\partial \mathbf{x}_{\mathbb{T}_2}}} \boldsymbol{\Delta}_{\mathbb{T}_2} + \cancelto{\mathbf{0}_{2\times 1}}{\frac{\partial \mathbf{x}_{\mathbb{T}_5}}{\partial \mathbf{x}_{\mathbb{T}_3}}} \boldsymbol{\Delta}_{\mathbb{T}_3} + \frac{\partial \mathbf{x}_{\mathbb{T}_5}}{\partial \mathbf{x}_{\mathbb{T}_4}} \boldsymbol{\Delta}_{\mathbb{T}_4} \\
&= \begin{bmatrix} 0 & w_5 & 0 & 0 \end{bmatrix} \mathbf{I}_{4\times 4} + \begin{bmatrix} x_2 \end{bmatrix} \begin{bmatrix} w_{18} & w_{17} & w_{23} & w_{24} \end{bmatrix} \\
&= \begin{bmatrix} 0 & w_5 & 0 & 0 \end{bmatrix} + \begin{bmatrix} w_{25} & w_{26} & w_{27} & w_{28} \end{bmatrix} \\
&= \begin{bmatrix} w_{25} & w_{29} & w_{27} & w_{28} \end{bmatrix} \ ,
\end{aligned}
\tag{3.54}
$$

where the additional auxiliary substitutions are created

$$
\begin{aligned}
w_{25} &= \frac{\gamma \tan(\omega t)}{\gamma - \tan(\omega t)} = x_2 w_{18} = \text{Mul}(x_2, w_{18}) \\
w_{26} &= -\frac{\nu \gamma \tan(\omega t)}{(\gamma - \tan(\omega t))^2} = x_2 w_{17} = \text{Mul}(x_2, w_{17}) \\
w_{27} &= -\frac{\nu \gamma t \tan(\omega t) \sec^2(\omega t)}{(\gamma - \tan(\omega t))^2} + \frac{\nu \gamma t \sec^2(\omega t)}{\gamma - \tan(\omega t)} = x_2 w_{23} = \text{Mul}(x_2, w_{23}) \\
w_{28} &= \frac{\nu \gamma \omega \tan(\omega t) \sec^2(\omega t)}{(\gamma - \tan(\omega t))^2} + \frac{\nu \gamma \omega \sec^2(\omega t)}{\gamma - \tan(\omega t)} = x_2 w_{24} = \text{Mul}(x_2, w_{24}) \\
w_{29} &= \frac{\nu \tan(\omega t)}{\gamma - \tan(\omega t)} - \frac{\nu \gamma \tan(\omega t)}{(\gamma - \tan(\omega t))^2} = w_5 + w_{26} = \text{Add}(w_5, w_{26}) \ .
\end{aligned}
$$

The Jacobian matrix is finally expressed in its full form using the hSAD equation (eq. (3.47)) as

$$\begin{aligned}\mathbf{G} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} &= \cancel{\frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_0}}}^{\mathbf{0}_{2\times 4}} \boldsymbol{\Delta}_{\mathbb{T}_0} + \cancel{\frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_1}}}^{\mathbf{0}_{2\times 1}} \boldsymbol{\Delta}_{\mathbb{T}_1} + \cancel{\frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_2}}}^{\mathbf{0}_{2\times 1}} \boldsymbol{\Delta}_{\mathbb{T}_2} + \cancel{\frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_3}}}^{\mathbf{0}_{2\times 2}} \boldsymbol{\Delta}_{\mathbb{T}_3} \\ &\quad + \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_4}} \boldsymbol{\Delta}_{\mathbb{T}_4} + \frac{\partial \mathbf{y}}{\partial \mathbf{x}_{\mathbb{T}_5}} \boldsymbol{\Delta}_{\mathbb{T}_5} \\ &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} w_{18} & w_{17} & w_{23} & w_{24} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} w_{25} & w_{29} & w_{27} & w_{28} \end{bmatrix} \\ &= \begin{bmatrix} w_{18} & w_{17} & w_{23} & w_{24} \\ w_{25} & w_{29} & w_{27} & w_{28} \end{bmatrix}.\end{aligned} \tag{3.55}$$

Comparing the eight entries of $\mathbf{G}$ generated using hSAD (eq. (3.55)) to their corresponding expressions found using MD (eqs. (3.4) to (3.11)) shows exact agreement in their analytic representations. This illustrates how hSAD is able to produce partial derivative matrices whose entries are exact analytical derivatives.

In this example, the hSAD pass generated 23 new auxiliary variables, w_7 to w_{29}. This is in addition to the six auxiliary variables, w_1 to w_6, that were required to represent the ET of the target function. The 23 new auxiliary variables can be added to the target function's expression graph as additional auxiliary nodes. The matrix output by hSAD, $\mathbf{G}$ (eq. (3.55)), can be used to generate an ET for the computation of the target function's Jacobian. Such an ET is shown in table 3.4. This ET can, in turn, be used to generate computer code which can be compiled to produce a callable function. The computer code can then be used to numerically evaluate the Jacobian at a desired set of inputs.

3.4 Complexity and Properties

3.4.1 Computational Complexity

In order to assess the practical applicability of hSAD, its computational and memory complexity must be compared to that of established methods. Forward-mode AD, which has been shown to be functionally equivalent to hyper-dual approximation [113], will primarily be considered due to its proven performance when applied to numerically solving OCPs [10]. To derive the computational complexity of both forward-mode AD and hSAD, a generic vector function $\mathbf{y} = \mathbf{f}(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$ will be considered. Explicitly, $\mathbf{f}$ has the n input variables $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ and m

Independent Variables
$x_1 = \nu$
$x_2 = \gamma$
$x_3 = \omega$
$x_4 = t$
Auxiliary Variables
$w_1 = x_3 x_4 = \text{Mul}(x_3, x_4)$
$w_2 = \tan(w_1) = \text{Tan}(w_1)$
$w_3 = x_2 - w_2 = \text{Sub}(x_2, w_2)$
$w_4 = x_1 w_2 = \text{Mul}(x_1, w_2)$
$w_5 = \frac{w_4}{w_3} = \text{Div}(w_4, w_3)$
$w_6 = x_2 w_5 = \text{Mul}(x_2, w_5)$
$w_7 = \sec(w_1) = \text{Sec}(w_1)$
$w_8 = (w_7)^2 = \text{Sqr}(w_7)$
$w_9 = x_4 w_8 = \text{Mul}(x_4, w_8)$
$w_{10} = x_3 w_8 = \text{Mul}(x_3, w_8)$
$w_{11} = -w_9 = \text{Neg}(w_9)$
$w_{12} = -w_{10} = \text{Neg}(w_{10})$
$w_{13} = x_1 w_9 = \text{Mul}(x_1, w_9)$
$w_{14} = x_1 w_{10} = \text{Mul}(x_1, w_{10})$
$w_{15} = \frac{1}{w_3} = \text{Recip}(w_3)$
$w_{16} = w_5 w_{15} = \text{Mul}(w_5, w_{15})$
$w_{17} = -w_{16} = \text{Neg}(w_{16})$
$w_{18} = w_2 w_{15} = \text{Mul}(w_2, w_{15})$
$w_{19} = w_{11} w_{17} = \text{Mul}(w_{11}, w_{17})$
$w_{20} = w_{13} w_{15} = \text{Mul}(w_{13}, w_{15})$
$w_{21} = w_{12} w_{17} = \text{Mul}(w_{12}, w_{17})$
$w_{22} = w_{14} w_{15} = \text{Mul}(w_{14}, w_{15})$
$w_{23} = w_{19} + w_{20} = \text{Add}(w_{19}, w_{20})$
$w_{24} = w_{21} + w_{22} = \text{Add}(w_{21}, w_{22})$
$w_{25} = x_2 w_{18} = \text{Mul}(x_2, w_{18})$
$w_{26} = x_2 w_{17} = \text{Mul}(x_2, w_{17})$
$w_{27} = x_2 w_{23} = \text{Mul}(x_2, w_{23})$
$w_{28} = x_2 w_{24} = \text{Mul}(x_2, w_{24})$
$w_{29} = w_5 + w_{26} = \text{Add}(w_5, w_{26})$
Dependent Variables
$G_{11} = w_{18}$
$G_{12} = w_{17}$
$G_{13} = w_{23}$
$G_{14} = w_{24}$
$G_{21} = w_{25}$
$G_{22} = w_{29}$
$G_{23} = w_{27}$
$G_{24} = w_{28}$

Table 3.4: Jacobian ET for the lighthouse example using hSAD.

output variables $\mathbf{y} = [y_1, y_2, \ldots, y_n]$. $\mathbf{f}$ can be considered to have been expressed as either an analytic function or as source code in a computer program, such that it is decomposable to an ET of L atomic operations. These ordered atomic operations correspond to a set of L auxiliary variables, $\{w_1, w_2, \ldots, w_L\}$. All derivative-taking methods aim to provide a way of numerically evaluating the Jacobian matrix of $\mathbf{y}$ with respected to $\mathbf{x}$,

$$\mathbf{G} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}}, \tag{3.56}$$

at a specific value of the input variables, $\mathbf{x}_0$. Let $\hat{c}$ denote the average cost of evaluating a generic auxiliary variable, w_k, such that the cost of evaluating $\mathbf{f}$ is $\hat{c}L$.

To compute $\mathbf{G}$ using forward-mode AD, a cost of cnL relative to the cost of computing the target function alone is incurred [136]. c is typically a small constant where $1 \leq c \leq 4$ [136]. Similarly, the cost of computing $\mathbf{G}$ using reverse-mode AD is cmL relative to the cost of computing the target function alone [136].

Derivation for hSAD

hSAD is functionally equivalent to AD because it similarly generates an ET for the target function's partial derivatives by repeated application of the chain rule to the target function. It, however, does this symbolically in a preprocessing step rather than numerically at runtime.

Using the same terminology defined at the beginning of section 3.4.1, assume that the auxiliary variables associated with $\mathbf{f}$, w_k for $k = (1, \ldots, L)$ have undergone tier partitioning and have been partitioned in to τ tiers, $\{\mathbb{T}_1, \mathbb{T}_2, \ldots, \mathbb{T}_\tau\}$. For the full Jacobian matrix to be populated, delta matrices for each tier, $\{\boldsymbol{\Delta}_{\mathbb{T}_1}, \boldsymbol{\Delta}_{\mathbb{T}_2}, \ldots, \boldsymbol{\Delta}_{\mathbb{T}_\tau}\}$, need to be produced. Each delta matrix effectively expresses the total derivatives of all of its tier's auxiliary variables with respect to the target function's input variables. In other words, $\boldsymbol{\Delta}_{\mathbb{T}_t}$ expresses the sensitivities of all auxiliary variables in $\mathbb{T}_t$ with respect to $\mathbf{x}$.

Due to the hierarchical nature of the algorithm, the computational complexity of hSAD can be shown by determining the computational complexity of computing a single auxiliary variable and extrapolating this across all L auxiliary variables. This is because the influence of a single auxiliary variable on the construction of the final derivative matrix only involves auxiliary variables from lower-precedence tiers, which have already been analysed and manipulated. Consider the auxiliary variable w_K, which maps to an atomic operation F_K. Following tier partitioning, w_K is in $\mathbb{T}_T$, in which it is the pth variable. w_K exists in the target function's expression graph and can be connected at the beginning of a directed edge (i.e.

its operation has lower-precedence variables as its operands) and at the end of a direct edge (i.e. it itself is an operand for a higher-precedence auxiliary variable's operation). w_K-related terms will, therefore, appear in the delta matrix equations in two places.

Firstly, w_K-related terms can appear in the pth rows of $\frac{\partial \mathbf{x}_{\mathbb{T}_T}}{\partial \mathbf{x}_{\mathbb{T}_t}}$ for $t = (0, 1, \ldots, T-1)$ in the $\mathbf{\Delta}_{\mathbb{T}_T}$ equation. In these cases, the terms are partial derivatives of F_K with respect to its operands. There are only as many terms of this type as there are operands of F_K. For example, if F_K is a unary operation with the sole operand w_a, only one partial derivative term corresponding to w_K, $\frac{\partial F_K(w_a)}{\partial w_a}$, will appear across all pth rows in all $\frac{\partial \mathbf{x}_{\mathbb{T}_T}}{\partial \mathbf{x}_{\mathbb{T}_t}}$ in the $\mathbf{\Delta}_{\mathbb{T}_T}$ equation. Alternatively, if F_K is a binary operation with the operands w_a and w_b, then two partial derivative terms corresponding to w_K, $\frac{\partial F_K(w_a,w_b)}{\partial w_a}$ and $\frac{\partial F_K(w_a,w_b)}{\partial w_b}$, will appear across all pth rows in $\frac{\partial \mathbf{x}_{\mathbb{T}_T}}{\partial \mathbf{x}_{\mathbb{T}_t}}$ for $t = (0, 1, \ldots, T-1)$ in the $\mathbf{\Delta}_{\mathbb{T}_T}$ equation.

Secondly, w_K-related terms can appear in the pth columns of $\frac{\partial \mathbf{x}_{\mathbb{T}_t}}{\partial \mathbf{x}_{\mathbb{T}_T}}$ for $t = (T+1, \ldots, \tau-1, \tau)$ in the $\left\{\mathbf{\Delta}_{\mathbb{T}_{T+1}}, \ldots, \mathbf{\Delta}_{\mathbb{T}_{\tau-1}}, \mathbf{\Delta}_{\mathbb{T}_\tau}\right\}$ equations. In these cases, the terms are partial derivatives of atomic operations, in which w_K is an operand, with respect to w_K. Here, however, w_K is treated as a variable rather than a composite function and, therefore, computations involving it do not directly increase the computational cost associated with the calculation of w_K. It follows that, to analyse the computational complexity associated with w_K, only the former of the two scenarios needs to be considered.

The cost associated with w_K arises entirely from the delta matrix equation corresponding to the computation of $\mathbf{\Delta}_{\mathbb{T}_T}$, that is

$$\mathbf{\Delta}_{\mathbb{T}_T} = \frac{\partial \mathbf{x}_{\mathbb{T}_T}}{\partial \mathbf{x}_{\mathbb{T}_0}} \mathbf{\Delta}_{\mathbb{T}_0} + \frac{\partial \mathbf{x}_{\mathbb{T}_T}}{\partial \mathbf{x}_{\mathbb{T}_1}} \mathbf{\Delta}_{\mathbb{T}_1} + \ldots + \frac{\partial \mathbf{x}_{\mathbb{T}_T}}{\partial \mathbf{x}_{\mathbb{T}_{T-1}}} \mathbf{\Delta}_{\mathbb{T}_{T-1}} \,. \tag{3.57}$$

To reiterate, a delta matrix expresses the total derivatives of all of a tier's auxiliary variables with respect to all of the target function's input variables. Therefore, the pth row and only the pth row of $\mathbf{\Delta}_{\mathbb{T}_T}$ is associated with w_K and w_K exclusively. Thus, all other rows of $\mathbf{\Delta}_{\mathbb{T}_T}$ can be ignored, and only the pth rows of $\frac{\partial \mathbf{x}_{\mathbb{T}_T}}{\partial \mathbf{x}_{\mathbb{T}_t}}$ and pth columns of $\mathbf{\Delta}_{\mathbb{T}_t}$ for $t = (0, 1, \ldots, T-1)$ need to be considered in relation to the cost associated with w_K.

Firstly, consider the simplest scenario where w_K corresponds to the unary operation $F_K(w_a)$. In this case, F_K has only one nonzero partial derivative, $\frac{\partial F_K(w_a)}{\partial w_a}$. Therefore, across all pth rows of $\frac{\partial \mathbf{x}_{\mathbb{T}_T}}{\partial \mathbf{x}_{\mathbb{T}_t}}$ for $t = (0, 1, \ldots, T-1)$, there will be only a single nonzero entry. If $\mathbf{A}$ and $\mathbf{B}$ are matrices, evaluating M_{ij} (the element in the ith row and jth column of $\mathbf{M}$) where $\mathbf{M} = \mathbf{AB}$ requires the dot product between the ith row of $\mathbf{A}$ and the jth column of $\mathbf{B}$. If the ith row of $\mathbf{A}$ contains

only one nonzero, then evaluating M_{ij} will require only a single scalar multiplication, provided that calculations involving zeros can be ignored, as is the case when sparse matrix representations and arithmetic are used. Therefore, in this case, each element in the pth row of $\boldsymbol{\Delta}_{\mathbb{T}_T}$, of which there will be n, can be determined with a single multiplication. Note that in the context of hSAD, a single multiplication corresponds to creating a new auxiliary variable associated with a multiplication operation and associating that new auxiliary variable with the expression graph. Thus, the total cost associated with w_K is the cost of adding $n + \gamma$ new auxiliary variables to the expression graph, with γ denoting the average number of additional operations required to compute a single partial derivative of an auxiliary variable. n of these operations are binary multiplications, with the remaining γ operations being those associated with $\frac{\partial F_K(w_a)}{\partial w_a}$.

Next, consider the scenario where w_K corresponds to the binary operation $F_K(w_a, w_b)$. Here, F_K has two nonzero partial derivatives, $\frac{\partial F_K(w_a,w_b)}{\partial w_a}$ and $\frac{\partial F_K(w_a,w_b)}{\partial w_b}$, and there would now be two nonzero entries across all pth rows of $\frac{\partial \mathbf{x}_{\mathbb{T}_T}}{\partial \mathbf{x}_{\mathbb{T}_t}}$ for $t = (0, 1, \ldots, T-1)$. If w_a and w_b are in the same tier, then the $\frac{\partial F_K(w_a,w_b)}{\partial w_a}$ and $\frac{\partial F_K(w_a,w_b)}{\partial w_b}$ terms will appear in the same $\frac{\partial \mathbf{x}_{\mathbb{T}_T}}{\partial \mathbf{x}_{\mathbb{T}_t}}$ matrix. Evaluating $M_{i,j}$, where $\mathbf{M} = \mathbf{AB}$, with two nonzero entries in the ith row of $\mathbf{A}$ now requires two scalar multiplications and a single scalar addition. Therefore, each element in the pth row of $\boldsymbol{\Delta}_{\mathbb{T}_T}$ can be determined with three simple binary arithmetic operations. If w_a and w_b were in different tiers, the cost would still be the same because a scalar multiplications corresponding to each of the two matrix multiplications would still be required, together with a single scalar addition corresponding to the matrix addition of the two product matrices from the two matrix multiplications. Thus, the total cost associated with w_K would be the cost of adding $3n + 2\gamma$ new auxiliary variables to the expression graph, $2n$ of which are binary multiplication operations, n of which are binary addition operations and 2γ of which are the operations associated with $\frac{\partial F_K(w_a,w_b)}{\partial w_a}$ and $\frac{\partial F_K(w_a,w_b)}{\partial w_b}$.

Finally, consider the scenario where w_K corresponds to an operation with $\omega \geq 3$ operands, $F_K(w_a, w_b, \ldots, w_\omega)$. Following the previous arguments, the total cost associated with w_K would instead be $(2\omega - 1)n + \omega\gamma$. ωn correspond to binary multiplications, $(\omega - 1)n$ correspond to binary additions and the remaining $\omega\gamma$ correspond to the operations associated with each of the ω partial derivatives.

For the target function as a whole, conducting the hSAD pass involves extrapolating the complexity associated with w_K across all L auxiliary variables. Let Ω denote the average number of operands across all auxiliary variable operations. Typically $1.0 < \Omega < 2.0$ as the vast majority of atomic operations will be either unary or binary. Therefore, the cost of the hSAD pass will involve the creation of

$((2\Omega - 1)\, n + \Omega\gamma)\, L$ new auxiliary nodes, of which $(2\Omega - 1)\, n$ correspond to simple binary operations and $\Omega\gamma$ correspond to the generation of partial derivative expressions. Typically $0 \leq \gamma \leq 3$ [20, 136]. Thus, the cost, c_{hSAD}, is $\mathcal{O}(nL)$, the same as for forward-mode AD.

In addition to the cost of conducting the hSAD pass, there will also be a computational cost associated with numerically evaluating the resulting Jacobian. This will require numerically evaluating all of the nodes in the modified expression graph (i.e. the expression graph as it is after the hSAD pass) as all of these will be required, either directly or indirectly, to populate the Jacobian. If compiled to a callable function, this function will involve $((2\Omega - 1)\, n + \Omega\gamma + 1)\, L$ floating point operations.

3.4.2 Memory Complexity

hSAD operates on a target function's DAG and involves conducting the hSAD pass to modify the DAG to contain all of the new auxiliary variables required to exactly populate the Jacobian. This modified DAG can then be used to produce an ET for the Jacobian, which in turn can be compiled to a callable that numerically evaluates the matrix. The memory complexity of hSAD, therefore, involves two components: the memory requirement for carrying out the matrix arithmetic to conduct the hSAD pass; and the memory requirement for storing the modified DAG.

These two components are, however, intrinsically linked as the hSAD pass is itself responsible for all modifications made to the DAG. Each distinct step in the hSAD pass involves computing a new delta matrix for each successive tier until all tiers have an associated delta matrix. Following this, the Jacobian can be populated using all of the delta matrices and the hSAD equation. Each delta matrix equation contains two types of matrices: partial derivative matrices and lower-tier delta matrices. The nonzeros within all of these matrices directly correspond to new variables that are required within the modified DAG. Therefore, all that needs to be considered in relation to the delta matrix and hSAD equations is the memory complexity of the structuring of these matrices. The memory complexity of the modified DAG can be considered in place of the memory cost associated with each nonzero.

The partial derivative matrices together detail the sensitivities of all of a tier's auxiliary variables with respect to their respective operands. The number of nonzeros across all of a single tier's derivative matrices corresponds to the number of directed edges entering that tier. If a tier contains on average $\frac{L}{\tau}$ auxiliary variables

and each auxiliary variable has on average Ω operands (i.e. is the head of on average Ω directed edges), then each tier has on average $\frac{\Omega L}{\tau}$ nonzeros across all of its partial derivative matrices. Across all tiers, there will be ΩL nonzeros across all partial derivative matrices. However, once the delta matrix for a tier has been calculated, the partial derivative matrices for that tier are never again required for use in a later calculation. Therefore, after this point, a tier's partial derivative matrices can be discarded and their memory freed. As partial derivative matrices have a lifetime corresponding to the time taken for their associated delta matrix calculation, the maximum memory usage associated with partial derivative matrices will be in the delta matrix calculation for the tier that has the most directed edges entering it.

The delta matrices on the RHS of a delta matrix equation contain the absolute partial derivatives of the tier's auxiliary variables operation's operands with respect to the target function's input variables. The number of nonzeros across all of these corresponds to the reachability of each auxiliary variable node from each input variable node. In an average tier containing $\frac{L}{\tau}$ auxiliary variables, the delta matrix will have dimensions $nL \times \tau$. The point at which hSAD has the largest delta matrix-related memory requirement is during the computation of the hSAD equation, in which all delta matrices are required. Therefore, at this point there are nL total matrix entries. Note, however, that it is unlikely that the DAG will be fully interconnected and there will, therefore, be fewer than nL entries across all delta matrices. If a sparse matrix representation is used for the delta matrices, then each of the delta matrix nonzeros can be expressed be three pieces of information: a pointer to the corresponding node in the expression graph; and two location identifiers (e.g. a row and column index). In either case, the memory complexity associated with the delta matrices will be $\mathcal{O}(nL)$.

Section 3.4.1 previously showed that the modified DAG will be required to contain $((2\Omega - 1)n + \Omega)L$ new auxiliary variables, in addition to the L variables in the target function's DAG. Therefore, the memory complexity of the modified DAG, μ_{hSAD}, is $\mu_{\text{hSAD}} = ((2\Omega - 1)n + \Omega + 1)\mu L$ where μ is the memory requirement of storing all the information relating to a single node in an expression graph (e.g. its operation, operands, tier and delta matrix indices). Again, as Ω is a small constant, the memory complexity of hSAD can be considered to be $\mathcal{O}(\mu nL)$.

3.4.3 Case Performance

The actual performance of hSAD will depend on the specific target function to which it is being applied and the structure of the target function's DAG. As computational and memory complexity have both been shown to be $\mathcal{O}(nL)$, general performance

of hSAD will scale linearly with both n and L. However, the absolute performance will depend on the operations associated with each node in the target function's DAG and interconnectivity between these nodes. Specifically, the reachability of the output nodes from the input nodes is of particular importance as these determine the structural sparsity of the resulting Jacobian.

The specific operations associated with each auxiliary variable influence both computational and memory complexity as they determine how many additional nodes are required in the modified DAG to express each node's operation's partial derivatives with respect to said node's operands. There are two aspects to complexity relating to specific operations. Firstly, certain operations generate more complex derivatives and, therefore, require more additional nodes in the modified DAG to express them. For example, an exponential function, $\exp(x)$, is the simplest such operation as its partial derivative, $\frac{\partial}{\partial x}(\exp(x)) = \exp(x)$, is itself and so can be expressed without the need to create additional nodes in the modified DAG. Conversely, complex trigonometric functions, such as $\tanh(x)$ or $\tan^{-1}(x)$, typically require three additional nodes to be added to the modified DAG to correctly express their partial derivatives. Secondly, operations with more operands will exhibit worse performance as they will have more nonzero partial derivative terms in their partial derivative matrices and will, therefore, grow the modified DAG further. Therefore, performance of hSAD will be superior for a target function containing mostly auxiliary variables associated with simple unary and binary operations (such as additions, multiplications and natural exponentials) in comparison to one containing many complex trigonometric and multiple-operand operations. In this respect, the benefit hSAD offers relative to AD is that these expensive portions of the DAG only need to be computed once. With AD, they may need to be recomputed on each pass causing expensive and unnecessary recalculation.

Reachability of nodes in the DAG directly influences the performance of hSAD in a specific case as this determines the sparsity of the partial derivative matrices and, more importantly, the delta matrices. If the target function's DAG is less interconnected, the delta matrices will contain more nonzero elements and the delta matrix equations can be computed more efficiently using sparse matrix arithmetic. hSAD is, therefore, most efficient when applied to target functions with low interconnectivity such that each output variable is only a function of a minor subset of the input variables. If this is the case, then the complexity of hSAD can approach $\mathcal{O}(L)$ and offer a significant performance benefit over AD.

3.4.4 Properties of hSAD

The primary purpose of hSAD is to provide a means of evaluating the exact first-order partial derivatives of a target vector function with respect to its inputs. Additional to this, hSAD has many beneficial properties which allow it to perform better than AD.

In AD, determination of a Jacobian's structural sparsity is conducted as a step separate to the AD processing of the target function, with a graph colouring algorithm [125] usually being used. With hSAD, exact structural sparsity information is contained within the delta matrices. An element can be known to be structurally zero because its entry will not contain a named auxiliary variable. Therefore, when the Jacobian is constructed by the hSAD equation, all structural zeros are known. This exact structural sparsity information is generated in tandem with the information for correctly evaluating each nonzero entry during the hSAD pass. Therefore, the additional graph colouring preprocessing step is not required with hSAD.

Another beneficial property of hSAD is that it generates a set of auxiliary variables that can be used to compute all of the partial derivatives in a Jacobian exactly, as well as a mapping of which auxiliary variables populate which entries in the Jacobian. It, therefore, allows an ET for the target function's Jacobian to be produced, that will populate all nonzeros in a single pass. This has the benefit of ensuring that, when an executable corresponding to the ET is compiled, auxiliary variables are reused in places where partial derivatives contain common subexpressions. In addition, due to the fact that the entire Jacobian can be populated from a single ET, the scenario encountered in AD where the same auxiliary variables are recomputed on successive passes with different seeds does not occur. Combined, these two factors result in hSAD being able to produce theoretically more computationally efficient numerical evaluation compared to AD.

hSAD applies the chain rule to the target function in a symbolic manner. In AD, where seeds are 0.0 and 1.0 are propagated through the target function numerous times per derivative evaluation, multiplications by 0.0 and 1.0 are frequently carried out. Due to the nature of how AD is conducted, however, by seeding these tangent or adjoint variables at runtime, it is not possible to remove them and AD is consigned to containing these inefficiencies. The symbolic application of the chain rule in hSAD has the beneficial side effect that certain trivial operations are removed automatically through the matrix arithmetic. For example, in each delta matrix calculation, numerous matrices are multiplied in pairs and their results summed (eq. (3.49)). When a zero is present in one of these matrices, the symbolic arithmetic ensures that the trivial zeros are not accumulated. This results in the modified DAG of

hSAD, and any resulting compiled callable, not containing these trivially-simplified wasteful operations.

Another benefit of the symbolic expression graph and its creation solely during preprocessing is that CAS expression simplification can be carried out. Using techniques of CASs, it is possible to simplify certain expressions to reduce the mathematical complexity of the DAG. This could be by removing trivial expressions, such as multiplications by 1.0. Alternatively, it could be by replacing known identities where there is a more simple expression available, such as replacing a set of auxiliary variables expressing $\sin^2(x) + \cos^2(x)$ with an auxiliary constant 1.0. This would, however, increase the complexity associated with the hSAD preprocessing step because, as with all CAS expression rewriting, efficiently analysing the expression graph to find these possible simplifications without human insight is a complex problem.

In summary, hSAD:

1. generates an evaluation procedure for evaluating all nonzero entries in the Jacobian of a target function with respect to its inputs;
2. ascertains the Jacobian's exact structural sparsity during preprocessing;
3. allows for all nonzeros entries to be computed in a single evaluation pass; and
4. eliminates mathematically trivial calculations, including multiplications by 0.0 and 1.0.

The Lighthouse Example

The beneficial properties of hSAD that have just been stated can be illustrated in the context of an example. Firstly, section 3.3.4 showed that the exact Jacobian could be produced using hSAD. The resulting Jacobian was compared to the Jacobian found through MD by hand and shown to be identical.

Secondly, the resulting Jacobian was produced with an exact sparsity structure. This is somewhat inconsequential in this case as the expressions for both of the outputs each contain all four of the input variables which results in a dense Jacobian. However, many of the delta matrices ($\{\boldsymbol{\Delta}_{\mathbb{T}_1}, \boldsymbol{\Delta}_{\mathbb{T}_2}, \boldsymbol{\Delta}_{\mathbb{T}_3}\}$) do contain structural zeros. If the output were to contain auxiliary variables from any of these tiers then the Jacobian matrix would also contain these structural zeros, whose locations would have been determined by the hSAD pass.

Thirdly, hSAD produced a modified DAG containing 29 auxiliary variables, six of which were present in the target function DAG. This set of 29 auxiliary variables is required to populate the full Jacobian, with none of these auxiliary variables being redundant. The forward-mode AD ET (table 3.2) requires 22 auxiliary variables (the six associated with the target function and 16 additional ones). With four input variables and, therefore, four forward-mode passes required, 88 auxiliary computations would be required for full Jacobian evaluation. Similarly, the reverse-mode AD ET (table 3.3) requires 35 auxiliary variables (the six associated with the target function and 29 additional ones). Of the 29 additional additional auxiliary variables, 10 are zero-assignments and could potentially be removed as an optimisation during compilation. With two output variables and, therefore, two reverse-mode passes required, 70 auxiliary computations would be required for the full Jacobian evaluation in the case of a general implementation, or 50 in the case of an optimised implementation. Even the best-performing optimised reverse-mode AD ET results in almost twice as many instructions compared to the hSAD ET. Therefore, if compiled to equivalent callables, the hSAD callable would offer faster evaluation than the reverse-mode AD callable.

Finally, of the 23 additional auxiliary variables, none of them correspond to trivial calculations, such as multiplications by 0.0 or 1.0, or additions with 0.0. This suggests that the ET produced by hSAD is in a highly simplified form and potentially close to optimal, if not optimal.

3.5 Algorithm Performance Optimisations

3.5.1 Function Nodes

The base hSAD algorithm requires that an entire function is expressed as a DAG, either explicitly or by means of an ET being available such that a DAG can be constructed. This DAG will contain nodes corresponding to atomic operations, such as unary trigonometric functions or binary arithmetic operations. However, in certain circumstances, it may be desirable to express function concepts at a higher level of abstraction. This may be, for example, to simplify the process of defining the function's DAG or to allow the reuse of common non-atomic functions such as a matrix inversion. hSAD supports this approach through an entity termed *function nodes*.

An implementation of the hSAD algorithm is built around the assumption that the partial derivatives of every node's operation with respect to its operands are

known. For many simple unary and binary operations, these rules of differentiation will be hardcoded. For example, the binary multiplication ab will have its partial derivatives hardcoded as $\frac{\partial(ab)}{\partial a} = b$ and $\frac{\partial(ab)}{\partial b} = a$. If more complex function abstractions are to be represented in the expression graph, then their partial derivatives will not be hardcoded in the same way and will need to be determined during the hSAD preprocessing.

Function nodes allow an expression graph to be built such that it contains nodes that represent non-atomic operations. These function nodes act somewhat like a black box, with a fixed number of inputs and outputs. Therefore, for a function node to be useful and support derivative generation via hSAD, it needs to be able to differentiate its output with respect to its input. This can be done by expressing the operation performed by a function node as its own DAG and applying the hSAD algorithm. This technique can be very powerful as it allows highly complicated expression graphs to be created with many layers of abstraction to simplify them.

It is useful to explain the concept and implementation of function nodes through example. In the lighthouse example (section 3.1.3) and its corresponding expression graph fig. 3.2, five auxiliary tiers were required following tier partitioning. Consider introducing a function node to represent the four nodes contained within $\mathbb{T}_2$, $\mathbb{T}_3$ and $\mathbb{T}_4$. In this example, all auxiliary variables will be denoted with tildes (e.g. $\tilde{w}_i$) to avoid confusion with the nomenclature used in section 3.3.4. Auxiliary variables in the function node expression graph will also be denoted with an apostrophe (e.g. $\tilde{w}'_j$) to clearly distinguish them from auxiliary variables in the main expression graph. The new function node's operation is denoted by F_1 such that the expression graph can be expressed by the auxiliary substitution

$$\begin{aligned}
\tilde{w}_1 &= \omega t = \tilde{x}_3 \tilde{x}_4 = \mathrm{Mul}\left(\tilde{x}_3, \tilde{x}_4\right) \\
\tilde{w}_2 &= \frac{\nu \tan\left(\omega t\right)}{\gamma - \tan\left(\omega t\right)} = F_1\left(\tilde{x}_1, \tilde{x}_2, \tilde{w}_1\right) \\
\tilde{w}_3 &= \frac{\nu\gamma \tan\left(\omega t\right)}{\gamma - \tan\left(\omega t\right)} = \tilde{x}_2 \tilde{w}_2 = \mathrm{Mul}\left(\tilde{x}_2, \tilde{w}_2\right)
\end{aligned}$$

and the output assignments

$$\begin{aligned}
\tilde{y}_1 &= \frac{\nu \tan\left(\omega t\right)}{\gamma - \tan\left(\omega t\right)} = \tilde{w}_2 \\
\tilde{y}_2 &= \frac{\nu\gamma \tan\left(\omega t\right)}{\gamma - \tan\left(\omega t\right)} = \tilde{w}_3 \,,
\end{aligned}$$

with the corresponding DAG shown in fig. 3.3a. The new operation, F_1, can itself

be expressed by the input assignments

$$\begin{aligned}\tilde{x}'_1 &= \nu \\ \tilde{x}'_2 &= \gamma \\ \tilde{x}'_3 &= \omega t = \tilde{w}_1 \,,\end{aligned}$$

the auxiliary substitutions

$$\begin{aligned}\tilde{w}'_1 &= \tan(\omega t) = \tan(\tilde{x}'_3) = \mathrm{Tan}(\tilde{x}'_3) \\ \tilde{w}'_2 &= \gamma - \tan(\omega t) = \tilde{x}'_2 - \tilde{w}'_1 = \mathrm{Sub}(\tilde{x}'_2, \tilde{w}'_1) \\ \tilde{w}'_3 &= \nu \tan(\omega t) = \tilde{x}'_1 \tilde{w}'_1 = \mathrm{Mul}(\tilde{x}'_1, \tilde{w}'_1) \\ \tilde{w}'_4 &= \frac{\nu \tan(\omega t)}{\gamma - \tan(\omega t)} = \frac{\tilde{w}'_3}{\tilde{w}'_2} = \mathrm{Div}(\tilde{w}'_3, \tilde{w}'_2)\end{aligned}$$

and the output assignment

$$\tilde{y}'_1 = \frac{\nu \tan(\omega t)}{\gamma - \tan(\omega t)} = \tilde{w}'_4 \,,$$

with the corresponding DAG shown in fig. 3.3b.

Note that the function node, $\tilde{w}_2$ in fig. 3.3a, is non-standard in comparison to other expression graph nodes that have been shown so far as it contains three inputs. This is inconsequential, however, as hSAD supports nodes to have an arbitrary number of inputs. This should be clear from the analysis in section 3.4.1.

The first step in hSAD is to tier partition the DAG using a suitable graph algorithm. For the example here, where there is a DAG corresponding to the function node as well as one for the target function, both DAGs need to be tier-partitioned. Both of these DAGs are tier-partitioned to contain three auxiliary node tiers. For the target function, these tiers are

$$\tilde{\mathbf{x}}_{\mathbb{T}_0} = \begin{bmatrix}\tilde{x}_1 & \tilde{x}_2 & \tilde{x}_3 & \tilde{x}_4\end{bmatrix} \tag{3.58}$$

$$\tilde{\mathbf{x}}_{\mathbb{T}_1} = \begin{bmatrix}\tilde{w}_1\end{bmatrix} \tag{3.59}$$

$$\tilde{\mathbf{x}}_{\mathbb{T}_2} = \begin{bmatrix}\tilde{w}_2\end{bmatrix} \tag{3.60}$$

$$\tilde{\mathbf{x}}_{\mathbb{T}_3} = \begin{bmatrix}\tilde{w}_3\end{bmatrix} \,, \tag{3.61}$$

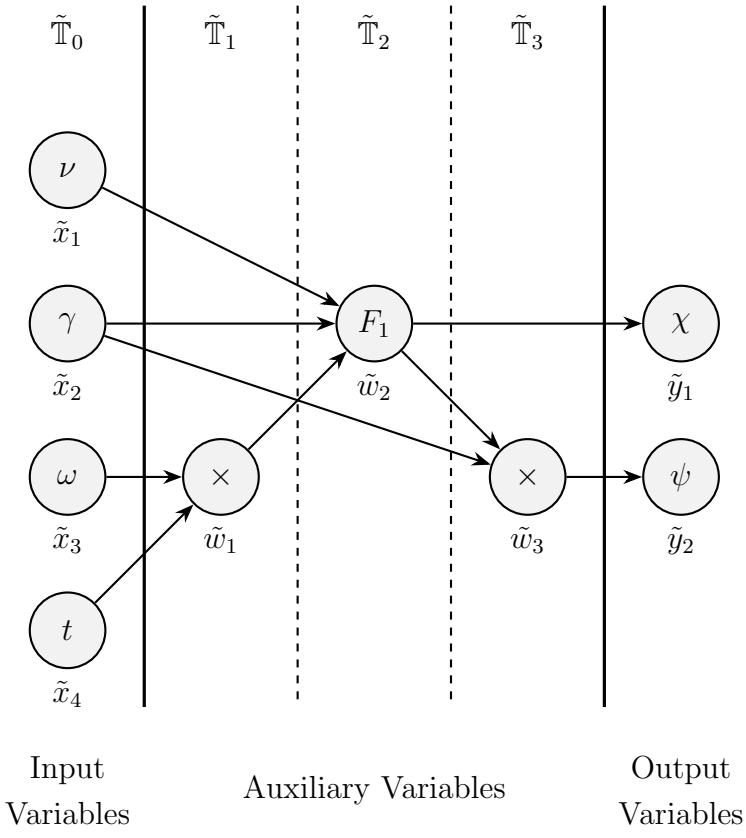
with the tiers for the function node being

$$\tilde{\mathbf{x}}'_{\mathbb{T}_0} = \begin{bmatrix}\tilde{x}'_1 & \tilde{x}'_2 & \tilde{x}'_3\end{bmatrix} \tag{3.62}$$

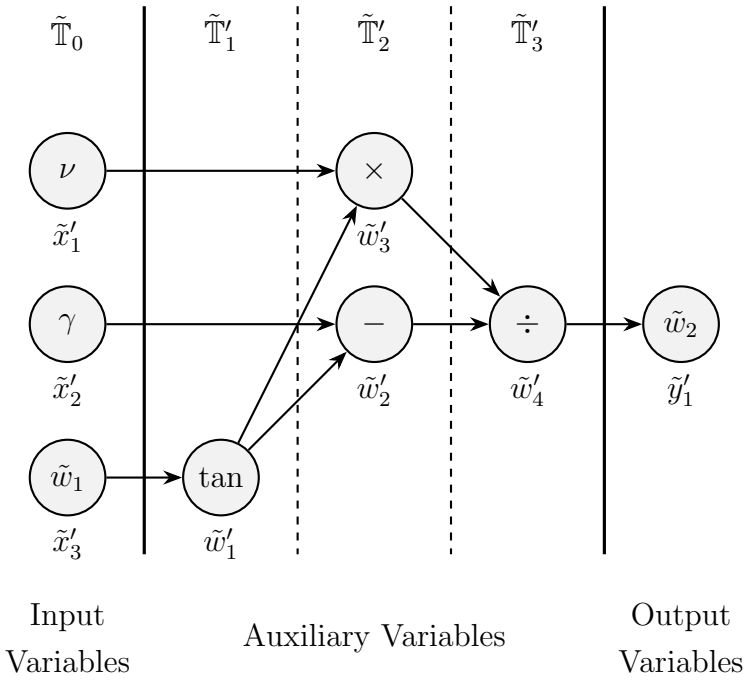
$$\tilde{\mathbf{x}}'_{\mathbb{T}_1} = \begin{bmatrix}\tilde{w}'_1\end{bmatrix} \tag{3.63}$$

$$\tilde{\mathbf{x}}'_{\mathbb{T}_2} = \begin{bmatrix}\tilde{w}'_2 & \tilde{w}'_3\end{bmatrix} \tag{3.64}$$

$$\tilde{\mathbf{x}}'_{\mathbb{T}_3} = \begin{bmatrix}\tilde{w}'_4\end{bmatrix} \,. \tag{3.65}$$



(a) DAG for the target function.



(b) DAG for the function node F_1.

Figure 3.3: DAG for the lighthouse example using hSAD with function nodes.

Following tier partitioning, the delta matrices are produced. When the target function contains function nodes, the function node's partial derivatives are required before hSAD can be applied to the target function. Therefore, the function node's delta matrices are computed first, followed by applying the hSAD equation to the function node before hSAD is applied to the target function. If a target function were to contain multiple function nodes, or if a function node was to itself contain a function node, then hSAD would be applied to the most deeply-nested function node first, with a DAG only being differentiated once all of its function nodes have themselves been addressed.

For the function node's DAG, the zeroth delta matrix, $\tilde{\mathbf{\Delta}}'_{\mathbb{T}_0}$, is the 3×3 identity matrix $\mathbf{I}_{3\times3}$. The first function node delta matrix, $\tilde{\mathbf{\Delta}}'_{\mathbb{T}_1}$, is calculated as

$$\begin{aligned}
\tilde{\mathbf{\Delta}}'_{\mathbb{T}_1} &= \frac{\partial \tilde{\mathbf{x}}'_{\mathbb{T}_1}}{\partial \tilde{\mathbf{x}}'_{\mathbb{T}_0}} \tilde{\mathbf{\Delta}}'_{\mathbb{T}_0} \\
&= \begin{bmatrix} 0 & 0 & \sec^2(\tilde{w}_1) \end{bmatrix} \mathbf{I}_{3\times3} \\
&= \begin{bmatrix} 0 & 0 & \sec^2(\tilde{w}_1) \end{bmatrix} \\
&= \begin{bmatrix} 0 & 0 & \tilde{w}_6 \end{bmatrix},
\end{aligned} \tag{3.66}$$

where the additional auxiliary substitutions

$$\begin{aligned}
\tilde{w}'_5 &= \sec(\omega t) = \sec \tilde{x}'_3 = \mathrm{Sec}(\tilde{x}'_3) \\
\tilde{w}'_6 &= \sec^2(\omega t) = (\tilde{w}'_5)^2 = \mathrm{Sqr}(\tilde{w}'_5)
\end{aligned}$$

are created. The second function node delta matrix, $\tilde{\mathbf{\Delta}}'_{\mathbb{T}_2}$, is calculated as

$$\begin{aligned}
\tilde{\mathbf{\Delta}}'_{\mathbb{T}_2} &= \frac{\partial \tilde{\mathbf{x}}_{\mathbb{T}_2}}{\partial \tilde{\mathbf{x}}_{\mathbb{T}_0}} \tilde{\mathbf{\Delta}}'_{\mathbb{T}_0} + \frac{\partial \tilde{\mathbf{x}}_{\mathbb{T}_2}}{\partial \tilde{\mathbf{x}}_{\mathbb{T}_1}} \tilde{\mathbf{\Delta}}'_{\mathbb{T}_1} \\
&= \begin{bmatrix} 0 & 1 & 0 \\ \tilde{w}'_1 & 0 & 0 \end{bmatrix} \mathbf{I}_{3\times3} + \begin{bmatrix} -1 \\ \tilde{x}'_1 \end{bmatrix} \begin{bmatrix} 0 & 0 & \tilde{w}_6 \end{bmatrix} \\
&= \begin{bmatrix} 0 & 1 & \tilde{w}'_7 \\ \tilde{w}'_1 & 0 & \tilde{w}'_8 \end{bmatrix},
\end{aligned} \tag{3.67}$$

where the additional auxiliary substitutions

$$\begin{aligned}
\tilde{w}'_7 &= -\sec^2(\omega t) = -\tilde{w}'_6 = \mathrm{Neg}(\tilde{w}'_6) \\
\tilde{w}'_8 &= \nu \sec^2(\omega t) = \tilde{x}'_1 \tilde{w}'_6 = \mathrm{Mul}(\tilde{x}'_1, \tilde{w}'_6)
\end{aligned}$$

are created. The third and final function node delta matrix, $\tilde{\mathbf{\Delta}}'_{\mathbb{T}_3}$, is calculated as

$$\begin{aligned}
\tilde{\mathbf{\Delta}}'_{\mathbb{T}_3} &= \cancelto{\mathbf{0}_{1\times 3}}{\frac{\partial \tilde{\mathbf{x}}'_{\mathbb{T}_3}}{\partial \tilde{\mathbf{x}}'_{\mathbb{T}_0}}} \tilde{\mathbf{\Delta}}'_{\mathbb{T}_0} + \cancelto{\mathbf{0}_{1\times 1}}{\frac{\partial \tilde{\mathbf{x}}'_{\mathbb{T}_3}}{\partial \tilde{\mathbf{x}}'_{\mathbb{T}_1}}} \tilde{\mathbf{\Delta}}'_{\mathbb{T}_1} + \frac{\partial \tilde{\mathbf{x}}'_{\mathbb{T}_3}}{\partial \tilde{\mathbf{x}}'_{\mathbb{T}_2}} \tilde{\mathbf{\Delta}}'_{\mathbb{T}_2} \\
&= \begin{bmatrix} -\frac{\tilde{w}'_3}{\left(\tilde{w}'_2\right)^2} & \frac{1}{\left(\tilde{w}'_2\right)^2} \end{bmatrix} \begin{bmatrix} 0 & 1 & \tilde{w}'_7 \\ \tilde{w}'_1 & 0 & \tilde{w}'_8 \end{bmatrix} \\
&= \begin{bmatrix} \tilde{w}'_{11} & \tilde{w}'_9 \end{bmatrix} \begin{bmatrix} 0 & 1 & \tilde{w}'_7 \\ \tilde{w}'_1 & 0 & \tilde{w}'_8 \end{bmatrix} \\
&= \begin{bmatrix} \tilde{w}'_1\tilde{w}'_9 & \tilde{w}'_{11} & \tilde{w}'_6\tilde{w}'_{11} + \tilde{w}'_7\tilde{w}'_9 \end{bmatrix} \\
&= \begin{bmatrix} \tilde{w}'_{12} & \tilde{w}'_{11} & \tilde{w}'_{15} \end{bmatrix},
\end{aligned} \tag{3.68}$$

where the additional auxiliary substitutions

$$\begin{aligned}
\tilde{w}'_9 &= \frac{1}{\gamma - \tan(\omega t)} = \frac{1}{\tilde{w}'_2} = \mathrm{Recip}\left(\tilde{w}'_2\right) \\
\tilde{w}'_{10} &= \frac{\nu \tan(\omega t)}{\gamma - \tan(\omega t)} = \tilde{w}'_4\tilde{w}'_9 = \mathrm{Mul}\left(\tilde{w}'_4, \tilde{w}'_9\right) \\
\tilde{w}'_{11} &= -\frac{\nu \tan(\omega t)}{\left(\gamma - \tan(\omega t)\right)^2} = -\tilde{w}'_{10} = \mathrm{Neg}\left(\tilde{w}'_{10}\right) \\
\tilde{w}'_{12} &= \frac{\tan(\omega t)}{\gamma - \tan(\omega t)} = \tilde{w}'_1\tilde{w}'_9 = \mathrm{Mul}\left(\tilde{w}'_1, \tilde{w}'_9\right) \\
\tilde{w}'_{13} &= \frac{\nu \tan(\omega t)\sec^2(\omega t)}{\left(\gamma - \tan(\omega t)\right)^2} = \tilde{w}'_{17}\tilde{w}'_{11} = \mathrm{Mul}\left(\tilde{w}'_7, \tilde{w}'_{11}\right) \\
\tilde{w}'_{14} &= \frac{\nu \sec^2(\omega t)}{\gamma - \tan(\omega t)} = \tilde{w}'_8\tilde{w}'_9 = \mathrm{Mul}\left(\tilde{w}'_8, \tilde{w}'_9\right) \\
\tilde{w}'_{15} &= \frac{\nu \tan(\omega t)\sec^2(\omega t)}{\left(\gamma - \tan(\omega t)\right)^2} + \frac{\nu \sec^2(\omega t)}{\gamma - \tan(\omega t)} = \tilde{w}'_{13} + \tilde{w}'_{14} = \mathrm{Add}\left(\tilde{w}'_{13}, \tilde{w}'_{14}\right)
\end{aligned}$$

are created. The Jacobian matrix corresponding to the function node, $\tilde{\mathbf{G}}'$ can then be expressed using the hSAD equation (eq. (3.47)) as

$$\begin{aligned}
\tilde{\mathbf{G}}' = \frac{\partial \tilde{\mathbf{y}}'}{\partial \tilde{\mathbf{x}}'} &= \cancelto{\mathbf{0}_{1\times 3}}{\frac{\partial \tilde{\mathbf{y}}'}{\partial \tilde{\mathbf{x}}'_{\mathbb{T}_0}}} \tilde{\mathbf{\Delta}}'_{\mathbb{T}_0} + \cancelto{\mathbf{0}_{1\times 1}}{\frac{\partial \tilde{\mathbf{y}}'}{\partial \tilde{\mathbf{x}}'_{\mathbb{T}_1}}} \tilde{\mathbf{\Delta}}'_{\mathbb{T}_1} + \cancelto{\mathbf{0}_{1\times 2}}{\frac{\partial \tilde{\mathbf{y}}'}{\partial \tilde{\mathbf{x}}'_{\mathbb{T}_2}}} \tilde{\mathbf{\Delta}}'_{\mathbb{T}_2} + \frac{\partial \tilde{\mathbf{y}}'}{\partial \tilde{\mathbf{x}}'_{\mathbb{T}_3}} \tilde{\mathbf{\Delta}}'_{\mathbb{T}_3} \\
&= \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} \tilde{w}'_{12} & \tilde{w}'_{11} & \tilde{w}'_{15} \end{bmatrix} \\
&= \begin{bmatrix} \tilde{w}'_{12} & \tilde{w}'_{11} & \tilde{w}'_{15} \end{bmatrix}.
\end{aligned} \tag{3.69}$$

$\tilde{\mathbf{G}}'$ can now be used to provide the partial derivatives $\frac{\partial \tilde{w}_2}{\partial \tilde{x}_1}$, $\frac{\partial \tilde{w}_2}{\partial \tilde{x}_2}$ and $\frac{\partial \tilde{w}_2}{\partial \tilde{w}_1}$ when hSAD is conducted on the target function's DAG.

For the target function's DAG, the zeroth delta matrix, $\tilde{\mathbf{\Delta}}_{\mathbb{T}_0}$, is the 4×4 identity

matrix $\mathbf{I}_{4\times4}$. The first target function delta matrix, $\tilde{\mathbf{\Delta}}_{\mathbb{T}_1}$, is calculated as

$$\begin{aligned}\tilde{\mathbf{\Delta}}_{\mathbb{T}_1} &= \frac{\partial\tilde{\mathbf{x}}_{\mathbb{T}_1}}{\partial\tilde{\mathbf{x}}_{\mathbb{T}_0}}\tilde{\mathbf{\Delta}}_{\mathbb{T}_0} \\ &= \begin{bmatrix}0 & 0 & \tilde{x}_4 & \tilde{x}_3\end{bmatrix}\mathbf{I}_{4\times4} \\ &= \begin{bmatrix}0 & 0 & \tilde{x}_4 & \tilde{x}_3\end{bmatrix}.\end{aligned} \tag{3.70}$$

The second target function delta matrix, $\tilde{\mathbf{\Delta}}_{\mathbb{T}_2}$, is calculated as

$$\begin{aligned}\tilde{\mathbf{\Delta}}_{\mathbb{T}_2} &= \frac{\partial\tilde{\mathbf{x}}_{\mathbb{T}_2}}{\partial\tilde{\mathbf{x}}_{\mathbb{T}_0}}\tilde{\mathbf{\Delta}}_{\mathbb{T}_0} + \frac{\partial\tilde{\mathbf{x}}_{\mathbb{T}_2}}{\partial\tilde{\mathbf{x}}_{\mathbb{T}_1}}\tilde{\mathbf{\Delta}}_{\mathbb{T}_1} \\ &= \begin{bmatrix}\tilde{w}'_{12} & \tilde{w}'_{11} & 0 & 0\end{bmatrix}\mathbf{I}_{4\times4} + \begin{bmatrix}\tilde{w}'_{15}\end{bmatrix}\begin{bmatrix}0 & 0 & \tilde{x}_4 & \tilde{x}_3\end{bmatrix} \\ &= \begin{bmatrix}\tilde{w}'_{12} & \tilde{w}'_{11} & 0 & 0\end{bmatrix} + \begin{bmatrix}0 & 0 & \tilde{w}_4 & \tilde{w}_5\end{bmatrix} \\ &= \begin{bmatrix}\tilde{w}'_{12} & \tilde{w}'_{11} & \tilde{w}_4 & \tilde{w}_5\end{bmatrix},\end{aligned} \tag{3.71}$$

where the additional auxiliary substitutions

$$\begin{aligned}\tilde{w}_4 &= \frac{\nu t\tan(\omega t)\sec^2(\omega t)}{(\gamma-\tan(\omega t))^2} + \frac{\nu t\sec^2(\omega t)}{\gamma-\tan(\omega t)} = \tilde{x}_4\tilde{w}'_{15} = \mathrm{Mul}(\tilde{x}_4,\tilde{w}'_{15}) \\ \tilde{w}_5 &= \frac{\nu\omega\tan(\omega t)\sec^2(\omega t)}{(\gamma-\tan(\omega t))^2} + \frac{\nu\omega\sec^2(\omega t)}{\gamma-\tan(\omega t)} = \tilde{x}_3\tilde{w}'_{15} = \mathrm{Mul}(\tilde{x}_3,\tilde{w}'_{15})\end{aligned}$$

are created. Note how the nonzero terms in the partial derivative matrices, $\frac{\partial\tilde{\mathbf{x}}_{\mathbb{T}_2}}{\partial\tilde{\mathbf{x}}_{\mathbb{T}_0}}$ and $\frac{\partial\tilde{\mathbf{x}}_{\mathbb{T}_2}}{\partial\tilde{\mathbf{x}}_{\mathbb{T}_1}}$, in eq. (3.71) are populated using entries from $\tilde{\mathbf{G}}'$ (eq. (3.69)). The third and final target function's delta matrix, $\tilde{\mathbf{\Delta}}_{\mathbb{T}_3}$, is calculated as

$$\begin{aligned}\tilde{\mathbf{\Delta}}_{\mathbb{T}_3} &= \frac{\partial\tilde{\mathbf{x}}_{\mathbb{T}_3}}{\partial\tilde{\mathbf{x}}_{\mathbb{T}_0}}\tilde{\mathbf{\Delta}}_{\mathbb{T}_0} + \cancelto{\mathbf{0}_{1\times1}}{\frac{\partial\tilde{\mathbf{x}}_{\mathbb{T}_3}}{\partial\tilde{\mathbf{x}}_{\mathbb{T}_1}}}\tilde{\mathbf{\Delta}}_{\mathbb{T}_1} + \frac{\partial\tilde{\mathbf{x}}_{\mathbb{T}_3}}{\partial\tilde{\mathbf{x}}_{\mathbb{T}_2}}\tilde{\mathbf{\Delta}}_{\mathbb{T}_2} \\ &= \begin{bmatrix}0 & \tilde{w}_2 & 0 & 0\end{bmatrix}\mathbf{I}_{4\times4} + \begin{bmatrix}\tilde{x}_2\end{bmatrix}\begin{bmatrix}\tilde{w}'_{12} & \tilde{w}'_{11} & \tilde{w}_4 & \tilde{w}_5\end{bmatrix} \\ &= \begin{bmatrix}0 & \tilde{w}_2 & 0 & 0\end{bmatrix} + \begin{bmatrix}\tilde{w}_6 & \tilde{w}_7 & \tilde{w}_8 & \tilde{w}_9\end{bmatrix} \\ &= \begin{bmatrix}\tilde{w}_6 & \tilde{w}_{10} & \tilde{w}_8 & \tilde{w}_9\end{bmatrix},\end{aligned} \tag{3.72}$$

where the auxiliary substitutions

$$\begin{aligned}\tilde{w}_6 &= \frac{\gamma\tan(\omega t)}{\gamma-\tan(\omega t)} = \tilde{x}_2\tilde{w}'_{12} = \mathrm{Mul}(\tilde{x}_2,\tilde{w}'_{12}) \\ \tilde{w}_7 &= -\frac{\nu\gamma\tan(\omega t)}{(\gamma-\tan(\omega t))^2} = \tilde{x}_2\tilde{w}'_{11} = \mathrm{Mul}(\tilde{x}_2,\tilde{w}'_{11}) \\ \tilde{w}_8 &= \frac{\nu\gamma t\tan(\omega t)\sec^2(\omega t)}{(\gamma-\tan(\omega t))^2} + \frac{\nu\gamma t\sec^2(\omega t)}{\gamma-\tan(\omega t)} = \tilde{x}_2\tilde{w}_4 = \mathrm{Mul}(\tilde{x}_2,\tilde{w}_4) \\ \tilde{w}_9 &= \frac{\nu\gamma\omega\tan(\omega t)\sec^2(\omega t)}{(\gamma-\tan(\omega t))^2} + \frac{\nu\gamma\omega\sec^2(\omega t)}{\gamma-\tan(\omega t)} = \tilde{x}_2\tilde{w}_5 = \mathrm{Mul}(\tilde{x}_2,\tilde{w}_5) \\ \tilde{w}_{10} &= \frac{\nu\tan(\omega t)}{\gamma-\tan(\omega t)} - \frac{\nu\gamma\tan(\omega t)}{(\gamma-\tan(\omega t))^2} = \tilde{w}_2 + \tilde{w}_7 = \mathrm{Add}(\tilde{w}_2,\tilde{w}_7)\end{aligned}$$

are created. The Jacobian matrix of the target function is finally expressed in its full form using the hSAD equation (eq. (3.47)) as

$$\begin{aligned}\tilde{\mathbf{G}} = \frac{\partial \tilde{\mathbf{y}}}{\partial \tilde{\mathbf{x}}} &= \cancelto{\mathbf{0}_{2\times 4}}{\frac{\partial \tilde{\mathbf{y}}}{\partial \tilde{\mathbf{x}}_{\mathbb{T}_0}}}\tilde{\mathbf{\Delta}}'_{\mathbb{T}_0} + \cancelto{\mathbf{0}_{2\times 1}}{\frac{\partial \tilde{\mathbf{y}}}{\partial \tilde{\mathbf{x}}_{\mathbb{T}_1}}}\tilde{\mathbf{\Delta}}'_{\mathbb{T}_1} + \frac{\partial \tilde{\mathbf{y}}}{\partial \tilde{\mathbf{x}}_{\mathbb{T}_2}}\tilde{\mathbf{\Delta}}'_{\mathbb{T}_2} + \frac{\partial \tilde{\mathbf{y}}}{\partial \tilde{\mathbf{x}}_{\mathbb{T}_3}}\tilde{\mathbf{\Delta}}'_{\mathbb{T}_3} \\ &= \begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}\tilde{w}'_{12} & \tilde{w}'_{11} & \tilde{w}_4 & \tilde{w}_5\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}\tilde{w}_6 & \tilde{w}_{10} & \tilde{w}_8 & \tilde{w}_9\end{bmatrix} \\ &= \begin{bmatrix}\tilde{w}'_{12} & \tilde{w}'_{11} & \tilde{w}_4 & \tilde{w}_5\\ \tilde{w}_6 & \tilde{w}_{10} & \tilde{w}_8 & \tilde{w}_9\end{bmatrix}.\end{aligned} \tag{3.73}$$

Comparison of the eight entries of $\tilde{\mathbf{G}}$ (eq. (3.73)), generated using hSAD with a function node, to the eight entries of $\mathbf{G}$ found either through basic hSAD (eq. (3.55)) or via MD (eqs. (3.4) to (3.11)) shows exact agreement in their analytic representations. This illustrates that the use of function nodes alongside hSAD still results in the production of exact analytical partial derivatives.

The original DAG associated with the target function contained three auxiliary variables ($\tilde{w}_1$, $\tilde{w}_2$ and $\tilde{w}_3$), while the original DAG associated with the function node contained four ($\tilde{w}'_1$, $\tilde{w}'_2$, $\tilde{w}'_3$ and $\tilde{w}'_4$). Note that, in total, this is one more than was required without the use of a function node. However, as the F_1 outputs $\tilde{w}'_4$ which is directly assigned to $\tilde{w}_2$, it can be argued that there are only six true operation nodes associated with the original target function. Indeed, if the target function with its function node were to be compiled, the assignment of $\tilde{w}'_4$ to $\tilde{w}_2$ could easily be removed by inlining all of the operations associated with the function node.

The hSAD pass on the F_1 function node generated 11 new auxiliary variables ($\tilde{w}'_5$ to $\tilde{w}'_{15}$), while the hSAD pass on the target function containing F_1 generated just seven new auxiliary variables ($\tilde{w}_4$ to $\tilde{w}_{10}$). Combined, these 18 new auxiliary variables are fewer than the 23 auxiliary variables generated when hSAD was applied to the same problem without the use of function nodes (section 3.3.4). This result suggests that the use of function nodes has the potential to produce more compact modified expression graphs through better reuse of common subexpressions. More compact expression graphs result in more efficient ETs and, therefore, the executables produced if function nodes have been used may be less computationally expensive than their counterpart without function nodes, as demonstrated here. An ET generated using the modified expression graphs produced by the hSAD passes to populate $\tilde{\mathbf{G}}$ and $\tilde{\mathbf{G}}'$ is shown in table 3.5.

Independent Variables
$\tilde{x}_1 = \nu$
$\tilde{x}_2 = \gamma$
$\tilde{x}_3 = \omega$
$\tilde{x}_4 = t$
Auxiliary Variables
$\tilde{w}_1 = x_3 x_4 = \text{Mul}(x_3, x_4)$
$\tilde{w}_2, \tilde{w}'_{12}, \tilde{w}'_{11}, \tilde{w}'_{15} = F_1(\tilde{x}_1, \tilde{x}_2, \tilde{w}_1)$
$\tilde{w}_3 = \tilde{x}_2 \tilde{w}_2 = \text{Mul}(\tilde{x}_2, \tilde{w}_2)$
$\tilde{w}_4 = \tilde{x}_4 \tilde{w}'_{15} = \text{Mul}(\tilde{x}_4, \tilde{w}'_{15})$
$\tilde{w}_5 = \tilde{x}_3 \tilde{w}'_{15} = \text{Mul}(\tilde{x}_3, \tilde{w}'_{15})$
$\tilde{w}_6 = \tilde{x}_2 \tilde{w}'_{12} = \text{Mul}(\tilde{x}_2, \tilde{w}'_{12})$
$\tilde{w}_7 = \tilde{x}_2 \tilde{w}'_{11} = \text{Mul}(\tilde{x}_2, \tilde{w}'_{11})$
$\tilde{w}_8 = \tilde{x}_2 \tilde{w}_4 = \text{Mul}(\tilde{x}_2, \tilde{w}_4)$
$\tilde{w}_9 = \tilde{x}_2 \tilde{w}_5 = \text{Mul}(\tilde{x}_2, \tilde{w}_5)$
$\tilde{w}_{10} = \tilde{w}_2 + \tilde{w}_7 = \text{Add}(\tilde{w}_2, \tilde{w}_7)$
Dependent Variables
$\tilde{G}_{11} = \tilde{w}'_{12}$
$\tilde{G}_{12} = \tilde{w}'_{11}$
$\tilde{G}_{13} = \tilde{w}_4$
$\tilde{G}_{14} = \tilde{w}_5$
$\tilde{G}_{21} = \tilde{w}_6$
$\tilde{G}_{22} = \tilde{w}_{10}$
$\tilde{G}_{23} = \tilde{w}_8$
$\tilde{G}_{24} = \tilde{w}_9$
Target Function

Independent Variables
$\tilde{x}'_1 = \tilde{x}_1$
$\tilde{x}'_2 = \tilde{x}_2$
$\tilde{x}'_3 = \tilde{w}_1$
Auxiliary Variables
$\tilde{w}'_1 = \tan(\tilde{x}'_3) = \text{Tan}(\tilde{x}'_3)$
$\tilde{w}'_2 = \tilde{x}'_2 - \tilde{w}'_1 = \text{Sub}(\tilde{x}'_2, \tilde{w}'_1)$
$\tilde{w}'_3 = \tilde{x}'_1 \tilde{w}'_1 = \text{Mul}(\tilde{x}'_1 \tilde{w}'_1)$
$\tilde{w}'_4 = \frac{\tilde{w}'_3}{\tilde{w}'_2} = \text{Div}(\tilde{w}'_3, \tilde{w}'_2)$
$\tilde{w}'_5 = \sec(\tilde{x}'_3) = \text{Sec}(\tilde{x}'_3)$
$\tilde{w}'_6 = (\tilde{w}'_5)^2 = \text{Sqr}(\tilde{w}'_5)$
$\tilde{w}'_7 = -\tilde{w}'_6 = \text{Neg}(\tilde{w}'_6)$
$\tilde{w}'_8 = \tilde{x}'_1 \tilde{w}'_6 = \text{Mul}(\tilde{x}'_1, \tilde{w}'_6)$
$\tilde{w}'_9 = \frac{1}{\tilde{w}'_2} = \text{Recip}(\tilde{w}'_2)$
$\tilde{w}'_{10} = \tilde{w}'_4 \tilde{w}'_9 = \text{Mul}(\tilde{w}'_4, \tilde{w}'_9)$
$\tilde{w}'_{11} = -\tilde{w}'_{10} = \text{Neg}(\tilde{w}'_{10})$
$\tilde{w}'_{12} = \tilde{w}'_1 \tilde{w}'_9 = \text{Mul}(\tilde{w}'_1, \tilde{w}'_9)$
$\tilde{w}'_{13} = \tilde{w}'_7 \tilde{w}'_{11} = \text{Mul}(\tilde{w}'_7, \tilde{w}'_{11})$
$\tilde{w}'_{14} = \tilde{w}'_8 \tilde{w}'_9 = \text{Mul}(\tilde{w}'_8, \tilde{w}'_9)$
$\tilde{w}'_{15} = \tilde{w}'_{13} + \tilde{w}'_{14} = \text{Add}(\tilde{w}'_{13}, \tilde{w}'_{14})$
Dependent Variables
$\tilde{y}'_1 = \tilde{w}'_1$
$\tilde{G}'_{11} = \tilde{w}'_{12}$
$\tilde{G}'_{12} = \tilde{w}'_{11}$
$\tilde{G}'_{13} = \tilde{w}'_{15}$
Function Node

Table 3.5: Jacobian ET for the lighthouse example using hSAD with function nodes.

3.5.2 Tier Checkpointing

Tier checkpointing is a strategy available with hSAD to reduce the algorithms memory cost. It is similar in approach to *checkpointing* in reverse-mode AD whereby the target function in partitioned into sections that can be processed independently and then recombined. The boundaries between these partitioned sections are known as checkpoints in AD terminology. Figure 3.4 shows what the DAG for the lighthouse example could look like if a single checkpoint tier were to be introduced at the approximate halfway point in the DAG.

