TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Flugsystemdynamik

Discrete Controls and Constraints in Optimal Control Problems

Dipl.-Ing. Univ. Rainer Matthias Rieck

Vollständiger Abdruck der von der Fakultät für Maschinenwesen der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr.-Ing. Manfred Hajek
Prüfer der Dissertation:	1. Univ.-Prof. Dr.-Ing. Florian Holzapfel
	2. Univ.-Prof. Dr. rer. nat. habil. Matthias Gerdts

Die Dissertation wurde am 05.10.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Maschinenwesen am 29.01.2017 angenommen.

Acknowledgement

First of all, I would like to thank Prof. Dr. Florian Holzapfel for his support and indispensable help during the preparation of this PhD thesis. Particularly, I would like to emphasize the cooperativeness and openness at the Institute of Flight System Dynamics. A very special thanks goes to Prof. Dr. rer. nat. Matthias Gerdts for supporting this dissertation and for chairing the Munich Aerospace research group.

Besides, my sincere thanks goes to Munich Aerospace for organizing my scholarship and for offering me fast and straightforward support in any administrative questions.

My colleagues contributed a lot to the success of this thesis. Among them, I want to highlight the flight trajectory optimization group members Maximilian Richter, Jakob Lenz, Matthias Bittner, Benedikt Grüter, Johannes Diepolder, and Patrick Piprek. We jointly developed the *FALCON.m* optimal control framework, led many fruitful discussions, and last but not least had a very pleasant working atmosphere.

This thesis would not have been possible without the love and support of my parents Elfriede and Hans, as well as my sister Christine, who accompanied me during the whole period of my studies that laid the foundation for this work. With all my heart, I thank my life partner Vera for her love and patience and for her support particularly during the months preceeding the submission and for the necessary distraction so that live didn't come off badly.

Abstract / Kurzfassung

Abstract

In this thesis, optimal control problems with discrete controls and dependent constraints are considered. Many dynamic systems have control inputs which can only take values from a fixed set. The switching structure of the discrete choices is subject to optimization. The resulting type of problem is called Mixed-Integer Optimal Control Problem (MIOCP). Additionally, constraint limits that depend on the optimized discrete control choice are taken into account. The problems are solved with direct optimal control methods using gradient based optimization algorithms. Therefore, discrete controls and constraints have to be reformulated continuously. Different approaches are stated and compared. In order to minimize the number of switches and to enforce discrete value feasibility, a novel switching cost penalty formulation is used. The resulting optimal control problems are solved in MATrix LABoratory (*MATLAB*) using the *FALCON.m* optimal control toolbox. It was developed as part of this thesis and enables the calculation of analytic derivatives of high fidelity and large scale problems. This is achieved through a novel derivative generation tool-chain. Furthermore, a discrete control toolbox extension for *FALCON.m* is created for user friendly problem definition. The method and toolbox developed are applied to a minimal lap time race track optimization with optimal gear switching sequence. Additionally, fuel minimal approaches subject to the optimal flap positions are solved.

Kurzfassung

In dieser Arbeit werden Optimalsteuerungsprobleme unter Berücksichtigung von diskreten Steuerungen und davon abhängigen Nebenbedingungen gelöst. Viele dynamische Systeme haben Steuergrößen, die nur Werte eines fixen diskreten Sets annehmen können. Die Schaltstruktur der diskreten Wahlmöglichkeiten wird durch die Optimierung ermittelt. Diese Problemklasse wird auch als ein gemischt ganzzahliges Optimalsteuerungsproblem bezeichnet. Zusätzlich müssen die Grenzen von Nebenbedingungen, die sich in Abhängigkeit der diskreten Steuergröße ändern, berücksichtigt werden. Die resultierenden Probleme werden mit direkten Methoden und Gradienten basierten Verfahren gelöst. Deshalb müssen die diskreten Steuerungen und Nebenbedingungen kontinuierlich umformuliert werden. Verschiedene Ansätze werden vorgestellt und verglichen. Um die Anzahl der Schaltvorgänge zu minimieren, wird eine neue Schaltkostenfunktion verwendet. Die Optimalsteuerungsprobleme werden in *MATLAB* mit dem *FALCON.m* Optimalsteuerungs-Werkzeug gelöst. Es wurde als Teil dieser Arbeit entwickelt und ermöglicht das Lösen von hoch-fidelen und sehr großen

Optimalsteuerungsproblemen. Dies wird durch einen neue Werkzeugkette zur Ableitungsgeneration bewerkstelligt. Des Weiteren werden die diskreten Steuerungen als benutzerfreundliche Erweiterung in das *FALCON.m* Werkzeug integriert. Die vorgestellten Methoden und Werkzeuge werden auf eine zeitminimale Rennstreckenoptimierung unter Berücksichtigung der Gangschaltung angewendet. Zusätzlich werden Sprit minimale Anflüge mit Optimierung der Landeklappenpositionen gelöst.

Contents

List of Figures

List of Tables

Acronyms

MATLAB MATrix LABoratory

3DOF Three-Degree Of Freedom

3DOF Three Degree of Freedom

ADEPT Automatic Differentiation using Expression Tables

ADEPT Automatic Differentiation using Expression Templates

ADiGator A MATLAB Automatic Differentiation Tool

ADiMat Automatisches Differenzieren für Matlab

ADOL-C Automatic Differentiation by OverLoading in C++

ADOL-C Automatic Differentiation by OverLoading in C++

AGL Above Ground Level

ATM Air Traffic Management

BFD Backward Finite Differences

BFGS Broyden-Fletcher–Goldfarb-Shanno

BT Build Trace

CFD Central Finite Differences

CG Center of Gravity

CPU Central Processing Unit

DFP Davidon-Fletcher-Powell

ECEF Earth Centered Earth Fixed

ECI Earth Centered Inertial

FBC Final Boundary Condition

FD Finite Differences

FFD Forward Finite Differences

GPS Global Positioning System

GPS Global Positioning System

HT Hyperbolic Tangent

ICAO International Civil Aviation Organisation

IP Interior Point

IPOPT Interior Point OPTimizer

ISA International Standard Atmosphere

ISS International Space Station

KERS Kinetic Energy Recovery System

KKT Karush-Kuhn-Tucker

LICQ Linear Independence Constraint Qualification

MEX MATLAB EXecutable

MINLP Mixed-Integer Non-Linear Program

MIOCP Mixed-Integer Optimal Control Problem

MP Multiple Phases

MPEC Mathematical Programm with Equilibrium Constraints

MPVC Mathematical Programm with Vanishing Constraint

NED North-East-Down

NIMA National Imagery and Mapping Ageny

NLP Non-Linear Program

OC Outer Convexification

OCP Optimal Control Problem

QAR Quick Access Recorder

RPM Revolutions Per Minute

SD Subsystem Derivative

SDB Subsystem Derivative Builder

SNOPT Sparse Nonlinear OPTimizer

SQP Sequential Quadratic Programming

SR1 Symmetric Ranke-One

VTT Variable Time Transformation

WGS84 World Geodetic System 1984

WORHP We Optimize Really Huge Problems

Symbols and Indices

Optimization and Optimal Control

0 Index specifying initial condition in calculus of variations.

A Active set of inequality constraints $\vec{g}$

H Hamiltonian

J_P Penalty cost function

J Objective function

J Functional integral in calculus of variations.

J Jacobian of system dynamics

J Bolza cost function

L Integrand function in the calculus of variations.

L Lagrange cost function

M Mayer cost function

Q Approximation of the optimization problem Hessian

R Offset vector

S Sensitivity matrix

T Scaling matrix

U Valid range of continuous control

α Scale of penalty cost function

α Step size of the descent / search direction $\vec{d}$

β Armijo rule step size reduction factor

δ Time of the Runge-Kutta substep

ϵ_{feas} Feasibility tolerance

ϵ_{opt} Optimality tolerance

ϵ_{opt} Optimality tolerance

η Barrier paramter of interior point method

i_{max} Maximum number of iterations

$\vec{Z}_{ini}$ Initial guess of optimization parameter vector

$\vec{Z}_{opt}$ Optimal solution of opimization parameter vector

$\hat{\vec{\lambda}}$ Optimal Lagrange multiplier of inequality constraint

$\hat{\vec{\mu}}$ Optimal multiplier of equality constraint

$\hat{\vec{z}}$ Optimal vector $\vec{z}$

$\mathcal{L}$ Lagrange function

σ Armijo rule steepness factor of step size acceptance condition

τ Normalized time variable

$\vec{F}$ Constraint vector of discretized optimal control problem

$\vec{K}$ Vector of Runge-Kutta scheme storing intermediate dynamic model evaluations

$\vec{Z}$ Vector of optimization variables of discretized optimal control problem

$\vec{\Psi}$ Scheme for numeric integration method

$\vec{\eta}_s$ Phase defect of interior point condition

$\vec{\eta}$ Phase defect

$\vec{\lambda}$ Lagrange multiplier of model dynamics equality constraint in indirect methods

$\vec{\lambda}$ Lagrange multiplier of inequality constraint

$\vec{\mu}$ Multiplier of equality constraint

$\vec{\nu}$ Search direction of multipliers in numeric optimization algorithm

$\vec{\nu}$ Multiplier of boundary condition in indirect methods

$\vec{\psi}_s$ Boundary condition of interior point condition of optimal control problem

$\vec{\psi}$ Boundary condition of optimal control problem

$\vec{d}$ Descent / search direction calculated in numeric optimization algorithm

$\vec{f}$ First order ordinary differential equation of model dynamics

$\vec{g}$ Path constraint in optimal control problem

$\vec{g}$ Inequality path constraint in optimal control problem

$\vec{g}$ Inequality constraint

$\vec{h}$ Equality path constraint in optimal control problem

$\vec{h}$ Equality constraint

$\vec{p}$ Parameter vector of dynamic model

$\vec{q}$ Vector valued function

$\vec{r}$ Deviation vector of current optimization vector $\vec{z}$

$\vec{u}$ Control vector of dynamic model

$\vec{x}$ State function in calculus of variations.

$\vec{x}$ State vector of dynamic model

$\vec{y}$ Delta of two iterates of optimization problem Jacobian

$\vec{z}$ Vector of optimization variables of parameter optimization problem

ζ Sparsity of a matrix (problem Jacobian and Hessian)

a Weighting matrix of Runge-Kutta scheme for intermediate state calculation

b Weighting vector of Runge-Kutta scheme for combined state derivative calculation

c Vector of Runge-Kutta scheme to determine the advancement in time of the current stage

f Index specifying final condition in calculus of variations.

h Step size between discretized time points

i Major iteration index

i Index of time step

j Inequality constraint Index

j Index of phase

j Iteration index of Runge-Kutta integration stage steps

k Equality constraint index

k Index of path constraint

k Index of initial state in multiple shooting approach

l_0 Multiplier of objective function J

l Iteration index used in Runge-Kutta method to calculate the intermediate state for dynamic model evaluations

m Number of Runge-Kutta stages

m Number of inequality constraints $\vec{g}$

n_p Number of parameters of the system dynamics

n_u Number of controls of the system dynamics

n_x Number of states of the system dynamics

n_z Number of optimization variables of the discretized optimal control problem

n_h Number of intervals the optimal control problem phase is discretized

n_{ph} Number of phases in the optimal control problem

n Number of optimization variables in $\vec{z}$ vector

p Number of equality constraints $\vec{h}$

q Order of Runge-Kutta integration scheme

s Slack variable of interior point method

t_0 Initial time

t_f Final time

t Time parameter in calculus of variations.

t Time variable

t_0 Initial time in calculus of variations.

t_f Final time in calculus of variations.

Mixed Integer Optimal Control

G Discrete control that shall vanish in the vanishing constraint approach

H Vanishing constraint control / switching function

M Matrix to map the weights of multiple discrete controls to the weights of a single discrete control

V Set of discrete control values

W Discrete control weight matrix that respresents all possible discrete control combinations (cartesian product)

δ Slack parameter used in Multi-Time Switching Cost penalty approach

γ Slack parameter used in the switching cost approach by Kirches

κ Relaxation variable or penalty scale parameter for the inner convexification / discrete control relaxation approach

κ Relaxation parameter for the vanishing constraint approach

σ Integer step representation of the discrete control weights

$\vec{\eta}$ Another discrete control value

$\vec{v}$ Discrete control value

$\vec{w}$ Vector of discrete control weights

$\vec{z}$ Optimization variable vector

a Gradient steepness multiplier for the hyperbolic tangent discrete control reformulation

k Iterate of discrete control choice

k Hyperbolic tangent discrete control reformulation summation iteration index

$J_{SC,i,k}$ Switching cost penalty cost function for current point i and discrete control k

δ Minor grid time discretization used in te Variable Time Transformation approach

n_η Number of discrete control values in the discrete set

n_v Number of discrete control choices

n_w Number of discrete control weights

$w_{i+1,k}$ Discrete control weight of the next time step

$w_{i,k}$ Discrete control weight of the current time steo

$w_{i-1,k}$ Discrete control weight of the previous time step

w Outer convexification or Variable Time Transformation discrete control weight

FALCON.m Toolbox

0 Initial indicator.

J Bolza cost function.

L Lagrange cost function

M Mayer cost function.

R Matrix function used to explain the Hessian chain rule update.

S Scaling matrix.

X Matrix input argument used to explain the Hessian chain rule update.

Y Matrix function used to explain the Hessian chain rule update.

$\dot{\vec{x}}$ Model state derivative vector

ϕ Scalar function used to explain the subsystem derivative method.

τ Normalized time.

$\tilde{\vec{f}}_J$ Scaled combined Bolza cost and constraint vector.

$\vec{c}$ Constant vector entering model, constraint, or cost function.

$\vec{f}_J$ Combined Bolza cost and constraint vector.

$\vec{f}$ Vector of nonlinear constraints.

$\vec{f}$ Vector function used to explain the subsystem derivative method.

$\vec{h}$ Point constraint vector.

$\vec{p}_{gm}$ Unified parameter vector.

$\vec{p}$ Model parameter vector

$\vec{u}$ Model control vector

$\vec{y}$ Model output vector

$\vec{z}$ Vector of optimization parameters.

f Final indicator.

f Entry of vector function used to explain the subsystem derivative method.

j Hessian block iterator.

m Model index used to determined model parameters.

r Scalar function used to explain the Hessian chain rule update.

t Real time.

x Scalar input argument used to explain the Hessian chain rule update.

y Scalar function used to explain the Hessian chain rule update.

z Entry of vector of optimization parameters.

A Placeholder matrix.

F Matrix function used to explain the subsystem derivative method.

G Matrix function used to explain the subsystem derivative method.

LB Lower bound indicator.

R Matrix function used to explain the subsystem derivative method.

UB Upper bound indicator.

X Matrix input argument of a function used to explain the subsystem derivative method.

Y Matrix input argument of a function used to explain the subsystem derivative method.

κ Slack or relaxation variable of vanishing constraint function.

λ Entry of Lagrange multiplier.

ψ Entry of vanishing constraint function.

$\vec{\alpha}$ Slack variable input in the discrete control outer convexification model.

$\vec{\lambda}$ Lagrange multiplier of constraint.

$\vec{\psi}$ Vanishing constraint function.

$\vec{\sigma}$ Combined vector of Lagrange multipliers of cost functions and constraints

$\vec{f_x}$ Model state derivative function.

$\vec{f_y}$ Model output function.

$\vec{g}$ Path constraint vector.

$\vec{p_c}$ Parameter vector entering a constraint or cost function.

$\vec{r}$ Offset vector.

$\vec{u_c}$ Subset of model control vector entering a constraint or cost function.

$\vec{v}$ Placeholder vector.

$\vec{w_c}$ Discrete control weights entering the vanishing constraint function.

$\vec{x_c}$ Subset of model state vector entering a constraint or cost function.

$\vec{x}$ Model state vector

$\vec{x}$ Vector input argument of a function.

$\vec{x}$ Vector input argument of a function used to explain the subsystem derivative method.

$\vec{y_c}$ Subset of model output vector entering a constraint or cost function.

f Entry of vector of nonlinear constraints.

f Scalar function.

g Entry of path constraint vector.

h Finite differences step size.

h Hessian w.r.t. user function inputs.

h Jacobian w.r.t. user function inputs.

j Local Hessian w.r.t. subsystem inputs.

j Local Jacobian w.r.t. subsystem inputs.

l_0 Lagrange multiplier of cost function.

m Number of rows of vector or matrix.

n_R Output size of R matrix function.

n_X Size of X matrix input argument.

n_Y Output size of Y matrix function.

n_h Number of time steps of a phase entering the point constraint.

n_t Number of time steps.

n Size of vector input argument.

n Number of columns of matrix.

p Number of rows of vector or matrix.

q Number of columns of matrix.

r Entry of offset vector.

t_0 Initial time.

t_f Final Time.

w_c Entry of discrete control weights entering the vanishing constraint function.

x Scalar input argument of a function.

Car Racing

A Effective flow surface $[m^2]$

R Wheel radius $[m]$

β Side slip angle between car longitudinal axes and the velocity vector of the center of gravity $[rad]$

δ Steering wheel angle $[rad]$

ϕ Normalized accelerator pedal position $[-]$

ψ Yaw angle $[rad/s]$

$\vec{r}$ Car position $[m]$

ξ Normalized brake pedal position $[-]$

g Gravitational acceleration $[m/s]$

v Car speed in center of gravity speed $[m/s]$

x Car x position $[m]$

y Car y position $[m]$

B_f Front tire stiffness factor PECEJKA

B_r Rear tire stiffness factor PECEJKA

C_f Front tire shape factor PECEJKA

C_r Rear tire shape factor PECEJKA

D_f Front tire peak value PECEJKA

D_r Rear tire peak value PECEJKA

E_f Front tire curvature factor PECEJKA

E_r Rear tire curvature factor PECEJKA

F_B Maximum brake force $[N]$

F_{Bf} Front wheel brake force $[N]$

F_{Br} Rear wheel brake force $[N]$

F_{Mr} Rear wheel engine force $[N]$

F_{Rf} Front wheel rolling friction $[N]$

F_{Rr} Rear wheel rolling friction $[N]$

F_{ax} Aerodynamic longitudinal force $[N]$

F_{ay} Aerodynamic side force $[N]$

F_{lf} Front wheel longitudinal force $[N]$

F_{lr} Rear wheel longitudinal force $[N]$

F_{sf} Front wheel side force $[N]$

F_{sr} Rear wheel side force $[N]$

I_{zz} Moment of inertia $[kg \cdot m^2]$

M_{mot} Engine torque $[Nm]$

M Mass $[kg]$

α_f Front wheel slip angle $[rad]$

α_r Rear wheel slip angle $[rad]$

ρ Air density $[kg/m^3]$

c_w Air drag coefficient $[-]$

e_{sp} Distance drag force mount to center of gravity $[m]$

f_R Speed dependent roll friction

f_1 Intermediate motor torque coefficient 1

f_2 Intermediate motor torque coefficient 2

f_3 Intermediate motor torque coefficient 3

f Index for car front

i_G Gear transmission ratio $[-]$

i_t Motor transmission ratio $[-]$

l_f Distance front wheel to center of gravity $[m]$

l_r Distance rear wheel to center of gravity $[m]$

l_w Half width of the car $[m]$

l Index for car left

n_{mot} Engine rounds per minute $[RPM]$

r Index for car rear

r Index of car right

v_f Front wheel speed $[m/s]$

v_r Rear wheel speed $[m/s]$

w_δ Steering angle rate $[rad/s]$

w_z Yaw angle rate $[rad/s]$

w_{mot} Engine Angular rotation speed $[rad/s]$

Aircraft Dynamics

A Index of the aerodynamic reference frame

C_F Fuel flow coefficient $[-]$

C_L Aircraft lift coefficient. $[-]$

C_T Thrust coefficient $[-]$

$C_{D,0}$ Aircraft parasitic drag coefficient. $[-]$

$C_{D,2}$ Aircraft lift induced drag coefficient. $[-]$

$C_{D,min}$ Aircraft minimum drag coefficient. $[-]$

C_D Aircraft drag coefficient. $[-]$

$C_{L,\alpha}$ Aircraft linear slope of the lift coefficient w.r.t. the angle of attack. $[1/rad]$

$C_{L,max,LDG}$ Aircraft maximum lift coefficient in the landing configuration. $[-]$

$C_{L,max}$ Aircraft maximum lift coefficient. $[-]$

C_{L0} Aircraft lift coefficient for zero angle of attack. $[-]$

C_Q Aircraft side force coefficient. $[-]$

D Aircraft drag force. $[N]$

D Ekman length. $[-]$

E Index of the earth centered earth fixed reference frame

GM Earth's gravitational constant $[m^3/s^2]$

G Aircraft center of gravity

H_G Geopotential altitude of aircraft. $[m]$

H_W Intermediate height in the wind calculation. $[m]$

I Index of the earth centered inertial reference frame

$J_{p,HL}$ Artificial high lift penalty. $[-]$

K_δ Thrust lever dynamics PT1 gain. $[-]$

K_m Diffuse coefficient. $[m^2/s]$

K Index of the kinematic reference frame

L_{HV} Fuel lower heating value $[m^2/s^2]$

L Aircraft lift force. $[N]$

M_W Wind rotation matrix.

M_ϕ Describes the curvature of a point on the World Geodetic System 1984 ellipsoid with a tangent circle of this radius in the meridian plane. $[m]$

M_{AO} Transformation of the north east down frame to the aerodynamic frame $[-]$

M_{KE} Transformation of the earth centered earth fixed frame to the kinematic frame $[-]$

M_{KO} Transformation of the north east down frame to the kinematic frame $[-]$

M_{OE} Transformation matrix from the earth centered earth fixed frame to the north east down frame $[-]$

M Aircraft Mach number. $[-]$

N_ϕ Distance from a point on the World Geodetic System 1984 ellipsoid perpendicular to the surface to the polar axis. $[m]$

O Index of the north east down reference frame

P World Geodetic System 1984 position above the ellipsoid

Q Aircraft side force. $[N]$

Q World Geodetic System 1984 reference point on the ellipsoid

R Universal gas constant. $[J/(K \cdot kg)]$

S_{ref} Aircraft wing reference area. $[m^2]$

T_δ Thrust lever dynamics PT1 time constant. $[s]$

T_s Air temperature at sea level. $[K]$

T Aircraft thrust force. $[N]$

T Air temperature at aircraft altitude. $[K]$

V_K^G Aircraft kinematic speed $[m/s]$

V_A Aircraft aerodynamic speed. $[m/s]$

$V_{A,stall}$ Aircraft aerodynamic stall speed. $[m/s]$

V_{APP} Aircraft final approach speed, landing speed. $[m/s]$

$V_{CAS,max}$ Aircraft maximum calibrated air speed. $[m/s]$

V_{CAS} Aircraft calibrated air speed. $[m/s]$

V_{HW} Speed of aircraft head wind. $[m/s]$

V_{LS} Aircraft minimal selectable speed. $[m/s]$

$V_{W,Pr}$ Wind speed in the Prandl layer. $[m/s]$

$V_{W,g}$ Geostrophic wind speed in high altitudes. $[m/s]$

V_W Wind speed at aircraft altitude.

W_{MTOW} Maximum takeoff weight $[N]$

$\alpha_{W,0}$ Angle of the wind rotation in the Ekman layer. $[rad]$

α_{max} Aircraft maximum angle of attack. $[rad]$

α Aircraft angle of attack. $[rad]$

$\bar{A}$ Index of the rotated aerodynamic reference frame

$\bar{q}$ Dynamic pressure. $[kg/(m \cdot s^2)]$

χ_K^G Aircraft kinematic flight path angle $[rad]$

χ_W Wind course angle. $[-]$

δ_T Thrust lever position. $[-]$

δ_{HL} Aircraft high lift setting. $[-]$

δ_{LG} Aircraft landing gear setting. $[-]$

δ_{SB} Aircraft speed brake setting. $[-]$

$\delta_{T,CMD}$ Thrust lever command. $[-]$

$\dot{V}_K^G$ Time derivative of the aircraft kinematic speed. $[m/s]$

$\dot{V}_A$ Time derivative of aircraft aerodynamic speed. $[m/s^2]$

$\dot{V}_{CAS,nm}$ Aircraft calibrated air speed change w.r.t. nautical mile distance traveled (Speed change is given in knots). $[kts/nm]$

$\dot{V}_{CAS}$ Time derivative of aircraft calibrated air speed. $[m/s^2]$

$\dot{\chi}_K^G$ Time derivative of the aircraft kinematic course angle. $[rad/s]$

$\dot{\delta}_T$ Time derivative of thrust lever position. $[1/s]$

$\dot{\gamma}_K^G$ Time derivative of the aircraft kinematic climbing angle. $[rad/s]$

$\dot{\lambda}^G$ Time derivative of aircraft longitude position. $[rad/s]$

$\dot{h}^G$ Time derivative of aircraft altitude above World Geodetic System 1984 ellipsoid. $[m/s]$

$\dot{u}_A^G$ x component of aerodynamic acceleration. $[m/s^2]$

$\dot{u}_K^G$ x component of kinematic acceleration. $[m/s^2]$

$\dot{v}_A^G$ y component of aerodynamic acceleration. $[m/s^2]$

$\dot{v}_K^G$ y component of kinematic acceleration. $[m/s^2]$

$\dot{w}_A^G$ z component of aerodynamic acceleration. $[m/s^2]$

$\dot{w}_K^G$ z component of kinematic acceleration. $[m/s^2]$

η_T Air temperature ratio. $[-]$

η_ρ Air density ratio. $[-]$

η_p Air pressure ratio. $[-]$

$\left(\vec{\omega}^{EK}\right)_K$ Rotation of the kinematic frame w.r.t. the earth centered earth fixed frame. Coordinates are given in the kinematic frame. $[rad/s]$

$\left(\vec{\omega}^{EO}\right)_K$ Rotation of the north east down frame w.r.t. the earth centered earth fixed frame. Coordinates are given in the kinematic frame. $[rad/s]$

$\left(\vec{\omega}^{EO}\right)_O$ Rotation of the north east down frame w.r.t. the earth centered earth fixed frame. Coordinates are given in the north east down frame. $[rad/s]$

$\left(\vec{\omega}^{IE}\right)_E$ Rotation of the earth centered earth fixed frame w.r.t. the earth centered inertial frame with coordinates given in the earth centered earth fixed frame $[rad/s]$

$\left(\vec{\omega}^{OK}\right)_K$ Rotation of the kinematic frame w.r.t. the north east down frame. Coordinates are given in the kinematic frame. $[rad/s]$

$\left(\dot{\vec{\omega}}^{IE}\right)_E^E$ Time derivative of rotation of the earth centered fixed frame w.r.t. the earth centered inertial frame with coordinates given in the earth centered earth fixed frame $[rad/s^2]$

$\left(\vec{F}_A^G\right)_A$ Aircraft aerodynamic force vector acting in the center of gravity given in the aerodynamic frame. $[N]$

$\left(\vec{F}_A^G\right)_K$ Aircraft aerodynamic force vector acting in the center of gravity given in the kinematic frame. $[N]$

$\left(\vec{F}_G^G\right)_K$ Aircraft gravitational force vector acting in the center of gravity given in the kinematic frame. $[N]$

$\left(\vec{F}_G^G\right)_O$ Aircraft gravitational force vector acting in the center of gravity given in the north east down frame. $[N]$

$\left(\vec{F}_P^G\right)_A$ Aircraft thrust force vector acting in the center of gravity given in the aerodynamic frame. $[N]$

$\left(\vec{F}_P^G\right)_K$ Aircraft thrust force vector acting in the center of gravity given in the kinematic frame. $[N]$

$\left(\vec{F}_T^G\right)_K$ Aircraft total force vector acting in the center of gravity given in the kinematic frame. $[N]$

$\left(\vec{r}^G\right)_E$ Aircraft center of gravity position vector in the earth centered earth fixed frame $[m]$

$\left(\vec{v}_A^G\right)_O^E$ Aerodynamic speed vector w.r.t. the earth centered earth fixed frame. Coordinates are given in north east down frame. $[m/s]$

$\left(\vec{v}_K^G\right)_E^E$ Kinematic speed vector w.r.t. the earth centered earth fixed frame. Coordinates are given in the earth centered earth fixed frame. $[m/s]$

$\left(\vec{v}_K^G\right)_K^E$ Kinematic speed vector w.r.t. the earth centered earth fixed frame. Coordinates are given in the kinematic frame. $[m/s]$

$\left(\vec{v}_K^G\right)_O^E$ Kinematic speed vector w.r.t. the earth centered earth fixed frame. Coordinates are given in the north east down frame. $[m/s]$

$\left(\vec{v}_K^G\right)_E^I$ Kinematic speed vector w.r.t. the earth centered inertial frame. Coordinates are given in the earth centered earth fixed frame. $[m/s]$

$\left(\vec{v}_W^G\right)_O^E$ Wind speed vector w.r.t. the earth centered earth fixed frame. Coordinates are given in north east down frame. $[m/s]$

$\left(\dot{\vec{r}}_K^G\right)_E^I$ Time derivative of aircraft center of gravity position vector (kinematic) w.r.t. the earth centered inertial frame with coordinates given in the earth centered earth fixed frame $[m/s]$

$\left(\dot{\vec{v}}_A^G\right)_O^{EE}$ Time derivative of the aerodynamic speed vector (acceleration). The speed and acceleration are given w.r.t. the earth centered earth fixed frame. Coordinates are given in the north east down frame. $[m/s^2]$

$\left(\dot{\vec{v}}_K^G\right)_E^{EE}$ Time derivative of the kinematic speed vector (acceleration). The speed and acceleration are given w.r.t. the earth centered earth fixed frame. Coordinates are given in the earth centered earth fixed frame $[m/s^2]$

$\left(\dot{\vec{v}}_K^G\right)_K^{EE}$ Time derivative of the kinematic speed vector (acceleration). The speed and acceleration are given w.r.t. the earth centered earth fixed frame. Coordinates are given in the kinematic frame. $[m/s^2]$

$\left(\dot{\vec{v}}_K^G\right)_O^{EE}$ Time derivative of the kinematic speed vector (acceleration). The speed and acceleration are given w.r.t. the earth centered earth fixed frame. Coordinates are given in the north east down frame. $[m/s^2]$

$\left(\dot{\vec{v}}_K^G\right)_K^{EK}$ Time derivative of the kinematic speed vector (acceleration). The speed is given w.r.t. the earth centered earth fixed frame. The acceleration is given w.r.t. the kinematic frame. The coordinates are given in the kinematic frame. $[m/s^2]$

$\left(\dot{\vec{v}}_K^G\right)_O^{EO}$ Time derivative of the kinematic speed vector (acceleration). The speed is given w.r.t. the earth centered earth fixed frame. The acceleration is given w.r.t. the north east down frame. The coordinates are given in the north east down frame. $[m/s^2]$

$\left(\dot{\vec{v}}_K^G\right)_E^{II}$ Time derivative of kinematic speed vector (acceleration). The speed and acceleration are given w.r.t. the earth centered inertial frame. Coordinates are given in the earth centered earth fixed frame $[m/s^2]$

$\left(\dot{\vec{v}}_K^G\right)_K^{II}$ Time derivative of the kinematic speed vector (acceleration). The speed and acceleration are given w.r.t. the earth centered inertial frame. Coordinates are given in the kinematic frame. $[m/s^2]$

$\left(\dot{\vec{v}}_W^G\right)_O^{EE}$ Time derivative of the wind speed vector (acceleration). The speed and acceleration are given w.r.t. the earth centered earth fixed frame. Coordinates are given in the north east down frame. $[m/s^2]$