Tier checkpointing divides the target function's DAG into multiple portions, with the output nodes from one portion functioning as the input nodes to the following portion. These nodes that function as both output and input nodes in adjacent portions are termed *checkpoint variables*. The checkpointed target function can then be considered as a composite function of its checkpointed portions. From fig. 3.4, the checkpoint variables $\{\hat{x}_2, \hat{w}_3, \hat{w}_4\}$ contained within the checkpoint tier, $\hat{\mathbb{T}}_0$, illustrate the concept of tier checkpointing. Note that in this version of the lighthouse example, all variables are denoted with hats (e.g. $\hat{w}_i$) to avoid confusion with the nomenclature used in sections 3.3.4 and 3.5.1. Additionally, auxiliary variables in the second checkpointed portion will also be denoted with an apostrophe (e.g. $\hat{w}'_j$) to clearly distinguish them from auxiliary variables in the first checkpointed portion. Inspecting fig. 3.4, it can be seen that one of the target function's input variables, $\hat{x}_2$, is carried directly over to the checkpoint tier, $\hat{\mathbb{T}}'_0$, while the other checkpoint tier variables, $\hat{w}_3$ and $\hat{w}_4$, come directly from the first portion's final tier, $\hat{\mathbb{T}}_3$. This illustrates that any variables can be mapped to checkpoint variables provided that they are of lower precedence in the DAG

If tier checkpoints are used, then slight modification to the hSAD algorithm is required. This is because when the hSAD pass is conducted on a checkpointed portion, the resulting Jacobian matrix details the partial derivatives of that portion's output variables with respect to its input variables. For the whole target function to be differentiated, a hSAD pass needs to be conducted on all checkpointed portions and the Jacobian matrices for all portions combined. Using the lighthouse example from fig. 3.4, which contains a single checkpoint tier and two checkpointed portions, if a hSAD pass were to be conducted then two Jacobian matrices, $\hat{\mathbf{G}}_A$ and $\hat{\mathbf{G}}_B$, are

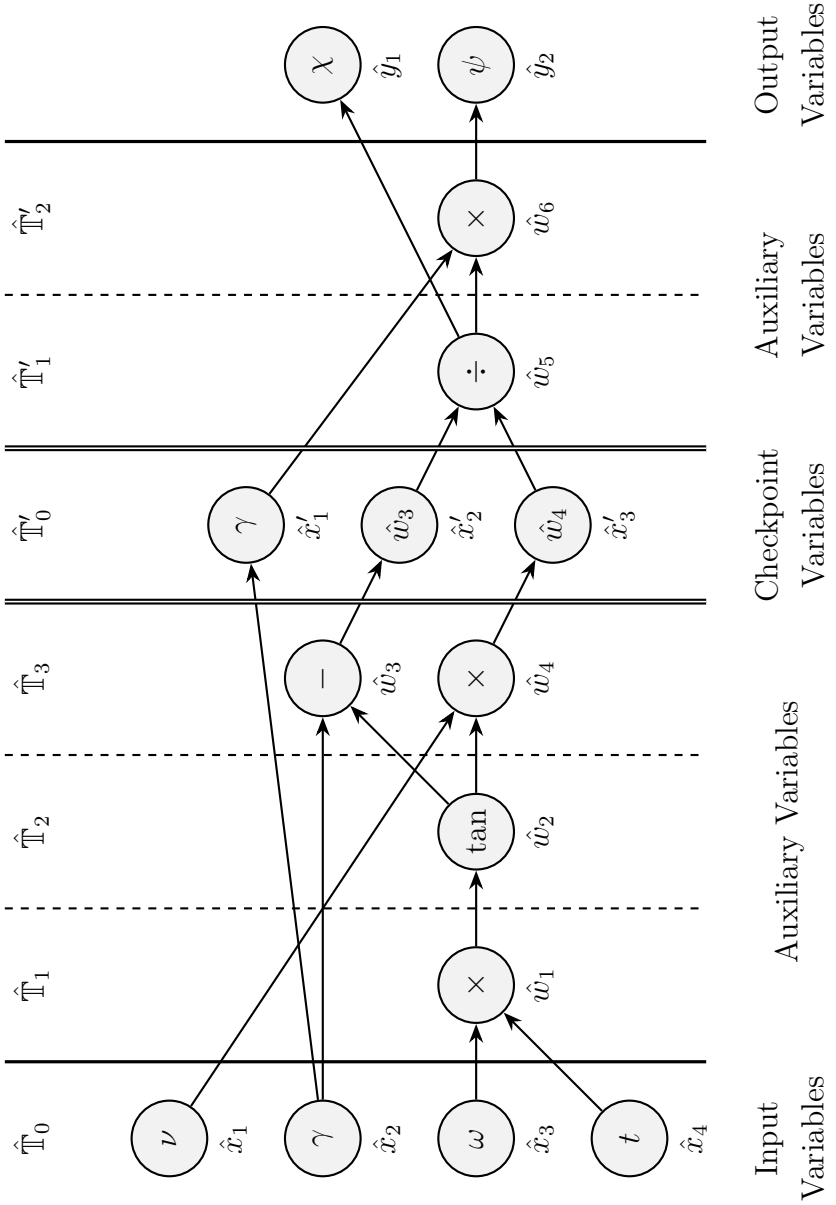


Figure 3.4: DAG for the lighthouse example using hSAD with tier checkpointing.

produced where

$$\hat{\mathbf{G}}_A = \frac{\partial \hat{\mathbf{x}}'}{\partial \hat{\mathbf{x}}} = \begin{bmatrix} \frac{\partial \hat{x}'_1}{\partial \hat{x}_1} & \frac{\partial \hat{x}'_1}{\partial \hat{x}_2} & \frac{\partial \hat{x}'_1}{\partial \hat{x}_3} & \frac{\partial \hat{x}'_1}{\partial \hat{x}_4} \\ \frac{\partial \hat{x}'_2}{\partial \hat{x}_1} & \frac{\partial \hat{x}'_2}{\partial \hat{x}_2} & \frac{\partial \hat{x}'_2}{\partial \hat{x}_3} & \frac{\partial \hat{x}'_2}{\partial \hat{x}_4} \\ \frac{\partial \hat{x}'_3}{\partial \hat{x}_1} & \frac{\partial \hat{x}'_3}{\partial \hat{x}_2} & \frac{\partial \hat{x}'_3}{\partial \hat{x}_3} & \frac{\partial \hat{x}'_3}{\partial \hat{x}_4} \end{bmatrix} \tag{3.74}$$

$$\hat{\mathbf{G}}_B = \frac{\partial \hat{\mathbf{y}}}{\partial \hat{\mathbf{x}}'} = \begin{bmatrix} \frac{\partial \hat{y}_1}{\partial \hat{x}'_1} & \frac{\partial \hat{y}_1}{\partial \hat{x}'_2} & \frac{\partial \hat{y}_1}{\partial \hat{x}'_3} \\ \frac{\partial \hat{y}_2}{\partial \hat{x}'_1} & \frac{\partial \hat{y}_2}{\partial \hat{x}'_2} & \frac{\partial \hat{y}_2}{\partial \hat{x}'_3} \end{bmatrix} . \tag{3.75}$$

The additional required step is therefore the recombination of $\hat{\mathbf{G}}_A$ and $\hat{\mathbf{G}}_B$ using the chain rule (eq. (3.19)) such that

$$\hat{\mathbf{G}} = \frac{\partial \hat{\mathbf{y}}}{\partial \hat{\mathbf{x}}} = \frac{\partial \hat{\mathbf{y}}}{\partial \hat{\mathbf{x}}'} \frac{\partial \hat{\mathbf{x}}'}{\partial \hat{\mathbf{x}}} = \hat{\mathbf{G}}_A \hat{\mathbf{G}}_B \, . \tag{3.76}$$

The fact that the DAG is partitioned by its checkpoint tiers means that each portion of the checkpointed expression graph is independent from all others. This means that each portion can be processed using hSAD independently before their recombination using eq. (3.76), which results in both potential computational and memory benefits. Firstly, the fact that the Jacobian matrix of each independent checkpointed portion can be produced using hSAD completely independently from all others means that this problem is *embarrassingly parallel*. It is therefore greatly suited for parallel implementation. If each checkpointed portion's Jacobian is produced using a separate process then there is a potential computational speed up for the hSAD passes proportional to the number of processes used. If p checkpoint portions are used (i.e. if there are $p-1$ checkpoints), then the Jacobian recombination will require $p-1$ linear matrix multiplications. However, if these recombinations are conducted in parallel as pairs of matrix multiplications, then the wall time runtime could be reduced to the time associated with only $\lceil \log_2 p \rceil$ matrix multiplications.

Secondly, if excessive memory consumption is a point of consideration then tier checkpointing can be used to improve the memory cost of hSAD. This is because, at most, only memory corresponding to the hSAD memory cost of a single checkpointed portion in addition to the memory required to store the accumulated Jacobian needs to be allocated. Using the lighthouse example, if the first checkpointed portion is processed in its entirety first, then the total memory cost is that required to produce $\hat{\mathbf{G}}_A$ using hSAD. That is the memory required to store $\hat{\mathbf{G}}_A$ plus the memory associated with the first checkpointed portion's delta and partial derivative matrices. Once $\hat{\mathbf{G}}_A$ has been computed, the delta and partial derivative matrices associated with the first checkpointed portion are no longer needed and this memory can be freed. $\hat{\mathbf{G}}_B$ can then be computed in the same manner, with the total memory consumption now being that associated with $\hat{\mathbf{G}}_A$, $\hat{\mathbf{G}}_B$ and the delta and partial

derivative matrices of the second checkpointed tier. Once $\hat{\mathbf{G}}_B$ has been computed in addition to $\hat{\mathbf{G}}_A$, the delta and partial derivative matrices associated with the second checkpointed portion are no longer needed and this memory can again be freed. The combined Jacobian can then be accumulated by multiplying $\hat{\mathbf{G}}_A$ and $\hat{\mathbf{G}}_B$. In this case of the lighthouse example, this would result in the complete combined Jacobian. However, if there were further checkpointed portions, these could be computed and the combined Jacobian accumulated without increasing the maximum memory consumption further. This is because only the resulting product of $\hat{\mathbf{G}}_A$ and $\hat{\mathbf{G}}_B$ would need to be stored for later use. Therefore, memory for only one matrix would be required and the memory associated with both $\hat{\mathbf{G}}_A$ and $\hat{\mathbf{G}}_B$ could be freed.

As with checkpoints employed during reverse-mode AD, tier checkpointing in hSAD also improves memory consumption of the algorithm at the expense of introducing a moderate number of additional floating point operations [20, 136]. This is because tier checkpointing essentially introduces additional nodes in to the expression graph. Although these additional nodes do not have an operation associated with them, and therefore differentiate to 1 with respect to their parent variable, they add further directed edges to the DAG. These larger DAGs will result in less computationally efficient numerical evaluation if the ET is compiled to a callable in comparison to basic hSAD. As tier checkpointing typically increases the number of auxiliary variables in the hSAD derivative trace and therefore increases computational complexity of the resulting derivative, it should only be employed when the expression graph associated with a function is large enough that the computational implementation is consuming excessive memory.

3.5.3 Lagrangian Hessian

When numerically solving an OCP using a direct collocation method, the resulting NLP subproblem is of the form given by eqs. (2.64) to (2.66). To solve this NLP subproblem efficiently using a gradient-based NLP solver, first- and second-order derivative information is required. These first-order derivatives are the gradient of the objective function

$$\mathbf{g} = \frac{\partial \mathcal{J}}{\partial \mathbf{X}} \tag{3.77}$$

and the Jacobian of the constraints

$$\mathbf{G} = \frac{\partial \mathbf{C}}{\partial \mathbf{X}}\,. \tag{3.78}$$

The second-order derivatives are the Lagrangian Hessian

$$\mathbf{H} = \frac{\partial^2 \mathcal{L}}{\partial \mathbf{X}^2}\,, \tag{3.79}$$

where the Lagrangian, $\mathcal{L}$, is

$$\mathcal{L} = \sigma\mathcal{J} + \sum_{1}^{n_C} \lambda_i C_i \,. \tag{3.80}$$

In eq. (3.80), σ is the objective factor and n_C is the number of constraints in $\mathbf{C}$. Additionally, λ_i and C_i are the ith Lagrange multiplier and constraint respectively.

First-order derivative matrices can be readily computed using hSAD. Thus, $\mathbf{g}$ and $\mathbf{G}$ can be produced by conducting a single hSAD pass on each of $\mathcal{J}$ and $\mathbf{C}$ respectively. The second-order derivative matrix, $\mathbf{H}$, can be naively computed using hSAD by constructing a DAG for $\mathcal{L}$, conducting an hSAD pass to produce the n_C long vector $\frac{\partial\mathcal{L}}{\partial\mathbf{X}}$ and then conducting a further hSAD pass on the vector $\frac{\partial\mathcal{L}}{\partial\mathbf{X}}$ to produce the matrix $\mathbf{H}$. It is possible, however, to compute $\mathbf{H}$ using only a single hSAD pass by reusing results from the first-order derivatives.

Differentiating eq. (3.80) with respect to $\mathbf{X}$ gives

$$\frac{\partial\mathcal{L}}{\partial\mathbf{X}} = \sigma\frac{\partial\mathcal{J}}{\partial\mathbf{X}} + \sum_{1}^{n_C} \lambda_i \frac{\partial C_i}{\partial\mathbf{X}} = \sigma\mathbf{g} + \boldsymbol{\lambda}\mathbf{G}\,, \tag{3.81}$$

where $\boldsymbol{\lambda}$ is the vector of length n_C of Lagrange multipliers. Thus, if $\mathbf{g}$ and $\mathbf{G}$ are computed first using hSAD, $\frac{\partial\mathcal{L}}{\partial\mathbf{X}}$ can be readily computed using: the auxiliary variables from the modified DAGs of $\mathcal{J}$ and $\mathbf{c}$; a symbolic scalar multiplication with σ; a symbolic vector-matrix product with $\boldsymbol{\lambda}$; and a symbolic vector addition. This is a considerably more computationally efficient means by which to calculate $\frac{\partial\mathcal{L}}{\partial\mathbf{X}}$ than conducting a hSAD pass on $\mathcal{L}$ as using eq. (3.81) does not require the computation of any new partial derivatives

$\mathbf{H}$ can then be computed using a single hSAD pass on $\frac{\partial\mathcal{L}}{\partial\mathbf{X}}$. The three required derivatives (one first-order vector of partial derivatives, one first-order matrix of partial derivatives and one second-order matrix of partial derivatives) can, thus, be computed using only three hSAD passes plus a small amount of additional expression graph manipulation.

3.6 Software Implementation: *Dash*

The theoretical performance of hSAD has been discussed in section 3.4.3. However, in order to assess its performance in practice, a computational implementation needed to be developed. This section details the development of a supplementary derivative backend (section 2.6.2) for the open-source general purpose optimal control software *Pycollo* (section 2.6). This derivative backend, known as the *Dash*

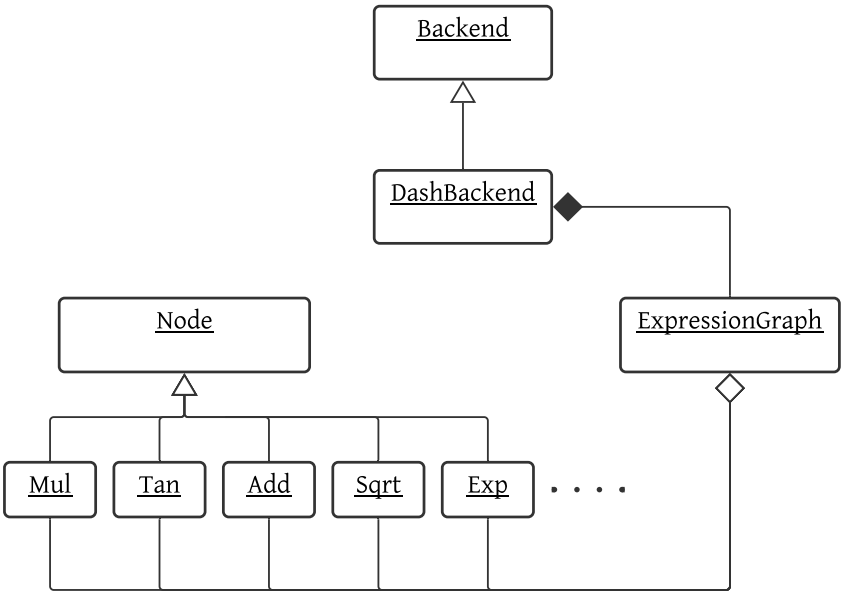


Figure 3.5: A simplified diagram of the architecture of the *Dash* backend to *Pycollo* using the *Unified Modelling Language* (UML).

backend, implements the hSAD algorithm and acts as a second selectable derivative backend to *Pycollo*, alongside the *CasADi* backend (section 2.6). While the core functionality of the *Dash* backend (sections 3.6.1 to 3.6.3) is generalised, the backend also implements OCP-specific functionality (sections 3.6.4 to 3.6.6) to support efficient interoperability with *Pycollo*.

3.6.1 Architecture

The *Dash* backend utilises an OOP architecture to facilitate the implementation of hSAD, primarily so that OO can be leveraged. OO is used to facilitate integration with the *Pycollo application programming interface* (API). This allows user-supplied expressions to be defined easily and the DAG to efficiently modified during the application of the hSAD equations (eqs. (3.47) and (3.49)).

The *Dash* backend defines a pair of base classes, the `Node` and `ExpressionGraph` classes, which form the building blocks of the hSAD software implementation (fig. 3.5). Note that if an explicit namespace is not stated in the following explanation, all functions and classes are part of the `pycollo` namespace. As such, the terminology used here is equivalent to having imported *Pycollo* using `from pycollo import *`. At the lowest level, `Node` objects are instantiated by the *Pycollo* `Variable` alias (for reference, `casadi.SX.sym` objects are instead instantiated when the *CasADi* backend is used). Aliased `Node` objects are used to construct the definitions of the OCP

Arithmetic Operation	Implemented As	Node Class	Number of Operands	Differentiates To
$a + b$	`a + b` `add(a, b)`	`Add`	2	`1, 1`
$a - b$	`a - b` `sub(a, b)`	`Sub`	2	`1, -1`
ab	`a * b` `mul(a, b)`	`Mul`	2	`b, a`
$\frac{a}{b}$	`a / b` `div(a, b)`	`Div`	2	`Recip(b)`, `Neg(Mul(Div(a, b), Recip(b)))`
$-a$	`- a` `neg(a)`	`Neg`	1	`-1`
$\frac{1}{a}$	`1 / a` `recip(a)`	`Recip`	1	`Neg(Sqr(Recip(a)))`
$\sqrt{a}$	`sqrt(a)`	`Sqrt`	1	`Recip(Mul(Sqrt(a), 2))`
a^2	`a**2` `sqr(a)`	`Sqr`	1	`Mul(a, 2)`
a^3	`a**3` `cube(a)`	`Cube`	1	`Mul(Sqr(a), 3)`
a^b	`a**b` `pow(a, b)`	`Pow`	2	`Mul(b, Pow(a, Sub(b, 1)))`, `Mul(Pow(a, b), Log(a))`
$\exp(a)$	`exp(a)`	`Exp`	1	`Exp(a)`
$\log(a)$	`log(a)`	`Log`	1	`Recip(a)`
$\sin(a)$	`sin(a)`	`Sin`	1	`Cos(a)`
$\cos(a)$	`cos(a)`	`Cos`	1	`Neg(Sin(a))`
$\tan(a)$	`tan(a)`	`Tan`	1	`Sqr(Sec(a))`
$\sec(a)$	`sec(a)`	`Sec`	1	`Mul(Sec(a), Tan(a))`
$\csc(a)$	`csc(a)`	`Csc`	1	`Neg(Mul(Csc(a), Cot(a)))`
$\cot(a)$	`cot(a)`	`Cot`	1	`Neg(Sqr(Csc(a)))`
$\arcsin(a)$	`asin(a)`	`Asin`	1	`Recip(Sqrt(Sub(1, Sqr(a))))`
$\arccos(a)$	`acos(a)`	`Acos`	1	`Neg(Recip(Sqrt(Sub(1, Sqr(a)))))`
$\arctan(a)$	`atan(a)`	`Atan`	1	`Recip(Add(1, Sqr(a)))`
$\operatorname{arccot}(a)$	`acot(a)`	`Acot`	1	`Neg(Recip(Add(1, Sqr(a))))`
$\sinh(a)$	`sinh(a)`	`Sinh`	1	`Cosh(a)`
$\cosh(a)$	`cosh(a)`	`Cosh`	1	`Sinh(a)`
$\tanh(a)$	`tan(a)`	`Tanh`	1	`Sub(1, Sqr(Tanh(a)))`
$\operatorname{arcsinh}(a)$	`asinh(a)`	`Asinh`	1	`Recip(Sqrt(Add(Sqr(a), 1)))`
$\operatorname{arccosh}(a)$	`acosh(a)`	`Acosh`	1	`Recip(Sqrt(Sub(Sqr(a), 1)))`
$\operatorname{arctanh}(a)$	`atan(a)`	`Atanh`	1	`Recip(Sub(1, Sqr(a)))`

Table 3.6: Operations supported by the *Dash* backend and their implementation details. `a` and `b` are used to represent generic instantiated `Node` objects.

functions using *Pycollo* syntax. OO overloads the primitive arithmetic operators (e.g. the `*` multiplication operator) and functions representing other atomic operations (e.g. the `tan()` function to support the trigonometric operation) are also supplied. Both the overloaded operators and the atomic operation functions return `Node` objects representing the operation applied with references to the other `Node` objects that are operands. This is implemented by the `Node` class being subclassed many times, with a child class existing to represent each of the atomic operations supported by the implementation. For example, the overloaded `*` multiplication operator returns a `Mul` object while the `tan`() function returns a `Tan` object, with both the `Mul` and `Tan` classes being subclasses of the `Node` class. The full list of operations supported by the *Dash* backend is shown in table 3.6. The set of operations was chosen to include all of those recommended as essential by [136] plus some additional operations that were found to be useful when implementing test problems during development. Note that all of the supported operations are differentiable to smooth continuous functions. This is to ensure that all of the derivatives generated by the *Dash* backend are smooth, as is required to ensure the efficient and successful solving of OCPs [36, 43, 131].

`Node` objects can represent both scalars and variable length vectors. This is to ensure that OCP derivatives can be generated efficiently by treating continuous-time variables, which will be discretised to multiple decision variables in the NLP subproblem, as single nodes in the expression graph. This is valid as all of the discretised variables associated with a single continuous-time variable will be actioned by the same operations and, therefore, many repeated parallel calculations can be avoided. As such, state and control variables are automatically treated as variable length vector nodes by the *Dash* backend, while integral, time and static parameter variables are treated as scalar nodes. The *Dash* backend implements propagation rules to ensure that operations involving variable length vector nodes are handled correctly. That is, if an operation has a variable length vector node as at least one of its operands, then the operation is assumed to apply element-wise. Another assumption made by the *Dash* backend is that all variable length vector nodes will have the same length when used in computations. This has the consequence that all vector and matrix expressions need to be expressed element-wise when constructing equations using the *Pycollo* API.

The *Dash* backend instantiates a single `ExpressionGraph` which is responsible for building a combined DAG for all of the user-supplied functions. The single DAG approach is critical as it allows the Lagrangian Hessian to be derived efficiently using the already-generated objective function gradient and constraints Jacobian (section 3.5.3). As such, the `ExpressionGraph` class possesses attributes that explicitly map to the OCP functions. This means that the required OCP derivatives can easily

be generated by the *Dash* backend and in a form where the predictable structure of the NLP subproblem can be exploited (section 3.6.4). The other main responsibility of the `ExpressionGraph` object is to collate all of the OCP variables such that when the hSAD algorithm is applied to the OCP functions, analytic derivatives are generated with respect to all of the required variables simultaneously.

3.6.2 hSAD Algorithm

Once the `ExpressionGraph` object has been constructed, it will contain attributes detailing how the OCP object function and constraints can be populated using nodes contained within the DAG. The hSAD algorithm can then be used to generate the first-order objective function gradient, the first-order constraint Jacobian and the second-order Lagrangian Hessian. These three derivatives are produced using five hSAD passes. Five derivative passes are required as both the constraint Jacobian and Lagrangian Hessian contain functions of both state and state endpoint variables. As such, one pass is required for the continuous variables and another for the endpoint variables for both derivative matrices. To be explicit, the defect constraints will be constructed by differentiating the state equations with respect to (among other variables) the continuous state variables. On the other hand, the endpoint constraints can only be functions of the state at the phase initial and final times. Therefore, the `ExpressionGraph` object constructs two sets of variables, those that correspond to the continuous functions (e.g. state equations, path constraints and integrand functions) and those that correspond to the endpoint functions (e.g. objective function and endpoint constraints).

Before any hSAD passes are conducted, the entire DAG is tier partitioned using a DFS algorithm [80]. Auxiliary variables are then assigned an arbitrary index in their tier so that matrix indices will remain consistent between all matrices in the hSAD equations and also between distinct hSAD passes. The first pass derives the objective function gradient by differentiating the objective function with respect to the endpoint variables. This is done using a sparse matrix implementation of the hSAD equations with the partial derivative matrices being populated dynamically using each `Node` objects subclass type to provide the partial derivatives, and its tier index and `operands` attribute to determine the correct matrix indices. The second and third hSAD passes differentiate the constraints with respect to the continuous and endpoint variables respectively, deriving the continuous and endpoint constraint Jacobians.

Before the Lagrangian Hessian can be derived, new variables to represent the objective factor and Lagrange multipliers are introduced into the DAG as variable

nodes. The first-order Lagrangian gradient can then be constructed using these new variables, the previously derived objective function gradient and the previously derived constraint Jacobian. This is done by modifying the `ExpressionGraph` object's DAG with the previously mentioned primitives and additional `Add` and `Mul` nodes. The Lagrangian gradient is then differentiated twice, once with respect to the continuous variables in the fourth hSAD pass and once with respect to the the endpoint variables in the fifth, to derive the Lagrangian Hessian. With this complete, all of the required OCP derivative information has been generated.

Note that the *Dash* backend implements a pure-*Python* implementation of sparse matrices to facilitate the application of the hSAD equations. This was necessary in order to support the OO required to dynamically modify the DAG with the required additional nodes generated during the hSAD passes. Existing sparse matrix implementations (such as that in the *Sparse* subpackage of *SciPy* [321]) could not be used along with the *Dash* backend's `Node` class as they only support certain numeric data types and do not support custom types. As this sparse matrix implementation is pure-*Python*, it would almost certainly be inefficient in comparison to a dedicated implementation written in a low-level language like *C* or *C++*.

3.6.3 Dynamic Code Generation

With the full modified DAG containing the objective function gradient, constraint Jacobian and Lagrangian Hessian constructed, callables to evaluate these (along with the objective function and constraint) can be generated using dynamic code generation. For each OCP function and derivative function in turn, the output is analysed and the subset of nodes required as auxiliary variables for its evaluation is generated. These auxiliary variables are then topologically sorted based on their tier number to produce a serial ordered evaluation procedure. The $\mathbb{T}_0$ variables are extracted as these constitute the inputs. ST is then used to convert the evaluation procedure into source code. This is done by generating code for a new function definition with a unique name and with the inputs as its arguments. Each auxiliary variable generates a single line of source code within the function body representing how its auxiliary variable can be generated mathematically from previously evaluated auxiliary variables. For example, if the node w_3 represents a binary multiplication between two other nodes, w_1 and w_2, (i.e. $w_3 = w_1 w_2$) this will be represented by a `Mul` object corresponding to w_3 in the `ExpressionGraph` object's DAG, which will generate the string `w3 = w1 * w2`. Finally, the source code is then used to generate a callable using *Python*'s `exec` built-in.

To improve the computational performance of the generated callables, *just-in-*

time (JIT) compilation by *Numba* is leveraged. This ensures that any numerical evaluation is conducted using low-level code operating on primitive data types. As a result, the JIT-compiled callables are able to compete with compiled *C* and *C++* code on speed [206]. *Numba* handles the JIT compilation of the source code containing operations on array types in an efficient manner such that the callables need to be recompiled if array values change in length between calls, as is almost always the case between mesh iterations. As *Numba* is used to JIT compile the dynamically generated callables, additional optimisations, such as utilising *single instruction, multiple data* (SIMD) to efficiently evaluate vector expressions, are applied automatically by the *LLVM* [209] compiler behind *Numba* [206].

3.6.4 Sparse Matrices

The sparse derivative matrices, namely the constraint Jacobian and Lagrangian Hessian, are well-known to have predictable structure and can be constructed efficiently from the derivatives of the OCP functions and collocation scheme-specific weighting matrices [8, 41, 264]. Specifically, the constraint Jacobian, contains 20 different types of block, resulting from it representing four different types of constraint function (defect, path, integral and endpoint) and having been differentiated with respect to five different types of variable (state, control, integral, time and static parameter). Of these 20 blocks, three are zero matrices (the partial derivatives of the defect constraints with respect to the integral variables, the path constraints with respect to the integral variables and the endpoint constraints with respect to the control variables) and one is the identity matrix (the partial derivatives of the integral constraints with respect to the integral variables) [8, 41, 264]. The remaining 16 blocks also follow a predictable predominantly sparse structure. The full sparse constraint Jacobian matrix can, therefore, be constructed by first generating the 12 nontrivial submatrices plus an identity matrix and then combining these as a sparse block matrix.

The Lagrangian Hessian also exhibits a predictable structure of the same form, this time containing 15 different types of block. This results from the Lagrangian Hessian representing a quantity twice-differentiated with respect to five different types of variable in addition to the fact that it is symmetric and therefore the upper right triangle of entries can be ignored [8, 41, 264]. Of the 15 required blocks, one is a zero matrix (the second-order partial derivative of the Lagrangian with respect to the control and integral variables) [8, 41, 264]. Again, by generating the 14 nontrivial submatrices and combining these as a sparse block matrix, the lower triangular symmetric sparse Lagrangian Hessian can be constructed.

The *Dash* backend utilises the `sparse` subpackage of *SciPy* to construct the sparse block matrices numerically. Firstly, the vector of decision variables is reshaped such that all discretised decision variables corresponding to a single continuous-time (state or control) OCP variable are combined into a single array data type. This enables the callables compiled by the dynamic code generator, which have the OCP variables as their arguments, to be called directly, evaluating the mesh-specific OCP functions to return array data types as well. The output is then partitioned by type, such that each block of the sparse matrix being constructed has access to only the relevant nonzeros. Sparse submatrices are then structured using the sparse matrix operations from *SciPy* before these are combined to produce the final block matrix. The nonzeros are then sorted by their indices before they are returned.

This block sparse matrix construction methodology is used because it is a simple way to ensure that all nonzero indices are correctly computed. This approach, while being vastly more efficient than determining the derivative matrices of the full NLP subproblem directly, is still inefficient. This is because it involves constructing the actual sparse matrices on every function call rather than just determining the nonzeros in a predictable order. A much better approach, which has yet to be implemented for the *Dash* backend, would involve having the nonzero values be returned directly by the callables compiled by the dynamic code generator. This would then require a different algorithm for determining the indices of the nonzero prior to solving the NLP subproblem. Specifically, the indices of the nonzeros would need to be ordered such that they correctly correspond to the output of the constraint Jacobian and Lagrangian Hessian callables.

3.6.5 Sparsity Structure Detection

Each ET produced by the hSAD algorithm corresponds to the analytic derivative of the target function. Furthermore, the ETs populate the entire Jacobian and Hessian matrices in a single function call. This means that only the absolute sparsity structure of these matrices needs to be determined in order to construct the sparse NLP matrices [8] and this can be done by propagating *not a number* (NaN) values through each of the function calls in turn. Fortunately, the absolute sparsity structure is determined as one output of the hSAD algorithm. As such, no further OCP-specific preprocessing is required here.

With the sparsity structure of the OCP determined, the mesh-specific sparsity structures associated with a NLP subproblem can be determined. Section 3.6.4 details how the NLP constraint Jacobian and Lagrangian Hessian can be constructed using the OCP functions. The *Dash* backend utilises the functions that numerically

evaluate the NLP constraint Jacobian and Lagrangian Hessian. These functions construct the entire block sparse matrices before sorting and returning just the nonzeros. Therefore, the indices of the nonzeros can be readily extracted from this function after the nonzeros have been sorted.

3.6.6 Interface with *Ipopt*

Ipopt requires that it is linked with callables for the numerical evaluation of the objective function, objective function gradient, constraints, constraints Jacobian and Hessian Lagrangian provided at a point in the decision space. These will typically be called at least once per NLP iteration. The *Dash* backend links the callables compiled using its dynamic code generator to *Ipopt* before each mesh iteration. Note that although these callables are generated as part of a mesh-independent preprocessing step, each mesh iteration requires a new NLP subproblem be defined. Therefore, these callables need to be linked with a new *Ipopt* NLP before each mesh iteration is solved.

Ipopt also requires that the nonzero indices of the constraint Jacobian and Lagrangian Hessian be supplied before the NLP is solved. The indices of the nonzeros in these two sparse matrices are determined using the method described in section 3.6.5 and are linked to *Ipopt* at the start of each mesh iteration.

3.6.7 Properties

As a result of the design decisions described by sections 3.6.1 to 3.6.6, the computational implementation has the following properties:

1. allows target functions to be expressed as a DAG;
2. applies hSAD to produce ETs for analytic first derivatives;
3. efficiently generates Lagrangian Hessians from already-generated analytic representations of objective function gradients and constraint Jacobians;
4. compiles ETs to efficient low-level executable functions that can be called directly by NLP solvers such as *Ipopt*;
5. callables treat the discretised state and control variables as vector functions (of unspecified) length, so that, subject to some reshaping, they can be called irrespective of the mesh discretisation; and

6. conducts all of the differentiation and compilation steps (excluding the mesh-specific sparsity detection, which has a computational cost less than that of a single NLP iteration) as a preprocessing stage of the OCP solve.

3.7 *Dash* Investigations

In order to investigate the practical benefits of applying hSAD to numerically solve OCPs and to assess the performance of the implemented *Dash* backend to *Pycollo*, a number of investigations were conducted. These investigations compared the performance of the supplementary *Dash* backend to *Pycollo* (section 3.6) against the previously implemented *CasADi* backend (section 2.6). While the two backends interface with *Pycollo* identically, they take different approaches in their operation and so their comparison enables the practical properties of hSAD to be investigated in the context of a well-established prevalent method.

As outlined in section 2.6, *Pycollo* supports the use of the AD software package *CasADi* to supply first- and second-order derivative information and to interface with the NLP solver *Ipopt*. *CasADi* abstracts away some of the complexities of formulating NLPs, such a correctly calculating the Hessian of the Lagrangian using only the objective function and constraints. This has the implication that *CasADi* interfaces directly with the NLP solver. In the context of *Pycollo*, this means that the full mesh-specific NLP subproblem must be defined for each mesh iteration. As such, *CasADi* must reconduct all derivative generation (and required compilation so that derivative information can be numerically evaluated) for each new mesh on which the OCP is to be solved. This does not matter if the OCP can be solved while meeting the desired mesh tolerance on the initial mesh. However, if mesh refinement is required, as is the case in the majority of practical cases, then this redetermination and recompilation of all of the derivative information for each mesh iteration results in a large amount of repetition of very similar tasks.

The *Dash* backend takes a different approach and attempts to front-load all derivative calculation and all of the compilation of the callables required by the NLP solver. It is able to do this by applying hSAD to the OCP constraints (or constraint-related functions in the OCP definition) and constructing the NLP subproblem derivatives on a per-mesh-iteration basis by exploiting the well-known block sparsity structure of the derivative matrices [8, 41, 264]. The exact sparsity pattern for a specific mesh iteration is readily determined algorithmically by constructing the block sparse matrices using a sparse matrix representation, in combination with the symbolic derivative information obtained from the hSAD pass.

The *Dash* and *CasADi* backends to *Pycollo* take contrasting approaches to how and when OCP derivative information is generated during the numerical solving of an OCP. The *CasADi* backend requires all derivatives to be determined (including their exact sparsity structure using a graph colouring algorithm [20]) and compiled before each mesh iteration. The *Dash* backend, on the other hand, determines all of the derivative information required and compiles discretisation-independent callables during a preprocessing step before the initial mesh iteration. It then only requires a single derivative-related per-mesh-iteration preprocessing step to determine the exact sparsity structure of the mesh-specific constraint Jacobian and Lagrangian Hessian. The approach taken by the *Dash* backend should, theoretically, offer a significant performance improvement over the approach taken by the *CasADi* backend by avoiding much of the repeated similar work.

A set of three performance metrics were selected in order to compare the two backends. The first metric was the time spent preprocessing the derivatives during OCP setup. This is denoted by $\mathcal{T}$, with appropriate subscripts to identify the two backends. This metric is important as it captures the actual differentiation of the OCP functions and compilation of the NLP derivative callables that are conducted by the *Dash* backend. As the *CasADi* backend does not conduct any preprocessing of this type, $\mathcal{T}_{\text{CasADi}} = 0$ in all situations. The second metric, the time spent preprocessing the derivatives per mesh iteration, is denoted by $\mathcal{P}$. $\mathcal{P}_{\text{CasADi}}$ accounts for all derivative processing done by the *CasADi* backend, which includes: constructing the mesh-specific NLP subproblem; differentiating the NLP subproblem to produce the NLP derivatives; compiling callables for evaluating the NLP functions and NLP derivatives; and determining the sparsity of the NLP Jacobian and Hessian. Conversely, $\mathcal{P}_{\text{hSAD}}$ only accounts for the determination of the mesh-specific constraint Jacobian and Lagrangian Hessian sparsities. This is because all other steps are conducted beforehand and attributed to $\mathcal{T}_{\text{hSAD}}$. It is important to note that even if an OCP requires many mesh iterations to be solved to a desired mesh tolerance, the cost associated with $\mathcal{T}$ will only be incurred once. On the other hand, the cost associated with $\mathcal{P}$ is incurred on each mesh iteration. Thus, the total cost relating to $\mathcal{P}$ increases proportional to the number of mesh iterations. The third and final metric, the average time spent evaluating the external derivative callables called by *Ipopt* during a single NLP iteration, is denoted by $\mathcal{I}$. This metric is important as it helps quantify the efficiency of the callables that are provided to the NLP solver by both backends. Taking all of these metrics in to account, if m mesh iterations are required to accurately solve the OCP and $\bar{n}$ NLP iterations are required on average to solve a single NLP subproblem, then the total cost associated with derivative-related operations will be given by $\mathcal{T} + m\left(\mathcal{P} + \bar{n}\mathcal{I}\right)$.

As both backends produce exact analytical derivatives, they should in theory

generate identical derivative information. This is not actually the case in practice due to the fact that there will be slight differences in the numerical evaluation procedures produced by their compiled callables. However, as any differences arise due to uncontrollable factors associated with compilation, this is, therefore, not of primary concern in relation to these investigations and will not be considered here. Furthermore, it was found during testing that, NLP solving progressed identically when both backends were used. That is, provided that the same mesh discretisation and initial guess were used, the same number of NLP iterations were required to converge to the same solution within the specified NLP tolerance. Therefore, for the purposes of this investigation, metrics related to NLP performance (e.g. NLP solve time, number of mesh iterations required etc.) are not included, particularly as these were the focus of section 2.7.

A subset of three problems, all of which have previously been solved using *Pycollo* (section 2.7), were selected for this investigation. The first problem selected was the hypersensitive problem [36, 284] (section 2.7.4). The second problem was the space station attitude control problem [36, 267] (section 2.7.6). These two problems were specifically selected as, when formulated in *Pycollo* and analysed using the *Dash* backend, it was found that they yielded the smallest and largest expression graphs (by node count) of all of the five solvable example problems in section 2.7. They should, therefore, correspond to the example problems with the cheapest and most expensive to derive and evaluate derivatives of the example problems, thus providing good lower and upper bounds on expected real-world performance of hSAD and *Pycollo* with its *Dash* backend. As both of these problems are single phase, a third problem, the multiphase tumour anti-angiogenesis problem [36, 211] (section 2.7.8), was also selected. Between these three example problems, all problem types solvable by *Pycollo* are covered. This includes: single and multiphase problems; problems with state, control, integral, time and static parameter variables; and problems with defect, path, integral, endpoint and state endpoint constraints.

Each of the three chosen example problems was investigated across a range of different mesh densities and, therefore, NLP problem sizes. Meshes containing $K = \{10, 20, 50, 100, 200, 500, 1000\}$ mesh sections, with each mesh section containing eight discretisation nodes, were used. The lowest value of $K = 10$ was chosen as this is the default initial mesh density in *Pycollo*, making it a sensible lower bound. The greatest value of $K = 1000$ is approximately an order of magnitude greater than the densest mesh observed during the benchmarking experiments in section 2.7 and, therefore, was assessed as a sensible upper bound that might be observed in the case of a highly nonlinear and complex OCP. These values for K, therefore, cover the range of magnitudes of mesh densities that are likely to be seen in practice. Seven approximately logarithmically separated values for K were chosen because this was

K	$\mathcal{T}_{\text{hSAD}}$ (s)	$\mathcal{T}_{\text{CasADi}}$ (s)	$\mathcal{P}_{\text{hSAD}}$ (s)	$\mathcal{P}_{\text{CasADi}}$ (s)	$\mathcal{I}_{\text{hSAD}}$ (s)	$\mathcal{I}_{\text{CasADi}}$ (s)
10	0.0115	0	0.0199	0.0658	0.007 71	0.000 063 0
20	0.0114	0	0.0229	0.124	0.0106	0.000 089 0
50	0.0114	0	0.0351	0.295	0.0190	0.000 134
100	0.0115	0	0.0656	0.611	0.0310	0.000 247
200	0.0114	0	0.133	1.37	0.0603	0.000 537
500	0.0114	0	0.340	5.70	0.204	0.001 40
1000	0.0114	0	1.08	19.5	0.618	0.004 19

Table 3.7: Performance comparison for the hypersensitive problem by *Pycollo* using its *Dash* and *CasADi* backends. K denotes the number of mesh sections, $\mathcal{T}$ denotes the time spent preprocessing the derivatives during OCP setup, $\mathcal{P}$ denotes the time spent preprocessing the derivatives per mesh iteration, and $\mathcal{I}$ denotes the average time spent evaluating the external derivative callables called by *Ipopt* during a single NLP iteration.

deemed a reasonable sample size and because the lower and upper limits for K differed by two orders of magnitude. Eight discretisation nodes per mesh section were used despite this being greater than the default *Pycollo* value of four. This was for two reasons. Firstly, the mesh refinement algorithm in *Pycollo* can adjust the number of discretisation nodes per mesh section. By default, *Pycollo* limits this to be between four and 12 and, thus, a value of eight is the average of what will be encountered using the default *Pycollo* settings. Secondly, using more discretisation nodes per mesh section results in larger block sizes when constructing the derivative matrices. This results in a greater cost associated with constructing the sparse block constraint Jacobian and Lagrangian Hessian matrices when using the *Dash* backend. Thus, a more realistic comparison to the *CasADi* backend (which does not need to construct these matrices in a block-wise fashion on each NLP iteration) is presented than if the default value of four had been used.

3.7.1 Hypersensitive Problem

The full definition of the hypersensitive problem is presented in section 2.7.4. This OCP was selected for this set of investigations as it resulted in the most compact expression graph when formulated in *Pycollo* using the *Dash* backend. Therefore, it represents a good lower bound on the computational complexity expected when solving real-world OCPs. Table 3.7 gives the measured benchmark metrics ($\mathcal{T}$, $\mathcal{P}$ and $\mathcal{I}$) for all values of K using both the *Dash* and *CasADi* backends. As expected,

$\mathcal{T}_{\text{hSAD}}$ was found to be almost identical for all values of K, with the maximum variation found to be $\pm 0.44\%$, due to the fact that the derivative preprocessing to which this metric corresponds is mesh independent. Additionally, $\mathcal{T}_{\text{CasADi}}$ was zero for all values of K as the kind of derivative preprocessing associated with $\mathcal{T}$ is not conducted by this backend.

$\mathcal{P}_{\text{hSAD}}$ was greater than $\mathcal{T}_{\text{hSAD}}$ for all values of K (table 3.7). Both $\mathcal{P}_{\text{hSAD}}$ and $\mathcal{P}_{\text{CasADi}}$ were found to increase with K. $\mathcal{P}_{\text{CasADi}}$ was found to be 3.31 times larger than $\mathcal{P}_{\text{hSAD}}$ when $K = 10$, increasing to 18.1 times larger for $K = 1000$.

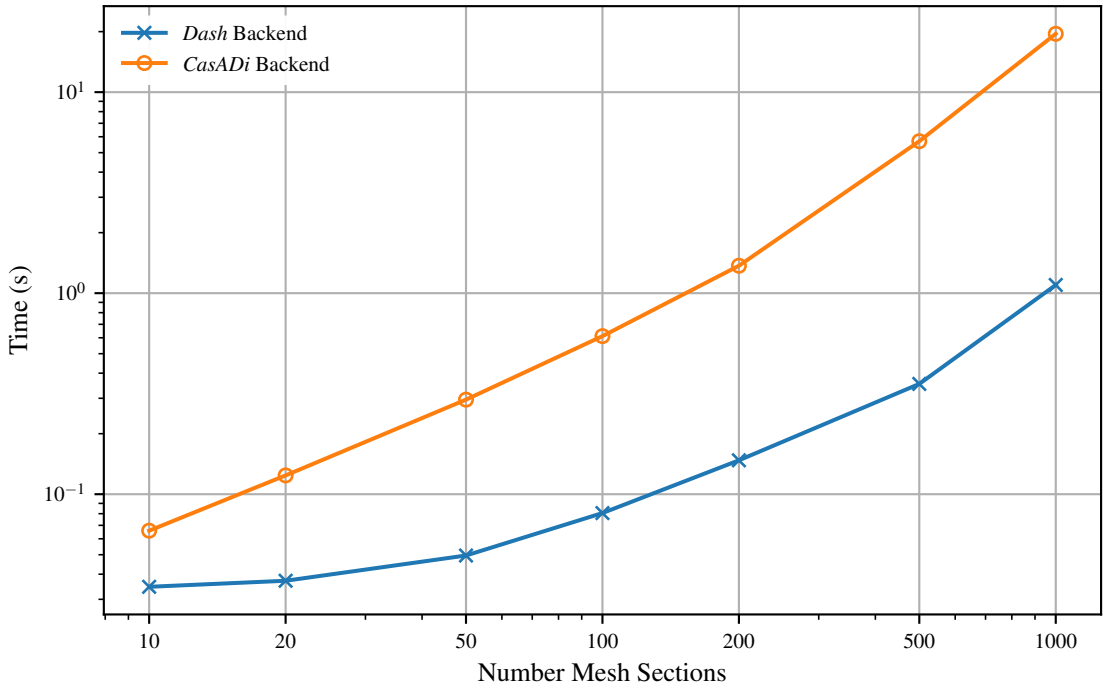
The combined metric, $\mathcal{T} + \mathcal{P}$, which essentially represents the total derivative preprocessing time if an OCP were to be solved in a single mesh iteration, is plotted against K in fig. 3.6a. This is plotted on log-log axes due to the multiple orders of magnitude covered by both K and the total preprocessing time, $\mathcal{T} + \mathcal{P}$. It can be seen that the total derivative preprocessing time incurred by the *CasADi* backend is always greater than with the *Dash* backend for the same value of K. Furthermore, the rate of increase in the total derivative preprocessing time with increased K is greater for the *CasADi* backend and is illustrated by the steeper line gradient.

$\mathcal{I}_{\text{CasADi}}$ was found to be at least 108 times faster ($K = 20$) than $\mathcal{I}_{\text{hSAD}}$ and at most 148 times faster ($K = 1000$). There was, however, no consistent correlation between the magnitude of the difference between $\mathcal{I}_{\text{hSAD}}$ and $\mathcal{I}_{\text{CasADi}}$ with K. It is likely that there was some inaccuracy in the profiling of the NLP iterations using the *CasADi* backend due to the fact that the times were so fast. Therefore, it is more reliable to say that, for this problem, $\mathcal{I}_{\text{hSAD}}$ was approximately two orders of magnitude slower than $\mathcal{I}_{\text{CasADi}}$.

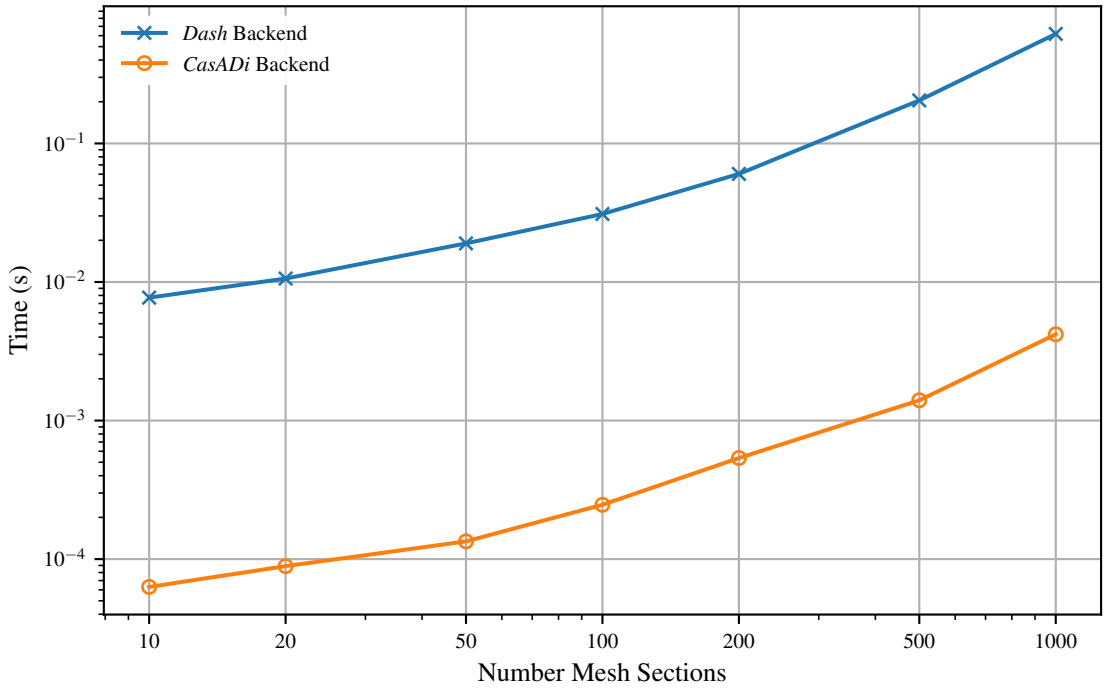
3.7.2 Space Station Attitude Control

The full definition of the space station attitude control problem is presented in section 2.7.6. This OCP was selected for this set of investigations as it resulted in the largest expression graph by node count, when formulated in *Pycollo* using the *Dash* backend. Therefore, it represents a sensible upper bound on the likely computational complexity frequently encountered when solving real-world OCPs. Table 3.8 gives the measured benchmark metrics ($\mathcal{T}$, $\mathcal{P}$ and $\mathcal{I}$) for all values of K using both the *Dash* and *CasADi* backends. Again, $\mathcal{T}_{\text{hSAD}}$ is almost identical for all values of K, with the maximum variation found to be $\pm 0.75\%$, while $\mathcal{T}_{\text{CasADi}}$ was zero for all values of K.

$\mathcal{T}_{\text{hSAD}}$ was greater than $\mathcal{P}_{\text{hSAD}}$ for $K \leq 200$, but exceeded $\mathcal{P}_{\text{hSAD}}$ for $K \geq 500$. Both $\mathcal{P}_{\text{hSAD}}$ and $\mathcal{P}_{\text{CasADi}}$ were found to increase with K. However, it was not possible



(a) Time taken during preprocessing of derivatives



(b) Average time taken per NLP iteration

Figure 3.6: Comparison of the time taken during numerical solving of the hypersensitive problem by *Pycollo* using its *Dash* and *CasADi* backends.

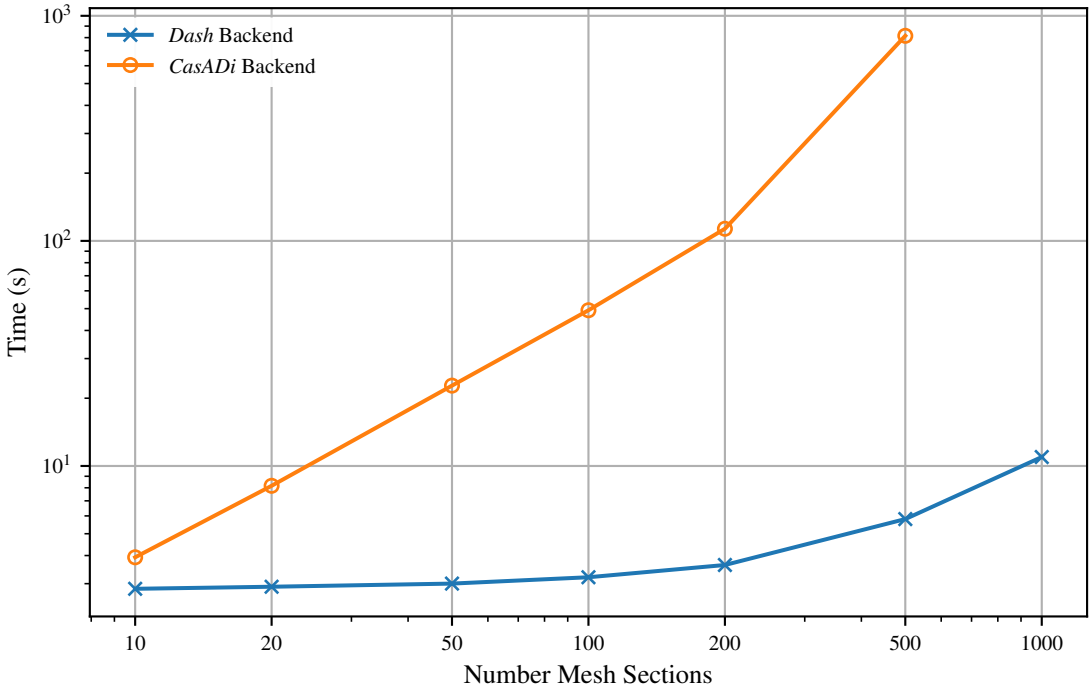
K	$\mathcal{T}_{\text{hSAD}}$ (s)	$\mathcal{T}_{\text{CasADi}}$ (s)	$\mathcal{P}_{\text{hSAD}}$ (s)	$\mathcal{P}_{\text{CasADi}}$ (s)	$\mathcal{I}_{\text{hSAD}}$ (s)	$\mathcal{I}_{\text{CasADi}}$ (s)
10	2.69	0	0.152	3.93	0.0951	0.004 52
20	2.70	0	0.209	8.16	0.127	0.008 43
50	2.67	0	0.325	22.8	0.249	0.0221
100	2.70	0	0.498	49.2	0.417	0.0438
200	2.69	0	0.938	113	0.866	0.0907
500	2.66	0	3.15	815	2.65	0.235
1000	2.70	0	8.25	-	6.94	-

Table 3.8: Performance comparison for the space station attitude control problem by *Pycollo* using its *Dash* and *CasADi* backends. K denotes the number of mesh sections, $\mathcal{T}$ denotes the time spent preprocessing the derivatives during OCP setup, $\mathcal{P}$ denotes the time spent preprocessing the derivatives per mesh iteration, and $\mathcal{I}$ denotes the average time spent evaluating the external derivative callables called by *Ipopt* during a single NLP iteration.

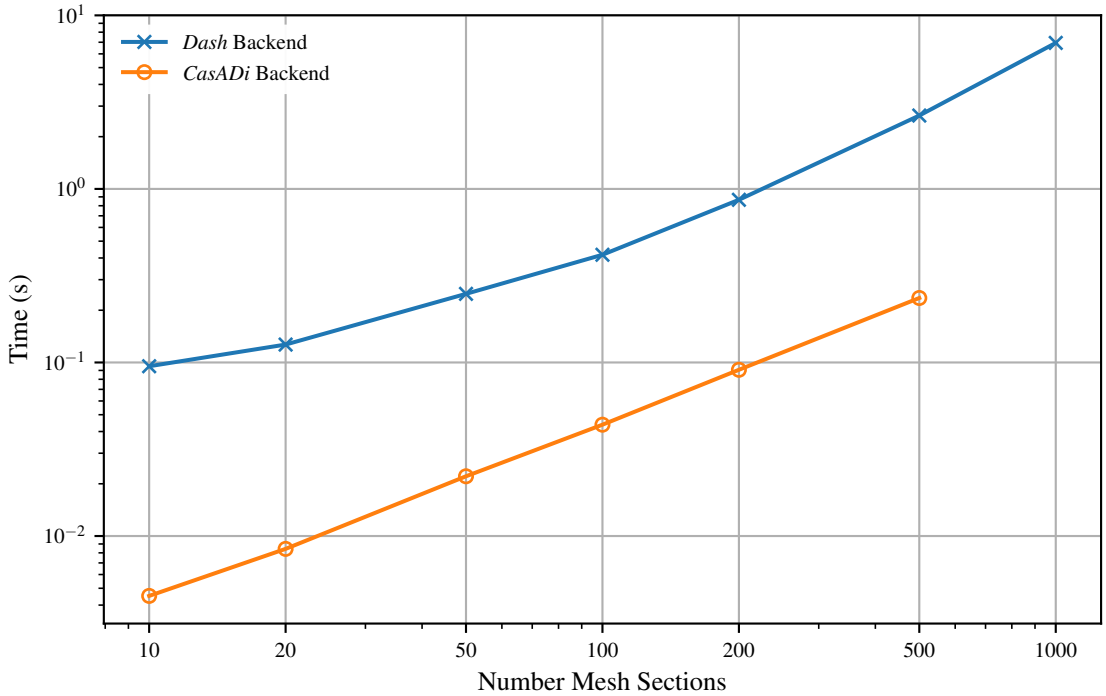
to benchmark the OCP using the *CasADi* backend with $K = 1000$ due to the kernal killing the *Python* process running the investigation test during the mesh iteration derivative preprocessing step. The process was killed due to it exceeding the available memory. For the values of K that could be successfully investigated using both backends, $\mathcal{P}_{\text{hSAD}}$ was found to be between 25.9 ($K = 10$) and 259 ($K = 500$) times faster than $\mathcal{P}_{\text{CasADi}}$. There was a consistent correlation, with the ratio of $\mathcal{P}_{\text{hSAD}}$ to $\mathcal{P}_{\text{CasADi}}$ increasing with each tested value of K.

A log-log plot of $\mathcal{T} + \mathcal{P}$ against K is shown in fig. 3.7a. $\mathcal{T}_{\text{hSAD}} + \mathcal{P}_{\text{hSAD}}$ was universally less than $\mathcal{T}_{\text{CasADi}} + \mathcal{P}_{\text{CasADi}}$, but only by 38.3% for $K = 10$. However, as illustrated by the gradient of the line for the *CasADi* backend, $\mathcal{T}_{\text{CasADi}} + \mathcal{P}_{\text{CasADi}}$ increased with K at a much greater rate than $\mathcal{T}_{\text{hSAD}} + \mathcal{P}_{\text{hSAD}}$.

$\mathcal{I}$ increased with K, as would be expected, when both backends were used. $\mathcal{I}_{\text{CasADi}}$ was again found to be consistently less than $\mathcal{I}_{\text{hSAD}}$ (fig. 3.7b). The largest difference was found when $K = 10$, with $\mathcal{I}_{\text{CasADi}}$ being 21.0 times faster than $\mathcal{I}_{\text{hSAD}}$. This ratio reduced to a minimum of 9.52 times when $K = 100$, with it also being 9.55 times when $K = 200$. However, the ratio between $\mathcal{I}_{\text{CasADi}}$ and $\mathcal{I}_{\text{hSAD}}$ was reasonably consistent for $50 \leq K \leq 500$ so, due to the expected variability associated with benchmarking fast numerical computations, a more reliable conclusion is that $\mathcal{I}_{\text{CasADi}}$ was approximately one order of magnitude faster than $\mathcal{I}_{\text{hSAD}}$ for $K \geq 50$.



(a) Time taken during preprocessing of derivatives



(b) Average time taken per NLP iteration

Figure 3.7: Comparison of the time taken during numerical solving of the space station attitude control problem by *Pycollo* using its *Dash* and *CasADi* backends.

K	$\mathcal{T}_{\text{hSAD}}$ (s)	$\mathcal{T}_{\text{CasADi}}$ (s)	$\mathcal{P}_{\text{hSAD}}$ (s)	$\mathcal{P}_{\text{CasADi}}$ (s)	$\mathcal{I}_{\text{hSAD}}$ (s)	$\mathcal{I}_{\text{CasADi}}$ (s)
10	0.0426	0	0.0315	0.204	0.0153	0.000 095 2
20	0.0422	0	0.0409	0.504	0.0234	0.000 280
50	0.0424	0	0.0764	2.26	0.0453	0.000 538
100	0.0421	0	0.146	7.47	0.0815	0.000 912
200	0.0429	0	0.234	27.4	0.156	0.001 97
500	0.0420	0	0.513	162	0.411	0.004 63
1000	0.0421	0	1.278	623	0.960	0.0102

Table 3.9: Performance comparison for the tumour anti-angiogenesis problem by *Pycollo* using its *Dash* and *CasADi* backends. K denotes the number of mesh sections, $\mathcal{T}$ denotes the time spent preprocessing the derivatives during OCP setup, $\mathcal{P}$ denotes the time spent preprocessing the derivatives per mesh iteration, and $\mathcal{I}$ denotes the average time spent evaluating the external derivative callables called by *Ipopt* during a single NLP iteration.

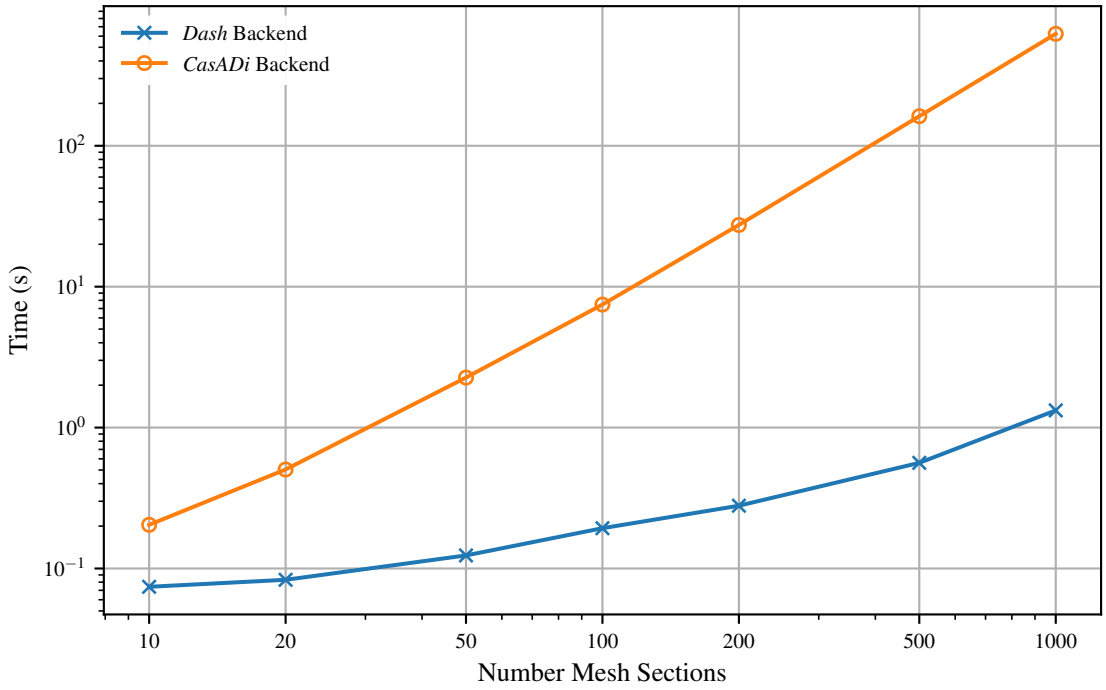
3.7.3 Tumour Anti-Angiogenesis

The full definition of the tumour anti-angiogenesis problem is presented in section 2.7.8. This OCP was selected for this set of investigations as an example of a multiphase problem that can be solved by *Pycollo*. It should be noted that, despite being a multiphase problem, this OCP can be considered to be less complex than the space station attitude control problem presented previously owing to a number of reasons, including: a smaller number of state variables; less complex dynamical equations; no control variables in the second phase; and bang-bang optimal solution to the control between a constant value and zero.

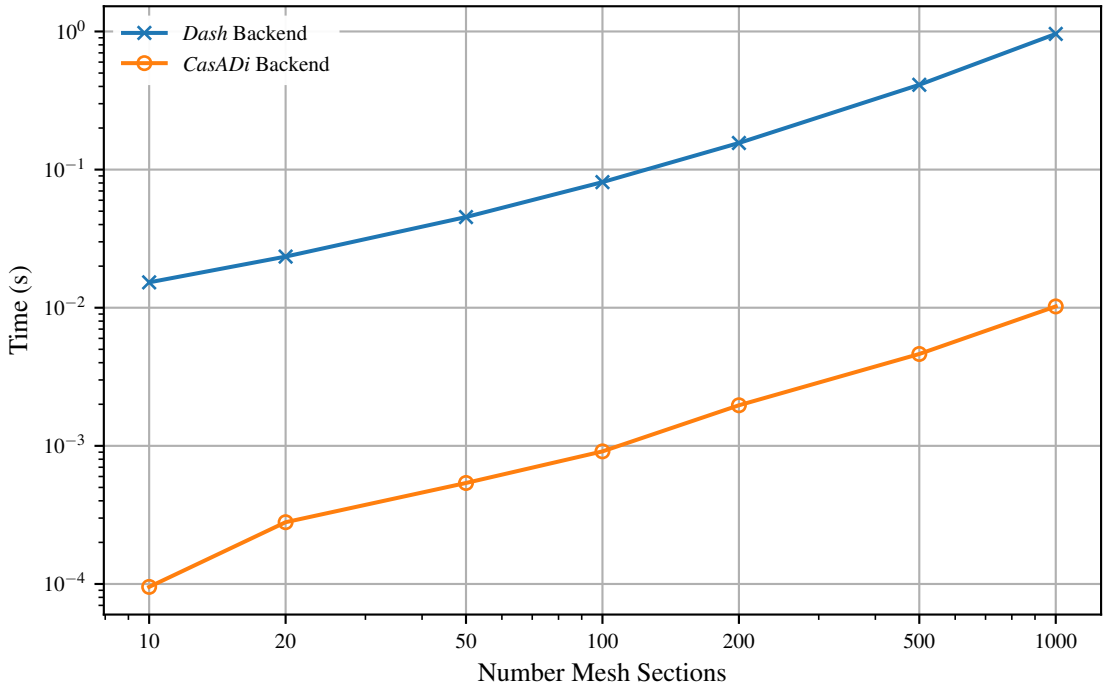
Table 3.9 gives the measured benchmark metrics ($\mathcal{T}$, $\mathcal{P}$ and $\mathcal{I}$) for all values of K using both the *Dash* and *CasADi* backends. Like for the previous two example problems, $\mathcal{T}_{\text{CasADi}}$ was zero for all values of K while $\mathcal{T}_{\text{hSAD}}$ varied by at most 1.06%.

In the case of the tumour anti-angiogenesis problem, $\mathcal{T}_{\text{hSAD}}$ exceeded $\mathcal{P}_{\text{hSAD}}$ for $K \leq 20$, with $\mathcal{P}_{\text{hSAD}}$ exceeding $\mathcal{T}_{\text{hSAD}}$ for $K \geq 50$. As was the case for the two previous example problems, both $\mathcal{P}_{\text{hSAD}}$ and $\mathcal{P}_{\text{CasADi}}$ were found to increase with K. Once again, the ratio of $\mathcal{P}_{\text{CasADi}}$ and $\mathcal{P}_{\text{hSAD}}$ increases with K, from a minimum of $\mathcal{P}_{\text{hSAD}}$ being 6.48 times faster for $K = 10$ to a maximum of $\mathcal{P}_{\text{hSAD}}$ being 487 times faster for $K = 1000$.

As before, a log-log plot of $\mathcal{T}_{\text{hSAD}}+\mathcal{P}_{\text{hSAD}}$ against K is shown in fig. 3.8a. $\mathcal{T}_{\text{hSAD}}+\mathcal{P}_{\text{hSAD}}$ was, again, universally less than $\mathcal{T}_{\text{CasADi}} + \mathcal{P}_{\text{CasADi}}$. For $K = 10$, $\mathcal{T}_{\text{CasADi}} +$



(a) Time taken during preprocessing of derivatives



(b) Average time taken per NLP iteration

Figure 3.8: Comparison of the time taken during numerical solving of the tumour anti-angiogenesis problem by *Pycollo* using its *Dash* and *CasADi* backends.

$\mathcal{P}_{\text{CasADi}}$ was found to be only 2.75 times slower than $\mathcal{T}_{\text{hSAD}} + \mathcal{P}_{\text{hSAD}}$. However, the difference between the total preprocessing times incurred by each backend increased consistently with K and $\mathcal{T}_{\text{CasADi}} + \mathcal{P}_{\text{CasADi}}$ was found to be a maximum of 486 times slower for $K = 1000$.

Finally, increasing K once again increased both $\mathcal{I}_{\text{hSAD}}$ and $\mathcal{I}_{\text{CasADi}}$. $\mathcal{I}_{\text{CasADi}}$ was found to be a maximum of 161 times faster than $\mathcal{I}_{\text{hSAD}}$ ($K = 10$) and a minimum of 79.2 times faster ($K = 200$). Again, there was variability in the ratio between $\mathcal{I}_{\text{hSAD}}$ and $\mathcal{I}_{\text{CasADi}}$. However, $\mathcal{I}_{\text{CasADi}}$ was, on average, 97.1 times faster. Therefore, a reasonable assessment is that, for this problem, $\mathcal{I}_{\text{CasADi}}$ was approximately two orders of magnitude faster than $\mathcal{I}_{\text{hSAD}}$.

3.8 Discussion

The results and analysis of the investigations conducted in section 3.7 will now be discussed. The measured values for $\mathcal{T}_{\text{hSAD}}$ varied by $\pm 0.44\%$, $\pm 0.75\%$ and $\pm 1.06\%$ for the hypersensitive problem (section 3.7.1), space station attitude control problem (section 3.7.2) and tumour anti-angiogenesis problem (section 3.7.3) respectively. Despite the fact that $\mathcal{T}_{\text{hSAD}}$ is theoretically independent of K and should, therefore, be exactly the same across all tests for the same problem, it is to be expected that recorded execution times will differ slightly between successive runs. This is because accurately and precisely benchmarking the execution time of computer code is difficult due to the fact that modern operating systems may, uninitiated by the experimenter, interrupt the execution of a script by interweaving multiple processes on the same CPU core during the benchmarking. While the potential influence of this can be mitigated by attempting to have the benchmarking machine not do anything else while running the investigations, it is very difficult to do this in practice. Furthermore, external factors, such as the room temperature or load history of the CPU in the minutes leading up to the benchmark run, may further influence the recorded times as the operating system throttles the CPU or allows overclocking based on instantaneous thermals. The fact that all of the measured values of $\mathcal{T}_{\text{hSAD}}$ were of the order of 1% or less supports the statement that the benchmarking results arising from this set of investigations are reliable.

As indicated by table 3.8 and fig. 3.7, the space station attitude control problem could not be solved for $K = 1000$ when the *CasADi* backend was being used. This was due to the kernel killing the *Python* process running the investigation during the derivative preprocessing step. The process killing occurred during internal *CasADi* function calls and was likely a result of excessive memory usage. The dynamics

involved in the space station attitude control problem are complicated and a large ET is required to express them in full. By formulating the NLP in full, as is the case in the approach taken by the *CasADi* backend, derivative matrices involving many thousands of variables and constraints are required when $K = 1000$. It is clear from the result of this specific investigation that the AD approach taken by *CasADi* results in excessive memory usage when an attempt is made to populate these derivative matrices. This confirms that taking an approach which does not exploit the predictable structure of the NLP subproblem is not only inefficient but is also unfeasible for dense discretisation meshes.

Across all three test problems, $\mathcal{P}_{\text{hSAD}}$ was consistently found to be at least an order of magnitude faster than $\mathcal{P}_{\text{CasADi}}$. In all cases, both $\mathcal{P}_{\text{hSAD}}$ and $\mathcal{P}_{\text{CasADi}}$ increased as K increased. However, the rate of increase in $\mathcal{P}_{\text{hSAD}}$ with K was found to be significantly less than that for $\mathcal{P}_{\text{CasADi}}$. This meant that for dense meshes, $\mathcal{P}_{\text{CasADi}}$ was typically two orders of magnitude slower than $\mathcal{P}_{\text{hSAD}}$. Furthermore, it is also likely that, judging by the gradient of the line for the *CasADi* backend in fig. 3.7a, $\mathcal{P}_{\text{hSAD}}$ would have been close to three orders of magnitude faster than $\mathcal{P}_{\text{CasADi}}$ in the case of the space station attitude control problem had it been possible to run the test for $K = 1000$. As $\mathcal{P}$ corresponds to the mesh-specific preprocessing, it was expected that $\mathcal{P}_{\text{CasADi}}$ would involve significantly more cost than $\mathcal{P}_{\text{hSAD}}$ due to the fact that the *CasADi* backend has to conduct all of its derivative preprocessing on the full dense NLP subproblem while the *Dash* backend exploits knowledge of its underlying structure. The fact that OCPs will not typically be solved in a single mesh iteration, as illustrated by section 2.7, reinforces the benefits of a smaller $\mathcal{P}$. These results confirm the predicted performance of the *Dash* backend and support its approach. Namely that as much as possible of the derivative preprocessing be conducted in a mesh-independent manner and the known structure of the NLP subproblem be exploited to construct the derivative matrices from the derivatives of the functions defining the OCP.

Using the number of nodes in the expression graph as an approximate measure of problem complexity, of the three test problems, the hypersensitive problem is the least complex. Conversely, the space station attitude control problem is the most complex. Comparing $\mathcal{P}_{\text{hSAD}}$ between the three test problems shows that $\mathcal{P}_{\text{hSAD}}$ increases with problem complexity. $\mathcal{P}_{\text{hSAD}}$ only involves the computation of the mesh-specific indices of the nonzeros in the Jacobian and Hessian. The reason that this trend is observed is due to the method currently utilised by the *Dash* backend to determine these indices. The *Dash* backend does this by calling the callables for the Jacobian and Hessian linked to *Ipopt* with an array populated by floating point NaN values as the argument. This approach is suboptimal and should be replaced by an algorithm to directly construct these indices from the known sparsity pattern of

the NLP subproblem. If this was done then it would be expected that $\mathcal{P}_{\text{hSAD}}$ would increase slightly with increasing K, but would be independent of the complexity of the OCP being solved.

Section 3.7 showed that $\mathcal{I}_{\text{hSAD}}$ was consistently slower than $\mathcal{I}_{\text{CasADi}}$ across all three example problems. While there was variation in the measured ratio between $\mathcal{I}_{\text{hSAD}}$ and $\mathcal{I}_{\text{CasADi}}$ for all problems, there was on average a bigger discrepancy between the cost of the NLP function calls when the OCP was less complex. In the case of the hypersensitive problem, $\mathcal{I}_{\text{hSAD}}$ was on average 129 times slower than $\mathcal{I}_{\text{CasADi}}$. This decreased to 97.1 times for the tumour anti-angiogenesis problem and decreased further to 13.0 times for the space station attitude control problem. hSAD should, theoretically, lead to cheaper numerical evaluation of NLP derivatives than AD for the reasons outlined in section 3.4. However, this assumes a like-for-like implementation. This is not the case for the *CasADi* and *Dash* backends. Therefore, the large differences in $\mathcal{I}_{\text{hSAD}}$ and $\mathcal{I}_{\text{CasADi}}$ measured can be explained by two factors. Firstly, the *CasADi* backend's code writer generates optimised *C* code to numerically evaluate the derivatives. Conversely, the *Dash* backend's code writer generates *Python* functions. It is well known that interpreted *Python* code is typically between 10 and 100 times slower than equivalent compiled *C* code. While the *Dash* backend attempts to improve the execution speed of these through JIT compilation using *Numba* [206], it is not possible to achieve performance on par with compiled code from a low-level language using this approach. This is exaggerated by the fact that *CasADi* is an established package and its code writers have undergone significant optimisations during its many years of development [20]. Secondly, the *CasADi* backend generates the derivative matrices directly. Conversely, the *Dash* backend first evaluates all of the nonzeros using the OCP functions (and their derivatives) and then constructs the NLP derivatives as sparse matrices using a block sparse matrix construction. This added complexity on each NLP function call is a consequence of exploiting the known sparsity of the NLP subproblem. It is also one of the main reasons why $\mathcal{P}_{\text{hSAD}}$ is significantly less than $\mathcal{P}_{\text{CasADi}}$. It is likely that the implementation of the sparse block construction of the NLP matrices in the *Dash* backend could be greatly improved with future work. Considering these things, there is no reason why $\mathcal{I}_{\text{hSAD}}$ could not be reduced to come close to matching $\mathcal{I}_{\text{CasADi}}$ with refinement of the implementation of the *Dash* backend.

3.9 Conclusions

The derivative-taking algorithm hSAD has been developed, specifically to assist with efficient numerical solving of OCPs. hSAD:

- allows a DAG for first-order derivatives to be generated, which can then be transformed into an evaluation trace and compiled to a callable, such that the derivatives can be numerically evaluated;
- enables second-order derivatives to be generated by repeated passes of the hSAD algorithm;
- demonstrates theoretical complexity at least as good as forward-mode AD;
- efficiently generates the first-order derivatives required to solve the NLP subproblem associated with a discretised OCP in an exact analytic form; and
- incorporates a method for efficiently generating the second-order Lagrangian Hessian derivative matrix from other first-order derivatives.

Dash, an additional derivative backend for the open-source general-purpose optimal control software *Pycollo*, was developed, which:

- provides a computational implementation of the hSAD algorithm;
- supports code generation for the numerical evaluation of gradients, Jacobians and Lagrangian Hessians; and
- interfaces directly with the NLP solver *Ipopt*.

The *Dash* backend for *Pycollo* has been benchmarked against the already-existing *CasADi* backend across three varied OCPs from the literature. These investigations found that:

- the *Dash* backend offers very favourable preprocessing cost in comparison to the *CasADi* backend, especially for dense mesh discretisations;
- a practical implementation of the hSAD algorithm leads to very efficient derivative generation when combined with exploitation of the known structure of the NLP subproblem formed when using direct collocation to numerically solve an OCP;
- the numerical evaluation times measured were between one and two orders of magnitude slower when the *Dash* backend was used in comparison to the *CasADi* backend; and
- slow evaluation times using the *Dash* backend can be attributed to differences in the choice of programming language used and inefficiencies due to implementation decisions taken when developing it.

The open-source provision of *Pycollo* and its supplementary *Dash* backend will allow researchers in this field to further develop the hSAD algorithm and its specific application to the numerical solving of OCPs. In particular, it is recommended that future research investigates:

- developing a further mode of hSAD that is analogous to reverse-mode AD (as opposed to forward-mode AD) as this will potentially yield performance benefits when evaluating derivatives for functions with many more outputs than inputs;
- the application of the hSAD concepts of function nodes and tier checkpointing further, to make recommendations on their application and use; and
- optimising and reimplementing *Dash* in a lower-level programming language, such that it is capable of numerically evaluating derivatives with performance equivalent to the *CasADi* backend, while also offering significantly cheaper derivative preprocessing costs.

Chapter 4

Multibody Dynamics

This chapter describes on the development of a software package for the definition and solving of *optimal control problems* (OCPs) involving multibody dynamics, a need not met by the existing software packages in this field. It commences with a review of the related academic literature and a discussion of relevant background material in section 4.1. The third objective from section 1.3 is restated, and a number of sub-objectives required to meet it are then laid out in section 4.2. Section 4.3 details the development and attributes of a new software package designed to meet the stated objectives. A set of investigations using the developed software package were conducted to validate its accuracy and reliability, and to test and evaluate alternative computational approaches. The results from these investigations and their implications are discussed in section 4.5. The chapter concludes by assessing the extent to which the objective and sub-objectives laid out in section 4.2 have been met.

4.1 Background, Theory and Review

In recent years, musculoskeletal modelling has become an increasingly applied methodology in biomechanics research. With the skeleton being a critical part of any vertebrate, a musculoskeletal model requires an underpinning model of the skeletal system. As such, multibody dynamics plays a significant role in biomechanics research.

Methods and software for simulating multibody dynamics were originally developed in the fields of mechanical and aerospace engineering to solve problems within these contexts [165, 170, 189]. Early advancements in these fields were critically important for establishing algorithms by which the *equations of motion* (EoMs)

defining a system could be derived, together with methods for converting these formulations into efficient software for running simulations [292].

Another area where the modelling and simulation of multibody dynamics has been prevalent is in physics engines for computer games. Prominent examples of games physics engines include *PhysX* [254], which is used in the two most well-known game engines, *Unreal Engine* [101] and *Unity* [141]. In addition, *ODE* [305], has been used by biomechanics researchers to investigate the gait of dinosaurs [296]. As well as supporting the modelling and simulation of rigid body dynamics, physics engines such as *PhysX* and *ODE* also support additional functionality such as soft body dynamics and collision detection.

Computer game physics engines have the overarching requirement of providing real-time performance. This is because real-time responses are required to achieve performance suitable for reactive gameplay. As such, computation speed is prioritised, typically at the expense of accuracy, and these tools cannot be used reliably for quantitative engineering [254, 292]. For example, *PhysX* ignores Coriolis forces making it unsuitable for applications where accuracy is of any importance [102].

Furthermore, game physics engines have been designed solely with forward simulation in mind. This means that they construct EoMs to be in a form focused solely on use by *ordinary differential equation* (ODE) solvers [292]. Typically only numerical evaluation of velocity and acceleration quantities is supported. As outlined in chapters 2 and 3, OCPs require first- and second-order derivative information to be solved efficiently and reliably. Therefore, the output of physics engines does not provide the information about dynamical quantities in a form suitable for use in OCPs.

Due to the lack of accuracy, focus on real-time computation and incompatibility with formulating OCPs, research involving multibody dynamics in the field of biomechanics has been developed based on the methodologies and software from aerospace and mechanical engineering.

4.1.1 Deriving Equations of Motion

In order to fully describe how any applied forces will articulate a musculoskeletal model, dynamical EoMs for the system need to be derived. Many methods for deriving the EoMs governing a particular multibody system exist. These methods of classical mechanics include, among others, the Newton-Euler equations, Langrangian mechanics and Kane's method. The dynamics of any system are uniquely described and all derivation methods produce mathematically equivalent EoMs [217]. How-

ever, different derivation methods will produce different analytic expressions for these, in terms of how they are factorised or which terms have been cancelled, with some being more compact and efficient than others [292]. Consequently, the resulting formulation of the EoMs and their complexity will have implications on run-time performance when used as part of any simulation or optimisation [292].

Newton-Euler Equations

The Newton-Euler method, based on Newton's second law, allows the EoMs for a system to be derived by relating the motion of each particle or rigid body in the system, to the sum of all external forces and torques acting upon it. While it is possible to use this method to derive a system of EoMs equal in size to the number of *degrees of freedom* (DoFs), this requires careful selection of the directions in which forces are resolved, which is difficult to do correctly when implementing the method algorithmically in a software implementation [191]. Consequently, this method typically results in larger, more coupled systems of equations as the number of particles increases [57, 74, 191].

Lagrangian Mechanics

Instead of looking at the forces acting on or within a system, Lagrangian mechanics looks at the energies. This approach requires multiple first-order derivatives to be taken. These derivatives include ones with respect to the generalised coordinates, the first-order time derivatives of the generalised coordinates (generalised speeds) and first-order time derivatives. For all but the simplest of examples, analytical expressions for these derivatives are non-trivial to obtain and the expression yielded are not guaranteed to be in simplified forms [190, 191, 292].

Kane's Method

Kane's method differs from the two previously presented methods because where methods based on classical mechanics utilise virtual quantities of variational mechanics, Kane's method instead uses specific known quantities [191]. It was developed in the late twentieth century with the goal of creating a method of dynamics more systematic and physically intuitive than the other methods described previously [191]. As such, in the seminal textbook on the approach [191], it is stated that the method can produce EoMs for a system that can be solved more easily than with Langrangian mechanics.

Kane's method can be derived using Newton's second law and the concept of partial velocities [191, 291]. The result of the derivation gives Kane's Dynamical Equations

$$\tilde{F}_r + \tilde{F}_r^* = 0\,. \tag{4.1}$$

In eq. (4.1), $\tilde{F}_r$ denotes the generalised active forces and is defined as

$$\tilde{F}_r = \sum_{i=1}^{\nu} \tilde{\mathbf{v}}_r^{P_i} \cdot \mathbf{R}_i\,, \quad (i = 1, \ldots, p)\ , \tag{4.2}$$

where $\mathbf{R}_i$ is the resultant of all contact and distance forces acting on P_i, $\tilde{\mathbf{v}}_r^{P_i}$ is the rth nonholonomic partial velocity of P_i and all other nomenclature is as before. Similarly, $\tilde{F}_r^*$ from eq. (4.1) denotes the generalised inertial forces and is defined as

$$\tilde{F}_r^* = \sum_{i=1}^{\nu} \tilde{\mathbf{v}}_r^{P_i} \cdot (-m_i \mathbf{a}_i)\ , \quad (i = 1, \ldots, p) \tag{4.3}$$

where $\mathbf{a}_i$ is the acceleration of P_i in N.

One advantage of Kane's method is that it is systematic and, as such, EoMs can be derived methodically by following a series of standard steps. These steps are:

1. identify points that are locations of centres of mass or locations of applied forces;
2. select a set of generalised coordinates ($\mathfrak{q}_r$) and generalised speeds ($\mathfrak{u}_r$);
3. generate expressions for angular velocity and acceleration of all bodies, and velocity and acceleration of the important points;
4. determine partial velocities for all points with respect to all generalised speeds;
5. construct Kane's Dynamical Equations from the generalised active forces and generalised inertial forces;
6. rearrange Kane's Dynamical Equations into the form $\mathbf{M}\dot{\mathbf{x}} = \mathbf{k}$, where $\dot{\mathbf{x}}$ is the column vector of first-order time derivatives of all of the generalised coordinates and speeds; and
7. (optionally) solve for $\dot{\mathbf{x}}$.

To further explain the method, the EoMs for a classic system, the simple pendulum, are derived. Illustrated in fig. 4.1, the simple pendulum system $\mathcal{S}$ involves a bob with mass m suspended by a light string of length ℓ from point O. The bob hangs freely under gravity, where the acceleration due to gravity is g, and the clockwise angle that the string makes with the downwards vertical is θ. It is also useful

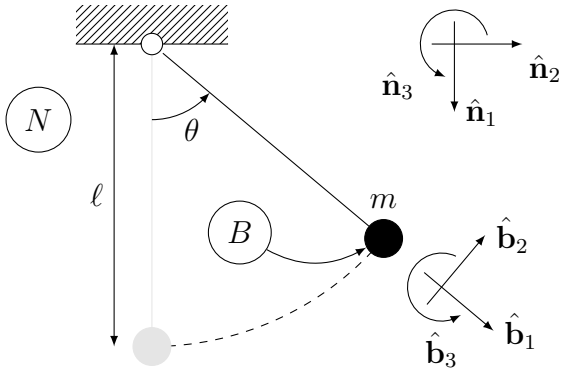


Figure 4.1: Diagram of the simple pendulum system.

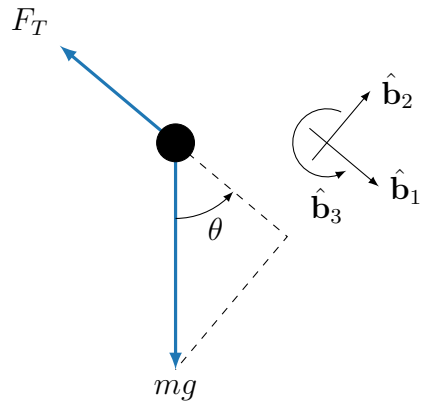


Figure 4.2: Free body diagram of the pendulum bob. Only applied forces are shown.

to define two reference frames, an earth-fixed reference frame N and a bob-fixed reference frame B. Both N and B are defined by a set of mutually orthogonal unit vectors as is illustrated in fig. 4.1.

To derive the EoMs for the simple pendulum system $\mathcal{S}$ using Kane's method, the steps above can be followed sequentially. Note that the tilde notation of eqs. (4.2) and (4.3) can be dropped as $\mathcal{S}$ is a holonomic system [191]. First, it needs to be noted that the simple single DoF system consists of only a single particle with no additional constraints. Therefore, $\nu = p = 1$. Second, a sensible choice for the single generalised coordinate $\mathfrak{q}_1 = \theta$ can be made, along with a sensible choice for the single generalised speed $\mathfrak{u}_1 = \dot{\theta}$. It is common to select the generalised speeds to be the first-order time derivatives of the generalised coordinates. Third, the velocity of the bob can be determined as

$$^N\mathbf{v}^m = \dot{\theta}\ell\hat{\mathbf{b}}_2 \tag{4.4}$$

and the acceleration of the bob can be determined as

$$^N\mathbf{a}^m = \frac{d}{dt}\left(^N\mathbf{v}^m\right) = \frac{d}{dt}\left(\dot{\theta}\ell\hat{\mathbf{b}}_2\right) = \ddot{\theta}\ell\hat{\mathbf{b}}_2 + \dot{\theta}^2\ell\dot{\hat{\mathbf{b}}}_2 = \ddot{\theta}\ell\hat{\mathbf{b}}_2 - \dot{\theta}^2\ell\hat{\mathbf{b}}_1\,. \tag{4.5}$$

Fourth, the partial velocity of the bob with respect to θ can easily be determined

from the expression for the velocity of the bob as

$$\mathbf{v}_1^m = \ell\hat{\mathbf{b}}_2\,. \tag{4.6}$$

Fifth, before the generalised active force F_1 can be calculated, the resultant force on the bob $\mathbf{R}_1$ needs to be found. By examining the *free body diagram* (FBD) shown in fig. 4.2

$$\mathbf{R}_1 = mg\hat{\mathbf{n}}_1 - F_T\hat{\mathbf{a}}_1 = mg\left(\cos\left(\theta\right)\hat{\mathbf{a}}_1 - \sin\left(\theta\right)\hat{\mathbf{a}}_2\right) - F_T\hat{\mathbf{a}}_1 \tag{4.7}$$

and so, using eq. (4.2),

$$\begin{aligned} F_1 &= \mathbf{v}_1^m \cdot \mathbf{R}_1 \\ &= \left(\ell\hat{\mathbf{b}}_2\right) \cdot \left(mg\left(\cos\left(\theta\right)\hat{\mathbf{b}}_1 - \sin\left(\theta\right)\hat{\mathbf{b}}_2\right) - F_T\hat{\mathbf{b}}_1\right) \\ &= mg\ell\left(\cos\left(\theta\right)\hat{\mathbf{b}}_2 \cdot \hat{\mathbf{b}}_1 - \sin\left(\theta\right)\hat{\mathbf{b}}_2 \cdot \hat{\mathbf{b}}_2\right) - F_T\ell\hat{\mathbf{b}}_2 \cdot \hat{\mathbf{b}}_1 \\ &= -mg\ell\sin\left(\theta\right), \end{aligned} \tag{4.8}$$

noting that the dot product between any pair of orthogonal vectors is 0 and the dot product of any unit vector with itself is 1. Similarly, using eq. (4.3)

$$\begin{aligned} F_1^* &= \mathbf{v}_1^m \cdot \left(-m^N\mathbf{a}^m\right) \\ &= \left(\ell\hat{\mathbf{b}}_2\right) \cdot \left(-m\left(\ddot{\theta}\ell\hat{\mathbf{b}}_2 - \dot{\theta}^2\ell\hat{\mathbf{b}}_1\right)\right) \\ &= -m\ell^2\ddot{\theta}\hat{\mathbf{b}}_2 \cdot \hat{\mathbf{b}}_2 + m\ell^2\dot{\theta}^2\hat{\mathbf{b}}_2 \cdot \hat{\mathbf{b}}_1 \\ &= -m\ell^2\ddot{\theta}\,. \end{aligned} \tag{4.9}$$

Kane's Dynamical Equations can then be constructed by combining eqs. (4.8) and (4.9) to give

$$F_1 + F_1^* = -mg\ell\sin\left(\theta\right) - m\ell^2\ddot{\theta} = 0\,. \tag{4.10}$$

Finally, combining the sixth step and optional final step, rearranging eq. (4.10) yields the EoM

$$\ddot{\theta} = -\frac{g}{\ell}\sin\left(\theta\right). \tag{4.11}$$

While this example may appear to make Kane's method appear laborious, it is worth noting that the approach is the same for all systems, no matter their size or complexity.

Discussion of Methods

There are numerous examples in the literature showing that hand-crafted software implementations of EoMs are the most performant [56, 57, 190]. Hand-crafted

implementations require manual derivation of the EoMs, allowing significant rearrangement and simplification of the equations, as well as manual optimisation of any computer code [190]. However, derivation by hand is time consuming and error prone for all but the simplest of systems [259]. Furthermore, such an approach is system-specific so would need to be completely redone if the system was changed even slightly. This approach is, therefore, seldom practical. As such an algorithmic approach that can be automated in software is required for both general multi-body modelling software [292] and biomechanical modelling software [259].

All three of the approaches presented above have the potential to be converted into systematic software implementations. While all three methods will generate mathematically equivalent EoMs for a system, the formulations will not necessarily be equivalently efficient [292]. Efficiency here can refer to both the size of the resulting EoMs as well as the susceptibility of the formulation to be translated into performant software. Dynamical calculations typically account for a significantly large proportion of overall computation time during multibody simulations [292]. This is especially true within the context of an OCP because this can result in millions of consecutive evaluations of the EoMs [74]. Compact EoMs are, therefore, critically important for ensuring that they can be numerically evaluated efficiently as part of any software which they are associated with.

The method based on the Newton-Euler equations generally leads to exceedingly compact expressions [292]. However, as the approach requires treating each particle in the system as decoupled, with any internal or constraint forces replaced by explicitly named unknowns, the resulting number of equations can far exceed the number of required generalised coordinates [74]. This is because in addition to the generalised coordinates, the resulting system of equations will also contain all of the symbolic internal and constraint forces as unknowns. As a result, the additional expression manipulation required to eliminate the excess unknowns can result in extensive expression bloat [292].

A systematic method for deriving EoMs of biomechanical systems based on the Newton-Euler equations has been proposed [74]. In this approach, the Newton-Euler force and torque equations for each segment in a musculoskeletal model are derived. These are then used to solve the system's acceleration states using matrix manipulations. While this method does present a simple and logical way of procedurally deriving the EoMs for any *two-dimensional* (2D) system, the final matrix representation of the EoMs also includes all internal and external reaction forces. This results in a coupled linear system of equations that is larger than required to describe the system. This approach is, therefore, not suitable for use as a methodology in a general-purpose package. Furthermore, the approach is not applicable to *three-*

dimensional (3D) systems making it further unsuitable for modern biomechanical analysis [74].

Lagrangian mechanics and Kane's method contrast approaches based on the Newton-Euler equations in that exactly one expression is yielded per generalised coordinate. This results in an as-compact-as-possible linear system of equations defining the generalised coordinates and speeds. Unlike Kane's method, Lagrangian mechanics requires a substantial number of scalar energy quantities to be differentiated as part of the derivation process. This results in potentially bloated expressions containing complicated trigonometric functions [292]. This has also been stated to be the case in biomechanical applications [74]. While it is possible to simplify the expressions of Lagrangian mechanics, it is not easy to do this systematically. This is primarily because, as stated in section 3.1, it is difficult to ascertain when a system of equations has been simplified sufficiently. Kane's method, on the other hand, has been shown to lead directly to the simplest possible EoMs [190].

Kane's method yields one equation per generalised coordinate. However, if internal or constraint forces are explicitly required, then Kane's method can also easily produce these by using auxiliary speeds [191] to yield the required additional equations [291]. Kane's method, therefore, offers the ability to easily control which unknown quantities are present in the resulting EoMs. As such, this method is well suited for use as the basis of algorithmically generating EoMs [189, 190, 292]. For all of the reasons mentioned above, Kane's method is also advantageous when applied to deriving EoMs for musculoskeletal models [299, 331].

4.1.2 Explicit and Implicit Formulations

No matter which method is used to derive the EoMs for a system, a system of linear equations in the form $\mathbf{M}\dot{\mathbf{x}} = \mathbf{k}$ will be yielded. In this system, $\mathbf{M}$ is the mass matrix, $\mathbf{k}$ is the forcing vector and $\dot{\mathbf{x}}$ is the column vector of first-order time derivatives of the generalised coordinates and speeds. Typically, $\dot{\mathbf{x}}$ would need to be found by inverting $\mathbf{M}$ to transform the linear system into $\dot{\mathbf{x}} = \mathbf{M}^{-1}\mathbf{k}$ so that $\dot{\mathbf{x}}$ can be directly used either as a *differential-algebraic system of equations* (DAE) in an *initial value problem* (IVP) or as a state equation in an OCP. If the system is transformed so that $\dot{\mathbf{x}}$ has been found, then the system can be termed *explicit*. If, on the other hand, the EoMs remain in the form $\mathbf{M}\dot{\mathbf{x}} = \mathbf{k}$, then the system of equations are coupled in the highest-order derivative and, as such, are termed *implicit*.

The simplest and most intuitive approach to formulating a set of EoMs for use in an OCP is to explicitly solve for the first-order time derivatives of the generalised

coordinates and speeds. As the generalised speeds are typically chosen to be the first-order time derivatives of the generalised coordinates, only the generalised accelerations need to be explicitly found. This is the approach taken by the majority of multibody software packages [246, 299]. Obtaining $\dot{\mathbf{x}}$ explicitly is conceptually advantageous as it means that state equations for the generalised speeds can be directly supplied to the OCP software unaltered. One potential major downside of this approach, however, is that by explicitly inverting the mass matrix, the resulting expressions for $\dot{\mathbf{x}}$ can become exceedingly large and complex. This will have implications on the computational performance when solving the OCP as these state equations and their derivatives will be computationally expensive to numerically evaluate. An additional downside is that the mass matrix can be near-singular for biomechanical models due to the small masses and inertias of certain bodies in the system [57]. Therefore, inverting the mass matrix can result in very stiff equations for $\dot{\mathbf{x}}$ with large eigenvalues [57].

To avoid the numerical complications associated with a near-singular mass matrix, an implicit formulation is required as this completely avoids the requirement of computing $\mathbf{M}^{-1}$ [57]. An implicit formulation is usually produced by introducing a set of auxiliary controls, equal in size to the number of generalised speeds [57, 108]. The generalised speeds' state equations are directly mapped to the auxiliary controls while the EoMs are enforced by introducing them as algebraic constraints. For an OCP being solved using a direct collocation method, these algebraic constraints correspond to introducing n path constraints per generalised speed, where n is the number of discretisation nodes in the temporal mesh of the *nonlinear programming problem* (NLP). Therefore, an implicit formulation has the effect of significantly increasing the size of the NLP subproblem, both in decision variables and in constraints. This is a drawback as larger NLPs tend to be more difficult to solve [131]. Furthermore, the introduction of additional path constraints is not ideal as these increase the complexity of the NLP and its difficulty to solve, and should be avoided if possible [36].

Both implicit and explicit EoMs have been proposed as being more suitable for musculoskeletal simulations. It has been argued that an implicit formulation produces sparser matrices which allows more efficient matrix derivative calculations and results in a more well conditioned numerical problem than a comparative explicit formulation [57, 81]. There are many examples in which implicit formulations have been used [57, 85, 91, 108]. However, others have reported success in producing efficient and well-conditioned simulations and OCPs using explicit formulations [212]. There is yet to be a direct comparison of the performance of explicit and implicit formulations for the same problem.

4.1.3 Assumptions in Biomechanical Multibody Dynamics

Rigid Body Mechanisms

The skeletal system is generally modelled as a mechanism of linked rigid members [53, 54, 74, 251, 274]. Segments are assumed to be rigid as this allows the use of rigid body dynamics in the derivation of EoMs [74]. This assumption is valid if the modelled movement tasks do not involve impacts, which have been shown to cause non-rigid behaviour [93, 137, 221].

To incorporate modelling of large-scale soft tissue movement relative to the bone, wobbling mass models have been used [137]. This is typically done by attaching a second rigid element to a rigid body via a nonlinear damped passive spring [258, 338]. Inclusion of wobbling masses can be crucial for accurate modelling and has been shown to reduce loading on the system by nearly 50% compared to an equivalent rigid body model [258]. However, such models increase the number of bodies present in a model and thus increase system and simulation complexity [338].

Anthropometric information describing any modelled segment (including lengths, locations of centres of mass, masses, and moments of inertia) is required. Careful consideration must be given to how these values are parameterised as they can have a large influence on movements generated by simulations [338]. Parameters can either be determined by estimating values based on the literature, or by measuring a specific individual. For the former, regression equations based on measurements of cadaver segments can be used [164, 341], although this approach is only valid if the morphologies of the participant being modelling and the original cadaver are similar [338].

Alternatively, a geometric model with segment densities derived from cadaver measurement can be used [153, 334]. Using this approach, error values of approximately 2% have been reported [334]. Measurement of live subjects using techniques such as *computed tomography* (CT) and *magnetic resonance imaging* (MRI) is also possible [73], making accurate, subject-specific evaluation of these quantities viable.

Mechanism Complexity

Many multi-joint movement studies have assumed that the motion of the skeletal system mechanism is constrained to the sagittal plane. This assumption is valid for movement tasks where performance is almost entirely attributed to extension and flexion of the lower limb joints, and where joint centres remain coincident [234,

338]. Planar hip and knee extension and flexion dominate the power supplied to the cranks during maximal pedalling [99, 120, 228, 233]. Similarly, in jumping tasks these exertions are almost entirely responsible for accelerating the body's centre of mass during take off [15, 54, 261, 295, 308].

Planar assumptions are not valid in cases that involve non-sagittal movement where projections of the hip and shoulder joints become non-coincident [234]. Examples include movements with side-on positions, such as javelin throwing, fast bowling in cricket, and overhead racket strokes. Imposing constraints on the relative locations of joints has been used to create a planar model of fast bowling in cricket that accurately recreates system dynamics and forces [112]. However, such an approach limits the creation of pure predictive simulations as movement data from real athletes is used in part to define the resulting movement.

3D modelling for multi-joint tasks has become more prominent in recent years due to increases in computational power making these studies more feasible. For balance, stance, and posture tasks, 3D modelling is more appropriate. These tasks require fine motor control of pelvic tilt and rotation in addition to hip extension and flexion which is only allowed if the hip is modelled as a three DoF ball-and-socket joint [19, 222]. While balance, stance, and posture have also been investigated using 2D models, limited insight into the fine motor controls required was gained [156, 239].

A general-purpose multibody modelling package should provide the ability to generate governing EoMs for 3D systems due to the requirement for this in certain applications. However, as 2D assumptions are prevalent and valid in many cases, such a software package should be able to create a 2D model in 3D that is as efficient as if the calculations had been conducted entirely in 2D. This is so that this simplifying assumption can be used, as intended, to reduce computational complexity without incurring any additional overhead due to such a package's 3D capabilities.

Joint Models

For simplicity, joints between the rigid bodies used to model the segments of the skeletal system are often assumed to be pin joints [53, 54, 74, 251, 274]. The hip, knee, and ankle joints have all frequently been modelled as pin joints because this reduces the number of DoFs in the system and enables very efficient computation [259]. More complex models of the lower body joints have been developed to describe, for example, the modified hinge joint nature of the knee joint [127, 332]. Such models of the knee joint have been implemented in numerous biomechanical models that have

been used to simulate pedalling [53, 274, 279, 307]. Other highly complex 2D and 3D, anatomically accurate models of the lower limb joints have been developed [19, 222]. However, these have seen limited application in biomechanical models because they contain many parameters relating to their geometry that cannot be accurately determined [210]. Modelling the joints of the lower leg as pins is a valid assumption for performance tasks where additional complexity increases uncertainty without sufficiently improving the accuracy of results [13].

Contact Mechanics

Modelling multibody systems often demands the modelling of contact between bodies. These contacts can be between any components in the system, either the internal forces between a multibody mechanism or the surface contact between two or more bodies. In reality, contact forces arise due to the deformation of the two contacting bodies. This is particularly true in biomechanical systems where such contacting bodies may be compliant biomaterials which noticeably deform.

Contacts, especially those between the bodies of a multibody mechanism, are often modelled as rigid contacts to reduce the complexity of the resulting system [259]. That is, such contacts may be idealised as a pin joint. When compliant contact needs to be accounted for, contact modelling based on either Hertz contact theory [157, 185] or the Elastic Foundation Model [47, 185] can be used. Hertz contact theory assumes that the contact is between two linearly elastic solids. In the Elastic foundation model, the contacting bodies are assumed to be rigid apart from a thin surface layer of elastic material.

Deformable contact modelling is typically only used within a biomechanical context to model foot-ground contacts [85, 87, 145, 299]. For internal contacts between bodies in a biomechanical model, rigid contacts are almost universally assumed [299]. As not all biomechanical models contain deformable contacts, this functionality should be considered as of secondary importance during the development of any new multibody dynamics software.

4.1.4 Multibody Dynamics Software

Software packages designed and developed within the field of mechanical and aerospace engineering have historically been essential in biomechanical research [88, 259, 346]. Examples of commercial packages include *AUTOLEV* [214], *SD/FAST* [169], *ADAMS* [293] and *DADS* [304]. All of these packages focus on forward simulation and, as such,

none natively support the formulation or solving of multibody OCPs.

Using commercial software packages can be problematic in research applications as it reduces the transparency and flexibility when developing new techniques. This is of particular importance in a biomechanical context as the nature of skeletal modelling differs to that of a purely mechanical system. For example, biomechanical joints may consist of multiple parts and may not perform simple rotations about a fixed axis. Similarly, soft biomaterial may deform during actuation meaning that moment arms vary and locations of force application are not single points. In addition, parameterisation of models is difficult as these values may not be possible to measure directly and may differ substantially depending on operating conditions [259, 299].

These factors have resulted in the development of the open-source biomechanics-specific *C++* multibody dynamics package, *Simbody*. The primary focus of *Simbody* is to allow users to build multibody models and formulate their EoMs easily. *Simbody* does this by providing an *object-oriented programming* (OOP) library of model components and utilises Kane's method for EoM derivation. Previous discussion has explained why Kane's method is a good choice for such a package. An OOP library of model components is user-friendly because it provides a simple way for users to reliably construct a multibody model and derive its EoMs, without requiring explicit dynamics domain knowledge. Due to these attributes, *Simbody* has seen widespread use within the academic biomechanics community as it is the dynamics engine underpinning the open-source biomechanical modelling software, *OpenSim* [87] (see section 5.1).

In addition to enabling the derivation of EoMs, *Simbody* also supports contact modelling, numerical integration and handling of real-time interactions. *Simbody* focuses on high accuracy and performant simulation that is able to handle real-time interactions. As such, it treats all simulations as IVPs, which it solves numerically. However, as a consequence of these design decisions, *Simbody*, without modification, is ill-suited for direct application to OCPs because it does not provide dynamical equations in a form where they can easily be used to yield OCP derivatives [108]. *Simbody* has been extended, by *Moco* [91], to provide OCP functionality. However, for reasons further outlined in section 1.1, this approach is limited. This is due to *Simbody*'s focus on numerical computation meaning that only *finite differencing* (FD) can be used for OCP derivative generation, rather than *algorithmic differentiation* (AD) or *hybrid-symbolic-algorithmic differentiation* (hSAD) (section 3.3), which limits its performance [91, 108].

The only established open-source multibody dynamics software specifically de-

signed with OCPs in mind is the *SymPy* [240], *PyDy* [246] and *opty* [247] stack. *SymPy*, through its classical mechanics module, allows symbolic EoMs to be generated for a multibody system using either Lagrangian mechanics or Kane's method. In order to generate EoMs using Kane's method, it requires the user to describe the modelled system using five sets of equations: holonomic constraints, non-holonomic constraints, kinematic differential equations, dynamic equations, and differentiated non-holonomic equations. *SymPy* offers a small library of low-level of abstraction OOP components with which a user can construct a multibody model. These components include, among others, dynamics symbols for describing position and velocity coordinates, particles and inertia dyadics. For example, defining even a basic model, such as one of the simple pendulum, requires the user to:

1. select a set of generalised coordinates and a set of generalised speeds;
2. create a global reference frame and origin;
3. explicitly define the locations of all points, masses and bodies relative to the origin;
4. map the generalised speeds to the time derivatives of the generalised coordinates (or similar, depending on the choice of generalised coordinates and speeds);
5. manually set the velocities of all important points in the system;
6. collect together the five required equations that describe the system; and
7. call a function to compute the symbolic EoMs using Kane's method.

SymPy is, therefore, a dynamics toolbox and not a high-level of abstraction OOP component library. As such, the package is not suitable for use by users without specific knowledge of multibody dynamics. Furthermore, even if the user has good knowledge of multibody dynamics, because of the steps required as outlined above, constructing EoMs using this package is laborious.

PyDy extends *SymPy* and supports numerical simulation of any system for which symbolic EoMs have been generated. It does this by compiling callables to numerically evaluate the EoMs. It then interfaces with *SciPy* for the numerical integration, using one of its ODE solvers to solve an IVP. The basic OCP software package, *opty*, interfaces with both *SymPy* and *PyDy*, allowing problems with symbolic EoMs to be solved. As outlined in section 2.1, *opty* is limited as a general-purpose OCP software package in that it only implements very simple collocation schemes (e.g. backward Euler and midpoint) and therefore is limited in accuracy

Packages	Features	Limitations	Source
AUTOLEV	High level of abstraction modelling, efficient code generation	No native OCP functionality	Closed
SD/FAST	High level of abstraction modelling, efficient code generation	No native OCP functionality	Closed
ADAMS	High level of abstraction modelling, efficient code generation	No native OCP functionality	Closed
DADS	High level of abstraction modelling, efficient code generation	No native OCP functionality	Closed
Simbody, *Moco*	High level of abstraction modelling, efficient code generation	OCP functionality restricted by *Simbody* design decisions and architecture, OCP derivatives by FD	Open
Sympy, *PyDy*, *opty*	Designed specifically for multibody OCPs, exact OCP derivatives	Low level of abstraction modelling, OCP derivatives by *symbolic differentiation* (SD), low accuracy collocation methods	Open

Table 4.1: Features and limitations of the available software packages for solving multibody OCPs.

compared to other similar software packages. Furthermore, *opty* does not provide any form of mesh error calculation or algorithmic mesh refinement, further limiting its accuracy.

Table 4.1 provides a summary of the available software packages for solving multibody OCPs, and their features and limitations. There is currently no open-source, high-level of abstraction multibody dynamics software package that has been specifically designed for use as part of an OCP software stack.

4.2 Research Objectives

Section 1.3 laid out the objective of developing and critically evaluating a highly performant, easy-to-use, open-source software package for modelling multibody systems and their dynamics, specifically tailored for use in OCPs. This software package should form a core element of the *Biomechanics Predictive Simulation Toolkit* (BPST).

From the analysis and review of past work in section 4.1, a number of limitations and constraints associated with the current software provision in this area were identified. To address these, and meet the overall objective above, the following

sub-objectives are laid out:

- enable users to efficiently construct multibody models using a high-level of abstraction (current OCP-focused multibody modelling software only contains low-level of abstraction modelling components);

- enable users without an expertise in dynamics to derive the EoMs governing their modelled multibody system (current OCP-focused multibody modelling software requires users to manually assemble dynamical equations describing a modelled system);

- enable users to construct and accurately solve OCPs involving a modelled multibody system while seeking to minimise complexity for the user (current OCP-focused multibody modelling software is inaccurate and requires users to manually formulate many aspects of the OCP definition);

- support the formulation of OCPs using either explicit or implicit dynamics (there is no clear concensus in the literature about whether explicit or implicit dynamics is preferable for use when formulating and solving multibody OCPs);

- directly compare the performance of OCPs using explicit dynamics to ones using implicit dynamics (the academic literature contains no examples of direct comparisons between the same OCP solved using explicit and implicit dynamics); and

- validate the developed software using established problems from the multibody dynamics literature.

4.3 Software Implementation: *Pynamics*

In this section, the development of a multibody dynamics package is described. The package is called *Pynamics*, with its name derived from *Python* and dynamics. *Pynamics* has been developed as an open-source software package and uses *Python* as its development language for the same reasons as outlined in section 2.6. In addition to this, *Pynamics* was designed to be an open-source package written in *Python* so that it could be built on top of *Pycollo* to provide native support for OCP solving.

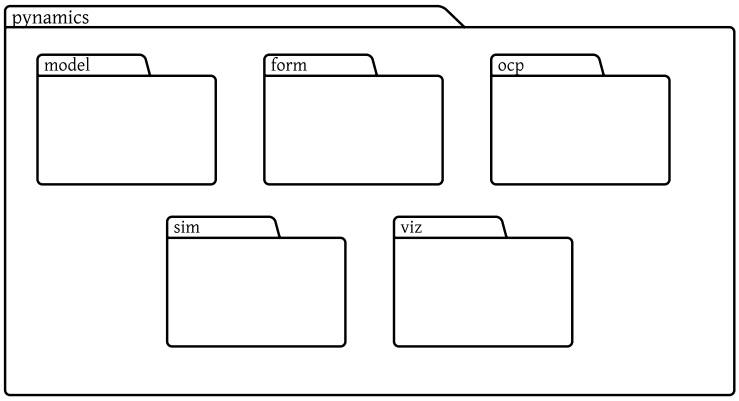


Figure 4.3: Module structure of *Pynamics*.

4.3.1 Overview

Pynamics is a general-purpose multibody modelling and dynamics package. It has been designed specifically to enable users to formulate robust, high performance, minimal-coordinate OCPs involving multibody dynamics. As the scope of *Pynamics* covers both multibody dynamics and optimal control, the package aims to be accessible to users who do not have in-depth domain-specific knowledge in either of these fields. As such, a user base from a diverse range of fields wishing to solve multibody dynamics-related OCPs is expected.

Architecture

Like *Pycollo*, *Pynamics* uses an OOP architecture. This makes access easy for a wide range of users of the package as OOP is conceptually intuitive due to the tangible nature of objects. At the top level, *Pynamics* consists of five main modules. These modules are `model`, `form`, `ocp`, `sim` and `viz`. It should be noted that the separation between these modules is for package structuring purposes only and, as such, their existence is not directly apparent to the user. That is to say, all classes and functions available as part of the *Pynamics application programming interface* (API) are accessible directly via the `pynamics` namespace. As with *Pycollo*, American English spellings are used throughout the *Pynamics* codebase because this is the standard used for the vast majority of scientific *Python* packages.

The `model` module can be thought of as a library of modelling components that can be used to construct a multibody model. This includes, amongst others, objects such as bodies, joints, interactions and constraints. Section 4.3.2 details the library of model objects available from the `model` module.

Multibody dynamics involves both the modelling of a system and the investigation of its behaviour. Whether this is by means of forward simulation or as an OCP, the EoMs governing the system need to be derived. The `form` module, detailed in section 4.3.3, is responsible for analysing the modelled system and deriving its EoMs.

The main focus of *Pynamics* is to assist users in developing multibody models that can be used within OCP problems. The `ocp` module formulates OCPs based on a multibody model defined using the `model` module. It generates all of the components required to correctly formulate an OCP as well as providing a direct interface to *Pycollo*. The structure, functionality and operation of the `ocp` module is detailed in section 4.3.4

While the primary focus of *Pynamics* is on OCPs, it also enables users to conduct forward simulations via the `sim` module. The `sim` and `ocp` modules are similar in many regards and share some processing, analysis and formulation steps. The `sim` module interfaces with the *SciPy*'s integration module and is outlined in section 4.3.5.

To facilitate ease of use, *Pynamics* also provides visualisation functionality via the `viz` module. The `viz` module supports basic plotting and animation capabilities visualising the results of forward simulations or OCPs. Section 4.3.6 briefly discusses the contents and functionality of the `viz` module.

Integration with *SymPy* Classical Mechanics

Pynamics utilises the classical mechanics module of *SymPy* as the basis for certain multibody dynamics functionality. *SymPy* provides objects representing vectors, points, reference frames and inertia dyadics. It is also capable of generating the EoMs for a system via either Lagrangian mechanics or Kane's method. When discussing objects from *SymPy*'s classical mechanics module, the namespace shorthand convention `me` will be used. For all other *SymPy* objects (provided that they are members of the package's main namespace), the namespace convention `sym` will be used instead.

To generate EoMs using Kane's method, *SymPy* provides the `me.KanesMethod` class. Creating an instance of the `me.KanesMethod` class allows the mass matrix and forcing vector for a system to be computed simply by calling its `kanes_equations` method. In order to instantiate a `me.KanesMethod` object, the user must supply as arguments:

1. a `me.ReferenceFrame`, in which the rest of the system is defined;
2. a set of independent generalised coordinates;
3. a set of independent generalised speeds;
4. a set of kinematic differential equations relating the generalised coordinates to the generalised speeds;
5. a set of dependent generalised coordinates, if any;
6. a set of dependent generalised speeds, if any;
7. a set of configuration constraints, if any;
8. a set of velocity constraints, if any;
9. a set of acceleration constraints, if any; and
10. a set of auxiliary speeds, if any.

This list of requirements is lengthy and constructing it can be complicated and error-prone, even for a familiar user. Therefore, while *Pynamics* uses and builds on this functionality, it abstracts away these complexities, providing a more user-friendly and higher-level API.

4.3.2 Object Library (`model` Module)

The `model` module of *Pynamics* can be thought of as a library of components with which a user can construct a model of a multibody system. All uses of *Pynamics* begin with the user creating an instance of the `Model` class. Note that when a named object is referenced in the following explanation, it can be assumed to be part of the *Pynamics* namespace unless stated explicitly otherwise. The `Model` class is the basis for a multibody dynamics model and as such contains important components that all systems require, such as an origin and global reference frame. The `Model` class acts as a container for all of the other components that constitute the multibody system being constructed.

To model a complete multibody system, components such as rigid bodies, pin joints and forces need to be added. To support this, *Pynamics* provides an extensive library of classes describing different model components. Each of these classes inherits from an *abstract base class* (ABC), `ModelObject`, which defines useful attributes, properties and methods common across all model components. These

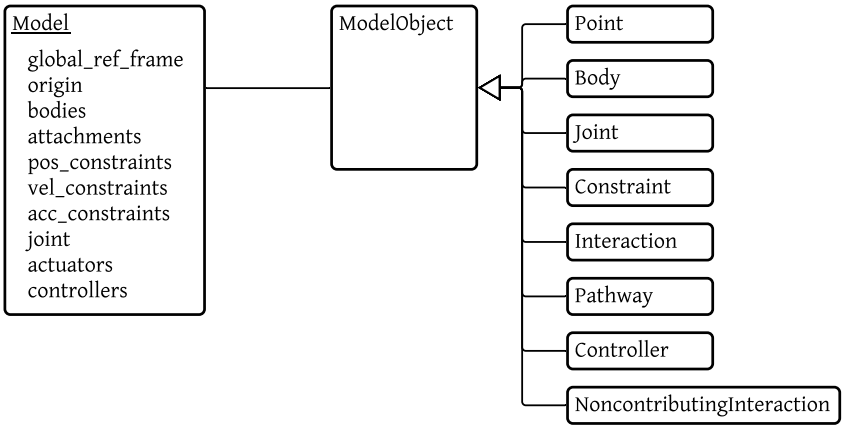


Figure 4.4: Categories of model component available in *Pynamics*.

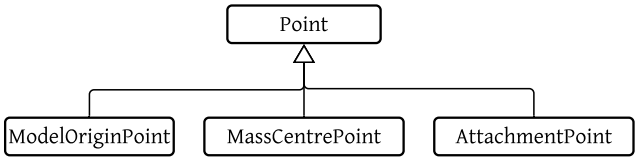


Figure 4.5: Class inheritance diagram for `Point` classes in *Pynamics*.

include names, identifiers, the parent `Model` and an accessor to the global reference frame. The general categories of model components, which are also shown in fig. 4.4, are `Point`, `Body`, `Joint`, `Constraint`, `Interaction`, `Pathway`, `Controller` and `NoncontributingInteraction`.

Points

`Point` objects in *Pynamics* are responsible for defining where and how objects in the `Model` connect to one another. *Pynamics* contains three main object types that inherit from `Point`, shown in fig. 4.5. These are `ModelOriginPoint`, `MassCenterPoint` and `AttachmentPoint`.

`ModelOriginPoint` defines the location of the system's origin, relative to which all positions are defined. It is a singleton and is instantiated automatically when a `Model` object is created. Similarly, `MassCenterPoint` objects, which define the location of the centre of mass of a body, are owned by each `Body` object and are automatically instantiated during their initialisation. An `AttachmentPoint` object is used to define an important point on a body, such as the location at which a joint is attached or the point of action of a force. They are, therefore, used extensively in the construction of any model within *Pynamics*. In contrast, `ModelOriginPoint`

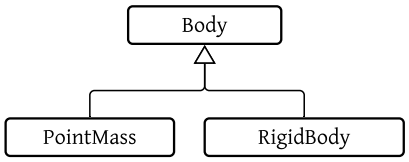


Figure 4.6: Class inheritance diagram for `Body` classes in *Pynamics*.

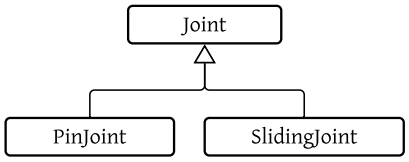


Figure 4.7: Class inheritance diagram for `Joint` classes in *Pynamics*.

and `MassCenterPoint` are not accessible as part of the *Pynamics* namespace and so cannot be directly instantiated by the user.

Bodies

Only point masses and rigid bodies are modelled within *Pynamics* so that rigid body dynamics and Kane's method can be used to derive EoMs. These are represented by the `PointMass` and `RigidBody` classes respectively, which subclasses `Body` (fig. 4.6). The `PointMass` and `RigidBody` classes are similar in all regards except that the former does not have an inertia attribute. A `Body` object instantiates a `MassCenterPoint` during its initialisation which ensures that the mass and inertia (if applicable) properties of the body are correctly parameterised. It also instantiates a `me.ReferenceFrame` during its initialisation that is specific to the body. This additional body-fixed reference frame is designed specifically to simplify the process of defining how other objects attach relative to the `Body`.

Joints

`Joint` objects in *Pynamics* allow `Body` objects to be connected. To attach a `Joint` object into a `Model`, it is typically required that the `Joint` is parameterised with two `AttachmentPoint` objects on two `Body` objects defining the parent and child bodies of the joint. `Joint` objects are responsible for introducing generalised coordinates and speeds, the quantity of which is governed by the number of DoFs that the joint allows. The two main joint classes are `PinJoint` and `SlidingJoint` (fig. 4.7). Both offer a single DoF, with this being a rotational DoF about a specified axis for the

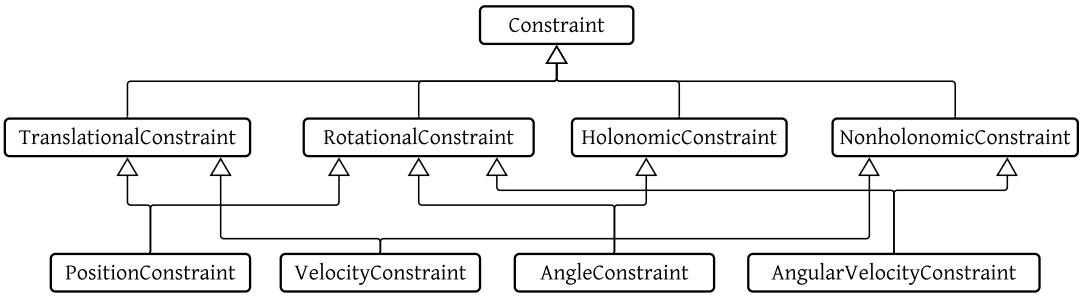


Figure 4.8: Class inheritance diagram for `Constraint` classes in *Pynamics*.

former and a translational DoF along a specified axis for the latter. It was decided to not include further joint types as more complex joints tend to be system-specific when they arise. Such joints can either be implemented by the user by subclassing the `Joint` class with a custom class or by using the required set of constraints between two bodies. There is scope to increase the library of `Joint` subclasses in future releases if required.

Constraints

Pynamics contains classes to enable the placing of constraints on a system. These constraints can be both translational and rotational, as well as both holonomic and nonholonomic, and are available as the `PositionConstraint`, `VelocityConstraint`, `AngleConstraint` and `AngularVelocityConstraint` classes. These classes are implemented using multiple inheritance, as detailed in fig. 4.8. For example, an `AngleConstraint` object could be used to mimic a clamped joint and a `AngularVelocityConstraint` object could be used to enforce an isokinetic condition of a rigid body about a specified axis.

Interactions

`Interaction` objects allow forces and torques to be applied to a model. These can be point interactions, acting at a single `AttachmentPoint`, which are implemented using the `PointForce` and `PointTorque` classes (fig. 4.9). Alternatively, these can be actuators acting between a pair of attachment points. *Pynamics* has been designed to offer many different types of actuator, including springs, dampers and linear actuators, a selection of which are also shown in fig. 4.9. All `Interaction` objects work directly in conjunction with a `Controller` object, which enables the magnitude and timing of force or torque application to be prescribed.

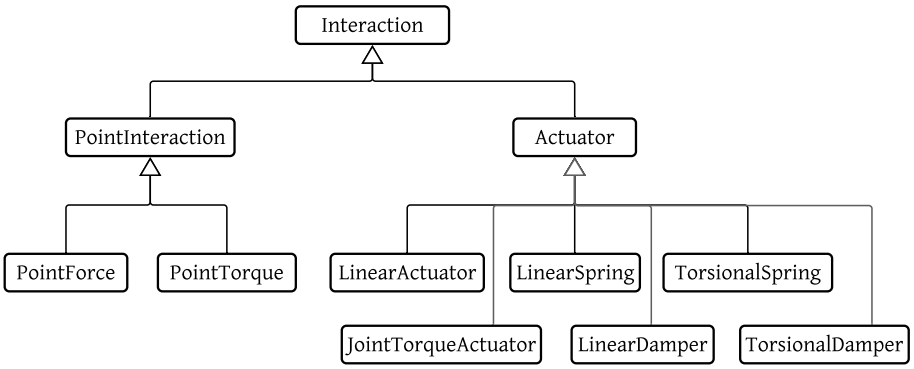


Figure 4.9: Class inheritance diagram for `Interaction` classes in *Pynamics*.

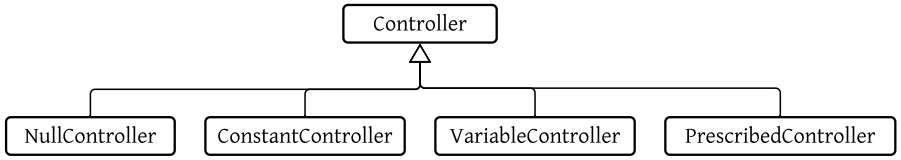


Figure 4.10: Class inheritance diagram for `Controller` classes in *Pynamics*.

Pathways

Pynamics contains `Pathway` objects so that the line of action of `Actuator` objects can be defined. *Pynamics* implements a single subclass, `LinearPathway`, which is suitable for modelling actuators that act across a straight line between two points. The `Pathway` class is included to allow easy extension of *Pynamics*, which is utilised in section 5.3.

Controllers

As mentioned previously, `Controller` classes define the magnitude and timing of interactions. The different types of controller included in *Pynamics* are shown in fig. 4.10. A `NullController` is required by `Actuator` objects that exert a passive force or torque. A `ConstantController` is similar to a `NullController`, except it provides a single parameter defining the magnitude of the interaction. This is useful when the constant magnitude of an interaction needs to be included as a static parameter in an OCP. A `VariableController` provides a single parameter, which is itself a function of time, that allows the control to vary continuously between upper and lower bounds. A `PrescribedController` allows a symbolic function to be used to describe the magnitude of control that needs to be exerted.

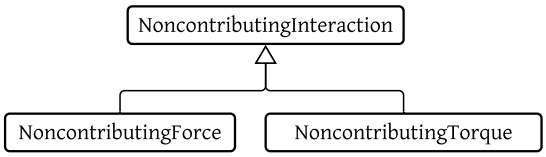


Figure 4.11: Class inheritance diagram for `NoncontributingInteraction` classes in *Pynamics*.

Noncontributing Interactions

`NoncontributingInteraction` objects serve the sole purpose of introducing auxiliary speeds into Kane's equations and thus enable noncontributing forces and torques to be brought into evidence. *Pynamics* provides both the `NoncontributingForce` and `NoncontributingTorque` classes (fig. 4.11). To bring into evidence an internal force or torque, all that is required is to create an instance of one of these classes and link it to a joint in the model.

4.3.3 Equations of Motion Generation (`form` Module)

As described in section 4.3.1, *Pynamics* uses functionality from *SymPy*'s classical mechanics module to derive EoMs using Kane's method. The main function of the `form` module of *Pynamics* is to conduct this EoMs derivation. To enable it to do so, the `form` module also analyses the user-created `Model` object to ensure that it describes a valid system, constructs the various quantities that are required by *SymPy*'s `KanesMethod` class, and generates the mass matrix and forcing vector that describe the system's EoMs.

Mechanism Analysis

The first step in producing a system's EoMs involves analysing the system with the main purpose of deducing suitable sets of independent and dependent generalised coordinates and speeds. A system with any number of constraints is guaranteed to have more generalised coordinates and speeds than DoFs. Therefore, to form a minimal-coordinate system, a choice needs to be made as to which generalised coordinates and speeds will be considered independent and will, therefore, appear in the EoMs.

Pynamics analyses the model to ensure that all bodies are connected to the rest of the system by either a joint, actuator or constraint. This ensures that the

system cannot be decomposed into two or more independent systems, as if this were the case, it would be more efficient to compute the EoMs separately for each independent subsystem. All generalised coordinates and speeds are initially considered to be independent. The generalised coordinates are separated into independent and dependent sets separately from the generalised speeds. For the generalised coordinates, each body is analysed in turn, starting from the system's origin and iterating through the bodies sequentially using a *depth-first search* (DFS) approach. For each body, the same number of generalised coordinates as there are holonomic constraints associated with it are transferred from the independent set to the dependent set. This approach of starting from the system's origin ensures that there are sufficient independent coordinates to describe the system in 3D. The same process is repeated for the generalised speeds, with the number of generalised speeds moved to the dependent set instead being governed by the number of nonholonomic constraints each body has associated with it.

Kane's Equation

With the independent generalised coordinates and speeds determined, the velocities of all points in the system can be set. This is done using the geometry of the system and ensures that the velocity of each point can be described in the global reference frame such that the partial velocities can be computed for the F_r equations and the accelerations can be computed for the F_r^* equations.

SymPy's `me.KanesMethod` class requires three matrices to be passed as arguments, describing in turn the position, velocity and acceleration constraints on the system. *Pynamics* determines these constraints from the `Constraint` objects. `PositionConstraint` and `AngleConstraint` objects both add a row to each of the position, velocity and acceleration constraint matrices. `VelocityConstraint` and `AngularVelocityConstraint` objects are similar except they both only add a row to each of the velocity and acceleration constraint matrices.

The final step before the EoMs can be generated is to describe the forces and torques acting on the system. *Pynamics* compiles a set of all interactions and their locations of action. For point interactions this is simple, but becomes more involved for more complex `Interaction` subclasses. Each `Interaction` subclass is responsible for defining how the forces and torques it generates influence the system. For example, `Actuator` objects, which attach to the mechanism at two locations, contribute two force-point pairs.

With these steps complete the EoMs are computed, yielding the mass matrix

$\mathbf{M}$ and the forcing vector $\mathbf{k}$ which form part of the linear system

$$\mathbf{M}\dot{\mathbf{u}} = \mathbf{k}\,, \tag{4.12}$$

where $\dot{\mathbf{u}}$ is the column vector of all generalised accelerations.

4.3.4 Optimal Control Problem Construction (`ocp` Module)

The `ocp` module of *Pynamics* is responsible for formulating an OCP based on a multibody model. The architecture and functionality of *Pynamics* has been designed and developed with the intention of *Pycollo* being used as its optimal control engine. Consequently, *Pycollo* is a firm dependency of *Pynamics*. The `ocp` module wraps *Pycollo* in order to fully integrate *Pycollo*'s functionality natively into *Pynamics*. This architecture abstracts away many of the complexities in formulating OCPs from the user. As such, formulating and solving OCPs in *Pynamics* is intended to be straightforward once a `Model` has been defined.

Pynamics subclasses the `pycollo.OptimalControlProblem` class (section 2.6) with its own `OptimalControlProblem` class. Instantiating and initialising an `OptimalControlProblem` object is the main responsibility of the `ocp` module. *Pynamics* implements its own `OptimalControlProblem` class to automate some of the OCP setup, thus simplifying this process on behalf of the user. The `OptimalControlProblem` class initialises the same as its superclass but implements other initialiser methods and calls these during its initialisation. These other initialiser methods are responsible for defining all of the required OCP variables, the OCP functions, the OCP bounds, an initial guess and any required settings. *Pynamics* is capable of defining OCPs with both explicit and implicit dynamics, the formulations of both follow. When instantiating an `OptimalControlProblem` object, it is possible to specify whether the explicit or implicit formulation should be generated.

Explicit Formulation

An OCP requires state, control, integral, time and static parameter variables to be defined. Additionally, required OCP functions include the state equations, path constraints, integrand functions, endpoint constraints and the objective function. In an explicit formulation this is relatively straightforward. All independent generalised coordinates and speeds become state variables. Any state or control variables associated with the model's `Interaction` objects are also added to the OCP definition. Integral variables are added only if they are required as part of the user-defined objective function or endpoint constraints. Time variables are only added if the

user has supplied unequal upper and lower bounds to the `Model` instance's initial or final time attributes. Finally, any constant model parameters that have unequal user-defined upper and lower bounds are added to the OCP definition as static parameter variables.

In the explicit formulation, all state equations are defined explicitly. This is trivial for the generalised coordinates as $\mathfrak{q}_i$, the ith generalised coordinate, is mapped directly to $\mathfrak{u}_i$, the ith generalised speed, in the state equations. That is, the ith state equation is $\dot{\mathfrak{q}}_i = \mathfrak{u}_i$.

To define the state equations for the generalised speeds, the linear system of eq. (4.12) needs to be solved for $\dot{\mathbf{u}}$. *Pynamics* does this by defining a set of symbols, one for each of the independent generalised speeds which have become state variables. An auxiliary equation corresponding to the solved linear system of eq. (4.12) is then added to the OCP definition. For the ith independent generalised speed $\mathfrak{u}_i$, this means introducing the additional symbol $\mathfrak{a}_i$ and adding the auxiliary equation

$$\mathfrak{a}_i = \left[\mathbf{M}^{-1}\mathbf{k}\right]_{i,:} , \tag{4.13}$$

where the trailing matrix subscript $i,:$ denotes the ith row of the matrix. The inversion of the mass matrix is abstracted away from *Pynamics* and is conducted by *Pycollo*.

Objective functions can be defined in two ways. Firstly, *Pynamics* offers a number of default options for objective functions that can be automatically implemented using keywords. These include the minimisation of time, via the `"minimize_time"` keyword, and the minimisation of the sum of all `Controller` control magnitudes, via the `"minimize_squared_controls"` keyword. Alternatively, a user is able to supply a custom symbolic objective function via the `objective_function` attribute.

Implicit Formulation

Pynamics' implicit formulation is similar in many regards to the explicit formulation. The main difference arises in the state equations, and also has implications for the control variables and path constraints. In the implicit formulation the linear system of eq. (4.12) remains unsolved for $\dot{\mathbf{u}}$. To get around this, *Pynamics* introduces a new control variable u_j for each independent generalised speed $\mathfrak{u}_i$. The state equations are then defined as

$$\dot{\mathfrak{u}}_i = u_j \, . \tag{4.14}$$

To enforce the EoMs, a path constraint is introduced for each of these additional control variables. These path constraints are in the form

$$\mathbf{M}_{i,:} \cdot \mathbf{u}^* - k_i = 0\,, \tag{4.15}$$

where $\mathbf{M}_{i,:}$ is the ith row of the mass matrix, $\mathbf{u}^*$ is the concatenation of all of the newly introduced control variables and k_i is the ith entry in the forcing vector. The remainder of the OCP formulation remains the same as that of the explicit formulation.

Multiphase OCPs

Pynamics is designed to support the formulation of a range of the most common multiphase OCPs. When constructing a `Model`, users are able to specify state constraints symbolically in terms of the global time symbol, accessed via the `Model` objects `time` attribute. If state constraints are specified at times that do not equal the endpoint times then *Pynamics* automatically generates a multiphase OCP. In this scenario the phase boundaries fall at the required times or within the requisite bounds. *Pynamics* does not currently support multiphase problems where the geometry of the system changes between phases as these are uncommon. Supporting this functionality would have introduced an unnecessary level of complexity with little practical benefit. The exception to this is if impulsive or discontinuous forces act on the system as these can be implemented using the appropriate `Controller` object alongside an `Interaction` where required.