γ_K^G Aircraft kinematic climbing angle $[rad]$

γ_W Wind rotation angle for the rotation matrix. $[rad]$

κ_K Kármán constant. $[-]$

κ Adiabatic exponent. $[-]$

λ^G Aircraft longitude position $[rad]$

$\left(X_T^G\right)_K$ Aircraft total force x component acting in the center of gravity given in the kinematic frame. $[N]$

$\left(Y_T^G\right)_K$ Aircraft total force y component acting in the center of gravity given in the kinematic frame. $[N]$

$\left(Z_T^G\right)_K$ Aircraft total force z component acting in the center of gravity given in the kinematic frame. $[N]$

μ_A Aerodynamic bank angle. $[rad]$

μ_K Kinematic bank angle. $[rad]$

ϕ^G Aircraft latitude position $[rad]$

ρ_s Air density at sea level. $[kg/m^3]$

ρ Air density at aircraft altitude. $[kg/m^3]$

$\vec{\omega}^{IE}$ Rotation of the earth centered earth fixed frame w.r.t. the earth centered inertial frame $[rad/s]$

$\vec{f_x}$ Aircraft model state derivative function

$\vec{f_y}$ Aircraft model output function

$\vec{u}$ Aircraft control vector

$\vec{v}_{BADA}$ Aircraft parameter vector

$\vec{x}$ Aircraft state vector

$\vec{y}$ Aircraft output vector

a_0 Speed of sound at sea level. $[m/s]$

a Speed of sound. $[m/s]$

a World Geodetic System 1984 semi-major axis $[m]$

b World Geodetic System 1984 semi-minor axis $[m]$

e World Geodetic System 1984 eccentricity $[-]$

f_{flow} Fuel flow $[kg/s]$

f Coriolis parameter. $[1/s]$

f World Geodetic System 1984 flattening $[-]$

g Gravitational acceleration $[m/s^2]$

h^G Aircraft altitude above World Geodetic System 1984 ellipsoid $[m]$

h_E Ekman layer height. $[m]$

h_P Height of the Prandtl layer. $[m]$

h_{GL} Altitude above ground. $[m]$

l_N World Geodetic System 1984 distance of position from the earth's rotational axis $[m]$

m Aircraft mass $[kg]$

n_z Aircraft vertical load factor. $[-]$

$n_{z,Margin}$ Aircraft load factor margin. $[-]$

n Polytropic exponent. $[-]$

p_s Air pressure at sea level. $[Pa]$

p Air pressure at aircraft altitude. $[Pa]$

r_E Earth radius. $[m]$

t_{rx} Intermediate value of translation equations of motion x value

t_{ry} Intermediate value of translation equations of motion y value

t_{rz} Intermediate value of translation equations of motion z value

t Time. $[s]$

u_A^G x component of aerodynamic speed. $[m/s]$

u_K^G x component of kinematic speed. $[m/s]$

u_W^G x component of wind speed. $[m/s]$

u^* Shear velocity. $[m/s]$

v_A^G y component of aerodynamic speed. $[m/s]$

v_K^G y component of kinematic speed. $[m/s]$

v_W^G y component of wind speed. $[m/s]$

w_A^G z component of aerodynamic speed. $[m/s]$

w_K^G z component of kinematic speed. $[m/s]$

w_W^G z component of wind speed. $[m/s]$

x^G x position in north east down frame. $[m]$

x_A x-Axis of the aerodynamic reference frame

x_E x-Axis of the earth centered earth fixed reference frame

x_I x-Axis of the earth centered inertial reference frame

x_K x-Axis of the kinematic reference frame

x_O x-Axis of the north east down reference frame

y^G y position in north east down frame. $[m]$

y_A y-Axis of the aerodynamic reference frame

y_E y-Axis of the earth centered earth fixed reference frame

y_I y-Axis of the earth centered inertial reference frame

y_K y-Axis of the kinematic reference frame

y_O y-Axis of the north east down reference frame

z^G z position in north east down frame. $[m]$

z_A z-Axis of the aerodynamic reference frame

z_E z-Axis of the earth centered earth fixed reference frame

z_I z-Axis of the earth centered inertial reference frame

z_K z-Axis of the kinematic reference frame

z_O z-Axis of the north east down reference frame

V Velocity $[m/s]$

χ_A Aerodynamic course angle $[rad]$

χ Course angle $[rad]$

$\dot{\phi}^G$ Time derivative of aircraft latitude position. $[rad/s]$

γ_A Aerodynamic climbing angle $[rad]$

γ Climb angle $[rad]$

μ Bank angle $[rad]$

μ Bank angle $[rad]$

$\vec{x}$ Aircraft position $[m]$

n_y Lateral load factor $[-]$

n_z Vertical load factor $[-]$

Splines

B B-Spline control point or spline function

L Index indication spline for left border of race track

N B-Spline basis function

R Index indication spline for right border of race track

σ Race track width

$\vec{\eta}$ B-Spline position

$\vec{\eta}$ Spline center position

a Spline coefficient

b Spline coefficient

c Spline coefficient

d B-Spline distance

d Spline coefficient

i B-Spline control point iterate

j Spline break iteration parameter

k B-Spline order iterate

n B-Spline number of control points (minus 1)

r B-Spline knot vector

s B-Spline parameter

s Cubic spline parameter

v B-Spline speed

x B-Spline x position

y B-Spline y position

z B-Spline z position

B_i B-Spline control point

n_s Number of break point of cubic spline

x_s Spline center x position

y_s Spline center y position

Chapter 1

Introduction

In this thesis, a method that can solve large scale optimal control problems with continuous and discrete controls is further developed and applied. Special focus is on finding the optimal switching sequence for the discrete controls without prior knowledge of the structure and the elimination of high frequent switches.

An Optimal Control Problem (OCP) aims at determining the optimal state and control history for a dynamic system in order to minimize a predefined cost function. At the same time, boundary conditions and other constraints must be fulfilled. Its trajectories are compliant with the system dynamics and exploit its full potential w.r.t. the cost function. Thus, optimal control theory enables a reduction of operating cost or performance improvement of existing systems [1]. Many real world problems are formulated as OCP. For instance, the optimal control theory shows that the fuel minimal trajectory of an aircraft to an airport is a continuous descent approach. Space applications are another example as minor changes in weight have a huge impact on the overall mission. For instance, optimal control methods have been used to rotate the International Space Station (ISS) with a zero-propellant maneuver [2, 3]. Additionally, space missions using gravity assists are planned with optimal control methods [4].

Other applications are found in chemical engineering or production robots. Optimal control methods are also widely used in competitive applications. For instance, they are applied to find the time minimal trajectory for a car or plane through a given race course [5, 6].

1.1 Motivation

Optimal control methods can maximize the performance of a given system. However, the applicability of the solution is determined by the quality of the mathematical model that describes the underlying dynamic system. This includes highly non-linear dynamic models, which can implement large experimental aerodynamic data sets, as well as discrete control inputs. In the classic optimal control theory, only continuous controls entering the dynamic system are considered. Examples are the steering wheel rotation in a car or the thrust lever position in an aircraft. Additionally, many applications have discrete controls, which can only take values from a finite set. Among others, this includes the selectable gears in a car, flaps and landing gear on a civil aircraft, or valves in a process plant. A change in the discrete control introduces a discrete change in the behavior of the dynamic model.

For a realistic optimization, these discrete inputs have to be considered. A car on a race track does not achieve the minimal lap time if the gear is fixed to a single selection. Since the OCP contains integer decisions, the problems are called Mixed-Integer Optimal Control Problem (MIOCP). Control values that are not part of the discrete set must not appear in the solution. For instance, a fractional car gear selection has no physical meaning. Apart from automotive applications [7, 6, 8], MIOCP have been applied to water treatment plants [9], in biology [10, 11], for distillation processes [12, 13, 14], and for Air Traffic Management (ATM) operations [15]. In an aircraft, discrete flap settings are used to calculate optimal approach trajectories [16, 17], or to extent the flight envelope of remotely piloted vehicles [18].

Apart from realistic dynamic models, the constraints that have to be fulfilled shape the optimal solution significantly. Constraints are required as they ensure that the obtained solution complies with safety specifications, structural limits, or other regulations. Furthermore, they may reduce mechanical wear and thus increase the life-span of a product. The constraints are dependent on the system's state and controls. Therefore, they can depend on a discrete control choice as well. Similarly to the system dynamics, a switch in a discrete control results in a switch of the discrete control dependent constraint bounds. Alternatively, a constraint may become inactive for certain discrete conditions. Examples for discrete constraints can be found in the change of stall and maximum speed with different flap positions on an aircraft [19, 16, 20, 17], or engine torque and speed limits of a heavy duty truck w.r.t. the gear choice [21].

In order to minimize the cost function, the optimal switching sequence for the discrete control must be determined. Whereas in some cases the optimal switching sequence is obvious (e.g. gear selection during straight line acceleration of a car), in many cases it is not as trivial. For instance, in case the car on a race track enters a turn it has to decelerate. After the turn, the car should accelerate as fast as possible. The optimal gear selection depends on the radius of the turn. In an aircraft, the optimal extraction of the flaps can be calculated during approach. Small unmanned aerial vehicles may use flaps not only for takeoff and landing but also to perform a slow flying segment within the mission profile.

In general the optimal switching structure for the discrete controls cannot be known a priori. Therefore, its determination shall be completely subject to the optimization. At the same time, constraints that depend on the choice of the discrete control have to be considered. Finally, the number of switches must be limited. For some optimal control problems, the optimal solution for the discrete control switches an infinite number of times on an arbitrarily small time scale [22]. Such problems known as ZENO's phenomenon[1] are not desired in real world applications. High frequent switches lead to increased mechanical wear or are unrealistic for certain applications. Therefore, the number of switches must be reduced to a reasonable amount.

The aim of this thesis is to further develop the methods that enable the consideration of discrete controls in optimal control problems. They are applied to gear changes in a car as well as flap and landing gear deployment of an aircraft.

[1] ZENO OF ELEA, Greek philosopher [23]

1.2 State of the art

In this section, the state of the art regarding OCP and MIOCP is discussed. A special focus lies on direct optimal control methods. This thesis significantly contributes to the performance of the Institute of Flight System Dynamics' *FALCON.m* optimal control toolbox. Therefore, automatic derivative generation and optimal control toolboxes are discussed as well.

1.2.1 Continuous Optimal Control Methods

The OCP defines an optimization problem based on the system dynamics involved. In addition to the cost function, constraints, and boundary conditions are considered. Strategies for solving OCPs can be divided into two main categories, namely indirect and direct methods. In the former case, optimality conditions are derived. These lead to a two point boundary value problem, which can be solved either analytically or numerically [24, 25, 26]. However, this approach can become very cumbersome. Especially for highly nonlinear problems, the conditions of optimality or an analytic solution may not be found. Therefore, direct methods currently pose the best approach for realistic applications [27, 28].

In direct methods, the OCP is discretized in time. Thus, the infinite OCP is approximated by a finite parameter optimization problem. At every discretized time step, the optimal control variables need to be determined. Dependent on the discretization of the states, the method can be further differentiated. Multiple shooting methods discretize states at certain points in time called nodes [29, 30, 31] and use simulation to determine the state history in between. In collocation methods the state is fully discretized [32]. An optimization variable is introduced for every state at every time step. Thus, the full discretization enables parallel evaluation in the underlying algorithms. Due to performance reasons, collocation methods are used in this thesis. However, similar methods have also been applied to multiple shooting optimal control problems [16, 33, 21].

The resulting parameter optimization problem is usually referred to as a Non-Linear Program (NLP) and can be solved with a variety of approaches. Dynamic programming [8] and genetic algorithms [34] both belong to the non-gradient based solvers. The latter implements an iterative approach and thus requires a starting point called the initial guess. However, with increasing problem size, both algorithms suffer from the curse of dimensionality. Gradient based optimization algorithms offer an alternative. They require the first and sometimes second order derivatives of the current iteration state to calculate a candidate for the next. Common algorithms are the Sequential Quadratic Programming (SQP) [31, 35, 36] and the Interior Point (IP) method [37, 38, 39]. Well known implementations are found with Interior Point OPTimizer (IPOPT) [40], Sparse Nonlinear OPTimizer (SNOPT) [41], and We Optimize Really Huge Problems (WORHP) [42].

1.2.2 Discrete Controls / Mixed Integer Optimal Control

In this thesis, discrete controls are considered in OCPs. Within an OCP there may exist other state dependent discrete events [6]. These describe changes in the dynamic

behavior dependent on its state. Examples include contact forces of walking robots or the landing gears of an aircraft. In chemical reactors, a phase change leads to a different behavior of the chemical process [43, 44]. Within this thesis, such events are not considered. The primary focus is on systems where the discrete control influence enters from the outside.

Discrete Controls

The first MIOCP was solved by Bock [1]. It involved the energy minimal control of the New York subway, which has different discrete operation modes. As with the continuous optimal control theory, indirect [1, 45] and direct optimal control strategies exist. The latter is commonly used to solve these type of problems. Therefore, the MIOCP is converted to a Mixed-Integer Non-Linear Program (MINLP) by applying a discretization scheme. Due to the fact that the discrete value enters the dynamics as a control, it is fully discretized in time. Thus, the resulting optimization problem contains a large number of discrete variables. Due to their combinatory nature, this type of problem is extremely hard to solve [46].

Different solution strategies can be applied to MINLP. As before non-gradient based algorithms such as genetic algorithms [34], or dynamic programming [8] may be used. Additionally, branch and bound methods are also an option [7, 47, 48]. The main benefit of these algorithms is that they can handle discrete variables well. However, all of these algorithms suffer from the curse of dimensionality and are thus less suitable for large scale applications.

Alternatively, gradient based algorithms can be used to solve the MINLP. Due to the fact that derivatives w.r.t. discrete variables cannot be obtained, the discrete controls need to be reformulated. In [49, 16] the switching structure is fixed and the switching points are optimized. Alternatively, a time dependent step function can be defined [16, 17]. In both approaches, the switching sequence is not subject to optimization.

In order to find the optimal switching sequence, [11, 50] relax the discrete control with a continuous input and use constraints and penalties to ensure discrete control feasibility. However, this approach has many numerical issues [23]. In the Variable Time Transformation (VTT) [51, 52] and the control parameter enhancing technique [53, 54], small time segments are defined, each having a discrete choice associated. The time segments are scaled using continuous variables. By reducing time segments to zero, the optimal switching structure can be obtained.

Similar to the VTT, [6, 55, 18, 33, 56, 57] use an Outer Convexification (OC) approach. All possible discrete choices are evaluated and weighted. The weighting factors are optimizable and thus the optimal switching structure can be obtained through optimization.

Discrete Constraints

As mentioned above constraints can be dependent on the discrete control selection. Either the bounds change or the constraint has to be disabled entirely. As with the discrete controls, this logical switch must be formulated appropriately for gradient based optimization algorithms. In the end, all approaches mimic a logical constraint activation using a continuously differentiable formulation.

Only few approaches exist that can be used for discrete constraints. In [16] a logical switch was approximated using the hyperbolic tangent function, but due to large gradient values numerical issues may arise. The most common method used to consider this type of constraint is by implementing a vanishing constraint. They were first introduced in truss structural optimization [58] and later further developed and adapted to optimal control [59, 60, 61, 62, 63, 64, 65]. Due to the fact that the default formulation violates the constraint qualification [60, 66] either a relaxation approach can be used [18, 21], or the constraint is reformulated entirely [59]. Applications of these vanishing constraints can be found for instance in automotive applications [21], and aerospace research [16, 18, 33]. These type of MIOCP is called Mathematical Programm with Vanishing Constraint (MPVC).

Alternatively, the vanishing constraints can be reformulated as an equality constraint [67]. However, the resulting Mathematical Programm with Equilibrium Constraints (MPEC) is even harder to solve than the MPVC [59] and thus less applicable.

Switch Limitation And Discrete Control Feasibility

Although the optimal switching sequence shall be subject to optimization, the number of switches must be reduced to a reasonable amount. Additionally, due to the continuous reformulation of the discrete controls, the optimal solution may contain non-discrete selections. Therefore, not only the number of switches but also the discrete control feasibility (also called integer feasibility) must be enforced.

A limitation of switches and / or integer feasibility can be achieved through either constraints or a penalty cost approach. Constraints have been applied by Fisch [17], who disallows specific sequences of discrete controls using an equality constraint. However, as stated above, the resulting MPEC is difficult to solve and the approach does not ensure integer feasibility. Alternatively, a constraint formulation that is only feasible for the discrete selection may be used [68, 69]. Apart from the disjoint feasible set, which is introduced in the optimization problem, this approach only ensures integer feasibility but does not necessarily reduce the number of switches.

Instead of using constraints, penalty cost approaches can be used. The constraint formulations above can be introduced as penalties in the optimal control problem [11, 16, 15]. However, these cost approaches normally require some sort of homotopy approach. A penalty scaling factor is driven to a large value until a certain tolerance w.r.t. the discrete control feasibility is achieved. This may lead to numerical issues.

In [6], a switching cost penalty approach that uses two adjacent discretization points is formulated. Switches in the discrete control are penalized. If no switch occurs the resulting cost is zero. Additionally, this approach generates integer feasibly solutions. However, this formulation requires that for every discrete choice at every discretization time step an additional optimization variable is introduced. Thus, the number of optimization variables associated with the discrete controls are doubled.

1.2.3 Automatic Differentiation Methods

In this thesis, gradient based optimization algorithms are used. Therefore, the first and sometimes second order derivatives of the overall OCP have to be passed to the

optimizer. For efficient gradient calculation, the derivatives of the user supplied functions, namely model dynamics, constraints, and cost functions, are required at some point. Manual differentiation is tiresome and error prone. Hence, automatic means are mandatory.

In this thesis, *MATLAB* is used to solve the OCPs. Therefore, approaches that work with *MATLAB* as well as with C/C++ are considered. The latter can be integrated into *MATLAB* using MATLAB EXecutable (MEX) files. Finite differences may be used (forward, backward, central), but they have high computational requirements for relatively low accuracy. Complex step finite differences achieve machine accuracy [70], but may not be supported in all *MATLAB* functions .

Automatic Differentiation using Expression Tables (ADEPT) [71] and Automatic Differentiation by OverLoading in C++ (ADOL-C) [72] are both C++ libraries that use operator overloading to generate derivatives. Instead of numeric values, class instances are used to evaluate the code. Thus, the derivatives can be traced. ADOL-C supports the generation of the second order derivatives (Hessian).

In *MATLAB*, a source code transformation can be carried out using the Symbolic Math Toolbox [73]. A *MATLAB* function is evaluated with symbolic variables. The symbolic representation of the function can be differentiated and written back to code. The result is an analytic implementation of the derivatives [74]. However, with increasing complexity of the original function, this approach requires long computational times.

Both, ADigator [75] and ADIMAT [76] implement the operator overloading approach in *MATLAB*. Additionally, they create *MATLAB* files containing the derivative calculation chain. Thus, they use a hybrid approach of operator overloading and source code transformation. Both ADigator and ADIMAT require *MATLAB* features that are not supported by code generation. Therefore, the execution of the resulting code cannot be sped up by e.g. compilation.

It was proposed in [77] to split large dynamic systems into smaller subsystems, which can be differentiated locally. The overall derivatives are joined by the chain rule in order to obtain the overall derivatives. In [77] the approach is used to reduce complexity for the manual implementation of analytic derivatives in Simulink.

1.2.4 Optimal Control Toolboxes

The mentioned direct optimal control methods can be automated. As a consequence, numerous software packages exist helping a user to solve OCPs. Extensive knowledge of the methods and underlying software architecture is not required. At this point, a short overview on widely used packages is given.

In *MATLAB*, which is used in this thesis to solve the OCP, the software packages DIDO [78] and GPOPS-II [79] are widely used and known. Both offer a flexible problem formulation and some have unique features. DIDO has the ability to automatically create initial guesses. GPOPS-II interfaces to the ADigator [75] package in order to provide automatic derivatives to the solver.

In course of this thesis, an own optimal control toolbox called *FALCON.m* [80] has been developed. An own software tool was preferred as a custom implementation allows any arbitrary changes to the software package. *FALCON.m* is an object oriented *MATLAB* toolbox. It implements the collocation method and is able to calculate first

and second order analytic derivatives of highly generic and modular problem formulations. Many nonlinear optimization algorithms can be interfaced to the software package (IPOPT [40], SNOPT [41], WORHP [42]) in order to solve the underlying parameter problem.

1.3 Contribution of this thesis

In this section, the main contributions of this thesis are presented. They extend the current state of the art of MIOCP, automatic analytic derivative generation, and optimal control toolboxes.

Multi Time Switching Cost Approach

A novel switching cost penalty formulation for the OC and VTT approach is presented. It is especially suitable to generate low frequent switches in the optimal solution. Three adjacent time discretization points are used implementing a simple assumption. If the previous and next discretized points have the same choice for the discrete control, the current shall have the same choice as well. In case no switch occurs, the associated cost is zero. The formulation removes high frequent switches from the optimal solutions and ensures discrete control feasibility. Additionally, no additional optimization variables need to be introduced.

Consideration of Multiple Discrete Controls

The classic theory on discrete controls in optimal control problems usually considers a single discrete control input only. Multiple discrete control inputs are considered by representing all possible discrete combinations with a single discrete control. However, in case discrete constraints and switching cost approaches are used, only the participating discrete controls shall be used. For instance, a switch in one discrete control should not penalize another discrete control. A mapping method, which allows the consideration of multiple discrete controls is introduced. Discrete constraints and switching penalties are calculated w.r.t. individual discrete control inputs.

Extension to Vanishing Constraints

Vanishing constraints were originally developed for structural truss optimization. For better numerical behavior, the formulation is relaxed. This allows for a slight violation of the physical constraints, which can be mitigated by reducing the relaxation parameter. An adaptation of the relaxation approach, which is suitable for the OC approach is given.

Optimal Control Toolbox *FALCON.m*

A *MATLAB* optimal control toolbox aiming at large scale non-linear problems is developed. The predecessor and the OCP formulation have been developed within the optimization group. The main contribution is the new development of the derivative

generation and evaluation toolchain. It allows solving high-fidelity large scale optimal control problems. The individual main contributions of the author are explained in the following.

FALCON.m Subsystem Derivative Builder: First and Second Order Automatic Analytic Derivatives for High-Fidelity Dynamic Models

The idea of [77] to create subsystem derivatives is applied to *MATLAB* implementations of dynamic models, constraints, and cost functions. A tool which automatically calculates the subsystems' first and second order derivatives using the Symbolic Math Toolbox is developed. Subsystems can be defined using *MATLAB* functions, `matlab.System` classes, or anonymous functions. A simple modeling language that enables a user friendly definition of the evaluation chain of subsystems and their interconnection is developed. The information is used to fully automate the derivative generation process and the chain rule application in order to generate the overall derivatives. Extensive consistency checks ensure feasibility.

Outputs of the software are (with intermediate steps) a differentiated *MATLAB* file, a C++ coded version of it, and a *MATLAB* MEX file for multiple time evaluations (multi-threading capable). To the best of the author's knowledge it is the first automated tool that is capable of creating codable analytic derivatives comfortably from *MATLAB*. The tool is integrated into *FALCON.m* and has been successfully applied to high-fidelity models (e.g. 6 degree of freedom aircraft model with actuator dynamics [81]). The underlying algorithms are generic and can thus be used in other applications as well.

FALCON.m Direct Sparsity Sorting: Fast and Memory Efficient Implementation of Large Scale Jacobian and Hessian of Optimal Control Problems

The derivatives of the user function have to be applied to the derivatives of the overall OCP. It is required to write the derivative data of the user supplied functions to the correct position in the problem derivative matrices. For large scale optimal control problems, dense matrices cannot be used as the memory consumption is too high.

A method is developed and integrated into *FALCON.m* that writes local dense derivatives supplied by the user functions into the correct position of the *MATLAB* sparse matrix representation. Thus, the whole problem derivative calculation is reduced to a linear mapping between two array elements. This enables a memory efficient and high evaluation speed of large problem formulations. The method has been successfully used to solve problems with more than 600,000 optimization variables and 500,000 constraints in *MATLAB* on a consumer PC.

FALCON.m Discrete Control Toolbox Extension

The discrete control methods applied in this thesis are implemented in an user friendly discrete control extension for the *FALCON.m* optimal control toolbox. It allows for simple definition of multiple discrete controls, discrete constraints using vanishing constrains, and for automated solution of the problem.

1.4 Outline of this thesis

In chapter 2, the fundamental theory of optimal control methods is introduced. Optimization methods, indirect / direct optimal control methods, and implementation aspects are discussed. In chapter 3, the consideration of discrete controls in continuous OCPs is discussed. Different reformulation approaches are introduced and the use of the OC approach is motivated. Vanishing constraints together with relaxation and reformulation approaches are stated. A novel switching cost approach is introduced. Finally, an expansion to multiple discrete controls is explained and the two stage solution approach for MIOCP is motivated.

Chapter 4 introduces the optimal control toolbox *FALCON.m*. The basic problem definition and derivative generation from the user side are explained. Additional information regarding the interface of the user functions as well as the derivative structure is given. The implementation of the derivative generation toolchain is explained, followed by the direct sparsity sorting method. The chapter concludes with the discrete control toolbox extension.

After theory and implementation, two applications are presented. In chapter 5 the minimal lap time of a car model is determined under the consideration of gear changes. Additionally, the stability of the switching cost approach w.r.t. various parameters is shown. Aircraft approach trajectories w.r.t. the flap and landing gear deployment are optimized in chapter 6. Parameter studies are carried out in order to determine the influence on the switching sequence.

Chapter 7 concludes the thesis by giving an outlook on future research. Additional information can be found in the appendix.

Chapter 2

Theory of Continuous Optimal Control Problems

The optimal control theory describes methods to find a control law or control history for a dynamic system that minimizes a cost function while at the same time fulfilling constraints. These can be boundary conditions or path constraints that act throughout the whole time interval. There exist many approaches for solving these type of problems. Detailed description can be found in e.g. [24, 25, 26, 82, 83]. In this thesis, gradient based numerical methods are used to solve OCPs which require the overall problem to be at least once continuously differentiable. The infinite OCP is transformed into a parameter optimization problem through discretization in time.

The following chapter aims at giving an overview how continuous OCPs can be solved. In the first section 2.1, optimization problems are introduced for the unconstrained and constrained case. The conditions for optimality are stated. Additionally, numerical methods are introduced, for the case that the analytic solution cannot be derived. Section 2.2 introduces the general OCP in its standard form together with transformation methods into other formulations. To solve these OCPs section 2.3 describes the calculus of variation approach to obtain the conditions of optimality. It is argued that for real world applications analytical solutions are practically impossible to find. Therefore, in section 2.4 direct methods are introduced together with different discretization methods. Finally, section 2.5 discusses important implementation aspects regarding convergence and performance of the gradient based approaches.

This chapter does not claim to discuss OCPs in all mathematical detail but shall give a brief overview helpful for the further understanding of the thesis. Non-gradient based algorithms (e.g. genetic algorithms, dynamic programming) are not discussed as they are not relevant for this thesis. For more detailed information please refer to the literature cited above.

2.1 Optimization Problems

As stated above, in direct methods the OCP is discretized to create a parameter optimization problem. In simple cases, these problems can be solved analytically. However, in general, numerical methods are used. After a brief statement of mathematical preliminaries, this section introduces the unconstrained optimization problem followed by the constrained case. Throughout this section it is assumed that the optimiza-

tion problem is sufficiently often continuously differentiable and at least one minimum exists.

2.1.1 Mathematical Preliminaries

Let $J : \mathbb{R}^n \to \mathbb{R}$ be a scalar-valued function that is sufficiently often continuously differentiable and $\vec{z} = (z_1, z_2, \ldots, z_n)^T \in \mathbb{R}^n$ a vector. The gradient and Hessian of J at the point $\vec{z}$ are defined by:

$$\nabla J(\vec{z}) = \begin{pmatrix} \frac{\partial J(\vec{z})}{\partial z_1} \\ \vdots \\ \frac{\partial J(\vec{z})}{\partial z_n} \end{pmatrix}, \qquad \nabla^2 J(\vec{z}) = \begin{pmatrix} \frac{\partial^2 J(\vec{z})}{\partial z_1 \partial z_1} & \cdots & \frac{\partial^2 J(\vec{z})}{\partial z_1 \partial z_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 J(\vec{z})}{\partial z_n \partial z_1} & \cdots & \frac{\partial^2 J(\vec{z})}{\partial z_n \partial z_n} \end{pmatrix} \tag{2.1}$$

The level curve of J is defined by

$$N_J(c) = \{\vec{z} \in \mathbb{R}^n | J(\vec{z}) = c\} \tag{2.2}$$

and states all points where the condition $J(\vec{z}) = c$ is true. The gradient is always perpendicular to the level curve (see Figure 2.1) [84].

Let $\vec{q} : \mathbb{R}^n \to \mathbb{R}^m$ be a vector-valued function. The Jacobian of $\vec{q}$ by $\vec{z}$ is given by

$$\vec{q}\,'(\vec{z}) = \begin{pmatrix} \frac{\partial \vec{q}_1(\vec{z})}{\partial z_1} & \cdots & \frac{\partial \vec{q}_1(\vec{z})}{\partial z_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \vec{q}_m(\vec{z})}{\partial z_1} & \cdots & \frac{\partial \vec{q}_m(\vec{z})}{\partial z_n} \end{pmatrix} \tag{2.3}$$

where $\nabla \vec{q}(\vec{z})^T = \vec{q}\,'(\vec{z})$ holds for the case $m = 1$.

Throughout this chapter, optimal values are denoted by $\hat{\square}$.

2.1.2 Unconstrained Optimization

Let $J : \mathbb{R}^n \to \mathbb{R}$ be a continuously differentiable function. The unconstrained optimization problem is given by

$$\min J(\vec{z}), \qquad \vec{z} \in \mathbb{R}^n \tag{2.4}$$

where it is assumed that J is twice continuously differentiable.

Necessary and Sufficient Conditions

At every local minimum $\hat{\vec{z}}$ of J the necessary condition

$$\nabla J(\hat{\vec{z}}) = 0 \tag{2.5}$$

must bed fulfilled [83]. All points $\hat{\vec{z}}$ fulfilling (2.5) are called stationary points. The first order necessary condition does not determine the type of point (minimum, maximum, or saddle point). If the second order sufficient condition holds at a stationary point, a local minimum was found. The condition requires the function to have a positive definite Hessian $\nabla^2 J(\hat{\vec{z}})$ in the stationary point (i.e. eigenvalues are larger than zero).

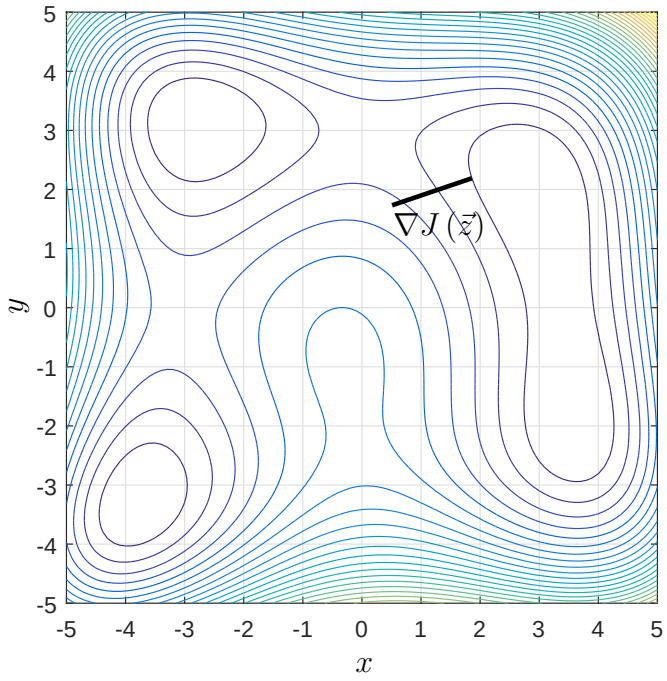


Figure 2.1: Level curve of Himmelblau's function $f = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$ with perpendicular gradient (black).

In case the Hessian is positive semi-definite, $\hat{\vec{z}}$ may be a saddle point, a minimum, or a maximum [83].

Using the necessary and sufficient conditions stationary points and thus minima are found analytically. However, for many functions, an analytic solution may not be possible. In the following, numerical unconstrained optimization are discussed.

2.1.3 Numerical Unconstrained Optimization

Numeric optimization algorithms find stationary points / minima iteratively. Therefore, they require a starting point called the initial guess. The overall idea of the algorithm is as follows [85]:

1. Set a starting point (initial guess) $\vec{z}_i$ and set $i = 0$.
2. If the stopping criteria is fulfilled: STOP!
3. Calculate a $\vec{z}_{i+1}$ for which the condition $J(\vec{z}_{i+1}) < J(\vec{z}_i)$ holds.
4. Set $i := i + 1$ and continue at (2).

The idea of the algorithm is valid not only for the unconstrained but also for the constraint case. Within the algorithm, step (3) is obviously very critical for the performance. The calculation of the next iteration point can be achieved in various ways. In this chapter only gradient based algorithms are discussed which use the first and second order derivatives of the function J to calculate a descent direction $\vec{d} \in \mathbb{R}^n$. How far

this descent direction is followed is determined by the step size $\alpha \in]0, 1]$. One drawback of this approach is that only local minima can be found. To which minimum the algorithm converges is subject to the initial guess provided (see Figure 2.2).

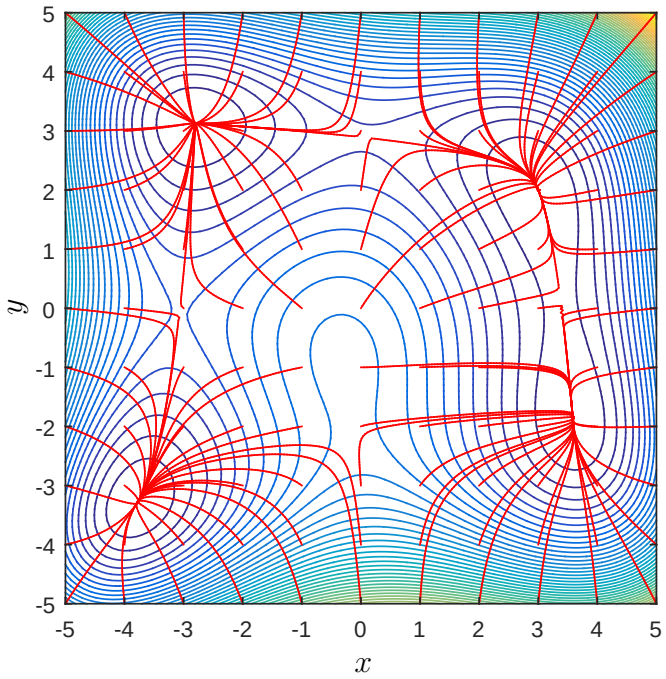


Figure 2.2: Convergence area of Himmelblau's function. For different initial guesses the algorithm converges to different minima.

Other methods such as genetic algorithms use evolutionary strategies to identify potential search directions. Thus, they have the ability to find the global minimum (which is never guaranteed) and can cope with non-differentiable problems. However, ignoring the gradient of the problem requires evaluating many points during optimization [86]. Therefore, the curse of dimensionality becomes prominent immediately. This makes these methods less suitable for problems that may contain tenths of thousands of optimization variables.

Descent Direction

As stated above, the descent direction in gradient based algorithms is determined using the first and second order derivatives. Let $J : \mathbb{R}^n \to \mathbb{R}$ be a scalar function of the vector variable $\vec{z} \in \mathbb{R}^n$. The vector $\vec{d} \in \mathbb{R}^n$ is a descent direction of J at the point $\vec{z}$ if there exist an $\hat{\alpha} > 0$ with

$$J\left(\vec{z} + \alpha \cdot \vec{d}\right) < J\left(\vec{z}\right) \qquad \forall\, 0 < \alpha \leq \hat{\alpha}. \tag{2.6}$$

The possible descent directions are between $\pm 90°$ from the negative gradient (see Figure 2.3) which can be reformulated to the sufficient condition

$$\nabla J(\vec{z})^T \vec{d} < 0 \tag{2.7}$$

using the law of cosines. However, this condition fails at stationary points since the gradient is zero. In this case other strategies need to be used which are discussed in

[85]. In the following, three methods (steepest descent, Newton, Quasi-Newton) are presented to determine a descent direction.

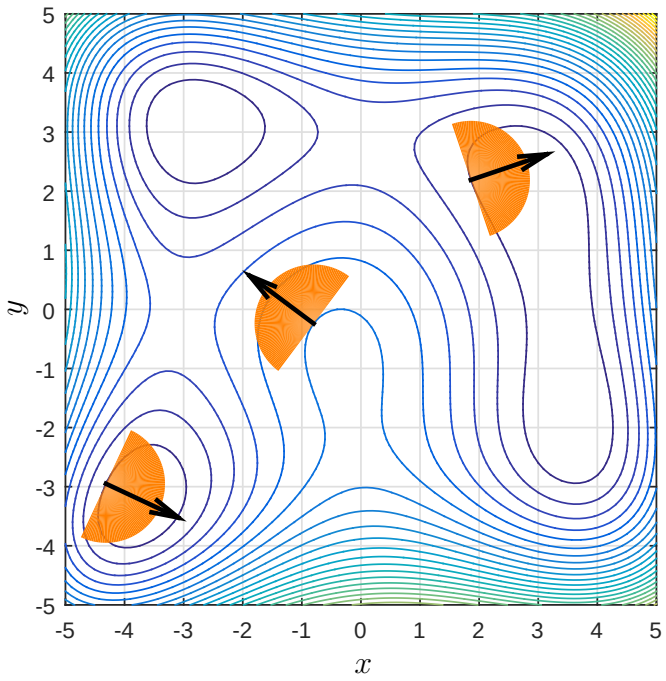


Figure 2.3: Possible descent directions for points in Himmelblau's function. Negative gradient displayed in black and valid descent directions in orange.

Steepest Descent Method

The negative gradient points in the direction of the steepest descent (see Figure 2.3). Therefore, it is a valid idea to always follow this direction

$$\vec{d_i} = -\nabla J(\vec{z_i}). \tag{2.8}$$

The steepest descent approach converges linearly [83, 85]. A better performance can be achieved using the Newton method.

Newton Method

The Newton method uses second order derivative information to determine a descent direction. It can be derived from the quadratic approximation (2$^{\text{nd}}$ order Taylor)

$$J(\vec{r}) = J(\vec{z}) + \nabla J(\vec{z})^T \cdot (\vec{r} - \vec{z}) + \frac{1}{2} \cdot (\vec{r} - \vec{z})^T \nabla^2 J(\vec{z}) \cdot (\vec{r} - \vec{z}) \tag{2.9}$$

around the current point $\vec{z}$ where $\vec{r}$ is a value near $\vec{z}$. The minimum can be found by solving the linear equation

$$\nabla J(\vec{z}) + \nabla^2 J(\vec{z}) \cdot (\vec{r} - \vec{z}) = 0 \tag{2.10}$$

which determines the point where the gradient of the approximation becomes zero. The vector from the current point $\vec{z}_i$ to the minimum of the approximation is the search direction of the Newton method:

$$\vec{d}_i = (\vec{r}_i - \vec{z}_i) = -\left(\nabla^2 J(\vec{z}_i)\right)^{-1} \nabla J(\vec{z}_i). \tag{2.11}$$

A descent direction is found if $\nabla J(\vec{z}_i) \neq 0$ and $\nabla^2 J(\vec{z}_i)$ is positive definite. If the exact Hessian is not positive definite the calculated direction may point to the nearest saddle point or maximum (see Figure 2.4). In these cases the Hessian matrix needs to be augmented [85].

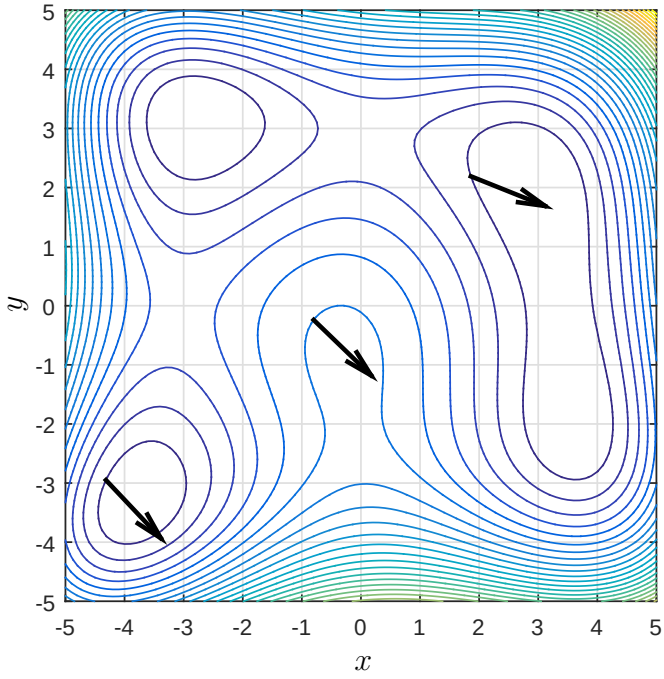


Figure 2.4: Newton descent directions for different points in Himmelblau function.

The main benefit of the Newton method is quadratic convergence near the local minimum [85]. However, the calculation of the second order derivative may be very complicated and computationally expensive. Additionally, it may not be positive definite. Therefore, in many cases the Quasi-Newton method is used instead.

Quasi-Newton Method

The idea of the Quasi-Newton method is to approximate the Hessian of the problem by a matrix Q_i

$$\vec{d}_i = -Q_i^{-1} \nabla J(\vec{z}_i) \tag{2.12}$$

in a way that it is always positive definite and the calculated search direction is better than the one obtained by steepest descent. The approximation matrix is updated after every iteration using the already calculated search direction and gradient information. The new iterate of the Hessian approximation Q_{i+1} shall be positive definite as well.

There exist many different approaches, but most commonly the well known update by Broyden-Fletcher–Goldfarb-Shanno (BFGS)

$$Q_{i+1} = Q_i + \frac{\vec{y} \cdot \vec{y}^T}{\vec{y}^T \cdot \vec{d}} - \frac{\left(Q_i \cdot \vec{d}\right) \cdot \left(Q_i \cdot \vec{d}\right)^T}{\vec{d}^T \cdot Q_i \cdot \vec{d}} \tag{2.13}$$

$$\text{with} \qquad \vec{d_i} = \vec{z}_{i+1} - \vec{z}_i \tag{2.14}$$

$$\vec{y_i} = \nabla J(\vec{z}_{i+1}) - \nabla J(\vec{z}_i) \tag{2.15}$$

is used [87, 88, 89, 90]. The new matrix iterate will be positive definite if the condition

$$\vec{d}_i^T \cdot \vec{y}_i > 0 \tag{2.16}$$

holds. Usually, the matrix is initialized with an identity of sufficient size which means that the first search direction will be the steepest descent. Other approximate Hessian methods are for instance Davidon-Fletcher-Powell (DFP) [91] and Symmetric Ranke-One (SR1) [92]. The Quasi-Newton method still converges superlinear [85].

Step Size Selection

The steepest descent method approximates the function $J(\vec{z})$ linearly whereas the (Quasi-) Newton methods use a quadratic approximation. In case of the Newton method (exact Hessian) the calculated descent direction points to the minimum of the approximation. Therefore, for quadratic functions

$$J(\vec{z}) = \vec{z}^T R \vec{z} \tag{2.17}$$

where R is a constant symmetric positive definite matrix this method converges to the minimum in a single iteration step.

For other non-linear functions this quadratic approximation may be very poor. Especially at steep gradients, a descent direction vector $\vec{d_i}$ may result in a new iteration point further off the minimum than the previous. This may lead to convergence into some distant minimum and to situations where the algorithm cannot converge or even diverges. Therefore, after $\vec{d_i}$ is calculated it has to be determined how far this direction is followed. This is called the step size selection. Introducing this step is also called globalization of the optimization algorithm [24]. It shall increase the convergence area but must not be mistaken with the search of a global minimum.

The step size α_i is a scaling factor for the descent direction

$$\vec{z}_{i+1} = \vec{z}_i + \alpha_i \cdot \vec{d_i}, \qquad \alpha_i \in]0, 1] \tag{2.18}$$

where in case of $\alpha_i = 1$ the current descent direction is fully trusted. The aim of the step size selection algorithm is to find the step size that leads to a significant descent in the objective function. If a full step $\alpha_i = 1$ along the descent direction achieves this, it is accepted. In the other case, (2.6) states that the calculated direction must lead to a descent. Thus, the step size is reduced iteratively until a reduction in the objective value is found. Since every evaluation of the objective function $J(\vec{z})$ is computationally costly, an efficient algorithm is required.

Different step size selection algorithms exist. Among them the Armijo rule is very popular. Although it is not the best method regarding efficiency [85], it is simple to understand and implement. Other methods such as the Wolfe-Powell-Rule [93] or the Goldstein-Rule [94] can be seen as an extension of this algorithm. The Armijo Rule

$$J(\vec{z}_{i+1}) = J(\vec{z}_i + \alpha\vec{d}_i) \leq J(\vec{z}_i) + \sigma \cdot \alpha_i \cdot \nabla J(\vec{z}_i)^T \cdot \vec{d}_i \tag{2.19}$$

uses a directional derivative of the current point and a constant $\sigma \in [0, 1[$ in order to determine if the new point $\vec{z}_{i+1}$ meets the required descent performance. It states that the further away the new iterative point $\vec{z}_{i+1}$ is, the better the descent in the objective value must be (see Figure 2.5). The required performance is defined by the gradient at the point $\vec{z}_i$ and by σ. For a $\sigma = 0$ (2.19) becomes

$$J(\vec{z}_i + \alpha\vec{d}_i) \leq J(\vec{z}_i) \tag{2.20}$$

which means that any descent in the objective value is accepted. In case (2.19) is not fulfilled, the step size α_i is updated by a constant factor $\beta \in [0, 1]$

$$\alpha_i := \alpha_i \cdot \beta \tag{2.21}$$

and the condition is reevaluated. Thus, the Armijo algorithm has the following steps:

1. Initialize $\alpha_i = 1$
2. If $J(\vec{z}_{i+1}) = J(\vec{z}_i + \alpha\vec{d}_i) \leq J(\vec{z}_i) + \sigma \cdot \alpha_i \cdot \nabla J(\vec{z}_i)^T \cdot \vec{d}_i$ is fulfilled: STOP!
3. Set $\alpha_i := \alpha_i \cdot \beta$ and continue at (2)

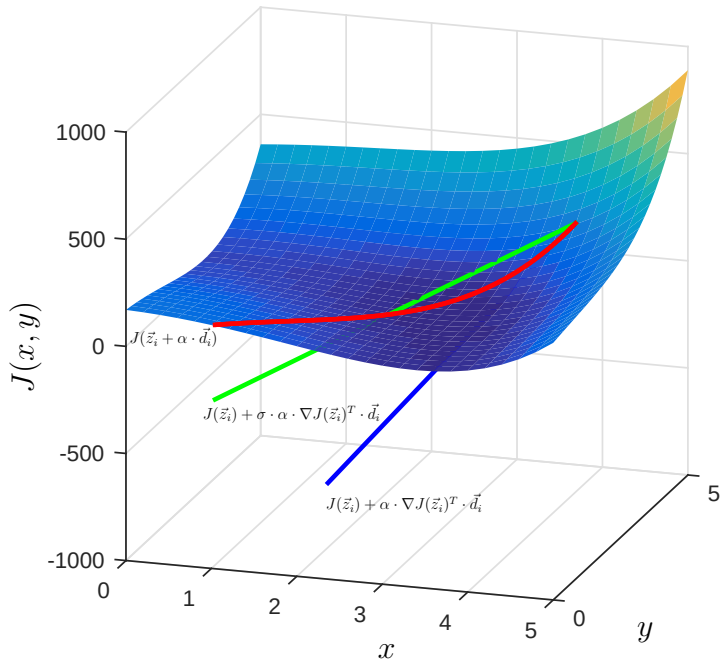


Figure 2.5: Visualization of the Armijo rule.

The two constants σ and β influence the performance significantly and have to be chosen with care. [95] and [96] discuss their selection.

Stopping Criteria

Due to numerical errors and nonlinearities, it is unlikely that a $\vec{z}_i$ is found that fulfills the necessary condition (2.5) exactly. Therefore, an optimality tolerance $\epsilon_{opt} > 0$ is defined. The necessary condition becomes

$$\|\nabla J(\vec{z}_i)\| \leq \epsilon_{opt}. \tag{2.22}$$

Additionally, a maximum number of iterations i_{max} is defined

$$i \geq i_{max} \tag{2.23}$$

that stops the optimization algorithm if for instance a solution cannot be obtained.

2.1.4 Constrained Optimization

In the constraint case, the standard optimization problem states to

$$\min J(\vec{z}), \qquad \vec{z} \in \mathbb{R}^n \tag{2.24}$$

subject to the constraints

$$g_j(\vec{z}) \leq 0, \quad j = 1, \ldots, m, \tag{2.25}$$
$$h_k(\vec{z}) = 0, \quad k = 1, \ldots, p. \tag{2.26}$$

If an inequality constraint $g_i < 0$ is fulfilled it is regarded to be inactive. Thus, the active inequality constraints are defined by the active set

$$A(\vec{z}) := \{j \mid g_j(\vec{z}) = 0, 1 \leq j \leq m\}. \tag{2.27}$$

In constrained optimization the Lagrange function

$$\mathcal{L}(\vec{z}, l_0, \vec{\lambda}, \vec{\mu}) = l_0 J(\vec{z}) + \vec{\lambda}^T \vec{g}(\vec{z}) + \vec{\mu}^T \vec{h}(\vec{z}) = l_0 J(\vec{z}) + \sum_{j=1}^{m} \lambda_j g_j(\vec{z}) + \sum_{k=1}^{p} \mu_k h_k(\vec{z}) \tag{2.28}$$

is defined, where $l_0, \vec{\lambda}, \vec{\mu}$ are multipliers. Solutions to the constraint optimization problem need to fulfill the first order necessary condition. The second order conditions can be found e.g. in [85].

First Order Necessary Condition (Fritz-John)

Let $\hat{\vec{z}} \in \mathbb{R}^n$ be a local minimum to the standard constrained optimization problem and the functions $J, \vec{g}, \vec{h}$ be continuously differentiable. Then, multipliers $l_0 \in \mathbb{R}, \vec{\lambda} \in \mathbb{R}^m, \vec{\mu} \in \mathbb{R}^p$ exist with $(l_0, \vec{\lambda}, \vec{\mu}) \neq 0$ that fulfill the following Fritz-John conditions [85]:

1. Sign Condition

$$l_0 \geq 0, \qquad \hat{\lambda}_j \geq 0, \quad j = 1, \ldots, m \tag{2.29}$$

2. Optimality Condition

$$\nabla_z \mathcal{L}(\hat{\vec{z}}, l_0, \vec{\lambda}, \vec{\mu}) = l_0 \nabla_z J(\hat{\vec{z}}) + \sum_{j=1}^{m} \hat{\lambda}_j \nabla_z g_j(\hat{\vec{z}}) + \sum_{k=1}^{p} \hat{\mu}_k \nabla_z h_k(\hat{\vec{z}}) = 0 \tag{2.30}$$

3. Feasibility

$$g_j(\vec{z}) \leq 0, \quad j = 1, \ldots, m \tag{2.31}$$
$$h_k(\vec{z}) = 0, \quad k = 1, \ldots, p \tag{2.32}$$

4. Complementary Condition

$$\hat{\lambda}_j g_j(\hat{\vec{z}}) = 0, \qquad j = 1, \ldots, m \tag{2.33}$$

The following remarks can be drawn

- In case the constraint $g_j(\hat{\vec{z}}) < 0$ is inactive the corresponding multiplier $\hat{\lambda}_j$ has to be equal to zero to fulfill the complementary condition. Alternatively, in case the constraint $g_j(\hat{\vec{z}}) = 0$ is active the multiplier $\hat{\lambda}_j$ is greater than or equal to zero to fulfill the optimality condition (2.30).

- For the special case $l_0 = 1$ the Fritz-John condition becomes the Karush-Kuhn-Tucker (KKT) condition.

- In case the Linear Independence Constraint Qualification (LICQ) is fulfilled in a local minimum (gradients of the active inequality constraints $\nabla_z g_j(\hat{\vec{z}}), j \in A(\hat{\vec{z}})$ and the equality constraints $\nabla_z \vec{h}(\hat{\vec{z}})$ are linearly independent), then the KKT conditions hold (with $l_0 = 1$). Furthermore, in case the LICQ is fulfilled, the KKT states that the multipliers $\hat{\vec{\lambda}}$ and $\hat{\vec{\mu}}$ fulfill the optimality condition in an unique way.

- Assume $l_0 = 1$ and LICQ / KKT are true. From the optimality condition the geometric representation can be derived

$$-\nabla_z J(\hat{\vec{z}}) = \sum_{j=1}^{m} \hat{\lambda}_j \nabla_z g_j(\hat{\vec{z}}) + \sum_{k=1}^{p} \hat{\mu}_k \nabla_z h_k(\hat{\vec{z}}) \tag{2.34}$$

  which states that the negative gradient of the objective function at the local minimum $\hat{\vec{z}}$ is a linear combination of the active inequality and the equality constraint gradients.

For simple cases, the first order necessary conditions can be used to find solution candidates of the optimization problem in an analytical way. However, for highly nonlinear optimization problems or if the number of variables is very large, finding an analytical solution becomes difficult or even impossible. Therefore, numerical algorithms are needed that solve these problems iteratively.

2.1.5 Numerical Constraint Optimization

For the constraint case, the optimization algorithms follow the same logic as in the unconstrained case. This means that a descent direction and a step size need to be calculated at every iteration step. In the following, two very common algorithms, the Sequential Quadratic Programming (SQP) and the Interior Point (IP) method, are presented.

Sequential Quadratic Programming

In the SQP algorithm the descent direction is obtained by solving a quadratic problem at every iteration step. The basic SQP algorithm only supports equality constraints. An expansion for inequality constraints is given at the end of this section. Assume the equality constraint optimization problem

$$\min J(\vec{z}), \qquad \vec{z} \in \mathbb{R}^n \tag{2.35}$$

$$h_k(\vec{z}) = 0, \qquad k = 1, \ldots, p \tag{2.36}$$

is given with the Lagrangian

$$\mathcal{L} = J(\vec{z}) + \sum_{k=1}^{p} \mu_k h_k(\vec{z}) \,. \tag{2.37}$$

The KKT ($l_0 = 1$) conditions

$$\nabla_z \mathcal{L}\left(\hat{\vec{z}}, \hat{\vec{\mu}}\right) = \nabla_z J\left(\hat{\vec{z}}\right) + \sum_{k=1}^{p} \hat{\mu}_k \nabla_z h_k\left(\hat{\vec{z}}\right) = 0 \tag{2.38}$$

$$\vec{h}_k\left(\hat{\vec{z}}\right) = 0, \quad k = 1, \ldots, p \tag{2.39}$$

are assumed to hold. Both conditions can be expressed by a system of nonlinear equations

$$F\left(\hat{\vec{z}}, \hat{\vec{\mu}}\right) := \begin{pmatrix} \nabla_z \mathcal{L}\left(\hat{\vec{z}}, \hat{\vec{\mu}}\right) \\ \vec{h}\left(\hat{\vec{z}}\right) \end{pmatrix} = 0 \tag{2.40}$$

that have to be fulfilled as a necessary condition. Applying the Newton method to (2.40) of the current iteration i results in a linear system of equations

$$\begin{pmatrix} \nabla^2_{zz} \mathcal{L}(\vec{z}_i, \vec{\mu}_i) & \nabla_z \vec{h}(\vec{z}_i) \\ \nabla_z \vec{h}(\vec{z}_i)^T & 0 \end{pmatrix} \begin{pmatrix} \vec{d}_i \\ \vec{\nu}_i \end{pmatrix} = - \begin{pmatrix} \nabla_z \mathcal{L}(\vec{z}_i, \vec{\mu}_i) \\ \vec{h}(\vec{z}_i) \end{pmatrix} \tag{2.41}$$

which has to be solved iteratively to obtain the next iteration

$$\vec{z}_{i+1} = \vec{z}_i + \vec{d}_i, \qquad \vec{\mu}_{i+1} = \vec{\mu}_i + \vec{\nu}_i \tag{2.42}$$

of the optimization variables and multipliers. The search directions of the optimization variable and the multipliers are given by $\vec{d}_i$ and $\vec{\nu}_i$ respectively.

In general, the exact Hessian $\nabla^2_{zz}\mathcal{L}$ is not positive definite and may be very difficult to calculate. Therefore, in most cases the Hessian is approximated using the BFGS update method (see 2.1.3).

The next iteration point $\vec{z}_{i+1}, \vec{\mu}_{i+1}$ does not necessarily have to be feasible. Additionally, the convergence area of a minimum is unknown. Therefore, as in the unconstrained case, a step size selection algorithm is used (see 2.1.3). Whether a new point is better than the previous is quantified by the merit function

$$M_q\left(\vec{z}, \eta\right) = J\left(\vec{z}\right) + \eta \left(\sum_{k=1}^{p} \left| h_k\left(\vec{z}\right) \right|^q \right)^{1/q} \tag{2.43}$$

that decreases in case the objective function or the constraint violation is reduced [97]. The norm type is selected by q and the influence of the feasibility term is set by η. The merit function presented here is just an example. Other suitable formulations can be used.

As stated above, the quadratic problem is formulated for equality constraints only. Inequality constraints are taken into account by introducing the active set A as equality constraints. During optimization the active set may change. Therefore, an update strategy is needed which is not discussed here (see [24]). A popular SQP implementation is the SNOPT solver [41, 98].

Interior Point Method

The second approach presented is the IP algorithm. In this method, the inequality constraints $\vec{g} \leq 0$ are transformed to equality constraints

$$g_j\left(\vec{z}\right) + s_j = 0, \quad j = 1, \ldots, m \tag{2.44}$$

$$h_k\left(\vec{z}\right) = 0, \quad k = 1, \ldots, p \tag{2.45}$$

using slack variables s_j. The slack variable is penalized using a barrier function

$$\min J\left(\vec{z}\right) - \eta \sum_{j=1}^{m} \log\left(s_j\right) \tag{2.46}$$

where the barrier parameter η continuously tends to zero until convergence is achieved. The problem can be solved using the SQP method described above. Since all inequality constraints are transformed into equality constraints, the IP approach does not require an active set strategy.

The barrier function penalizes constraints that are close to the boundary. Due to the reduction of the barrier parameter η, only inequality constraints that are very close to zero remain to have a strong effect on the overall objective value. The heuristics used to reduce the barrier parameter are discussed in for instance [38, 40].

Since the natural logarithm function is not defined for negative numbers, the initial guess provided must fulfill the inequality constraints. Some algorithms are capable of manipulating the initial guess to make it feasible. During optimization, the step size α is bounded by the maximum step size that is still feasible. A popular implementation of this algorithm is IPOPT [38, 40].

Optimality and Feasibility Tolerance

As in the unconstrained case, the necessary condition will not be met exactly and has to be compared to the optimality tolerance:

$$\left| \nabla_z J\left(\hat{\vec{z}}\right) + \sum_{k=1}^{p} \hat{\mu}_k \nabla_z h_k \left(\hat{\vec{z}}\right) \right| \leq \epsilon_{opt}. \tag{2.47}$$

Additionally, a feasibility tolerance

$$\left|\vec{h}_j\right| \leq \epsilon_{feas}, \qquad j = 1, \ldots, p \tag{2.48}$$

is introduced for the equality constraints. Usually, both can be chosen to the similar values (e.g. $\epsilon_{opt} = \epsilon_{feas} = 1 \cdot 10^{-6}$).

2.2 Optimal Control Problem

An optimal control problem (OCP) can be regarded as an optimization problem where $\vec{x}$ represents a state trajectory of a dynamic system $\dot{\vec{x}} = \vec{f}(\vec{x}, \vec{u})$ over time. In essence, the solution of the optimal control problem is the optimal control history that produces the optimal state trajectory that minimizes a given cost function. Therefore, the optimal control problem is stated as follows:

Find the optimal state history $\vec{x}(t)$, control history $\vec{u}(t)$, and parameters $\vec{p}$

$$\vec{x}_{lb} \leq \vec{x}(t) \leq \vec{x}_{ub}, \qquad \vec{u}_{lb} \leq \vec{u}(t) \leq \vec{u}_{ub}, \qquad \vec{p}_{lb} \leq \vec{p} \leq \vec{p}_{ub} \tag{2.49}$$

as well as initial time t_0 and final time t_f

$$t_{0,lb} \leq t_0 \leq t_{0,ub}, \qquad t_{f,lb} \leq t_f \leq t_{f,ub} \tag{2.50}$$

that minimize a cost function

$$J = M\left(\vec{x}_0, t_0, \vec{x}_f, t_f, \vec{p}\right) + \int_{t_0}^{t_f} L\left(\vec{x}(t), \vec{u}(t), t, \vec{p}\right) dt \tag{2.51}$$

subject to the system dynamics

$$\dot{\vec{x}}(t) = \vec{f}\left(\vec{x}(t), \vec{u}(t), t, \vec{p}\right), \tag{2.52}$$

constraints

$$\vec{g}_{lb} \leq \vec{g}(\vec{x}(t), \vec{u}(t), t, \vec{p}) \leq \vec{g}_{ub}, \tag{2.53}$$

and initial and final boundary conditions

$$\vec{x}_{lb,0} \leq \vec{x}(t_0) \leq \vec{x}_{ub,0}, \qquad \vec{x}_{lb,f} \leq \vec{x}(t_f) \leq \vec{x}_{ub,f}. \tag{2.54}$$

The cost function displayed in (2.51) is called the Bolza cost function and consists of two parts. Within the integral, the Lagrange cost function L is integrated over time. If the cost function is only dependent on the initial or final state of the OCP it is called a Mayer cost function M.

2.2.1 System Dynamics

The system dynamics are given by a set of ordinary differential equations

$$\dot{\vec{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \\ \dot{x}_{n_x} \end{bmatrix} = \begin{bmatrix} f_1\left(\vec{x}\left(t\right), \vec{u}\left(t\right), t, \vec{p}\right) \\ f_2\left(\vec{x}\left(t\right), \vec{u}\left(t\right), t, \vec{p}\right) \\ \vdots \\ f_{n_x}\left(\vec{x}\left(t\right), \vec{u}\left(t\right), t, \vec{p}\right) \end{bmatrix} = \vec{f}\left(\vec{x}\left(t\right), \vec{u}\left(t\right), t, \vec{p}\right) \tag{2.55}$$

in explicit first order form. States, controls and parameters

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n_x} \end{bmatrix} \in \mathbb{R}^{n_x \times 1}, \qquad \vec{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n_u} \end{bmatrix} \in \mathbb{R}^{n_u \times 1}, \qquad \vec{p} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_{n_p} \end{bmatrix} \in \mathbb{R}^{n_p \times 1} \tag{2.56}$$

are vectors entering the differential equations.