Bounds and Guesses

Pynamics prescribes sensible bounds and guesses for all variables and constraints where required by the OCP formulation. For parameters where there is a physical reason to do so, *Pynamics* applies numerical bounds. For example, *Pynamics* will default to bounding all angles to the domain $[-2\pi, 2\pi]$. Guesses are also generated, if appropriate, by linearly interpolating between the relevant endpoint conditions. Where it is not possible to make a sensible assumption about bounds or guesses, *Pynamics* falls back on *Pycollo*'s default behaviour to either evaluate or handle these. It is, however, recommended that a user supplies bounds and guesses using their knowledge of the problem being solved to improve the reliability and performance of *Pycollo* [36].

Pycollo Interface and OCP Solving

As `OptimalControlProblem` subclasses its *Pycollo* counterpart, user interaction with an instance remains the same. Therefore, once an `OptimalControlProblem` instance has been created and an OCP has been fully formulated, it can be solved in the same manner as in *Pycollo* by calling the `solve` method. Similarly, if aspects of the OCP need to be adjusted by the user, such as defining a custom objective function or amending any of the default *Pycollo* settings, this can be done after instantiation and before the `solve` method is called.

4.3.5 Forward Simulation (`sim` Module)

Pynamics supports the forward simulation of modelled systems in addition to OCP formulation and solving. To do this, *Pynamics* constructs an IVP and solves this using an ODE solver. An IVP is similar in formulation to an OCP with explicit dynamics in many respects. In fact, all parts of the definition of an IVP are already contained within the definition of an OCP. Therefore, *Pynamics* implements numerous private functions that are called when formulating both OCPs and IVPs.

Analogous to *Pynamics*'s `OptimalControlProblem` class used to define an OCP, *Pynamics* also has a `ForwardSimulation` class to define an IVP. `ForwardSimulation` formulates the IVP in a very similar manner to how `OptimalControlProblem` generates its explicit formulation. As IVPs are simpler than OCPs in definition, initialising a `ForwardSimulation` object is much less involved. *Pynamics* uses an ODE solver from *SciPy*'s integrate module to solve the IVP. As with `OptimalControlProblem`, the `solve` method of `ForwardSimulation` can be called to solve the IVP.

4.3.6 Visualisation (`viz` Module)

Pynamics offers visualisation functionality to assist with analysis and debugging by means of the `viz` module. This includes the ability to generate 2D plots of the modelled system. Additional to this, *Pynamics* also supports plotting and animation of the solutions to both OCPs and IVPs via the `OptimalControlProblem` and `ForwardSimulation` classes respectively. In the case of plotting OCP solutions, this is implemented by *Pynamics* wrapping *Pycollo*'s `viz` module.

4.4 *Pynamics* Investigations

In order to test the accuracy and performance of *Pynamics*, two multibody dynamics OCPs from the academic literature were investigated. Both of these problems have previously been solved by others and so their solutions are known [194]. As *Pycollo* has been previously validated, these investigations focused on showing that *Pynamics* correctly derives EoMs and successfully forms and solves OCPs.

The literature review in section 4.1.2 established that there are conflicting opinions about the relative merits of using explicit and implicit dynamics when formulating multibody OCPs. Consequently, an objective of this chapter is to further investigate this issue by providing the first direct comparison, specific to OCPs, of the two approaches. *Pynamics* has been designed to be able to formulate OCPs with both explicit and implicit dynamics, enabling it to be used to solve the same OCP with the dynamics formulated in both manners. Performance between the two formulations was tested in a number of different areas, including:

1. the objective function value at the solution;
2. the state and control at the solution;
3. the *central processing unit* (CPU) time spent initialising the OCP;
4. the CPU time spent solving the OCP;
5. the number of NLP iterations required for the NLP subproblem to converge; and
6. the mesh error at the solution.

Taken together these factors allow the numerical properties and computational performance of the formulations to be compared. Like section 3.7, in cases where timing data is given, results were derived from five runs, with the fastest and slowest times discarded, with the mean of the remaining three being the number reported. All investigations were conducted on the same system as described in section 2.7.

The first of the two problems selected was the cart-pole swing-up problem [194]. This is a simple OCP, typically used as a teaching example in multibody dynamics and optimal control. The system has two DoFs and as such the system's EoMs are simple enough to be expressed in full and can also be derived easily by hand. This simple example is useful in clearly demonstrating that *Pynamics* correctly derives EoMs, and correctly formulates and solves OCPs.

The second example problem is the five-link bipedal walker problem from [194]. This problem is significantly more involved than the previous example as it contains more DoFs, as well as periodicity and an impulsive force at the OCP endpoint. Taken together, the pair of problems form a complementary set of examples showcasing a range of different multibody models and OCP complexities.

4.4.1 Cart-Pole Swing-Up

The cart-pole swing-up problem, as described in [194], involves the multibody cart-pole system shown in fig. 4.13. The system was modelled in *Pynamics* using two `PointMass` objects, one each to describe the cart and the point mass on the end of the pendulum. A `SlidingJoint` was used to restrict the motion of the cart to be in a single horizontal direction only and a `PinJoint` object was used to allow the pendulum arm to freely rotate around a horizontal axis through the cart. A `PointForce` (with an automatically inferred `VariableController`) was also used to apply a time-varying horizontal point force to the cart. The *Pynamics* code required to construct the cart-pole model and solve the swing-up OCP is shown in fig. 4.12.

EoMs for the system were automatically generated by *Pynamics* on calling the `Model` instance's `optimal_control_problem` method. *Pynamics* selected the cart position ($\mathfrak{q}_1$) and pendulum angle ($\mathfrak{q}_2$) as the independent generalised coordinates, while the cart velocity ($\mathfrak{u}_1$) and pendulum angular velocity ($\mathfrak{u}_2$) were selected as the independent speeds. No dependent coordinates or speeds were required as the system has two DoFs with no additional constraints. *Pynamics* formulated the system's compact mass matrix to be

$$\mathbf{M} = \begin{bmatrix} m_1 + m_2 & m_2\ell\cos(\mathfrak{q}_2) \\ m_2\ell\cos(\mathfrak{q}_2) & m_2\ell^2 \end{bmatrix} \tag{4.16}$$

and compact forcing vector to be

$$\mathbf{k} = \begin{bmatrix} F + m_2\ell\sin(\mathfrak{q}_2)\dot{\mathfrak{q}}_2^2 \\ -m_2 g\ell\sin(\mathfrak{q}_2) \end{bmatrix} . \tag{4.17}$$

Note that the compact mass matrix and forcing vector form the linear system

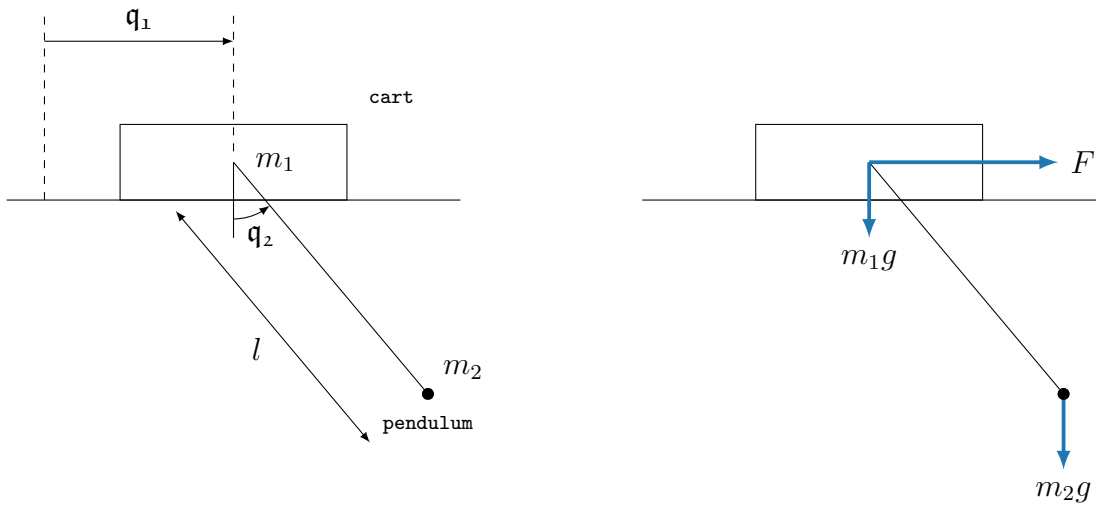
$$\mathbf{M}\dot{\mathbf{u}} = \mathbf{k} \tag{4.18}$$

describing the generalised speeds only. This is because, by design, *Pynamics* constructs systems such that the Q generalised coordinates are trivially related to the generalised speeds as

$$\dot{\mathfrak{q}}_i = \mathfrak{u}_i\,, \quad (i = 1, \ldots, Q)\ . \tag{4.19}$$

```
from pynamics import (Model, SlidingJoint, PinJoint, PointMass, PointForce,
    ↪ PI)

l = 0.5  # pendulum length (in m)
m1 = 1.0  # block mass (in kg)
m2 = 0.3  # pendulum mass (in kg)
T = 2.0  # OCP duration (in s)
d_max = 2  # maximum cart displacement (in m)
F_max = 20  # maximum force (in N)

cart_pole = Model("cart-pole")
slider = SlidingJoint("slider", model=cart_pole,
    ↪ parent_attachment=cart_pole.origin, axis="x", minimum_position=-d_max,
    ↪ maximum_position=d_max)
cart = PointMass("cart", model=cart_pole, parent_joint=slider, mass=m1,
    ↪ initial_position=0, final_position=0, initial_velocity=0,
    ↪ final_velocity=0)
pin = PinJoint("pin", model=cart_pole, parent_attachment=cart.center_of_mass,
    ↪ axis="z")
pendulum = PointMass("pendulum", model=cart_pole, parent_joint=pin,
    ↪ offset=-PI/2, position_x=l, mass=m2, initial_angle=0, final_angle=PI,
    ↪ initial_angular_velocity=0, final_angular_velocity=0)
force = PointForce("force", model=cart_pole, minimum_force=-F_max,
    ↪ maximum_force=F_max)

ocp = cart_pole.optimal_control_problem(final_time=T,
    ↪ objective_function="minimise_squared_controls",
    ↪ dynamics_formulation="implicit")
ocp.phases[0].mesh.num_mesh_sections = 20
ocp.settings.max_mesh_iterations = 1
solution = ocp.solve()
solution.plot()
solution.animate()
```

Figure 4.12: Creation of the cart-pole model and swing-up OCP using *Pynamics*.



(a) Geometry **(b)** Forces

Figure 4.13: System diagram of the cart-pole. m_1 and m_2 denote the masses of the cart and the pendulum respectively, l denotes the pendulum length, g denotes the acceleration due to gravity, F denotes the axial horizontal force applied to the cart, $\mathfrak{q}_1$ denotes the cart's horizontal axial DoF, and $\mathfrak{q}_2$ denotes the pendulum's planar rotational DoF.

These EoMs correspond exactly to the EoMs given in [194] confirming that *Pynamics* correctly derived the dynamics for this system.

The cart-pole swing-up problem involves determining the optimal way to force the cart such that the pendulum arm, initially hanging vertically below the cart at rest, is swung to a point of inverted balance vertically above the cart. Upper and lower bounds, such as bounds on the maximum horizontal distance the cart could travel and on the magnitude of the point force, were applied by setting the appropriate object properties (fig. 4.12). Similarly, initial and final states were prescribed. The initial position of the cart was set to 0. The initial and final positions of the pendulum were set to 0 and π respectively, while the initial and final velocity of the cart and angular velocity of the pendulum were all set to 0.

The objective function for the cart-pole swing-up problem is to minimise

$$\mathcal{J} = \int_0^T F\left(t\right)^2 dt\,, \tag{4.20}$$

where F is the point force applied to the cart and $T = 2$. *Pynamics* offers certain preconstructed objective functions to simplify the specification of this aspect of the OCP. One such objective function is the minimisation of the equally-weighted sum of all `Controller` control parameters. As this utility was directly relevant here, it was used by defining the objective function using *Pynamics*' `"minimise_squared_controls"` keyword.

Explicit Formulation

Generating the EoMs and OCP in explicit dynamics mode resulted in an OCP with four state variables

$$\mathbf{y} = \begin{bmatrix} \mathfrak{q}_1(t) & \mathfrak{q}_2(t) & \mathfrak{u}_1(t) & \mathfrak{u}_2(t) \end{bmatrix} \tag{4.21}$$

and one control variable

$$\mathbf{u} = \begin{bmatrix} F(t) \end{bmatrix} . \tag{4.22}$$

As an integrand function was required to express the objective function, a single integral variable q_1 was automatically added by *Pycollo* (on behalf of *Pynamics*).

The four state equations were formed as

$$\dot{\mathfrak{q}}_1 = \mathfrak{u}_1 \tag{4.23}$$

$$\dot{\mathfrak{q}}_2 = \mathfrak{u}_2 \tag{4.24}$$

$$\dot{\mathfrak{u}}_1 = a_1 \tag{4.25}$$

$$\dot{\mathfrak{u}}_2 = a_2 , \tag{4.26}$$

where a_1 and a_2 correspond to the auxiliary substitutions

$$a_1 = \frac{F + m_2 g \cos(\mathfrak{q}_2) \sin(\mathfrak{q}_2) + m_2 \ell \sin(\mathfrak{q}_2) \dot{\mathfrak{q}}_2^2}{m_1 + m_2 \sin^2(\mathfrak{q}_2)} \tag{4.27}$$

$$a_2 = -\frac{F \cos(\mathfrak{q}_2) + (m_1 + m_2) g \sin(\mathfrak{q}_2) + m_2 \ell \cos(\mathfrak{q}_2) \sin(\mathfrak{q}_2) \dot{\mathfrak{q}}_2^2}{\ell \left(m_1 + m_2 \sin^2(\mathfrak{q}_2)\right)} \tag{4.28}$$

generated by *Pycollo* by symbolically solving eq. (4.18) for

$$\dot{\mathbf{u}} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} . \tag{4.29}$$

The single integrand function g_1, which corresponds to the integral variables q_1, was formulated as

$$g_1 = F^2 , \tag{4.30}$$

with the objective function being

$$\mathcal{J} = q_1 . \tag{4.31}$$

Implicit Formulation

Generating the EoMs and OCP in explicit dynamics mode resulted in an OCP with the same four state variables as eq. (4.21). Unlike the explicit formulation, however, the implicit formulation contained three control variables

$$\mathbf{u} = \begin{bmatrix} F(t) & u_2(t) & u_3(t) \end{bmatrix} . \tag{4.32}$$

Again, as with the explicit formulation, a single integral variable q_1 was required.

The four state equations were formed as

$$\dot{\mathfrak{q}}_1 = \mathfrak{u}_1 \tag{4.33}$$

$$\dot{\mathfrak{q}}_2 = \mathfrak{u}_2 \tag{4.34}$$

$$\dot{\mathfrak{u}}_1 = u_2 \tag{4.35}$$

$$\dot{\mathfrak{u}}_2 = u_3 \,. \tag{4.36}$$

The main discrepancy arose whereby the implicit formulation also required the inclusion of two path constraints to enforce the dynamics of eq. (4.18). These were

$$(m_1 + m_2)\, u_2 + m_2 \ell \cos(\mathfrak{q}_2) u_3 - F - m_2 \ell \sin(\mathfrak{q}_2) \dot{\mathfrak{q}}_2^2 = 0 \tag{4.37}$$

$$m_2 \ell \cos(\mathfrak{q}_2) u_2 + m_2 \ell^2 u_3 + m_2 g \ell \sin(\mathfrak{q}_2) = 0 \,. \tag{4.38}$$

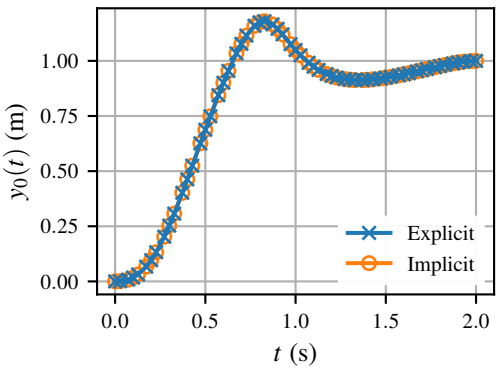
Note that these path constraints correspond directly to the EoMs of eqs. (4.16) and (4.17) via a slight rearrangement and using the substitutions $u_2 = \dot{\mathfrak{u}}_1$ and $u_3 = \dot{\mathfrak{u}}_2$. The integrand function and objective function were identical to those of the explicit formulation shown in eqs. (4.30) and (4.31).

OCP Solutions

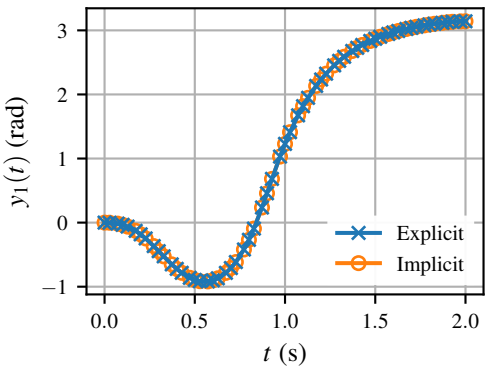
Two default *Pycollo* settings were changed during the solving of both OCP formulations. Firstly, the number of mesh sections in the initial mesh was increased from 10 to 20 in order to match the number of mesh sections used in the implementation provided by [194]. Secondly, automatic mesh refinement was turned off as the purpose of this experiment was assess the accuracy of *Pynamics* and to compare the explicit and implicit formulations. As such, fair comparison is only possible when the meshes on which the NLP subproblems are being solved are identical, which is only guaranteed on the first NLP iteration.

The objective function evaluations at the solution using the explicit and implicit formulations were identical to machine precision, both being 58.8163. While [194] did not provide a numerical value for the solution of the problem, solving the cart-pole swing-up problem using the software provided supplementary to [194] resulted in an objective function at the solution of 58.8076. This difference of 0.0148% can be attributed to differences in the discretisations used by the two software packages. Agreement of the objective function evaluations using *Pynamics* and [194] confirms that *Pynamics* was able to correctly solve the cart-pole swing-up problem using both explicit and implicit OCP formulations.

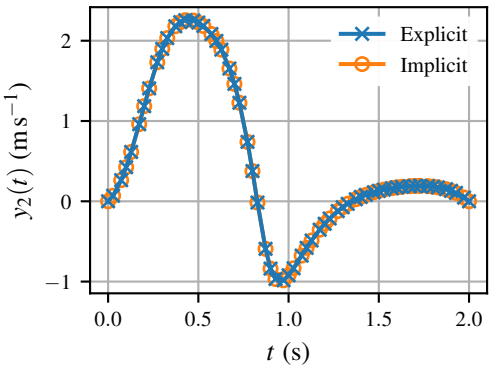
The optimal state found using both OCP formulations is shown in fig. 4.14. Similarly, the optimal control is shown in fig. 4.15. As can clearly be seen, the optimal



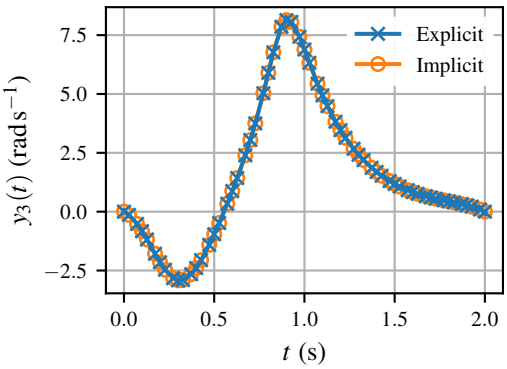
(a) Position solution



(b) Angle solution

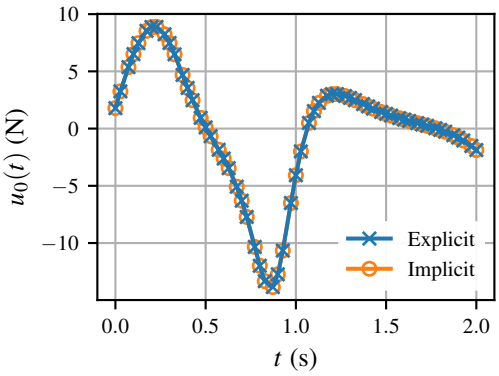


(c) Velocity solution

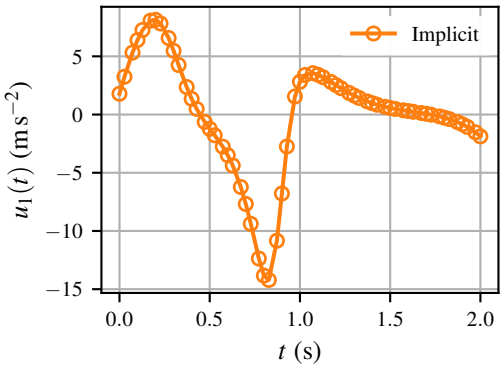


(d) Angular velocity solution

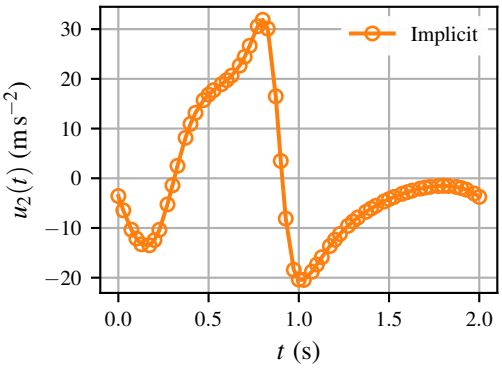
Figure 4.14: Comparison of the optimal state solutions to the cart-pole swing-up problem obtained using *Pynamics* in explicit and implicit modes.



(a) Force solution



(b) Acceleration solution



(c) Angular acceleration solution

Figure 4.15: Comparison of the optimal control solutions to the cart-pole swing-up problem obtained using *Pynamics* in explicit and implicit modes. Note that solutions for the acceleration $u_1(t)$ and angular acceleration $u_2(t)$ are only shown for the implicit formulation as these variables did not form part of the explicit formulation.

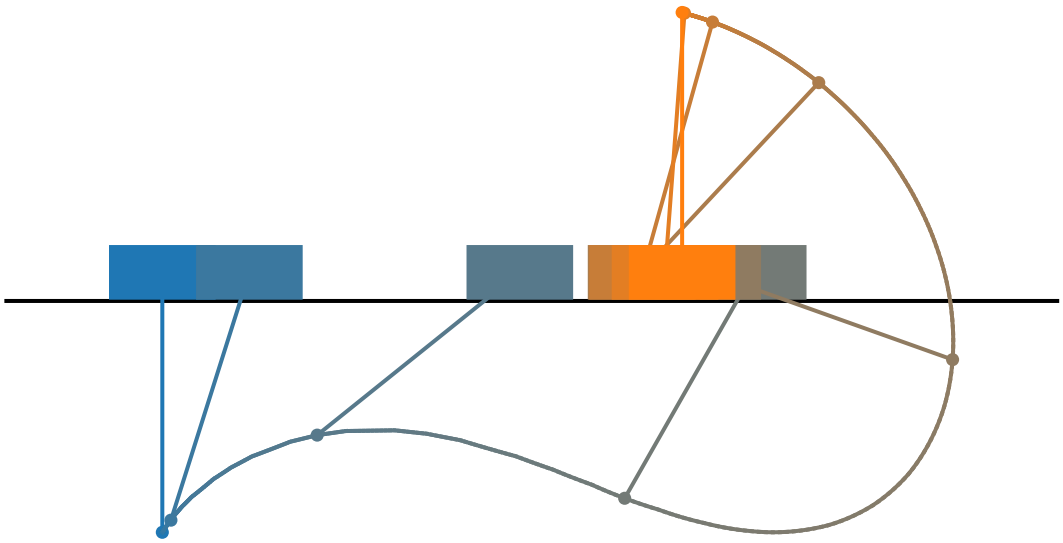


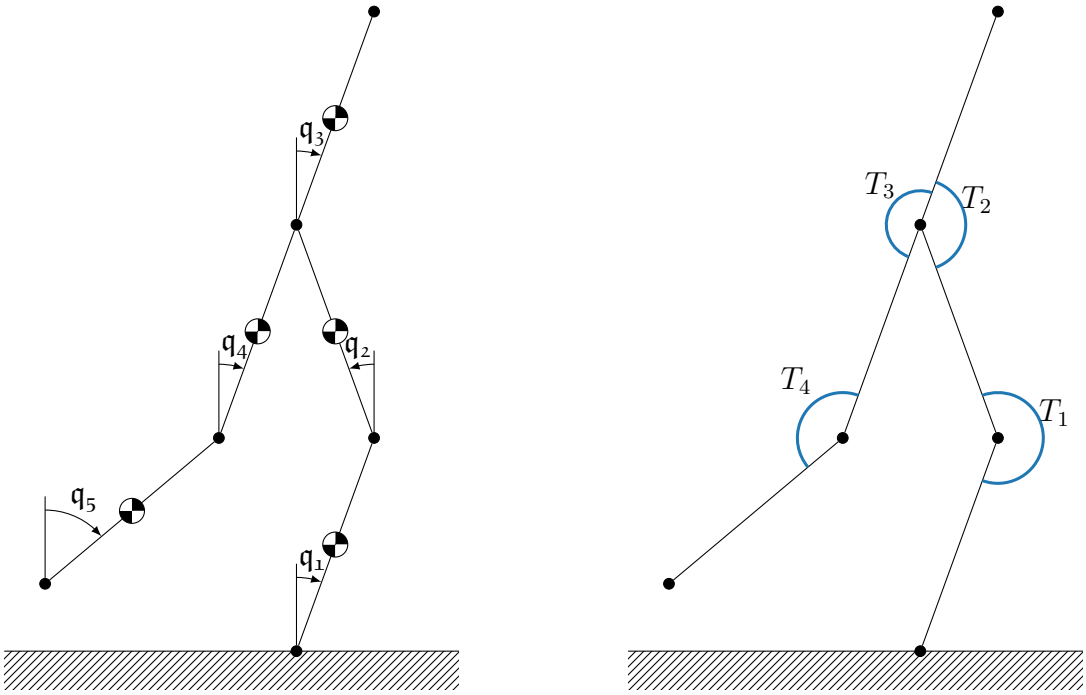
Figure 4.16: Illustration of the optimum cart-pole swing-up trajectory. Nine uniformly spaced frames at 0.25 s intervals are shown, from start (blue) to end (orange).

state and control found using both approaches are identical, further confirming the accuracy of *Pynamics* using both explicit and implicit dynamics. Finally, fig. 4.16 illustrates the optimal trajectory taken by the cart-pole during the swing-up.

The differences between the explicit and implicit OCPs arose in the time taken to solve both OCPs, the progression of the NLP subproblems and the resulting mesh errors at the solutions. The explicit and implicit formulations required 186.29 ms and 198.39 ms to be completely solve respectively. The explicit formulation required 156.50 ms to initialise the OCP compared to 147.00 ms for the implicit formulation. This relative timing reversed for the time taken to solve the NLP subproblems with the explicit and implicit formulations of the OCP being solved in 29.79 ms and 51.39 ms respectively. This also corresponds to the number of NLP iterations required in both cases, with the explicit OCP converging in 12 iterations compared to 16 iterations for the implicit OCP. Finally, the implicit OCP solved with a smaller mesh error of 1.422×10^{-5} compared to 3.282×10^{-4} for the explicit OCP.

4.4.2 Five-Link Biped

The five-link biped OCP [194] involves determining a periodic gait for a biped. The system is a symmetric planar model consisting of five rigid bodies representing a torso and two two-segment legs. The rigid bodies are connected by five idealised pin joints, four of which represent the two hips and two knees. The two knees and two hips are also actuated by controllable joint torques. The final pin joint attaches the walker to the ground, resulting in a stance leg which supports the walker's weight



(a) Geometry **(b)** Forces

Figure 4.17: System diagram of the five-link walker.

and a swing leg which is free to move above the ground. An illustration of the system is shown in fig. 4.17.

The system was created in *Pynamics* using five `RigidBody` objects connected by five `PinJoint` objects. One of these `PinJoint` objects was used to attached the stance leg to the `ModelOriginPoint`. The model was actuated using four `JointTorqueActuator` objects paired with four `VariableController` objects, one at each of the stance knee, stance hip, swing hip and swing knee joints. The model components were parameterised taking the same values from [194], which were themselves selected to match [77].

The resulting model has five DoFs. *Pynamics* described the system using five generalised coordinates, one for the angle of each rigid body, and five generalised speeds, one for the angular velocity of each rigid body. *Pynamics* generated the 5×5 nontrivial mass matrix $\mathbf{M}_{5\times5}$ and corresponding forcing vector $\mathbf{k}_{5\times1}$ describing the time derivatives of the generalised speeds, which formed the linear system

$$\mathbf{M}_{5\times5} \begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \\ \dot{u}_3 \\ \dot{u}_4 \\ \dot{u}_5 \end{bmatrix} = \mathbf{k}_{5\times1} \,. \tag{4.39}$$

The governing equations of motion for the system's dynamics are not included here due to their complexity and length.

The objective function for the five-link biped walker problem is to minimise

$$\mathcal{J} = \int_0^T T_1^2(t) + T_2^2(t) + T_3^2(t) + T_4^2(t)\, dt\,, \tag{4.40}$$

where T_i for $i = 1, 2, 3, 4$ are the joint torques applied to the stance knee, stance hip, swing hip and swing knee respectively. This objective function is used because it produces smooth, well-behaved solutions, and penalises large torques which are usually undesirable in real systems [194]. In this investigation, a value of $T = 0.7\,\mathrm{s}$ was used to match [194]. As was the case for the cart-pole swing-up problem, *Pynamics*' `"minimise_squared_control"` keyword was used to implement the objective function of eq. (4.40).

In order to have the biped produce a sensible gait, additional constraints were required. Firstly, as a single stride was being simulated, periodicity between the stance and swing legs was imposed. This was done by ensuring that the angles of the swing leg at the initial time matched the angles of the stance leg at the final time and visa versa. Similarly, the angle of the torso at the initial and final times were constrained to be equal. Periodicity of the angular velocities was less trivial as the problem requires an impulsive heel-strike to take place. The heel-strike map of [194] was implemented as five additional endpoint constraints by having *Pynamics* generate five equations conserving angular momentum of the system before and after the application of an impulsive point force at the swing foot instantaneously before heel-strike. Secondly, a pair of constraints to enforce the stride length were imposed. This was done by using the symbolic equation describing the position of the swing foot generated by *Pynamics* to form a constraint on the swing foot's horizontal position and vertical position at the final time. These position constraints were equal to $D = 0.5\,\mathrm{m}$ and 0 respectively, enforcing a stride length of D on a flat surface. Thirdly, similar constraints on the velocity of the swing foot at the initial and final times were imposed to ensure that the swing foot leaves the ground at the moment of the initial time and only touches the ground at the moment of the final time. These constraints were required due to the method used to derive the heel-strike map [194]. Finally, a set of path constraints were introduced to ensure that the swing foot did not cross the horizontal plane representing the ground at any point during the gait phase. The symbolic equation describing the vertical position of the swing foot was used again here.

Explicit Formulation

The OCP formulated by *Pynamics* using explicit dynamics had ten state variables and four control variables. The ten state variables corresponded to the five generalised coordinates and five generalised speeds present in the EoMs

$$\mathbf{y} = \begin{bmatrix} \mathfrak{q}_1 & \mathfrak{q}_2 & \mathfrak{q}_3 & \mathfrak{q}_4 & \mathfrak{q}_5 & \mathfrak{u}_1 & \mathfrak{u}_2 & \mathfrak{u}_3 & \mathfrak{u}_4 & \mathfrak{u}_5 \end{bmatrix} . \tag{4.41}$$

The four control variables corresponded to the control parameters describing the magnitudes of the torques exerted by the four `JointTorqueActuator` objects

$$\mathbf{u} = \begin{bmatrix} T_1 & T_2 & T_3 & T_4 \end{bmatrix} . \tag{4.42}$$

The ten state equations were formed as

$$\dot{\mathfrak{q}}_1 = \mathfrak{u}_1 \tag{4.43}$$
$$\dot{\mathfrak{q}}_2 = \mathfrak{u}_2 \tag{4.44}$$
$$\dot{\mathfrak{q}}_3 = \mathfrak{u}_3 \tag{4.45}$$
$$\dot{\mathfrak{q}}_4 = \mathfrak{u}_4 \tag{4.46}$$
$$\dot{\mathfrak{q}}_5 = \mathfrak{u}_5 \tag{4.47}$$
$$\dot{\mathfrak{u}}_1 = a_1 \tag{4.48}$$
$$\dot{\mathfrak{u}}_2 = a_2 \tag{4.49}$$
$$\dot{\mathfrak{u}}_3 = a_3 \tag{4.50}$$
$$\dot{\mathfrak{u}}_4 = a_4 \tag{4.51}$$
$$\dot{\mathfrak{u}}_5 = a_5 \, , \tag{4.52}$$

where a_i for $i = 1, 2, 3, 4, 5$ corresponded to the five auxiliary substitutions generated by *Pycollo* (on behalf of *Pynamics*) by symbolically solving eq. (4.39) for

$$\dot{\mathbf{u}} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} . \tag{4.53}$$

The single integrand function g_1, which corresponds to the integral variables q_1, was formulated as

$$g_1 = T_1^2 + T_2^2 + T_3^2 + T_4^2 \, , \tag{4.54}$$

with the objective function being

$$\mathcal{J} = q_1 \, . \tag{4.55}$$

Implicit Formulation

Generating the EoMs and OCP using implicit dynamics resulted in an OCP with the same ten state variables as when explicit dynamics were used (eq. (4.41)). Unlike the explicit formulation, however, the implicit formulation contained an additional five control variables, resulting in nine total

$$\mathbf{u} = \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \\ u_9 \end{bmatrix} . \tag{4.56}$$

Again, as with the explicit formulation, a single integral variable q_1 was required.

The ten state equations were formed as

$$\dot{\mathrm{q}}_1 = \mathrm{u}_1 \tag{4.57}$$
$$\dot{\mathrm{q}}_2 = \mathrm{u}_2 \tag{4.58}$$
$$\dot{\mathrm{q}}_3 = \mathrm{u}_3 \tag{4.59}$$
$$\dot{\mathrm{q}}_4 = \mathrm{u}_4 \tag{4.60}$$
$$\dot{\mathrm{q}}_5 = \mathrm{u}_5 \tag{4.61}$$
$$\dot{\mathrm{u}}_1 = u_5 \tag{4.62}$$
$$\dot{\mathrm{u}}_2 = u_6 \tag{4.63}$$
$$\dot{\mathrm{u}}_3 = u_7 \tag{4.64}$$
$$\dot{\mathrm{u}}_4 = u_8 \tag{4.65}$$
$$\dot{\mathrm{u}}_5 = u_9 \,, \tag{4.66}$$

while the dynamics of eq. (4.39) were enforced by introducing the five path constraints

$$\mathbf{M}_{5\times5} \begin{bmatrix} u_5 \\ u_6 \\ u_7 \\ u_8 \\ u_9 \end{bmatrix} - \mathbf{k}_{5\times1} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} . \tag{4.67}$$

The integrand function and objective function were identical to those of the explicit formulation shown in eqs. (4.54) and (4.55).

OCP Solutions

Pycollo defaults were used for all settings except one. As with the cart-pole swing-up problem, automatic mesh refinement was turned off. Again, this was to ensure that the results of the two investigations were not influenced by the discretisation mesh and could be validly compared.

Pynamics was able to successfully solve the OCP using both explicit and implicit dynamics. Both the explicit and implicit formulations led to an objective function evaluation of 445.9796 for the solved OCP. A numerical value for the solved OCP was not given in [194], however a computational implementation of the same problem has been provided elsewhere [107]. Using [107] and default settings, an objective function of 432.5045 results. This increases to 446.7251 if the mesh density is increased by a factor of 10. There is close agreement between the objective function achieved using *Pynamics* and using [107] indicating that *Pynamics* correctly solves the OCP. Note that the discrepancy between the objective functions is likely due to [107] using the backward-Euler collocation scheme which is inaccurate in comparison to the orthogonal collocation scheme implemented by *Pycollo* on behalf of *Pynamics*.

The optimal state found using both OCP formulations for the angles and angular velocities are shown in fig. 4.18 and fig. 4.19 respectively. Similarly, the optimal control for the joint torques are shown in fig. 4.20. The optimal control for the angular accelerations, the additional five control variables in the implicit formulation, are not shown as they were only present in one formulation and are also the time derivatives of the angular velocity state variables. Exact agreement of the optimal state and control found using both approaches is clearly illustrated. This further confirms that the explicit and implicit formulations generated by *Pynamics* are equally valid and accurate. Finally, an illustration of the five-link biped walker's optimal gait is shown in fig. 4.21.

Solution of the NLP subproblems unfolded similarly, with the explicit formulation requiring 15 NLP iterations compared to 16 required for the implicit formulation. The explicit NLP subproblem was solved with a resulting mesh error of 1.168×10^{-5}. This was more than an order of magnitude larger than the mesh error of 8.352×10^{-7} achieved at the solution to the implicit NLP subproblem. The main difference between the two formulations was seen in the time taken to formulate

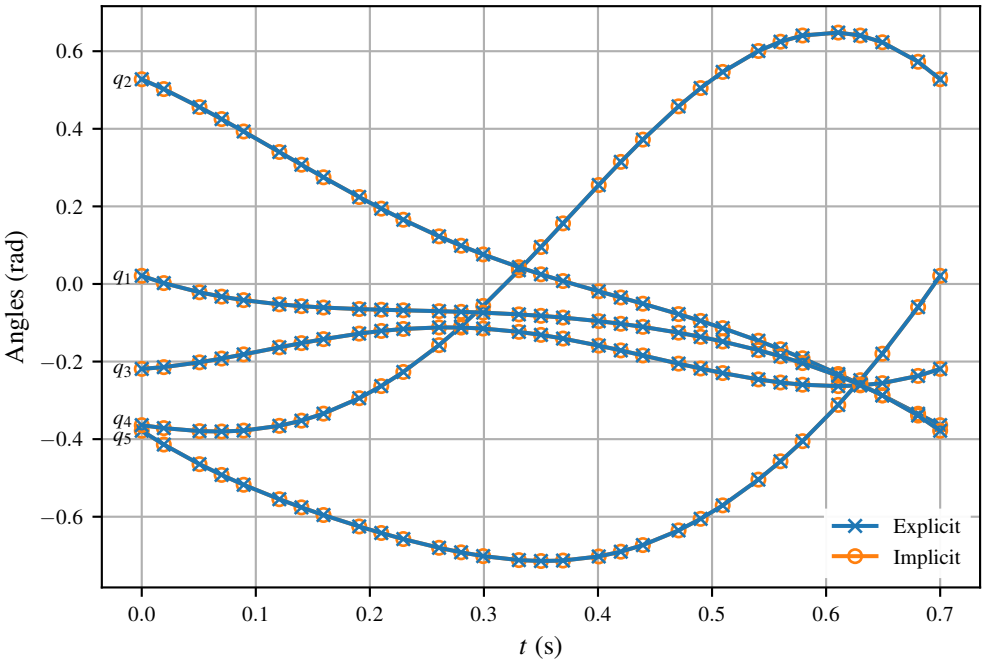


Figure 4.18: Comparison of the optimal angle state solutions to the five-link biped walker problem obtained using *Pynamics* in explicit and implicit modes.

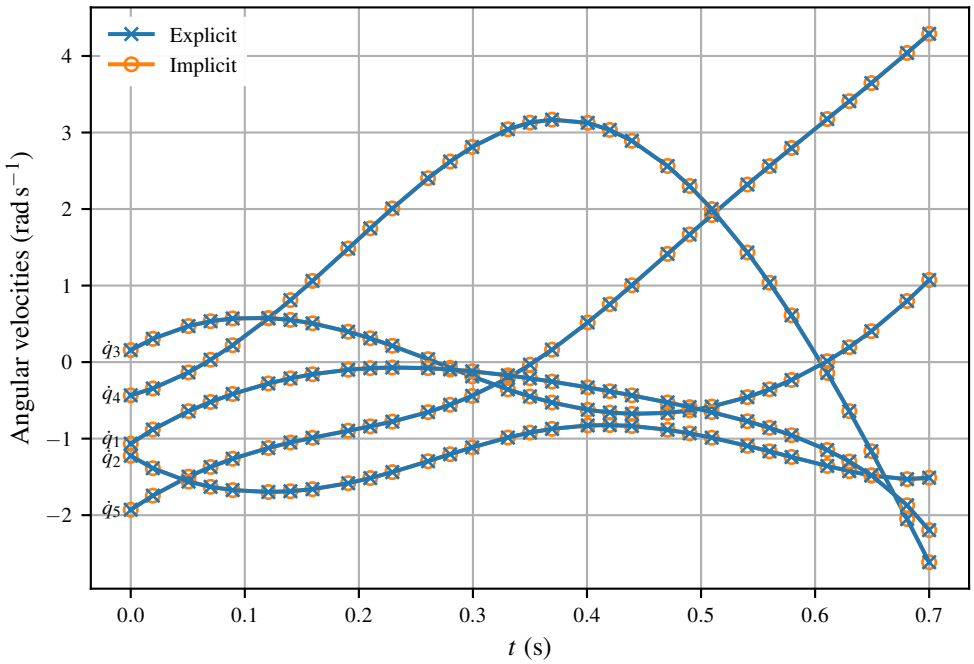


Figure 4.19: Comparison of the optimal angular velocity state solutions to the five-link biped walker problem obtained using *Pynamics* in explicit and implicit modes.

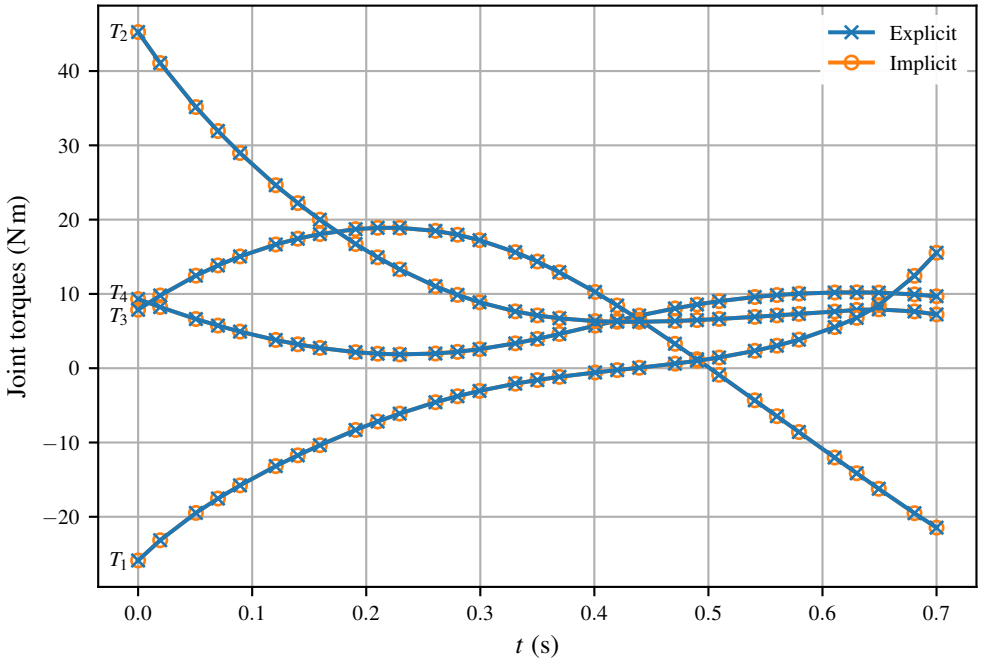


Figure 4.20: Comparison of the optimal joint torque control solutions to the five-link biped walker problem obtained using *Pynamics* in explicit and implicit modes.

and solve the OCP. The implicit formulation required 493.90 ms to initialise and 79.67 ms to solve the OCP; 573.57 ms in total. The explicit formulation required 281.01 s to initialise and 11.65 s to solve the OCP, a total of 292.66 s. Note that the times for the implicit formulation are given in ms, while the times for the explicit formulation are given in s.

4.5 Discussion

The results in section 4.4 confirm that *Pynamics* is able to accurately and reliably formulate and solve OCPs based on multibody dynamics. Furthermore, *Pynamics* is capable of this using both explicit and implicit EoMs, with the same solution being obtained using both methods for the two example problems investigated.

Formulating an OCP using implicit dynamics resulted in a larger NLP subproblem, in terms of both variables and constraints. The increase in decision variables was attributable to the additional control variables being introduced to act as the state equations for the generalised speeds. Additionally, the increase in constraints was attributable to additional path constraints being required to enforce the system dynamics. For the cart-pole swing-up problem, 298 variables and 241 constraints

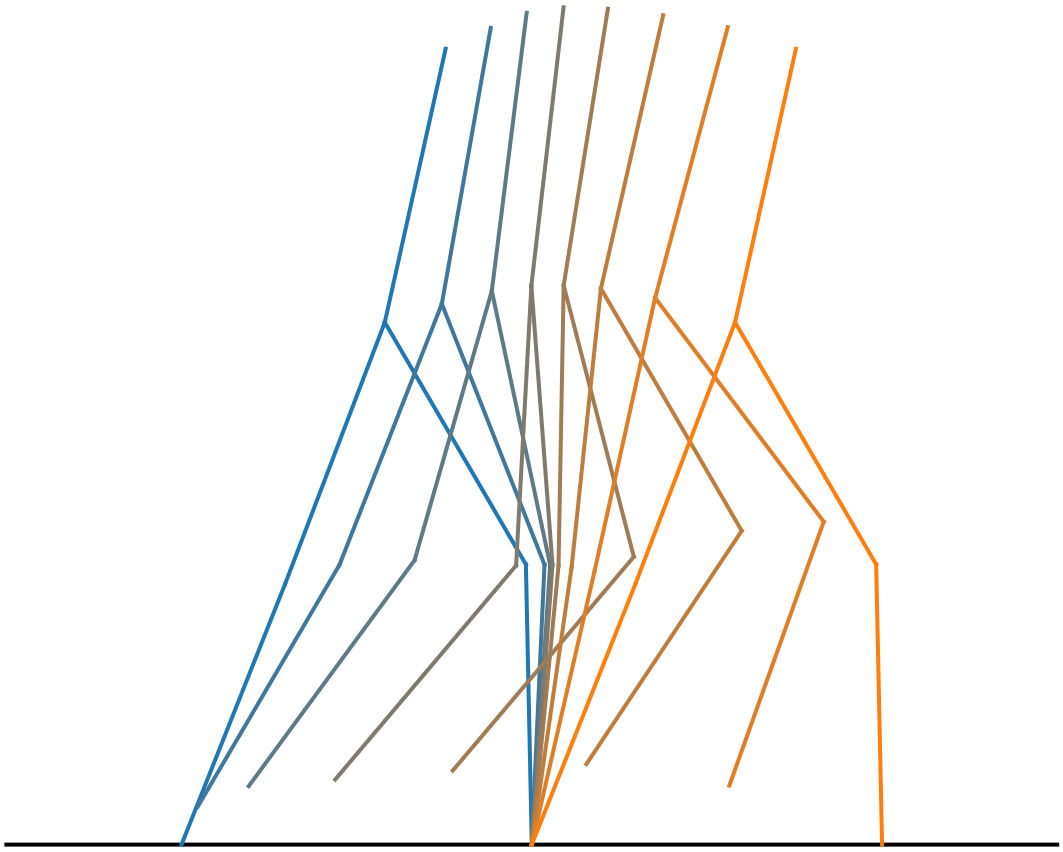


Figure 4.21: Illustration of the five-link walker trajectory that minimises the integral of the sum of squared torques. Eight uniformly spaced frames at 0.1 s intervals are shown, from start (blue) to end (orange).

were required when explicit dynamics were used. This was in comparison to 420 variables and 363 constraints when implicit dynamics were used. Similarly, for the five-link biped walker problem, 435 variables and 313 constraints were required when explicit dynamics were used. This was in comparison to 590 variables and 468 constraints when implicit dynamics were used. This corresponds to between a 35.6% and 40.9% increase in variables, and between a 49.5% and 50.6% increase in constraints when moving from an explicit to an implicit formulation. This constitutes a substantial increase in the size of the NLP subproblem. However, in the case of the two test problems, both the explicit and implicit OCPs were solved equally successfully, indicated by the relatively similar and small number of NLP iterations required for the NLP subproblems to converge. Therefore, it can be concluded that the increase in size of the NLP subproblem does not detrimentally affect the convergence properties of the generated OCPs as was suggested might be a consideration in section 4.1.

For both problems, the initialisation times accounted for the majority of the total OCP processing times irrespective of formulation. For the cart-pole swing-up problem, the smaller of the two OCPs investigated, there was limited difference in the time taken to initialise the OCP between the two formulations. Initialisation time for the explicit OCP was 6.46% greater than for the implicit OCP. This was not the case for the larger example problem, the five-link biped walker problem, in which initialising the explicit OCP took 569 times longer than the implicit OCP. The absolute time required to initialise the explicit OCP (which accounted for 96.0% of the total OCP processing time) was of the order of minutes. This was in stark contrast to the implicit OCP which initialised and solved in under one second. The explicit formulation of this problem was therefore not suitable for practical use. This discrepancy was due entirely to the increased size of the expression graph that *Pycollo* was required to generate as a result of having to invert the mass matrix in the explicit formulation. Solving a linear system of n equations, as is required in order to formulate the explicit EoMs, has complexity up to $\mathcal{O}(n^3)$ depending on the algorithm used [312]. Therefore, as the number of DoFs in the modelled system increases, the computational cost associated with deriving the dynamics explicitly grows exponentially. The implications of the larger expression graph compound when computing the second-order OCP derivatives via either AD or hSAD because the computational cost of deriving these scales proportionally to the square of the size of the expression graph.

For both of the test problems, the solution to the implicit OCP yielded a mesh error more than an order of magnitude smaller than the solution to the explicit OCP. This indicates that the implicit OCP was solved with greater accuracy. In addition, as both OCPs were solved on the same temporal mesh, it also indicates that the

implicit equations are more stable. The state equations are a key component in evaluating the mesh error. Therefore, if two formulations of the same dynamical equations result in different mesh errors, this indicates that the two formulations are not equally stable. The formulation resulting in a larger mesh error must have stiffer dynamical equations as a change to the discretisation mesh results in a larger change in the values that the function evaluates to. As such, the explicit dynamical equations are stiffer and less numerically stable than the implicit ones.

In the case of both test problems, the number of NLP iterations required for the NLP subproblem to converge was fewer when explicit dynamics were used than implicit dynamics. However, due to the small total number of NLP iterations required in all cases, this result is not considered to be significant. Of greater importance are the relative times taken to solve the OCPs. For the smaller cart-pole swing-up problem, the explicit NLP subproblem was solved 1.73 times faster than the implicit one. In this case, the majority of this time difference can be attributed to the additional NLP iterations required for the implicit NLP subproblem to converge. This was not the case for the other larger example problem, where the implicit problem was solved 146 times faster, despite requiring an additional NLP iteration. Here, the large increase in solve time was due to the more expensive-to-evaluate NLP derivatives resulting from the explicit formulation. A single evaluation of the constraints Jacobian was 231.17 ms and 230.45 µs for the explicit and implicit formulations respectively. Similarly, a single evaluation of the Lagrangian Hessian was 445.07 ms for the explicit formulation compared to 398.56 µs for the implicit formulation. In both cases these large derivative matrices were in excess of 1000 times more computationally expensive to compute when explicit dynamics were used. This was to be expected given the explicit formulations' significantly larger expression graphs. Therefore, it should be expected that for systems with more than two DoFs, the NLP subproblem will take a longer time to solve even if it can be solved in fewer NLP iterations.

Even though the OCPs could be successfully solved using both explicit and implicit dynamics, the analysis of the results of these investigations strongly indicates that implicit dynamics should be used when formulating OCPs based on multibody systems. For systems containing more than two DoFs, there are sound theoretical reasons and experimental evidence to support this [10, 36, 136]. These reasons can be summarised as follows:

1. it is less computationally expensive to initialise the OCP when implicit dynamics are used because inversion of the mass matrix is not required, resulting in a significantly smaller expression graph which needs to be differentiated through to determine the OCP derivatives [57, 136];

2. it is less computationally expensive to solve the NLP subproblem, despite it being more than 50% larger, because the NLP derivatives resulting from the implicit dynamics are significantly simpler and thus computationally cheaper to evaluate [10]; and

3. the implicit equations are more numerically stable and result in smaller mesh errors [36, 71].

4.6 Conclusions

Pynamics, an open-source *Python* package for optimal control involving multibody dynamics, has been developed. It:

- allows users to construct computational models of multibody systems using few lines of code, a consequence of its concise API and extensive library of high level of abstraction multibody model components;
- greatly assists the user in constructing the multibody OCP, once a multibody model has been constructed using its API, by:
  - efficiently determining compact EoMs for the modelled system;
  - formulating the OCP by determining the required state variables, control variables, static parameter variables, state equations, path constraints, integrand functions and endpoint constraints; and
  - interfacing with *Pycollo* to initialise and solve the OCP using a direct collocation method;
- is capable of formulating OCPs using both explicit and implicit dynamics; and
- provides additional facilities to aid analysis, with modules for conducting forward simulations and animating solutions.

Explicit and implicit formulations of dynamics were compared by using *Pynamics* to solve a pair of multibody OCPs from the literature. The investigations:

- demonstrated that *Pynamics* is capable of accurately and reliably solving multibody OCPs;
- concluded that using implicit dynamics to formulate multibody OCPs is significantly more computationally efficient and numerically robust for systems with more than two DoFs; and

- recommended that implicit formulations of dynamics should be the preferred approach.

The open-source provision of *Pynamics* will allow researchers and practitioners without specific expertise in multibody dynamics or optimal control to investigate OCPs involving multibody systems. In particular, it is recommended that future research investigates:

- how handling of event detection can be incorporated into multibody OCPs so that systems with altered dynamics can be modelled and predictively simulated using direct collocation;
- making additions to the *Pynamics* component library; and
- adding support for contact modelling in *Pynamics*.

Chapter 5

Musculoskeletal Modelling

This chapter focuses on the unmet need for methodologies and software specifically focused on the formulation and solution of predictive simulations involving musculoskeletal models. A review of the related academic literature and background material begin this chapter. The relevant objective from section 1.3 is restated, and a set of sub-objectives laid out, in section 5.2. The development and attributes of a software package are described in section 5.3, along with a number of biomechanical modelling methods and components specifically for predictive simulation. Section 5.4 details the extensive verification and validation of the developed software, including replication studies and sensitivity analyses. This is followed by a set of recommendations for the formulation and solving of musculoskeletal predictive simulations, and suggestions for how the work contained within this chapter can be extended in the future. Finally, the objective and sub-objectives stated in section 5.2 are reviewed, and an assessment of the extent to which they have been met is made.

5.1 Background, Theory and Review

Biomechanical modelling in conjunction with simulation is a valuable analysis technique due to its ability to provide a quantitative interpretation of movement tasks. Furthermore, this technique has become increasingly applied in recent years due to improvements in computing power now making it practical to model the human anatomy with sufficient detail that realistic movements can be simulated [259]. Biomechanical models of the human body fall broadly into two types: torque-actuated models articulated by net torques about each of the joints, and muscle-actuated models articulated by numerous muscles spanning the joints and pairs of

joints.

5.1.1 Torque-Actuated Models

Various *two-dimensional* (2D) torque-actuated models have been developed and used to investigate gait [128, 238, 245, 260], crutch walking [110, 111], maximal jumping [198, 199, 216], tumbling [337], pedalling [120, 134, 175, 176, 177, 193], fast bowling in cricket [112], and tennis ground strokes [195]. In a torque-actuated model, torque generators represent the net effect of all muscles spanning a particular joint [234, 338]. These torque generators typically model a muscle-tendon complex, consisting of a contractile component and a series elastic component, such that both passive and active torque generation are modelled accurately [234]. Furthermore, separate agonist and antagonist torque generators are used to model opposite exertions as this helps to achieve realistic kinematics and activation dynamics [234]. This is of particular importance in situations where co-contraction occurs, for example during impact landings [339].

As torque generators represent the net effect of all muscles about a particular joint, they can not be used to determine the role and contributions of individual muscles [234]. Instead, their use in predictive simulation has focussed on predicting global performance and kinematics. In these cases of performance prediction, subject-specific modelling is particularly desirable. Torque-actuated models are advantageous as they can be readily parameterised from measured maximal isometric and isokinetic exertions on a force dynamometer [14, 116, 197, 340]. A torque generator's contractile component and series elastic component can be parameterised by solving an optimisation problem in which the difference between measured and simulate joint torques is minimised [79, 116, 199].

Torque-actuated models have typically used monoarticular (spanning a single joint) torque generators, which can limit insight into intermuscular coordination due to only the net effect of muscle moments about each joint being represented. Torque-actuated models of pedalling have led to incorrect interpretations of muscles' functions in such movements. A torque-actuated model was used to conclude that the ankle moment is only responsible for transferring power produced by the hip and knee to the cranks [120]. However, experimental research has shown that the ankle joint also contributes approximately 20% of total output crank power [103].

Monoarticular torque generators also assume that the torque function is based solely on the kinematics of the primary joint [116]. This assumption is invalid as it ignores the effects of biarticular (spanning more than one joint) muscles [79], which

have been shown to impact performance [306]. When monoarticular torque generators have been used, no insight into the role of biarticular muscles was gained because of the modelling joint torques being independent of one another [120]. Models involving biarticular torque generators have been developed [216]. Such models have been shown to produce more realistic simulations of jumping tasks [216].

One notable consideration regarding torque-actuated models is that it is difficult to obtain a full set of subject-specific torque parameters in *three-dimensional* (3D) [112, 338]. A 3D torque-actuated model of crutch walking has been created [110]. However, this model was used to solve an inverse-dynamics problem, and this category of model is yet to be used in a predictive simulation. As such, for the majority of movement simulations, this approach is limited to 2D [234].

5.1.2 Muscle-Actuated Models

The first muscle-actuated model of the human body, and simulation capability to use it to investigate human motion, was developed in 1976 [150, 151]. Other general, planar, 2D, muscle-actuated biomechanical models of the lower extremity have since been developed [173]. Similar models have been used to investigate pedalling [53, 250, 251, 252, 273, 274, 279, 280, 307, 343, 344] and other movement tasks like gait [84] and jumping [55, 154, 295, 306, 308].

3D biomechanical modelling of the lower limb has also been conducted and used to analyse and investigate human movement [22, 86, 202]. The development of the open-source musculoskeletal modelling software *OpenSim* [87] has facilitated the development and sharing within the biomechanics community of 3D biomechanical models. These models have predominantly been used to investigate gait [16, 17, 18, 19, 145, 146, 266]. In addition, jumping [17, 269], squatting [75] and lifting [98] have also been investigated.

Anatomically and physiologically accurate biomechanical models can be developed using *OpenSim* and it has therefore been predominantly employed to investigate novel movement tasks with clinical applications [286]. Examples include surgery planning [117] and the rehabilitation of movement-restricting neurological conditions like osteoarthritis and strokes [108, 117, 127, 200, 241]. Research has predominantly focused on injury prevention rather than performance when *OpenSim* has been used to investigate sporting applications [286]. Investigation has, however, been conducted to determine the contributions of various muscles' forces to propulsion in running [145, 146, 297].

Muscle-actuated biomechanical models require [259]:

1. a model of the skeletal system and corresponding dynamical equations;
2. a model of musculotendon actuation as a function of muscular activation;
3. musculoskeletal coupling described by musculotendon origins, insertions and pathways; and
4. a model of the dynamics between neural excitation and muscular activation.

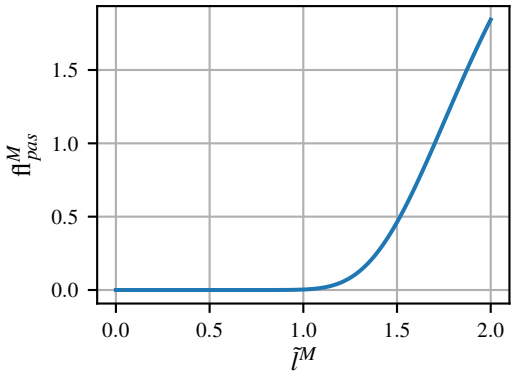
5.1.3 Musculotendon Architecture

Like many biological structures, muscle exhibits a highly hierarchical structure [345]. The fundamental force generating units that make up muscle are *sarcomeres* [182]. Sarcomeres are long, thin structures (typically 2 µm to 3 µm in length and 1 µm in diameter) comprised of fibrous protein filaments that slide over one another to produce force. Numerous sarcomeres attached in series make up a *muscle fibre* [69], the smallest scale longitudinal unit in a muscle. Parallel bundles of muscle fibres form *muscle fascicles*, with a muscle consisting of numerous fascicles arranged in parallel. As muscle fibres often run the full length of a muscle, fibre lengths within a single muscle are usually equivalent. However, fibre lengths between muscles can vary significantly as a result of muscle size [345]. The *physiological cross-sectional area* (PCSA) of a muscle can be estimated by measuring its volume and dividing this by the average fibre length.

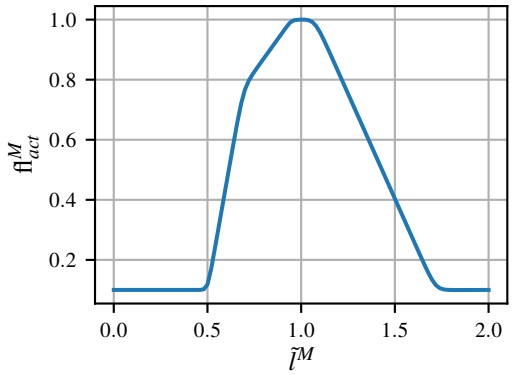
Muscle fibres are connected to *aponeuroses* [345]. An aponeurosis is itself connected to tendinous tissue, which is in turn attached to the skeleton. Aponeuroses exhibit the same properties as the tendinous tissue to which they are attached [270, 276]. Therefore, the tendinous tissue and aponeurosis can be considered as a single entity tendon [345]. A tendon transmits the force produced by its muscle to the skeleton. Therefore, muscle and tendon must be considered as a single unit, the musculotendon.

5.1.4 Musculotendon Properties

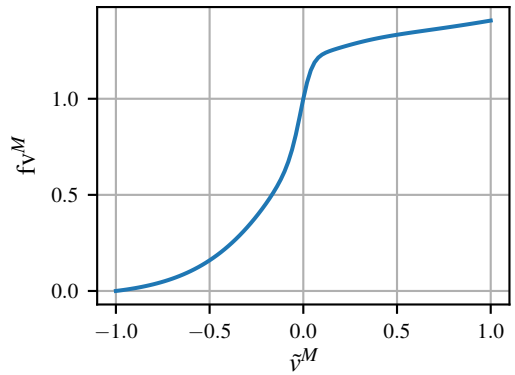
To develop an accurate musculotendon model, it is important to understand the properties of muscle and tendon so that such a model accurately replicates these. Muscle fibres exhibit both passive and active properties as a consequence of its architecture and its ability to contract and produce force. These properties depend on a muscle's length and its shortening velocity. Conversely, tendons only exhibit passive properties as they are not able to contract.



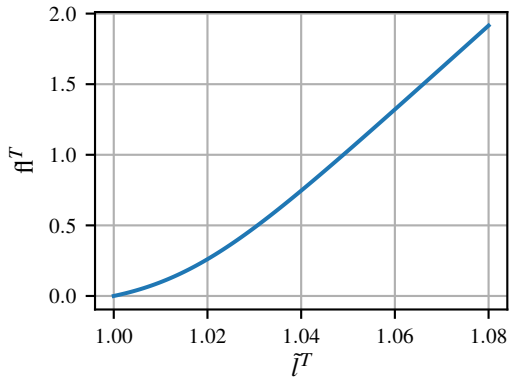
(a) Passive muscle fibre force-length $\mathrm{fl}^M_{\mathrm{pas}}\left(\tilde{l}^M\right)$.



(b) Active muscle fibre force-length $\mathrm{fl}^M_{\mathrm{act}}\left(\tilde{l}^M\right)$.



(c) Muscle fibre force-velocity $\mathrm{fv}^M\left(\tilde{v}^M\right)$.



(d) Tendon force-length $\mathrm{fl}^T\left(\tilde{l}^T\right)$.

Figure 5.1: Examples of typical musculotendon force-length and force-velocity characteristics, adapted from [242]. $\tilde{l}^T$ denotes normalised tendon length, $\tilde{l}^M$ denotes normalised muscle fibre length, and $\tilde{v}^M$ denotes normalised muscle fibre shortening velocity.

Muscle Passive Force-Length

The passive properties of static muscle can be measured using an isolated specimen by stretching it to a number of constant lengths, with no stimulation, and measuring the resulting force [135, 275]. The muscle fibre length l^M at which passive muscle develops force is known as the optimal muscle fibre length l^M_{opt}. When stretched to lengths beyond l^M_{opt} the passive force increases exponentially due to inter-fibre elasticity (caused by the protein, titin) [224]. Figure 5.1a shows this passive force-length relationship $\mathrm{fl}^M_{\mathrm{pas}}\left(\tilde{l}^M\right)$ for a typical muscle.

Muscle Active Force-Length

A further component to the isometric force-length function of muscle $\mathrm{fl}^M_{\mathrm{act}}\left(\tilde{l}^M\right)$ occurs when the muscle tissue is activated by a neural stimulus, known as *activation*. Fully activated muscle also develops a steady force when under isometric conditions. The difference between this and $\mathrm{fl}^M_{\mathrm{pas}}\left(\tilde{l}^M\right)$ is known as active muscle force $\mathrm{fl}^M_{\mathrm{act}}\left(\tilde{l}^M\right)$. Active muscle force is generally produced in the region $0.5l^M_{\mathrm{opt}} < l^M < 1.5l^M_{\mathrm{opt}}$ [135, 275]. At $l^M = l^M_{\mathrm{opt}}$ muscle force F^M also reaches its peak active value F^M_{max} [135, 275].

It is possible for muscle to be stimulated to a level lower than full excitation if not all fibres are active, or if the muscle is stimulated by a low frequency pulse train [345]. In this case the $\mathrm{fl}^M\left(\tilde{l}^M\right)$ can be considered as a scaled version of the difference between $\mathrm{fl}^M_{\mathrm{pas}}\left(\tilde{l}^M\right)$ and $\mathrm{fl}^M_{\mathrm{act}}\left(\tilde{l}^M\right)$ [152, 327]. This scaling argument is predicated on the fact that the forces produced by multiple fibres in the same muscle sum in parallel [183]. Figure 5.1b shows $\mathrm{fl}^M_{\mathrm{act}}\left(\tilde{l}^M\right)$ for a typical muscle.

Muscle Force-Velocity

Muscle can only produce tensile force. Muscle length shortens when contracting concentrically and thus does useful work. During a concentric contraction, F^M is weaker than for an equivalent isometric contraction [162]. As shortening velocity v^M increases, the magnitude of F^M continues to fall. At $l^M = l^M_{\mathrm{opt}}$ there exists a maximum contractile velocity v^M_{max} at which the muscle cannot produce any tensile force even when fully activated [162]. Muscle can contract eccentrically to resist lengthening. Eccentric contractions are stronger than concentric ones and therefore, during lengthening, if the muscle is fully activated, it is able to produce a force $F^M > F^M_{\mathrm{max}}$ [162]. A typical muscle force-velocity relationship $\mathrm{fv}^M\left(\tilde{v}^M\right)$ is shown in fig. 5.1c.

$\mathrm{fv}^M\left(\tilde{v}^M\right)$ is dependent on l^M and activation in addition to v^M. However, experiments have found that assuming a constant relationship between l^M and activation has negligible effect on the results of a muscle coordination analysis [24, 152, 345].

Due to the specific shape of $\mathrm{fv}^M\left(\tilde{v}^M\right)$, the point at which a contracting muscle can deliver maximum power output occurs at approximately $0.3v^M_{\mathrm{max}}$ [162]. Therefore, for a task like cycling that requires a net power input to the system in order to generate propulsion, concentric contractions constitute the majority of muscular activity. As frictional loses in joints and tendons are minimal [345], if energy (in the form of kinetic and potential energy of limb segments) needs to be dissipated, this is

done by eccentric contractions. The greater the stretching force a muscle is subject to during lengthening, the faster it will lengthen [186, 192, 230]. However, the maximum tensile force a muscle can be subjected to is between 1.1 and 1.8 $F^M_{\max}$ [186, 192, 230], so care must be taken to not exceed this in order to avoid catastrophic injury.

Tendon Force-Length

Tendon consists of two components: internal tendon (or aponeurosis) and external tendon (which exists outside the muscle). Experimental data suggests that the strain experienced by internal and external tendon is the same and therefore it is justified to consider both components as a single entity with the same material properties [270, 276]. The length below which a tendon cannot produce any force is known as the tendon slack length l^T_{slack}. When tendon is stretched beyond this it generates a nonlinear passive elastic contractile force (fig. 5.1d).