2.2.2 Different Formulations

An OCP may appear in different formulations. The OCP shown at the beginning of the section is user-friendly since all constraints are formulated as box constraints. In mathematics, the following formulation is used most commonly [99, 100]

$$\textit{Minimize} \qquad J = M\left(\vec{x}(t_0), \vec{x}(t_f)\right) + \int_{t_0}^{t_f} L\left(\vec{x}(t), \vec{u}(t), t\right) dt \tag{2.57}$$

$$\textit{s.t.} \qquad \dot{\vec{x}} = f\left(\vec{x}(t), \vec{u}(t), t\right) \tag{2.58}$$

$$\vec{\psi}\left(\vec{x}(t_0), \vec{x}(t_f)\right) = 0 \tag{2.59}$$

$$\vec{g}\left(\vec{x}(t), \vec{u}(t), t\right) \leq 0 \tag{2.60}$$

$$\vec{h}\left(\vec{x}(t), \vec{u}(t), t\right) = 0 \tag{2.61}$$

$$\vec{u}(t) \in U, \quad t \in [t_0, t_f] \tag{2.62}$$

where U is the control range and $\vec{\psi}$ the boundary state condition for the initial and final state. The constraints are divided into equality $\vec{h}$ and inequality constraints $\vec{g}$. In the following, transformations are stated to show that the different variations of the optimal control problem can be transformed back to the standard case.

Transformation of box constraints

All constraints are formulated as equality or inequality constraints. Box constraints

$$\vec{x}_{lb} \leq \vec{x} \leq \vec{x}_{ub} \quad \rightarrow \quad \vec{x} - \vec{x}_{ub} \leq 0, \qquad \vec{x}_{lb} - \vec{x} \leq 0 \tag{2.63}$$

can be converted to inequality constraints which are less or equal to zero.

Transformation of Lagrange cost to Mayer cost

Any Lagrange cost function

$$J_L = \int_{t_0}^{t_f} L\left(\vec{x}(t), \vec{u}(t), t\right) dt \tag{2.64}$$

can be converted to a Mayer cost function by introducing an additional "Lagrange" state

$$\dot{x}_L(t) = L\left(\vec{x}(t), \vec{u}(t), t\right), \quad x_L(t_0) = x_{L,0} \tag{2.65}$$

which is integrated alongside the dynamics [99]. In return, Mayer costs can be converted into Lagrange cost functions

$$L_M = \frac{d}{dt} M\left(\vec{x}_0, t_0, \vec{x}_f, t_f\right) \tag{2.66}$$

by differentiating with respect to time.

Transformation of free end time to fixed end time

If the dynamic system is not dependent on time explicitly, it is called autonomous. However, every non-autonomous dynamic system can be easily converted to an autonomous one by introducing the time

$$\dot{x}_t = 1 \tag{2.67}$$

as an additional state [99]. Furthermore, the whole integration can be transformed to normalized time

$$\tau = \frac{t - t_0}{t_f - t_0} \tag{2.68}$$

which removes the initial and final time

$$J = M\left(\vec{x}_0, t_0, \vec{x}_f, t_f\right) + \int_0^1 \left(t_f - t_0\right) \cdot L\left(\vec{x}(\tau), \vec{u}(\tau), \tau \cdot \left(t_f - t_0\right) + t_0\right) d\tau \tag{2.69}$$

from the integral. This transformation is used in approaches that solve optimal control problems numerically. Since the integration time is divided by the duration to normalize it, the Lagrangian and model dynamics need to be scaled by the duration:

$$\dot{\vec{x}}\left(\tau\right) = \left(t_f - t_0\right) \cdot \vec{f}\left(\vec{x}(\tau), \vec{u}(\tau), \tau \cdot \left(t_f - t_0\right) + t_0\right). \tag{2.70}$$

Transformation of inner point conditions

Optimal control problems usually have state conditions that happen within the trajectory at some time $t_s \in]t_0, t_f[$. To account for these inner state conditions, the OCP is split into two integrations with a new boundary condition

$$\vec{\psi_s}\left(\vec{x}(t_s), t_s\right) = 0 \tag{2.71}$$

in between [99].

2.3 Indirect Methods

Using indirect methods, it is possible to find the analytic solution to an OCP. However, for realistic and nonlinear problems an analytic solution becomes extremely hard to obtain. Therefore, these methods are not used in this thesis and thus only described briefly.

2.3.1 Calculus of Variations

The determination of the analytic solution of optimal control problems has its origins in the calculus of variations. The Brachistochrone problem, posed by BERNOULLI, can be regarded as the first variation problem ever discussed. It poses the question how a curve that carries a point mass frictionless in minimal time from one point to another with gravity would look like.

W.r.t. optimal control, the calculus of variations can be regarded as the determination of the state trajectory of a dynamic system that minimizes a certain performance measure. However, the inclusion of controls is done below.

In the calculus of variations a function $\vec{x}(t)$ shall be determined that minimizes a functional

$$J\left(\vec{x}\right) := \int_{t_0}^{t_f} L\left(t, \vec{x}(t), \dot{\vec{x}}(t)\right) dt \tag{2.72}$$

subject to initial and final boundary conditions

$$\vec{x}(t_0) = \vec{x}_0, \qquad \vec{x}(t_f) = \vec{x}_f. \tag{2.73}$$

Similar to the derivative of a function w.r.t. its argument, the variation represents a linear approximation of the functional J w.r.t. its function $\vec{x}(t)$. In the extrema the variation must vanish. This main necessary condition from the calculus of variations is the Euler-Lagrange Function (derived e.g. in [25] chapter 4)

$$\frac{\partial L}{\partial \vec{x}}(\hat{\vec{x}}(t), \dot{\hat{\vec{x}}}(t), t) - \frac{d}{dt}\left[\frac{\partial L}{\partial \dot{\vec{x}}}(\hat{\vec{x}}(t), \dot{\hat{\vec{x}}}(t), t)\right] = 0, \tag{2.74}$$

which has to be fulfilled by the optimal function $\hat{\vec{x}}(t)$ throughout the time interval $t \in [t_0, t_f]$ regardless of the boundary conditions. Using the necessary condition, the problem is converted into a two point boundary value problem. This can be solved either analytically or numerically.

2.3.2 Calculus of Variations and Optimal Control

In the following, the calculus of variations is applied to the standard optimal control problem in (2.57). For simplicity, no mathematical derivation is made and only the results are presented for optimal control problems without state constraints. A detailed explanation can be found in [26, 25].

The augmented functional results in

$$J_C = M\left(\vec{x}_0, t_0, \vec{x}_f, t_f\right) + \vec{\nu}^T \vec{\psi}\left(\vec{x}_0, t_0, \vec{x}_f, t_f\right) + \int_{t_0}^{t_f} \left[L\left(\vec{x}(t), \vec{u}(t), t\right) + \vec{\lambda}^T(t)\left[\vec{f}\left(\vec{x}(t), \vec{u}(t), t\right) - \dot{\vec{x}}\right]\right] dt \tag{2.75}$$

where the boundary conditions and the system dynamics are introduced as adjoint terms with multipliers $\vec{\lambda}(t)$ and $\vec{\nu}$. Since $\vec{\lambda}(t)$ is now a function over time, it is named co-state. Applying the variational approach gives the following necessary conditions [101]:

- state equation

$$\dot{\hat{\vec{x}}} = \vec{f}\left(\hat{\vec{x}}(t), \hat{\vec{u}}(t), t\right) \tag{2.76}$$

- co-state equation

$$\dot{\hat{\vec{\lambda}}}^T = -\frac{\partial L}{\partial \hat{\vec{x}}} - \hat{\vec{\lambda}}^T \frac{\partial \vec{f}}{\partial \hat{\vec{x}}} = -\frac{\partial H}{\hat{\vec{x}}} \tag{2.77}$$

- stationary condition

$$0 = \frac{\partial L}{\partial \hat{\vec{u}}} + \hat{\vec{\lambda}}^T \frac{\partial \vec{f}}{\partial \hat{\vec{u}}} = \frac{\partial H}{\hat{\vec{u}}} \tag{2.78}$$

- boundary conditions

$$\hat{\vec{\lambda}}(\hat{t}_f)^T = \frac{\partial M}{\partial \hat{\vec{x}}(\hat{t}_f)} + \hat{\vec{\nu}}^T \frac{\partial \vec{\psi}}{\partial \hat{\vec{x}}(\hat{t}_f)} \tag{2.79}$$

$$\hat{\vec{\lambda}}(\hat{t}_0)^T = \frac{\partial M}{\partial \hat{\vec{x}}(\hat{t}_0)} + \hat{\vec{\nu}}^T \frac{\partial \vec{\psi}}{\partial \hat{\vec{x}}(\hat{t}_0)} \tag{2.80}$$

$$0 = \frac{\partial M}{\partial \hat{t}_f} + \hat{\vec{\nu}}^T \frac{\partial \vec{\psi}}{\partial \hat{t}_f} + H(\hat{t}_f) \tag{2.81}$$

$$0 = \frac{\partial M}{\partial \hat{t}_0} + \hat{\vec{\nu}}^T \frac{\partial \vec{\psi}}{\partial \hat{t}_0} + H(\hat{t}_0) \tag{2.82}$$

where H is called the Hamiltonian

$$H = L + \vec{\lambda}^T \cdot \vec{f}. \tag{2.83}$$

The general solution strategy is the following:

1. Evaluate the necessary conditions for optimality.
2. Solve (2.78) for $\vec{u}(\vec{\lambda}, \vec{x})$ and replace it in the dynamic model and the co-state equation (2.77).
3. Solve the two point boundary value problem to obtain $\hat{\vec{x}}(t)$ and $\hat{\vec{\lambda}}(t)$.

2.3.3 Equality and Inequality Constraints

In the previous section, the optimal control problem has been solved for the unconstrained case. However, in realistic applications limitations arise that can be translated to state, control, or mixed constraints.

In general, equality constraints can be introduced as adjoint terms in the Hamiltonian. However, inequality constraints must only be taken into consideration if the constraint is active. If a bound is reached the optimality conditions change. Thus, the two point boundary value problem is transformed into multi-point boundary value problem.

The structure where inequality constraints are active is unknown. The number of active constraints and their locations need to be known a priori. Therefore, these problem types are extremely hard to handle [24]. Additionally, it is generally not possible to solve the problem unconstrained and to saturate the controls and states once the bounds are reached [25].

State-only inequality constraints are difficult to handle since it is unclear how the state constraint influences the optimal control history. To take these constraints into account [101] suggests a differentiation in time which generates state derivatives. These are replaced by the system dynamics. The process is repeated until a control dependency is found.

In this thesis, the switching structure and switching times of discrete controls and their constraints are subject to the optimization. Therefore, the fact that for indirect methods the structure needs to be known in advance stands in contradiction to the aim of this thesis.

Overall, it can be seen that for real world applications these conditions become highly nonlinear with an unknown structure in the inequality constraints. In general, such problems can only be solved using numerical algorithms. Therefore, in the course of this thesis, indirect methods are omitted in favor of direct methods.

2.4 Direct Methods

In the indirect methods, the optimality conditions for the OCP were formulated and the solution strategy for the two-point boundary value problem was introduced. If the solution cannot be derived analytically, numerical methods have to be used to solve the boundary problem. Therefore, the indirect process is usually described by ”optimize then discretize”.

Flipping the two stages of the process gives the direct methods ”discretized then optimize”. In [24] the process is defined as:

1. Convert the infinite optimal control problem into an ordinary optimization problem (Discretization)
2. Solve the resulting parameter optimization problem with a NLP solver (Optimize)
3. Assess the accuracy of the obtained solution and repeat the process if necessary

To achieve the discretization different transcription methods can be used which are discussed in this section.

2.4.1 General Aspects

The time interval $t \in [t_0, t_f]$ is divided into n_h intervals where

$$t_0 \leq t_1 \leq t_2 \leq t_i \leq \cdots \leq t_{n_h} = t_f, \qquad i = 0, \ldots, n_h \tag{2.84}$$

$$h_i = t_{i+1} - t_i \tag{2.85}$$

define the discretized time points t_i and step sizes h_i of the grid. In the optimal control theory using direct methods, the problem's solution of states $\vec{x}(t)$ and controls $\vec{u}(t)$ is given at the discretized points. Intermediate values are calculated by applying an interpolation scheme. At every time step of (2.84) a discretized control is introduced as an optimization variable. Dependent on the discretization method the number of introduced state optimization variables varies. They are either discretized fully on the time grid (2.84), or only at specific points of grid. The discretized states and controls make up the optimization variable vector

$$\vec{Z} = [t_0, t_f, \vec{x}_0, \ldots, \vec{x}_{n_h}, \vec{u}_0, \vec{u}_1, \ldots, \vec{u}_{n_h}]^T \tag{2.86}$$

where the initial and final time are introduced as additional variables in case they are optimizable. Every optimization variable has a lower and upper bound

$$\vec{Z}_{lb} \leq \vec{Z} \leq \vec{Z}_{ub}. \tag{2.87}$$

It is important to notice that $\vec{Z}$ is a vector containing all discretized variables. The optimizer used in the OCP does not have any information about their physical meaning. The interpretation of $\vec{Z}$ and the evaluation of the model dynamics, constraints, and cost functions are subject to the optimal control software interfacing with the optimizer.

Constraints

Constraints in an optimal control problem can appear in two locations. Either in the optimization vector $\vec{Z}$ described above, or in another vector called the constrained vector $\vec{F}$.

Box constraints of optimization variables are directly considered in the lower and upper bound of the $\vec{Z}$ vector (2.87). All controls on the time grid (2.84) are introduced as optimization variables. Therefore, control bounds are considered by $\vec{Z}_{lb}$ and $\vec{Z}_{ub}$. Since the initial state is always introduced as an optimization variable, the initial boundary condition is usually considered in the $\vec{Z}$ bounds as well. The same holds for other optimization parameter (e.g. final time).

All other constraints that are either dependent on the right hand side of the dynamics or combine multiple optimization variables must be considered in the constraint vector $\vec{F}$. Path constraints act along the whole time interval $t \in [t_0, t_f]$ but are only evaluated w.r.t. a single point in time t_i on the time discretization grid (2.84). The bounds of the path constraints are constant

$$\vec{g}_{lb} \leq \vec{g}_i(\vec{x}_i, \vec{u}_i, t_i, \vec{p}) \leq \vec{g}_{ub}, \qquad i = 0, \ldots, n_h. \tag{2.88}$$

Dependent on the discretization, the final state may not appear in the optimization vector. The consideration of the final boundary condition in the $\vec{Z}$ or $\vec{F}$ is dependent on the discretization scheme used. Additionally, the transcription scheme used specifies whether the feasibility w.r.t. the system dynamics is ensured through simulation or by introducing additional equality constraints called defects in the constraint vector.

Normalized Time

As discussed in section 2.2.2, the time in the optimal control problem can be normalized. This transformation is used in direct methods as well. Thus, the discretization is given in normalized time

$$0 = \tau_0 \leq \tau_1 \leq \tau_2 \leq \tau_i \leq \cdots \leq \tau_{n_h} = 1, \qquad \tau_i = \frac{t_i - t_0}{t_f - t_0}, \qquad i = 0, \ldots, n_h \tag{2.89}$$

$$h_{\tau,i} = \frac{t_{i+1} - t_i}{t_f - t_0}. \tag{2.90}$$

This formulation has two main benefits:

- The initial and final times are removed from the integration limits. As discussed above (see section 2.2.2), the model dynamics are scaled with the duration of the real time integral. Thus, derivatives w.r.t. the initial and final times can be formulated more easily.
- Due to the normalization, the normalized time grid τ_i and the normalized step size $h_{\tau,i}$ are constant and do not change in case a time parameter is optimizable. Thus, the algorithm becomes easier to implement and more stable [24].

Multiple Phases

The time interval $t \in [t_0, t_f]$ together with the boundary conditions and constraints define a so-called phase in an optimal control problem. In case of interior point conditions

$$t_s \in]t_0, t_f[\tag{2.91}$$

such as waypoints, it is suggested in section 2.2.2 to split the integration. Therefore, in direct optimal control, multiple phases are introduced and the interior point condition is transformed into an initial or final boundary condition (see Figure 2.6).

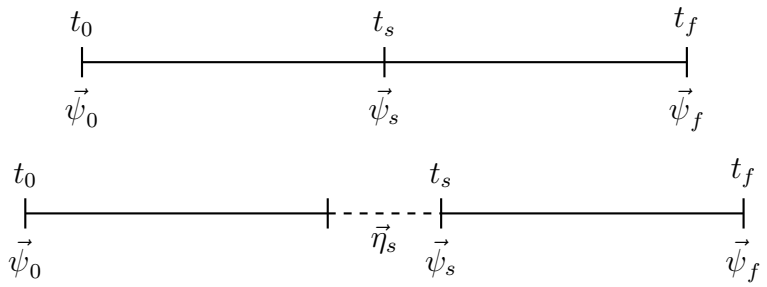


Figure 2.6: Multiple phases in direct optimal control with phase defect.

All phases in an optimal control problem are independent. Therefore, to ensure continuity in the states, phase defects

$$\vec{\eta}_s = \vec{x}_{f,j} - \vec{x}_{0,j+1} = 0, \qquad j = 1, \ldots, n_{ph} - 1 \tag{2.92}$$

have to be introduced in the $\vec{F}$ vector, where n_{ph} is the number of phases in the optimal control problem. Due to the phase defect, the interior point boundary condition must be applied to a single phase only.

Derivatives

The derivatives of the cost function and the constraint vector

$$\nabla_Z J = \frac{\partial J}{\partial \vec{Z}}, \qquad \nabla_Z \vec{F} = \frac{\partial \vec{F}}{\partial \vec{Z}} \tag{2.93}$$

need to be provided to the optimizer. The information is used inside the solver to generate a suitable descent direction (see 2.1.3). Exact knowledge of the non-zero elements as well as the analytic derivative of the overall problem derivatives (2.93) is crucial for a good performance.

Usually, the Hessian of the OCP is approximated using the BFGS rule. However, some optimization algorithms (e.g IPOPT) allow the use of the Newton method and require the exact Hessian

$$\nabla_{ZZ}\mathcal{L} = \nabla_{ZZ} J + \sum_k \lambda_k \cdot \nabla_{ZZ} g_k \tag{2.94}$$

of the problem. However, calculating the second derivative may become very complicated.

The calculation of the derivatives of the optimal control toolbox *FALCON.m* used in this thesis is explained in chapter 4. Furthermore, a highly efficient algorithm for the OCP's Jacobian and Hessian is given.

Toolbox Implementation

One of the main benefits of direct optimal control is the fact that the optimality conditions of the Hamiltonian do not need to be calculated. Evaluating the stationary condition (2.78) and thus deriving a formulation for the optimal control $\vec{u}$ is not required [24]. The OCP is represented by a parameter optimization problem and therefore its conditions for optimality are used (see section 2.1). This enables the implementation of a software kit that solves optimal control problems for any kind of model and constraints. The software developed and used in this thesis *FALCON.m* is presented in chapter 4.

2.4.2 Single Shooting

The shooting method comes from the idea that the two point boundary value problem can be expressed by an initial value problem

$$\vec{x}(t) = \vec{x}_0 + \int_{t_0}^{t_f} \vec{f}(\vec{x}(t), \vec{u}(t), t, \vec{p}) dt \tag{2.95}$$

where the final condition

$$\vec{\psi}_f = \vec{x}(t_f) - \vec{x}_f = 0 \tag{2.96}$$

is introduced as a constraint. In numerical optimal control, the integration occurs on the discretized grid

$$\vec{x}_{i+1} = \vec{x}_i + \vec{\Psi}(\vec{x}_i, \vec{u}_i, t_i, t_{i+1}, \vec{p}) \tag{2.97}$$

and the Final Boundary Condition (FBC) is introduced as a constraints of the last discretized state. The function $\vec{\Psi}$ represents an arbitrary explicit integration scheme.

The process above is called single shooting. It has its name from the idea that shooting a cannon can be regarded as a two point boundary value problem. The inclination angle and the amount of black powder act as the optimization variables. In order to hit a target at a certain distance with minimal amount of black powder, both parameters must be optimized.

The single shooting method has some advantages. The dynamics are integrated and thus automatically fulfilled. Regarding the discretization, only the first state needs to be discretized. All other states are obtained through integration (see Figure 2.7). Thus, the number of optimization variables is reduced. For an example discretization with $n_h = 9$, the optimization vector and constraint vector with path constraint are given as follows

$$\vec{Z} = \left[t_f, \vec{x}_0^T, \vec{u}_0^T, \vec{u}_1^T, \vec{u}_2^T, \vec{u}_3^T, \vec{u}_4^T, \vec{u}_5^T, \vec{u}_6^T, \vec{u}_7^T, \vec{u}_8^T, \vec{u}_9^T\right]^T \tag{2.98}$$

$$\vec{F} = \left[\vec{g}_0^T, \vec{g}_1^T, \vec{g}_2^T, \vec{g}_3^T, \vec{g}_4^T, \vec{g}_5^T, \vec{g}_6^T, \vec{g}_7^T, \vec{g}_8^T, \vec{g}_9^T, \vec{\psi}_f^T\right]^T \tag{2.99}$$

$$\vec{x} \in \mathbb{R}^3, \qquad \vec{u} \in \mathbb{R}^2, \qquad \vec{g} \in \mathbb{R}^2 \tag{2.100}$$

where it is assumed that the initial time t_0 is fixed.

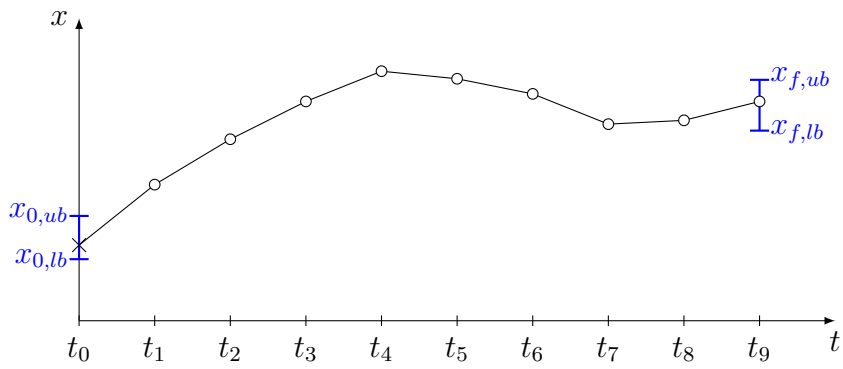


Figure 2.7: Single shooting discretization method.

However, there are also some drawbacks. First of all, the formulation is numerically less stable. As in the cannon example, a small change in the initial variables may have a huge influence on the final distance. The same behavior can be seen in OCP if nonlinear or unstable dynamics are involved. Second of all, the Jacobian of the OCP becomes a dense lower triangular matrix (see Figure 2.8). This reflects the fact that optimization variables at the beginning of the time interval have an influence on the final state. Optimization algorithms are usually optimized for sparse matrices. Therefore, large single shooting methods with are more difficult to solve.

2.4.3 Multiple Shooting

In order to stabilize the optimization and to increase the sparsity, the multiple shooting method is introduced [30]. The integration is split into multiple parts. In the optimization vector $\vec{Z}$, a set of initial states are introduced

$$\bar{X} = \{\bar{\vec{x}}_0, \bar{\vec{x}}_k, \dots\} \qquad k \subset i = 0, 1, \dots, n_h - 1 \tag{2.101}$$

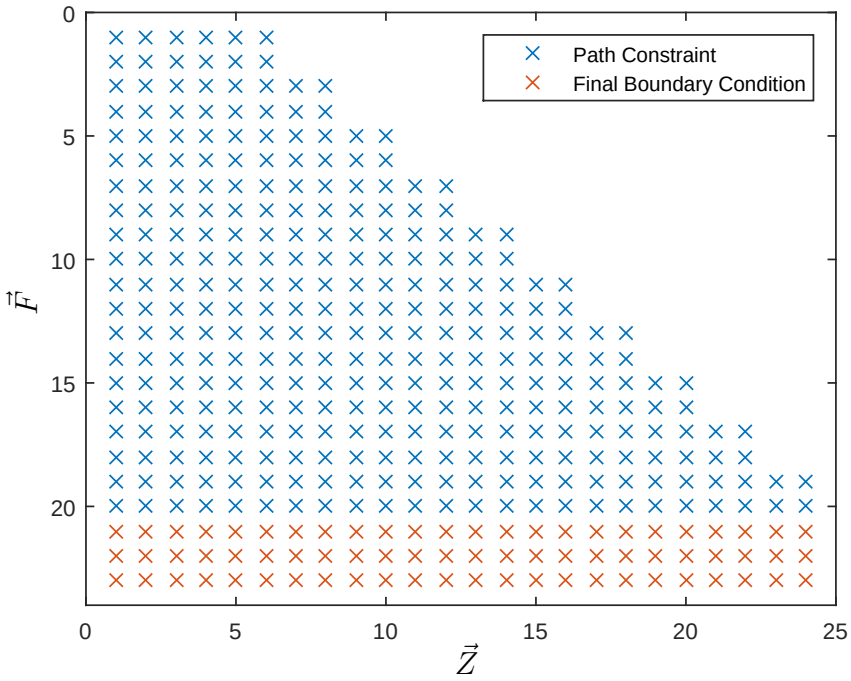


Figure 2.8: Single shooting sparsity pattern.

that must contain the first state at t_0. The initial states are called multiple shooting nodes and can be chosen at any point in time on the discretization grid. The resulting integration intervals

$$\vec{x}(t) = \int_{t_k}^{t} \vec{f}(\vec{x}(\xi), \vec{u}(\xi), \xi, \vec{p}) d\xi + \bar{\vec{x}}_k, \qquad t \in [t_k, t_{k+1}] \tag{2.102}$$

are called segments.

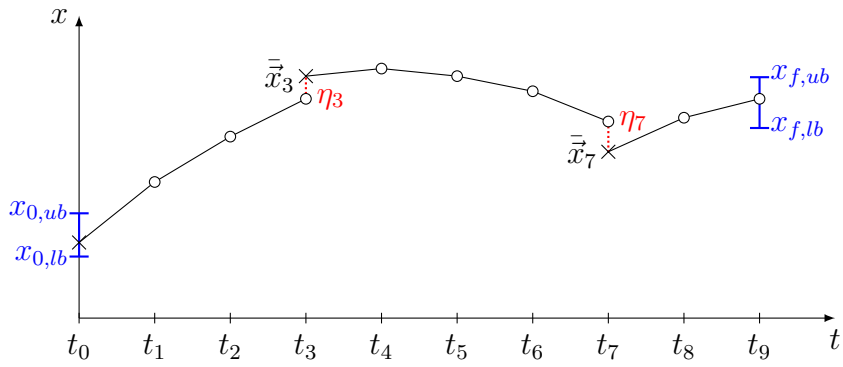


Figure 2.9: Multiple shooting discretization method.

Figure 2.9 shows that the multiple shooting segments are independent and produce discontinuities in the states history. These multiple shooting defects must vanish in the optimal solution. Therefore, they are introduced as constraints

$$\vec{\eta}_k = \vec{x}_k - \bar{\vec{x}}_k = 0 \tag{2.103}$$

in the $\vec{F}$ vector, where $\vec{x}_k$ is the final state of the previous multiple shooting segment and $\bar{\vec{x}}_k$ the initial state of the following segment. Thus, the optimization and constraint vectors have the following structure:

$$\vec{Z} = \left[t_f, \vec{x}_0^T, \vec{u}_0^T, \vec{u}_1^T, \vec{u}_2^T, \vec{x}_3^T, \vec{u}_3^T, \vec{u}_4^T, \vec{u}_5^T, \vec{u}_6^T, \vec{x}_7^T, \vec{u}_7^T, \vec{u}_8^T, \vec{u}_9^T\right]^T \tag{2.104}$$

$$\vec{F} = \left[\vec{g}_0^T, \vec{g}_1^T, \vec{g}_2^T, \vec{\eta}_3^T, \vec{g}_3^T, \vec{g}_4^T, \vec{g}_5^T, \vec{g}_5^T, \vec{\eta}_7^T, \vec{g}_7^T, \vec{g}_8^T, \vec{g}_9^T \vec{\psi}_f^T\right]^T \tag{2.105}$$

$$\vec{x} \in \mathbb{R}^3, \qquad \vec{u} \in \mathbb{R}^2, \qquad \vec{g} \in \mathbb{R}^2. \tag{2.106}$$

In the example additional multiple shooting nodes are introduced at the discretization step $k = 3$ and $k = 7$.

While the number of optimization variables increases slightly, the sparsity of the Jacobian is improved drastically (see Figure 2.10). Since the shooting segments are independent, their evaluation can be parallelized and thus improve the computational performance. The distribution of the multiple shooting is subject to the user.

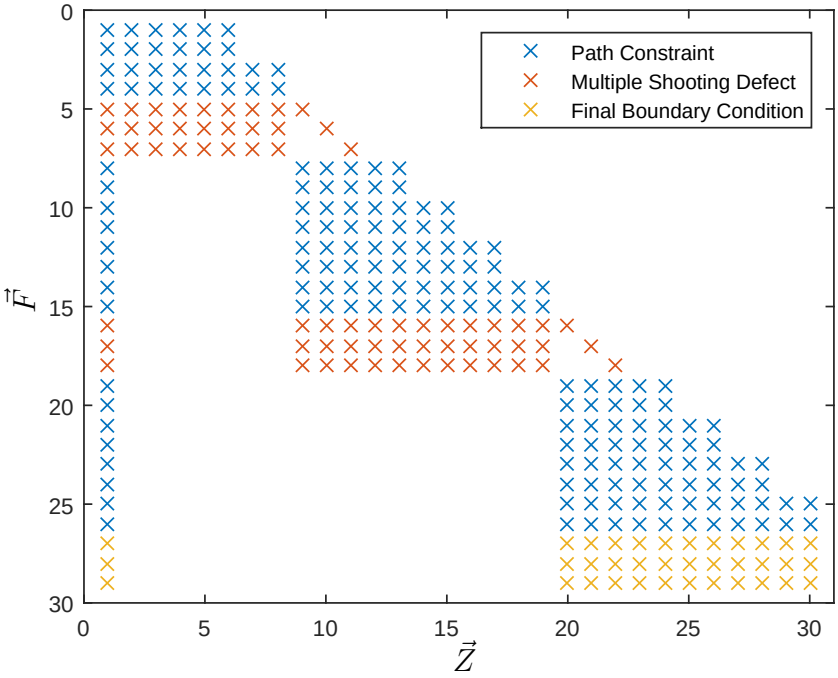


Figure 2.10: Mulitple shooting sparsity pattern.

2.4.4 Comparison of Single Shooting and Multiple Shooting

It is stated above that the multiple shooting approach is more stable regarding the convergence than single shooting. Due to the integration, initial states or controls have an influence on the trajectory at the end. This is described by the so-called sensitivity which quantifies state trajectory changes w.r.t. a variable deviation. The sensitivity equations are derived in section 2.5.2.