Pennation

The fibres in muscle may either be orientated parallel to the tendon (parallel-fibred muscle) or at an acute angle (pennate muscle). This subtended angle is known as the pennation angle α. Parallel-fibred and pennate muscle are equivalent when $\alpha = 0$. The more pennated a muscle is, the shorter its average fibre length and slower its shortening velocity [237]. However, pennate muscles are generally stronger than parallel-fibred muscles of the same volume because their architecture allows a greater number of muscle fibres to contract simultaneously [237].

Fibre Type

Muscle fibres can also be of different types, namely slow-twitch (type I) and fast-twitch (type II). Fast-twitch muscle fibres can develop greater force and do so faster than their slow-twitch counterpart, albeit with worse resistance to fatigue [69]. Muscles with predominantly type II fibres exhibit higher maximum and optimal shortening velocities than their type I counterparts, and thus are up to five times more powerful [315]. Therefore, for performance during a maximal task, a high proportion of type II fibres is desirable. Conversely, a high proportion of type I fibres is desirable for endurance tasks. Different muscles contain different relative proportions of the two main fibre types and as such exhibit different force-producing and fatigue-resistance properties. Therefore, when modelling real musculotendons, it is

important to parameterise the musculotendon models so that they are representative of their counterpart muscles' properties [108, 318, 319].

History Dependence

In addition to dependence on length and velocity, muscle force is also influenced by time history. Several studies have shown that prior active shortening of muscle reduces the amount of force it can produce relative to if it had undergone no prior shortening (*shortening-induced force-depression*) [2, 158]. Conversely, a muscle that has actively lengthened will be able to produce a higher force than if it had just maintained the same length (stretch-induced force-enhancement) [2, 159].

5.1.5 Musculotendon Dynamics

Second-Order Systems Models

Second-order systems models assume that muscle is a simple force generator and do not attempt to model the elastic characteristics [162, 328]. These assumptions allow the muscle-joint system to be treated purely as a second-order model relating some input (usually activation level) to joint angle, joint angular velocity and joint angular acceleration. Although this allows the system to be described by a second-order *ordinary differential equation* (ODE) [11], these assumptions do have significant shortcomings:

1. no velocity dependence (which has clearly been demonstrated for muscle [162]) is considered; and
2. model parameters are highly task and range of motion dependent and need to be adjusted for different tasks.

This category of model is not recommended for musculoskeletal models and simulations of human movements [24, 328].

Hill-Type Lumped-Parameter Models

Hill-type lumped-parameter models (based on the model proposed in [162]) accurately describe the phenomenological characteristics of muscle and are the most common model type in musculoskeletal models used to simulate human movement [150,

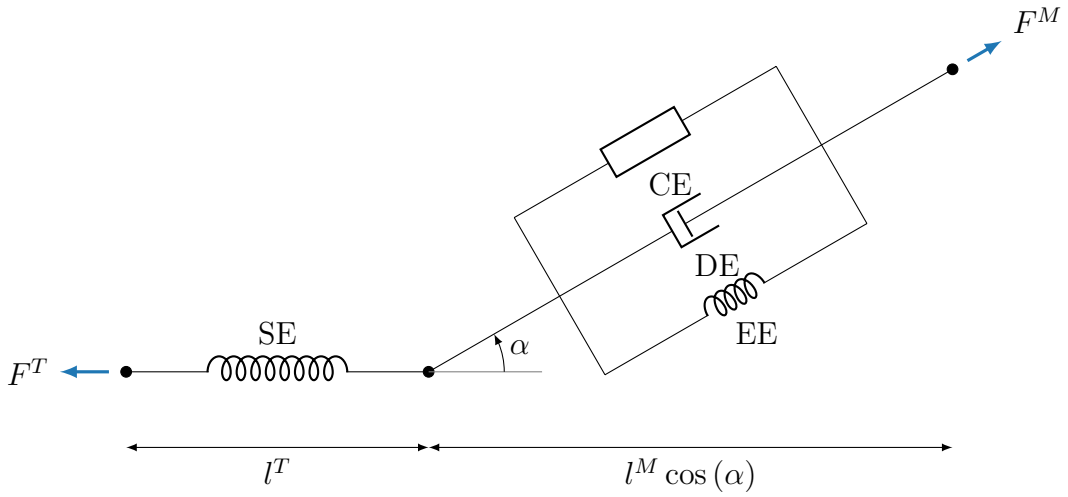


Figure 5.2: Schematic representation of a Hill-type musculotendon model. SE denotes the series elastic element, CE denotes the contractile element, DE denotes the parallel damping element, and EE denotes the parallel elastic element. l^T denotes the tendon length, l^M denotes the muscle fibre length, α denotes the pennation angle, F^T denotes the tendon force, and F^M denotes the muscle force.

242, 316, 328, 345]. These models typically consist of a force-producing contractile element CE in parallel with an elastic element EE and damping element DE to represent the muscle [242, 316]. This parallel arrangement is connected to a further elastic element in series SE to represent the tendon (fig. 5.2). With the exception of DE, all of the basic elements in Hill-type models are inherently nonlinear (fig. 5.1):

1. the properties of CE are given by $\mathrm{fl}^M_{\mathrm{act}}\left(\tilde{l}^M\right)$ and $\mathrm{fv}^M\left(\tilde{v}^M\right)$;
2. the properties of EE are given by $\mathrm{fl}^M_{\mathrm{pas}}\left(\tilde{l}^M\right)$; and
3. the properties of SE are given by $\mathrm{fl}^T\left(\tilde{l}^T\right)$.

These characteristics are scaled such that

$$\tilde{l}^M = \frac{l^M}{l^M_{\mathrm{opt}}} \tag{5.1}$$

$$\tilde{v}^M = \frac{v^M}{v^M_{\mathrm{max}}} \tag{5.2}$$

$$\tilde{l}^T = \frac{l^T}{l^T_{\mathrm{slack}}}, \tag{5.3}$$

where $\tilde{l}^M$ is nondimensional muscle length, $\tilde{v}^M$ is nondimensional muscle shortening velocity, and $\tilde{l}^T$ is nondimensional tendon length. At any instantaneous time, the length of the musculotendon actuator l^{MT} is

$$l^{MT} = l^T + l^M \cos(\alpha) \tag{5.4}$$

and the muscle force is therefore given by

$$F^M = F^M_{\max}\left(a\,\mathrm{fl}^M_{\mathrm{act}}\left(\tilde{l}^M\right)\mathrm{fv}^M\left(\tilde{v}^M\right) + \mathrm{fl}^M_{\mathrm{pas}}\left(\tilde{l}^M\right) + \beta\tilde{v}^M\right), \tag{5.5}$$

where a is muscular activation (section 5.1.6).

The parameters l^T_{slack}, l^M_{opt}, $F^M_{\max}$ and α_{opt} must be defined for a musculotendon model, numerous data sets for which have been published [22, 73, 89, 202]. β is the coefficient of damping that describes the linear damper DE, with $\beta = 0.1\,\mathrm{s\,m^{-1}}$ having been suggested [242]. Though strong damping has not been experimentally observed [167], inclusion of DE seems reasonable due to the high water content of muscle [320]. It has also been stated that DE improves the numerical conditioning of the Hill-type model [242].

$v^M_{\max} = 10 l^M_{\mathrm{opt}}\mathrm{s}^{-1}$ is typically assumed in simulation studies [16, 17, 252, 261, 274, 279] as this represents the summed effect of slow-, intermediate-, and fast-twitch muscle fibres [345]. However, no study to date has examined this assumption and the sensitivity of simulations to $v^M_{\max}$. Attempts have been made to better describe the influence of varying proportions of fast- and slow-twitch muscle fibres on contraction dynamics, however these have seen little validation or widespread use in simulation studies [44].

Tendon is usually modelled as a nonlinear elastic spring [89, 173, 345], although linear force-length relationships have also been used [15, 16, 261] to represent the contractile force produced by the tendon as it is stretched beyond l^T_{slack}. The value of l^T_{slack} can have a large influence on the characteristics of the modelled musculotendon actuator as it will determine the magnitude of the peak force developed and the joint angle at which peak force occurs [89, 173, 345]. Thus the ratio between l^T_{slack} and l^M_{opt} is important in musculoskeletal models.

A rigid tendon assumption ($v^T = 0$) allows l^M to be calculated as a function of skeletal pose and has been made in simulation studies to improve computational performance [85, 241]. This assumption has been investigated and it was found that for short, stiff tendons computation times were significantly improved (by 2 to 54 times relative to an elastic tendon) without introducing additional error [242]. For longer and less-stiff tendons, the errors in muscle force values produced by the rigid tendon model were approximately twice as large as those produced by the elastic tendon models [242]. The characteristic force-length and force-velocity functions published by [242] (which are used by *OpenSim* [87]) were adapted to ensure they remained first- and second-order continuous [85]. This adaptation made the musculotendon model more suitable for inclusion in *optimal control problems* (OCPs) where smooth derivative functions are required for efficient and reliable convergence [36, 282].

Phenomenological models of history dependent effects which accurately model shortening-induced force-depression and stretch-induced force-enhancement have been published [235, 236]. Simulation of jumping showed that including the phenomenological models of history dependent effects produced a more realistic countermovement jump [236]. However, others stated that history dependent effects have little impact on natural movements [64] and it is therefore unclear whether the inclusion of such models is important when simulating human movements like cycling.

Huxley-Based Distributed-Parameter Models

Huxley-based distributed-parameter models (based on the work of [182]) attempt to identify and model the physiological principles behind the contractile mechanism rather than modelling phenomenological properties [97, 149, 163, 329]. *Partial differential equations* (PDEs) are used to describe the process of muscle fibres sliding over one another in order to produce contractile force [163, 329]. Huxley-based models are derived from biological principles and can therefore explain a full range of muscle lengthening phenomena. However, they are mathematically complicated and include many parameters that cannot be easily determined experimentally [328]. As such, Huxley-based models have not seen use in simulations of human movement [242].

Passive Joint Structures

Passive joint structures which contribute to joint stiffness (such as ligaments, cartilage, and menisci) are generally not included in performance-based biomechanical models, as within normal ranges of motion their effect can be modelled in combination with the elastic properties of muscle [289, 303]. Passive torques provided by ligaments, if modelled, are represented by functions which exponentially increase in magnitude towards the extremes of the joint's range of motion [16, 24, 155, 226, 342]. Cartilage and menisci act to decrease joint force transmission and therefore only need to be modelled if this is the investigation's focus [302].

5.1.6 Activation Dynamics

Excitation-Activation Coupling

Muscle force production is the combination of muscle contraction dynamics and activation dynamics, with activation dynamics being the transformation of neural excitation into muscular activation [345]. The processes of activation and relaxation cannot occur instantaneously as they involve the conversion of an electrical signal from the nervous system into a chemical signal [96]. Although a second-order, critically damped filter models the delay between changes in neural excitation e and changes in muscular activation a very accurately [31], it is usually modelled as a first-order process [16, 251, 261, 274, 307, 328, 345]. The first-order assumption is valid because it accurately describes the rate-limiting diffusion of calcium ions [152, 345] and preference for first-order equations has prevailed because it reduces simulation times [345].

All models of activation dynamics are similar implicit ODEs. These ODE represent the rate of change in activation $\frac{da}{dt}$ as a function of a, e, and time constants for activation τ_{act} and deactivation τ_{deact} [57, 150, 242, 326]. Deactivation occurs at a much slower rate than activation [253]. $10\,\text{ms} \leq \tau_{\text{act}} \leq 20\,\text{ms}$ and $20\,\text{ms} \leq \tau_{\text{deact}} \leq 200\,\text{ms}$ are typically used in simulation studies [16, 57, 85, 242, 251, 261, 274, 307, 328, 345].

A fundamental feature of many of the first-order equations used to describe activation dynamics is that discontinuities occur at the transition between stimulation and relaxation [57, 242]. This is problematic if such a model is to be used in an OCP as smooth functions and first- and second-derivatives are important for efficient and reliable convergence [282]. Amendments to the equations provided by [242] have been proposed [85]. These amendments involved a hyperbolic tangent function to smooth this transition between activation and deactivation, and make the activation dynamics models better conditioned for OCPs.

Rate of Force Development

It is worth noting that studies into the rate of force development during maximal exertions have shown that the time between zero and maximum force can range from 40 ms to 200 ms [1, 65], slower than the times for τ_{act} and τ_{deact} given above. This is because the presence of an elastic tendon introduces an additional source of delay to force development, whereby an increase in muscle force initially acts to increase tendon strain until muscle and tendon force equilibrium is reached. Numerous

simulation studies have concluded that activation dynamics are the most important factor in determining the optimal neural excitation timings for movement tasks such as cycling [253, 279, 280, 307]. Therefore, if a rigid tendon assumption is made (and thus the additional delay in force generation is not intrinsic to the model) then the magnitudes of τ_{act} and τ_{deact} should be carefully considered.

5.1.7 Musculotendon Pathways

The numerous muscles present in the human body act to articulate it through contraction. The forces produced by muscle contractions are transferred to the skeleton through the tendons at either end of the musculotendon, attached to the skeleton at the musculotendon's origin and insertion [345]. It is important to be able to calculate a musculotendon's length and shortening velocity as functions of skeletal pose and kinematics because these quantities are required to evaluate the equations of musculotendon dynamics [85, 242, 316]. Musculotendon actuators are assumed to originate from, and insert into, the skeleton at single points [16, 84, 150, 173, 251, 261, 274, 279, 307, 332]. However, when a muscle's physical origin or insertion spans across a large area of bone, it is sometimes separated into multiple components [89, 248].

In between their origin and insertion, musculotendons can follow complex paths, wrapping around skeletal features and other musculotendons [345]. While simple straight-line methods have been implemented to model the pathway of the muscle between origin and insertion, these have been shown not to produce meaningful results when a muscle wraps around bone or another muscle [123, 184]. Consequently, being able to accurately model such paths is important for the development of realistic musculoskeletal models [123].

Obstacle-Set Method

The obstacle-set method is more accurate than straight-line methods as it describes a musculotendon's pathway using a series of points that represent its cross-sectional centroid [123, 184]. Via-points, connected by straight segments, accurately describe a musculotendon's path even when skeletal articulation causes joint angles to change [62, 89, 123]. The via-points are fixed relative to specific skeletal locations and take into account musculotendon wrapping. While via-points may be taken to be single points [87], others have extended the obstacle-set method to include wrapping around regular-shaped 3D rigid bodies, such as spheres and cylinders [123]. When wrapping surfaces such as these are involved, the obstacle-set method typi-

cally assumes that the musculotendon pathway is made up of a series of joined linear and circular arc segments. In the obstacle-set method, via-points can be considered either *active* or *inactive* as functions of relevant joint angles [123], which allows the musculotendon pathway to vary as a function of skeletal pose. The obstacle-set method has become the standard methodology for describing musculotendon pathways in biomechanical models [87].

Published datasets of full musculoskeletal and musculotendon geometry (produced by medical imaging and cadaver dissection) facilitate the implementation of the obstacle-set method [22, 73, 89, 202]. While these data sets are only representative of the individual from which they were produced, scaling techniques can be used to tailor them for subject specific models. Many non-invasive, image-based scaling methods (such as bone surface morphing [278], muscle volume registration [73], and musculotendon path identification [73]) have been developed in recent years to assist with this [73]. Furthermore, numerous studies have concluded that use of these image based methods provide an efficient and accurate means of producing subject-specific musculoskeletal models [21, 48, 73].

Pathway Approximation and Simplification

Using obstacle-set musculotendon pathways in OCPs can present two problems. Firstly, pathways that include multiple via-points or wrapping surfaces can be expensive to describe analytically as part of a system's *equations of motion* (EoMs), leading to prohibitively computationally expensive OCP functions [108]. Secondly, obstacle-set pathways that include via-points that switch between active and inactive at different joint angles require the introduction of conditional statements into computational implementations which are not suited for OCPs [36]. To circumvent these issues, it has been suggested that polynomial approximations of musculotendon pathways can be used [58, 108]. In this approach, data for musculotendon lengths and moment arms as functions of joint angles is produced by analysing geometric musculotendon pathways across a large number of skeletal poses. Polynomial approximations to these are then produced using stepwise regression, increasing the order of the approximation polynomial until a desired accuracy threshold is met [58]. Approximations to musculotendon shortening velocities can either be produced following a similar approach but by also including a range of joint angular velocities in the dataset, or by differentiating the polynomial fit to musculotendon length. To date, there has been no study formally investigating the accuracy and efficiency of such an approach.

5.2 Research Objectives

Section 1.3 laid out the objective of developing and critically evaluating a highly performant, easy-to-use, open-source software package capable of formulating and solving musculoskeletal predictive OCPs. This will be done by adding musculoskeletal modelling functionality to the *Biomechanics Predictive Simulation Toolkit* (BPST), while leveraging the capabilities of the other BPST packages.

From the analysis and review of past work in section 5.1, a number of limitations and constraints associated with the current software provision in this area were identified. To address these, and meet the overall objective above, the following sub-objectives are laid out:

- investigate current limitations in musculoskeletal modelling when applied to OCPs with the purpose of developing approaches and solutions that can be incorporated into a software package, including:
  - the functions used to describe musculotendon properties;
  - the formulation of musculotendon dynamics as part of an OCP; and
  - efficient musculotendon pathway approximation;
- enable users to efficiently construct biomechanical models using a high-level of abstraction (current software provision for this has not been developed with predictive simulation as a core functionality);
- enable users without an expertise in biomechanics to derive the musculoskeletal EoMs governing their modelled biomechanical system (by placing OCP functionality at the centre of design creates opportunities for more efficient and reliable modelling methodologies to be explored);
- enable the construction of OCPs involving modelled biomechanical systems while seeking to minimise complexity for the user (the current software provision in this area focuses on inverse dynamics rather than predictive simulation);
- support the formulation of OCPs using a variety of formulations of musculotendon dynamics (to enable the rigorous comparison of such formulations and their implications on performance when solving predictive simulations);
- efficiently model and approximate musculotendon pathways without compromising accuracy (there has been no study to date investigating the implications of approximate musculotendon pathways on OCP performance and solution accuracy);

- validate the developed software against previously published predictive simulation results.

5.3 Software Implementation: *Pyomechanics*

In this section, the development of a biomechanical modelling package, with primary focus on biomechanical predictive simulation, is described. The package is called *Pyomechanics*, with its name derived from *Python* and biomechanics. *Pyomechanics* is a biomechanics extension to *Pynamics* (section 4.3) and as such has also been developed to be an open-source software package written in *Python*.

5.3.1 Overview

As outlined in section 5.1, biomechanical modelling is dependent on multibody dynamics, musculotendon dynamics, activation dynamics and musculoskeletal coupling. *Pyomechanics* leverages *Pynamics* to provide multibody dynamics functionality. It does this by thinly wrapping *Pynamics*, exposing all of the functionality of *Pynamics* to the user. *Pyomechanics* also extends *Pynamics* to provide biomechanics-specific modelling and OCP setup.

As *Pyomechanics* is an extension to *Pynamics*, it possesses the same structure, also having `model`, `form`, `ocp`, `sim` and `viz` modules (section 4.3.1). The main extension occurs in the `model` module, in which additional biomechanics-specific classes are added to the component library. Firstly, there are the `Musculotendon` classes, which subclass *Pynamics*' `Actuator` class (section 5.3.2). These enable the creation of musculoskeletal models that account for musculotendon and activation dynamics. Secondly, there is the `ObstacleSetPathway` class, a subclass of `Pathway` (section 5.3.6), which allows nonlinear musculotendon pathways to be modelled.

5.3.2 Musculotendon Dynamics

Biological muscle is complex and many simplifications and assumptions are made when it is modelled [242]. *Pyomechanics* implements Hill-type musculotendon models [162, 345]. These modelled musculotendons are assumed to be massless, frictionless, extensible strings which are able to contract and produce force. As illustrated in fig. 5.3, modelled musculotendon is made up of a tendon in series with a set of muscle fibres, which are responsible for producing the contractile force. The tendon

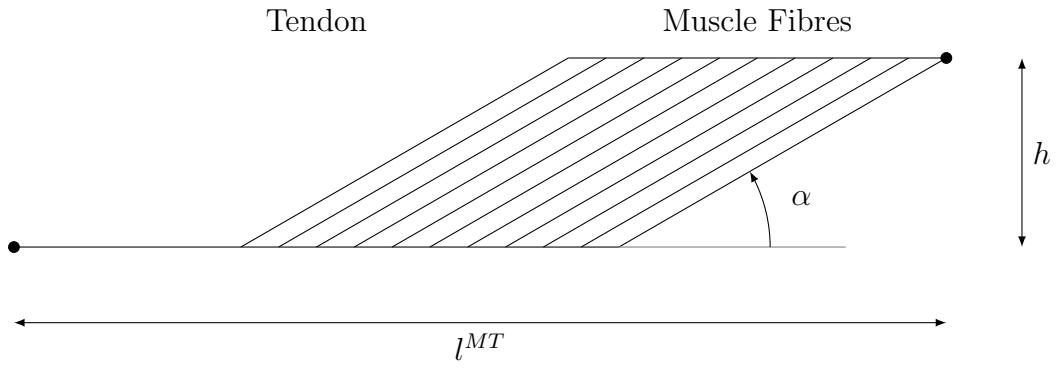


Figure 5.3: Simplified diagram of a modelled musculotendon. l^{MT} denotes the musculotendon length, α denotes the pennation angle, and h denotes the muscle fibre height.

has length l^T and the muscle fibres have length l^M, arranged at the pennation angle α to the tendon (fig. 5.3). Thus the length of the musculotendon l^{MT} is

$$l^{MT} = l^T + l^M \cos(\alpha), \tag{5.6}$$

noting that l^{MT} and v^{MT} are functions of skeletal pose and kinematics respectively. The muscle fibres are assumed to be constant height, such that

$$l^M \sin(\alpha) = l^M_{\text{opt}} \sin(\alpha_{\text{opt}}), \tag{5.7}$$

in order to approximate the constant-volume property of biological muscle [242]. Resolving the tendon and fibre forces of fig. 5.3 horizontally gives

$$F^T = F^M \cos(\alpha), \tag{5.8}$$

where F^T, the tendon force, is the force imparted onto the skeleton at the musculotendon's origin and insertion. *Pyomechanics* uses dimensionless quantities to describe the properties of muscle and tendon (see section 5.3.3). As such, these dimensionless properties are scaled using five parameters in order to represent a specific musculotendon. These parameters are tendon slack length l^T_{slack}, optimal fibre length l^M_{opt}, maximum fibre velocity v^M_{max}, maximum isometric fibre force F^M_{max}, and pennation angle at optimal fibre length α_{opt}, which allow nondimensional quantities describing the musculotendon to be defined. Nondimensional quantities describing the musculotendon include normalised tendon length

$$\tilde{l}^T = \frac{l^T}{l^T_{\text{slack}}}, \tag{5.9}$$

normalised tendon velocity

$$\tilde{v}^T = \frac{v^T}{l^T_{\text{slack}}}, \tag{5.10}$$

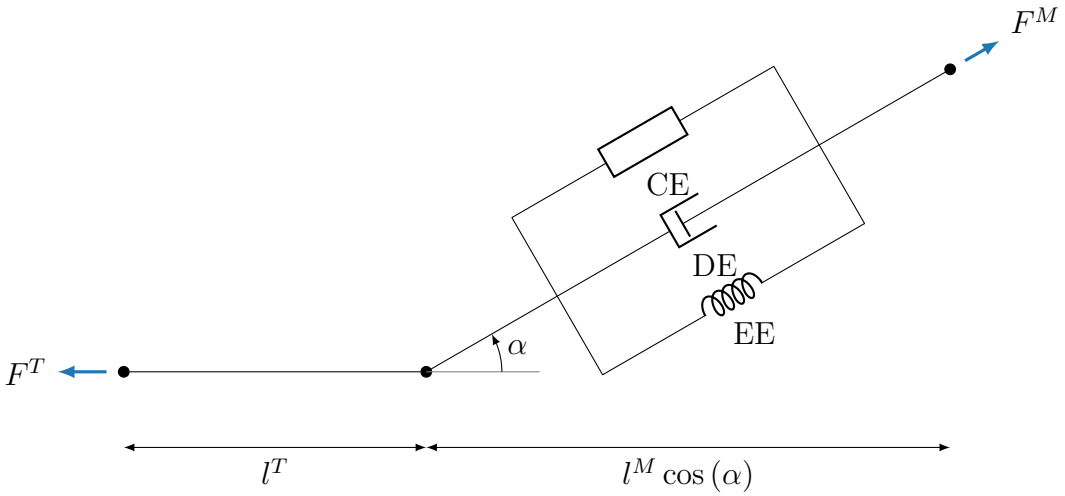


Figure 5.4: Schematic of the damped rigid tendon musculotendon model used in *Pyomechanics*. CE denotes the contractile element, DE denotes the parallel damping element, and EE denotes the parallel elastic element. l^T denotes the tendon length, l^M denotes the muscle fibre length, α denotes the pennation angle, F^T denotes the tendon force, and F^M denotes the muscle fibre force.

normalised tendon force

$$\tilde{F}^T = \frac{F^T}{F^M_{\text{max}}}\,, \tag{5.11}$$

normalised muscle fibre length

$$\tilde{l}^M = \frac{l^M}{l^M_{\text{opt}}}\,, \tag{5.12}$$

normalised muscle fibre velocity

$$\tilde{v}^M = \frac{v^M}{v^M_{\text{max}}} \tag{5.13}$$

and normalised muscle fibre force

$$\tilde{F}^M = \frac{F^M}{F^M_{\text{max}}}\,. \tag{5.14}$$

Pyomechanics provides both rigid and elastic tendon musculotendon models, which are discussed below.

Rigid Tendon

Figure 5.4 shows a schematic of the rigid tendon musculotendon model used by *Pyomechanics*. In the rigid tendon model $l^T = l^T_{\text{slack}}$ and $v^T = 0$. As such, l^M and v^M can be calculated directly as functions of skeletal pose and kinematics due to the tendon being fixed-length. This means that

$$l^M = \sqrt{\left(l^{MT} - l^T_{\text{slack}}\right)^2 + \left(l^M_{\text{opt}} \sin\left(\alpha_{\text{opt}}\right)\right)}\,, \tag{5.15}$$

found by eliminating α from eqs. (5.6) and (5.7). Furthermore,

$$v^M = v^{MT}\left(\frac{l^{MT} - l^T_{\text{slack}}}{l^M}\right), \tag{5.16}$$

found by differentiating eqs. (5.6) and (5.7) with respect to time, eliminating $\frac{d\alpha}{dt}$ from the resulting pair of first-order ODEs and eliminating $\cos(\alpha)$ using eq. (5.6). The force produced by the muscle fibres is a combination of the active force produced by the contractile element and the passive forces produced by the passive parallel elastic and damping elements (fig. 5.4). Using this model, muscle force

$$F^M = F^M_{\text{max}}\left(a\,\text{fl}^M_{\text{act}}\left(\tilde{l}^M\right)\text{fv}^M\left(\tilde{v}^M\right) + \text{fl}^M_{\text{pas}}\left(\tilde{l}^M\right) + \beta\tilde{v}^M\right), \tag{5.17}$$

where β is an additional property of muscle fibre, the muscle fibre damping coefficient. The terms $\text{fl}^M_{\text{pas}}\left(\tilde{l}^M\right)$, $\text{fl}^M_{\text{act}}\left(\tilde{l}^M\right)$ and $\text{fv}^M\left(\tilde{v}^M\right)$ correspond to the normalised passive force-length, normalised active force-length and normalised force-velocity properties of muscle fibre respectively, all of which are discussed in section 5.3.3. Given the muscle activation a (see section 5.3.4), l^{MT} and v^{MT}, the tendon force F^T can be calculated by

1. determining l^M using eq. (5.15);
2. normalising l^M using eq. (5.12) to give $\tilde{l}^M$;
3. determining v^M using eq. (5.16);
4. normalising v^M using eq. (5.13) to give $\tilde{v}^M$;
5. evaluating $\text{fl}^M_{\text{pas}}\left(\tilde{l}^M\right)$ using $\tilde{l}^M$ and an appropriate muscle fibre passive force-length equation from section 5.3.3;
6. evaluating $\text{fl}^M_{\text{act}}\left(\tilde{l}^M\right)$ using $\tilde{l}^M$ and an appropriate muscle fibre active force-length equation from section 5.3.3;
7. evaluating $\text{fv}^M\left(\tilde{v}^M\right)$ using $\tilde{v}^M$ and an appropriate muscle fibre force-velocity equation from section 5.3.3;
8. evaluate eq. (5.17) to give F^M; and
9. determining F^T using eq. (5.8) with eq. (5.7) to eliminate $\cos(\alpha)$.

An example of how a rigid tendon musculotendon can be instantiated using the *Pyomechanics application programming interface* (API) is given in section 5.3.5.

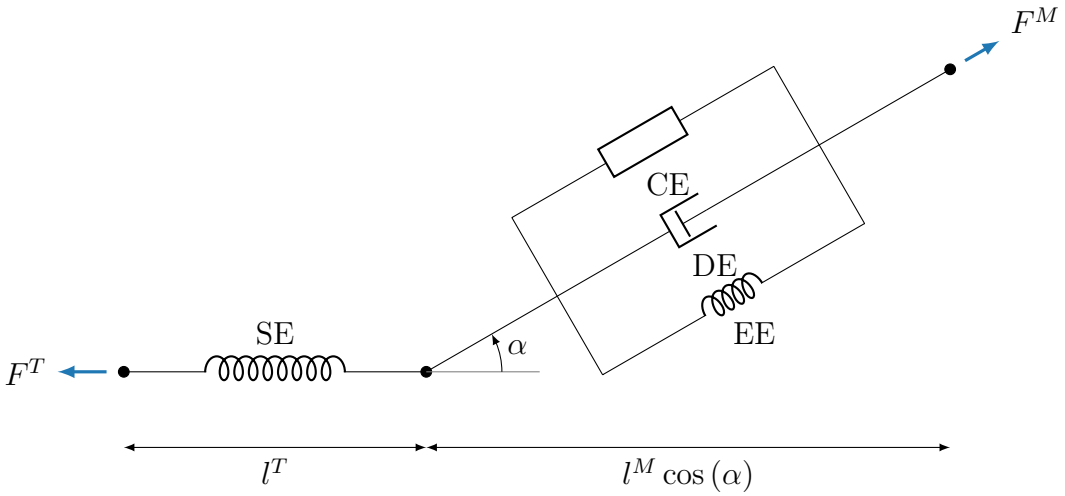


Figure 5.5: Schematic of the damped elastic tendon equilibrium musculotendon model used in *Pyomechanics*. SE denotes the series elastic element, CE denotes the contractile element, DE denotes the parallel damping element, and EE denotes the parallel elastic element. l^T denotes the tendon length, l^M denotes the muscle fibre length, α denotes the pennation angle, F^T denotes the tendon force, and F^M denotes the muscle fibre force.

Elastic Tendon

Pyomechanics also supports the modelling of musculotendons with elastic tendons using a damped equilibrium model [242], a schematic of which is shown in fig. 5.5. Unlike the rigid tendon model, the elastic tendon model models the tendon as a nonlinear spring in series with the parallel contractile, elastic and damping elements representing the muscle fibres. The tendon force F^T is given by

$$F^T = F^M_{\text{max}} \text{fl}^T \left(\tilde{l}^T \right) , \tag{5.18}$$

where $\text{fl}^T \left(\tilde{l}^T \right)$ corresponds to the normalised force-length relationship of tendon, which is discussed in section 5.3.3. The muscle fibres are modelled the same way in both the rigid and elastic tendon models; the muscle fibre force is also given using eq. (5.17). The five eqs. (5.6) to (5.8), (5.17) and (5.18) define a system of nonlinear simultaneous equations with the five unknowns F^T, F^M, l^T, l^M and α. Unlike in the rigid tendon model, where F^T can be stated in explicit form by finding an algebraic solution to these equations using the rigid tendon assumption, the elastic tendon system cannot be solved unless an additional state variable is defined. Furthermore, unless the damping coefficient $\beta = 0$, the muscle dynamics cannot be expressed explicitly. Instead, they must be expressed implicitly by introducing a further additional variable alongside the already-required additional state variable. *Pyomechanics* supports two distinct ways of resolving this system of five equations such that F^T can be calculated; by treating either $\tilde{l}^M$ or $\tilde{F}^T$ as a state variable.

Moreover, depending on the value of β, *Pyomechanics* will also formulate either explicit or implicit musculotendon dynamics.

In the first formulation, $\tilde{l}^M$ is treated as a state variable. With $\tilde{l}^M$ known, enough additional information about the musculotendon's configuration is provided such that F^T can determined by

1. denormalising $\tilde{l}^M$ using a rearrangement of eq. (5.12) to give l^M;
2. determining l^T as
$$l^T = l^{MT} - \sqrt{\left(l^M\right)^2 - \left(l^M_{\text{opt}} \sin\left(\alpha_{\text{opt}}\right)\right)^2}\,, \tag{5.19}$$
produced by combining eqs. (5.6) and (5.7) by eliminating $\cos(\alpha)$ using the Pythagorean trigonometric identity $\cos^2(\alpha) + \sin^2(\alpha) = 1$ and rearranging;
3. normalising l^T using eq. (5.9) to give $\tilde{l}^T$;
4. determining $\cos(\alpha)$ as
$$\cos(\alpha) = \frac{l^{MT} - l^T}{l^M}\,, \tag{5.20}$$
a rearrangement of eq. (5.6);
5. evaluating $\text{fl}^T\left(\tilde{l}^T\right)$ using $\tilde{l}^T$ and an appropriate tendon force-length equation from section 5.3.3; and
6. determining F^T using eq. (5.18).

As *Pyomechanics* formulates OCPs involving these musculotendon models, the time derivatives of any state variables are also required so that these can be used as state equations in any OCP formulation. If $\beta = 0$, it is possible to state the musculotendon dynamics explicitly. Therefore, *Pyomechanics* also determines $\frac{d}{dt}\left(\tilde{l}^M\right)$ in the case of this first formulation by

7. determining F^M as
$$F^M = \frac{F^T}{\cos(\alpha)}\,, \tag{5.21}$$
a rearrangement of eq. (5.8);
8. normalising F^M using eq. (5.14) to give $\tilde{F}^M$;
9. evaluating $\text{fv}^M\left(\tilde{v}^M\right)$ as
$$\text{fv}^M\left(\tilde{v}^M\right) = \frac{\tilde{F}^M - \text{fl}^M_{\text{pas}}\left(\tilde{l}^M\right)}{a\,\text{fl}^M_{\text{act}}\left(\tilde{l}^M\right)}\,, \tag{5.22}$$
a rearrangement of eq. (5.17);

10. determining $\tilde{v}^M$ using $\text{fv}^M\left(\tilde{v}^M\right)$ from the previously evaluated eq. (5.22) and an appropriate inverse muscle fibre force-velocity equation from section 5.3.3; and

11. evaluating $\frac{d}{dt}\left(\tilde{l}^M\right)$ as

$$\frac{d}{dt}\left(\tilde{l}^M\right) = \frac{1}{l^M_{\text{opt}}}\frac{d}{dt}\left(l^M\right) = \frac{1}{l^M_{\text{opt}}}v^M = \frac{v^M_{\text{max}}}{l^M_{\text{opt}}}\tilde{v}^M\,. \tag{5.23}$$

If $\beta > 0$, or if requested by the user, *Pyomechanics* will formulate the musculotendon dynamics implicitly. It does this by introducing $\tilde{v}^M$ as a control variable such that eq. (5.23) can be evaluated directly. Steps 1 to 6 remain the same, with the musculotendon dynamics then being imposed by introducing

$$\left(a\text{fl}^M_{\text{act}}\left(\tilde{l}^M\right)\text{fv}^M\left(\tilde{v}^M\right) + \text{fl}^M_{\text{pas}}\left(\tilde{l}^M\right) + \beta\tilde{v}^M\right)\cos\left(\alpha\right) - \text{fl}^T\left(\tilde{l}^T\right) = 0 \tag{5.24}$$

as an equality path constraint.

In the second formulation, $\tilde{F}^T$ is treated as a state variable. With $\tilde{F}^T$ known, F^T can be determined with ease by

1. denormalising $\tilde{F}^T$ using a rearrangement of eq. (5.11) to give F^T.

As before, the time derivative $\frac{d}{dt}\left(\tilde{F}^T\right)$ of the introduced state variable $\tilde{F}^T$ is also required so that *Pyomechanics* can formulate OCPs. The required state equation involving $\frac{d}{dt}\left(\tilde{F}^T\right)$ is determined in *Pyomechanics* by

2. recognising that $\tilde{F}^T = \text{fl}^T\left(\tilde{l}^T\right)$;

3. determining $\tilde{l}^T$ using $\text{fl}^T\left(\tilde{l}^T\right)$ and an appropriate inverse tendon force-length equation from section 5.3.3;

4. denormalising $\tilde{l}^T$ using a rearrangement of eq. (5.9) to give l^T;

5. determining l^M as

$$l^M = \sqrt{\left(l^{MT} - l^T\right)^2 + \left(l^M_{\text{opt}}\sin\left(\alpha_{\text{opt}}\right)\right)^2}\,, \tag{5.25}$$

   produced by combining eqs. (5.6) and (5.7) by eliminating $\cos\left(\alpha\right)$ using the Pythagorean trigonometric identity $\cos^2\left(\alpha\right) + \sin^2\left(\alpha\right) = 1$ and rearranging;

6. normalising l^M using eq. (5.12) to give $\tilde{l}^M$;

7. determining $\cos\left(\alpha\right)$ using eq. (5.20);

8. determining F^M using eq. (5.21);
9. normalising F^M using eq. (5.14) to give $\tilde{F}^M$;
10. evaluating $\text{fv}^M\left(\tilde{v}^M\right)$ using eq. (5.22);
11. determining $\tilde{v}^M$ using $\text{fv}^M\left(\tilde{v}^M\right)$ the previously evaluated eq. (5.22) and an appropriate inverse muscle fibre force-velocity equation from section 5.3.3;
12. determining v^T as
$$v^T = v^{MT} - \frac{v^M}{\cos\left(\alpha\right)}, \tag{5.26}$$
given by rearranging and taking the first-order time derivative of eq. (5.6);
13. normalising v^T using eq. (5.10) to give $\tilde{v}^T$; and
14. evaluating $\frac{d}{dt}\left(\tilde{F}^T\right)$ using the first-order time derivative of an appropriate tendon force-length equation from section 5.3.3.

Again, *Pyomechanics* will formulate implicit musculotendon dynamics by introducing $\tilde{v}^M$ as a control variable at the user's request or if $\beta > 0$. Steps 1 to 7 and 12 to 14 are followed as before while steps 8 to 11 are simply removed as $\tilde{v}^M$ is known. As for the formulation with $\tilde{l}^M$ as the state variable, the musculotendon dynamics are enforced using eq. (5.24) as an equality path constraint. Examples of how elastic tendon musculotendons can be instantiated in *Pyomechanics* using both of the state formulations described above are given towards the end of section 5.3.5.

5.3.3 Musculotendon Curves

Section 5.3.2 showed that the incorporation of eq. (5.17) in the musculotendon dynamics requires $\text{fl}^M_{\text{pas}}\left(\tilde{l}^M\right)$, $\text{fl}^M_{\text{act}}\left(\tilde{l}^M\right)$ and $\text{fv}^M\left(\tilde{v}^M\right)$, dimensionless functions representing the passive force-length, active force-length and force-velocity characteristics of muscle fibre respectively. This subsection details the mathematical expressions that *Pyomechanics* uses to represent these musculotendon characteristics. As *Pyomechanics* formulates OCPs, the mathematical expressions representing these characteristic curves are required to be at least second-order continuous so that they are compatible with the direct collocation method implemented by *Pycollo* (section 2.6, [36, 43]). Additional numerical properties are also required for some of these characteristic curves.

For each characteristic musculotendon curve, *Pyomechanics* provides two implementations. *Pyomechanics* implements the OCP-suitable musculotendon curves

from [85], termed the *De Groote curves*. *Pyomechanics* also implements a set of characteristic musculotendon curves that have been produced as best-fits to the non-OCP-suitable characteristic musculotendon curves from [242], termed the *Millard curves*. The curves from [242] are not directly suitable for use in OCPs as they are constructed using quintic Bézier splines. While these meet the continuity requirements, such interpolating functions are not readily differentiable by OCP software (section 2.6). It was desirable for *Pyomechanics* to provide OCP-suitable implementations of musculotendon curves similar to those from [242] as these curves are the most widely used musculotendon curves in biomechanical modelling and differ substantially to the curves of [85] in places (see below).

Tendon Force-Length

The dimensionless tendon force-length characteristic $\mathrm{fl}^T\left(\tilde{l}^T\right)$ is required as part of elastic tendon modelling. Any mathematical expression describing $\mathrm{fl}^T\left(\tilde{l}^T\right)$ is required to be third-order continuous as when $\tilde{F}^T$ is used as a musculotendon state, the first-order time derivative of $\mathrm{fl}^T\left(\tilde{l}^T\right)$ is required to define the state equation involving $\frac{d}{dt}\left(\tilde{F}^T\right)$. Furthermore, the function needs be algebraically invertible so that $\tilde{l}^T$ can be readily determined for a specific value of $\mathrm{fl}^T\left(\tilde{l}^T\right)$, as is specifically required by the explicit elastic tendon musculotendon dynamics when $\tilde{F}^T$ is treated as a state. The De Groote $\mathrm{fl}^T\left(\tilde{l}^T\right)$ curve is given by the equation [85]

$$\mathrm{fl}^T_{\mathrm{DeGroote}}\left(\tilde{l}^T\right) = d_1 \exp\left(d_4\left(\tilde{l}^T - d_2\right)\right) - d_3\,, \tag{5.27}$$

with the constants d_i for $i = 1, \ldots, 4$ given in table 5.1. Note that the value of d_4 given in table 5.1 has been adjusted from that in [85] to ensure that $\mathrm{fl}^T_{\mathrm{DeGroote}}\left(\tilde{l}^T\right)$ goes through the point $\mathrm{fl}^T_{\mathrm{DeGroote}}\left(\tilde{l}^T = 1.049\right) = 1.0$. This is because tendon force-length properties are commonly parameterised such that they produce one normalised force unit at a strain of 1.049 [87]. Equation (5.27) can be inverted to give

$$\tilde{l}^T = \frac{\log\left(\frac{\mathrm{fl}^T\left(\tilde{l}^T\right)+d_3}{d_1}\right)}{d_4} + d_2 \tag{5.28}$$

and differentiated with respect to time to give

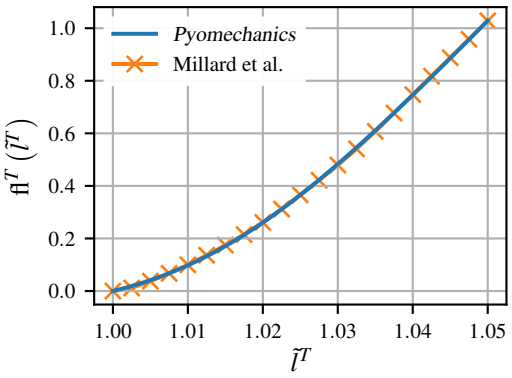
$$\frac{d}{dt}\left(\tilde{F}^T\right) = \frac{d}{dt}\left(\mathrm{fl}^T_{\mathrm{DeGroote}}\left(\tilde{l}^T\right)\right) = d_1 d_4 \exp\left(d_4\left(\tilde{l}^T - d_2\right)\right)\tilde{v}^T\,. \tag{5.29}$$

The OCP-suitable Millard $\mathrm{fl}^T\left(\tilde{l}^T\right)$ curve is given by the equation

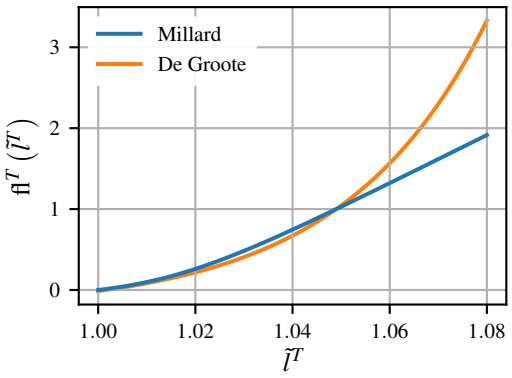
$$\mathrm{fl}^T_{\mathrm{Millard}}\left(\tilde{l}^T\right) = c_3 \log\left(1 + \exp\left(c_1\left(\tilde{l}^T - c_2\right)\right)\right) - c_4\,, \tag{5.30}$$

$\mathrm{fl}^T_{\text{DeGroote}}\left(\tilde{l}^T\right)$		$\mathrm{fl}^T_{\text{Millard}}\left(\tilde{l}^T\right)$	
d_1	0.200	c_1	91.192
d_2	0.995	c_2	1.0130
d_3	0.250	c_3	0.327 74
d_4	33.937	c_4	0.087 201

Table 5.1: Constants for parameterising the $\mathrm{fl}^T_{\text{DeGroote}}\left(\tilde{l}^T\right)$ and $\mathrm{fl}^T_{\text{Millard}}\left(\tilde{l}^T\right)$ curves representing tendon force-length characteristics in *Pyomechanics*.



(a) Comparison of *Pyomechanics*'s $\mathrm{fl}^T_{\text{Millard}}\left(\tilde{l}^T\right)$ curve and $\mathrm{fl}^T\left(\tilde{l}^T\right)$ from [242].



(b) Comparison of *Pyomechanics*'s $\mathrm{fl}^T_{\text{Millard}}\left(\tilde{l}^T\right)$ and $\mathrm{fl}^T_{\text{DeGroote}}\left(\tilde{l}^T\right)$ curves.

Figure 5.6: Tendon force-length characteristics in *Pyomechanics*.

with the constants c_i for $i = 1, \ldots, 4$ also given in table 5.1. The tendon force-length curve in [242] is represented by a straight line segment $\mathrm{fl}^T\left(\tilde{l}^T < 1.0\right) = 0$, another straight line $\tilde{l}^T > 1.037$ and a curved Bézier spline in between. Therefore, a *softplus* function [95] was chosen to represent $\mathrm{fl}^T_{\text{Millard}}\left(\tilde{l}^T\right)$ because this allowed the tight curve in the region of $\tilde{l}^T = 1.0$ to be accurately fitted to, while also enabling the linear region at larger tendon strains to be represented well. Furthermore, the softplus function of eq. (5.30) can be readily algebraically inverted to give

$$\tilde{l}^T = \frac{\log\left(\exp\left(\frac{\left(\mathrm{fl}^T\left(\tilde{l}^T\right)+c_4\right)}{c_3}\right)-1\right)}{c_1} + c_2\,. \tag{5.31}$$

Finally, eq. (5.30) can be differentiated with respect to time to give the state equation

$$\frac{d}{dt}\left(\tilde{F}^T\right) = \frac{d}{dt}\left(\mathrm{fl}^T_{\text{Millard}}\left(\tilde{l}^T\right)\right) = \frac{c_1 c_3 \exp\left(c_1\left(\tilde{l}^T - c_2\right)\right)}{\exp\left(c_1\left(\tilde{l}^T - c_2\right)+1\right)}\tilde{v}^T\,. \tag{5.32}$$

The $\mathrm{fl}^T_{\text{Millard}}\left(\tilde{l}^T\right)$ constants (table 5.1) were determined using a nonlinear least squares fit to sample data for the tendon force-length curve of [242]. The *OpenSim*

implementation of the musculotendon curves from [242] were used to generate the sample data. A set of 101 linearly-spaced values for $1.0 \leq \tilde{l}^T \leq 1.05$ were generated and used to evaluate the `TendonForceLengthCurve` associated with the `Millard-2012EquilibriumMuscle` class in *OpenSim*. The sample data is shown in fig. 5.6a (only a subset of sample data is included to increase clarity of the figure). Lagrange multipliers of 1 were used at each sample point except for $\tilde{l}^T = 1.0$ and $\tilde{l}^T = 1.049$ where Lagrange multipliers of 0.01 were used. This was to ensure that the fit was forced through the points $(1.0, 0.0)$ and $(1.049, 1.0)$. The fit of table 5.1 and fig. 5.6a resulted in a maximum error of 0.00765 and *root-mean-square error* (RMSE) of 0.00304. Note that these quoted errors need not be normalised as the tendon force-length curves are themselves normalised such that the function evaluates to $[0, 1]$ over the range of points of interest. A comparison of the De Groote and Millard tendon force-length curves implemented in *Pyomechanics* is shown in fig. 5.6b. Note that the De Groote curve is more compliant for strains $1.0 \leq \tilde{l}^T \leq 1.049$ but increases in stiffness exponentially above this range in comparison to the Millard curve.

Muscle Fibre Passive Force-Length

The parallel elastic element representing the passive elastic properties of the muscle fibres, illustrated in both figs. 5.4 and 5.5, is also modelled as a nonlinear spring. The De Groote $\mathrm{fl}^M_{\mathrm{pas}}\left(\tilde{l}^M\right)$ curve is given by the equation [85]

$$\mathrm{fl}^M_{\mathrm{pas,DeGroote}}\left(\tilde{l}^M\right) = \frac{\exp\left(\frac{d_1\left(\tilde{l}^M - 1\right)}{d_2}\right) - 1}{\exp\left(d_1\right) - 1}, \tag{5.33}$$

with the constants d_1 and d_2 given in table 5.2. The OCP-suitable Millard $\mathrm{fl}^M_{\mathrm{pas}}\left(\tilde{l}^M\right)$ curve is given by the equation

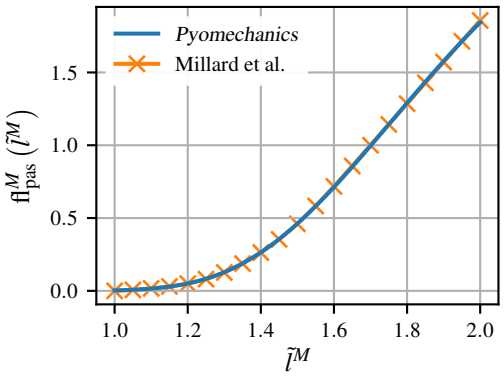
$$\mathrm{fl}^M_{\mathrm{pas,Millard}}\left(\tilde{l}^M\right) = c_1 \exp\left(c_2 \left(\exp\left(\tilde{l}^M - c_4\right)\right)^{c_3}\right), \tag{5.34}$$

with the constants c_i for $i = 1, \ldots, 4$ also given in table 5.2. Like the tendon force-length curve, the muscle fibre passive force-length curve in [242] is represented by a straight line segment $\mathrm{fl}^M_{\mathrm{pas}}\left(\tilde{l}^M < 1.0\right) = 0$, another straight line $\tilde{l}^T > 1.7$ and a curved Bézier spline in between. A generalised logistic function [288] was used to represent $\mathrm{fl}^M_{\mathrm{pas,Millard}}\left(\tilde{l}^M\right)$ as this type of function has been used by other authors to model OCP-suitable muscle fibre passive force-length curves [108].

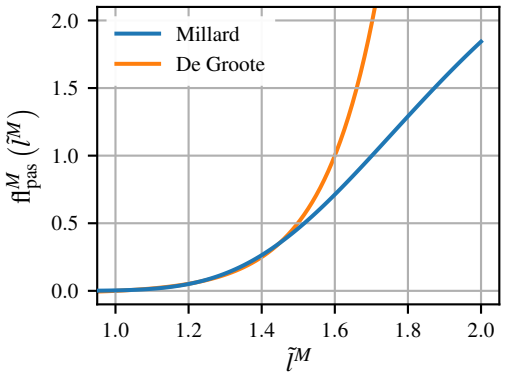
The $\mathrm{fl}^M_{\mathrm{pas,Millard}}\left(\tilde{l}^M\right)$ constants (table 5.2) were again determined using a non-linear least squares fit to sample data for the muscle fibre passive force-length curve of [242]. A similar approach to that described above was used, in which 101 linearly-spaced values for $0.0 \leq \tilde{l}^M \leq 2.0$ were generated and used to evaluate the `Fiber-ForceLengthCurve` associated with the `Millard2012EquilibriumMuscle` class in

$\mathrm{fl}^M_{\mathrm{pas,DeGroote}}\left(\tilde{l}^M\right)$		$\mathrm{fl}^M_{\mathrm{pas,Millard}}\left(\tilde{l}^M\right)$	
d_1	4.0	c_1	3.1436
d_2	0.60	c_2	−76.908
		c_3	−2.5581
		c_4	0.056 965

Table 5.2: Constants for parameterising the $\mathrm{fl}^M_{\mathrm{pas,DeGroote}}\left(\tilde{l}^M\right)$ and $\mathrm{fl}^M_{\mathrm{pas,Millard}}\left(\tilde{l}^M\right)$ curves representing muscle fibre passive force-length characteristics in *Pyomechanics*.



(a) Comparison of *Pyomechanics*'s $\mathrm{fl}^M_{\mathrm{pas,Millard}}\left(\tilde{l}^M\right)$ curve and $\mathrm{fl}^M_{\mathrm{pas}}\left(\tilde{l}^M\right)$ from [242].



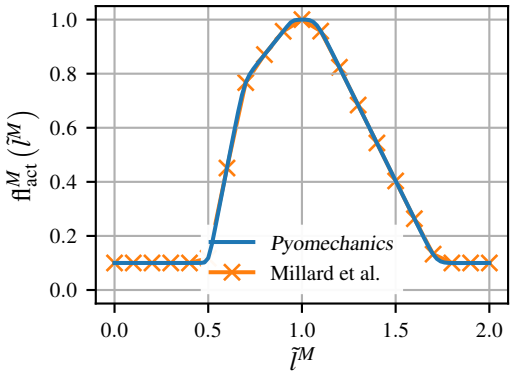
(b) Comparison of *Pyomechanics*'s $\mathrm{fl}^M_{\mathrm{pas,Millard}}\left(\tilde{l}^M\right)$ and $\mathrm{fl}^M_{\mathrm{pas,DeGroote}}\left(\tilde{l}^M\right)$ curves.

Figure 5.7: Muscle fibre passive force-length characteristics in *Pyomechanics*.

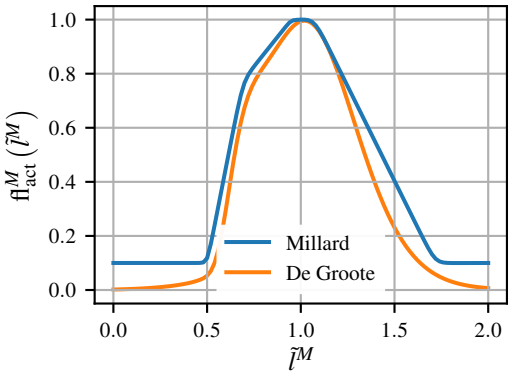
OpenSim. A subset of the sample data is shown in fig. 5.7a. The fit of table 5.2 and fig. 5.7a resulted in a maximum error of 0.0137 and RMSE of 0.00295. Again, these errors need not be normalised as the fibre passive force-length curves are themselves normalised such that the function evaluates to $[0, 1]$ over the range of points of interest. A comparison of the De Groote and Millard muscle fibre passive force-length curves implemented in *Pyomechanics* is shown in fig. 5.7b. Both curves exhibit the property that for $\tilde{l}^M < 0$, normalised passive muscle fibre force is approximately zero. Similar to the tendon force-length curves, the De Groote curve is again marginally more compliant for small muscle fibre strains greater than l^M_{opt} but increases in stiffness exponentially faster than the Millard curve for $\tilde{l}^M > 1.5$.

Muscle Fibre Active Force-Length

The contractile element in both figs. 5.4 and 5.5 includes the active force-length property of the muscle fibres. The De Groote $\mathrm{fl}^M_{\mathrm{act}}\left(\tilde{l}^M\right)$ curve is given by the equa-



(a) Comparison of *Pyomechanics*'s $\mathrm{fl}^M_{\mathrm{act,Millard}}\left(\tilde{l}^M\right)$ curve and $\mathrm{fl}^M_{\mathrm{act}}\left(\tilde{l}^M\right)$ from [242].



(b) Comparison of *Pyomechanics*'s $\mathrm{fl}^M_{\mathrm{act,Millard}}\left(\tilde{l}^M\right)$ and $\mathrm{fl}^M_{\mathrm{act,DeGroote}}\left(\tilde{l}^M\right)$ curves.

Figure 5.8: Muscle fibre active force-length characteristics in *Pyomechanics*.

tion [85]

$$\mathrm{fl}^M_{\mathrm{act,DeGroote}}\left(\tilde{l}^M\right) = \sum_{i=1}^{3}\left(d_{4i-3}\exp\left(-\frac{\left(\tilde{l}^M - d_{4i-2}\right)^2}{2\left(d_{4i-1} + d_{4i}\tilde{l}^M\right)}\right)\right), \tag{5.35}$$

a sum of three Gaussian functions, with the constants d_j for $j = 1, \ldots, 12$ given in table 5.3. The OCP-suitable Millard $\mathrm{fl}^M_{\mathrm{act}}\left(\tilde{l}^M\right)$ curve is given by the equation

$$\mathrm{fl}^M_{\mathrm{act,Millard}}\left(\tilde{l}^M\right) = c_1 + \sum_{i=1}^{5}\left(c_{3i+1}\log\left(1 + \exp\left(c_{3i-1}\left(\tilde{l}^M - c_{3i}\right)\right)\right)\right), \tag{5.36}$$

with the constants c_i for $i = 1, \ldots, 16$ also given in table 5.3. The muscle fibre active force-length curve in [242] is constructed using Bézier splines connecting six linear sections. The Millard curve was described using a sum of five offset and scaled softplus functions as this allowed the shape of the curve from [242] to be replicated with high accuracy while also ensuring that the function used was a continuous and differentiable algebraic function, as required by *Pycollo* and OCPs in general.

The `ActiveForceLengthCurve` associated with the `Millard2012EquilibriumMuscle` class in *OpenSim* was used to produce the sample data, a subset of which is show in fig. 5.8a. Lagrange multipliers of 1 were used at each sample point except for $\tilde{l}^M = 0.0$, $\tilde{l}^M = 1.0$ and $\tilde{l}^M = 2.0$ where Lagrange multipliers of 0.01 were used. This was to ensure that the fit was forced through the points $(0.0, 0.1)$, $(1.0, 1.0)$ and $(2.0, 0.1)$. The fit of table 5.3 and fig. 5.8a resulted in a maximum error of 0.00109 and RMSE of 0.000398, indicating that an accurate fit was achieved. A comparison of the De Groote and Millard muscle fibre active force-length curves implemented in *Pyomechanics* is shown in fig. 5.8b. Both active force-length curves go through the point $(1.0, 1.0)$. However, the De Groote curve exhibits a narrower

$\mathrm{fl}^M_{\mathrm{act,DeGroote}}\left(\tilde{l}^M\right)$		$\mathrm{fl}^M_{\mathrm{act,Millard}}\left(\tilde{l}^M\right)$	
d_1	0.815	d_1	0.1
d_2	1.055	d_2	128.21
d_3	0.162	d_3	0.500 65
d_4	0.063	d_4	0.027 576
d_5	0.433	d_5	71.680
d_6	0.717	d_6	0.692 02
d_7	−0.030	d_7	−0.037 232
d_8	0.200	d_8	136.52
d_9	0.100	d_9	0.947 33
d_{10}	1.000	d_{10}	−0.005 983 4
d_{11}	0.354	d_{11}	54.636
d_{12}	0.000	d_{12}	1.070 74
		d_{13}	−0.026 509
		d_{14}	56.000
		d_{15}	1.7177
		d_{16}	0.025 004

Table 5.3: Constants for parameterising the $\mathrm{fl}^M_{\mathrm{act,DeGroote}}\left(\tilde{l}^M\right)$ and $\mathrm{fl}^M_{\mathrm{act,Millard}}\left(\tilde{l}^M\right)$ curves representing muscle fibre active force-length characteristics in *Pyomechanics*.

peak and smoother curves. Furthermore, the Millard curve has a minimum value of 0.1, which it exhibits at the extremes of normalised fibre length, while the De Groote curve transitions gradually to a normalised value of 0.0 when normalised muscle fibre length is either 0.0 or 2.0.

Muscle Fibre Force-Velocity

The contractile element used to model muscle fibre's force-producing properties also exhibits a force-velocity relationship. Consequently, a dimensionless muscle fibre force-velocity curve $\mathrm{fv}^M\left(\tilde{v}^M\right)$ is required to model this. Like the tendon force-length characteristics, the inverse of any $\mathrm{fv}^M\left(\tilde{v}^M\right)$ curve is also required so that $\tilde{v}^M$ can be determined for a given value of $\mathrm{fv}^M\left(\tilde{v}^M\right)$. This is the case for any elastic tendon model, no matter whether $\tilde{l}^M$ or $\tilde{F}^T$ is used as the additional state variable.

The De Groote $\mathrm{fv}^M\left(\tilde{v}^M\right)$ curve is given by the equation [85]

$$\mathrm{fv}^M_{\mathrm{DeGroote}}\left(\tilde{v}^M\right) = d_1 \operatorname{arcsinh}\left(d_2\tilde{v}^M + d_3\right) + d_4\,, \tag{5.37}$$

with the constants d_i for $i = 1, \ldots, 4$ given in table 5.4. Equation (5.37) can be algebraically inverted to give

$$\tilde{v}^M = \frac{\sinh\left(\frac{\mathrm{fv}^M\left(\tilde{v}^M\right) - d_4}{d_1}\right) - d_3}{d_2}\,. \tag{5.38}$$

The shape of the $\mathrm{fv}^M\left(\tilde{v}^M\right)$ curve in [242] is such that it was not possible to construct an accurate approximation using an algebraically invertible function. Therefore, separate fits were produced for the Millard $\mathrm{fv}^M\left(\tilde{v}^M\right)$ curve and its inverse. This approach is valid provided that the pair of fitted curves exhibit a high level of accuracy to each other when one is numerically inverted. The OCP-suitable Millard $\mathrm{fv}^M\left(\tilde{v}^M\right)$ curve is given by the equation

$$\mathrm{fv}^M_{\mathrm{Millard}}\left(\tilde{v}^M\right) = c_1 + \sum_{i=1}^{4}\left(c_{3i+1}\log\left(1 + \exp\left(c_{3i-1}\left(\tilde{v}^M - c_{3i}\right)\right)\right)\right)\,, \tag{5.39}$$

with the constants c_i for $i = 1, \ldots, 13$ also given in table 5.4. Once again, a summation of offset and scaled softplus functions allowed the Bézier curve from [242] to be approximated accurately using an OCP-suitable function. The OCP-suitable Millard inverse $\mathrm{fv}^M\left(\tilde{v}^M\right)$ curve is given by the equation

$$\begin{aligned}\tilde{v}^M = \hat{c}_1 &+ \hat{c}_2\log\left(\hat{c}_3\left(\mathrm{fv}^M\left(\tilde{v}^M\right) - \hat{c}_4\right)\right)\\ &+ \hat{c}_5\log\left(1 + \exp\left(\hat{c}_6\left(\mathrm{fv}^M\left(\tilde{v}^M\right) - \hat{c}_7\right)\right)\right)\\ &+ \hat{c}_8\log\left(1 + \exp\left(\hat{c}_9\left(\mathrm{fv}^M\left(\tilde{v}^M\right) - \hat{c}_{10}\right)\right)\right)\,,\end{aligned} \tag{5.40}$$

with table 5.4 again giving the constants $\hat{c}_i$ for $i = 1, \ldots, 10$. The summation of a logarithmic function and a pair softplus functions was required to enable fitting to the shape of the inverse $\mathrm{fv}^M\left(\tilde{v}^M\right)$ curve from [242], with the logarithmic function primarily responsible for the shape of the Millard curve for small values of $\mathrm{fv}^M\left(\tilde{v}^M\right)$.

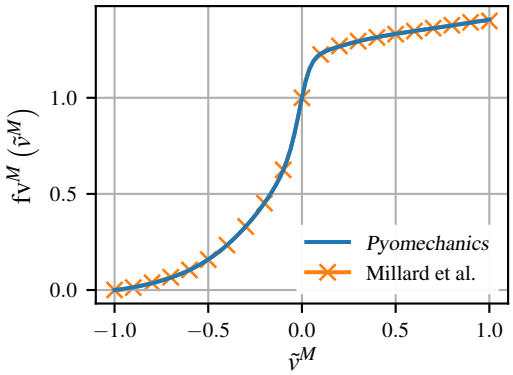
The `ForceVelocityCurve` associated with the `Millard2012EquilibriumMuscle` class in *OpenSim* was sampled to generate the sample data for fitting, a subset of which is shown in fig. 5.9a. Lagrange multipliers of 1 were used at each sample point except for $\tilde{v}^M = -1.0$ and $\tilde{v}^M = 0.0$ where Lagrange multipliers of 0.01 were used. This ensured that the Millard $\mathrm{fv}^M\left(\tilde{v}^M\right)$ curve goes through the pair of points $(-1.0, 0.0)$ and $(0.0, 1.0)$. The fit of table 5.4 and fig. 5.9a resulted in a maximum error of 0.00801 and RMSE of 0.00218. A comparison of the De Groote and Millard muscle fibre force-velocity curves implemented in *Pyomechanics* is shown in fig. 5.9b. Note that both curves pass through $(-1.0, 0.0)$ and $(0.0, 1.0)$ but that the De Groote

$\mathrm{fv}^M_{\mathrm{DeGroote}}\left(\tilde{v}^M\right)$		$\mathrm{fv}^M_{\mathrm{Millard}}\left(\tilde{v}^M\right)$			
d_1	-0.318	c_1	$-0.032\,110$	$\hat{c}_1$	$0.193\,81$
d_2	-8.149	c_2	1.0139	$\hat{c}_2$	$0.197\,08$
d_3	-0.374	c_3	3.7900	$\hat{c}_3$	$0.372\,19$
d_4	0.886	c_4	$-0.097\,403$	$\hat{c}_4$	$-0.021\,670$
		c_5	1.2590	$\hat{c}_5$	$-0.047\,561$
		c_6	31.897	$\hat{c}_6$	-10.141
		c_7	$-0.005\,725\,5$	$\hat{c}_7$	$0.504\,97$
		c_8	$-0.377\,95$	$\hat{c}_8$	$0.311\,07$
		c_9	5.5718	$\hat{c}_9$	25.586
		c_{10}	$0.190\,61$	$\hat{c}_{10}$	1.2873
		c_{11}	-1.2424		
		c_{12}	33.562		
		c_{13}	$0.005\,294\,5$		

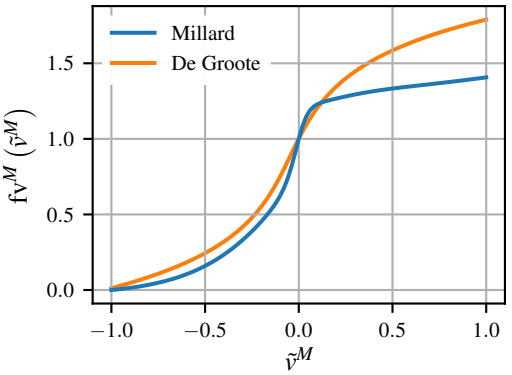
Table 5.4: Constants for parameterising the $\mathrm{fv}^M_{\mathrm{DeGroote}}\left(\tilde{v}^M\right)$ and $\mathrm{fv}^M_{\mathrm{Millard}}\left(\tilde{v}^M\right)$ curves representing muscle fibre force-velocity characteristics in *Pyomechanics*.

curve exhibits more gradual changes in $\mathrm{fv}^M\left(\tilde{v}^M\right)$ as $\tilde{v}^M$ changes than the Millard curve, as well as demonstrating a greater ability to generate eccentric force.

The sample data was inverted to provide the dataset for fitting the inverse Millard $\mathrm{fv}^M\left(\tilde{v}^M\right)$ curve, with a subset shown in fig. 5.10a. A similar approach using Lagrange multipliers was followed, except the points $(0.0, -1.0)$ and $(1.0, 0.0)$, the inverse of before, were used. The fit of table 5.4 and fig. 5.10a resulted in a maximum error of 0.0191 and RMSE of 0.00486. If however, only the central 90% of the inverse Millard curve is considered, then the maximum error was only 0.00792. This confirms that the fit is highly accurate across the range of values that correspond to the normal operating conditions of musculotendons. A comparison of the De Groote and Millard inverse muscle fibre force-velocity curves implemented in *Pyomechanics* is shown in fig. 5.10b. As expected from the $\mathrm{fv}^M\left(\tilde{v}^M\right)$ curves, the De Groote curve exhibits more gradual changes in value and a higher normalised eccentric force-velocity value for which the curve can be inverted.