Figure 2.11 shows the altitude trajectory of an aircraft. The OCP was solved with single and multiple shooting. Both produced the same result (black line). Additionally,

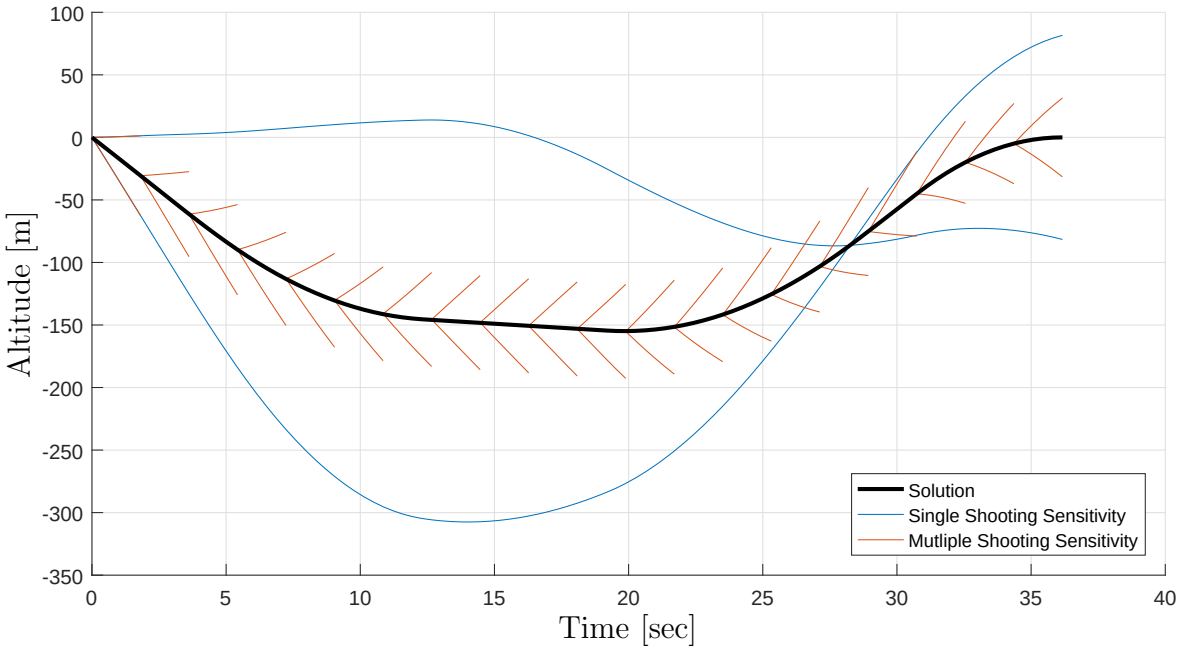


Figure 2.11: Single shooting and multiple shooting sensitivity comparison.

the influence of a change in the initial climbing angle on the altitude trajectory is shown for single shooting (blue) and multiple shooting (red). Since the state is discretized at several times in the multiple shooting approach, multiple initial climbing angles exist, each having their own sensitivity influence (fishbone structure). The changes in the trajectory are shown for a deviation in the initial climbing angle of $\pm 10°$. The following can be observed:

- In single shooting, the altitude trajectory is very sensitive to the initial value of the climbing angle. A small change by the solver during the optimization will produce significant changes in the following trajectory. In the example above, the influence even changes its sign. If the system in question is unstable, the sensitivity can diverge [24].

- In multiple shooting, the sensitivity is accumulated over shorter time intervals producing an almost linear influence. At every state discretization point the sensitivity is reset. Overall, the influence stays in a more compact band around the optimal solution.

- In shooting methods, the sensitivity results are used to calculate the analytic derivatives of the constraints in the $\vec{F}$ vector. In single shooting, the Jacobian and Hessian of the optimal control problems will contain entries with many different orders of magnitude. Therefore, the overall OCP becomes badly conditioned.

2.4.5 Integration Methods

In (2.97) the function $\vec{\Psi}$ was introduced as a placeholder for a generic increment function. The exact solution of $\vec{\Psi}$ in continuous time

$$\vec{x}_{i+1} = \vec{x}_i + \int_{t_i}^{t_{i+1}} \vec{f}(\vec{x}(t), \vec{u}(t), t, \vec{p}) dt \tag{2.107}$$

$$\approx \vec{x}_i + \vec{\Psi}(\vec{x}_i, \vec{u}_i, t_i, t_{i+1}, \vec{p}) \tag{2.108}$$

is the integral of the model dynamics between the discretized points in time. The analytic solution is approximated by a numerical integration scheme.

In this thesis, Runge Kutta [102] methods are used for numerical integration. The integration time step

$$h_i = t_{i+1} - t_i \tag{2.109}$$

is divided into substeps

$$\delta_{ij} = t_i + h_i \cdot c_j, \qquad 1 \leq j \leq m, \qquad 0 \leq c_1 \leq c_2 \leq \cdots \leq c_m \leq 1 \tag{2.110}$$

at which the dynamic model is evaluated. Thereby, m represents the number of evaluation stages for the integration step. The Runge Kutta integration scheme

$$\vec{x}_{i+1} = \vec{x}_i + h_i \cdot \dot{\vec{x}}_K = \vec{x}_i + h_i \cdot \sum_{j=1}^{m} b_j \cdot \vec{K}_j \tag{2.111}$$

$$\vec{K}_j = \vec{f}\left(\vec{x}_i + h_i \cdot \sum_{l=1}^{m} a_{jl} \cdot \vec{K}_l, \vec{u}(\delta_{ij}), \delta_{ij}, \vec{p}\right) \tag{2.112}$$

weights intermediate model evaluations $\vec{K}_j$ with the constant vector elements b_j. The result is a combined state derivative $\dot{\vec{x}}_K$ used for the propagation (2.111). Similar to $\dot{\vec{x}}_K$, the states used for the intermediate model evaluations are calculated by weighting the intermediate evaluations $\vec{K}_l$ with a_{jl}. The controls are interpolated using a suitable interpolation scheme [99].

The coefficients / weights a, b and c are usually represented in the so-called Butcher tableau

$$\begin{array}{c|ccc} c_1 & a_{11} & \dots & a_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ c_m & a_{m1} & \dots & a_{mm} \\ \hline & b_1 & \dots & b_m \end{array} \tag{2.113}$$

which can be created for different number of stages m. A scheme has the order q if the accuracy is of $\mathcal{O}(h^q)$. In general q the order increases with the number of stages m [99, 103].

A Runge Kutta method is called explicit if the matrix is strictly lower triangular:

$$a_{jl} = 0, \qquad l \geq j. \tag{2.114}$$

In this case, the calculation of the intermediate states for the model evaluation (2.112) does not require unknown future model evaluations.

If a_{jl} is not strictly lower triangular, the Runge Kutte method is called implicit. Implicit schemes have superior numerical stability especially for stiff dynamic systems [100]. However, when integrating an initial value problem, a system of non-linear equations has to be solved at every integration step. For this reason, implicit methods are usually not applied in shooting methods.

In (2.115) to (2.120) commonly used butcher tableaus are shown. (2.115) to (2.117) are explicit whereas (2.118) to (2.120) are implicit. Implicit methods can be used in collocation approach described in the following.

Euler Forward

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \tag{2.115}$$

Heun

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & 1/2 & 1/2 \end{array} \tag{2.116}$$

Classic Runge Kutta

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array} \tag{2.117}$$

Euler Backward

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \tag{2.118}$$

Trapezoidal

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1/2 & 1/2 \\ \hline & 1/2 & 1/2 \end{array} \tag{2.119}$$

Hermite-Simpson

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1/2 & 5/24 & 1/3 & -1/24 \\ 1 & 1/6 & 2/3 & 1/6 \\ \hline & 1/6 & 2/3 & 1/6 \end{array} \tag{2.120}$$

2.4.6 Collocation

The previous section showed that increasing the number of multiple shooting nodes stabilizes the OCP. Discretizing the states at every time step yields the collocation method [104, 105]. The actual integration consists of only one step, which can be reformulated as a constraint

$$\vec{\eta}_i = \vec{x}_{i+1} - \vec{x}_i - \vec{\Psi}(\vec{x}_i, \vec{x}_{i+1}, \vec{u}_i, \vec{u}_{i+1}, t_i, t_{i+1}) = 0 \tag{2.121}$$

named collocation defect (see Figure 2.12). Here, the generic integration step function $\vec{\Psi}$ includes states and controls of the current time step t_i and next time step t_{i+1}. Since the state is fully discretized, future state values are known in the evaluation. This

enables the use of implicit integration schemes in the collocation defect. This is one of the benefits of using collocation instead of shooting.

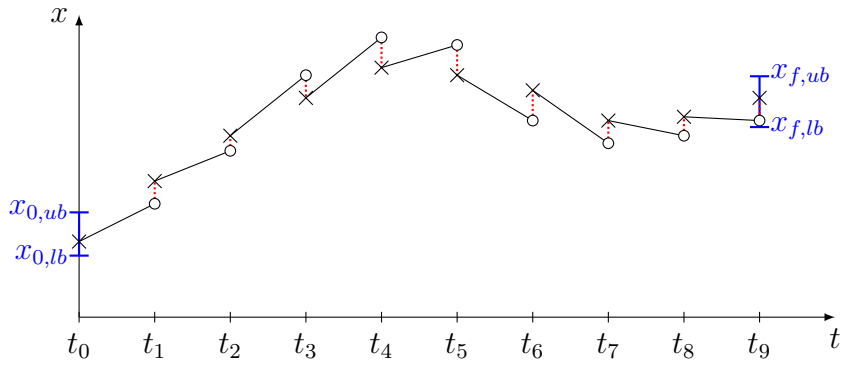


Figure 2.12: Collocation discretization method.

Another benefit is that time step evaluations of the model dynamics and consequently the evaluation of the collocation defects are completely independent. Thus, the code implementing the collocation method can be highly parallelized. Additionally, the final boundary condition can be taken into account in the $\vec{Z}$ vector.

Due to the full discretization of the state and the collocation defects on the time grid (2.84), the vectors $\vec{Z}$ and $\vec{F}$

$$\vec{Z} = \left[t_f, \vec{x}_0^T, \vec{u}_0^T, \vec{x}_1^T, \vec{u}_1^T, \vec{x}_2^T, \vec{u}_2^T, \vec{x}_3^T, \vec{u}_3^T, \ldots, \vec{x}_9^T, \vec{u}_9^T\right]^T \tag{2.122}$$

$$\vec{F} = \left[\vec{g}_0^T, \vec{\eta}_0^T, \vec{g}_1^T, \vec{\eta}_1^T, \vec{g}_2^T, \vec{\eta}_2^T, \vec{g}_3^T, \vec{\eta}_3^T, \ldots, \vec{g}_8^T, \vec{\eta}_8^T, \vec{g}_9^T\right]^T \tag{2.123}$$

$$\vec{x} \in \mathbb{R}^3, \qquad \vec{u} \in \mathbb{R}^2, \qquad \vec{g} \in \mathbb{R}^2 \tag{2.124}$$

become much larger. Thus, the size of the Jacobian increases significantly compared to the shooting case (see Figure 2.13). Although the Jacobian is increased in size, it is much sparser than in the shooting method. This has two benefits. First, most of the optimization algorithms are optimized for sparse matrices. Second, [24] states that especially with nonlinear models the complexity of the constraints is reduced. Therefore, the *FALCON.m* optimal control framework presented in chapter 4 implements this discretization method.

2.5 Function Generator

An OCP solution approach with direct methods mainly consists of 3 parts (see Fig. 2.14):

Model/Constraints/Cost User supplied information of the OCP.

Parameter Optimizer Optimization algorithm that solves the parameter optimization problem. Usually, an off-the-shelf optimizer is used.

Transcription Toolbox Discretizes the OCP and interfaces the user supplied functions with the optimization algorithm. Often, multiple optimizers are interfaced and supported. Additionally, the toolbox needs to provide the derivatives (Jacobian and sometimes Hessian) of the OCP to the solver.

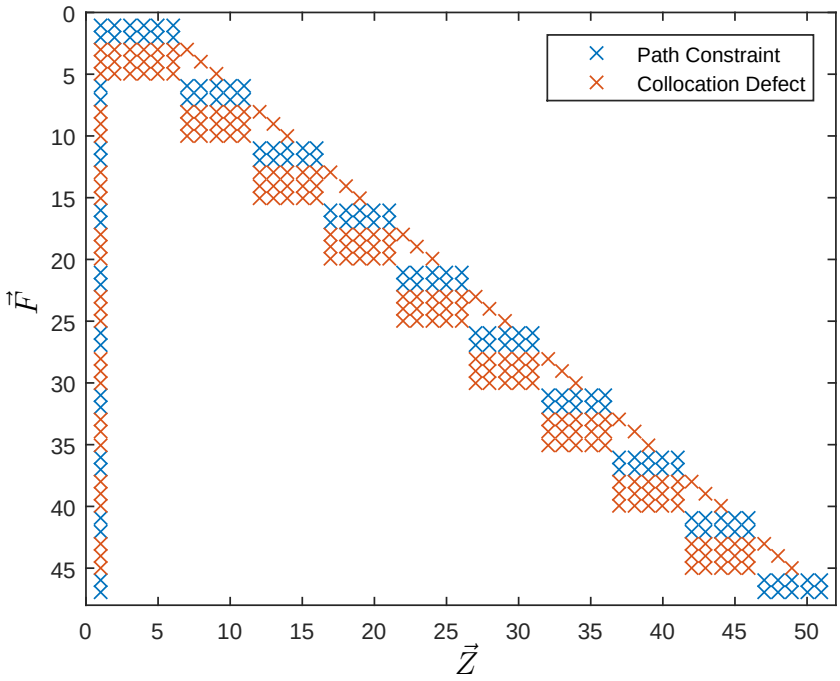


Figure 2.13: Collocation sparsity pattern.

Based on information provided by a setup script, the transcription toolbox prepares the problem for the optimization step once upon program start in a processing step. After it is prepared, the toolbox calls the NLP optimizer with the initial guess of the optimization parameter vector $\vec{Z}_{ini}$. During the optimization loop, the optimizer calls the toolbox to obtain the current cost, constraint values, and their derivatives. The toolbox function that provides this information to the NLP optimizer is called the Function Generator [24] (see Figure 2.15).

Within the function, the states, controls, and parameters contained in the current optimization vector $\vec{Z}$ are extracted and used to call the user supplied functions. The returned values are used to calculate the cost, constraints, and derivatives of the OCP. The Function Generator is very crucial for the overall performance. Especially, the calculation of the derivatives must be efficient. For large OCPs, this is not a trivial task.

Apart from an efficient calculation, [24] states that the function generator must be consistent and accurate. This means that for every $\vec{Z}$ evaluated (iteration), the same arithmetic operations must be performed. This is especially important for the derivatives. They must resemble the exact operations that were used to determine the constraints $\vec{F}$ and the cost J. For instance, if the model is evaluated with a variable step size integrator, but the gradients are calculated on a fixed step, the derivatives are not consistent to the model evaluation. This slight deviation from the exact derivatives can degrade the convergence significantly or the optimization could even fail completely [24]. In case the derivatives are calculated using e.g. finite differences, a suitable accuracy needs to be ensured.

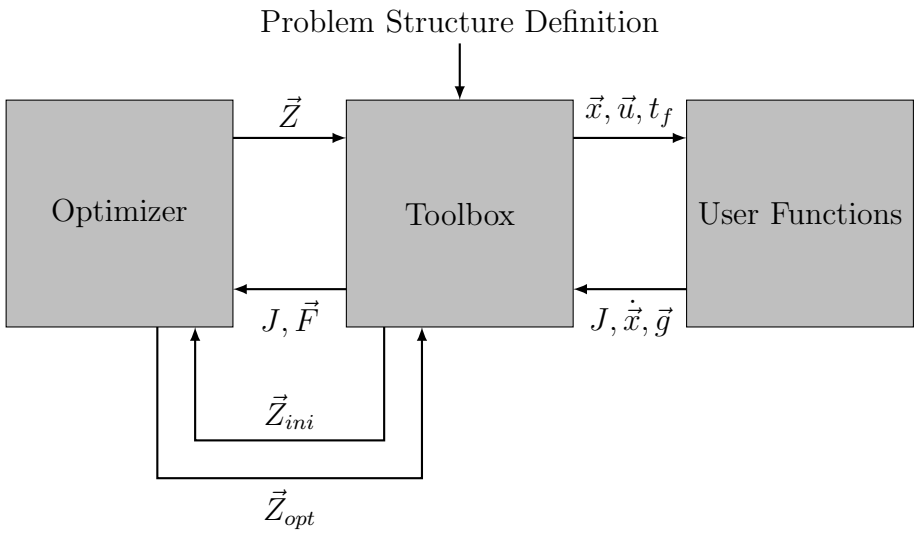


Figure 2.14: General interface of optimal control toolbox.

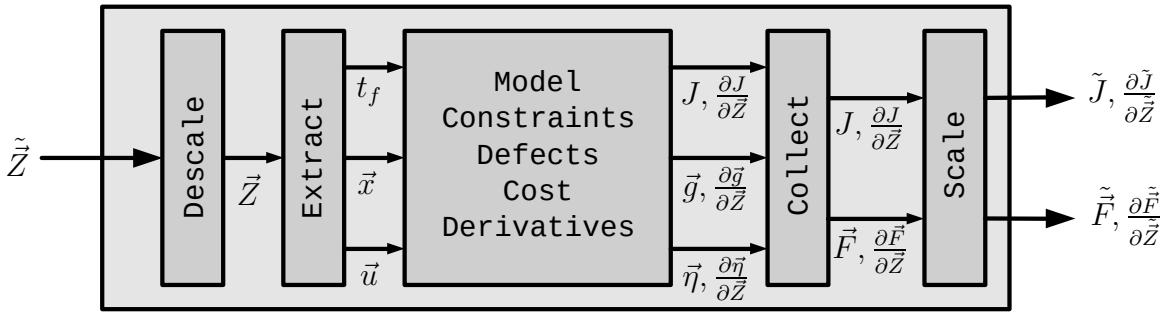


Figure 2.15: Function generator information flow scheme.

2.5.1 Scaling and Offset

States, controls, and other parameters may have different orders of magnitude. For instance, an aircraft's mass range is 50.000-70.000 kilogram (A320) but the course angle is given in radians. This introduces values with different orders of magnitudes in the OCP's Jacobian and Hessian. In addition, the feasibility and optimality tolerance become difficult to fulfill.

Assume the feasibility tolerance is $\epsilon_{feas} = 1 \cdot 10^{-6}$ and an equality constraint is imposed on the mass of the aircraft. Thus, the optimization algorithm has to fulfill the mass constraint to microgram accuracy. This unrealistic accuracy request makes the problem much harder to solve. Therefore, the vectors $\vec{Z}$, $\vec{F}$, and their derivatives need to be scaled. The scaled values contained in the vectors no longer have their physical meaning. For this reason, in the beginning of the Function Generator the input $\vec{Z}$ is de-scaled to regain the physical values (see Figure 2.15). After the evaluation of the cost function and the constraints, the outputs of the Function Generator are scaled for the optimizer.

Nearly all optimization algorithms offer an automatic scaling feature. However, the scaling will normally occur with a constant factor on the overall derivatives and does not solve the issues described above. Therefore, the scaling of the optimization

variables $\vec{Z}$, the cost function J, and the constraint values $\vec{F}$

$$\tilde{\vec{Z}} = T_Z \cdot \left(\vec{Z} - R_Z\right) \tag{2.125}$$

$$\tilde{J} = T_J \cdot (J - R_J) \tag{2.126}$$

$$\tilde{\vec{F}} = T_F \cdot \left(\vec{F} - R_F\right) \tag{2.127}$$

is done element-wise, where the tilde ($\tilde{\square}$) denotes the scaled case seen by the optimizer, T is an diagonal scaling matrix, and R is an offset vector. The scaled Jacobian is thus given by

$$\frac{\partial \tilde{J}}{\partial \tilde{\vec{Z}}} = \frac{\partial}{\partial \vec{Z}} \left[T_J \cdot (J - R_J)\right] \cdot \frac{\partial \vec{Z}}{\partial \tilde{\vec{Z}}} \tag{2.128}$$

$$= T_J \cdot \frac{\partial J}{\partial \vec{Z}} \cdot T_Z^{-1} \tag{2.129}$$

$$\frac{\partial \tilde{\vec{F}}}{\partial \tilde{\vec{Z}}} = \frac{\partial}{\partial \vec{Z}} \left[T_F \left(\vec{F} - R_F\right)\right] \cdot \frac{\partial \vec{Z}}{\partial \tilde{\vec{Z}}} \tag{2.130}$$

$$= T_F \cdot \frac{\partial \vec{F}}{\partial \vec{Z}} \cdot T_Z^{-1} \tag{2.131}$$

where the inverse of diagonal matrix T_Z can be calculated by inverting the diagonal elements. The scaled Hessian

$$\frac{\partial^2 \tilde{\mathcal{L}}}{\partial \tilde{\vec{Z}}^2} = \left[T_J \cdot \frac{\partial^2 J}{\partial \vec{Z}^2} + \sum_{i=1}^{n_F} \lambda_i \cdot T_{F,i} \cdot \frac{\partial^2 F_i}{\partial \vec{Z}^2}\right] \cdot \left(T_Z^{-1}\right)^2 \tag{2.132}$$

of the Lagrangian

$$\tilde{\mathcal{L}} = \tilde{J} + \vec{\lambda}^T \cdot \tilde{\vec{F}} \tag{2.133}$$

is calculated in a similar manner. Here, n_F is the number of constraints and $T_{F,i}$ the i$^{\text{th}}$ element on the diagonal. Compared to the problem Jacobian, the scaling of the Hessian has to be done during the summation.

The choice of the scaling diagonal matrix T and the offset vector R is relatively simple. Each entry in the $\vec{Z}$ and $\vec{F}$ vectors represents a parameter, state, control, defect, constraint, or cost. Therefore, for each of the variables introduced, a scaling and offset is defined. The aim is to achieve a unified range of every variable and constraint in the overall OCP [24]. For the defects, the scaling of the states can be used.

2.5.2 Derivative Calculation and Sparsity

The performance of the overall NLP solver is determined by the quality of the Jacobian (and sometimes Hessian) returned by the Function Generator. It has three contribution factors:

- Accuracy of the derivative values: analytic, finite differences
- Structure of the resulting matrix that exactly determines the potential non-zero elements (sparsity)

- The implementation of the algorithm

All of these influences are crucial for the overall performance. In the following, the factors are discussed in more detail.

Quality of the Derivatives

Derivatives are either analytical, representing the actual values, or an approximation derived by other numerical means (e.g. finite differences). The former are still widely used in many applications. Although there are situations in which finite differences are a good choice (e.g. black box functions) sometimes they are used for convenience. The result is an easily implemented code, but at the cost of accuracy and a significant performance reduction.

In order to calculate the Jacobian of the OCP problem with forward finite differences,

$$\frac{\partial \vec{F}}{\partial \vec{Z}} \approx \frac{\vec{F}(\vec{Z} + \Delta Z) - \vec{F}(\vec{Z})}{\Delta \vec{Z}} \tag{2.134}$$

the routine calculating J and $\vec{F}$ must be called $n_z + 1$ times for a single iteration step. The achieved accuracy is dependent on the deviation $\Delta \vec{Z}$ and is usually much lower compared to analytical means. Thus, the convergence of the OCP becomes degraded [24] and may only be achieved for larger optimality tolerances ϵ_{opt}. A more detailed evaluation of the accuracies can be found in section 4.5.1. The calculation of the OCP Hessian with finite differences does not apply for real world applications due to bad conditioning [85]. Therefore, it is better to obtain the derivatives in an analytical way.

Analytic Derivatives

For the analytic Jacobian and Hessian of the OCP, the corresponding derivatives of the cost function and constraints must be determined. The actual analytic calculation is dependent on the discretization method used. Therefore, first the derivatives for the shooting method are discussed followed by collocation. For simplicity, only the Jacobian calculation is shown.

As introduced above (2.103), the multiple shooting defect is defined as

$$0 = \vec{\eta}_k = \vec{x}_k - \bar{\vec{x}}_k \tag{2.135}$$

where $\vec{x}_k$ represents the final state of the previous segment and $\bar{\vec{x}}_k$ the initial state of the following segment. Taking the Jacobian of $\bar{\vec{x}}_k$ with respect to all optimization variables

$$\frac{\partial \bar{\vec{x}}_k}{\partial \vec{Z}} = \left[0, \ldots, 0, \frac{\partial \bar{\vec{x}}_k}{\partial \bar{\vec{x}}_k}, 0, \ldots, 0\right] = [0, \ldots, 0, I, 0, \ldots, 0] \tag{2.136}$$

gives the identity at a certain point in the Jacobian. All other entries are zero. However, the Jacobian for the final state $\vec{x}_k$ of the previous segment is unknown. The derivative is dependent on the initial state of the previous segment and on all controls of the integration.

To resolve this issue it is necessary to obtain the derivative of the states w.r.t. the optimization variables over time. As the states are integrated on the discretized time grid,

the derivatives shall be obtained in a similar manner. Thus, the sensitivity differential equation is motivated. As mentioned in 2.2.2 and 2.4.1, the derivative calculation becomes much easier to handle if the integration is normalized in time. The normalized time model is used

$$\dot{\vec{x}}(\tau) = t_f \cdot \vec{f}\left(\vec{x}(\tau), \vec{u}(\tau)\right), \tag{2.137}$$

where for simplicity the initial time t_0 is set to zero and the system is assumed to be autonomous.

The sensitivity is defined

$$S(\tau) = \frac{\partial \vec{x}(\tau)}{\partial \vec{Z}} \tag{2.138}$$

as the Jacobian of the state $\vec{x}$ w.r.t. the optimization variables $\vec{Z}$ at a given time τ. In order to obtain a differential equation, the definition is differentiated w.r.t. time

$$\dot{S}(\tau) = \frac{d}{d\tau}\left(S(\tau)\right) = \frac{\partial \dot{\vec{x}}}{\partial \vec{Z}} \tag{2.139}$$

and the time derivative is applied to the state. Inserting the system dynamics

$$\frac{\partial \dot{\vec{x}}}{\partial \vec{Z}} = \frac{\partial}{\partial \vec{Z}}\left[t_f \cdot \vec{f}\left(\vec{x}(\tau), \vec{u}(\tau)\right)\right] \tag{2.140}$$

$$= \frac{\partial t_f}{\partial \vec{Z}} \cdot \vec{f} + t_f \cdot \frac{\partial \vec{f}}{\partial \vec{x}} \cdot \frac{\partial \vec{x}}{\partial \vec{Z}} + t_f \cdot \frac{\partial \vec{f}}{\partial \vec{u}} \cdot \frac{\partial \vec{u}}{\partial \vec{Z}} \tag{2.141}$$

gives a differential equation for the sensitivity matrix. The definition of the sensitivity (2.138) reappears on the right hand side. Additional sensitivity matrices

$$S_{t_f} = \frac{\partial t_f}{\partial \vec{Z}}, \qquad S_u = \frac{\partial \vec{u}}{\partial \vec{Z}} \tag{2.142}$$

are defined. Together with the Jacobians of the model dynamics

$$J_x = \frac{\partial \vec{f}}{\partial \vec{x}}, \qquad J_u = \frac{\partial \vec{f}}{\partial \vec{u}} \tag{2.143}$$

they give the sensitivity differential equation

$$\dot{S}(\tau) = \vec{f}(\vec{x}, \vec{u}) \cdot S_{t_f} + t_f \cdot J_x \cdot S(\tau) + t_f \cdot J_u \cdot S_u(\tau), \qquad S(\tau_0) = S_0. \tag{2.144}$$

where S_0 is the initial sensitivity (analogous to the initial state). The initial state sensitivity can easily be determined since it represents the derivative of the initial state w.r.t. the optimization variables. Since the initial state is discretized, S_0 is a zero matrix with the identity at the place of the discretized initial state.

The additionally introduced sensitivities have the following meanings: $S_{t_f} \in \mathbb{R}^{1 \times n_z}$ represents the final time sensitivity and is constant. The control sensitivity $S_u \in \mathbb{R}^{n_u \times n_z}$ can be regarded as a "control input" to the sensitivity differential equation and is variable over time. The matrix is discretized on the time grid and interpolated with the same interpolation method as the controls. An example for the sensitivity matrices can be found in appendix A.1.

Using the sensitivity matrix, the Jacobian of the state w.r.t. the optimization variables is known at every discretized point in time. Usually, the sensitivity matrix is integrated alongside the system dynamics. Due to the fact that the control sensitivity is variable over time, every time step introduces additional non-zero elements in the state sensitivity. Thus, the triangle shape in the gradient sparsity is explained (see Figure 2.8). Despite the matrix operations involved, the sensitivity approach is still much faster than finite differences. Additional performance can be achieved by reducing the sensitivity matrices on the relevant $\vec{Z}$ subset.

In the collocation method, the state is fully discretized. The derivative of the collocation defects

$$\vec{\eta}_i = \vec{x}_{i+1} - \vec{x}_i - \vec{\Psi}(\vec{x}_i, \vec{x}_{i+1}, \vec{u}_i, \vec{u}_{i+1}, t_i, t_{i+1}) = 0. \tag{2.145}$$

can thus be calculated much easier. As an example the trapezoidal step

$$\vec{\eta}_{T,i} = \vec{x}_{i+1} - \vec{x}_i - t_f \cdot \frac{\tau_{i+1} - \tau_i}{2} \cdot \left(\vec{f}(\vec{x}_{i+1}, \vec{u}_{i+1}) + \vec{f}(\vec{x}_i, \vec{u}_i) \right) = 0 \tag{2.146}$$

is used. The derivatives

$$\frac{\partial \vec{\eta}_{T,i}}{\partial \vec{x}_i} = -I - t_f \cdot \frac{\tau_{i+1} - \tau_i}{2} \cdot J_{x,i}, \qquad \frac{\partial \vec{\eta}_{T,i}}{\partial \vec{u}_i} = -t_f \cdot \frac{\tau_{i+1} - \tau_i}{2} \cdot J_{u,i} \tag{2.147}$$

$$\frac{\partial \vec{\eta}_{T,i}}{\partial \vec{x}_{i+1}} = I - t_f \cdot \frac{\tau_{i+1} - \tau_i}{2} \cdot J_{x,i+1}, \qquad \frac{\partial \vec{\eta}_{T,i}}{\partial \vec{u}_{i+1}} = -t_f \cdot \frac{\tau_{i+1} - \tau_i}{2} \cdot J_{u,i+1} \tag{2.148}$$

$$\frac{\partial \vec{\eta}_{T,i}}{\partial t_f} = -\frac{\tau_{i+1} - \tau_i}{2} \cdot \left(\vec{f}(\vec{x}_{i+1}, \vec{u}_{i+1}) + \vec{f}(\vec{x}_i, \vec{u}_i) \right) \tag{2.149}$$

depend only on the two time steps involved and are constructed with identity and Jacobian matrices. Therefore, the overall structure of the problem Jacobian can be determined more easily.

Sparsity and Structure of Derivatives

The sparsity of the Jacobian and Hessian is very crucial. All modern NLP solvers expect information about the non-zero elements in the derivatives when called. This enables the algorithm to exploit the structure of the gradient and results in much faster solution of the quadratic problem (2.41). During optimization, only the non-zero elements are expected from the Function Generator. For instance, the *MATLAB* interface of IPOPT [40] requires the Jacobian as a sparse matrix and the Hessian as a lower triangular sparse. SNOPT [41] requires the non-zero elements of the Jacobian as row column pair vectors in the optimizer call. During optimization, only the vector containing the values of the non-zero elements is required. Since *MATLAB* stores sparse matrices in rcv format (row, column, value), both formulations are equivalent (see Figure 2.16).