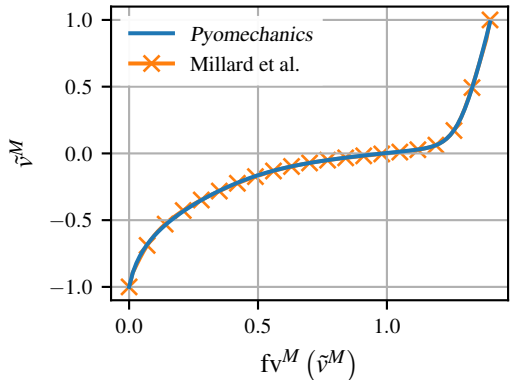


(a) Comparison of *Pyomechanics*'s $\mathrm{fv}^M_{\mathrm{Millard}}(\tilde{v}^M)$ curve and $\mathrm{fv}^M(\tilde{v}^M)$ from [242].

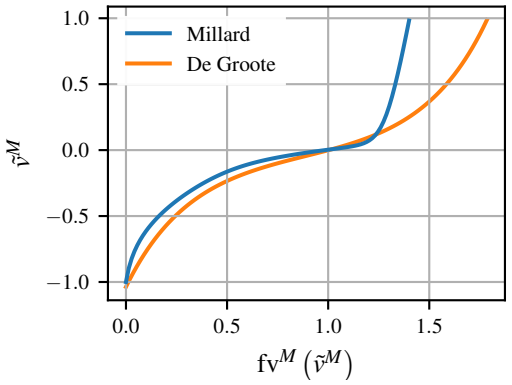


(b) Comparison of *Pyomechanics*'s $\mathrm{fv}^M_{\mathrm{Millard}}(\tilde{v}^M)$ and $\mathrm{fv}^M_{\mathrm{DeGroote}}(\tilde{v}^M)$ curves.

Figure 5.9: Muscle fibre force-velocity characteristics in *Pyomechanics*.



(a) Comparison of *Pyomechanics*'s inverse $\mathrm{fv}^M_{\mathrm{Millard}}(\tilde{v}^M)$ curve and inverse $\mathrm{fv}^M(\tilde{v}^M)$ from [242].



(b) Comparison of *Pyomechanics*'s inverse $\mathrm{fv}^M_{\mathrm{Millard}}(\tilde{v}^M)$ and inverse $\mathrm{fv}^M_{\mathrm{DeGroote}}(\tilde{v}^M)$ curves.

Figure 5.10: Inverse muscle fibre force-velocity characteristics in *Pyomechanics*.

5.3.4 Activation Dynamics

Activation dynamics describe the delay relationship between neural excitation signals and muscular activation. In neuromusculoskeletal modelling, excitation e is related to activation a by some form of ODE. *Pyomechanics* implements both zeroth- and first-order activation dynamics. In zeroth-order activation dynamics, the excitation is mapped directly to the activation

$$a = e \tag{5.41}$$

and as such is an algebraic equation not an ODE. Zeroth-order activation dynamics do not, therefore, introduce an additional state into OCP formulations, with eq. (5.41) instead being treated as an auxiliary substitution. They are, therefore, useful when it is desired that activation dynamics be ignored, typically in situations

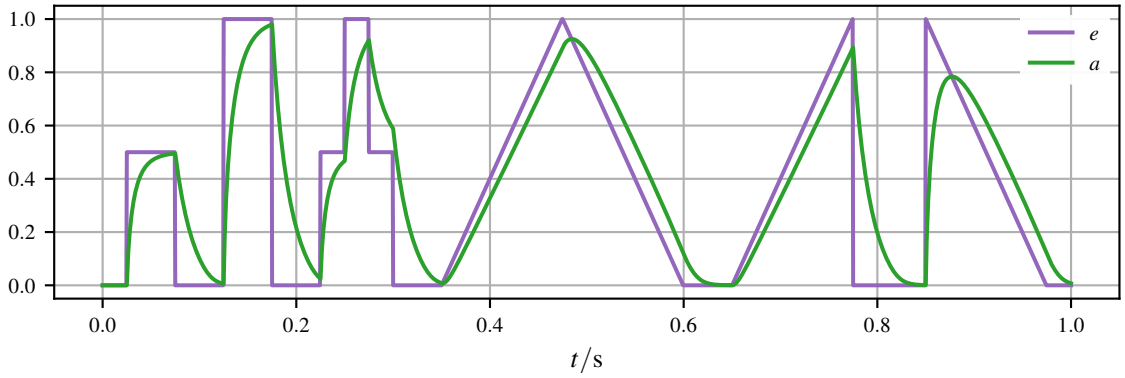


Figure 5.11: Relationship between excitation e and activation a subject to first-order activation dynamics from eq. (5.44) [85]. A set of arbitrary excitations and corresponding activations using default values of $\tau_{\text{act}} = 0.015\,\text{s}$ and $\tau_{\text{deact}} = 0.060\,\text{s}$ are shown.

where a user may want to simplify the equations governing a musculotendon in a large or complicated model [87].

First-order activation dynamics are the most prevalent form of activation dynamics and are implemented using (a slight rearrangement of) the equations from [85] (which are themselves based on [316, 326])

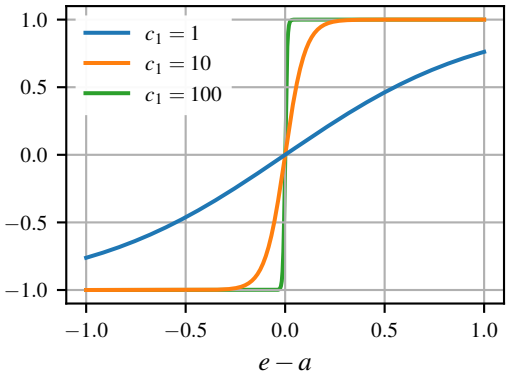
$$f_1 = \tanh\left(c_1\left(e - a\right)\right) \tag{5.42}$$

$$f_2 = 1 + 3a \tag{5.43}$$

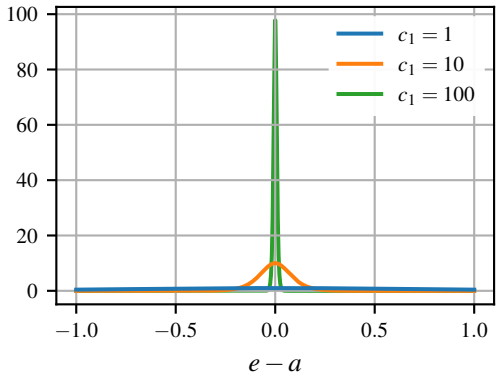
$$\frac{d}{dt}\left(a\right) = \left[\frac{1 + f_1}{\tau_{\text{act}} f_2} + \frac{f_2\left(1 - f_1\right)}{4\tau_{\text{deact}}}\right]\left(e - a\right) . \tag{5.44}$$

The action of eq. (5.44) on activation for an arbitrary excitation signal is shown in fig. 5.11. The activation dynamics constant $c_1 = 10$ has previously been proposed [85], however, *Pyomechanics* uses a default value of $c_1 = 100$ (see section 5.4.6). Figure 5.12 illustrates f_1 from eq. (5.44) and its derivatives for different values of c_1. It can be seen that while $c_1 = 10$ ensures that the magnitudes of the first- and second-order derivatives are kept small, the range over which the sigmoid switches is still a significant portion of possible values of $(e - a)$. On the other hand, $c_1 = 100$ results in rapid switching between the activation and deactivation components of eq. (5.44), at the expense of nonlinear and large derivatives in the region of $e - a = 0$.

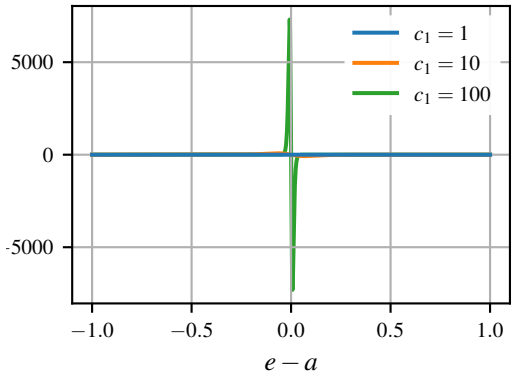
Pyomechanics implements an `Activation` *abstract base class* (ABC), and two specific subclasses: `ZerothOrderActivation` and `FirstOrderActivation`. An instance of one of these subclasses is created and owned by each `Musculotendon` instance during its initialisation. The `Activation` classes are responsible for specifying which quantities should be considered state variables, control variables and auxiliary substitutions; the state equations (if there are any); and suitable guesses



(a) $\tanh(c_1(e-a))$



(b) $\frac{d}{da}(\tanh(c_1(e-a)))$



(c) $\frac{d^2}{da^2}(\tanh(c_1(e-a)))$

Figure 5.12: Comparison of different values of c_1 within the hyperbolic tangent function used for sigmoidal smoothing of the activation dynamics equation in eq. (5.44).

and bounds (that can be overwritten by the user if required) for any OCP variables. For example, the `FirstOrderActivation` sets a as a state variable, e as a control variable, eq. (5.44) mapped to a as a state equation, and user-settable constants τ_{act} and τ_{deact} with default values of 0.015 s and 0.060 s respectively.

5.3.5 Musculotendon Implementations (`Musculotendon` Classes)

Pyomechanics implements two subclasses of `Musculotendon`. These are the `DeGroote-Musculotendon` and `MillardMusculotendon` classes, which implement the De Groote and Millard musculotendon curves (section 5.3.3) respectively. To illustrate the ease with which a biomechanical model involving musculotendons can be created in *Pyomechanics*, a simple code example (based on the supplementary documentation of [87]) is presented. In fig. 5.13a a planar model of an arm is created using the *Pyomechanics* API. The model created is illustrated in fig. 5.14a and consists of a humerus and radius, with the elbow modelled as a pin joint and the shoulder as a clamped joint. While classes such as `RigidBody`, `PinJoint` and `AttachmentPoint` from the

Pyomechanics namespace are used, these are actually unmodified *Pynamics* classes made available as part of the *Pyomechanics* API.

To add a musculotendon to a model, origin and insertion points on bodies need to be created as `AttachmentPoint` objects, as shown in fig. 5.13b. A musculotendon can be instantiated by passing these origin and insertion points as arguments to a `Musculotendon`. In fig. 5.13b this is done using the `MillardMusculotendon` class so that the instantiated musculotendon uses the Millard curves. To use the De Groote curves, a `DeGrooteMusculotendon` can instead be used (fig. 5.13c). Figure 5.13c also demonstrates the broad range of musculotendon properties that can be adjusted by the user, including F^M_{max}, l^T_{slack}, l^M_{opt}, v^M_{max}, α_{opt}, β, τ_{act} and τ_{deact}. It also shows how the variants of musculotendon dynamics discussed in sections 5.3.2 and 5.3.4 can be specified. These include using a rigid tendon model, whether $\tilde{l}^M$ or $\tilde{F}^T$ is used as the musculotendon state, and the order of the activation dynamics. The structure of the *Pyomechanics* API enables users to easily create custom biomechanical models using minimal lines of code. Furthermore, the breadth of available modelling options allows biomechanical models to be easily customised. This facilitates robust comparison and evaluation of different OCP formulations.

5.3.6 Musculotendon Pathways (`Pathway` Classes)

As outlined in section 5.1.7, it is important to be able to model nonlinear musculotendon pathways, as are encountered when musculotendons are required to wrap around bone or other musculotendons. This enables accurate musculotendon lengths and shortening velocities, as functions of the skeletal kinematics, to be determined. Parallel to the `LinearPathway` class implemented in *Pynamics*, *Pyomechanics* provides another subclass of `Pathway`; the `ObstacleSetPathway` class.

Obstacle-Set Pathways

The obstacle-set method adjusts a musculotendon's pathway between its origin and insertion by adding via-points. These via-points can be either points at which linear sections of the musculotendon's path connect, or wrapping surfaces along which a musculotendon's path smoothly wraps. The `ObstacleSetPathway` allows a user to define a nonlinear musculotendon pathway by adding these via-points and wrapping surfaces. This approach allows segmented pathways (as typically used in *OpenSim* [87]) to be modelled, as well as pathways that wrap along the surface of a cylindrical obstacle (as has been used in a multitude of past biomechanical modelling work [53, 54, 154, 251, 274, 344]).

```python
from pyomechanics import (Model, ClampedJoint, RigidBody, AttachmentPoint, PinJoint, MillardMusculotendon, DeGrooteMusculotendon)

arm = Model("arm")
shoulder = ClampedJoint("shoulder", model=arm, parent_attachment=arm.origin, axis="z", angle=-0.5*PI)
humerus = RigidBody("humerus", model=arm, parent_joint=shoulder, com_position_x=0.2, mass=1)
epicondyl = AttachmentPoint("epicondyl", model=arm, parent_body=humerus, position_x=0.2)
elbow = PinJoint("elbow", model=arm, parent_attachment=epicondyl, axis="z", min_angle=0, max_angle=PI)
radius = RigidBody("radius", model=arm, parent_joint=elbow, com_position_x=0.2, mass=1)
styloid = AttachmentPoint("styloid", model=arm, parent_body=radius, position_x=0.2)
```

(a) Creation of bodies and joints.

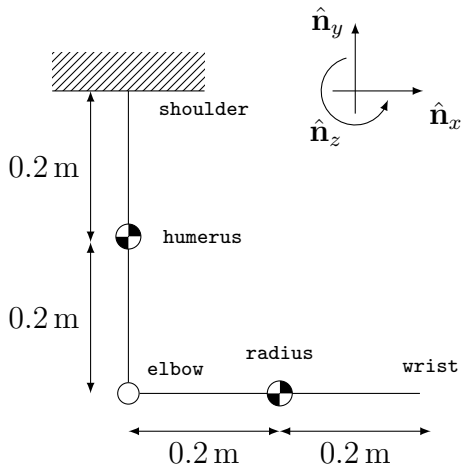
```python
origin = AttachmentPoint("origin", model=arm, parent_body=humerus, position_x=-0.12)
insertion = AttachmentPoint("insertion", model=arm, parent_body=radius, position_x=-0.08)
bicep = MillardMusculotendon("bicep", model=arm, origin=origin, insertion=insertion, peak_isometric_force=200, tendon_slack_length=0.32, optimal_fiber_length=0.36, rigid_tendon=True)
```

(b) Creation of an origin, an insertion and a musculotendon with Millard musculotendon curves.

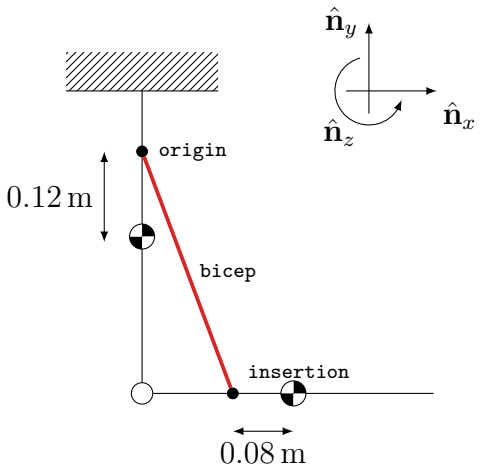
```python
bicep = DeGrooteMusculotendon("bicep", model=arm, origin=origin, insertion=insertion, peak_isometric_force=200, tendon_slack_length=0.32, optimal_fiber_length=0.36, maximal_fiber_velocity=3.6, optimal_pennation_angle=0, fiber_damping_coefficient=0.1, activation_time_constant=0.01, deactivation_time_constant=0.04, rigid_tendon=False, musculotendon_dynamics_state="l_M_tilde", activation_dynamics_order=1)
```

(c) Creation of a musculotendon with De Groote musculotendon curves and extended settings.

Figure 5.13: Code example of creating a simple arm model using the *Pyomechanics* API. The simple arm model consists of two rigid bodies, a humerus and a radius. The humerus is clamped at the shoulder such that it is vertical. The elbow is modelled as a pin joint with a constrained range of motion.



(a) Arm system created by the *Pyomechanics* code in fig. 5.13a.



(b) Arm system with musclotendon created by the *Pyomechanics* code in fig. 5.13b.

Figure 5.14: Illustration of the simple arm model created using the *Pyomechanics* code in fig. 5.13.

To create a nonlinear musculotendon pathway, an `ObstacleSetPathway` instance is created. This class holds information about a musculotendon's origin and insertion points. As such, an instance of this class can be passed with the `pathway` keyword argument when instantiating a `Musculotendon`. This can be done instead of explicitly passing `AttachmentPoint` objects to its `origin` and `insertion` arguments.

Via-Points

To create a segmented pathway, one or more `AttachmentPoint` objects can be associated with an `ObstacleSetPathway`, using its `via` argument on initialisation. This is illustrated in fig. 5.15a, in which a tricep musculotendon is added to the simple arm created in fig. 5.13a. This code results in a segmented musculotendon pathway with the via-points inserted in order between the origin and insertion as illustrated in fig. 5.16a.

Wrapping Surfaces

In order to allow users to create musculotendon pathways in which a musculotendon wraps along a surface, *Pyomechanics* provides the `WrappingCylinder` class. A `WrappingCylinder` is infinite in length, however, for a musculotendon to interact with it, a `Musculotendon` object's `pathway` must be explicitly linked to it. This is done by passing a `WrappingCylinder` along with the `via` argument when creating

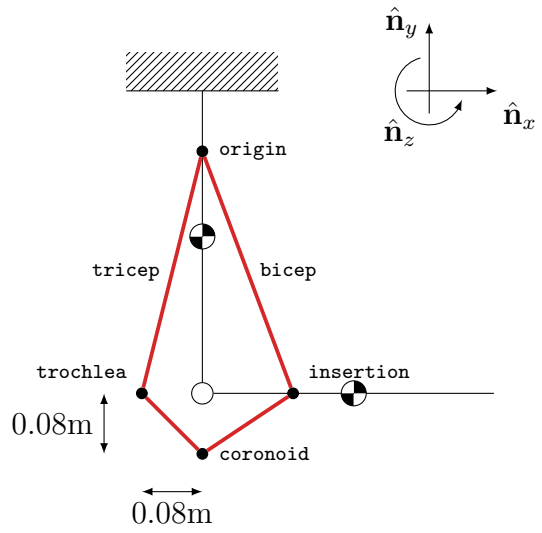
```
13 trochlea = AttachmentPoint("trochlea", model=arm, parent_body=humerus,
   ↪ position_x=0.20, position_y=-0.08)
14 coronoid = AttachmentPoint("coronoid", model=arm, parent_body=radius,
   ↪ position_x=-0.20, position_y=-0.08)
15 pathway = ObstacleSetPathway("pathway", model=arm, origin=origin,
   ↪ insertion=insertion, via=[trochlea, coronoid])
16 tricep = MillardMusculotendon("tricep", model=arm, pathway=pathway,
   ↪ peak_isometric_force=200, tendon_slack_length=0.32,
   ↪ optimal_fiber_length=0.44)
```

(a) Creation of two point pathway points and a tricep musculotendon.

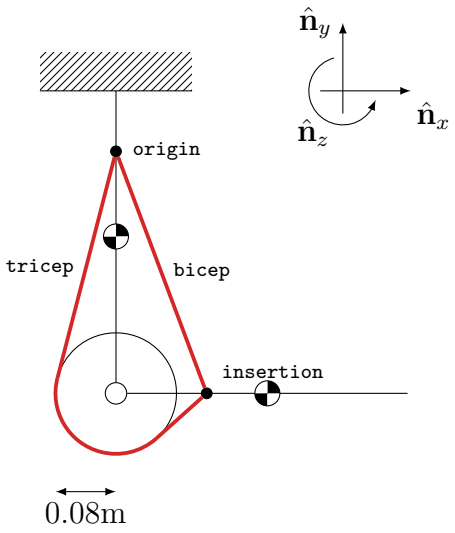
```
13 trochlea = WrappingCylinder("trochlea", model=arm, parent_body=humerus,
   ↪ position_x=0.20, radius=0.08, axis="z")
14 pathway = ObstacleSetPathway("pathway", model=arm, origin=origin,
   ↪ insertion=insertion, via=trochlea)
15 tricep = MillardMusculotendon("tricep", model=arm, pathway=pathway,
   ↪ peak_isometric_force=200, tendon_slack_length=0.32,
   ↪ optimal_fiber_length=0.44)
```

(b) Creation of a cylindrical wrapping surface and a tricep musculotendon.

Figure 5.15: Code example of adding a tricep musculotendon to the simple arm model (figs. 5.13 and 5.14) created using the *Pyomechanics* API. The tricep musculotendon exhibits a nonlinear pathway such that its actuation causes extension of the elbow. The model produced by the code is illustrated in fig. 5.16.



(a) Arm system with tricep pathway defined with two via points created by the *Pyomechanics* code in fig. 5.15a.



(b) Arm system with tricep pathway defined using a cylindrical wrapping surface created by the *Pyomechanics* code in fig. 5.15b.

Figure 5.16: Illustration of the simple arm model including a tricep muscle and simple wrapping around elbow created used the *Pyomechanics* code in figs. 5.13 and 5.15.

an `ObstacleSetPathway` (fig. 5.15b). Note that the side of the wrapping surface around which a musculotendon wraps can be switched by passing the `WrappingCylinder` to the `ObstacleSetPathway` with a prepended minus sign. The model created using the code in fig. 5.15b is illustrated in fig. 5.16b.

Pathway Approximation

Even simple nonlinear musculotendon pathways can require highly complex and lengthy analytical expressions to describe a musculotendon's length and shortening velocity as a function of skeletal kinematics [58, 107]. *Pyomechanics* implements musculotendon pathway approximation in which polynomials are used to represent a musculotendon's length, shortening velocity, moment arms and directions of force application, as functions of skeletal joint angles and angular velocities. To fit the approximating polynomials, the skeletal model is placed in a range of poses covering the full range of all joint angles and the musculotendon pathways are evaluated numerically to create a set of sample data. The order of approximating polynomials required will vary between musculotendons. *Pyomechanics* will attempt to use as few terms as possible in the approximating polynomials, such that the accuracy level specified in the user-adjustable settings of *Pyomechanics* is met. A default maximum allowable normalised error of 0.2% is used. A polynomial fitting routine from *SciPy* is used to determine the required polynomial order and appropriate coefficients. In the case of internal forces, these are replaced by torques on the relevant segments, with these torques being calculated as the product of musculotendon force and moment arm.

5.3.7 Optimal Control Problem Construction (`ocp` Module)

The majority of the `ocp` module of *Pyomechanics* functions similarly to that of *Pynamics*. One area where it differs substantially is in the tools it provides for initial guess generation. Generating suitable guesses for the musculotendon states in an OCP is particularly important. This is because each of the musculotendon curves has a range of values for its input over which its output is valid. Outside these ranges, however, the curves can evaluate unfeasibly large forces, or tendon or fibre lengths outside acceptable ranges. If such values are encountered in the process of solving the OCP they can result in overflow during numerical calculations and cause convergence problems or prohibit a solution to the OCP being found. *Pyomechanics* allows a user to fully specify an initial guess or to use a forward simulation to generate a dynamically-feasible initial guess. Additional to these options, *Pyomechanics* can also attempt to generate a sensible initial guess on behalf of the user. To generate

a sensible and feasible initial guess for the musculotendon states, *Pyomechanics* requires the user to provide initial guesses for the dynamics states, such as positions and velocities. If this condition is met then *Pyomechanics* will:

1. interpolate the dynamics states to produce a linearly spaced set of timepoints over which the guess will be generated;
2. evaluate the length and velocity of all musculotendons at each of these timepoints; and
3. equilibrate each musculotendon at each timepoint to find the value of its state variable for which the tendon and muscle fibre forces are in equilibrium.

The equilibration routine uses a bisection algorithm to find the value of either $\tilde{l}^M$ or $\tilde{F}^T$ (depending on which formulation of the musculotendon dynamics is being used) that solve the equation

$$\tilde{F}^T - \tilde{F}^M \cos(\alpha) = 0\,. \tag{5.45}$$

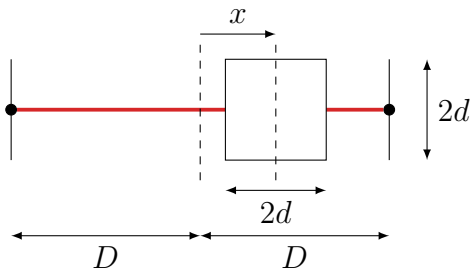
If a musculotendon cannot be equilibrated for the set of dynamics states specified by the user a warning is raised.

5.4 *Pyomechanics* Validation

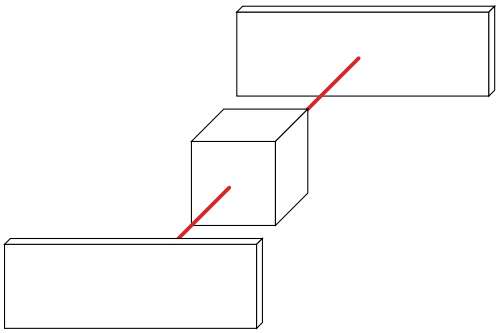
It is important to validate the results produced by any novel software package. *Pyomechanics* depends on both *Pycollo* (section 2.6) and *Pynamics* (section 4.3), both of which have been previously independently validated (sections 2.7 and 4.4). One focus of this section will, therefore, be validating the biomechanics-specific aspects of *Pyomechanics*. It has been stated that for biomechanical modelling software, this process should involve: verifying the software; validating the software's results by comparing models and simulations to independent experiments and other models; and testing the robustness of the software by evaluating the sensitivity of its results to model parameters and other modelling choices [161].

5.4.1 Tug of War (Lee and Umberger, 2016)

To facilitate the verification, validation and sensitivity analysis of *Pyomechanics*, a problem with a solution previously published in the academic literature was used. This is the tug of war OCP [212], which is based on an example biomechanical model from *OpenSim* [87]. The tug of war model involves a block of mass $m = 20\,\text{kg}$ and



(a) Diagram showing the position state x of the block in the tug of war model.



(b) Illustration of the full system in the tug of war model.

Figure 5.17: The tug of war model is a single *degree of freedom* (DoF) system involving a block actuated by two musculotendons. D denotes the distance between the central line and each fixed wall, d denotes the block half side length, and x denotes the single horizontal axial DoF.

half side length $d = 0.05\,\mathrm{m}$ mobilised by a single DoF in a horizontal direction. This DoF is parameterised by the variable x, which describes the block's displacement relative to its central position in the direction of the DoF, and its time derivative $v = \frac{dx}{dt}$ (fig. 5.17a). The block is attached to two musculotendons, one on each of two opposite faces, such that the block can be actuated along its DoF (fig. 5.17b). Each of the musculotendons attaches at its other end to a fixed wall at a distance $D = 0.35\,\mathrm{m}$ from the central line. As such, $l^{MT}_{\mathrm{LHS}} = l^{MT}_{\mathrm{RHS}} = 0.30\,\mathrm{m}$ when $x = 0$. The musculotendons are parameterised by

$$\begin{aligned} l^{M}_{\mathrm{opt}} &= 0.25\,\mathrm{m} & \alpha_{\mathrm{opt}} &= 0 \\ v^{M}_{\mathrm{max}} &= 2.5\,\mathrm{m\,s^{-1}} & \beta &= 0.1\,\mathrm{s\,m^{-1}} \\ l^{T}_{\mathrm{slack}} &= 0.05\,\mathrm{m} & \tau_{\mathrm{act}} &= 0.055\,\mathrm{s^{-1}} \\ F^{M}_{\mathrm{max}} &= 1000\,\mathrm{N} & \tau_{\mathrm{deact}} &= 0.065\,\mathrm{s^{-1}}\,. \end{aligned}$$

The tug of war OCP involves finding the two musculotendon excitations e_{LHS} and e_{RHS} that enable a trajectory subject to

$$\begin{aligned} x\,(0.0\,\mathrm{s}) &= -0.08\,\mathrm{m} & v\,(0.0\,\mathrm{s}) &= 0.0\,\mathrm{m\,s^{-1}} \\ x\,(0.5\,\mathrm{s}) &= 0.08\,\mathrm{m} & v\,(0.5\,\mathrm{s}) &= 0.0\,\mathrm{m\,s^{-1}} \\ x\,(1.0\,\mathrm{s}) &= -0.08\,\mathrm{m} & v\,(1.0\,\mathrm{s}) &= 0.0\,\mathrm{m\,s^{-1}}\,, \end{aligned}$$

while minimising the objective function

$$J = \int_{t=0.0}^{1.0} a^2_{\mathrm{LHS}} + a^2_{\mathrm{RHS}} dt\,, \tag{5.46}$$

where a_{LHS} and a_{RHS} are the activations of the two musculotendons. The movement must also be periodic, such that the musculotendon states and activations at $t = 0.0\,\mathrm{s}$ and $t = 1.0\,\mathrm{s}$ are equal for each musculotendon.

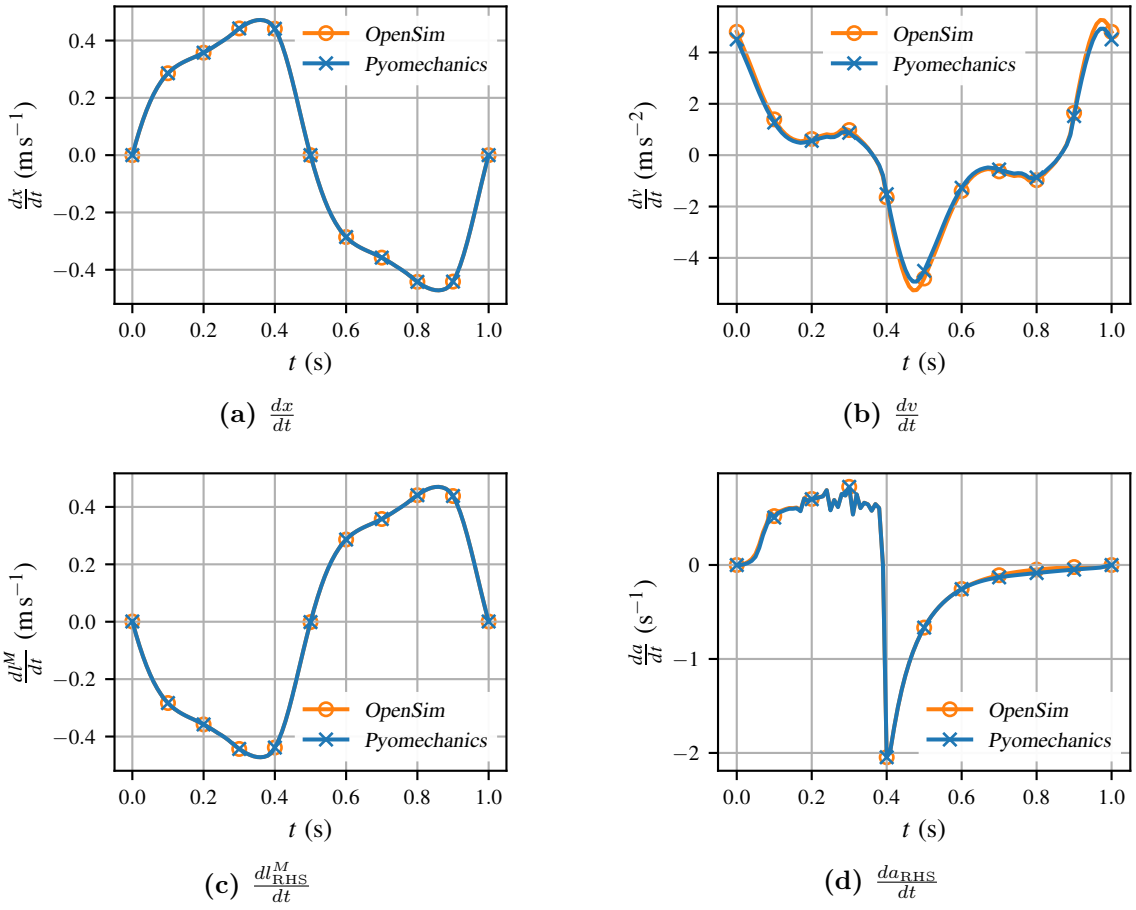
It was not possible to validate *Pyomechanics* against *Moco* using this OCP as *Moco* does not support multiphase OCPs. It is, therefore, not possible to formulate or solve the tug of war OCP using *Moco*. However, the *MATLAB* source code used by Lee and Umberger to solve the tug of war OCP, is available online as a supplement to the publication. This source code uses *OpenSim* to construct the tug of war model, which includes two musculotendons based on [242]. As such, the system is parameterised by six state and two control variables. The six state variables are: the block position x and velocity v; the two musculotendon activations a_{LHS} and a_{RHS}; and the two muscle fibre lengths l^M_{LHS} and l^M_{RHS}. Note that the muscle fibre lengths are not normalised, unlike the normalised states $\tilde{l}^M$ that *Pyomechanics* uses in one of its formulations of musculotendon dynamics. The two control variables are the two musculotendon excitations e_{LHS} and e_{RHS}. The source code implements a simplistic direct collocation framework to solve the OCP, which uses the backwards Euler discretisation scheme [36] and *MATLAB*'s interior-point *nonlinear programming problem* (NLP) solver `fmincon`. *OpenSim* was used to generate the six state equations, the derivatives of each of the six state variables with respect to time. Figure 5.25 shows the optimal state and control for the tug of war OCP, solved using the software implementation described above, the hardware and software described in section 2.7, and *OpenSim* 4.3. A discretisation with 100 collocation nodes was used as the resulting NLP subproblem took 3 h 23 min to solve and attempts to use a denser mesh became prohibitively computationally expensive. The optimal cost was 1.7269×10^{-2}. This replication of Lee and Umberger's results provides a benchmark against which *Pyomechanics* can be validated.

5.4.2 Software Verification

Musculotendon and Activation Dynamics

To verify the modelling of musculotendon and activation dynamics in *Pyomechanics*, the governing equations need to be checked to ensure that they produce accurate and physiologically realistic results. The musculotendon and activation dynamics include both novel and previously published equations. The novel musculotendon equations in *Pyomechanics* occur in the Millard musculotendon curves. *Pyomechanics*' Millard musculotendon curves were constructed by fitting functions to previously verified and validated models (figs. 5.6a, 5.7a, 5.8a, 5.9a and 5.10a). Therefore, these have been shown to have been suitably verified, given that maximum errors and RMSEs were within acceptable thresholds.

In cases where previously published equations have been implemented in soft-



(a) $\frac{dx}{dt}$ **(b)** $\frac{dv}{dt}$

(c) $\frac{dl^M_{\text{RHS}}}{dt}$ **(d)** $\frac{da_{\text{RHS}}}{dt}$

Figure 5.18: Comparison of musculotendon state derivative computations in *Pyomechanics* and *OpenSim*. x denotes the block position, v denotes the block velocity, l^M_{RHS} denotes the muscle fibre length of the RHS musculotendon, and a_{RHS} denotes the activation of the RHS musculotendon.

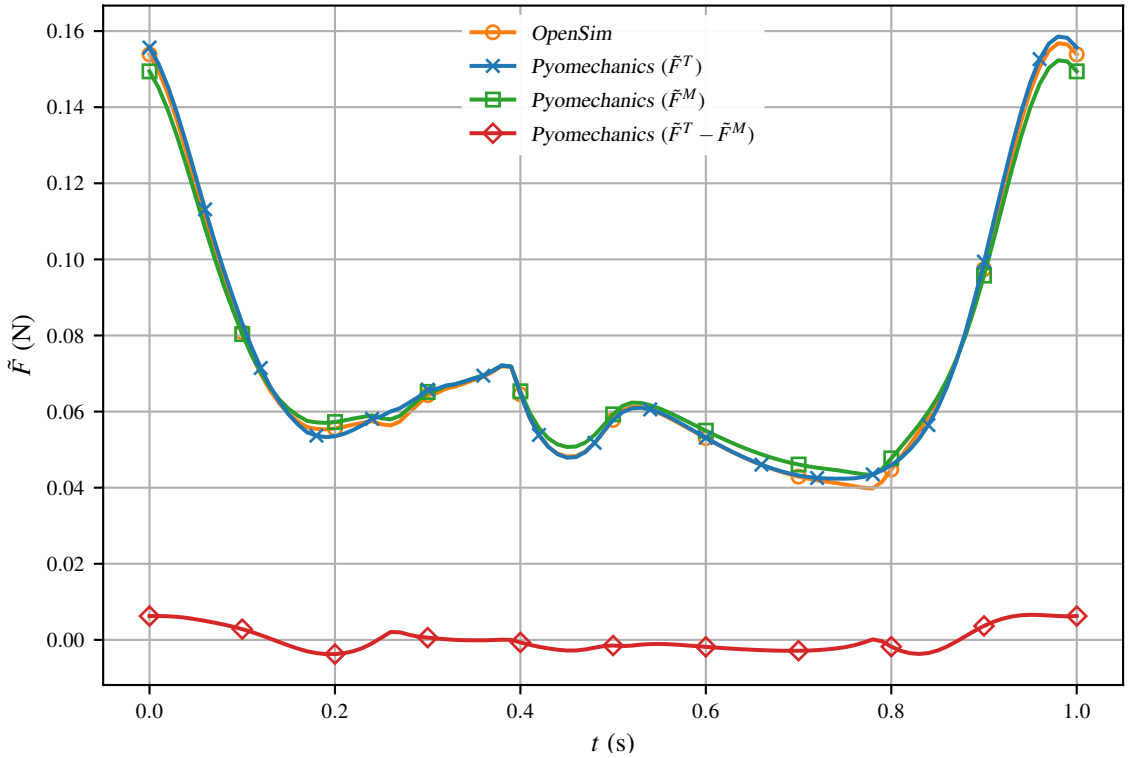


Figure 5.19: Comparison of the normalised muscle fibre forces $\tilde{F}^M$ and normalised tendon forces $\tilde{F}^T$ computed by *Pyomechanics* and *OpenSim*.

ware, verification can be conducted by evaluating the novel implementations and comparing the output to results from software that has been widely applied and well-validated elsewhere [161]. *Pyomechanics*'s musculotendon and activation dynamics equations were verified using this approach. The solution to the tug of war OCP described in section 5.4.1 was used for this. Additionally, *OpenSim* was used to evaluate the first-order time derivatives of all state variables at the solution. *Pyomechanics* was then verified by using the state and control solution as inputs, and recomputing the first-order time derivatives of the tug of war model's state variables using *Pyomechanics*' Millard musculotendon curves and the musculotendon parameters detailed in section 5.4.1. Figure 5.18 shows the results of this and clearly demonstrates the close agreement between the state derivatives computed using *OpenSim* and *Pyomechanics*. There was exact agreement in $\frac{dx}{dt}$ (fig. 5.18a) and $\frac{dl^M}{dt}$ (fig. 5.18c) between the two software packages. Close agreement in $\frac{dv}{dt}$ (fig. 5.18b) and $\frac{da}{dt}$ (fig. 5.18d) between the two software packages was also demonstrated. Note that values for only one of the musculotendons are shown as the solution to the tug of war OCP is symmetric. For $\frac{da}{dt}$ the normalised maximum error was 0.0128 and the *normalised root-mean-square error* (NRMSE) was 0.0062, where the errors were normalised by the difference in maximum and minimum values in the *OpenSim* data. The difference in $\frac{da}{dt}$ was demonstrated to be due entirely to the sigmoidal smoothing constant (c_1 in eq. (5.44)).

The normalised maximum error was 0.0330 and the NRMSE was 0.0171 for $\frac{dv}{dt}$. For the tug of war model, $\frac{dv}{dt}$ is given by the equation

$$\frac{dv}{dt} = \frac{F^T_{\text{RHS}} - F^T_{\text{LHS}}}{m}\,. \tag{5.47}$$

Figure 5.19 compares the value of F^T_{LHS} computed by *OpenSim* and the values of F^T_{LHS} and F^M_{LHS} computed using *Pyomechanics*. As $\tilde{l}^M$ and $\tilde{l}^T$ were known from the *OpenSim* data, *Pyomechanics* computed F^T_{LHS} using the Millard tendon force-length curve, while F^M_{LHS} was computed using eq. (5.17). It can be observed from fig. 5.19 that these values differ by no more than $0.01F^M_{\text{max}}$, with the error being significantly less than this over the majority of the trajectory. The differences can be attributed to the slight differences between *OpenSim*'s musculotendon curves and *Pyomechanics*' Millard musculotendon curves. *Pyomechanics* uses a path constraint in its musculotendon dynamics to ensure that

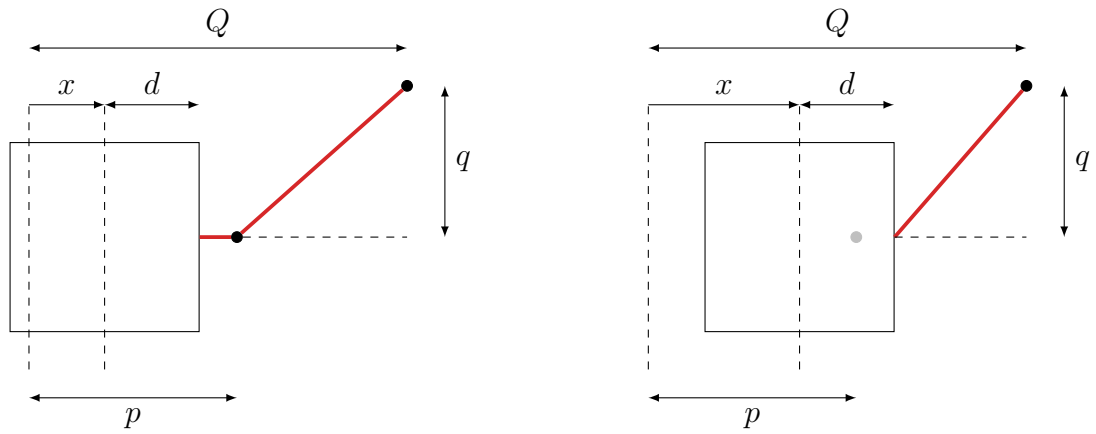
$$F^M - F^T = 0\,. \tag{5.48}$$

Therefore, any discrepancy between F^M and F^T would result in *Pyomechanics* adjusting the value of $\tilde{v}^M$ (a control variable in *Pyomechanics*' elastic tendon damped musculotendon model) to ensure that eq. (5.48) is satisfied. As neither *OpenSim*'s nor *Pyomechanics*' musculotendon curves are more correct than one another, the musculotendon and activation dynamics in *Pyomechanics* can be considered to have been verified.

5.4.3 Musculotendon Pathway Approximation

Pyomechanics implements pathway approximation in which polynomials are used to approximate musculotendon lengths, shortening velocities, moment arms and force vectors for complex 2D and 3D musculotendon geometries (section 5.3.6). Pathway approximation is important in biomechanical OCPs as it provides a means by which musculotendon pathways involving via-points that switch between active and inactive (depending on system geometry) can be used. *Pyomechanics* supports the creation of musculotendon pathways using the obstacle-set method, and allows these pathways to be defined using both via-points and wrapping surfaces.

The modelling of musculotendon pathways in *Pyomechanics* was verified using a test suite built on the *Python* testing framework *pytest* [204]. The *Pyomechanics* test suite contains unit and integration tests in which *Pyomechanics*' *object-oriented programming* (OOP) pathway modelling is verified against a number of numerical geometric test cases. *Pyomechanics* relies on *SciPy* [321] for the polynomial fitting used to approximate musculotendon pathways. As this is a widely applied and



(a) $x \leq (p - d)$. **(b)** $x > (p - d)$.

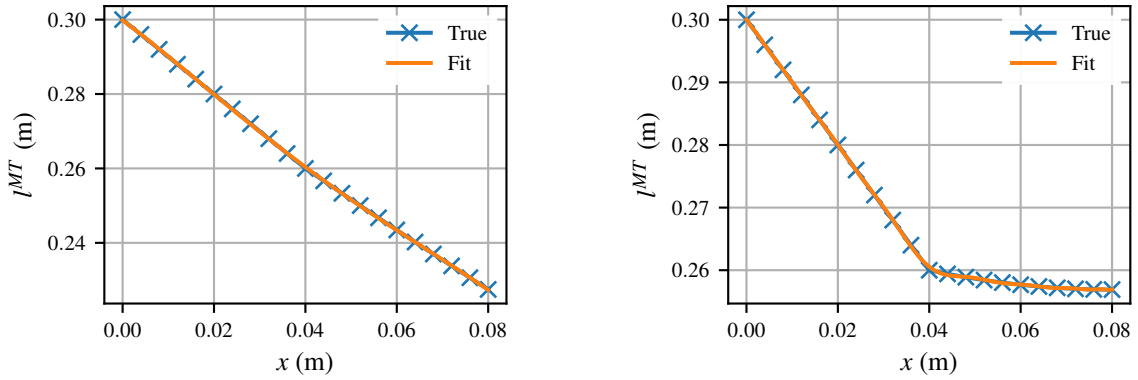
Figure 5.20: Diagram of the musculotendon pathway in test case $\mathcal{M}_1$. The pathway modifies the tug of war model and involves a via point that is active for $x \leq (p - d)$ and inactive otherwise. As the musculotendon path is 3D, figs. 5.20a and 5.20b correspond to both elevation (from $\hat{\mathbf{n}}_z$ direction) and plan (from $\hat{\mathbf{n}}_y$ direction) views. Q denotes the horizontal distance between the central point and the spatially-fixed musculotendon attachment, q denotes the distance between the spatially-fixed musculotendon attachment and the centre of the axis of motion in two orthogonal directions, d denotes the block half width, p denotes the distance from the central point to the via-point, and x denotes the single horizontal axial DoF.

well-validated open-source package, the numerical fitting aspect of *Pyomechanics*' pathway approximation can be considered to have been verified [161]. A description of a pair of test cases, implemented in the *Pyomechanics* test suite, for the verification of *Pyomechanics*' pathway modelling and approximation follow. Both are modified versions of the tug of war model, with the modification required so that the model included nonlinear musculotendon pathways that could be tested.

In the first test case $\mathcal{M}_1$, illustrated in fig. 5.20, the tug of war model is modified by adding a via-point to the *right hand side* (RHS) musculotendon at $(p, 0.5, 0)$, denoted in 3D cartesian coordinates (i.e. (x, y, z)). Its origin point was moved from $(0.35, 0.5, 0)$ to $(Q, 0.5 + q, q)$ where Q and q were selected such that $l^{MT} = (D - d)$ at $x = 0$, and $p = 0.09\,\text{m}$. The resulting pathway is, therefore, 3D. The via-point was defined such that it was active for $x \leq (p - d)$ and inactive otherwise, such that

$$l^{MT}_{\text{RHS}} = \begin{cases} D - (x + d) & x \leq (p - d) \\ \sqrt{2q^2 + \left(\sqrt{(D - p)^2 - 2q^2} - (x + d - p)\right)^2} & x > (p - d) \end{cases}, \tag{5.49}$$

where $d = 0.05\,\text{m}$ is the block half width and $D = 0.35\,\text{m}$ is the wall offset as before (section 5.4.1). A symmetrical change was also made for the *left hand side* (LHS)



(a) $q = 0.1$, polynomial approximation order 4.

(b) $q = 0.18166$, polynomial approximation order 18.

Figure 5.21: Approximations of the musculotendon pathway in test case $\mathcal{M}_1$ for different values of musculotendon attachment offset q.

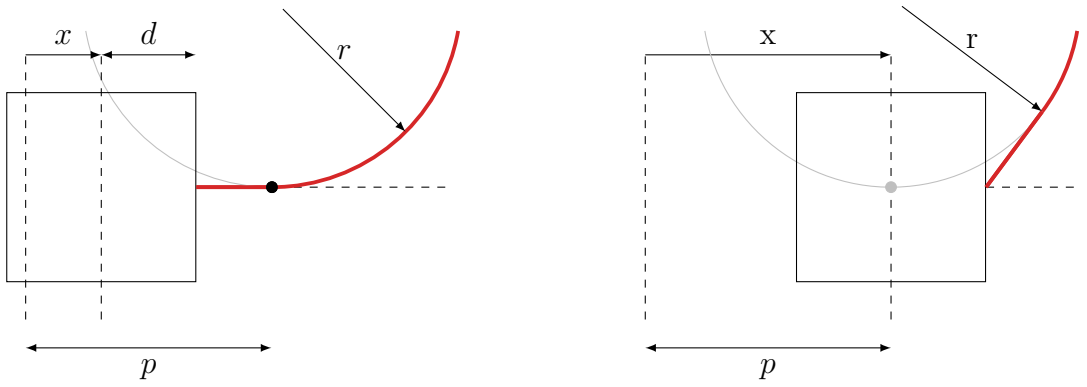
musculotendon.

It was verified that *Pyomechanics* correctly models the musculotendon pathway when $\mathcal{M}_1$ is implemented using the package. Approximations to the musculotendon pathway for a range of values of q were tested, with the approximations for $q = 0.1\,\text{m}$ (a generic nonlinear pathway) and $q = 0.181\,66\,\text{m}$ (the upper limit on q) shown in fig. 5.21. For $q = 0.1\,\text{m}$, a fourth order polynomial was required to meet the default normalised maximum error tolerance $\tilde{e}_{\text{max}}$ of 0.2%.(fig. 5.21a). For $q = 0.181\,66\,\text{m}$, a polynomial of order 18 was required to meet $\tilde{e}_{\text{max}}$ (fig. 5.21a). Such a high-order polynomial approximation was required as the use of a via-point in this extreme case leads to a sharp change in l^{MT} as the via-point transitions from being active to inactive. Approximations to the $\mathcal{M}_1$ musculotendon shortening velocity and force vector were also verified but are not shown here.

In the second test case $\mathcal{M}_2$, illustrated in fig. 5.22, a cylindrical wrapping surface with radius r was placed with its axis of symmetry oriented parallel to the z-axis and passing through the point $(p, 0.5 + r, 0)$. The RHS musculotendon's origin was moved to attach to the cylinder at a point on its surface such that $l^{MT} = D - d$ when $x = 0$. The pathway is such that for $x \leq p$ the musculotendon force acts parallel to the x-axis, but for $x > p$ the musculotendon becomes increasing detached from the wrapping surface at its insertion on the block. This results in

$$l^{MT}_{\text{RHS}} = \begin{cases} D - (x + d) & x \leq (p - d) \\ \\ x + D + d - 2p - 2r \arctan\left(\frac{x+d-p}{r}\right) & x > (p - d) \end{cases}. \tag{5.50}$$

Pyomechanics was also verified to accurately model the $\mathcal{M}_2$ pathway. Approx-



(a) $x \leq (p - d)$.

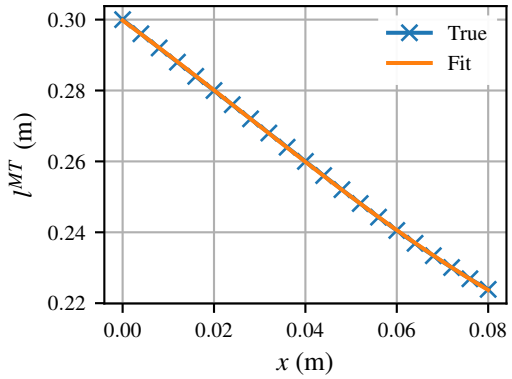
(b) $x > (p - d)$.

Figure 5.22: Diagram of the musculotendon pathway in test case $\mathcal{M}_2$. The pathway modifies the tug of war model and involves a cylindrical wrapping surface. Both figs. 5.22a and 5.22b are plan views (from $\hat{\mathbf{n}}_z$ direction). r denotes the radius of the cylindrical musculotendon wrapping surface, d denotes the block half width, p denotes the distance from the central point to the via-point, and x denotes the single horizontal axial DoF.

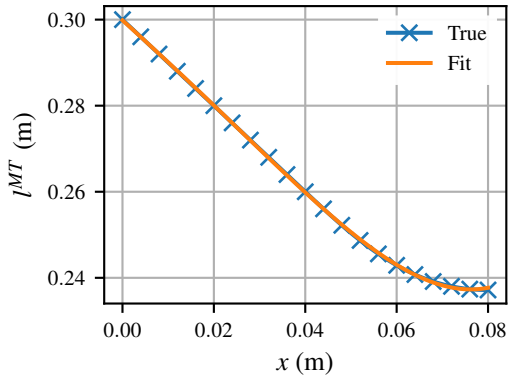
imations to the musculotendon pathway for a range of values of r were tested, with the approximations for $r = 0.1\,\text{m}$ (a generic nonlinear pathway) and $r = 0.04\,\text{m}$ (the upper limit on r before the direction of force action on the block changes) shown in fig. 5.23. In order to meet $\tilde{e}_{\text{max}}$, approximating polynomials of orders 3 and 4 were required for $r = 0.1\,\text{m}$ and $r = 0.04\,\text{m}$ respectively. As for $\mathcal{M}_1$, musculotendon shortening velocity and force vector in $\mathcal{M}_2$ were also verified but are not shown here.

5.4.4 Validation Against Published Results

To validate *Pyomechanics*, the package was used to construct the tug of war model and solve the tug of war OCP described in section 5.4.1. The *Pyomechanics* code to create the tug of war model, and to construct and solve the tug of war OCP, is shown in fig. 5.24. *Pyomechanics* solved the OCP in 450.60 ms, determining the optimal cost to be 1.2574×10^{-2}. In the example code (as is the case for all further investigations following in section 5.4) the number of mesh iterations was limited to one, as it was found that, in all cases, mesh refinement did not change the optimal cost up to five significant figures. All other settings were left as the *Pyomechanics* defaults. 29 NLP iterations were required for the OCP to converge. The optimal state and control are shown in fig. 5.25 (alongside the solution obtained using the software from [212]). Note that, again, only a_{RHS}, e_{RHS}, $\tilde{l}^M_{\text{RHS}}$ and $\tilde{F}^T_{\text{RHS}}$ (the values corresponding to the RHS musculotendon) are shown due to the solution being symmetric.



(a) $r = 0.1$, polynomial approximation order 3.



(b) $r = 0.04$, polynomial approximation order 4.

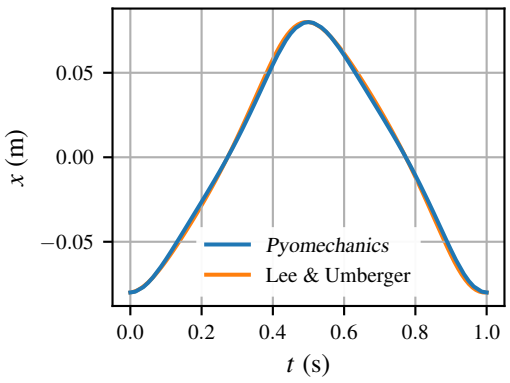
Figure 5.23: Approximations of the musculotendon pathway in test case $\mathcal{M}_2$ for different values of musculotendon wrapping surface radius r.

The optimal cost obtained using *Pyomechanics* was 31.5% less than that obtained using the methods of Lee and Umberger, 2016 (1.7269×10^{-2}). Comparing the optimal states, it can be seen that the optimal displacements of the block (fig. 5.25a) were nearly identical. However, inspecting the optimal velocities (fig. 5.25b), it can be seen that the block is accelerated away from stationary endpoint positions faster in *Pyomechanics*. This can be explained by the optimal tendon forces (fig. 5.25d). While $\tilde{F}^T_{\text{RHS}}$ at $t = 0$ is similar in magnitude in both cases, *Pyomechanics*' optimal $\tilde{F}^T_{\text{LHS}}$ was smaller at $t = 0$ (equivalent to $\tilde{F}^T_{\text{RHS}}$ at $t = 0.5\,\text{s}$ due to the system's symmetry). As the block acceleration is governed by eq. (5.47), *Pyomechanics*' smaller forces in the musculotendon pulling in the direction opposite to the motion allow the prescribed movement to be achieved with lower musculotendon activations (fig. 5.25e). As the objective function is the sum of squared activations, a lower optimal cost results.

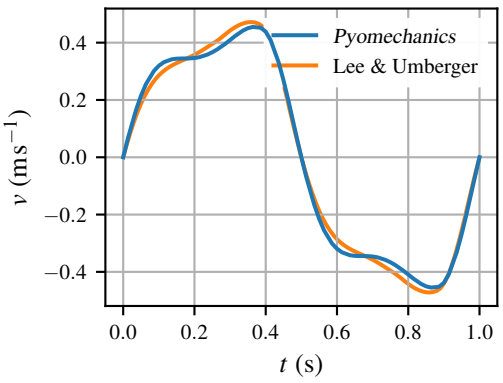
It is hypothesised that the difference between the two solutions can be explained by two factors. Firstly, *Pyomechanics* was likely able to solve the OCP to a significantly greater accuracy due to the high-order orthogonal collocation scheme employed by *Pycollo* in comparison to the simple backward Euler scheme used in the software implementation from Lee and Umberger. When the *OpenSim* version of the OCP was solved, it was found that increasing the density of the discretisation mesh significantly affected the optimal cost. Using 20 and 50 mesh sections resulted in optimal costs of 3.8690×10^{-2} and 2.1441×10^{-2} respectively, 124% and 24% larger than the optimal cost found when 100 mesh sections were used. This was not the case in *Pyomechanics* where the optimal cost remained constant across a range of mesh densities due to the high-order orthogonal collocation scheme employed. It was unfortunately not possible to confirm the relative contribution to the discrepancy in optimal costs obtained by solving the tug of war OCP using the *MATLAB*

```
from pyomechanics import (Model, SlidingJoint, RigidBody, AttachmentPoint,
↪ MillardMusculotendon)

d = 0.05  # block half width (in m)
D = 0.35  # distance to walls (in m)
m = 20  # mass (in kg)
T = 1.0  # OCP duration (in s)
x = 0.08  # distance to start/end position (in m)

lTslack = 0.05  # tendon slack length (in m)
lMopt = 0.25  # optimal fibre length (in m)
FMmax = 1000  # peak isometric force (in N)
beta = 0.1  # fibre damping coefficient (in s/m)
tau_a = 0.055  # activation time constant (in /s)
tau_d = 0.065  # deactivation time constant (in /s)

tow = Model("tug_of_war")
slider = SlidingJoint("slider", model=tow, parent_attachment=tow.origin,
↪ axis="x")
block = RigidBody("block", model=tow, mass=m, initial_position=-x,
↪ final_position=-x, intermediate_position=(T/2, x), initial_velocity=0,
↪ final_velocity=0, intermediate_velocity=(T/2, 0))
lhs_origin = AttachmentPoint("lhs_origin", model=tow, parent_body=tow,
↪ position_x=-D, position_y=d)
lhs_insertion = AttachmentPoint("lhs_insertion", model=block, position_x=-d,
↪ position_y=d)
lhs_muscle = MillardMusculotendon("lhs_muscle", model=tow, origin=lhs_origin,
↪ insertion=lhs_insertion, peak_isometric_force=FMmax,
↪ tendon_slack_length=lTslack, optimal_fiber_length=lMopt,
↪ fiber_damping_coefficient=beta, activation_time_constant=tau_a,
↪ deactivation_time_constant=tau_d)
rhs_origin = AttachmentPoint("rhs_origin", model=tow, parent_body=tow,
↪ position_x=D, position_y=d)
rhs_insertion = AttachmentPoint("rhs_insertion", model=block, position_x=d,
↪ position_y=d)
rhs_muscle = MillardMusculotendon("rhs_muscle", model=tow, origin=rhs_origin,
↪ insertion=rhs_insertion, peak_isometric_force=FMmax,
↪ tendon_slack_length=lTslack, optimal_fiber_length=lMopt,
↪ fiber_damping_coefficient=beta, activation_time_constant=tau_a,
↪ deactivation_time_constant=tau_d)

ocp = tow.optimal_control_problem(final_time=T,
↪ objective_function="minimise_squared_activations",
↪ enforce_periodicity=True)
ocp.settings.max_mesh_iterations = 1
solution = ocp.solve()
solution.plot()
```

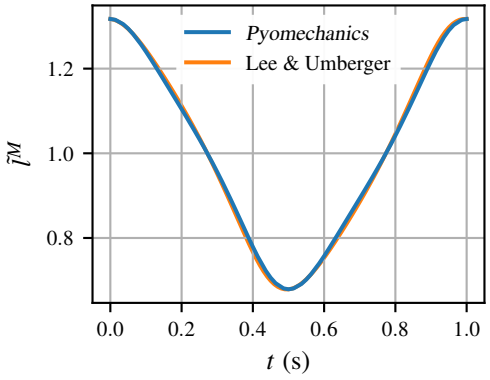
Figure 5.24: Creation of the tug of war model and OCP using *Pyomechanics*.



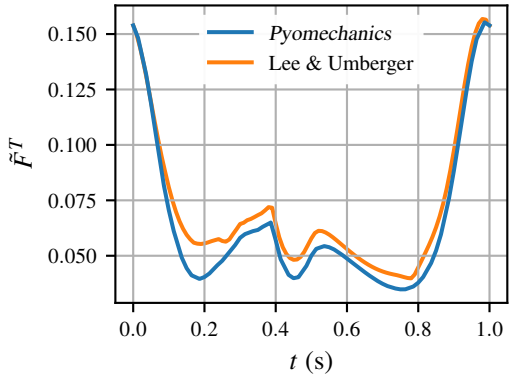
(a) $x(t)$



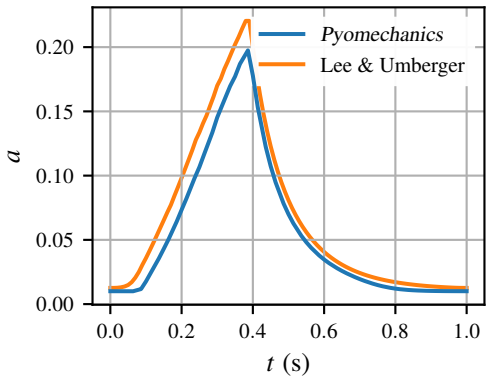
(b) $v(t)$



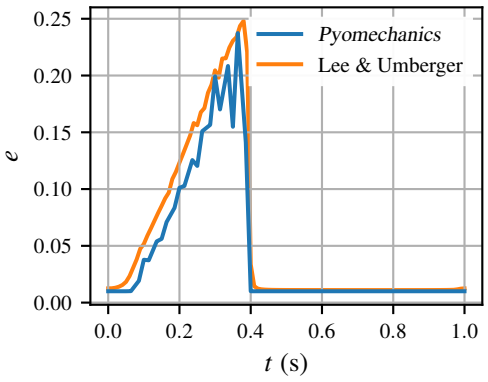
(c) $\tilde{l}^M_{\text{RHS}}(t)$



(d) $\tilde{F}^T_{\text{RHS}}(t)$



(e) $a_{\text{RHS}}(t)$



(f) $e_{\text{RHS}}(t)$

Figure 5.25: Comparison of the solutions to the tug of war OCP obtained using *Pyomechanics* and the *MATLAB* and *OpenSim* implementation from [212]. As the solution is periodic, only the values for the RHS musculotendon properties are shown. x denotes the block position, v denotes the block velocity, $\tilde{l}^M$ denotes the normalised muscle fibre length, F^T denotes the normalised tendon force, a denotes the musculotendon activation, and e denotes the musculotendon excitation.

and *OpenSim* implementation with a denser mesh due to prohibitive computational expense.

Secondly, small discrepancies between *Pyomechanics*' and *OpenSim*'s musculotendon curves will result in different musculotendon forces being developed for the same $\tilde{l}^T$ and $\tilde{l}^M$. This has already been demonstrated in fig. 5.19. Examination of fig. 5.25c shows that the muscle fibres lengthen and shorten through a range greater than $0.6l^M_{\text{opt}}$ during the optimal trajectory. Therefore, large ranges of operating conditions, corresponding to large portions of the musculotendon curves, are encountered. Depending on the model's and OCP's sensitivity to the modelling parameters, the slight differences in the musculotendon curves observed could have a magnified effect on the optimal solution. Investigation into the sensitivity of the tug of war OCP to its musculotendon parameters and modelling decisions follows in sections 5.4.5 and 5.4.6.

5.4.5 Sensitivity to Modelling Parameters

The robustness of any study involving biomechanical modelling should be tested by evaluating the sensitivity of the results to the model parameters [161]. A parametric study was conducted to determine the sensitivity of the tug of war OCP's solution to the core musculotendon modelling parameters [144]. This type of sensitivity analysis was chosen because, due to the small number of parameters involved, it was possible to determine the interaction effects of each of them.

In section 5.4.4 the tug of war OCP was solved using *Pyomechanics* on a mesh with 10 mesh sections ($K = 10$). For the sensitivity analysis, it was also solved on a denser mesh with 100 mesh sections ($K = 100$). Sensitivity to seven musculotendon parameters (l^M_{opt}, v^M_{max}, F^M_{max}, $\tilde{l}^T$, β, τ_{act} and τ_{deact}) was investigated. This was done by solving the OCP a number of times in succession and varying each parameter by up to $\pm 10\%$.

A subset of the results of this investigation is shown in table 5.5, giving the optimal cost $\mathcal{J}$, change in optimal cost relative to the baseline $\Delta\mathcal{J}$, the number of NLP iterations required for the subproblem to converge $\mathcal{N}$ and its change relative to the baseline solve $\Delta\mathcal{N}$. The optimal cost was found to be most sensitive to reductions in l^M_{opt}, with $\mathcal{J}$ increasing by up to 133% given a 5% reduction in l^M_{opt}. The optimal cost was also found to have high sensitivity to changes in v^M_{max} and l^T_{slack}. Relative sensitivity of the optimal cost to each parameter remained consistent across the different mesh densities investigated. These results suggest that the optimal solution to the tug of war OCP exhibits high sensitivity to the choice of musculotendon parameters.

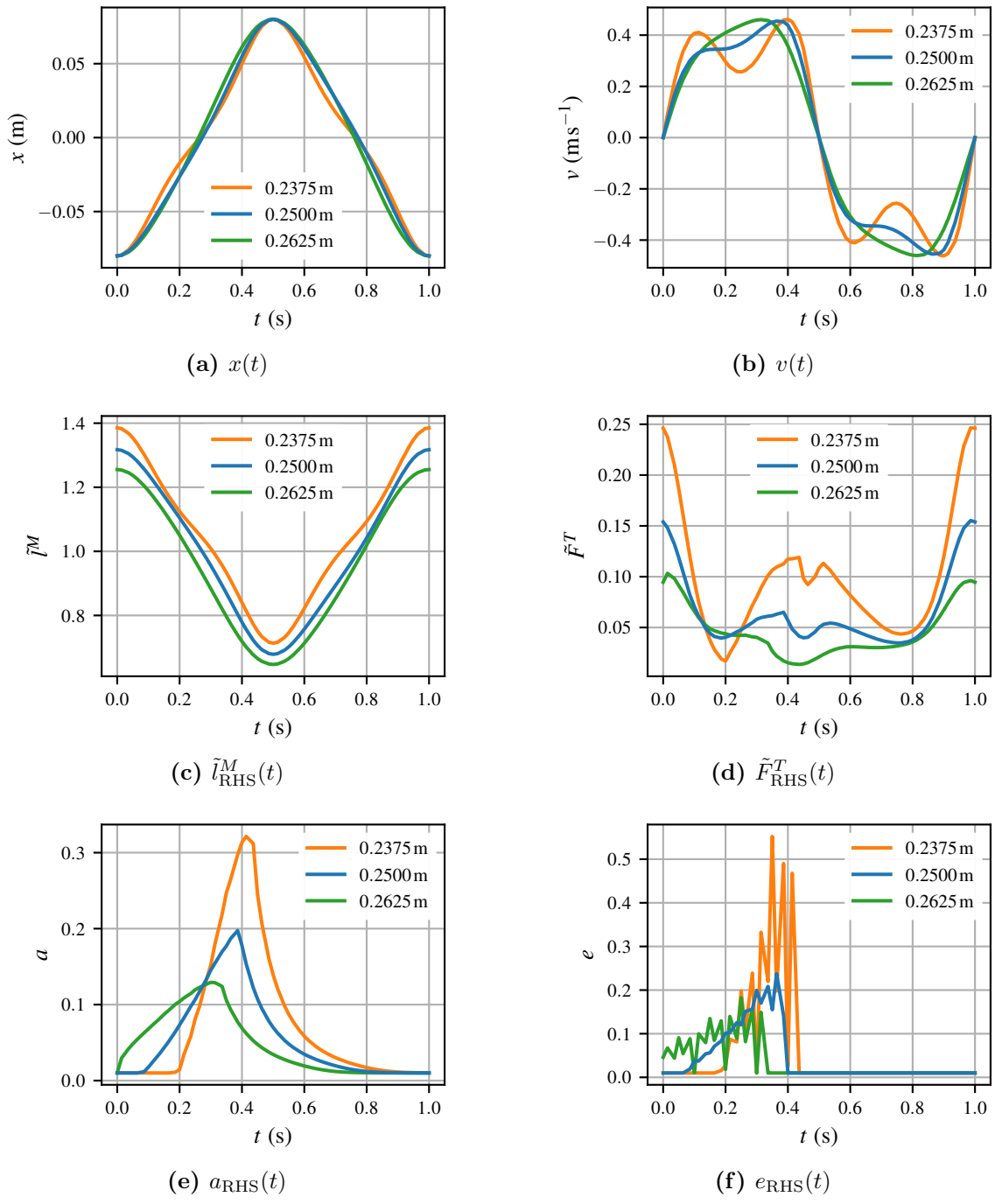
As the modelling parameters used to describe the musculotendons are intrinsically linked to the musculotendon curves, this can help to explain the differences in the optimal solutions found by *Pyomechanics* and Lee and Umberger, 2016.

For most musculotendon parameters, there was no significant pattern observed between changes in the parameter values and the number of NLP iterations required for the NLP subproblem to converge. The one exception was again $\tilde{l}^M$, whereby increases in this parameter's value resulted in disproportionately large increases in $\mathcal{N}$. It is hypothesised that this was because the fibres spent the majority of the trajectory at lengths near or below $\tilde{l}^M$ when $\tilde{l}^M$ was increased. This results in the motion being governed, in majority, by the active force characteristics of the muscle. Consequentially a more complex optimal control is required and more nonlinear optimal trajectories for $\tilde{F}^T$ are produced. This optimal solution appears to be more difficult for the NLP solver to converge to, perhaps due to the sensitivity of the activation and tendon force states to the excitations, thus requiring additional NLP iterations.

Additional to the musculotendon parameters, the sigmoidal smoothing constant (c_1 in eq. (5.44)) associated with the activation dynamics was also investigated. In the case of c_1, sensitivity across a range of magnitudes was investigated rather than a narrow linear range as for the other parameters. This was due to the exponential nature of eq. (5.44) requiring order-of-magnitude changes in c_1 for a noticeable difference to be observed. Values of 10, 100, 1000 and 10 000 were investigated, with results shown in fig. 5.27. It can be seen that the optimal state and control differed significantly between $c_1 = 10$ and larger values. There was limited difference between $c_1 = 100$ and $c_1 = 1000$, and negligible difference when c_1 was increased further (not shown). These differences were reflected in the optimal costs with $\mathcal{J}$ increasing to a plateau as c_1 was increased (table 5.5). This indicates that larger values of c_1 result in reduced sensitivity of the solution to the activation dynamics equations. This is because in the region of activation of the sigmoid, the activation and deactivation portions of eq. (5.44) are linearly combined, resulting in an unphysiological model. The larger c_1 is, the narrower this window and the more realistically the activation dynamics will be modelled.

5.4.6 Sensitivity to Modelling Decisions

The tug of war model described in section 5.4.1 contains a number of aspects that can be modelled in other ways. These include the use of: first-order activation dynamics in preference to zeroth-order; elastic tendons in preference to rigid tendons; musculotendon length to describe the musculotendon state in preference to using

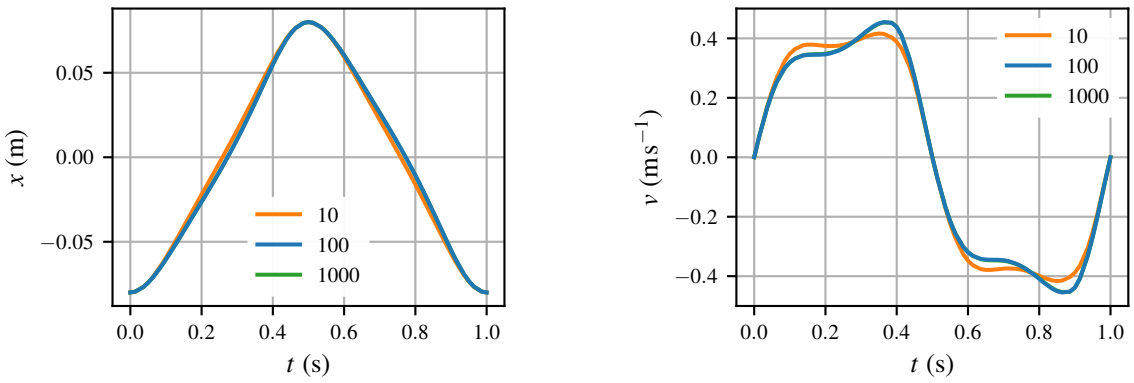


(a) $x(t)$ **(b)** $v(t)$

(c) $\tilde{l}^M_{\text{RHS}}(t)$ **(d)** $\tilde{F}^T_{\text{RHS}}(t)$

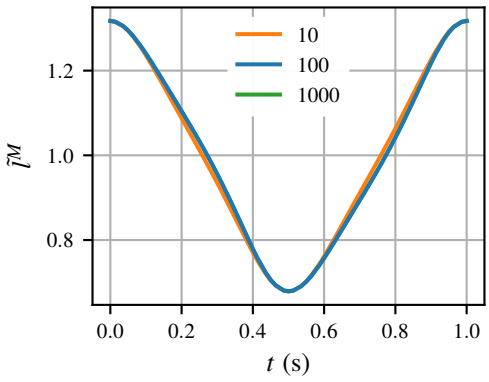
(e) $a_{\text{RHS}}(t)$ **(f)** $e_{\text{RHS}}(t)$

Figure 5.26: Comparison of sensitivity of the solution to the tug of war OCP to optimal fibre length l^M_{opt}. Results were obtained using *Pyomechanics*. As the solution is periodic, only the values for the RHS musculotendon properties are shown. x denotes the block position, v denotes the block velocity, $\tilde{l}^M$ denotes the normalised muscle fibre length, F^T denotes the normalised tendon force, a denotes the musculotendon activation, and e denotes the musculotendon excitation.

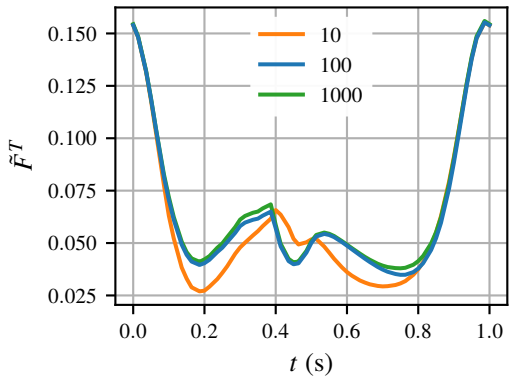


(a) $x(t)$

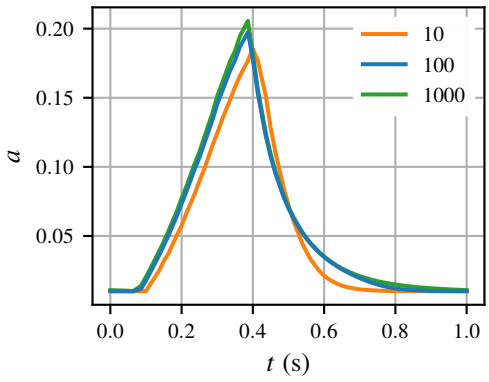
(b) $v(t)$



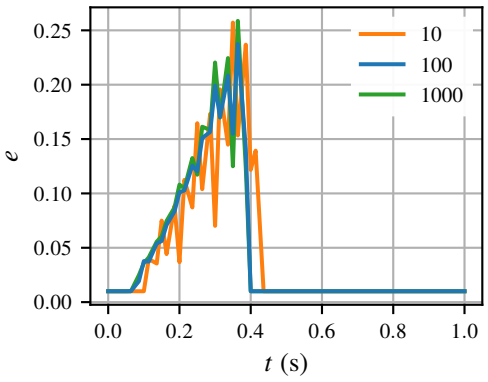
(c) $\tilde{l}^M_{\text{RHS}}(t)$



(d) $\tilde{F}^T_{\text{RHS}}(t)$



(e) $a_{\text{RHS}}(t)$



(f) $e_{\text{RHS}}(t)$

Figure 5.27: Comparison of sensitivity of the solution to the tug of war OCP to the activation dynamics smoothing parameter c_1. Results were obtained using *Pyomechanics*. As the solution is periodic, only the values for the RHS musculotendon properties are shown. x denotes the block position, v denotes the block velocity, $\tilde{l}^M$ denotes the normalised muscle fibre length, F^T denotes the normalised tendon force, a denotes the musculotendon activation, and e denotes the musculotendon excitation.

tendon force; a damping element in the muscle fibre model; and parallel-fibred muscle in preference to pennate muscle. *Pyomechanics* supports investigating each of these modelling decisions and their alternatives with ease. Each of these modelling decisions was investigated to:

1. determine the sensitivity of the solution to the tug of war OCP to the modelling decision; and

2. provide recommendations about relevant modelling decisions when creating and solving novel biomechanical OCPs.

Optimal cost $\mathcal{J}$ was used to measure sensitivity of the OCP solution to the modelling decision. In addition, the number of NLP iterations required for the NLP subproblem to converge $\mathcal{N}$ was used as a measure of the sensitivity of the OCP convergence to the modelling decision.

Activation Dynamics Order

The tug of war OCP was solved using zeroth- and first-order activation dynamics for a range of other modelling parameters (table 5.6). The results show that zeroth-order activation dynamics universally allowed a lower optimal cost to be obtained. This corresponded to between an 8.5% to 53.0% reduction in the optimal cost when zeroth-order activation dynamics were used compared to first-order. A lower optimal cost if zeroth-order activation dynamics are used is to be expected as they place fewer constraints on the musculotendon dynamics and therefore allow a more expansive solution space. The number of NLP iterations required was also universally lower when zeroth-order activation dynamics were used.

Choice of Musculotendon State

The musculotendon models in *Pyomechanics* allow musculotendon state to be parameterised using both $\tilde{l}^M$ (formulations B and D in table 5.6) and $\tilde{F}^T$ (formulations C and E in table 5.6). Comparison of formulations D and E (damped elastic tendon musculotendon model) show that the optimal costs were identical to at least five significant figures in all cases when first-order activation dynamics were used. Furthermore, neither formulation consistently exhibited better convergence performance than the other. This suggests that, based on this OCP, using either $\tilde{l}^M$ or $\tilde{F}^T$ to parameterise musculotendon state is equally suitable.

The undamped elastic tendon musculotendon model can be parameterised in four ways in *Pyomechanics*. In the first two of ways, either $\tilde{l}^M$ or $\tilde{F}^T$ is used to parameterise the musculotendon state and explicit equations for the musculotendon dynamics are used (formulations B and C respectively). In the second two of ways, either $\tilde{l}^M$ or $\tilde{F}^T$ is again used, but the formulation includes the additional control variable $\tilde{v}^M$ and the musculotendon dynamics are formulated implicitly and enforced with a path constraint (formulations D and E respectively). $\mathcal{J}$ was higher in all cases when an implicit formulation was used. Little difference in $\mathcal{N}$ was observed between the implicit and explicit formulations with first-order activation dynamics. However, when zeroth-order activation dynamics were used, $\mathcal{N}$ was significantly larger for the explicit formulations. This indicates that implicit formulations should be preferred.

Tendon Elasticity

Pyomechanics allows a user to easily switch between the rigid and elastic tendon musculotendon models. Solving the tug of war OCP using both tendon models and first-order activation dynamics, it was found that the optimal costs and corresponding optimal trajectories (not shown) differed between the two models but with no consistent pattern as other parameters changed. Comparing $\mathcal{N}$ between the two tendon models also showed no consistent relationship. Therefore, no performance advantage was observed from using the simplified rigid tendon model over the elastic tendon model, despite the elastic tendon model introducing one additional state and control variable each per musculotendon into the OCP.

Muscle Fibre Damping

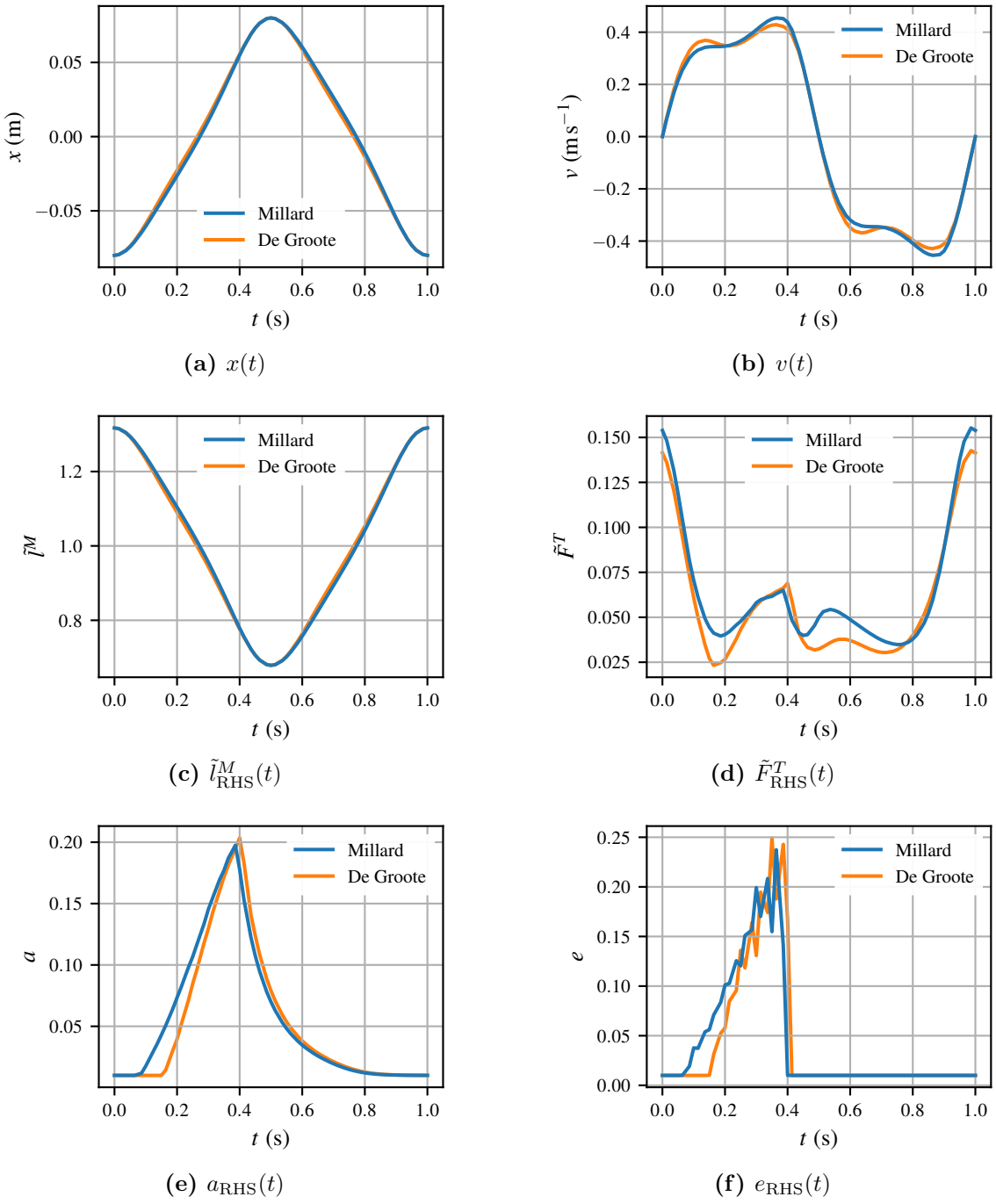
It has been suggested that including muscle fibre damping into a musculotendon model improves the numerical conditioning, despite there being little experimental evidence for this [242]. Solving the tug of war OCP using both undamped and damped musculotendon models found that $\mathcal{J}$ was minimised when the muscle fibre damping coefficient $\beta = 0$. As β was increased, $\mathcal{J}$ was also found to increase. This is to be expected as the parallel damping element in the damped musculotendon model dissipates energy and the larger that β is, the greater this dissipation will be. No consistent trend in convergence properties was observed when muscle fibre damping was modelled and included in the tug of war OCP.

Muscle Fibre Pennation

The tug of war model (as formulated in [212]) assumed parallel muscle fibres (i.e. no pennation). *Pyomechanics* makes it easy to enable pennation in a musculotendon model; simply by setting $\alpha_{\text{opt}} > 0$. The effect of including pennation in the tug of war musculotendons on $\mathcal{J}$ and $\mathcal{N}$ is shown in table 5.6c. Logarithmically spaced values of α_{opt} are shown so that the marginal effect of including pennation could be investigated. As expected, small values of α_{opt} close to zero had little effect on $\mathcal{J}$. As the value of α_{opt} was increased, $\mathcal{J}$ also increased at an exponential rate. This is due to that fact that a musculotendon's ability to produce force decreases as pennation angle increases. Therefore, larger activations are required to produce the same force in pennate muscle than in equivalently parameterised parallel-fibred muscle. This effect is likely to be particularly pronounced in the tug of war OCP due to the fibre lengths in the optimal trajectory covering a large operating range of $l^M_{\text{opt}} \pm 0.3l^M_{\text{opt}}$. As for muscle fibre damping, no relationship between the inclusion of modelling pennation angle or variation in α_{opt} and $\mathcal{N}$ was observed.

Musculotendon Curves

Pyomechanics offers two sets of musculotendon curves. These are the De Groote curves (implementations of the equations from [85]) and the novel Millard curves (OCP suitable equations fitted to the data from [242]). The tug of war model from [212] used the musculotendon curves from [242] and in section 5.4.4 the tug of war OCP was solved using *Pyomechanics* and the Millard curves. The tug of war OCP was also solved using *Pyomechanics* and the De Groote curves. The optimal cost achieved using the De Groote curves (1.2104×10^{-2}) was similar to, albeit marginally less than, the optimal cost achieved using the Millard curves (1.2574×10^{-2}). Furthermore, the tug of war OCP required one fewer NLP iterations to converge with the De Groote curves (37) than with the Millard curves (38). Comparison of the optimal states and controls obtained using the two sets of musculotendon curves is shown in fig. 5.28. It can be seen that although the optimal costs were similar, the optimal trajectories showed greater disparity. The solution using the De Groote curves exhibited later and faster activation, a marginally higher peak activation and marginally later deactivation. Consequently, faster initial accelerations of the block from stationary and lower overall tendon forces were also observed. The De Groote curves are more compliant than the Millard curves at low strains and only become stiffer at fibre lengths $\tilde{l}^M > 1.5$ (section 5.3.3). This explains the lower forces observed in the De Groote curves solution as the optimal trajectory using both sets of curves involved the fibres operating within the



(a) $x(t)$ **(b)** $v(t)$

(c) $\tilde{l}^M_{\text{RHS}}(t)$ **(d)** $\tilde{F}^T_{\text{RHS}}(t)$

(e) $a_{\text{RHS}}(t)$ **(f)** $e_{\text{RHS}}(t)$

Figure 5.28: Comparison of sensitivity of the solution to the tug of war OCP to the use of the two sets of musculotendon curves, the Millard curves and the De Groote curves, offered by *Pyomechanics*. As the solution is periodic, only the values for the RHS musculotendon properties are shown. x denotes the block position, v denotes the block velocity, $\tilde{l}^M$ denotes the normalised muscle fibre length, F^T denotes the normalised tendon force, a denotes the musculotendon activation, and e denotes the musculotendon excitation.

approximate range $0.7l^M_{\text{opt}} < \tilde{l}^M < 1.3l^M_{\text{opt}}$.

5.5 Discussion

As has been demonstrated, extensive initial verification, validation and sensitivity analysis of *Pyomechanics* has been conducted. This highlighted a number of recommendations for the construction of biomechanical models and formulation of biomechanical OCPs, as well as some areas for future work. This section concludes by also demonstrating that the objectives of this chapter have been met.

5.5.1 Musculotendon Pathways

Verification of *Pyomechanics*' musculotendon pathway modelling and approximation highlighted that the use of via-points in the obstacle-set method can lead to musculotendon pathways that exhibit rapid changes in musculotendon length, shortening velocity, moment arm and direction of force vector due to changes in the independent dynamics variables. This is undesirable as such pathways are not only unphysiological, but high-order approximating polynomials are required in order for them to be accurately represented. It is recommended that smooth wrapping surfaces be used in preference to via-points when creating nonlinear obstacle-set musculotendon pathways. This will result in more physiologically accurate and valid modelling, as well as improved OCP properties due to OCP formulations involving more numerically well-conditioned functions.

5.5.2 Order of Activation Dynamics

The use of zeroth- and first-order activation dynamics were investigated and compared. Zeroth-order activation dynamics were found to result in unphysiological optimal trajectories when included in biomechanical OCPs. This was due to the fact that mapping excitation to activation directly results in a control variable being effectively included in the musculotendon equations, which allows unphysical discontinuity in the musculotendon dynamics (particularly the tendon forces) to be achieved. For physiologically valid musculotendon dynamics, the use of first-order activation dynamics is recommended. However, zeroth-order activation dynamics exhibited better convergence properties than first-order activation dynamics. Therefore, the former presents a potentially useful tool in situations where it is difficult to achieve convergence of a biomechanical OCP. In this scenario, a potential approach

could be to simplify the biomechanical model to use zeroth-order activation dynamics and solve the OCP using this simplified model to generate an initial guess for the original model with first-order activation dynamics.

5.5.3 Explicit and Implicit Musculotendon Dynamics

Investigation of the different ways that *Pyomechanics* supports the formulation of musculotendon dynamics found that an implicit formulation was universally more performant and flexible than an explicit formulation. Comparison of the explicit and implicit formulations of identical OCPs found that the NLP subproblems frequently converge in fewer NLP iterations for the implicit formulation than the explicit formulation. This is despite implicitly formulated OCPs involving a greater number of variables and constraints. Furthermore, the implicit formulations allow both damped and undamped musculotendons to be modelled without modification, making them more widely applicable. Therefore, it is recommended that implicit formulations of the musculotendon equations are preferred and should be used by default.

5.5.4 Equations for First-Order Activation Dynamics

Pyomechanics implements the smoothed first-order activation dynamics equations from [85]. In the original publication, the sigmoidal smoothing parameter $c_1 = 10$ was proposed. Sensitivity testing using *Pyomechanics* demonstrated that OCP solutions are more sensitive to smaller values of c_1, while larger values of c_1 result in more NLP iterations being required for convergence to be achieved. It was concluded that $c_1 = 100$ yielded a sensible balance between sensitivity and computational cost. This value is, therefore, used as the default in *Pyomechanics*. However, if it is desired that computational cost be kept down when solving a biomechanical OCP and low $\mathcal{N}$ be prioritised, then it is recommended that a smaller value of c_1 be used. It is also advised that a lower bound of $c_1 = 10$ be considered as values smaller than this will result in too-gradual smoothing between activation and deactivation which will render the modelling of the activation dynamics unphysiological (fig. 5.12).

5.5.5 Choice of Musculotendon State

Section 5.4 investigated the effects of numerous modelling decisions on OCP solution and convergence. *Pyomechanics*supports parameterisation of the musculotendon

equations using both $\tilde{l}^M$ and $\tilde{F}^T$ as state variables. Both formulations yielded the exact same solutions and neither exhibited reliably improved OCP properties in comparison to the other. Therefore, the choice of whether $\tilde{l}^M$ or $\tilde{F}^T$ is used can be determined based on which is more convenient for the problem formulation.

5.5.6 Musculotendon Model Components

Pyomechanics allows users to easily add or remove a parallel damping element to their muscle fibre model, include or exclude pennation angle in the musculotendon equations, and switch between different musculotendon curves. The OCP solutions were found to be sensitive to all of these modelling decisions, while the convergence properties were not impacted. This means that users are able to select the most physically representative parameters without needing to consider such a decision's effects on the convergence properties of their OCP. This is contrary to the advice of Millard et al., 2013 [242], who suggested that damped musculotendon models offer improved numerical stability.

5.6 Conclusions

Pyomechanics, an open-source *Python* package for optimal control involving biomechanical models, with a particular focus on predictive simulation, has been developed. In particular, it:

- offers a library of OOP musculoskeletal modelling components and an easy to use API that enables users to efficiently construct biomechanical models using minimal lines of code;
- implements a set of musculotendon curves that replicate the most widely-used experimental data, with properties such that they are suitable for use in OCPs;
- offers two implicit formulations for damped elastic tendon musculotendon models;
- abstracts away the multibody and musculotendon modelling, thereby enabling users without expertise in the mathematics of these areas to derive the equations governing their modelled system; and
- supports users by formulating predictive simulation OCPs on their behalf.

Pyomechanics has been rigorously tested on a simple biomechanical OCP from the literature. A wide range of different OCP formulations have been compared and a set of recommendations made for how biomechanical OCPs should be constructed. These include using:

- smooth wrapping surfaces in preference to via-points when creating obstacle-set musculotendon pathways and their polynomial approximations;
- first-order activation dynamics to represent physiologically valid musculotendon dynamics;
- zeroth-order activation dynamics to generate a dynamically-valid initial guess in situations where complex OCPs are exhibiting poor convergence;
- implicit formulations of damped musculotendon dynamics, with no preference over whether musculotendon state is parameterised by $\tilde{l}^M$ or $\tilde{F}^T$; and
- a sigmoidal smoothing constant of 100 to accurately represent activation-deactivation switching in smoothed first-order activation dynamics, and only using smaller values when convergence properties are to be prioritised over physiological accuracy.

The open-source provision of *Pyomechanics* will allow researchers and practitioners without specific expertise in musculoskeletal modelling, multibody dynamics or optimal control to investigate biomechanical OCPs. In particular, it is recommended that future research investigates:

- applying the developed direct collocation methodologies to other biomechanics OCPs, including inverse-dynamics and motion tracking;
- musculotendon pathway approximation for complex 3D geometries;
- validating the package further on biomechanical models with more DoFs and a greater number of musculotendons; and
- applying *Pyomechanics*, and the BPST as a whole, to create subject-specific models of sprint cyclists and investigate the biomechanics of cycling by predictively simulating maximal pedalling.

		$K = 10$				$K = 100$			
		$\mathcal{J}_{10}$	$\Delta\mathcal{J}_{10}$ (%)	$\mathcal{N}_{10}$	$\Delta\mathcal{N}_{10}$ (%)	$\mathcal{J}_{100}$	$\Delta\mathcal{J}_{100}$ (%)	$\mathcal{N}_{100}$	$\Delta\mathcal{N}_{100}$ (%)
Baseline		0.01258		29		0.01257		65	
l^M_{opt}									
	−5%	0.02932	+133	28	−3.4	0.02921	+132	48	−25.0
	+5%	0.00787	−37.4	35	+20.7	0.00787	−37.4	158	+147
v^M_{max}									
	−5%	0.01399	+11.2	33	+13.8	0.01398	+11.2	55	−14.1
	+5%	0.01143	−9.15	32	+10.3	0.01143	−9.13	71	+10.9
F^M_{max}									
	−5%	0.01207	−4.05	37	+27.6	0.01206	−4.08	88	+37.5
	+5%	0.01306	+3.80	35	+20.7	0.01306	+3.85	56	−12.5
l^T_{slack}									
	−5%	0.01405	+11.72	39	+34.5	0.01407	+11.86	69	+7.8
	+5%	0.01135	−9.80	36	+24.1	0.01134	−9.82	69	+7.8
β									
	−5%	0.01197	−4.85	29	0.0	0.01196	−4.84	65	+1.6
	+5%	0.01321	+5.05	26	−10.3	0.01321	+5.04	70	+9.4
τ_{act}									
	−5%	0.01254	−0.31	34	+17.2	0.01253	−0.31	67	+4.7
	+5%	0.01262	+0.30	36	+24.1	0.01261	+0.30	67	+4.7
τ_{deact}									
	−5%	0.01231	−2.12	37	+27.6	0.01232	−2.04	55	−14.1
	+5%	0.01286	+2.27	48	+65.5	0.01285	+2.22	72	+12.5
c_1									
	×0.1	0.01108	−11.91	19	−34.5	0.01108	−11.84	25	−60.9
	×10	0.01351	+7.42	51	+75.9	0.01350	+7.40	155	+142
	×100	0.01352	+7.50	62	+113.8	0.01351	+7.48	447	+598

Table 5.5: Sensitivity of the optimal cost and convergence properties of the tug of war OCP to the musculotendon modelling parameters. Musculotendon modelling parameters investigated, and their baseline values, were: the optimal fibre length $l^M_{\text{opt}} = 0.25\,\text{m}$, the maximal fibre shortening velocity $v^M_{\text{max}} = 2.5\,\text{m}\,\text{s}^{-1}$, the peak isometric muscle fibre force $F^M_{\text{max}} = 1000\,\text{N}$, the tendon slack length $l^T_{\text{slack}} = 0.05\,\text{m}$, the fibre damping coefficient $\beta = 0.1\,\text{s}\,\text{m}^{-1}$, the activation time constant $\tau_{\text{act}} = 0.055\,\text{s}^{-1}$, the deactivation time constant $\tau_{\text{deact}} = 0.065\,\text{s}^{-1}$ and the activation smoothing constant $c_1 = 100$ (eq. (5.44)). K denotes the number of mesh sections, $\mathcal{J}$ and $\Delta\mathcal{J}$ denote the optimal cost and its change relative to the baseline, and $\mathcal{N}$ and $\Delta\mathcal{N}$ denote the number of NLP iterations for the subproblem to converge and its change relative to the baseline.

	A		B		C		D		E	
β	$\mathcal{J}$	$\mathcal{N}$	$\mathcal{J}$	$\mathcal{N}$	$\mathcal{J}$	$\mathcal{N}$	$\mathcal{J}$	$\mathcal{N}$	$\mathcal{J}$	$\mathcal{N}$
0	0.004 454	14	0.002 365	96	0.002 538	75	0.002 513	22	0.002 525	27
0.01	0.004 925	16					0.002 805	23	0.003 705	24
0.02	0.005 444	15					0.003 292	23	0.004 523	24
0.05	0.007 257	15					0.005 525	20	0.006 826	20
0.1	0.010 987	14					0.016 276	38	0.010 644	19
0.2	0.022 272	13					0.020 311	27	0.021 803	17
0.5	0.090 881	10					0.089 089	17	0.089 406	14
1	0.314 99	9					0.311 22	11	0.311 13	14

(a) Zeroth-order activation dynamics, pennation angle at optimal muscle fibre length $\alpha_{\text{opt}} = 0$

	A		B		C		D		E	
β	$\mathcal{J}$	$\mathcal{N}$	$\mathcal{J}$	$\mathcal{N}$	$\mathcal{J}$	$\mathcal{N}$	$\mathcal{J}$	$\mathcal{N}$	$\mathcal{J}$	$\mathcal{N}$
0	0.005 231	50	0.005 032	60	0.005 031	54	0.004 957	53	0.004 957	58
0.01	0.005 655	53					0.005 370	71	0.005 370	51
0.02	0.006 114	52					0.005 817	52	0.005 817	45
0.05	0.007 927	47					0.007 607	58	0.007 607	55
0.1	0.012 961	45					0.012 574	38	0.012 574	60
0.2	0.030 215	47					0.029 634	45	0.029 634	51
0.5	0.130 35	32					0.128 84	42	0.128 84	60
1	0.441 74	60					0.437 30	45	0.437 30	49

(b) First-order activation dynamics, pennation angle at optimal muscle fibre length $\alpha_{\text{opt}} = 0$

	A		B		C		D		E	
α_{opt}	$\mathcal{J}$	$\mathcal{N}$	$\mathcal{J}$	$\mathcal{N}$	$\mathcal{J}$	$\mathcal{N}$	$\mathcal{J}$	$\mathcal{N}$	$\mathcal{J}$	$\mathcal{N}$
0	0.012 961	45					0.012 574	38	0.012 574	60
0.01	0.012 964	33					0.012 578	48	0.012 578	42
0.02	0.012 974	59					0.012 588	46	0.012 588	51
0.05	0.013 044	34					0.012 663	53	0.012 663	42
0.1	0.013 310	40					0.012 945	41	0.012 945	43
0.2	0.014 579	55					0.014 321	44	0.014 321	38
0.5	0.031 319	41					0.035 003	47	0.035 003	44
1	0.189 95	70					0.281 37	43	0.281 37	37

(c) First-order activation dynamics, muscle fibre damping coefficient $\beta = 0.1$

Table 5.6: Sensitivity of the optimal cost ($\mathcal{J}$) and convergence properties ($\mathcal{N}$) of the tug of war OCP to the musculotendon modelling decisions. Five formulations are shown: formulation A used the rigid tendon model; formulation B used the undamped elastic tendon model with $\tilde{l}^M$ as the musculotendon state; formulation C used the undamped elastic tendon model with $\tilde{F}^T$ as the musculotendon state; formulation D used the damped elastic tendon model with $\tilde{l}^M$ as the musculotendon state and $\tilde{v}^M$ as an additional control variable; and formulation E used the undamped elastic tendon model with $\tilde{F}^T$ as the musculotendon state and $\tilde{v}^M$ as an additional control variable. Blank cells denote that the OCP could not be formulated in this way.

Chapter 6

Conclusions and Future Work

In this work, a comprehensive toolkit, the *Biomechanics Predictive Simulation Toolkit* (BPST), that enables both expert and non-expert users to construct and solve predictive simulation trajectory optimisation and *optimal control problems* (OCPs) involving musculoskeletal models has been developed. The BPST incorporates a number of the methods and algorithms developed as part of this thesis, along with other state-of-the-art theory and methods from the field of optimal control, which users can leverage. This chapter summarises the main conclusions from each previous chapter, all of which began with a thorough review of the relevant academic literature and software provision, and sets out a number of recommendations for further developing each area of the work.

6.1 Conclusions

6.1.1 Orthogonal Collocation (Chapter 2)

Chapter 2 detailed the development of a highly-performant, easy-to-use, open-source software package to solve general OCPs across a range of complexities. This *Python* package, *Pycollo*, forms the first component in the BPST.

The benchmarking of *Pycollo* against the industry-standard commercial software *GPOPS-II* showed it to be capable of accurately and correctly solving a range of recognised OCPs from the literature [94, 211, 267, 284, 294]. *Pycollo*'s overall performance across a range of measures was also demonstrated to be equivalent to that of *GPOPS-II*. These measures included the computation time required to solve the OCPs, the number of *nonlinear programming problem* (NLP) iterations required for

the generated NLP subproblems to converge, and the number of mesh refinements required to meet a specified mesh tolerance. *Pycollo*'s performance can, in part, be attributed to two factors. First is *Pycollo*'s framework for the automatic scaling of an OCP's transcribed NLP subproblems based on user-supplied variable bounds, random sampling of the search space, and analysis of the problem's function's derivatives. Sensibly scaling an OCP is important for ensuring its convergence, especially when the OCP variables span many orders of magnitude. Conducting this scaling on behalf of the user greatly facilitates ease-of-use. Second is *Pycollo*'s adapted *hp* mesh refinement algorithm that supports decreasing mesh sparsity in regions of the domain where the mesh tolerance is met, to improve the efficiency of the discretisation. Using a mesh refinement algorithm that can simultaneously increase and decrease mesh sparsity in different regions of the domain is important for use alongside a *Legendre-Gauss-Lobatto* (LGL)-based collocation scheme where oscillatory solutions can arise on sparse meshes due to the complementary collocation condition at interior mesh section boundaries.

6.1.2 Derivative Generation (Chapter 3)

Chapter 3 described the development of a highly-performant approach to determining first- and second-order derivative information for OCPs, termed *hybrid-symbolic-algorithmic differentiation* (hSAD). In addition, a computational implementation of the hSAD algorithm, *Dash*, has been incorporated into *Pycollo* as a second selectable derivative backend.

hSAD determines the sensitivities of a target function analytically in a preprocessing step. This enables more efficient evaluation traces for the computation of derivatives to be produced than with forward-mode *algorithmic differentiation* (AD). The greater the sparsity of the target function's derivative, the greater the theoretical relative performance of hSAD over forward-mode AD. For completely dense functions, hSAD's efficiency is no worse than that of forward-mode AD. This property is particularly valuable when applied to OCP derivatives where many thousands of numerical evaluations may be required throughout the duration of an OCP solve.

Benchmarking of *Pycollo*'s *Dash* backend against its *CasADi* backend demonstrated that the *Dash* backend enabled significantly faster preprocessing of the OCP derivatives, especially when dense mesh discretisations were used. This evidences that the hSAD algorithm leads to highly efficient formulations of the OCP derivatives when combined with exploitation of the known sparsity structure of the NLP formed by direct collocation. However, numerical evaluations of OCP derivatives

using the *Dash* backend were between one and two orders of magnitude slower than comparable evaluations using the *CasADi* backend. These differences can be attributed to design decisions made when implementing the *Dash* backend, specifically the choice of *Python* as the implementation language as opposed to the low-level *C* code generation employed by *CasADi*. This highlights that both algorithmic and computational performance are requirements when writing a highly-performant software implementation to generate OCP derivatives.

6.1.3 Multibody Dynamics (Chapter 4)

Chapter 4 detailed the development of a highly-performant, easy-to-use, open-source software package for modelling multibody systems and their dynamics. This package, *Pynamics*, is specifically tailored for use in OCPs and forms a core element of the BPST.

Pynamics is capable of formulating dynamical *equations of motion* (EoMs) both explicitly and implicitly. A direct comparison between explicit and implicit formulations of dynamics in OCPs demonstrated that implicit dynamics, in which the state equations of the generalised speeds are enforced using equality path constraints, should be preferred over explicit ones. This is because it is less computationally expensive to initialise the OCP when implicit dynamics are used because inversion of the mass matrix is not required, resulting in a significantly smaller expression graph which needs to be differentiated through to determine the OCP derivatives. Furthermore, it is less computationally expensive to solve the NLP subproblem, despite it being more than 50% larger, because the NLP derivatives resulting from the implicit dynamics are significantly simpler and thus computationally cheaper to evaluate. Finally, implicit equations are more numerically stable due to them not involving the inversion of near-singular mass matrices and yield smaller mesh errors as a result.

6.1.4 Musculoskeletal Modelling (Chapter 5)

Chapter 5 detailed the development of a highly-performant, easy-to-use, open-source software package capable of formulating and solving musculoskeletal predictive OCPs. This package, *Pyomechanics*, forms the final component, and primary *application programming interface* (API), of the BPST. It wraps *Pynamics* and *Pycollo*, and adds musculoskeletal modelling functionality.

Pyomechanics implements a set of musculotendon characteristic curves, based

on previously published interpolations of experimental data [242], specifically derived to be suitable for use in musculoskeletal OCPs. Their use has highlighted that meeting continuity and differentiation requirements is essential for the reliable convergence of NLPs. Additionally, in cases where it is not possible to algebraically invert the function describing a musculotendon curve, using a second function to describe its inverse is a suitable approach. Testing of the curves, as part of a musculoskeletal OCP, highlighted the sensitivity of the optimal trajectory and control to the musculotendon properties. Therefore, perfect numerical replication of previously published results may not be possible, as was demonstrated in this case, unless a model and optimisation framework identical to the original are used.

Pyomechanics also implements a damped equilibrium elastic tendon musculotendon model suitable for use in OCPs. This highly customisable model (section 5.3) allows users to select from a range of musculotendon modelling options. These options were extensively investigated as part of an OCP and the implicit formulation of musculotendon dynamics was shown to be able to accommodate all modelling options without the rate of convergence being penalised. This allows users to define musculoskeletal models containing musculotendons that accurately represent their system of interest, without needing to consider the potentially adverse effects of their modelling decisions on the convergence properties of their OCP.

6.2 Recommendations for Future Work

The research detailed in chapters 2 to 5, and conclusions stated in section 6.1, highlight a number of areas where there is opportunity to further develop and refine the contributions of this thesis.

There is potential to further refine the algorithms and methods for direct collocation, developed in chapter 2, and to improve *Pycollo* by further increasing computational performance and expanding support for the range of OCPs it can solve. Specific recommendations for future work in this area include:

- applying the mesh refinement algorithm to *Legendre-Gauss-Radau* (LGR) and *Legendre-Gauss* (LG) collocation;
- incorporating the latest ideas and developments from bang-bang mesh refinement into the adapted *hp* mesh refinement algorithm; and
- developing mesh error calculation functionality for LGR and LG collocation in *Pycollo*, so that the relative performance of LGL, LGR and LG collocation

can be directly compared.

Chapter 3 described a number of advanced concepts in hSAD that have the potential to increase the efficiency and performance of the algorithm. In particular, future work in this area should investigate:

- developing a further mode of hSAD that is analogous to reverse-mode AD (as opposed to forward-mode AD) as this will potentially yield performance benefits when evaluating derivatives for functions with many more outputs than inputs;
- the application of the hSAD concepts of function nodes and tier checkpointing further, to make recommendations on their application and use; and
- optimising and reimplementing *Dash* in a lower-level programming language, such that it is capable of numerically evaluating derivatives with performance equivalent to the *CasADi* backend, while also offering significantly cheaper derivative preprocessing costs.

The open-source provision of *Pynamics* will allow researchers in this field to extend the capabilities of the package in the future. Specific future work could include:

- how handling of event detection can be incorporated into multibody OCPs so that systems with altered dynamics can be modelled and predictively simulated using direct collocation;
- making additions to the *Pynamics* component library; and
- adding support for contact modelling in *Pynamics*.

Chapter 5 investigated predictive simulation of biomechanical models using a high-performance optimisation engine. To further advance the accessible application of state-of-the-art and novel methods and algorithms from other technical domains to biomechanics research, it is recommended that further work be conducted that investigates:

- applying the developed direct collocation methodologies to other biomechanics OCPs, including inverse-dynamics and motion tracking;
- musculotendon pathway approximation for complex *three-dimensional* (3D) geometries; and

- validating *Pyomechanics* further on biomechanical models with more *degrees of freedom* (DoFs) and a greater number of musculotendons.

In considering further work, it is important to be cognisant of the rate with which additional valuable research is being conducted in this area and to seek to build on emerging best-practice from the various streams of activity. The release of the BPST (incorporating *Pycollo*, *Pynamics* and *Pyomechanics*) as open-source packages will provide a robust and versatile platform on which researchers can continue to develop methods and algorithms for the predictive simulation of musculoskeletal models. Similarly, it is hoped that some of the contributions of this thesis can be used to inform developments in related software tools and new strands of research in the fields of optimal control theory, multibody dynamics and computational biomechanics.

Bibliography

[1] Aagaard, P., Simonsen, E. B., Andersen, J. L., Magnusson, P., and Dyhre-Poulsen, P. "Increased rate of force development and neural drive of human skeletal muscle following resistance training". In: *Journal of Applied Physiology* 93.4 (2002), pp. 1318–1326. ISSN: 8750-7587. DOI: 10.1152/japplphysiol.00283.2002. URL: http://jap.physiology.org/content/93/4/1318.short%7B%5C%%7D5Cnhttp://www.ncbi.nlm.nih.gov/pubmed/12235031.

[2] Abbott, B. C. and Aubert, X. M. "The force exerted by active striated muscle during and after change of length". In: *Journal of Physiology* 117 (1952), pp. 77–86. ISSN: 1469-7793. DOI: 10.1113/jphysiol.1952.sp004733.

[3] Abokhodair, A. A. "Complex differentiation tools for geophysical inversion". In: *Geophysics* 74.2 (2009). ISSN: 00168033. DOI: 10.1190/1.3052111.

[4] Abramowitz, M. and Stegun, I. A. *Handbook of mathematical functions with formulas, graphs, and mathematical tables*. US Government Printing Office, 1948.

[5] Ackermann, M. and Bogert, A. J. van den. "Predictive simulation of gait at low gravity reveals skipping as the preferred locomotion strategy". In: *Journal of Biomechanics* 45.7 (2012), pp. 1293–1298. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2012.01.029. URL: http://dx.doi.org/10.1016/j.jbiomech.2012.01.029.

[6] Ackermann, M. and Schiehlen, W. *Physiological methods to solve the force-sharing problem in biomechanics*. Dordrecht, Netherlands: Springer, 2009.

[7] Agamawi, Y. M., Hager, W. W., and Rao, A. V. "Mesh refinement method for solving bang-bang optimal control problems using direct collocation". In: *AIAA Scitech 2020 Forum* (2020), pp. 1–25. DOI: 10.2514/6.2020-0378. arXiv: 1905.11895.

[8] Agamawi, Y. M. and Rao, A. V. "Exploiting sparsity in direct orthogonal collocation methods for solving multiple-phase optimal control problems". In:

Space Flight Mechanics Meeting, 2018. 210009. 2018. DOI: 10.2514/6.2018-0724.

[9] Agamawi, Y. M. and Rao, A. V. "CGPOPS: A C++ software for solving multiple-phase optimal control problems using adaptive Gaussian quadrature collocation and sparse nonlinear programming". In: *arXiv* (2019). arXiv: 1905.11898.

[10] Agamawi, Y. M. and Rao, A. V. "Comparison of Derivative Estimation Methods in Optimal Control Using Direct Collocation". In: *AIAA Journal* 58.1 (2020). DOI: 10.2514/1.J058514.

[11] Agarwal, G. C., Berman, B. M., and Stark, L. "Studies in Postural Control Systems Part I: Torque Disturbance Input". In: *IEEE Transactions on Systems Science and Cybernetics* 6.2 (1970), pp. 116–121. ISSN: 21682887. DOI: 10.1109/TSSC.1970.300285.

[12] Åkesson, J., Årzén, K.-E., Gäfvert, M., Bergdahl, T., and Tummescheit, H. "Modeling and Optimization with Optimica and JModelica.org - Languages and tools for solving large-scale dynamic optimization problems". In: *Computers & Chemical Engineering* 34.11 (2010), pp. 1737–1749.

[13] Allen, S. J., King, M. A., and Yeadon, M. R. "Models incorporating pin joints are suitable for simulating performance but unsuitable for simulating internal loading". In: *Journal of Biomechanics* 45.8 (2012), pp. 1430–1436. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2012.02.019.

[14] Anderson, D. E., Madigan, M. L., and Nussbaum, M. A. "Maximum voluntary joint torque as a function of joint angle and angular velocity: Model development and application to the lower limb". In: *Journal of Biomechanics* 40.14 (2007), pp. 3105–3113. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2007.03.022.

[15] Anderson, F. C. and Pandy, M. G. "Storage and utilization of elastic strain energy during jumping". In: *Journal of Biomechanics* 26.12 (1993), pp. 1413–1427. ISSN: 00219290. DOI: 10.1016/0021-9290(93)90092-S. arXiv: 0021-9290(93)90092-s [10.1016].

[16] Anderson, F. C. and Pandy, M. G. "A Dynamic Optimization Solution for Vertical Jumping in Three Dimensions". In: *Computer Methods in Biomechanics and Biomedical Engineering* 2.3 (1999), pp. 201–231. ISSN: 1025-5842. DOI: 10.1080/10255849908907988.

[17] Anderson, F. C. and Pandy, M. G. "Static and dynamic optimization solutions for gait are pratical equivalent". In: *Journal of Biomechanics* 34.2 (2001), pp. 153–161.

[18] Anderson, F. C. and Pandy, M. G. "Dynamic Optimization of Human Walking". In: *Journal of Biomechanical Engineering* 123.5 (2001), pp. 381–390. ISSN: 00400262. DOI: 10.2307/1218045. URL: http://www.jstor.org/stable/1218045?origin=crossref.

[19] Anderson, F. C. and Pandy, M. G. "Individual muscle contributions to support in normal walking". In: *Gait and Posture* 17.2 (2003), pp. 159–169. ISSN: 09666362. DOI: 10.1016/S0966-6362(02)00073-5.

[20] Andersson, J. A., Gillis, J., Horn, G., Rawlings, J. B., and Diehl, M. "CasADi: a software framework for nonlinear optimization and optimal control". In: *Mathematical Programming Computation* 11.1 (2019), pp. 1–36. ISSN: 18672957. DOI: 10.1007/s12532-018-0139-4. URL: https://doi.org/10.1007/s12532-018-0139-4.

[21] Arnold, A. S., Salinas, S., Asakawa, D. J., and Delp, S. L. "Accuracy of muscle moment arms estimated from MRI-based musculoskeletal models of the lower extremity". In: *Computer Aided Surgery* 5.2 (2000), pp. 108–119. ISSN: 10929088. DOI: 10.1002/1097-0150(2000)5:2<108::AID-IGS5>3.0.CO;2-2.

[22] Arnold, E. M., Ward, S. R., Lieber, R. L., and Delp, S. L. "A model of the lower limb for analysis of human movement". In: *Annals of Biomedical Engineering* 38.2 (2010), pp. 269–279. ISSN: 00906964. DOI: 10.1007/s10439-009-9852-5. arXiv: s10439-009-9852-5 [10.1007].

[23] Ascher, U. M., Mattheij, R. M. M., and Russell, R. D. *Numerical solution of boundary value problems for ordinary differential equations*. Society for Industrial and Applied Mathematics, 1995.

[24] Audu, M. L. and Davy, D. T. "The influence of muscle model complexity in musculoskeletal motion modeling." In: *Journal of biomechanical engineering* 107.2 (1985), pp. 147–157. ISSN: 01480731. DOI: 10.1115/1.3138535.

[25] Axelsson, O. "Global Integration of differential equations through Lobatto quadrature". In: *Bit Numerical Mathematics* 4 (1964), pp. 69–86. ISSN: 14678462. DOI: 10.1111/j.1467-8462.1993.tb00770.x.

[26] Bardin, A. "Predicting horse limb responses to surface variations with a 3D musculoskeletal model". PhD thesis. Massey University, Manawatu, New Zealand, 2020.

[27] Barratt, P. R., Martin, J. C., Elmer, S. J., and Korff, T. "Effects of pedal speed and crank length on pedaling mechanics during submaximal cycling". In: *Medicine and Science in Sports and Exercise* 48.4 (2016), pp. 705–713. ISSN: 15300315. DOI: 10.1249/MSS.0000000000000817.

[28] Bashforth, F. and Adams, J. C. *An attempt to test the theories of capillary action by comparing the theoretical and measured forms of drops of fluid*. Cambridge: Cambridge University Press, 1883.

[29] Becerra, V. M. "Solving complex optimal control problems at no cost with PSOPT". In: *2010 IEEE International Symposium on Computer-Aided Control System Design*. 2010, pp. 1391–1396. URL: http://psopt.googlecode.com/files/PSOPT%7B%5C_%7DManual%7B%5C_%7DR2.pdf.

[30] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. "Cython: The Best of Both Worlds". In: *Computing in Science & Engineering* 13.2 (Mar. 2011), pp. 31–39. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.118. URL: http://ieeexplore.ieee.org/document/5582062/.

[31] Bellemare, F., Woods, J. J., Johansson, R., and Bigland-Ritchie, B. "Motor-unit discharge rates in maximal voluntary contractions of three human muscles." In: *Journal of neurophysiology* 50.6 (1983), pp. 1380–1392. ISSN: 0022-3077.

[32] Benson, D. A. "A Gauss Pseudospectral Transcription for Optimal Control". Ph.D. Massachusetts Institute of Technology, 2004. ISBN: 9781604138795. arXiv: arXiv:1011.1669v3.

[33] Benson, D. A., Huntington, G. T., Thorvaldsen, T. P., and Rao, A. V. "Direct Trajectory Optimization and Costate Estimation via an Orthogonal Collocation Method". In: *Journal of Guidance, Control, and Dynamics* 29.6 (2006), pp. 1435–1440. ISSN: 0731-5090. DOI: 10.2514/1.20478.

[34] Bernoulli, J. *Problema novum ad cujus solutionem Mathematici invitantur*. Acta Eruditorum, 1696, p. 269.

[35] Betts, J. T. "Survey of Numerical Methods for Trajectory Optimization". In: *Journal of Guidance, Control, and Dynamics* 21.2 (1998), pp. 193–207. ISSN: 0731-5090. DOI: 10.2514/2.4231.

[36] Betts, J. T. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming, Second Edition*. Philadelphia: Society for Industrial and Applied Mathematics, 2010. ISBN: 9780898716887. DOI: 10.1137/1.9780898718577.

[37] Betts, J. T. *Sparse Optimization Suite (SOS)*. Seattle, WA, 2013.

[38] Betts, J. T. "Using direct transcription to compute optimal low thrust transfers between libration point orbits". In: *Springer Optimization and Its Applications* 114 (2016), pp. 49–86. ISSN: 19316836. DOI: 10.1007/978-3-319-41508-6_2.

[39] Betts, J. T. *Practical Methods for Optimal Control Using Nonlinear Programming, Third Edition*. Philadelphia: Society for Industrial and Applied Mathematics, 2020. ISBN: 9781611976182. DOI: 10.1137/1.9781611976199.

[40] Betts, J. T. and Huffman, W. P. *Sparse Optimal Control Software SOCS*. Seattle, WA, 1997.

[41] Betts, J. T. and Huffman, W. P. "Exploiting sparsity in the direct transcription method for optimal control". In: *Computational Optimization and Applications* 14.2 (1999), pp. 179–201. ISSN: 09266003. DOI: 10.1023/A:1008739131724.

[42] Biegler, L. T. "An overview of simultaneous strategies for dynamic optimization". In: *Chemical Engineering and Processing: Process Intensification* 46.11 (2007), pp. 1043–1053. ISSN: 02552701. DOI: 10.1016/j.cep.2006.06.021.

[43] Biegler, L. T. and Zavala, V. M. "Large-scale nonlinear programming using IPOPT: An integrating framework for enterprise-wide dynamic optimization". In: *Computers and Chemical Engineering* 33.3 (2009), pp. 575–582. ISSN: 00981354. DOI: 10.1016/j.compchemeng.2008.08.006.

[44] Biewener, A. A., Wakeling, J. M., Lee, S. S., and Arnold, A. S. "Validation of hill-type muscle models in relation to neuromuscular recruitment and force-velocity properties: Predicting patterns of in vivo muscle force". In: *Integrative and Comparative Biology* 54.6 (2014), pp. 1072–1083. ISSN: 15577023. DOI: 10.1093/icb/icu070.

[45] Bischof, C., Carle, A., Corliss, G., Griewank, A., and Hovland, P. "ADIFOR-Generating Derivative Codes from Fortran Programs". In: *Scientific Programming* 1 (1992), pp. 11–29.

[46] Bisseling, R. W. and Hof, A. L. "Handling of impact forces in inverse dynamics". In: *Journal of Biomechanics* 39.13 (2006), pp. 2438–2444. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2005.07.021.

[47] Blankevoort, L., Kuiper, J. H., Huiskes, R., and Grootenboer, H. J. "Articular contact in a three-dimensional model of the knee". In: *Journal of Biomechanics* 24.11 (1991), pp. 1019–1031. ISSN: 00219290. DOI: 10.1016/0021-9290(91)90019-J.

[48] Blemker, S. S., Asakawa, D. S., Gold, G. E., and Delp, S. L. "Image-based musculoskeletal modeling: Applications, advances, and future opportunities". In: *Journal of Magnetic Resonance Imaging* 25.2 (2007), pp. 441–451. ISSN: 10531807. DOI: 10.1002/jmri.20805.

[49] Bliss, G. A. "The problem of Mayer with variable end points". In: *Transactions of the American Mathematical Society* 19.3 (1918), pp. 305–314.

[50] Bliss, G. A. "The problem of Lagrange in the calculus of variations". In: *American Journal of Mathematics* 52.4 (1930), pp. 673–744.

[51] Bliss, G. A. "The problem of Bolza in the calculus of variations". In: *Annals of Mathematics* (1932), pp. 261–274.

[52] Bliss, G. A. *Lectures on the Calculus of Variations*. Chicago: University of Chicago Press, 1946.

[53] Bobbert, M. F., Casius, R. L. J., and Van Soest, A. J. "The relationship between pedal force and crank angular velocity in sprint cycling". In: *Medicine and Science in Sports and Exercise* 48.5 (2016), pp. 869–878. ISSN: 15300315. DOI: 10.1249/MSS.0000000000000845.

[54] Bobbert, M. F. and van Ingen Schenau, G. J. "Coordination in vertical jumping". In: *Journal of Biomechanics* 21.3 (1988), pp. 249–262. ISSN: 00219290. DOI: 10.1016/0021-9290(88)90175-3. arXiv: 178.

[55] Bobbert, M. F. and Zandwijk, J. P. van. "A simulation study". In: 9457.94 (1987), pp. 1966–1969. DOI: 10.1103/PhysRevA.76.012504. arXiv: arXiv:1411.3848v2.

[56] Bogert, A. J. van den. "Musculoskeletal modelling: the DADS experience". In: *ISB Newsletter* 39 (1990), pp. 4–6.

[57] Bogert, A. J. van den, Blana, D., and Heinrich, D. "Implicit methods for efficient musculoskeletal simulation and optimal control". In: *Procedia IUTAM* 2 (2011), pp. 297–316. ISSN: 22109838. DOI: 10.1016/j.piutam.2011.04.027. arXiv: NIHMS150003. URL: http://dx.doi.org/10.1016/j.piutam.2011.04.027.

[58] Bogert, A. J. van den, Geijtenbeek, T., Even-Zohar, O., Steenbrink, F., and Hardin, E. C. "A real-time system for biomechanical analysis of human movement and muscle function". In: *Medical and Biological Engineering and Computing* 51.10 (2013), pp. 1069–1077. ISSN: 01400118. DOI: 10.1007/s11517-013-1076-z.

[59] Bogert, A. J. van den, Hupperets, M., Schlarb, H., and Krabbe, B. "Predictive musculoskeletal simulation using optimal control: effects of added limb mass on energy cost and kinematics of walking and running". In: *Special Issue Article Proc IMechE Part P: J Sports Engineering and Technology* 226.2 (2012), pp. 123–133. ISSN: 1754-3371. DOI: 10.1177/1754337112440644.

[60] Bolza, O. "Über den anormalen Fall beim Lagrangeschen und Mayerschen Problem mit gemischten Bedingungen und variablen Endpunkten". In: *Mathematische Annalen* 74 (1913), pp. 430–446.

[61] Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. *JAX: composable transformations of Python and NumPy programs*. 2018.

[62] Brand, R. A., Crowninshield, R. D., Wittstock, C. E., Pedersen, D. R., Clark, C. R., and Krieken, F. M. van. "A Model of Lower Extremity Muscular Anatomy". In: *Journal of Biomechanical Engineering* 104.4 (1982), pp. 304–310. ISSN: 0148-0731. DOI: 10.1115/1.3138363. URL: http://biomechanical.asmedigitalcollection.asme.org/article.aspx?articleid=1395925.

[63] Brockie, S. G. *Pycollo*. 2021. URL: https://github.com/brocksam/pycollo.

[64] Brown, I. E. and Loeb, G. E. "Measured and modeled properties of mammalian skeletal muscle: III. The effects of stimulus frequency on stretch-induced force enhancement and shortening-induced force depression". In: *Journal of Muscle Research and Cell Motility* 21.1 (2000), pp. 21–31. ISSN: 01424319. DOI: 10.1023/A:1005619014170.

[65] Brown, I. E. and Loeb, G. E. "Measured and modeled properties of mammalian skeletal muscle: IV. Dynamics of activation and deactivation". In: *Journal of Muscle Research and Cell Motility* 21.1 (2000), pp. 33–47. ISSN: 16130073. DOI: 10.1023/A. arXiv: 0005074v1 [arXiv:astro-ph].

[66] Broyden, C. G. "The convergence of a class of double-rank minimization algorithms 2. The New Algorithm". In: *IMA Journal of Applied Mathematics (Institute of Mathematics and Its Applications)* 6.1 (1970), pp. 222–231. ISSN: 02724960. DOI: 10.1093/imamat/6.1.76.

[67] Bryson, A. E., Desai, M. N., and Hoffman, W. C. "Energy-state approximation in performance optimization of supersonic aircraft". In: *Journal of Aircraft* 6.6 (1969), pp. 481–488. ISSN: 15333868. DOI: 10.2514/3.44093.

[68] Bryson, A. E. and Ho, Y.-C. *Applied Optimal Control*. New York: John Wiley and Sons, 1975.

[69] Burke, E. R. *Motor units: anatomy, physiology, and functional organization*. Wiley Online Library, 1981.

[70] Büskens, C. and Wassel, D. "The ESA NLP solver WORHP". In: *Springer Optimization and Its Applications* 73 (2013), pp. 85–110. ISSN: 19316836. DOI: 10.1007/978-1-4614-4469-5_4.

[71] Butcher, J. C. *Numerical Methods for Ordinary Differential Equations, Third Edition*. New York: John Wiley & Sons, 2016.

[72] Byrd, R. H., Nocedal, J., and Waltz, R. A. "KNITRO: An Integrated Package for Nonlinear Optimization". In: (2005), pp. 35–59. DOI: 10.1007/0-387-30065-1_4.

[73] Carbone, V., Fluit, R., Pellikaan, P., Krogt, M. M. van der, Janssen, D., Damsgaard, M., Vigneron, L., Feilkas, T., Koopman, H. F. J. M., and Verdonschot, N. "TLEM 2.0 - A comprehensive musculoskeletal geometry dataset for subject-specific modeling of lower extremity". In: *Journal of Biomechanics* 48.5 (2015), pp. 734–741. ISSN: 18732380. DOI: 10.1016/j.jbiomech.2014.12.034. URL: http://dx.doi.org/10.1016/j.jbiomech.2014.12.034.

[74] Casius, R. L. J., Bobbert, M. F., and Van Soest, A. J. "Forward dynamics of two-dimensional skeletal models. A newton-euler approach". In: *Journal of Applied Biomechanics* 20.4 (2004), pp. 421–449. ISSN: 10658483.

[75] Catelli, D. S., Wesseling, M., Jonkers, I., and Lamontagne, M. "A musculoskeletal model customized for squatting task". In: *Computer Methods in Biomechanics and Biomedical Engineering* 22.1 (2019), pp. 21–24. ISSN: 14768259. DOI: 10.1080/10255842.2018.1523396. URL: https://doi.org/10.1080/10255842.2018.1523396.

[76] Charles, J. P., Cappellari, O., Spence, A. J., Wells, D. J., and Hutchinson, J. R. "Muscle moment arms and sensitivity analysis of a mouse hindlimb musculoskeletal model". In: *Journal of Anatomy* 229.4 (2016), pp. 514–535. ISSN: 14697580. DOI: 10.1111/joa.12461.

[77] Chevallereau, C., Abba, G., Aoustin, Y., Plestan, F., Wit, C. C. de, Grizzle, J. W., and Westervelt, E. R. "RABBIT: A Testbed for Advanced Control Theory". In: *IEEE Control Systems Magazine* 23 (2013), pp. 57–79.

[78] Community, C.-f. *The conda-forge Project: Community-based Software Distribution Built on the conda Package Format and Ecosystem*. 2015. DOI: http://doi.org/10.5281/zenodo.4774216. URL: https://anaconda.org/conda-forge/pycollo.

[79] Conceição, F., King, M. A., Yeadon, M. R., Lewis, M. G., and Forrester, S. E. "An isovelocity dynamometer method to determine monoarticular and biarticular muscle parameters". In: *Journal of Applied Biomechanics* 28.6 (2012), pp. 751–760. ISSN: 15432688. DOI: 10.1123/jab.28.6.751.

[80] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Third Edition*. Boston, MA: The MIT Press, 2009.

[81] Crowninshield, R. D. and Brand, R. A. "A physiologically based criterion of muscle force prediction in locomotion". In: *Journal of Biomechanics* 14.11 (1981), pp. 793–801. ISSN: 00219290. DOI: 10.1016/0021-9290(81)90035-X. arXiv: 0021-9290(81)90035-x [10.1016].

[82] Darby, C. L., Hager, W. W., and Rao, A. V. "An hp-adaptive pseudospectral method for solving optimal control problems". In: *Optimal Control Applications and Methods* 32.4 (2011), pp. 476–502. ISSN: 01432087. DOI: 10.1002/oca.957.

[83] Darby, C. L., Hager, W. W., and Rao, A. V. "Direct trajectory optimization using a variable low-order adaptive pseudospectral method". In: *Journal of Spacecraft and Rockets* 48.3 (2011), pp. 433–445. ISSN: 15336794. DOI: 10.2514/1.52136.

[84] Davy, D. T. and Audu, M. L. "A dynamic optimization technique for predicting muscle forces in the swing phase of gait". In: *Journal of Biomechanics* 20.2 (1987), pp. 187–201. ISSN: 00219290. DOI: 10.1016/0021-9290(87)90310-1.

[85] De Groote, F., Kinney, A. L., Rao, A. V., and Fregly, B. J. "Evaluation of Direct Collocation Optimal Control Problem Formulations for Solving the Muscle Redundancy Problem". In: *Annals of Biomedical Engineering* 44.10 (2016), pp. 2922–2936. ISSN: 15739686. DOI: 10.1007/s10439-016-1591-9.

[86] Delp, S. L. "Surgery simulation: a computer graphics system to analyze and design musculoskeletal reconstructions of the lower limb". PhD thesis. 1990, p. 117. ISBN: 0612699625. DOI: 10.16953/deusbed.74839. URL: http://en.scientificcommons.org/33620246.

[87] Delp, S. L., Anderson, F. C., Arnold, A. S., Loan, P. J., Habib, A., John, C. T., Guendelman, E., and Thelen, D. G. "OpenSim: Open source to create and analyze dynamic simulations of movement". In: *IEEE transactions on bio-medical engineering* 54.11 (2007), pp. 1940–1950. ISSN: 0018-9294. DOI: 10.1109/TBME.2007.901024.

[88] Delp, S. L. and Loan, P. J. "A computational framework for simulating and analyzing human and animal movement". In: *Computing in Science & Engineering* 2.5 (2000), pp. 46–55.

[89] Delp, S. L., Loan, P. J., Hoy, M. G., Zajac, F. E., Topp, E. L., and Rosen, J. M. *An Interactive Graphics-Based Model of the Lower Extremity to Study Orthopaedic Surgical Procedures*. 1990. DOI: 10.1109/10.102791. arXiv: 169.

[90] Dembia, C. L. "Simulating Assistive Technology: Insights, Tools, and Open Science". PhD thesis. 2020, pp. 1–110. ISBN: 9798662510074. URL: https://www.proquest.com/docview/2430974784?accountid=28839%7B%5C%%7D0Ahttp://www.yidu.edu.cn/educhina/educhina.do?artifact=%7B%5C&%7Dsvalue=Simulating+Assistive+Technology%7B%5C%%7D3A+Insights%7B%5C%%7D2C+Tools%7B%5C%%7D2C+and+Open+Science%7B%5C&%7Dstype=2%7B%5C&%7Ds=on%7B%5C%%7D0Ahttp://pqdt.calis.edu.cn/Detail.aspx?pid=281.

[91] Dembia, C. L., Bianco, N. A., Falisse, A., Hicks, J. L., and Delp, S. L. "OpenSim Moco: Musculoskeletal optimal control". In: *bioRxiv* (2020), pp. 1–25. ISSN: 15537358. DOI: 10.1101/839381.

[92] Demircan, E., Besier, T. F., and Khatib, O. "Muscle force transmission to operational space accelerations during elite golf swings". In: *Proceedings - IEEE International Conference on Robotics and Automation* (2012), pp. 1464–1469. ISSN: 10504729. DOI: 10.1109/ICRA.2012.6225336.

[93] Dickinson, J. A., Cook, S. D., and Leinhardt, T. M. "The measurement of shock waves following heel strike while running". In: *Journal of Biomechanics* 18.6 (1985), pp. 415–422. ISSN: 00219290. DOI: 10.1016/0021-9290(85)90276-3.

[94] Dickmanns, E. D. and Well, K. H. "Approximate Solution of Optimal Control Problems Using Third-Order Hermite Polynomial Functions". In: *Optimization Techniques IFIP Technical Conference*. Springer, Berlin, Heidelberg, 1975, pp. 158–166. ISBN: 9783540071655. DOI: 10.1007/3-540-07165-2_21.

[95] Dugas, C., Bengio, Y., Bélisle, F., Nadeau, C., and Garcia, R. "Incorporating second-order functional knowledge for better option pricing". In: *Advances in Neural Information Processing Systems* (2001). ISSN: 10495258.

[96] Ebashi, S. and Endo, M. "Calcium ion and muscle contraction." In: *Progress in biophysics and molecular biology* 18 (1968), pp. 123–183. ISSN: 0079-6107. DOI: 10.1016/0079-6107(68)90023-0.

[97] Eisenberg, E., Hill, T. L., and Chen, Y. D. "Cross-bridge Model of Muscle Contraction: Quantitative Analysis". In: *Biophysical Journal* 29.6 (1980), pp. 195–227. ISSN: 00063495. DOI: 10.1016/S0006-3495(80)85126-5. URL: http://dx.doi.org/10.1016/S0006-3495(80)85126-5.

[98] El Ouaaid, Z., Shirazi-Adl, A., Arjmand, N., and Plamondon, A. "Coupled objective function to study the role of abdominal muscle forces in lifting using the kinematics-driven model". In: *Computer Methods in Biomechanics and Biomedical Engineering* 16.1 (2013), pp. 54–65. ISSN: 10255842. DOI: 10.1080/10255842.2011.607441.

[99] Elmer, S. J., Barratt, P. R., Korff, T., and Martin, J. C. "Joint-specific power production during submaximal and maximal cycling". In: *Medicine and Science in Sports and Exercise* 43.10 (2011), pp. 1940–1947. ISSN: 01959131. DOI: 10.1249/MSS.0b013e31821b00c5.