Apart form the better performance, the use of the sparse formulation greatly reduces the memory consumption. The sparsity

$$\zeta = \frac{\text{Number of Zero Elements}}{\text{Number of Elements}} \tag{2.150}$$

of the Jacobian using the collocation discretization is usually higher than $\zeta \geq 95\%$. Thus, storing the full matrix results in a much higher memory consumption and contains mostly zero values compared to the sparse approach. Due to the fact that the

$$M = \begin{bmatrix} 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 4 & 9 \\ 3 & 2 & 0 & 0 & 7 \\ 0 & 1 & 8 & 5 & 0 \end{bmatrix} \quad \underbrace{\begin{aligned} r &= \begin{bmatrix} 1 & 2 & 4 & 4 & 5 & 2 & 3 & 5 & 3 & 5 & 3 & 4 \end{bmatrix} \\ c &= \begin{bmatrix} 1 & 1 & 1 & 2 & 2 & 3 & 3 & 3 & 4 & 4 & 5 & 5 \end{bmatrix} \\ v &= \begin{bmatrix} 5 & 1 & 3 & 2 & 1 & 2 & 1 & 8 & 4 & 5 & 9 & 7 \end{bmatrix} \end{aligned}}_{MATLAB \text{ sparse matrix format}}$$

Figure 2.16: Sparse matrix example using row r, column c and value v format.

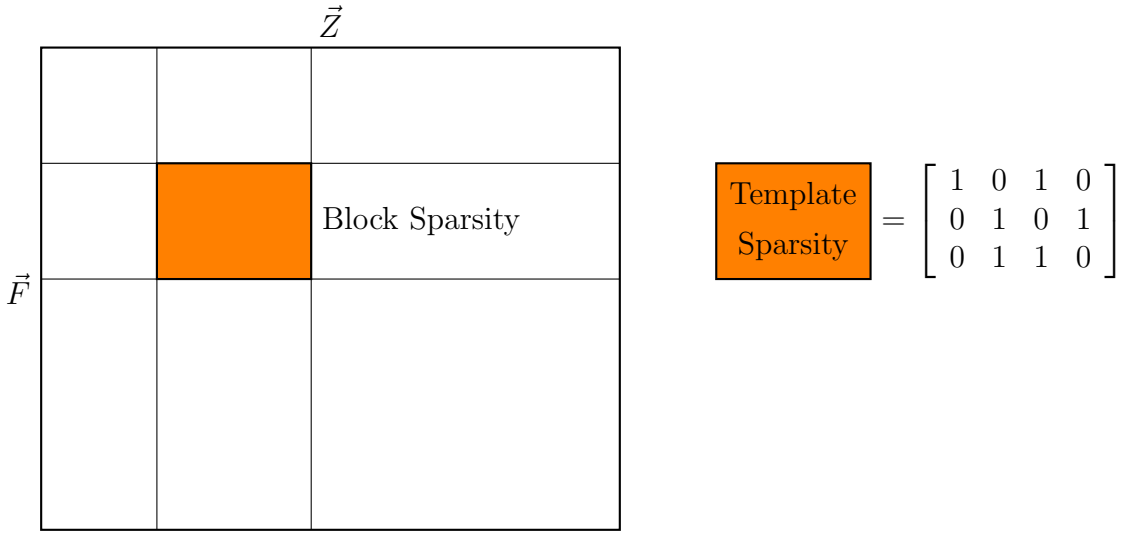


Figure 2.17: Block and template sparsity of a constraint Jacobian.

sparsity of the derivatives usually increases with finer discretization, the sparse implementation becomes especially important for large OCPs.

The sparsity calculation can be divided into two contributing factors, namely the block sparsity and the template sparsity (see Figure 2.17). From the location of a constraint in the $\vec{F}$ vector and the constraint relevant values from the $\vec{Z}$ vector the potential non-zero blocks in the Jacobian and Hessian can be determined. Thus, the block sparsity is defined. Additionally, a constraint may not be dependent on every input. Hence, a local sparsity template can be generated. By imposing this on the blocks in the OCP derivatives, the exact sparsity is calculated. Dependent on the problem, the template sparsity may lead to a significant reduction of non-zero elements [24].

Implementation of Derivatives

The time required for the evaluation of the first and second order derivatives of an OCP is crucial for the overall performance. It is influenced mainly by the dimension of the problem derivatives and by the non-linearity of the model and the constraints involved. Since realistic applications are usually extremely non-linear and produce large problem formulations efficient algorithms are mandatory. Two questions arise:

- How can the analytic first and second order derivatives of the high-fidelity models and constraints be evaluated without having to compromise on computational performance or difficult implementation on the user side?

- How can the derivatives required by the solver be calculated in the correct target format (sparse matrices)? This is especially important for large optimal control problems.

Both questions are answered in chapter 4 that introduces the optimal control framework *FALCON.m* [80]. *FALCON.m* is able to calculate first and second order analytic derivatives of the optimal control problem very fast. Additionally, the exact template sparsity of the constraint derivatives are calculated. The highly efficient function generator enables *FALCON.m* to solve large high fidelity optimal control problems on a consumer PC or laptop. In this thesis, discretized optimal control problems with 600k optimization variables and 500k constraints have been successfully solved. Additionally, *FALCON.m* is able to cope with an extremely flexible problem formulation including:

- Controls and states can be discretized independently. The same applies for multiple controls, which may have independent discretization as well. However, controls must always be a subset discretization of the states.
- Controls and other parameters can be fixed to remove them from the optimization problem. Constraints can be deactivated. The sparsity of the problem Jacobian and Hessian is adapted automatically. This allows the user to test different scenarios without having to rewrite the OCP. Additionally, optimization problems with fixed constraints can be used for parameter identification purposes [81].
- Constraints that combine arbitrary discretized points within the OCP.

Chapter 3

Theory of Mixed Integer Optimal Control Problems

In the previous chapter the continuous OCP with solution strategies and implementation aspects was introduced. However, the real world is not always continuous. In many applications discrete decisions are involved. In this case there exists only a set of options rather than a continuous choice between a minimum and maximum bound. Common examples are the gears in a car, the flaps and the landing gear on an aircraft or valves in a plant. A switch between the discrete options changes a system's behavior instantly. In this thesis, these decisions are modeled as discrete control inputs of the dynamic system. The optimal selection of the discrete control choices is subject to optimization.

Additionally, constraints in the OCP may be dependent on the discrete controls. Therefore, a change in the discrete value results in a switch on the discrete control dependent constraint bounds. These constraints must be considered in the OCP in oder to avoid infeasible discrete control selections and cannot be imposed afterwards. Furthermore, the optimal value for a discrete control may be between two available choices or the optimal solution contains frequent switches. In many applications, this behavior or non-feasible choices must be mitigated. Therefore, constraints or penalties must be introduced in the problem formulation.

In the following chapter, discrete controls are introduced in OCPs. Section 3.1 introduces the OCP with discrete controls. To solve these problems with the methods already presented in chapter 2, it is necessary to reformulate the discrete controls in the continuous domain. In section 3.2, different transformation approaches are introduced. Their applicability is discussed. Section 3.3 introduces reformulations for the discrete control dependent constraints. Their formulation depends on the discrete control transformation method. In order to avoid frequent switching, section 3.4 introduces formulations of switching cost. In section 3.5 an expansion to multiple discrete controls is made. Finally, a solution strategy for these type of problems is introduced in section 3.6.

3.1 Mixed-Integer Optimal Control Problem

The standard OCP (2.57) introduced in 2.2.2 is expanded

$$\textit{Minimize} \qquad J = M\left(\vec{x}(t_0), \vec{x}(t_f)\right) + \int_{t_0}^{t_f} L\left(\vec{x}(t), \vec{u}(t), t, \vec{v}(t)\right) dt \tag{3.1}$$

$$\textit{s.t.} \qquad \dot{\vec{x}} = \vec{f}\left(\vec{x}(t), \vec{u}(t), t, \vec{v}(t)\right) \tag{3.2}$$

$$\vec{\psi}\left(\vec{x}(t_0), \vec{x}(t_f)\right) = 0 \tag{3.3}$$

$$\vec{g}\left(\vec{x}(t), \vec{u}(t), t, \vec{v}(t)\right) \leq 0 \tag{3.4}$$

$$\vec{h}\left(\vec{x}(t), \vec{u}(t), t\right) = 0 \tag{3.5}$$

$$\vec{u}(t) \in U, \quad t \in [t_0, t_f] \tag{3.6}$$

$$\vec{v}(t) \in V = \{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_{n_v}\} \tag{3.7}$$

to include the discrete control $\vec{v}$ which can only have values from a fixed set V (see equation (3.7)). These type of optimal control problems are called Mixed-Integer Optimal Control Problem (MIOCP) as the choice of a value from the discrete set can be interpreted as an integer selection. The discrete control can appear in the cost function, in the system dynamics, and in the inequality constraints. As boundary conditions and Mayer cost functions are usually state dependent, the discrete control is not introduced there. The formulation of the discrete constraints assumes inequality constraints involved. Therefore, the discrete control does not appear in the equality constraints. They can be considered using e.g. two inequality constraints [11].

For now, only a single discrete input $\vec{v}$ with n_v discrete choices is assumed. Multiple discrete controls can always be taken into account by treating all possible combinations as a single discrete control. This will be further discussed in section 3.5 of this chapter. Discrete controls usually influence the dynamics by replacing certain parameters (e.g. aerodynamic coefficients on an aircraft, gear transmission ratio in a car). Therefore, these inputs can appear as scalars, vectors, or even matrices.

As in the continuous case, the MIOCP is discretized in time to transform it into a parameter optimization problem. Since it contains discrete parameters it is called a Mixed-Integer Non-Linear Program (MINLP). Due to the fact that the discrete choice acts as a control, the number of discrete parameters is dependent on the control discretization. Therefore, the high number of discrete optimization variables make these problems hard to solve [59].

There exist some algorithms and approaches that are able to handle MINLP directly, namely branch & bound [7], dynamic programming [8], genetic algorithms [34], or full enumeration (evaluating every possible combination) [23]. A description of the methods and a detailed discussion of these on MINLP can be found in [23]. However, due to the large number of discrete parameters involved, these approaches suffer from the curse of dimensionality. For instance, [11] states that branch & bound methods are currently less applicable to be applied to non-convex MINLP as the search effort is too expensive. Therefore, in the following, methods are presented to transform the discrete control parameters into continuous ones. The following limitations must be considered:

- The overall model interface structure must remain. The number and size of the

input variables must not change. States are continuous and must not vanish dependent on the discrete control selection.

- Logical decisions such as path or runway selection cannot be considered by this approach as they do not represent a discrete control.

3.2 Discrete Controls Continuous Reformulation

In this section, methods are presented to transform the MIOCP into a continuous OCP that can be solved with the approaches explained in chapter 2. Requirements to the reformulation are discussed. Afterwards, different approaches are presented. Some relax the problem in a way that non-discrete values may occur. This is a result of the transformation into the continuous domain. Mitigation of invalid values is discussed later in sections 3.3 and 3.4. Finally, the consideration of multiple discrete controls (3.5) and a solution strategy (3.6) are presented.

3.2.1 Evaluation Criteria

The discrete control reformulation approaches are evaluated w.r.t. the following criteria:

- The switching sequence of the discrete controls shall be determined automatically. Not only the times at which a switch occurs shall be optimizable, but also the number of switches and the discrete choice that is switched to.
- Multiple discrete controls shall be optimizable independently. Many dynamic systems have multiple discrete controls that are completely or at least partially independent.
- Usually, a switch in a discrete control is modeled as an instantaneous switch. However, in some applications, this switch from one discrete value to the next is performed continuously (e.g. linear change in the aircraft flap position). Additionally, a shift may introduce a dead time (e.g. no engine torque during a car gear shift).
- As some reformulation approaches relax the discrete choice during the optimization, non-discrete values may appear in the model dynamics. As non-discrete values may lead to unrealistic behavior, they must not appear in the final solution.
- Ideally, the discrete control can be optimized in a single optimization step. However, some approaches require a multi-stage homotopy approach. It is desired to have as little stages as possible.

3.2.2 Division into Multiple Phases

The simplest reformulation approach divides the optimal control problem into multiple phases [17, 16]. In this Multiple Phases (MP) approach a discrete choice is assigned

to each phase. Thus, the discrete control is transformed to a phase constant. This is a viable choice if the optimal switching structure is known. The resulting multi-phase problem can be solved with existing methods.

However, if the switching structure is not known, a guess is required. This becomes extremely hard if multiple discrete inputs exist in the dynamic model. The situation becomes even more difficult if the OCP contains interior point conditions (see section 2.2.2). There exist approaches where different switching structures are evaluated [106, 107]. The number of switches and their respective order is iterated until there is no significant decrease in the objective function. However, these approaches require to solve many OCPs.

In case the selected switching structure is not correct, the optimization algorithm may reduce the phase duration to zero. In this case, the dynamics of the phase become singular introducing numerical issues in the optimization algorithm [11].

3.2.3 Hyperbolic Tangent Function

The Hyperbolic Tangent (HT) approach exploits the fact that the hyperbolic tangent function resembles a step shape that changes from -1 to 1. Therefore, it can be used to model discrete changes over time

$$\tilde{\vec{v}} = \vec{v}_0 + \sum_{k=1}^{n-1} \left(\vec{v}_k - \vec{v}_{k-1}\right) \cdot \left[\tanh\left(a \cdot (t - t_k)\right) + 1\right]/2 \tag{3.8}$$

where n is an arbitrary number of discrete changes and a a steepness factor. The hyperbolic tangent function is transformed in such a way that at every time t_k the difference $\vec{v}_k - \vec{v}_{k-1}$ to the next discrete choice is added (see Figure 3.1). Additionally, the constraints

$$t_{k+1} - t_k \geq 0 \tag{3.9}$$

are introduced in the OCP. The order of the switches must not change as otherwise values that do not appear in the discrete set are created.

As with the MP approach, the switching structure of the discrete control must be known. However, multiple discrete controls can be considered independently. Additionally, this approach does not conflict with interior point conditions. Besides, eliminating a discrete choice by the optimization algorithm

$$t_{k+1} = t_k \tag{3.10}$$

does not introduce numerical issues.

However, this approach has some drawbacks. The model becomes time dependent. Although this is not unusual, it may not be desired and has to be addressed in the OCP. In case the optimal control toolbox does not support time dependencies, the time transformation described in section 2.2.2 can be used.

Since the discrete control usually influences model parameters (e.g. aerodynamic coefficients), these become time dependent. An adaptation of the model may involve significant implementation changes. Due to the fact that the hyperbolic tangent function changes between discrete values continuously, the model must be able to handle intermediate variables.

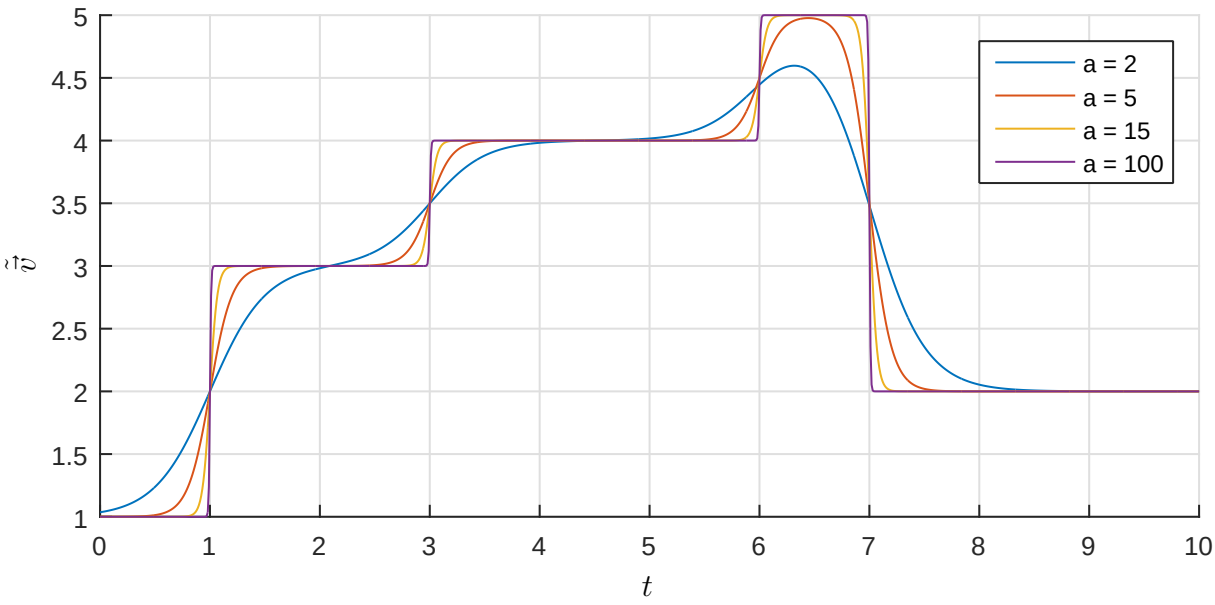


Figure 3.1: Discrete switches over time using the hyperbolic tangent function approach for different steepness factors.

In the actual optimization, the gradient of the hyperbolic tangent function is difficult to handle as well. In case an instant switch is required, the steepness factor a is chosen to a sufficiently large value (see Figure 3.1). Thus, the gradient of the switching function (3.8) w.r.t. time is very high at the switching instances t_k and almost zero in between. This leads to a badly conditioned gradient matrix. In many applications it can be found that the switching times may not be changed during the optimization at all. In order to resolve this, a homotopy approach needs to be used where a small value for a is chosen in the beginning [16]. In successive optimizations a is gradually increased. The number of steps, initial and final value of a is subject to the user.

3.2.4 Relaxation or Inner Convexification

In this approach, the discrete control is relaxed

$$\tilde{v} \in [\min(V), \max(V)], \qquad V = \{v_1, v_2, \ldots, v_{n_v}\} \tag{3.11}$$

by omitting the discrete choices in the optimization variable. The NLP can choose the variable continuously at every discretized point in time. Using constraints and penalties, discrete choices are enforced. In theory, this approach has the ability to find the optimal switching sequence. Multiple independent discrete controls can be considered. This approach is also referred to as the Inner Convexification [6].

In practice, this approach shows some issues. As before, intermediate (non-discrete) values are allowed in the optimization. The model has to be able to handle these values. Additionally, enforcing the discrete values during optimization remains an issue. In order to argue that this approach is not the ideal choice, it shall be discussed here in more detail.

A trivial method to enforce discrete values is by introducing a constraint

$$0 = (\tilde{v} - v_1) \cdot (\tilde{v} - v_2) \cdot \ldots \cdot (\tilde{v} - v_{n_v}) \tag{3.12}$$

as discussed in [68, 69]. However, it can easily be seen that such an approach produces a disjoint feasible set which makes the problem extremely hard to solve [11].

To mitigate this issue, a slack variable κ may be introduced. It transforms the equality constraint

$$-\kappa \leq (\tilde{v} - v_1) \cdot (\tilde{v} - v_2) \cdot \ldots \cdot (\tilde{v} - v_{n_v}) \leq \kappa \tag{3.13}$$

into an inequality box constraint. Then, in multiple optimization stages, the slack variable κ is slowly driven to zero [69].

Due to the difficulties with the constraints, [69] also proposes the use of a penalty approach. For a simple binary example $V = \{0, 1\}$, which is often used, the cost

$$J_P = \alpha \cdot \tilde{v} \cdot (1 - \tilde{v}) \tag{3.14}$$

can be easily formulated. Here, α represents a scaling factor that is increased in a homotopy approach. In case there are more than two discrete choices, the constraint function

$$J_P = \alpha \cdot (\tilde{v} - v_1)^2 \cdot (\tilde{v} - v_2)^2 \cdot \ldots \cdot (\tilde{v} - v_{n_v})^2 \tag{3.15}$$

is squared. In both penalty formulations, only non-discrete values contribute to the penalty cost. However, it can be seen that this approach introduces strong additional local minima in the NLP.

Although the homotopy approach may lead to a solution in both cases, the choice of κ or α is not trivial. Due to the fact that polynomials start to oscillate with the number and distribution of the discrete choices, a selection may be difficult. Overall, this approach is less suited for OCPs discussed in this thesis and is thus no longer considered.

3.2.5 Outer Convexification

While the Inner Convexification relaxes the actual discrete control input, the Outer Convexification (OC) applies a relaxation method around the original model. This approach is discussed in [6, 23, 11]. For every discrete choice $\vec{v}_k$

$$\vec{v} \in V, \qquad V = \{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_k, \ldots, \vec{v}_{n_v}\} \tag{3.16}$$

a weighting factor

$$\vec{w}_i = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,n_v} \end{bmatrix}, \qquad w_{i,k} \in [0, 1], \qquad k = 1, \ldots, n_v \tag{3.17}$$

is created. Overall, they make up a selection vector $\vec{w}_i$ of the current discretized time step i. A discrete choice is selected by setting the corresponding weight to one. Every $w_{i,k}$ can be seen as an additional control variable over time. Thus, the optimization vector $\vec{Z}$ grows significantly.

Instead of one evaluation per time step i, the model dynamics are evaluated for all discrete control choices

$$\dot{\vec{x}}_i = \sum_{k=1}^{n_v} w_{i,k} \cdot \vec{f}_k\left(\vec{x}_i, \vec{u}_i, \vec{v}_k\right) \tag{3.18}$$

and weighted using the multipliers $w_{i,k}$. The weighted state derivative $\dot{\vec{x}}_i$ is then used in the model simulation. Additionally, at every time step i, the summation constraint

$$\sum_{k=1}^{n_w} w_{i,k} = 1 \tag{3.19}$$

must be fulfilled. The weights make the switching structure subject to optimization. If a discrete choice is optimal, the optimizer can set the corresponding weight $w_{i,k}$ to one. All other weights at the current time step have to be zero.

In the OC, the model dynamics are always evaluated using the original discrete control values $\vec{v} \in V$. Due to the reformulation, all $\vec{v}_k$ choices become a constant in the overall OCP. Thus, potential issues with intermediate variable do not arise.

Although the weights are relaxed in the optimal solution, all weights should belong to the set

$$w_{i,k} \in \{0, 1\}\,. \tag{3.20}$$

Fractional values of $w_{i,k}$ occur if the optimal solution for a discrete control lies between two discrete values. A similar problem arises in case the optimal switch occurs between two discretization steps. Therefore, penalty approaches have to be applied (see section 3.4). [11] proves that a binary solution of the relaxed / convexified problem has the same optimal solution as the original MINLP.

One major drawback of this approach is the number of additional optimization variables that are introduced in the NLP. Dependent on the number of discrete control choices, the number of optimization variables in the $\vec{Z}$ vector can easily be doubled. Therefore, an efficient optimal control toolbox is mandatory.

3.2.6 Variable Time Transformation

The Variable Time Transformation (VTT) introduced by [51] is very similar to the OC. It is also known as the control parameter enhancing technique [53, 54, 108]. As before, the weights $w_{i,k}$ for every discrete option are created and the summation constraint (3.19) must hold at every time step. Both methods differ in the evaluation of the model dynamics. Whereas the OC weights the model evaluations per time step, the VTT scales sub grid intervals in an integration.

Each step in the state discretization (major grid)

$$t_{i+1} = t_i + h_i \tag{3.21}$$

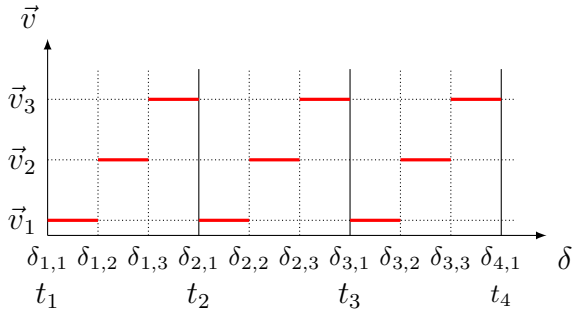
is divided into n_v sub steps

$$\delta_{i,k} = t_i + (k-1) \cdot \frac{h}{n_v}, \qquad k = 1, \ldots, n_v \tag{3.22}$$

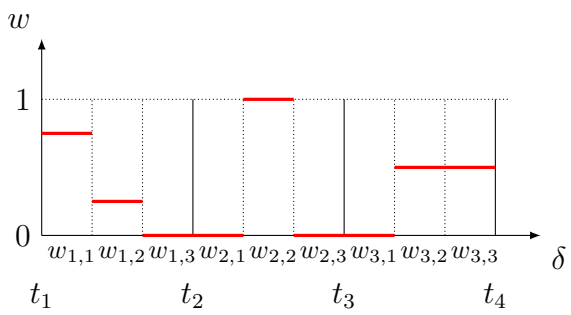
resulting in the minor grid. Each minor grid step $\delta_{i,k}$ is assigned to a discrete control $\vec{v}_k$ (see Figure 3.2a).

The system dynamics are integrated on the minor grid

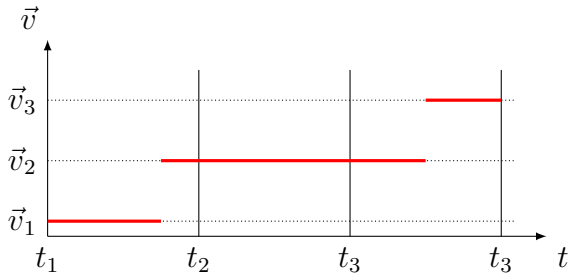
$$\dot{\vec{x}}_{i,k} = w_{i,k} \cdot \vec{f}\,(\vec{x}_{i,k}, \vec{u}_i, \vec{v}_k) \tag{3.23}$$



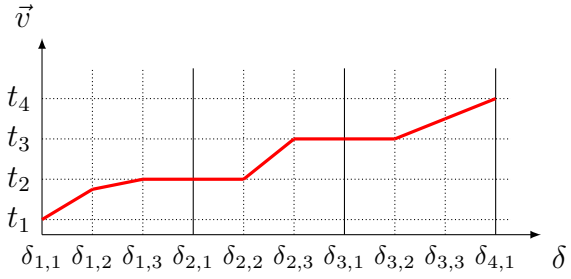
(a) Distribution of discrete control choices on the minor grid.



(b) Example for the discrete control weights.



(c) Discrete control history with applied variable time transformation for the example.



(d) Time transformation for the example.

Figure 3.2: Variable time transformation.

where $w_{i,k}$ scales the sub integration time intervals. For a value of $w_{i,k} = 0$ the minor grid interval is reduced to zero. Thus, it deactivates the discrete control in the major grid step. In case of $w_{i,k} = 1$ all other discrete choices have to be zero and the discrete control is stretched over the whole time interval (see second major grid step in Figure 3.2). Due to the time transformation of the minor integration steps, the choice of the discrete control is subject to optimization.

3.2.7 Comparison and Remarks

Table 3.1 compares the different approaches w.r.t. the evaluation criteria explained above (see section 3.2.1). The multi-phase approach as well as the Inner Convexification are both less viable choices for complex OCPs. Reasons have been discussed in the respective sections above. The HT approach can be used if the switching structure is well known [16, 20]. Furthermore, it can be used to formulate a continuous change between two discrete variables. Both OC and VTT are viable choices. In this thesis, the OC is used. The reasons for this choice are outlined in the following together with further remarks.

Binary Feasibility

As already mentioned, the OC as well as the VTT relax the weights on the interval $w_{i,k} \in [0, 1]$. Therefore, non-binary choices may appear in the optimal solution.

Table 3.1: Comparison of discrete control reformulation approaches.

	MP	**HT**	**IC**	**OC/VTT**
Automatic Switching Sequence			✓	✓
Independent Discrete Controls		✓	✓	✓
Instant / Continuous Switch	instant	continuous	instant	instant
Discrete Values Feasibility in Dynamics	✓			✓
Required Optimization Stages	1	mutliple	multiple	few

In most cases, a valid binary choice is found by the optimization, especially if the choices of the discrete control have a significant impact on the over cost. However, at switching points where the optimal solution switches to another discrete value, fractional values for $w_{i,k}$ may occur. In the following, these effects are mitigated by two influences.

The discrete constraints, which are introduced in section 3.3, eliminate infeasible discrete choices since constraints of the system dynamics must be fulfilled. Additionally, switching costs are introduced in section 3.4 mitigating remaining fractional solutions by enforcing a binary selection. Both approaches have a great impact on the solution of the OCP and must be chosen with care.

Realistic switches

The OC as well as the VTT implement instant switches. In reality, the change may be continuous from one discrete value to the next. The flaps on an aircraft can be regarded to change their setting linearly. Additionally, during an automotive gear change, the torque flow from the engine is disconnected from the wheels. Such dead times cannot be modeled with the approaches presented above.

A continuous change may be modeled by the HT approach. Alternatively, it would be possible to add dynamics to the system equations to implement continuous changes between discrete choices. However, in this case the optimization can exploit the dynamics to maintain a non-discrete value with periodic switching. Furthermore, the intermediate values have to be supported by the system dynamics. In the applications discussed in this thesis, the switching duration can be neglected w.r.t. the overall time frame. Therefore, this matter is not discussed further.

Variable Time Transformation vs. Outer Convexification

Due to the fact that the VTT approach integrates the dynamics along the minor grid it has the ability to model a switch in between two discretization time steps. However, this works only in the direction the discrete control selections are ordered on the minor grid interval. As shown in Figure 3.3, a switch in the ordered direction results in a switch within the major grid interval. In the opposite direction a non-realistic double switch occurs.

In this thesis, the OC approach is used in favor to the VTT for two reasons. Firstly, the discrete constraints and the switching cost approaches aim at producing integer solutions $w_{i,k} \in \{0, 1\}$ on the weights. In the binary feasible case, both approaches

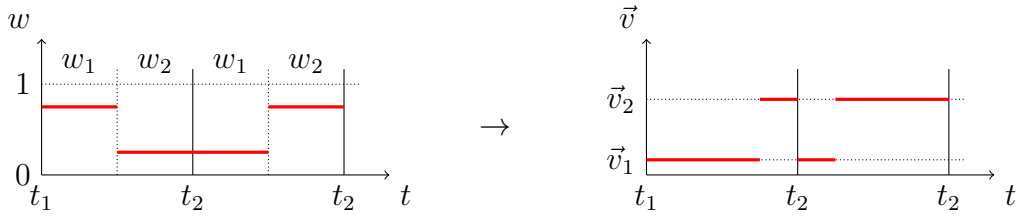


Figure 3.3: Switching direction of variable time transformation.

produce similar results. Secondly, the OC is much easier to implement in existing optimal control toolboxes.

Elimination of the Summation Equality Constraint

The last discrete control weight w_{i,n_v}

$$w_{i,n_v} = 1 - \sum_{k=1}^{n_v-1} w_{i,k} \geq 0 \tag{3.24}$$

can be calculate dependent on the others [23]. Thus, the number of optimization variables is reduced and the equality constraint

$$1 = \sum_{k=1}^{n_v} w_{i,k} \tag{3.25}$$

is replaced by the inequality constraint (3.24). Although the problem size is reduced, the sparsity is decreased. Every constraint dependent on the last weight w_{i,n_w} becomes dependent on all discretized weights. Thus, the dependencies become slightly more complex.

In this thesis, the full discretization is used in order to maintain the sparsity. Additionally, discretizing the weights fully is easier to implement as fewer special cases have to be considered.

3.3 Discrete Constraints Reformulation Methods

In an OCP containing discrete controls, there may be constraints that depend on the choice of the discrete controls. These are called mixed integer constraints or discrete constraints. They become active in an OCP if the corresponding discrete control is selected but must not influence the OCP in the other case. On an aircraft the gear or high lift dependent speed bounds are an example.

As with the discrete controls, these types of constraints introduce discontinuities in the OCP. Therefore, a continuous reformulation is necessary. This formulation is in turn dependent on the modeling approach used for the discrete controls. In the following, discrete constraints w.r.t. to OC and VTT are discussed. A formulation for the HT approach can be found in appendix A.2.

Apart from a continuous differentiable formulation of the discrete constraints, there is one other criterion. Even for smaller values of the weights $w_{i,k}$ the constraint must have a significant influence on the OCP. The ideal constraint

$$\vec{g}\left(\vec{x}(t), \vec{u}(t), t, \vec{v}(t)\right) \leq \begin{cases} 0 & w_{i,k} > 0 \\ \inf & w_{i,k} = 0 \end{cases} \tag{3.26}$$

is discontinuous and therefore must be approximated as closely as possible.

In the following the Vanishing Constraint approach is presented. Another formulation that was initially tried by the author can be found with the Stretching Constraints [16]. However, due to the fact that a parameter has to be driven to a large value this approach is less suitable for the approaches discussed in this thesis.

3.3.1 Vanishing Constraints

This approach was first introduced by Achtziger [58] as a Mathematical Programm with Vanishing Constraint (MPVC) and is used in many different applications [33, 109, 62, 66, 64]. Achtziger applied the vanishing constraints to a structural problem where a truss layout was optimized [58]. In case a connection was not needed, its corresponding constraints should vanish from the optimization problem as well, thus giving the name. Vanishing constraints can also be applied to optimal control problems.

The Vanishing Constraint

$$H(\vec{z}) \cdot G(\vec{z}) \leq 0, \qquad H(\vec{z}) \geq 0 \tag{3.27}$$

formulates a constraint where the inequality constraint G must be fulfilled in case a control function H is greater than zero. For $H = 0$ the constraint GH is automatically fulfilled. The inequality constraint G becomes unbounded and thus vanishes from the optimization problem.

This formulation is ideal for the OCP and VTT approaches, as the weights $w_{i,k}$ can be used as the control function H. Thus, the vanishing constraint considered in this thesis can be formulated to

$$w_{i,k} \cdot \vec{g}(\vec{x}, \vec{u}, t, \vec{p}, \vec{v}_k) \leq 0, \qquad w_{i,k} \geq 0 \tag{3.28}$$

where the discrete control set variable $\vec{v}$ remains a constant of the OCP. The constraint is only considered if the corresponding discrete control is selected. It can easily be seen that (3.27) and (3.28) are both similar. For better readability, in the following the notation of (3.27) is used.

Figure 3.4 shows the feasible set of the vanishing constraint (3.27). The set is clearly non-convex and violates most constraint qualifications at the origin [66, 64]. Thus, the KKT conditions cannot be used to determine optimality. Therefore, a relaxation or a reformulation for the vanishing constraint is proposed. Both are discussed further below.

MPVC are very similar to Mathematical Programm with Equilibrium Constraints (MPEC) [11, 6] where the equality constraint

$$H(\vec{z}) \cdot G(\vec{z}) = 0, \qquad H(\vec{z}) \geq 0 \tag{3.29}$$

forces at least one of $H(\vec{z})$ or $G(\vec{z})$ to zero. Apart from the fact that these type of constraints are much harder to solve [23, 66], they are not within the scope of this thesis and thus not further discussed.

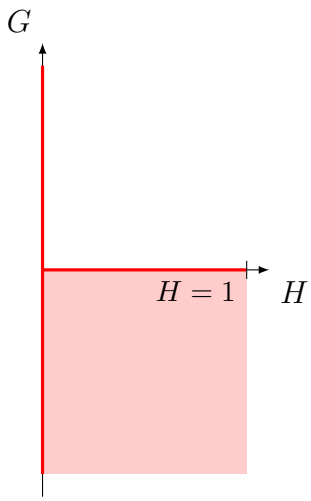


Figure 3.4: Feasible set of vanishing constraints.

3.3.2 Vanishing Constraint Relaxation and Reformulation

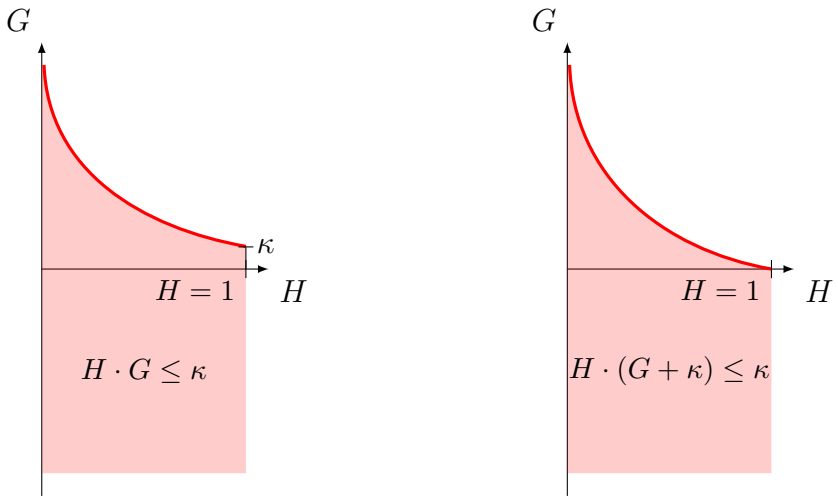
In the following, the relaxation and reformulation approach for the vanishing constraints are presented.

Relaxation Approach

Due to the violation of the constraint qualification of the vanishing constraints, [62] proposes a relaxed formulation

$$H(\vec{z}) \cdot G(\vec{z}) \leq \kappa, \qquad \kappa \geq 0 \tag{3.30}$$

where κ is a small value greater than zero. Thus, the feasible set at the origin becomes rounded off (see Figure 3.5a). The constraint formulation remains non-convex, but the constraint qualification is no longer violated.



(a) Classic relaxation approach. (b) Corrected relaxation approach.

Figure 3.5: Relaxation of vanishing constraints feasible set.

The relaxation parameter κ is subject to the application involved. In this thesis, values at around $\kappa \approx 1 \cdot 10^{-3}$ are used. Due to the relaxation, the physical constraint

$$\vec{g}(\vec{x}, \vec{u}, t, \vec{p}, \vec{v}_k) \leq \kappa, \qquad w_{i,k} = 1 \tag{3.31}$$

may be violated by κ. In the optimization problem discussed in this thesis, such a violation can be neglected. In case this violation must be prohibited, the reformulation

$$H(\vec{z}) \cdot (G(\vec{z}) + \kappa) \leq \kappa \tag{3.32}$$

is proposed (see Figure 3.5b). The relaxation of the vanishing constraint was applied in [16, 18, 33].

Reformulation Approach

The vanishing constraint can also be reformulated [66, 109]. It is approximated by an inequality constraint

$$\varphi^{\kappa}(G, H) := \frac{1}{2}\left(GH + \sqrt{G^2H^2 + \kappa^2} + \sqrt{H^2 + \kappa^2} - H\right) \leq \kappa, \qquad \kappa > 0 \tag{3.33}$$

where κ is a relaxation parameter. The feasible set structure is very similar to 3.5a.

3.3.3 Slack Variable Expansions

As was stated above, all formulation approaches of the discrete constraints require a relaxation variable. The difficulty that arises is how the value is chosen. Ideally, for the vanishing constraints and the reformulation, it should be very small in order to approximate the original formulation. A trade-off has to be made to avoid numerical difficulties.

To bypass this problem, the relaxation parameter κ can be introduced as an additional slack control

$$\kappa_u \in [\kappa_{u,\min}, \kappa_{u,\max}], \qquad 0 \leq \kappa_{u,\min} \leq \kappa_{u,\max} \tag{3.34}$$

in the optimization problem. It is discretized on the same grid as the vanishing constraint. Additionally, it is introduced as a penalty cost

$$J_P = \sum_i \left(\kappa_{u,i} - \kappa_{u,\min}\right) \tag{3.35}$$

which is added to the overall cost function. The slack control is initialized with a suitable value (e.g. $\kappa_u = \kappa_{u,\max} = 1 \cdot 10^{-3}$). If the discrete choice is clear, the optimization algorithm can reduce the corresponding κ_u to its lower bound (e.g. zero). In case the choice of discrete control is still unclear, the slack variable can be set to a higher value to relax the constraint. Thus, a higher relaxation variable may be used in the OCP.

3.4 Minimization of Switches and Binary Feasibility

If the switching structure is subject to optimization, the optimal solution may switch at every discretized point. There exist even discrete optimal control problems where the analytical solution switches infinite number of times on any arbitrary small time scale [22]. In many applications, the number of switches must be limited. Reasons may include mechanical wear, comfort, or other limitation by the dynamic system or process considered.

In order to limit the number of switches, additional constraints, or costs are introduced in the OCP. The formulation of these is dependent on the reformulation approach used for the discrete controls. Therefore, in this section, the OC and the VTT approach are considered.

Apart from the minimization of switches, the binary feasibility $w = \{0, 1\}$ shall be ensured. In this section, a novel approach, which performs both tasks at the same time, is presented.

3.4.1 Switching Constraints and Rounding Approaches

In this section, existing approaches are discussed. These include constraints, roundig approaches as well as penalty formulations.

Switching Constraints

In order to ensure binary feasibility of the weights w a simple suggestion is to include a constraint of the form

$$w \cdot (1 - w) = 0 \tag{3.36}$$

which is fulfilled for the binary case [68, 69]. The main problem of this approach was already discussed with the Inner Convexification approach (see 3.2.4). The feasible set is non-convex and disjoint making the OCP hard to solve. Additionally, this approach only targets binary feasibility but does not penalize discrete control switching.

Another possible constraint formulation

$$w_{i,k_1} \cdot w_{i+1,k_2} = 0, \qquad k_1 \neq k_2 \tag{3.37}$$

prevents switches of a discrete control in certain directions (e.g. from k_1 to k_2) [17]. However, the constraints are of type MPEC and are thus hard to solve [66].

Rounding

Another idea is to optimize the problem with the OC approach and to apply a rounding strategy afterwards. This idea was suggested and successfully used by [11, 23, 56]. This so-called "sum-up-rounding" approach efficiently reduces the number of switches and ensured binary feasibility. However, due to the fact that the control history is altered, the augmented solution of the OCP may no longer be feasible. An additional optimization with fixed discrete controls may even fail. Therefore, the application of rounding schemes is still subject of research. Recent advances can be found in [110].

In this chapter, a novel switching cost penalty is introduced that reduces the number of switches and ensures binary feasibility. Therefore, rounding strategies are not a focus of this thesis.

3.4.2 Existing Penalty Approaches

All penalty approaches introduce local minima in the optimal control problem. It is the only way a binary feasible switching structure can be enforced during the optimization. Therefore, these penalty approaches have a huge impact on the optimal solution. In the following, cost penalty approaches shall be discussed that ensure binary feasibility and / or minimize the number of switches. All have been applied to MIOCP successfully but have different drawbacks.

Binary Penalty

A simple way to ensure binary feasibility of the discrete control weights is by imposing the penalty

$$J_P = \alpha \cdot \sum_{i,k} w_{i,k} \cdot (1 - w_{i,k}) \tag{3.38}$$

where α is a scaling factor. This penalty approach was used by [11, 68]. Although the formulation has been successfully used it has some drawbacks:

- Only the current time discretization point is considered in the penalty formulation. Therefore, this approach may only ensure binary feasibility of $w_{i,k}$ but does not reduce the number of switches.

- The formulation is similar to the switching constraint introduced above. It is reformulated as a cost penalty in order to avoid the disjoint feasible set. The tradeoff is that local minima are introduced on every $w_{i,k}$ (see Figure 3.6). This effect is mainly influences by the magnitude of the scaling parameter. For large values, a binary feasible initial guess will most likely not be altered during optimization. This penalty formulation figuratively "freezes" the switching structure in place.

- In order to improve the local minima situation, [11] suggests the use of a homotopy approach. Multiple optimal control problems are solved where the penalty scale α is gradually increased from a small value to a large value of suitable magnitude. However, the selection of these and the number / distribution of the homotopy steps are unclear and subject to the specific application.

- Binary feasibility can only be ensured for α approaching infinity. This is not possible in practical applications as it introduces bad scaling in the problem gradient.

Another similar penalty approach that targets binary feasibility is used by [15]. The weights $w \in [0, 1]$ are replaced by $\beta \in [-1, 1]$ with a linear transformation

$$w = \frac{1}{2}(1 - \beta). \tag{3.39}$$

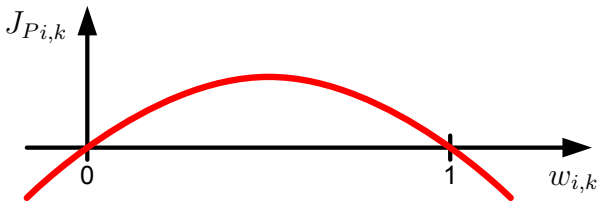


Figure 3.6: Binary penalty approach.

Additionally, a penalty cost is formulated

$$J_P = \alpha \cdot \sum_{i,k} l\left(|\beta_{i,k}|\right) \tag{3.40}$$

where $l : [0,1] \to \mathbb{R}$ represents a strictly monotone decreasing function with $l(1) = 0$ (e.g. linear). Apart from the fact that this penalty formulation is not continuously differentiable, the same issues as above can be expected.

Discrete Constraint Dependent Switching Cost

As with the binary feasibility, the switching constraint can be reformulated as a penalty cost

$$J_P = \alpha \cdot \sum_{i} w_{i,k_1} \cdot w_{i+1,k_2} \qquad k_1, k_2 \in \{1, \ldots, n_v\}. \tag{3.41}$$

Thus, a switch from the discrete control selection k_1 to k_2 between the time steps t_i and t_{i+1} is penalized. Obviously, the equation above must be expanded to account for other switching combinations and enables to formulate switching costs dependent on the direction of a switch. Although the idea of this approach was successfully applied [16] it requires the consideration of many different combinations. However, it is possibly a good choice to introduce additional penalties in order to prevent certain switches.

Switching Cost Penalty by Kirches

An application independent formulation of switching cost was proposed by [23]. It combines two adjacent time discretization points of the discrete control weights,

$$J_P = (2 \cdot \gamma_{i,k} - 1) \cdot (w_{i,k} + w_{i+1,k} - 1) + 1, \qquad \gamma_{i,k} \in [0,1] \tag{3.42}$$

where $\gamma_{i,k}$ represents a slack value that is introduced as an additional optimization variable. Dependent on the sum of both adjacent weights $w_{i,k} + w_{i+1,k}$ the optimal value for $\gamma_{i,k}$ changes:

$$w_{i,k} + w_{i+1,k} < 1 \qquad \gamma_{i,k} \to 1 \tag{3.43}$$

$$w_{i,k} + w_{i+1,k} = 1 \qquad \gamma_{i,k} = free \tag{3.44}$$

$$w_{i,k} + w_{i+1,k} > 1 \qquad \gamma_{i,k} \to 0. \tag{3.45}$$

Figure 3.7 shows a visual representation of this approach. The cost formulation formulates a plane that changes its optimal tilt $\gamma_{i,k}$ dependent on the sum $w_{i,k} + w_{i+1,k}$. Thus, the optimal solution is either $w_{i,k} = w_{i+1,k} = 0$ or $w_{i,k} = w_{i+1,k} = 1$ resulting in zero cost and binary feasibility. A switch in the discrete control is penalized.

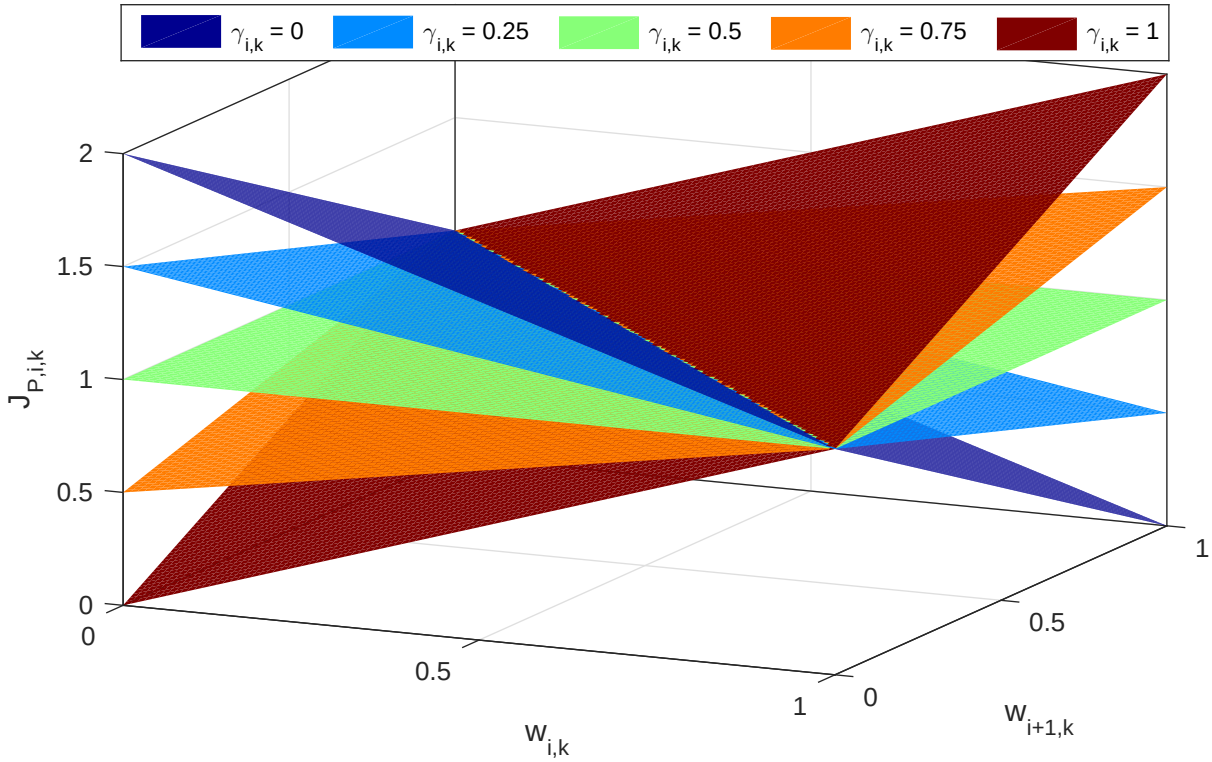


Figure 3.7: Visual representation of the switching cost approach introduced by [23].

This approach was successfully used in [23, 21, 18, 33]. However, for every discrete control weight an additional optimization variable must be introduced. Thus, the number of variables accounting for the discrete controls are doubled, having a huge impact on the overall, already large, problem size.

3.4.3 Multi-Time Switching Cost Penalty

The Multi-Time Switching Cost approach, initially presented by the author of this thesis in [111], is a novel formulation that aims at low frequent switches in the discrete controls. It is based on a simple assumption. Assume the current time step (t_i) has to adjacent time steps (t_{i-1}, t_{i+1}) with the same discrete choice. Then, the discrete choice at the current time (t_i) step shall be forced to the same selection. Thus, imaginary springs are introduced in the optimal control problem (see Figure 3.8).

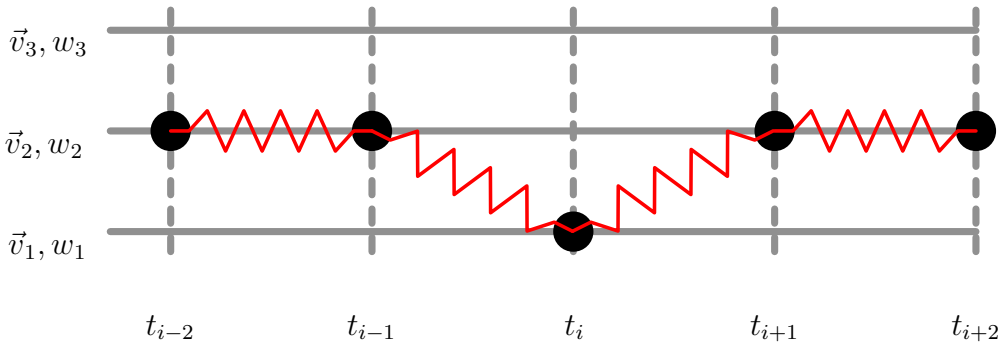


Figure 3.8: Multi-Time switching cost idea.

The following description is partially taken from the author's publication [111]. A switching penalty is created

$$J_{SC,i,k} = w_{i-1,k} + w_{i,k} \cdot (1 - 2 \cdot w_{i-1,k}) + w_{i+1,k} + w_{i,k} \cdot (1 - 2 \cdot w_{i+1,k}) \tag{3.46}$$

which uses the three adjacent discretized points in time (previous $i-1$, current i and next $i+1$) of the discrete control weights. The cost function is evaluated for every discrete control control choice k at every discretized point in time i. It can be split into two influences, namely the influence of the previous and of the following discretized time point. Both adjacent weights control the pitch of a linear function pivoting around the point $(0.5, 0.5)$ (see. Figure 3.9). The linear function is evaluated at the current discretized point in time $w_{i,k}$. Thus, deviations between the weights are penalized.

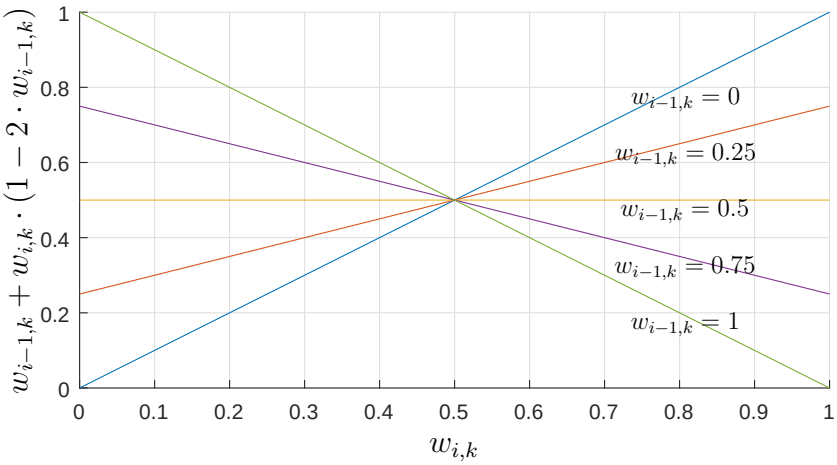


Figure 3.9: Multi-Time switching cost pivot function.

In Figure 3.10 the cost function is evaluated for different value combinations

$$w_{i-1,k}, w_{i+1,k} = \{0, 0.25, 0.5, 0.75, 1\} \tag{3.47}$$

of the previous $w_{i-1,k}$ and next $w_{i+1,k}$ discretized point in time. In the blue plots, the partial influence is shown, the red plots show the combined cost. In all cases, the cost values are plotted over the current discretized weight $w_{i,k} \in [0, 1]$. It can be seen that if one of the adjacent weights is closer to the desired binary choice,

$$w_{i+1,k} + w_{i-1,k} > 1 \quad \Rightarrow \quad w_{i,k} \to 1 \tag{3.48}$$

$$w_{i+1,k} + w_{i-1,k} < 1 \quad \Rightarrow \quad w_{i,k} \to 0 \tag{3.49}$$

then, due to the cost function, the optimal value for the current weight $w_{i,k}$ becomes the closer binary choice. If the sum of the adjacent weights is equal to one

$$w_{i+1,k} + w_{i-1,k} = 1 \tag{3.50}$$

the resulting line is flat. In this case the decision is solely dependent on the discrete control's influence on the cost function J.

This approach has the following benefits: Only the discrete control weights w contribute to the cost. Therefore, the approach can be applied to MIOCP of different applications. In case no switch occurs, the resulting penalty cost is zero. Any fractional

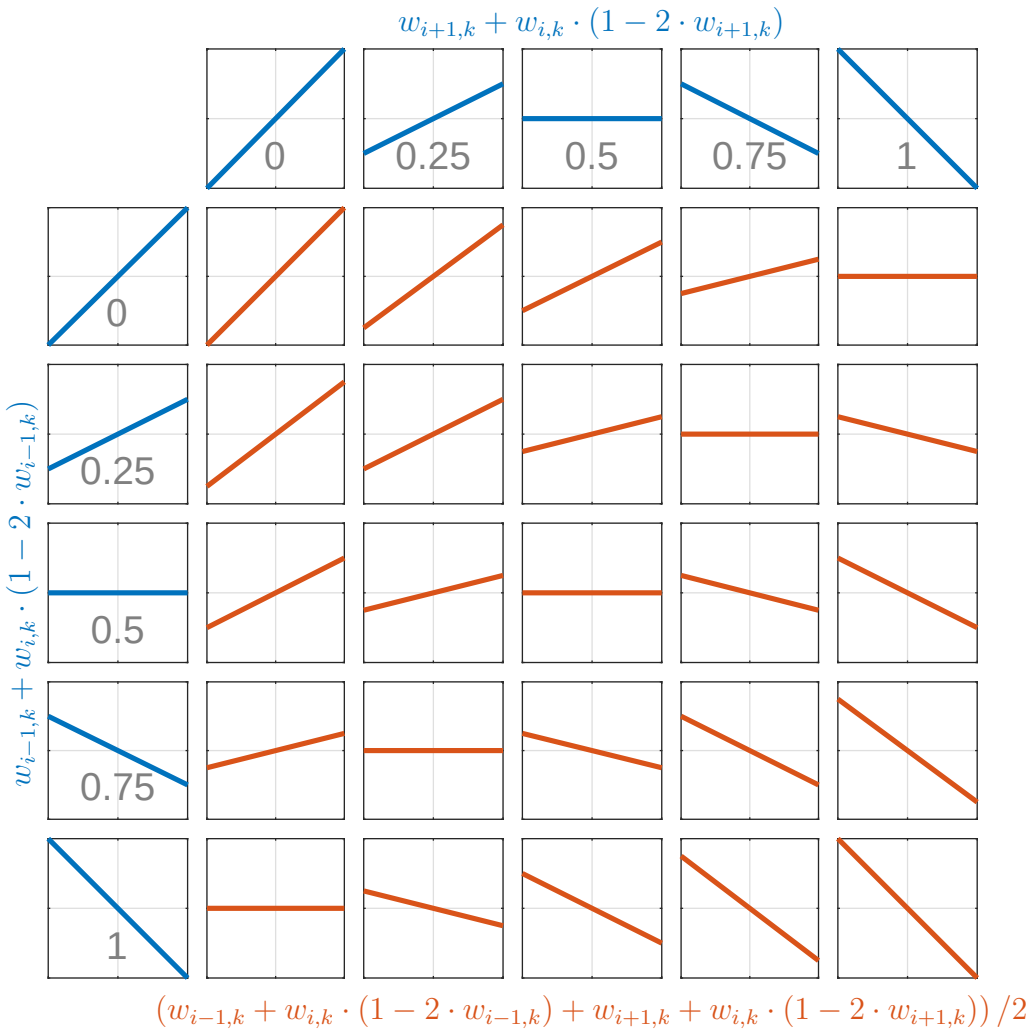


Figure 3.10: Multi-Time switching cost combination evaluation. All plot axes range from -1 to 1 and all x-axis represent $w_{i,k}$. The y-axis represent the respective colored penalty.

solution of the weights w results in a positive contribution to the penalty that the optimizer will try to eliminate. Additionally, no slack variables are required and the problem size remains the same.

If desired, a slack variable δ can be introduced in the penalty formulation

$$J_{SC,i,k} = [w_{i-1,k} + w_{i,k} \cdot (1 - 2 \cdot w_{i-1,k}) + w_{i+1,k} + w_{i,k} \cdot (1 - 2 \cdot w_{i+1,k})] \cdot (1 - \delta_i), \quad (3.51)$$

$$\delta_i \in [0, 1[. \quad (3.52)$$

It enables the optimization algorithm to reduce the impact of the switching cost locally. Additionally, the slack variables are introduces as an additional cost penalty

$$J_P = \sum_i \delta_i \quad (3.53)$$

in the overall cost function. Note that only one slack variable is introduced per time step. Thus, the number of optimization variables does not increase as drastically compared to [23].

3.5 Multiple Discrete Controls

Many dynamic systems may have multiple discrete control inputs. Regarding the OC approach, multiple discrete controls can always be considered by creating a set of combinations (Cartesian product). Let $\vec{v}$ and $\vec{\eta}$ be two discrete controls with their respective sets $\{\vec{v}_1, \dots \vec{v}_{n_v}\}$ and $\{\vec{\eta}_1, \dots \vec{\eta}_{n_\eta}\}$. The cartesian product of both is given by

$$\{\vec{v}_1, \dots \vec{v}_{n_v}\} \times \{\vec{\eta}_1, \dots \vec{\eta}_{n_\eta}\} = \left\{ \begin{array}{ccc} (\vec{v}_1, \vec{\eta}_1) & \dots & (\vec{v}_1, \vec{\eta}_{n_\eta}) \\ \vdots & \ddots & \vdots \\ (\vec{v}_{n_v}, \vec{\eta}_1) & \dots & (\vec{v}_{n_v}, \vec{\eta}_{n_\eta}) \end{array} \right\} \tag{3.54}$$

which is represented in a matrix for better readability. It contains all possible combinations as tuples and can thus be regarded as a single discrete control. The process can be repeated to account for additional discrete controls.

Applying the outer convexification results in a matrix of weights

$$W_{\vec{v},\vec{\eta}} = \left[\begin{array}{ccc} w_{\vec{v}_1,\vec{\eta}_1} & \dots & w_{\vec{v}_1,\vec{\eta}_{n_\eta}} \\ \vdots & \ddots & \vdots \\ w_{\vec{v}_{n_v},\vec{\eta}_1} & \dots & w_{\vec{v}_{n_v},\vec{\eta}_{n_\eta}} \end{array} \right] \tag{3.55}$$

which is used in the evaluation of the dynamics. It is clear, that the number of optimization variables increases drastically. In some cases unrealistic combinations can be omitted. Additionally it can be seen, that the sum of rows and columns represent the summed weights for the individual discrete values of $\vec{v} \in \{\vec{v}_1, \dots \vec{v}_{n_v}\}$ and $\vec{\eta} \in \{\vec{\eta}_1, \dots \vec{\eta}_{n_\eta}\}$. A row or column sum gives the weight the discrete control would have if it entered the dynamic system alone. Thus, the summed weight can be used to formulate discrete constraints and switching cost independent of other discrete controls and combinations. In the following, the matrix $W_{\vec{v},\vec{\eta}}$ is rewritten as a vector

$$\vec{w}_{\vec{v},\vec{\eta}} = \mathrm{vec}\left(W_{\vec{v},\vec{\eta}}\right) \tag{3.56}$$

where the columns of $W_{\vec{v},\vec{\eta}}$ are stacked vertically below each other.

Mapping

In order to generalize the example above, a mapping matrix M is defined for each discrete control

$$\left[\begin{array}{c} w_{\vec{v}_1} \\ \vdots \\ w_{\vec{v}_{n_v}} \end{array} \right] = M_{\vec{v}} \cdot \vec{w}_{\vec{v},\vec{\eta}} \tag{3.57}$$

$$\left[\begin{array}{c} w_{\vec{\eta}_1} \\ \vdots \\ w_{\vec{\eta}_{n_\eta}} \end{array} \right] = M_{\vec{\eta}} \cdot \vec{w}_{\vec{v},\vec{\eta}} \tag{3.58}$$

that maps the discrete control combination vector $\vec{w}$ to the weights of the individual sets of each discrete control. The mapping matrices are constant and the vector $\vec{w}$ is a column vector representation of matrix W. Additionally, it is possible to define a mapping for any logical combination of discrete controls.