[100] Elnagar, G., Kazemi, M. A., and Razzaghi, M. "The Pseudospectral Legendre Method for Discretizing Optimal Control Problems". In: *IEEE Transactions on Automatic Control* 40.10 (1995), pp. 1793–1796.

[101] Epic. *Unreal Engine*. 2019.

[102] Erez, T., Tassa, Y., and Todorov, E. "Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX". In: *Proceedings - IEEE International Conference on Robotics and Automation* 2015-June.June (2015), pp. 4397–4404. ISSN: 10504729. DOI: 10.1109/ICRA.2015.7139807.

[103] Ericson, M. O. "Mechanical muscular power output and work during ergometer cycling at different work loads and speeds". In: *European Journal of Applied Physiology and Occupational Physiology* 57.4 (1988), pp. 382–387. ISSN: 03015548. DOI: 10.1007/BF00417980.

[104] Fahroo, F. and Ross, I. M. "Costate estimation by a Legendre pseudospectral method". In: *Journal of Guidance, Control, and Dynamics* 24.2 (2001), pp. 270–277.

[105] Fahroo, F. and Ross, I. M. "Advances in pseudospectral methods for optimal control". In: *AIAA Guidance, Navigation and Control Conference and Exhibit* (2008). DOI: 10.2514/6.2008-7309.

[106] Fahroo, F. and Ross, I. M. "Pseudospectral methods for infinite-horizon optimal control problems". In: *Journal of Guidance, Control, and Dynamics* 31.4 (2008), pp. 927–936. ISSN: 15333884. DOI: 10.2514/1.33117.

[107] Falisse, A., Serrancolí, G., and De Groote, F. *Optimal control in biomechanics*. 2021.

[108] Falisse, A., Serrancolí, G., Dembia, C. L., Gillis, J., Jonkers, I., and De Groote, F. "Rapid predictive simulations with complex musculoskeletal models suggest that diverse healthy and pathological human gaits can emerge from similar control strategies". In: *Journal of the Royal Society Interface* 16.157 (2019). ISSN: 17425662. DOI: 10.1098/rsif.2019.0402.

[109] Falugi, P., Kerrigan, E., and Van Wyk, E. *Imperial College London Optimal Control Software User Guide (ICLOCS)*. London, England, UK, 2010.

[110] Febrer-Nafría, M., Pallarès-López, R., Fregly, B. J., and Font-Llagunes, J. M. "Comparison of different optimal control formulations for generating dynamically consistent crutch walking simulations using a torque-driven model". In: *Mechanism and Machine Theory* 154 (2020). ISSN: 0094114X. DOI: 10.1016/j.mechmachtheory.2020.104031.

[111] Febrer-Nafría, M., Pallarès-López, R., Fregly, B. J., and Font-Llagunes, J. M. "Prediction of three-dimensional crutch walking patterns using a torque-driven model". In: *Multibody System Dynamics* 51.1 (2021). ISSN: 1573272X. DOI: 10.1007/s11044-020-09751-z.

[112] Felton, P. J., Yeadon, M. R., and King, M. A. "Are planar simulation models affected by the assumption of coincident joint centers at the hip and shoulder?" In: *Journal of Applied Biomechanics* 35.2 (2019), pp. 157–163. ISSN: 15432688. DOI: 10.1123/jab.2018-0136.

[113] Fike, J. A. and Alonso, J. J. "The Development of Hyper-Dual Numbers for Exact Second-Derivative Calculations". In: *49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*. January. 2011. DOI: 10.2514/6.2011-886.

[114] Fletcher, R. "A new approach to variable metric algorithms". In: *The Computer Journal* 13.3 (1970), pp. 317–322.

[115] Fornberg, B. "Numerical Differentiation of Analytic Functions". In: *ACM Transactions on Mathematical Software (TOMS)* 7.4 (1981), pp. 512–526. ISSN: 15577295. DOI: 10.1145/355972.355979.

[116] Forrester, S. E., Yeadon, M. R., King, M. A., and Pain, M. T. "Comparing different approaches for determining joint torque parameters from isovelocity dynamometer measurements". In: *Journal of Biomechanics* 44.5 (2011), pp. 955–961. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2010.11.024. URL: http://dx.doi.org/10.1016/j.jbiomech.2010.11.024.

[117] Fregly, B. J. "Design of Optimal Treatments for Neuromusculoskeletal Disorders using Patient-Specific Multibody Dynamic Models." In: *International journal for computational vision and biomechanics* 2.2 (2009), pp. 145–155. ISSN: 0973-6778. URL: http://www.ncbi.nlm.nih.gov/pubmed/21785529%7B%5C%%7D5Cnhttp://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC3141573.

[118] Fregly, B. J., Fregly, C. D., and Kim, B. T. "Computational prediction of muscle moments during ARED squat exercise on the international space station". In: *Journal of Biomechanical Engineering* 137.12 (2015), pp. 1–8. ISSN: 15288951. DOI: 10.1115/1.4031795.

[119] Fregly, B. J., Reinbolt, J. A., Rooney, K. L., Mitchell, K. H., and Chmielewski, T. L. "Erratum: Design of patient-specific gait modifications for knee osteoarthritis rehabilitation (IEEE Transactions on Biomedical Engineering (2007))". In: *IEEE Transactions on Biomedical Engineering* 54.10 (2007), p. 1905. ISSN: 00189294. DOI: 10.1109/TBME.2007.907637.

[120] Fregly, B. J. and Zajac, F. E. "A state-space analysis of mechanical energy generation, absorption, and transfer during pedaling". In: *Journal of Biomechanics* 29.3 (1996), pp. 81–90.

[121] Garg, D., Patterson, M. A., Francolin, C., Darby, C. L., Huntington, G. T., Hager, W. W., and Rao, A. V. "Direct trajectory optimization and costate estimation of finite-horizon and infinite-horizon optimal control problems using a Radau pseudospectral method". In: *Computational Optimization and Applications* 49.2 (2009), pp. 335–358. ISSN: 09266003. DOI: 10.1007/s10589-009-9291-0.

[122] Garg, D., Patterson, M. A., Hager, W. W., Rao, A. V., Benson, D. A., and Huntington, G. T. "A unified framework for the numerical solution of optimal control problems using pseudospectral methods". In: *Automatica* 46.11 (2010), pp. 1843–1851. ISSN: 00051098. DOI: 10.1016/j.automatica.2010.06.048. URL: http://dx.doi.org/10.1016/j.automatica.2010.06.048.

[123] Garner, B. A. and Pandy, M. G. "The Obstacle-Set Method for Representing Muscle Paths in Musculoskeletal Models". In: *Computer Methods in Biomechanics and Biomedical Engineering* 3.1 (2000), pp. 1–30. ISSN: 1476-8259. DOI: 10.1080/10255840008915251. URL: http://dx.doi.org/10.1080/10255840008915251%7B%5C%%7D5Cnhttp://www.tandfonline.com/doi/pdf/10.1080/10255840008915251.

[124] Gear, W. C. *Numerical Initial-Value Problems in Ordinary Differential Equations*. Englewood Cliffs, New Jersey: Prentice-Hall, 1971.

[125] Gebremedhin, A. H., Manne, F., and Pothen, A. "What color is your Jacobian? Graph coloring for computing derivatives". In: *SIAM Review* 47.4 (2005), pp. 629–705. ISSN: 00361445. DOI: 10.1137/S0036144504444711.

[126] Geijtenbeek, T. "SCONE: Open Source Software for Predictive Simulation of Biological Motion". In: *Journal of Open Source Software* 4.38 (2019), p. 1421. ISSN: 2475-9066. DOI: 10.21105/joss.01421.

[127] Gerus, P., Sartori, M., Besier, T. F., Fregly, B. J., Delp, S. L., Banks, S. A., Pandy, M. G., D'Lima, D. D., and Lloyd, D. G. "Subject-specific knee joint geometry improves predictions of medial tibiofemoral contact forces". In: *Journal of Biomechanics* 46.16 (2013), pp. 2778–2786. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2013.09.005. URL: http://dx.doi.org/10.1016/j.jbiomech.2013.09.005.

[128] Gilchrist, L. A. and Winter, D. A. "A multisegment computer simulation of normal human gait". In: *IEEE Transactions on Rehabilitation Engineering* 5.4 (1997), pp. 290–299. ISSN: 10636528. DOI: 10.1109/86.650281.

[129] Gill, P. E., Murray, W., and Saunders, M. A. "SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization". In: *SIAM Journal on Optimization* 12.4 (2002), pp. 979–1006. ISSN: 1052-6234. DOI: 10.1137/S1052623499350013.

arXiv: 17444372724. URL: http://epubs.siam.org/doi/abs/10.1137/S1052623499350013.

[130] Gill, P. E., Murray, W., and Saunders, M. A. "SNOPT: An SQP algorithm for large-scale constrained optimization". In: *SIAM Review* 47.1 (2005), pp. 99–131. ISSN: 00361445. DOI: 10.1137/S0036144504446096.

[131] Gill, P. E., Murray, W., and Wright, M. H. *Practical Optimization*. Society for Industrial and Applied Mathematics, 1986.

[132] Goldfarb, D. "A Family of Variable-Metric Methods Derived by Variational Means". In: *Mathematics of Computation* 24.109 (1970), p. 23. ISSN: 00255718. DOI: 10.2307/2004873.

[133] Gong, Q., Fahroo, F., and Ross, I. M. "Spectral algorithm for pseudospectral methods in optimal control". In: *Journal of Guidance, Control, and Dynamics* 31.3 (2008), pp. 460–471. ISSN: 15333884. DOI: 10.2514/1.32908.

[134] Gonzalez, H. and Hull, M. L. "Multivariable optimization of cycling biomechanics". In: *Journal of Biomechanics* 22.11-12 (1989), pp. 1151–1161.

[135] Gordon, A. M., Huxley, A. F., and Julian, F. J. "The variation in isometric tension with sarcomere length in vertebrate muscle fibres." In: *The Journal of Physiology* 184.1 (1966), pp. 170–192. ISSN: 0022-3751, 1469-7793. DOI: 5921536. arXiv: 1111.6189v1. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=1357553%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract.

[136] Griewank, A. and Walther, A. *Evaluating Derivatives*. Society for Industrial and Applied Mathematics, Jan. 2008. ISBN: 978-0-89871-659-7. DOI: 10.1137/1.9780898717761. URL: http://epubs.siam.org/doi/book/10.1137/1.9780898717761.

[137] Gruber, K., Ruder, H., Denoth, J., and Schneider, K. "A comparative study of impact dynamics : wobbling mass model versus rigid body models". In: *Journal of Biomechanics* 31 (1998), pp. 439–444.

[138] Guillou, A. and Soulé, J. L. "La résolution numérique des problèmes différentiels aux conditions initiales par des méthodes de collocation". In: *Revue française d'informatique et de recherche opérationnelle. Série rouge* 3.3 (1969), pp. 17–44. ISSN: 0373-8000. DOI: 10.1051/m2an/196903r300171.

[139] Güneş Baydin, A., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. "Automatic differentiation in machine learning: A survey". In: *Journal of Machine Learning Research* 18 (2018), pp. 1–43. ISSN: 15337928. arXiv: 1502.05767.

[140] Gupta, D., Donnelly, C. J., and Reinbolt, J. A. "Optimizing whole-body kinematics using OpenSim Moco to reduce peak non-sagittal plane knee loads and ACL injury risk during single leg jump landing". In: 2020.

[141] Haas, J. K. "A history of the Unity game engine". In: *Worcester Polytechnic Institute* (2014).

[142] Hager, W. W., Hou, H., Mohapatra, S., Rao, A. V., and Wang, X. S. "Convergence rate for a Radau hp collocation method applied to constrained optimal control". In: *Computational Optimization and Applications* 74.1 (2019), pp. 275–314. ISSN: 15732894. DOI: 10.1007/s10589-019-00100-1. arXiv: 1605.02121.

[143] Hairer, E., Wanner, G., and Lubich, C. *Geometric Numerical Integration*. Vol. 31. Springer Series in Computational Mathematics. Berlin/Heidelberg, Germany: Springer-Verlag, 2006. ISBN: 3-540-30663-3. DOI: 10.1007/3-540-30666-8.

[144] Hamby, D. M. "A Review of Techniques for Parameter Sensitivity". In: *Environmental Monitoring and Assessment* 32.c (1994), pp. 135–154. URL: https://deepblue.lib.umich.edu/bitstream/handle/2027.42/42691/10661%7B%5C_%7D2004%7B%5C_%7DArticle%7B%5C_%7DBF00547132.pdf?sequence=1.

[145] Hamner, S. R. and Delp, S. L. "Muscle contributions to fore-aft and vertical body mass center accelerations over a range of running speeds". In: *Journal of Biomechanics* 46.4 (2013), pp. 780–787. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2012.11.024. arXiv: NIHMS150003. URL: http://dx.doi.org/10.1016/j.jbiomech.2012.11.024.

[146] Hamner, S. R., Seth, A., and Delp, S. L. "Muscle contributions to propulsion and support during running". In: *Journal of Biomechanics* 43.14 (2010), pp. 2709–2716. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2010.06.025. arXiv: 166. URL: http://dx.doi.org/10.1016/j.jbiomech.2010.06.025.

[147] Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Río, J. F. del, Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. ISSN: 0028-0836. DOI: 10.1038/s41586-020-2649-2. URL: http://www.nature.com/articles/s41586-020-2649-2.

[148] Hascoet, L. and Pascual, V. "The Tapenade automatic differentiation tool". In: *ACM Transactions on Mathematical Software* 39.3 (2013), pp. 1–43. ISSN: 0098-3500. DOI: 10.1145/2450153.2450158.

[149] Haselgrove, J. C. and Huxley, H. E. "X-ray evidence for radial cross-bridge movement and for the sliding filament model in actively contracting skeletal muscle". In: *Journal of Molecular Biology* 77.4 (1973). ISSN: 00222836. DOI: 10.1016/0022-2836(73)90222-2.

[150] Hatze, H. "The complete optimization of a human motion". In: *Mathematical Biosciences* 28.1-2 (1976), pp. 99–135. ISSN: 00255564. DOI: 10.1016/0025-5564(76)90098-5.

[151] Hatze, H. "A complete set of control equations for the human musculoskeletal system". In: *Journal of Biomechanics* 10.11-12 (1977), pp. 799–805. ISSN: 00219290. DOI: 10.1016/0021-9290(77)90094-X.

[152] Hatze, H. "A myocybernetic control model of skeletal muscle". In: *Biological Cybernetics* 25.2 (1977), pp. 103–119. ISSN: 03401200. DOI: 10.1007/BF00337268.

[153] Hatze, H. "A mathematical model for the computational determination of parameter values of anthropomorphic segments". In: *Journal of Biomechanics* 13.10 (1980), pp. 833–843. ISSN: 00219290. DOI: 10.1016/0021-9290(80)90171-2.

[154] Hatze, H. "A comprehensive model for human motion simulation and its application to the take-off phase of the long jump". In: *Journal of Biomechanics* 14.3 (1981), pp. 135–142. ISSN: 00219290. DOI: 10.1016/0021-9290(81)90019-1.

[155] Hatze, H. "A three-dimensional multivariate model of passive human joint torques and articular boundaries". In: *Clinical Biomechanics* 12.2 (1997), pp. 128–135. ISSN: 02680033. DOI: 10.1016/S0268-0033(96)00058-7.

[156] He, J., Levine, W. S., and Loeb, G. E. "Feedback Gains for Correcting Small Perturbations to Standing Posture". In: *IEEE Transactions on Automatic Control* 36.3 (1991), pp. 322–332. ISSN: 15582523. DOI: 10.1109/9.73565.

[157] Hertz, H. "On the contact of elastic solids". In: *Journal für die reine und angewandte Mathematik* 92 (1882), pp. 156–171.

[158] Herzog, W. and Leonard, T. R. "The history dependence of force production in mammalian skeletal muscle following stretch-shortening and shortening-stretch cycles". In: *Journal of Biomechanics* 33.5 (2000), pp. 531–542. ISSN: 00219290. DOI: 10.1016/S0021-9290(99)00221-3. arXiv: 184.

[159] Herzog, W. and Leonard, T. "Force enhancement following stretching of skeletal muscle: a new mechanism". In: *Journal of Experimental Biology* 202 (2002), pp. 1275–1283. ISSN: 0022-0949. DOI: 10.1016/S0021-9290(97)00079-1. arXiv: 185.

[160] Hicks, J. L., Schwartz, M. H., Arnold, A. S., and Delp, S. L. "Crouched postures reduce the capacity of muscles to extend the hip and knee during the single-limb stance phase of gait". In: *Journal of Biomechanics* 41.5 (2008), pp. 960–967. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2008.01.002.

[161] Hicks, J. L., Uchida, T. K., Seth, A., Rajagopal, A., and Delp, S. L. "Is my model good enough? Best practices for verification and validation of musculoskeletal models and simulations of human movement". In: *Journal of Biomechanical Engineering* 137.February (2015), p. 020905. ISSN: 1528-8951. DOI: 10.1115/1.4029304.

[162] Hill, A. V. "The Heat of Shortening and the Dynamic Constants of Muscle". In: *Proceedings of the Royal Society B: Biological Sciences* 126.843 (1938), pp. 136–195. ISSN: 0962-8452. DOI: 10.1098/rspb.1938.0050. arXiv: arXiv:1011.1669v3.

[163] Hill, T. L., Eisenberg, E., Chen, Y. D., and Podolsky, R. J. "Some self-consistent two-state sliding filament models of muscle contraction." In: *Biophysical journal* 15.4 (1975), pp. 335–372. ISSN: 00063495. DOI: 10.1016/S0006-3495(75)85823-1. URL: http://dx.doi.org/10.1016/S0006-3495(75)85823-1.

[164] Hinrichs, R. N. "Regression equations to predict segmental moments of inertia from anthropometric measurements: An extension of the data of Chandler et al. (1975)". In: *Journal of Biomechanics* 18.8 (1985), pp. 621–624. ISSN: 00219290. DOI: 10.1016/0021-9290(85)90016-8.

[165] Ho, J. Y. L. "Direct path method for flexible multibody spacecraft dynamics". In: *Journal of Spacecraft and Rockets* 14.2 (1977), pp. 102–110. ISSN: 00224650. DOI: 10.2514/3.57167.

[166] Hoang, H. X. and Reinbolt, J. A. "Posture Influences Ground Reaction Force: Implications for Crouch Gait". In: *Proceedings of 3D Analysis of Human Movement*. San Francisco, CA, USA, 2010.

[167] Hogan, N. "The mechanics of multi-joint posture and movement control". In: *Biological Cybernetics* 52.5 (1985), pp. 315–331. ISSN: 03401200. DOI: 10.1007/BF00355754.

[168] Hogan, R. J. "Fast reverse-mode automatic differentiation using expression templates in C++". In: *ACM Transactions on Mathematical Software* 40.4 (2014). ISSN: 15577295. DOI: 10.1145/2560359.

[169] Hollars, M. G., Rosenthal, D. E., and Sherman, M. A. "SD/FAST user's manual". In: *Symbolic Dynamics Inc* (1991).

[170] Hooker, W. W. "Equations of motion for interconnected rigid and elastic bodies: A derivation independent of angular momentum". In: *Celestial Mechanics* 11.3 (1975), pp. 337–359. ISSN: 00088714. DOI: 10.1007/BF01228811.

[171] Hou, H., Hager, W. W., and Rao, A. V. "Convergence of a Gauss pseudospectral method for optimal control". In: *AIAA Guidance, Navigation, and Control Conference 2012* August (2012), pp. 1–9. DOI: 10.2514/6.2012-4452.

[172] Houska, B., Ferreau, H. J., and Diehl, M. "ACADO Toolkit - An open-source framework for automatic control and dynamic optimization". In: *Optimal Control Applications and Methods* 32.3 (2011), pp. 298–312.

[173] Hoy, M. G., Zajac, F. E., and Gordon, M. E. "A musculoskeletal model of the human lower extremity: The effect of muscle, tendon, and moment arm on the moment-angle relationship of musculotendon actuators at the hip, knee, and ankle". In: *Journal of Biomechanics* 23.2 (1990), pp. 157–169. ISSN: 00219290. DOI: 10.1016/0021-9290(90)90349-8.

[174] Hubbard, M., Hibbard, R. L., Yeadon, M. R., and Komor, A. "A Multisegment Dynamic Model of Ski Jumping". In: *International Journal of Sport Biomechanics* 5.2 (2016), pp. 258–274. ISSN: 0740-2082. DOI: 10.1123/ijsb.5.2.258.

[175] Hull, M. L. and Gonzalez, H. "Bivariate optimization of pedalling rate and crank arm length in cycling". In: *Journal of Biomechanics* 21.10 (1988), pp. 839–849. ISSN: 00219290. DOI: 10.1016/0021-9290(88)90016-4.

[176] Hull, M. L., Gonzalez, H., and Redfield, R. "Optimization of Pedaling Rate in Cycling Using a Muscle Stress-Based Objective Function". In: *International Journal of Sports Biomechanics* 4.June 2016 (1988), pp. 1–20. ISSN: 0740-2082. DOI: 10.1123/ijsb.4.1.1.

[177] Hull, M. L. and Jorge, M. "A method for biomechanical analysis of bicycle pedalling". In: *Journal of Biomechanics* 18.9 (1985), pp. 631–644. ISSN: 00219290. DOI: 10.1016/0021-9290(85)90019-3.

[178] Hunter, J. D. "Matplotlib: A 2D Graphics Environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.55. URL: http://ieeexplore.ieee.org/document/4160265/.

[179] Huntington, G. T. and Rao, A. V. "Optimal reconfiguration of spacecraft formations using the gauss pseudospectral method". In: *Journal of Guidance, Control, and Dynamics* 31.3 (2008), pp. 689–698. ISSN: 15333884. DOI: 10.2514/1.31083.

[180] Hutchinson, J. R., Anderson, F. C., Blemker, S. S., and Delp, S. L. "Analysis of hindlimb muscle moment arms in Tyrannosaurus rex using a three-dimensional musculoskeletal computer model: implications for stance, gait, and speed". In: *Paleobiology* 31.4 (2005), p. 676. ISSN: 0094-8373. DOI: 10.1666/04044.1.

[181] Hutchinson, J. R., Rankin, J. W., Rubenson, J., Rosenbluth, K. H., Siston, R. A., and Delp, S. L. "Musculoskeletal modelling of an ostrich (Struthio camelus) pelvic limb: Influence of limb orientation onmuscular capacity during locomotion". In: *PeerJ* 2015.6 (2015), pp. 1–52. ISSN: 21678359. DOI: 10.7717/peerj.1001.

[182] Huxley, A. F. "The mechanism of muscular contraction". In: *Science* 164 (1969), pp. 1356–1366. ISSN: 0022-3751 (Print) 0022-3751 (Linking). DOI: 10.1113/jphysiol.1974.sp010740. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve%7B%5C&%7Ddb=PubMed%7B%5C&%7Ddopt=Citation%7B%5C&%7Dlist%7B%5C_%7Duids=4449057.

[183] Huxley, A. F. "Muscular Contraction". In: *The journal of Physiology* 243.1 (1974), pp. 1–43. ISSN: 00223751. DOI: 10.1113/jphysiol.1974.sp010740. URL: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve%7B%5C&%7Ddb=PubMed%7B%5C&%7Ddopt=Citation%7B%5C&%7Dlist%7B%5C_%7Duids=4449057.

[184] Jensen, R. H. and Davy, D. T. "An investigation of muscle lines of action about the hip: A centroid line approach vs the straight line approach". In: *Journal of Biomechanics* 8.2 (1975), pp. 103–110. ISSN: 00219290. DOI: 10.1016/0021-9290(75)90090-1.

[185] Johnson, K. L. *Contact mechanics*. Cambridge University Press, 1987.

[186] Joyce, G. C. and Rack, P. "Isotonic lengthening and shortening movements of cat soleus muscle." In: *The Journal of Physiology* 204.2 (1969), pp. 475–491. ISSN: 0140-6736. DOI: 10.1113/jphysiol.1969.sp008925. URL: http://jp.physoc.org/content/204/2/475.abstract.

[187] Jung, E., Lenhart, S., and Feng, Z. "Optimal control of treatments in a two-strain tuberculosis model". In: *Discrete and Continuous Dynamical Systems - Series B* 2.4 (2002), pp. 473–482. ISSN: 15313492. DOI: 10.3934/dcdsb.2002.2.473.

[188] Kameswaran, S. and Biegler, L. T. "Convergence rates for direct transcription of optimal control problems using collocation at Radau points". In: *Computational Optimization and Applications* 41.1 (2008), pp. 81–126. ISSN: 09266003. DOI: 10.1007/s10589-007-9098-9.

[189] Kane, T. R. and Levinson, D. A. "Formulation of Equations of Motion for Complex Spacecraft". In: *Journal of Guidance and Control* 3.2 (1980), pp. 99–112.

[190] Kane, T. R. and Levinson, D. A. "The Use of Kanes's Dynamical Equations in Robotics". In: *The International Journal of Robotics Research* 2.3 (1983), pp. 3–21. ISSN: 17413176. DOI: 10.1177/027836498300200301.

[191] Kane, T. R. and Levinson, D. A. *Dynamics Theory and Applications*. 1985, p. 402. ISBN: 0070378460. DOI: 10.1016/0094-114X(86)90059-5.

[192] Katz, B. "The relation between force and speed in muscular contraction". In: *Journal of Physiology* 96 (1939), pp. 45–64.

[193] Kautz, S. A. and Hull, M. L. "Dynamic optimization analysis for equipment setup problems in endurance cycling". In: *Journal of Biomechanics* 28.11 (1995), pp. 1391–1401. ISSN: 00219290. DOI: 10.1016/0021-9290(95)00007-5.

[194] Kelly, M. P. "An introduction to trajectory optimization: How to do your own direct collocation". In: *SIAM Review* 59.4 (2017), pp. 849–904. ISSN: 00361445. DOI: 10.1137/16M1062569.

[195] Kentel, B. B., King, M. A., and Mitchell, S. R. "Evaluation of a subject-specific, torque-driven computer simulation model of one-handed tennis backhand ground strokes". In: *Journal of Applied Biomechanics* 27.4 (2011), pp. 345–354. ISSN: 15432688. DOI: 10.1123/jab.27.4.345.

[196] King, M. A., Kentel, B. B., and Mitchell, S. R. "The effects of ball impact location and grip tightness on the arm, racquet and ball for one-handed tennis backhand groundstrokes". In: *Journal of Biomechanics* 45.6 (2012), pp. 1048–1052. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2011.12.028.

[197] King, M. A., Kong, P. W., and Yeadon, M. R. "Determining effective subject-specific strength levels for forward dives using computer simulations of recorded performances". In: *Journal of Biomechanics* 42.16 (2009), pp. 2672–2677. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2009.08.007. URL: http://dx.doi.org/10.1016/j.jbiomech.2009.08.007.

[198] King, M. A., Wilson, C., and Yeadon, M. R. "Evaluation of a Torque-Driven Model of Jumping for Height". In: *Journal of Applied Biomechanics* 22 (2006), pp. 264–274.

[199] King, M. A. and Yeadon, M. R. "Determining subject-specific torque parameters for use in a torque-driven simulation model of dynamic jumping". In: *Journal of Applied Biomechanics* 18.3 (2002), pp. 207–217. ISSN: 10658483. DOI: 10.1123/jab.18.3.207.

[200] Kinney, A. L., Besier, T. F., Silder, A., Delp, S. L., D'Lima, D. D., and Fregly, B. J. "Changes in in vivo knee contact forces through gait modification". In: *Journal of Orthopaedic Research* 31.3 (2013), pp. 434–440. ISSN: 07360266. DOI: 10.1002/jor.22240. arXiv: NIHMS150003.

[201] Kirk, D. E. *Optimal Control Theory: An Introduction*. New York: Dover Publications, 2004.

[202] Klein Horsman, M. D., Koopman, H. F. J. M., Helm, F. C. T. van der, Poliacu Prosé, L., and Veeger, H. E. J. "Morphological muscle and joint parameters for musculoskeletal modelling of the lower extremity". In: *Clinical Biomechanics* 22.2 (2007), pp. 239–247. ISSN: 02680033. DOI: 10.1016/j.clinbiomech.2006.10.003.

[203] Koelewijn, A. D. and Bogert, A. J. van den. "Joint contact forces can be reduced by improving joint moment symmetry in below-knee amputee gait simulations". In: *Gait and Posture* 49 (2016), pp. 219–225. ISSN: 18792219. DOI: 10.1016/j.gaitpost.2016.07.007. URL: http://dx.doi.org/10.1016/j.gaitpost.2016.07.007.

[204] Krekel, H., Oliveira, B., Pfannschmidt, R., Bruynooghe, F., Laugher, B., and Bruhin, F. *pytest*. 2004.

[205] Lai, K. L. and Crassidis, J. L. "Generalizations of the complex-step derivative approximation". In: *Collection of Technical Papers - AIAA Guidance, Navigation, and Control Conference 2006* 4 (2006), pp. 2540–2564. DOI: 10.2514/6.2006-6348.

[206] Lam, S. K., Pitrou, A., and Seibert, S. "Numba: a LLVM-based Python JIT compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM '15*. New York, New York, USA: ACM Press, 2015, pp. 1–6. ISBN: 9781450340052. DOI: 10.1145/2833157.2833162. URL: http://dl.acm.org/citation.cfm?doid=2833157.2833162.

[207] Langholz, J. B., Westman, G., and Karlsteen, M. "Musculoskeletal Modelling in Sports-Evaluation of Different Software Tools with Focus on Swimming". In: *Procedia Engineering* 147 (2016), pp. 281–287. ISSN: 18777058. DOI: 10.1016/j.proeng.2016.06.278. URL: http://dx.doi.org/10.1016/j.proeng.2016.06.278.

[208] Lantoine, G., Russell, R. P., and Dargent, T. "Using multicomplex variables for automatic computation of high-order derivatives". In: *ACM Transactions on Mathematical Software* 38.3 (2012), pp. 1–18. ISSN: 00653438.

[209] Lattner, C. and Adve, V. "{LLVM}: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, CA, USA, 2004, pp. 75–88.

[210] Leardini, A., Belvedere, C., Nardini, F., Sancisi, N., Conconi, M., and Parenti-Castelli, V. *Kinematic models of lower limb joints for musculo-skeletal modelling and optimization in gait analysis*. 2017. DOI: 10.1016/j.jbiomech.2017.04.029. URL: http://dx.doi.org/10.1016/j.jbiomech.2017.04.029.

[211] Ledzewicz, U. and Schattier, H. "Analysis of optimal controls for a mathematical model of tumour anti-angiogenesis". In: *Optimal Control Applications and Methods* 29.1 (2008), pp. 41–57. ISSN: 01432087. DOI: 10.1002/oca.814.

[212] Lee, L.-F. and Umberger, B. R. "Generating optimal control simulations of musculoskeletal movement using OpenSim and MATLAB". In: *PeerJ* 4 (2016). ISSN: 2167-8359. DOI: 10.7717/peerj.1638.

[213] Leineweber, D. B. "Efficient reduced SQP methods for the optimization of chemical processes described by large space DAE models". Ph.D. Universität Heidelberg, 1998.

[214] Levinson, D. A. and Kane, T. R. "AUTOLEV - A New Approach to Multibody Dynamics". In: *Multibody Systems Handbook*. Ed. by W. Schiehlen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 81–102. ISBN: 978-3-642-50995-7. DOI: 10.1007/978-3-642-50995-7_7. URL: https://doi.org/10.1007/978-3-642-50995-7%7B%5C_%7D7.

[215] Lewis, F. L. and Syrmos, V. L. *Optimal Control*. New York: John Wiley & Sons, 1995.

[216] Lewis, M. G., Yeadon, M. R., and King, M. A. "Are torque-driven simulation models of human movement limited by an assumption of monoarticularity?" In: *Applied Sciences* 11.3852 (2021). ISSN: 20763417. DOI: 10.3390/app11093852.

[217] Likins, P. W. "Point-connected rigid bodies in a topological tree". In: *Celestial Mechanics* 11.3 (1975), pp. 301–317. ISSN: 00088714. DOI: 10.1007/BF01228809.

[218] Lin, Y. C. and Pandy, M. G. "Three-dimensional data-tracking dynamic optimization simulations of human locomotion generated by direct collocation". In: *Journal of Biomechanics* 59 (2017), pp. 1–8. ISSN: 18732380. DOI: 10.1016/j.jbiomech.2017.04.038. URL: http://dx.doi.org/10.1016/j.jbiomech.2017.04.038.

[219] Liu, F., Hager, W. W., and Rao, A. V. "Adaptive mesh refinement method for optimal control using nonsmoothness detection and mesh size reduction". In: *Journal of the Franklin Institute* 352.10 (2015), pp. 4081–4106. ISSN: 00160032. DOI: 10.1016/j.jfranklin.2015.05.028. URL: http://dx.doi.org/10.1016/j.jfranklin.2015.05.028.

[220] Liu, F., Hager, W. W., and Rao, A. V. "Adaptive Mesh Refinement Method for Optimal Control Using Decay Rates of Legendre Polynomial Coefficients". In: *IEEE Transactions on Control Systems Technology* 26.4 (2018), pp. 1475–1483. ISSN: 10636536. DOI: 10.1109/TCST.2017.2702122.

[221] Liu, W. and Nigg, B. M. "A mechanical model to determine the influence of masses and mass distribution on the impact force during running". In: *Journal of Biomechanics* 33.2 (2000), pp. 219–224. ISSN: 00219290. DOI: 10.1016/S0021-9290(99)00151-7.

[222] Lund, M. E., Andersen, M. S., Zee, M. de, and Rasmussen, J. "Scaling of musculoskeletal models from static and dynamic trials". In: *International Biomechanics* 2.1 (2015), pp. 1–11. ISSN: 2333-5432. DOI: 10.1080/23335432.2014.993706. arXiv: arXiv:1401.4290v2. URL: http://www.tandfonline.com/doi/full/10.1080/23335432.2014.993706%7B%5C#%7Dabstract.

[223] Lyness, J. N. and Moler, C. B. "Numerical Differentiation of Analytic Functions". In: *SIAM Journal on Numerical Analysis* 4.2 (1967), pp. 202–210. ISSN: 15577295. DOI: 10.1145/355972.355979.

[224] Magid, A. and Law, D. J. "Myofibrils bear most of the resting tension in frog skeletal muscle." In: *Science (New York, N.Y.)* 230.4731 (1985), pp. 1280–1282. ISSN: 0036-8075. DOI: 10.1126/science.4071053.

[225] Manns, P., Sreenivasa, M., Millard, M., and Mombaur, K. "Motion Optimization and Parameter Identification for a Human and Lower Back Exoskeleton Model". In: *IEEE Robotics and Automation Letters* 2.3 (2017), pp. 1564–1570. ISSN: 23773766. DOI: 10.1109/LRA.2017.2676355. arXiv: 1803.05666.

[226] Mansour, J. M. and Audu, M. L. "The passive elastic moment at the knee and its influence on human gait". In: *Journal of Biomechanics* 19.5 (1986), pp. 369–373. ISSN: 00219290. DOI: 10.1016/0021-9290(86)90013-8.

[227] Margossian, C. C. and Jan, M. S. "A Review of Automatic Differentiation and its Efficient Implementation". In: (2019), pp. 1–32. arXiv: arXiv:1811.05031v2.

[228] Martin, J. C. and Brown, N. A. T. "Joint-specific power production and fatigue during maximal cycling". In: *Journal of Biomechanics* 42.4 (2009), pp. 474–479. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2008.11.015.

[229] Martins, J. R. R. A., Sturdza, P., and Alonso, J. J. "The complex-step derivative approximation". In: *ACM Transactions on Mathematical Software* 29.3 (2003), pp. 245–262. ISSN: 00983500. DOI: 10.1145/838250.838251.

[230] Mashima, H. "Force-velocity relation and contractility in striated muscles." In: *The Japanese journal of physiology* 34.1 (1984), pp. 1–17. ISSN: 0021-521X. DOI: 10.2170/jjphysiol.34.1.

[231] MATLAB. *Matlab version R2020a*. Natick, MA, USA, 2020.

[232] Mayer, A. "Zur Aufstellung der Kriterien des Maximums und Minimums der einfachen Integrale bei variabeln Grenwerten". In: *Leipziger Berichte* 36 (1884), pp. 99–128.

[233] McDaniel, J., Behjani, S. N., Elmer, S. J., Brown, N. A. T., and Martin, J. C. "Joint-specific power-pedaling rate relationships during maximal cycling". In: *Journal of Applied Biomechanics* 30.3 (2014), pp. 423–430. ISSN: 15432688. DOI: 10.1123/jab.2013-0246.

[234] McErlain-Naylor, S. A., King, M. A., and Felton, P. J. "A review of forward-dynamics simulation models for predicting optimal technique in maximal effort sporting movements". In: *Applied Sciences (Switzerland)* 11.4 (2021), pp. 1–20. ISSN: 20763417. DOI: 10.3390/app11041450.

[235] McGowan, C. P., Neptune, R. R., and Herzog, W. "A phenomenological model and validation of shortening-induced force depression during muscle contractions". In: *Journal of Biomechanics* 43.3 (2010), pp. 449–454. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2009.09.047. arXiv: 183. URL: http://dx.doi.org/10.1016/j.jbiomech.2009.09.047.

[236] McGowan, C. P., Neptune, R. R., and Herzog, W. "A phenomenological muscle model to assess history dependent effects in human movement". In: *Journal of Biomechanics* 46.1 (2013), pp. 151–157. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2012.10.034. URL: http://dx.doi.org/10.1016/j.jbiomech.2012.10.034.

[237] McMahon, T. A. *Muscles, Reflexes, and Locomotion*. Princeton University Press, 1984.

[238] Mena, D., Mansour, J. M., and Simon, S. R. "Analysis and synthesis of human swing leg motion during gait and its clinical applications". In: *Journal of Biomechanics* 14.12 (1981), pp. 823–832. ISSN: 00219290. DOI: 10.1016/0021-9290(81)90010-5.

[239] Menegaldo, L. L., Fleury, A. D. T., and Weber, H. I. "Biomechanical modeling and optimal control of human posture". In: *Journal of Biomechanics* 36.11 (2003), pp. 1701–1712. ISSN: 00219290. DOI: 10.1016/S0021-9290(03)00170-2.

[240] Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Roučka, Š., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., and Scopatz, A. "SymPy: symbolic computing in Python". In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. URL: https://peerj.com/articles/cs-103.

[241] Meyer, A. J., Eskinazi, I., Jackson, J. N., Rao, A. V., Patten, C., and Fregly, B. J. "Muscle Synergies Facilitate Computational Prediction of Subject-Specific Walking Motions". In: *Frontiers in Bioengineering and Biotechnology* 4.October (2016), p. 77. ISSN: 2296-4185. DOI: 10.3389/fbioe.2016.00077. URL: http://journal.frontiersin.org/article/10.3389/fbioe.2016.00077/full.

[242] Millard, M., Uchida, T. K., Seth, A., and Delp, S. L. "Flexing computational muscle: modeling and simulation of musculotendon dynamics". In: *J Biomech Eng* 135.2 (2013), p. 21005. ISSN: 1528-8951. DOI: 10.1115/1.4023390. URL: http://www.ncbi.nlm.nih.gov/pubmed/23445050%7B%5C%%7D5Cnhttp://biomechanical.asmedigitalcollection.asme.org/article.aspx?articleid=1666657%7B%5C%%7D5Cnhttp://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3705831%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract.

[243] Miller, R. H. and Hamill, J. "Optimal footfall patterns for cost minimization in running". In: *Journal of Biomechanics* 48.11 (2015), pp. 2858–2864. ISSN: 18732380. DOI: 10.1016/j.jbiomech.2015.04.019. URL: http://dx.doi.org/10.1016/j.jbiomech.2015.04.019.

[244] Millman, K. J. and Aivazis, M. "Python for Scientists and Engineers". In: *Computing in Science & Engineering* 13.2 (Mar. 2011), pp. 9–12. ISSN: 1521-9615. DOI: 10.1109/MCSE.2011.36. URL: http://ieeexplore.ieee.org/document/5725235/.

[245] Mochon, S. and McMahon, T. A. "Ballistic walking: an improved model". In: *Mathematical Biosciences* 52.3-4 (1980), pp. 241–260. ISSN: 00255564. DOI: 10.1016/0025-5564(80)90070-X. arXiv: arXiv:1011.1669v3.

[246] Moore, J. K. *PyDy*. 2016.

[247] Moore, J. K. and Bogert, A. J. van den. "opty: Software for trajectory optimization and parameter identification using direct collocation". In: *The Journal of Open Source Software* 3.21 (2018), p. 300. ISSN: 2475-9066. DOI: 10.21105/joss.00300.

[248] Moore, K. L., DalleyII, A. F., and Agur, A. M. R. *Clinically Oriented Anatomy, 7th Edition.* 7th ed. Vol. 27. 2. Lippincott Williams & Wilkins, 2014. ISBN: 9781451119459. DOI: 10.1002/ca.22316. arXiv: arXiv:1011.1669v3.

[249] Moulton, F. R. *New methods in exterior ballistics.* Chicago: University of Chicago Press, 1926.

[250] Neptune, R. R. "Optimization algorithm performance in determining optimal controls in human movement analyses." In: *Journal of Biomechanical Engineering* 121.2 (1999), pp. 249–252.

[251] Neptune, R. R. and Hull, M. L. "Evaluation of performance criteria for simulation of submaximal steady-state cycling using a forward dynamic model." In: *Journal of biomechanical engineering* 120.3 (1998), pp. 334–41. ISSN: 0148-0731. DOI: 10.1115/1.2797999. URL: http://www.ncbi.nlm.nih.gov/pubmed/10412400.

[252] Neptune, R. R. and Hull, M. L. "A theoretical analysis of preferred pedaling rate selection in endurance cycling". In: *Journal of Biomechanics* 32.4 (1999), pp. 409–415. ISSN: 00219290. DOI: 10.1016/S0021-9290(98)00182-1.

[253] Neptune, R. R. and Kautz, S. A. "Muscle activation and deactivation dynamics: the governing properties in fast cyclical human movement performance?" In: *Exercise and sport sciences reviews* 29.2 (2001), pp. 76–80. ISSN: 0091-6331. DOI: 10.1097/00003677-200104000-00007. arXiv: 00003677-200104000-00007 [10.1097]. URL: http://people.stfx.ca/smackenz/courses/DirectedStudy/Articles/Neptune%20and%20Kautz%202000%20Muscle%20Activation%20and%20Deactivation%20Dynamics.pdf.

[254] NVIDIA. *PhysX Physics Engine.* 2021. URL: https://github.com/NVIDIAGameWorks/PhysX.

[255] O'Neill, M. C., Umberger, B. R., Holowka, N. B., Larson, S. G., Reiser, P. J., and Slade, J. M. "Chimpanzee super strength and human skeletal muscle evolution". In: *Proceedings of the National Academy of Sciences of the United States of America* 114.28 (2017), pp. 7343–7348. ISSN: 10916490. DOI: 10.1073/pnas.1619071114.

[256] Oliphant, T. E. "Python for Scientific Computing". In: *Computing in Science & Engineering* 9.3 (2007), pp. 10–20. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.58. URL: http://ieeexplore.ieee.org/document/4160250/.

[257] Ong, C. F., Geijtenbeek, T., Hicks, J. L., and Delp, S. L. "Predicting gait adaptations due to ankle plantarflexor muscle weakness and contracture using physics-based musculoskeletal simulations". In: *PLoS computational biology* 15.10 (2019), e1006993. ISSN: 15537358. DOI: 10.1371/journal.pcbi.1006993.

[258] Pain, M. T. and Challis, J. H. "The influence of soft tissue movement on ground reaction forces, joint torques and joint reaction forces in drop landings". In: *Journal of Biomechanics* 39.1 (2006), pp. 119–124. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2004.10.036.

[259] Pandy, M. G. "Computer modeling and simulation of human movement". In: *Annals of Biomedical Engineering* 3.1 (2001), pp. 245–273. ISSN: 1523-9829. DOI: 10.1146/annurev.bioeng.3.1.245. URL: http://www.ncbi.nlm.nih.gov/pubmed/11447064.

[260] Pandy, M. G. and Berme, N. "Synthesis of human walking: A planar model for single support". In: *Journal of Biomechanics* 21.12 (1988), pp. 1053–1060. ISSN: 00219290. DOI: 10.1016/0021-9290(88)90251-5.

[261] Pandy, M. G., Zajac, F. E., Sim, E., and Levine, W. S. "An optimal control model for maximum-height human jumping". In: *Journal of Biomechanics* 23.12 (1990), pp. 1185–1198. ISSN: 00219290. DOI: 10.1016/0021-9290(90)90376-E.

[262] Patterson, M. A., Hager, W. W., and Rao, A. V. "A ph mesh refinement method for optimal control". In: *Optimal Control Applications and Methods* 36.4 (2015), pp. 398–421. ISSN: 10991514. DOI: 10.1002/oca.2114.

[263] Patterson, M. A. and Rao, A. V. "GPOPS-II: A MATLAB Software for Solving Multiple-Phase Optimal Control Problems Using hp-Adaptive Gaussian Quadrature Collocation Methods and Sparse Nonlinear Programming". In: *ACM Transactions on Mathematical Software* 41.1 (2012), pp. 1–39. ISSN: 00983500. DOI: 10.1145/1731022.1731032. arXiv: 1005.3014. URL: http://portal.acm.org/citation.cfm?doid=1731022.1731032.

[264] Patterson, M. A. and Rao, A. V. "Exploiting sparsity in direct collocation pseudospectral methods for solving optimal control problems". In: *Journal of Spacecraft and Rockets* 49.2 (2012), pp. 364–377. ISSN: 15336794. DOI: 10.2514/1.A32071.

[265] Piazza, S. and Delp, S. L. "Three-dimensional dynamic simulation of total knee replacement motion during a step-up task". In: *Journal of Biomechanical Engineering* 123.6 (2001), pp. 599–606. ISSN: 01480731. DOI: 10.1115/1.1406950.

[266] Piazza, S. J. "Muscle-driven forward dynamic simulations for the study of normal and pathological gait". In: *Journal of NeuroEngineering and Rehabilitation* 3 (2006), pp. 1–7. ISSN: 17430003. DOI: 10.1186/1743-0003-3-5.

[267] Pietz, J. A. "Pseudospectral Collocation Methods for the Direct Transcription of Optimal Control Problems". Masters. Rice University, 2003.

[268] Pontryagin, L. *The Mathematical Theory of Optimal Processes*. New York, 1962.

[269] Porsa, S., Lin, Y.-C., and Pandy, M. G. "Direct Methods for Predicting Movement Biomechanics Based Upon Optimal Control Theory with Implementation in OpenSim". In: *Annals of Biomedical Engineering* 44.8 (2016), pp. 2542–2557. ISSN: 1573-9686. DOI: 10.1007/s10439-015-1538-6. URL: http://link.springer.com/10.1007/s10439-015-1538-6%7B%5C%%7D5Cnhttp://www.ncbi.nlm.nih.gov/pubmed/26715209.

[270] Proske, U. and Morgan, D. L. "Tendon stiffness: Methods of measurement and significance for the control of movement. A review". In: *Journal of Biomechanics* 20.1 (1987), pp. 75–82. ISSN: 00219290. DOI: 10.1016/0021-9290(87)90269-7.

[271] Python Software Foundation. *Python Package Index - PyPI*. URL: https://pypi.org/project/pycollo/ (visited on 03/28/2021).

[272] Qiao, M. and Jindrich, D. L. "Leg joint function during walking acceleration and deceleration". In: *Journal of Biomechanics* 49.1 (2016). ISSN: 18732380. DOI: 10.1016/j.jbiomech.2015.11.022.

[273] Raasch, C. C. and Zajac, F. E. "Locomotor Strategy for Pedaling: Muscle Groups and Biomechanical Functions". In: *Journal of Neurophysiology* 82 (1999), pp. 515–525.

[274] Raasch, C. C., Zajac, F. E., Baoming, M., and Levine, W. S. "Muscle coordination of maximum-speed pedaling". In: *Journal of Biomechanics* 30.6 (1997), pp. 595–602.

[275] Rack, P. and Westbury, D. "The effects of length and stimulus rate on tension in the isometric cat soleus muscleq". In: *The Journal of physiology* 204.2 (1969), pp. 443–460. ISSN: 0022-3751. DOI: 10.1113/jphysiol.1969.sp008923.

[276] Rack, P. and Westbury, D. "Elastic properties of the cat soleus tendon and their functional importance." In: *The Journal of physiology* 347.1984 (1984), pp. 479–495. ISSN: 0022-3751. DOI: 10.1113/jphysiol.1984.sp015077. URL: http://jp.physoc.org/content/347/1/479.short.

[277] Rajagopal, A., Dembia, C. L., DeMers, M. S., Delp, D. D., Hicks, J. L., and Delp, S. L. "Full-Body Musculoskeletal Model for Muscle-Driven Simulation of Human Gait". In: *IEEE Transactions on Biomedical Engineering* 63.10 (2016), pp. 2068–2079. ISSN: 15582531. DOI: 10.1109/TBME.2016.2586891.

[278] Rajamani, K., Nolte, L., and Styner, M. "Bone morphing with statistical shape models for enhanced visualization". In: *SPIE Medical Imaging* 5367.February (2004), pp. 122–130. ISSN: 0277786X. DOI: 10.1117/12.535000.

[279] Rankin, J. W. and Neptune, R. R. "A theoretical analysis of an optimal chainring shape to maximize crank power during isokinetic pedaling". In: *Journal of Biomechanics* 41.7 (2008), pp. 1494–1502. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2008.02.015.

[280] Rankin, J. W. and Neptune, R. R. "The influence of seat configuration on maximal average crank power during pedaling: A simulation study". In: *Journal of Applied Biomechanics* 26.4 (2010), pp. 493–500. ISSN: 10658483. DOI: 10.1017/CBO9781107415324.004. arXiv: arXiv:1011.1669v3.

[281] Rankin, J. W., Rubenson, J., and Hutchinson, J. R. "Inferring muscle functional roles of the ostrich pelvic limb during walking and running using computer optimization". In: *Journal of the Royal Society Interface* 13.118 (2016). ISSN: 17425662. DOI: 10.1098/rsif.2016.0035.

[282] Rao, A. V. "A survey of numerical methods for optimal control". In: *Advances in the Astronautical Sciences* 135.1 (2009), pp. 497–528. ISSN: 1569-3953. DOI: 10.1515/jnum-2014-0003. URL: http://vdol.mae.ufl.edu/ConferencePublications/trajectorySurveyAAS.pdf.

[283] Rao, A. V., Benson, D. A., Darby, C., Patterson, M. A., Francolin, C., Sanders, I., and Huntington, G. T. "Algorithm 902: GPOPS, a Matlab software for solving multiple-phase Optimal Control Problems using the Gauss Pseudospectral Method". In: *ACM Transactions on Mathematical Software* 37.2 (2010), pp. 1–39.

[284] Rao, A. V. and Mease, K. D. "Eigenvector approximate dichotomic basis method for solving hyper-sensitive optimal control problems". In: *Optimal Control Applications and Methods* 21.1 (2000), pp. 1–19.

[285] Rasmussen, J., Damsgaard, M., Surma, E., Christensen, S. T., Zee, M. de, and Vondrak, V. "AnyBody - a software system for ergonomic optimization". In: *Fifth World Congress on Structural and Multidisciplinary Optimization, May 19-23, 2003, Lido di Jesolo - Venice, Italy* January (2003), 6–6 pp.

[286] Reinbolt, J. A., Seth, A., and Delp, S. L. "Simulation of human movement: Applications using OpenSim". In: *Procedia IUTAM* 2 (2011), pp. 186–198. ISSN: 22109838. DOI: 10.1016/j.piutam.2011.04.019. URL: http://dx.doi.org/10.1016/j.piutam.2011.04.019.

[287] Revels, J., Lubin, M., and Papamarkou, T. "Forward-Mode Automatic Differentiation in Julia". In: April (2016), pp. 7–10. arXiv: 1607.07892. URL: http://arxiv.org/abs/1607.07892.

[288] Richards, F. J. "A flexible growth function for empirical use". In: *Journal of Experimental Botany* 10.2 (1959), pp. 290–301. ISSN: 00220957. DOI: 10.1093/jxb/10.2.290.

[289] Riener, R. and Edrich, T. "Identification of passive elastic joint moments in the lower extremities". In: *Journal of Biomechanics* 32.5 (1999), pp. 539–544. ISSN: 00219290. DOI: 10.1016/S0021-9290(99)00009-3.

[290] Rohani, F., Richter, H., and Bogert, A. J. van den. "Optimal design and control of an electromechanical transfemoral prosthesis with energy regeneration". In: *PLoS ONE* 12.11 (2017), pp. 1–13. ISSN: 19326203. DOI: 10.1371/journal.pone.0188266.

[291] Roithmayr, C. M. and Hodges, D. H. *Dynamics: Theory and Application of Kane's Method*. New York, New York, USA: Cambridge University Press, 2016, pp. 1–511.

[292] Rosenthal, D. E. and Sherman, M. A. "High Performance Multibody Simulations via Symbolic Equation Manipulation and Kane'S Method". In: *Journal of the Astronautical Sciences* 34.3 (1986), pp. 223–239. ISSN: 00219142.

[293] Ryan, R. R. "ADAMS-Multibody system analysis software". In: *Multibody Systems Handbook* (1990), pp. 361–402.

[294] Sakawa, Y. "Trajectory Planning of a Free-Flying Robot by Using the Optimal Control". In: *Optimal Control Applications and Methods* 20.5 (1999), pp. 235–248. ISSN: 01432087. DOI: 10.1002/(SICI)1099-1514(199909/10)20:5<235::AID-OCA658>3.0.CO;2-I.

[295] Selbie, W. S. and Caldwell, G. E. "A simulation study of vertical jumpng from different starting postures". In: *Journal of Biomechanics* 29.9 (1996), pp. 1137–1146.

[296] Sellers, W. I., Margetts, L., Coria, R. A., and Manning, P. L. "March of the titans: The locomotor capabilities of sauropod dinosaurs". In: *PLoS ONE* 8.10 (2013). ISSN: 19326203. DOI: 10.1371/journal.pone.0078733.

[297] Seth, A., Sherman, M. A., Reinbolt, J. A., and Delp, S. L. "OpenSim: A musculoskeletal modeling and simulation framework for in silico investigations and exchange". In: *Procedia IUTAM* 2 (2011), pp. 212–232. ISSN: 22109838. DOI: 10.1016/j.piutam.2011.04.021. arXiv: 15334406. URL: http://dx.doi.org/10.1016/j.piutam.2011.04.021.

[298] Shanno, D. F. "Conditioning of Quasi-Newton Methods for Function Minimization". In: *Mathematics of Computation* 24.111 (1970), p. 647. ISSN: 00255718. DOI: 10.2307/2004840.

[299] Sherman, M. A., Seth, A., and Delp, S. L. "Simbody: Multibody dynamics for biomedical research". In: *Procedia IUTAM* 2 (2011), pp. 241–261. ISSN: 22109838. DOI: 10.1016/j.piutam.2011.04.023. arXiv: 15334406. URL: http://dx.doi.org/10.1016/j.piutam.2011.04.023.

[300] Shourijeh, M. S. and McPhee, J. "Forward Dynamic Optimization of Human Gait Simulations: A Global Parameterization Approach". In: *Journal of Computational and Nonlinear Dynamics* 9.3 (2014), p. 031018. ISSN: 1555-1415. DOI: 10.1115/1.4026266. URL: http://computationalnonlinear.asmedigitalcollection.asme.org/article.aspx?doi=10.1115/1.4026266.

[301] Shourijeh, M. S., Smale, K. B., Potvin, B. M., and Benoit, D. L. "A forward-muscular inverse-skeletal dynamics framework for human musculoskeletal simulations". In: *Journal of Biomechanics* 49.9 (2016). ISSN: 18732380. DOI: 10.1016/j.jbiomech.2016.04.007.

[302] Shrive, N. G., O'Connor, J. J., and Goodfellow, J. W. "Load-bearing in the knee joint." In: *Clin Orthop Relat Res.* 131 (1978), pp. 279–287. ISSN: 0009-921X. DOI: 10.1097/00003086-197803000-00046.

[303] Silder, A., Whittington, B., Heiderscheit, B., and Thelen, D. G. "Identification of passive elastic joint moment-angle relationships in the lower extremity". In: *Journal of Biomechanics* 40.12 (2007), pp. 2628–2635. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2006.12.017. arXiv: NIHMS150003.

[304] Smith, R. C. and Haug, E. J. "DADS-Dynamic Analysis and Design System". In: *Multibody Systems Handbook*. Ed. by W. Schiehlen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 161–179. ISBN: 978-3-642-50995-7. DOI: 10.1007/978-3-642-50995-7_11. URL: https://doi.org/10.1007/978-3-642-50995-7%7B%5C_%7D11.

[305] Smith, R. L. *The Open Dynamics Engine (ODE)*. 2021. URL: https://bitbucket.org/odedevs/ode/src/master/.

[306] Soest, A. J. van and Bobbert, M. F. "The contribution of muscle properties in the control of exprosive movements". In: *Biological Cybernetics* 69 (1993), pp. 195–204.

[307] Soest, A. J. van and Casius, R. L. J. "Which factors determine the optimal pedaling rate in sprint cycling?" In: *American College of Sports Medicine* (2000), pp. 1927–1934.

[308] Soest, A. J. van, Schwab, A. L., Bobbert, M. F., and van Ingen Schenau, G. J. "The Influence Gastrocnemius of the Biarticularity of the Muscle on Vertical-Jumping Achievement". In: *Journal of biomechanics* 26.1 (1993), pp. 1–8.

[309] Squire, W. and Trapp, G. "Using complex variables to estimate derivatives of real functions". In: *SIAM Review* 40.1 (1998), pp. 110–112. ISSN: 00361445. DOI: 10.1137/S003614459631241X.

[310] Srajer, F., Kukelova, Z., and Fitzgibbon, A. "A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning". In: *Optimization Methods and Software* 33.4-6 (2018), pp. 889–906. ISSN: 10294937. DOI: 10.1080/10556788.2018.1435651. arXiv: 1807.10129.

[311] Steele, K. M., Rozumalski, A., and Schwartz, M. H. "Muscle synergies and complexity of neuromuscular control during gait in cerebral palsy". In: *Developmental Medicine and Child Neurology* 57.12 (2015), pp. 1176–1182. ISSN: 14698749. DOI: 10.1111/dmcn.12826.Muscle. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4683117/pdf/nihms-697756.pdf.

[312] Strang, G. *Introduction to Linear Algebra, Volume 3*. Wellesley, MA, USA: Wellesley-Cambridge Press, 1993.

[313] Stryk, O. von. *User's Guide for DIRCOL Version 2.1: A Direct Collocation Method for the Numerical Solution of Optimal Control Problems*. Darmstadt, Germany, 1999.

[314] Stryk, O. von and Schlemmer, M. "Optimal Control of the Industrial Robot Manutec r3". In: *Computational Optimal Control* 115 (1994), pp. 367–382. DOI: 10.1007/978-3-0348-8497-6_30.

[315] Swoap, S. J., Caiozzo, V. J., and Baldwin, K. M. "Optimal shortening velocities for in situ power production of rat soleus and plantaris muscles". In: *American Journal of Physiology* 273.42 (1997), pp. 1057–1063.

[316] Thelen, D. G., Anderson, F. C., and Delp, S. L. "Generating dynamic simulations of movement using computed muscle control". In: *Journal of Biomechanics* 36.3 (2003), pp. 321–328. ISSN: 00219290. DOI: 10.1016/S0021-9290(02)00432-3. arXiv: S0021-9290(02)00432-3 [10.1016].

[317] Thompson, J. A., Tran, A. A., Gatewood, C. T., Shultz, R., Silder, A., and Delp, S. L. "Biomechanical effects of an injury prevention program in preadolescent female soccer athletes". In: *American Journal of Sports Medicine* 45.2 (2017), pp. 294–301.

[318] Uchida, T. K., Hicks, J. L., Dembia, C. L., and Delp, S. L. "Stretching your energetic budget: How tendon compliance affects the metabolic cost of running". In: *PLoS ONE* 11.3 (2016), pp. 1–19. ISSN: 19326203. DOI: 10.1371/journal.pone.0150378.

[319] Umberger, B. R., Gerritsen, K. G., and Martin, P. E. "Muscle fiber type effects on energetically optimal cadences in cycling". In: *Journal of Biomechanics* 39.8 (2006), pp. 1472–1479. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2005.03.025.

[320] Vinnars, E., Bergstom, J., and Furst, P. "Influence of the postoperative state on the intracellular free amino acids in human muscle tissue". In: *Annals of Surgery* 182.6 (1975), pp. 665–671. ISSN: 00034932. DOI: 10.1097/00000658-197512000-00001.

[321] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., Walt, S. J. van der, Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., and Mulbregt, P. van. "SciPy 1.0: fundamental algorithms for scientific computing in Python". In: *Nature Methods* 17.3 (Mar. 2020), pp. 261–272. ISSN: 1548-7091. DOI: 10.1038/s41592-019-0686-2. URL: http://www.nature.com/articles/s41592-019-0686-2.

[322] Vlases, W., Paris, S. W., Lajoie, R. M., Martens, M. J., and Hargraves, C. R. *Optimal Trajectories by Implicit Simulation*. OH, USA, 1990.

[323] Wächter, A. and Biegler, L. T. *On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming*. Vol. 106. 1. 2006, pp. 25–57. ISBN: 1010700405. DOI: 10.1007/s10107-004-0559-y.

[324] Walther, A., Griewank, A., and Vogel, O. "ADOL-C: Automatic Differentiation Using Operator Overloading in C++". In: *Pamm* 2.1 (2003), pp. 41–44. ISSN: 1617-7061. DOI: 10.1002/pamm.200310011.

[325] Weinstein, M. J., Patterson, M. A., and Rao, A. V. "Utilizing the algorithmic differentiation package adigator for solving optimal control problems using direct collocation". In: *AIAA Guidance, Navigation, and Control Conference, 2013* (2015), pp. 1–19. DOI: 10.2514/6.2015-1085.

[326] Winters, J. M. "An improved muscle-reflex actuator for use in large-scale neuromusculoskeletal models". In: *Annals of Biomedical Engineering* 23.4 (1995), pp. 359–374. ISSN: 00906964. DOI: 10.1007/BF02584437.

[327] Winters, J. M. and Stark, L. "Analysis of Fundamental Human Movement Patterns Through the Use of In-Depth Antagonistic Muscle Models". In: *IEEE Transactions on Biomedical Engineering* BME-32.10 (1985), pp. 826–839. ISSN: 15582531. DOI: 10.1109/TBME.1985.325498.

[328] Winters, J. M. and Stark, L. "Muscle models: What is gained and what is lost by varying model complexity". In: *Biological Cybernetics* 55.6 (1987), pp. 403–420. ISSN: 03401200. DOI: 10.1007/BF00318375.

[329] Wood, J. E. and Mann, R. W. "A sliding-filament cross-bridge ensemble model of muscle contraction for mechanical transients". In: *Mathematical Biosciences* 57.3-4 (1981), pp. 211–263. ISSN: 00255564. DOI: 10.1016/0025-5564(81)90105-X.

[330] Wright, K. "Some relationships between implicit Runge-Kutta, collocation and Lanczos τ methods, and their stability properties". In: *Bit Numerical Mathematics* 10.2 (1970), pp. 217–227. ISSN: 00063835. DOI: 10.1007/BF01936868.

[331] Yamaguchi, G. T. *Dynamic modeling of musculoskeletal motion: a vectorized approach for biomechanical analysis in three dimensions.* Springer Science & Business Media, 2005.

[332] Yamaguchi, G. T. and Zajac, F. E. "A planar model of the knee joint to characterize the knee extensor mechanism". In: *Journal of Biomechanics* 22.1 (1989), pp. 1–10. ISSN: 00219290. DOI: 10.1016/0021-9290(89)90179-6.

[333] Yamaguchi, G. T. and Zajac, F. E. "Restoring Unassisted Natural Gait to Paraplegics Via Functional Neuromuscular Stimulation: A Computer Simulation Study". In: *IEEE Transactions on Biomedical Engineering* 37.9 (1990), pp. 886–902. ISSN: 15582531. DOI: 10.1109/10.58599.

[334] Yeadon, M. R. "The Simulation of Aerial Movement - II.A Mathematical Inertia Model of the Human Body". In: *Journal of Biomechanics* 23.1 (1990), pp. 67–74. DOI: 10.1016/0021-9290(90)90370-i.

[335] Yeadon, M. R. and Hiley, M. J. "Twist limits for late twisting double somersaults on trampoline". In: *Journal of Biomechanics* 58 (2017), pp. 174–178. ISSN: 18732380. DOI: 10.1016/j.jbiomech.2017.05.002.

[336] Yeadon, M. R. and Hiley, M. J. "The limits of aerial and contact techniques for producing twist in reverse $1\frac{1}{2}$ somersault dives". In: *Human Movement Science* 66 (2019), pp. 390–398. ISSN: 18727646. DOI: 10.1016/j.humov.2019.05.010.

[337] Yeadon, M. R. and King, M. A. "Evaluation of a torque driven simulation model of tumbling". In: *Journal of Applied Biomechanics* 18 (2002), pp. 195–206.

[338] Yeadon, M. R. and King, M. A. "Computer simulation modelling in sport". In: *Biomechanical Evaluation of Movement in Sport and Exercise, 2nd edition.* London, England, UK: Routledge, 2018, pp. 221–254. ISBN: 9780415434683.

[339] Yeadon, M. R., King, M. A., Forrester, S. E., Caldwell, G. E., and Pain, M. T. "The need for muscle co-contraction prior to a landing". In: *Journal of Biomechanics* 43.2 (2010), pp. 364–369. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2009.06.058. URL: http://dx.doi.org/10.1016/j.jbiomech.2009.06.058.

[340] Yeadon, M. R., King, M. A., and Wilson, C. "Modelling the maximum voluntary joint torque/angular velocity relationship in human movement". In: *Journal of Biomechanics* 39.3 (2006), pp. 476–482. ISSN: 00219290. DOI: 10.1016/j.jbiomech.2004.12.012.

[341] Yeadon, M. R. and Morlock, M. "The appropriate use of regression equations for the estimation of segmental inertia parameters". In: *Journal of Biomechanics* 22.6-7 (1989), pp. 683–689. ISSN: 00219290. DOI: 10.1016/0021-9290(89)90018-3.

[342] Yoon, Y. S. and Mansour, J. M. "The passive elastic moment at the hip". In: *Journal of Biomechanics* 15.12 (1982), pp. 905–910. ISSN: 00219290. DOI: 10.1016/0021-9290(82)90008-2. arXiv: 190.

[343] Yoshihuku, Y. and Herzog, W. "Optimal design parameters of the bicycle-rider system for maximal muscle power output". In: *Journal of Biomechanics* 23.10 (1990), pp. 1069–1079. ISSN: 00219290. DOI: 10.1016/0021-9290(90)90322-T.

[344] Yoshihuku, Y. and Herzog, W. "Maximal muscle power output in cycling: a modelling approach." In: *Journal of sports sciences* 14.2 (1996), pp. 139–57. ISSN: 0264-0414. DOI: 10.1080/02640419608727696. URL: http://www.ncbi.nlm.nih.gov/pubmed/8737322.

[345] Zajac, F. E. "Muscle and tendon: properties, models, scaling, and application to biomechanics and motor control." In: *Critical reviews in biomedical engineering* 17.4 (1989), pp. 359–411.

[346] Zajac, F. E., Neptune, R. R., and Kautz, S. A. "Biomechanics and muscle coordination of human walking: Part I: Introduction to concepts, power transfer, dynamics and simulations". In: *Gait and Posture* 16.3 (2002), pp. 215–232. ISSN: 09666362. DOI: 10.1016/S0966-6362(02)00068-1.

[347] Zhang, Y., Ma, Y., and Liu, G. "Lumbar spinal loading during bowling in cricket: a kinetic analysis using a musculoskeletal modelling approach". In: *Journal of Sports Sciences* 34.11 (2016), pp. 1030–1035. ISSN: 1466447X. DOI: 10.1080/02640414.2015.1086014.

[348] Zhao, Y. and Tsiotras, P. "Density functions for mesh refinement in numerical optimal control". In: *Journal of Guidance, Control, and Dynamics* 34.1 (2011), pp. 271–277. ISSN: 15333884. DOI: 10.2514/1.45852.

[349] Zondervan, K. P., Bauer, T. P., Betts, J. T., and Huffman, W. P. "Solving the Optimal Control Problem Using a Nonlinear Programming Technique Part 3: Optimal Shuttle Reentry Trajectories". In: *Proceedings of the AIAA/AAS Astrodynamics Conference*. Seattle, WA, USA, 1984, AIAA–84–2039.