Reverse Mapping

The mapping of the discrete controls is used in section 3.6 to alter the optimal solution. Therefore, it is necessary to define a reverse mapping

$$\vec{w}_{\vec{v},\vec{\eta}} = \begin{bmatrix} M_{\vec{v}} \\ M_{\vec{\eta}} \end{bmatrix}^{"-1"} \cdot \begin{bmatrix} w_{\vec{v}_1} \\ \vdots \\ w_{\vec{v}_{n_v}} \\ w_{\vec{\eta}_1} \\ \vdots \\ w_{\vec{\eta}_{n_\eta}} \end{bmatrix} \tag{3.59}$$

to the discrete control weights of the Cartesian product. However, as the combined mapping matrix does not have full rank, this operation cannot be carried out directly.

Due to the fact that the weight w_i must be greater than or equal to zero, the reverse mapping is unique and can be solved with a non-negative least square problem:

$$\min_{\vec{w}_{\vec{v},\vec{\eta}}} \left\| \begin{bmatrix} M_{\vec{v}} \\ M_{\vec{\eta}} \end{bmatrix} \cdot \vec{w}_{\vec{v},\vec{\eta}} - \begin{bmatrix} w_{\vec{v}_1} \\ \vdots \\ w_{\vec{v}_{n_v}} \\ w_{\vec{\eta}_1} \\ \vdots \\ w_{\vec{\eta}_{n_\eta}} \end{bmatrix} \right\|_2^2, \qquad \vec{w}_{\vec{v},\vec{\eta}} \geq 0. \tag{3.60}$$

3.6 Solution Strategies

This section discusses aspects of the solution strategies for MIOCP. In general, it is not possible to solve such problems in a single optimization step. Therefore a two stage approach is used.

3.6.1 Two-Stage Optimization

As discussed above, all switching cost approaches introduce local minima in the OCP. This fact prevents the optimization algorithm to find the optimal switching sequence. In practice, this means that the optimal solution found remains close to the initial guess provided by the user. A strategy is required that finds the optimal switching sequence even for primitive / bad initial guesses of the discrete controls.

In this thesis, a two staged optimization strategy is used (see Figure 3.11). In the first stage, the OCP is solved with disabled switching cost:

$$J = J_{cost} + 0 \cdot J_P. \tag{3.61}$$

Thus, the optimization algorithm can find the optimal discrete control setting at every time step individually. The first optimization stage can also be regarded as an initial guess generation for the optimal switching sequence. In areas of the solution where

one discrete control choice is clearly optimal, binary feasibility will appear automatically. However fractional values may occur, especially around switches. These have to be mitigated by the switching cost.

In the second stage, the switching costs are activated

$$J = J_{cost} + \alpha \cdot J_P \tag{3.62}$$

by setting the penalty scaling parameter to a suitable value. It shall be chosen in a way that the switching cost adds a significant contribution to the overall cost function. As a good starting point 20% to 40% of the main cost function may be chosen but in practice different choices must be tested. However, in section 5.3.1 it is shown that the multi-time switching cost formulation produces stable results over several magnitudes of α.

With the switching cost activated, the optimization algorithm is warm-started with the previous solution. Therefore, the solution and the constraint multipliers of the first optimization stage must be saved. Many solvers, such as IPOPT [40], offer special warm start features.

Due to the two-stage approach it is possible to augment the intermediate solution (see Figure 3.11). Thus, influences that are undesired can be removed. Such may be spikes in the optimal solution that may not be removed by the second optimization stage. This issue and the removing algorithm is explained in the section 3.6.3.

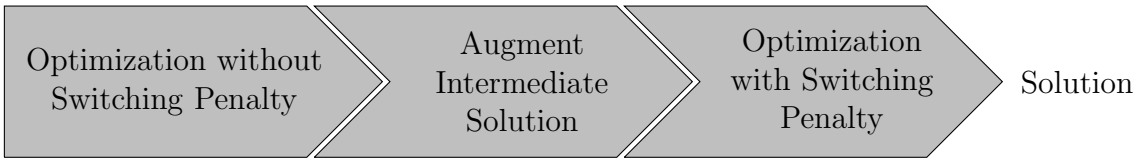


Figure 3.11: Discrete control two-stage solution approach with intermediate augmentation.

3.6.2 Step Representation

The discrete control weights of the OC are not intuitive in case a visual representation of the discrete control choice is required. Therefore, a transformation

$$\sigma_i = \begin{bmatrix} 1 & 2 & \dots & n_v \end{bmatrix} \cdot \begin{bmatrix} w_{i,1} \\ \vdots \\ w_{i,n_v} \end{bmatrix} \tag{3.63}$$

is used that maps the weights into an step shaped integer representation (see Figure 3.12).

The transformation assumes, that all weights are binary feasible. Thus, in the transformed form, the step representation is more meaningful than the weights alone. In case intermediate values are involved, the result represents a continuous change between two discrete values. The mapping is assumed to be applied to independent discrete control sets and not the Cartesian product. The step representation is not only used for visual purposes but also to allow manipulations of the switching sequence in the intermediate step (see section 3.6.3).

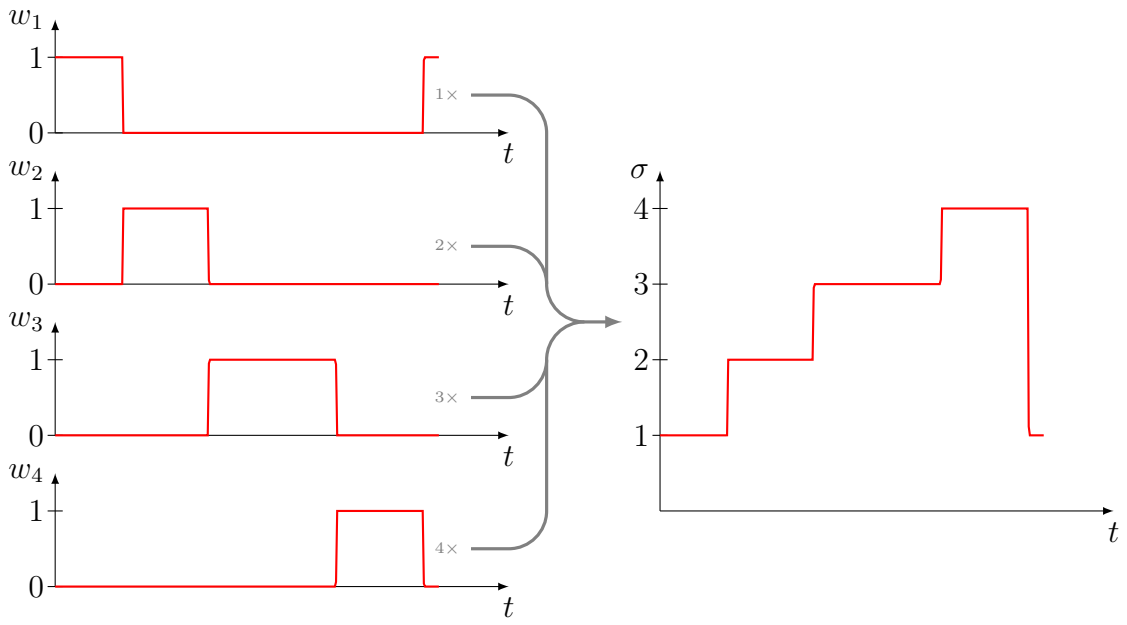


Figure 3.12: Step transformation of discrete control weights.

Since the intermediate solution is altered in the intermediate step, a back transformation is necessary. The weights are calculated

$$w_k = k + 1 - \sigma, \qquad w_{k+1} = \sigma - k, \qquad \sigma \in [k, k+1] \tag{3.64}$$

using the distances of the step representation to the neighboring integer values. All other weights are zero.

The step representation assumes that the discrete choices represent some sort of linear selection (gear changes, flap positions, etc.) and not independent sets. Additionally, the reverse transformation may introduce intermediate discrete control selections in case multiple discrete choices are skipped (gear change from 5^{th} to 1^{st} gear) and the change is continuous. However, these effects are eliminated by the multi-time switching penalty approach.

3.6.3 Spike Removal

The Multi-Time switching cost approach is able to remove high frequent switches from the optimal solution. However, there are situations where the discrete control switches to a setting for multiple discretization steps, yet remains much shorter than anticipated by realistic application (e.g. car switches gear for 0.2 seconds). Especially, discrete choices that overshoot the general selection may be undesired in the final solution (see Figure 3.13).

Dependent on the influence on the cost function and the fact that all switching cost approaches introduce local minima in the OCP, this unwanted discrete control selection may not be removed. Therefore, in this section, a filter algorithm is presented that eliminates short overshooting spikes in the optimal solution by manipulating the discrete control weights.

The algorithm uses the integer step representation explained in previous section. It is evaluated at the points in time $t, t + \Delta t$ giving the values $\sigma(t), \sigma(t + \Delta t)$. They

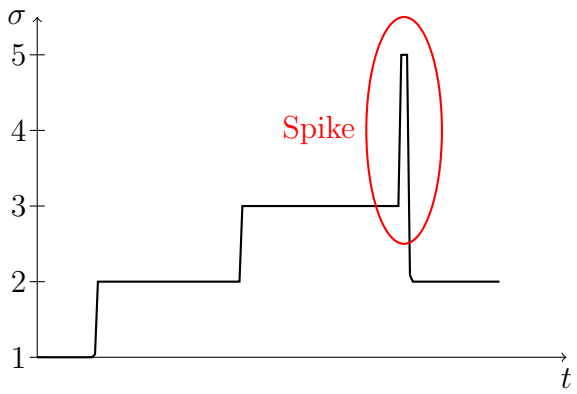


Figure 3.13: Overshooting spikes in step representation.

are used to define a box in the discrete control step history (see Figure 3.14). All step values that are outside are limited to the box bounds.

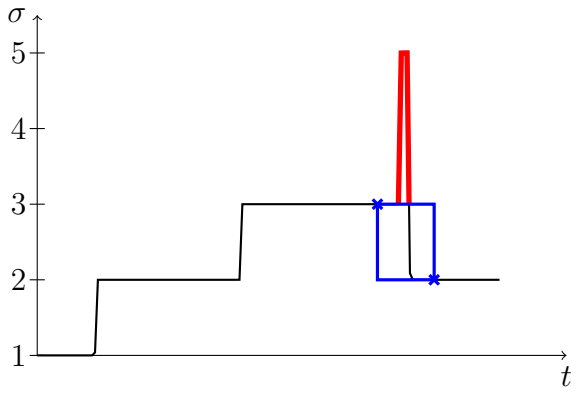


Figure 3.14: Box limit in the integer step representation.

The box filter is evaluated in the time interval $[t_0, t_f - \Delta t]$. The width of the box is a user specified time which can be used to define the width of the switches that shall be eliminated by this approach. After the filter processing, the resulting step history is transformed back to the OC weight representation.

It has to be noted that this filter approach is only suitable for discrete controls that are switched in a low frequency. Additionally, there is no guarantee that the undesired switch will not occur in the solution of the second optimization stage. The augmentation may only contribute an influence to a different local minimum. Furthermore, it cannot be stated in general how a removal of a high frequent switch influences the cost function. However, it can be seen in 5.3 that the intermediate augmentation of the solution improves the consistency within parameter studies involving many optimizations.

Chapter 4

Implementation of Optimal Control Framework *FALCON.m*

In this chapter, the *FALCON.m* optimal control toolbox, developed in course of this thesis, is introduced. It implements the collocation discretization method and is optimized for high-fidelity large scale optimal control problems.

Rather than using a commercial or free optimal control software suite, a new tool was developed. This enables more control in the development process of solution strategies. Especially for optimal control problems with discrete controls, a tool is required that is able to solve large optimal control problems with 60k to 600k optimization variables. *FALCON.m* is able to calculate the sparse analytic first and second order gradients of such problems on a consumer PC.

The chapter is organized as follows: Section 4.1 restates the optimal control problem that is implemented by *FALCON.m*. Section 4.2 explains the problem definition from the user side. This is complemented by a user guide providing more detail information. In section 4.3 the automatic analytic derivative generation of models, constraints and cost functions is explained from the user side. Section 4.4 explains the interface between user functions (dynamic models, constraints, and cost functions) and the *FALCON.m* toolbox. Afterwards, implementation details are discussed. Section 4.5 explains the toolchain behind the automatic analytic derivative generation of the user functions. The calculation of the overall sparse analytic problem derivatives is explained in section 4.6. The chapter closes in section 4.7 with an explanation of a discrete control extension to *FALCON.m*. It enables an user friendly approach to implement discrete controls in optimal control problems.

It is important to note that the development of *FALCON.m* was partially collaborative work. The user friendly definition of an optimal control problem in *FALCON.m* was designed among the team members of the trajectory optimization group (see section 4.2). The author of this thesis is responsible for the overall toolchain that calculates the analytic derivatives of user functions and the optimal control problem. This includes the definition of high fidelity models and user functions, the overall analytic derivative generation process of these as well as the calculation of the sparse analytic Jacobian of large scale optimal control problems. The problem Hessian calculation is based on the Jacobian algorithm. However, the implementation was collaborative work. Additionally, authorship is claimed for the implementation of the discrete control extension.

4.1 Optimal Control Problem

In this section, the optimal control problem implemented by *FALCON.m* is introduced. It is similar to the problem discussed in chapter 2. The problem is stated as follows: Minimize the cost function

$$J = M\left(\vec{y}(t_0), \vec{x}(t_0), t_0, \vec{y}(t_f), \vec{x}(t_f), t_f, \vec{p}, \vec{c}, \dots\right) + \int_{t_0}^{t_f} L\left(\vec{y}(t), \vec{x}(t), \vec{u}(t), \vec{p}, \vec{c}, \dots\right) dt \quad (4.1)$$

subject to the model dynamics

$$\dot{\vec{x}} = \vec{f}_x\left(\vec{x}(t), \vec{u}(t), \vec{p}, \vec{c}, \dots\right) \quad (4.2)$$

$$\vec{y} = \vec{f}_y\left(\vec{x}(t), \vec{u}(t), \vec{p}, \vec{c}, \dots\right) \quad (4.3)$$

where $\vec{x}$ is the state vector, $\vec{u}$ the control vector, $\vec{p}$ the parameter vector, and $\vec{c}$ multiple constant inputs entering the model or cost functions. All optimizable variables are limited by lower and upper bounds

$$\vec{x}_{LB} \leq \vec{x}(t) \leq \vec{x}_{UB}, \qquad \vec{u}_{LB} \leq \vec{u}(t) \leq \vec{u}_{UB}, \qquad \vec{p}_{LB} \leq \vec{p}(t) \leq \vec{p}_{UB}. \quad (4.4)$$

Additionally to the state derivatives, the model calculates an optional output vector $\vec{y}$. It contains additional or intermediate values that are calculated within the equations of the dynamic model. Similarly to the optimization variables, box constraint limits

$$\vec{y}_{LB} \leq \vec{y}(t) \leq \vec{y}_{UB} \quad (4.5)$$

can be defined for the model outputs. They enable a simple implementation of path constraints. For more complex cases, path constraints

$$\vec{g}_{LB} \leq \vec{g}\left(\vec{y}(t), \vec{x}(t), \vec{u}(t), \vec{p}, \vec{c}, \dots\right) \leq \vec{g}_{UB} \quad (4.6)$$

can be defined enabling any mathematical constraint formulation. The initial and final boundary conditions for the states

$$\vec{x}_{0,LB} \leq \vec{x}(t_0) \leq \vec{x}_{0,UB}, \qquad \vec{x}_{f,LB} \leq \vec{x}(t_f) \leq \vec{x}_{f,UB} \quad (4.7)$$

and times

$$t_{0,LB} \leq t_0 \leq t_{0,UB}, \qquad t_{f,LB} \leq t_f \leq t_{f,UB} \quad (4.8)$$

are set using box constraints as well. Equality constraints are considered by setting the lower and upper bounds to the same value.

The optimal control problem in *FALCON.m* may contain multiple phases (e.g. to introduce interior point constraints). By default all phases are independent. Model dynamics, constraints, boundary conditions, and cost functions can be defined for each phase individually. The state history of two phases is connected with a phase defect constraint

$$\vec{x}_{t_0,k_1} - \vec{x}_{t_f,k_2} = 0 \quad (4.9)$$

where k_1 and k_2 represent the phase indices. Since the phase defect is not bound to consecutive phases, periodic optimization problems can be formulated.

The phase defect is an example for a point constraint

$$\vec{h}_{LB} \leq \vec{h}(\vec{y}_{k_1}(t), \vec{x}_{k_1}(t), \vec{u}_{k_1}(t), \vec{y}_{k_2}(t), \vec{x}_{k_2}(t), \vec{u}_{k_2}(t), \dots, \vec{p}, \vec{c}, \dots) \leq \vec{h}_{UB} \tag{4.10}$$

spanning multiple phases. This constraint type enables a constraint formulation between any multiple points of the optimal control problem. This includes phase data (outputs $\vec{y}$, states $\vec{x}$, and controls $\vec{u}$) at various discretized time points, and additional parameters.

4.2 Problem Definition

This section gives an overview of the problem definition in *FALCON.m* regarding the most common usage. First, for better understanding, an overview about the general implementation and ideas of the toolbox is given. Afterwards, the problem definition is explained for the most common usage. Detailed information can be found in the official documentation [80].

4.2.1 General Principles

Before the problem setup in *FALCON.m* is described, this section shall give an overview on how the data is stored and handled within the optimal control toolbox. This includes, the definition of variables as well as their bounds, scaling, and offset. Time histories of variables are achieved through grids. The time discretization used in *FALCON.m* is introduced. Finally, a brief overview of the derivative handling with the toolbox is given.

Basic Implementation Idea

The *FALCON.m* optimal control toolbox is implemented as a library of *MATLAB* classes. For problem setup only a few classes are required. In order to sort classes and functions of the toolbox in hierarchical order, *MATLAB* namespaces are used. This has multiple benefits as it gives the tool a clear structure and only a single folder needs to be added to the *MATLAB* path. Furthermore, duplicate definitions by other toolboxes can be avoided easily since all functions and classes are referenced through the namespace `falcon`.

Problem definition is achieved through classes only. All parts that make up an optimal control problem such as models, constraints, or cost functions are implemented in their own respective class. Thus, a highly modular approach, which enables fast and efficient implementation of new features, is achieved. This modularity allows for high flexibility. Hence, a practically unlimited number of model inputs, phases, constraints, and cost functions can be used. It has to be noted that all cost functions are added together to become a single cost function. Due to efficient implementation of the core algorithms, even consumer PCs allow the solution of large optimal control problems.

All module classes belong to a single optimal control problem (`falcon.Problem`). As each child is instantiated by its parent, the factory method design pattern is implemented. Therefore, a new phase of type `falcon.core.Phase` is instantiated by the

problem using the method `falcon.Problem.addNewPhase()` instead of a standalone instantiation. This principle is used throughout *FALCON.m* as many user caused problem definition errors can be avoided automatically.

Data Storage

As introduced in chapter 2, all variables in the optimization vector and the constraint vector have a lower and upper bound. Additionally, the gradient matrix must be scaled in order to make it numerically well formed. Betts [24] suggests that a scaling for each state, control, etc. is defined. These scalings are used to scale the overall derivatives of the optimal control problem.

In *FALCON.m*, classes store this information for states (`falcon.State`), controls (`falcon.Control`), parameters (`falcon.Parameter`), and for path / point constraints (`falcon.Constraint`). Their constructor has the following interface

```
state = falcon.State('Name', LowerBound, UpperBound, Scaling, Offset)
control = falcon.Control('Name', LowerBound, UpperBound, ..)
constraint = falcon.Constraint('Name', LowerBound, UpperBound, ..)
```

setting the following properties:

Name All optimization variables are identified using their names in *FALCON.m*. Two class instances of e.g. `falcon.State` with equal names will be identified as the same variable. Therefore, in each array of variables (e.g. state array), all names must be unique.

Lower and Upper Bound Unscaled lower and upper bound defining the box constraint:

$$x_{LB} \leq x \leq x_{UB}. \tag{4.11}$$

Scaling and Offset Used to bring the optimal control problem in a scaled state. The scaling S and offset r are applied to value and bounds

$$\tilde{x} = (x - r_x) \cdot S_x \tag{4.12}$$
$$\tilde{x}_{LB} = (x_{LB} - r_x) \cdot S_x \tag{4.13}$$
$$\tilde{x}_{UB} = (x_{UB} - r_x) \cdot S_x \tag{4.14}$$

resulting in their scaled version denoted by a $\tilde{\square}$.

Each parameter in *FALCON.m* refers to a scalar variable. Compared to the other classes above, the parameter class holds the actual optimization variable directly. Therefore, the interface of the constructor

```
param = falcon.Parameter('Name', Value, LowerBound, UpperBound,
   Scaling, Offset)
```

requires an initial guess for the parameter value. Parameters and all other classes presented above can be used multiple times in different parts of a *FALCON.m* optimal control problem. As an example, the initial and final times of a phase are implemented by the parameter class. In a multi-phase problem, the final time of the previous phase is reused as the initial time of the following connected phase. Due to the fact that parameters in *FALCON.m* store the optimization value directly, multiple `falcon.Parameter` instances with the same name are not allowed. Otherwise, an unique identification of parameters is not possible.

Discretization Density and Data Sorting

Time history data (e.g. of states) is not stored in the corresponding class but in time grids. As mentioned in chapter 2, all data is stored w.r.t. normalized time. Each grid has two main properties, the datatype (e.g. `falcon.State`) and a normalized time discretization. Thus, a matrix is spanned, where the number of rows represent the datatype (e.g. states) and the number of columns the discretized time steps. All grids are created by *FALCON.m* automatically.

The main grid is the so-called `StateGrid` which holds the discretization of states. It implements the "smallest common denominator" of the discretized time of all other grids. Controls and constraints may be defined on a more sparse grid as the state grid. Additionally, multiple control grids can be defined each having a unique discretization (see Figure 4.1). For simulation, missing values in control grids are interpolated (linear, previous). *FALCON.m* automatically uses the state grid discretization in case no normalized time is provided by the user. All discretized states and controls as well as the constraints are sorted w.r.t. their normalized time. Thus, the general structure of the OCP derivatives resembles diagonal matrices.

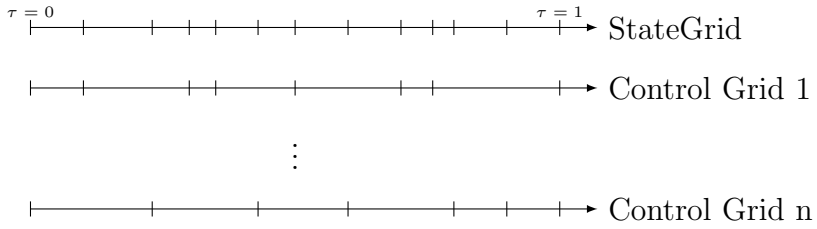


Figure 4.1: Different discretization densities of stategrid and control grids.

User Function Derivatives

FALCON.m uses gradient based optimizers and thus requires the derivatives (preferably analytic) of all user functions. Since manually calculating the first and second order derivatives may be very tiresome and error-prone, automatic methods are provided. In normal operation, the derivative calculation and handling is not visible to the user.

FALCON.m expects all user functions to return their local derivatives. Otherwise they cannot be used in *FALCON.m*. All user functions have to undergo a preparation step that generates their derivatives. This step has to be done once. Afterwards, they are referred to as differentiated or prepared. There are multiple ways to achieve the preparation, allowing a user to have much control over the derivative generation process.

In the following sections, for simplicity, it is assumed that all user functions have been prepared and thus implement the derivatives. An explanation on the derivative generation from the user side is given in section 4.3. The behind the scene derivative generation toolchain is described in section 4.5.

4.2.2 Problem

As mentioned above, the problem class (`falcon.Problem`) is the main class of the optimal control problem implementation. Figure 4.2 displays the problem class with important properties and methods. It holds all phases of the optimal control problem as well as information that references information from multiple phases. Therefore, point constraints and Mayer cost functions are found here. Additionally, a list of all optimizable parameters of the optimal control problem exists. This is important as a single parameter may be used in different phases, models, or constraints.

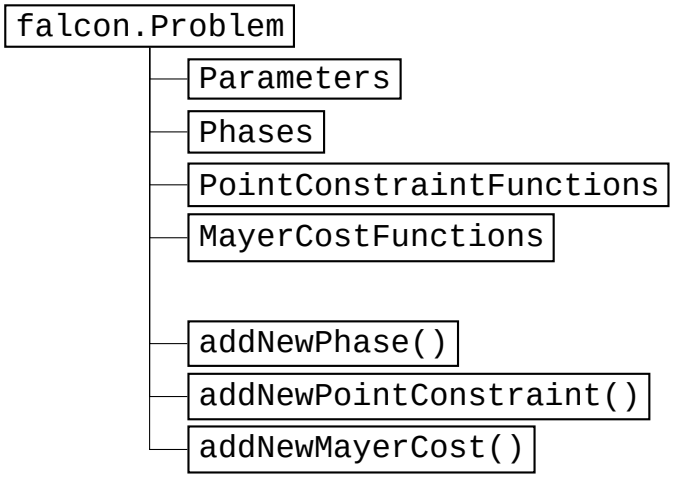


Figure 4.2: *FALCON.m* problem class important properties and methods.

4.2.3 Phases

The `falcon.core.Phase` class holds all phase relevant information such as the state grid, the control grids, the simulation model, the path constraints, and Lagrange cost functions (see Figure 4.3).

A new phase is added to the *FALCON.m* problem with the method

```
phase = problem.addNewPhase(@modelhandle, States, Tau, StartTime,
   FinalTime)
```

where a function handle to the model dynamics, a vector of state objects $\vec{x}$, the normalized time array τ, the start time t_0, and the final time t_f are required. *FALCON.m* automatically creates a `StateGrid` from this information. The real time

$$t = \tau \cdot (t_f - t_0) + t_0 \tag{4.15}$$

is available as a property in the phase. In case the phase model requires controls, a new control grid can be added

```
controlgrid = phase.addNewControlGrid(Controls, Tau)
```

specifying the controls $\vec{u}$ and the normalized time τ. The method may be called multiple times in order to have multiple control grids. The interpolation method for the control can be set using the method:

```
controlgrid.setInterpolationMethod(method).
```

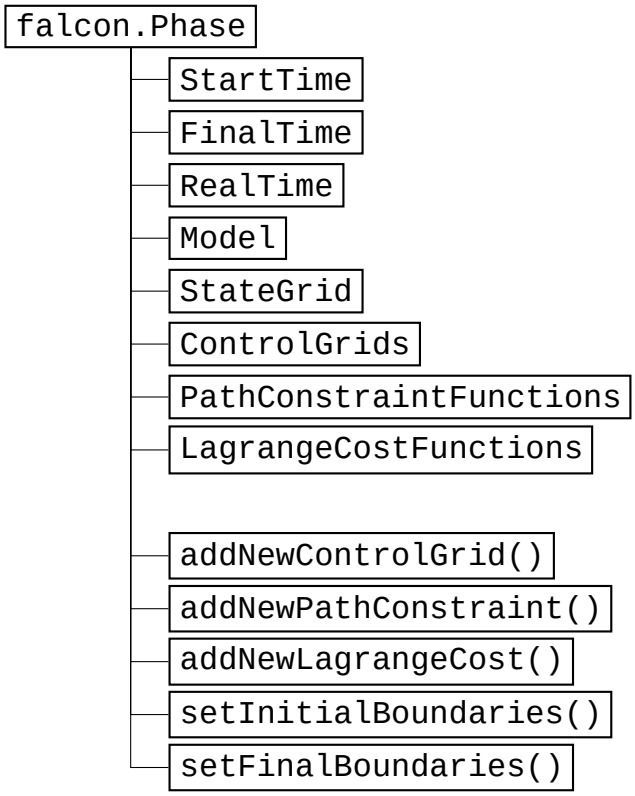


Figure 4.3: *FALCON.m* phase class important properties and methods.

Model

With the `addNewPhase` method, *FALCON.m* creates a model instance in the background. It is available as the `Model` property in the phase and calls the model dynamics (prepared user function). The results are stored in the `StatesDotGrid` and in the `ModelOutputGrid` (see Figure 4.4).

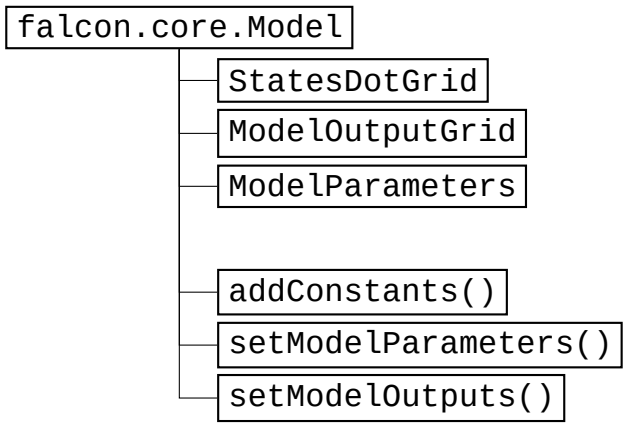


Figure 4.4: *FALCON.m* model class properties and methods.

If the model implements parameters or has additional constants, they can be set using the methods `setModelParameters` and `addConstants` respectively. The model outputs are limited by setting constraints (`falcon.Constraint`) for each output with the `setModelOutputs` method.

Boundary Conditions and Duration

For each phase, the initial and final boundary conditions can be set using the methods `setInitialBoundaries` and `setFinalBoundaries` respectively. There are several ways the bound may be set

```
phase.setInitialBoundaries(equalbound)
phase.setInitialBoundaries(lowerbound, upperbound)
phase.setInitialBoundaries(states, equalbound);
phase.setInitialBoundaries(states, lowerbound, upperbound);
```

allowing for the formulation of equality or inequality conditions. Bounds may be set for specific states by passing the relevant state objects to the method. States without a boundary condition keep their state object boundaries. The set methods may be called multiple times for an individual setting of the boundaries.

Apart from the initial and final boundaries, the duration Δt of the phase is limited. The bounds

$$0 \leq \Delta t_{LB} \leq t_f - t_0 \leq \Delta t_{UB} \tag{4.16}$$

are set using the `setDurationLimit` method. Additionally to the bounds, a scaling and an offset may be defined. By default, the duration must be positive $0 \leq t_f - t_0$.

In multi phase optimal control problems, a phase defect is required for continuity in the state history. By default, all phases are independent, meaning no phase defects exist. In *FALCON.m*, two phases are linked by setting the `ConnectedPhase` property to the next phase. This is achieved by the phase method

```
phase1.ConnectToNextPhase(phase2)
```

which requires the next phase as an input parameter. Using the problem method

```
problem.ConnectAllPhases()
```

all phases can be connected automatically.

Integration Methods

In *FALCON.m*, the collocation discretization scheme is implemented. Therefore, the collocation defects (2.121) must be implemented as constraints. *FALCON.m* implements different discretization schemes, namely trapezoidal, backward Euler, and others as classes in the `falcon.discretization` namespace. The default discretization is trapezoidal. Another discretization method is chosen by providing the class instance to `setDiscretizationMethod`. During optimization, the defect values are stored in the `DefectGrid` of the phase. At the end of a successful optimization, this grid contains only zero values within the feasibility tolerance.

4.2.4 Constraints and Cost Functions

The quality of the solution of the optimal control problem is driven by imposing realistic constraints. A simple way to set constraints is by setting the bounds of the states, controls, and parameters. Additional model outputs can be limited in a similar way.

FALCON.m allows the definition of nonlinear path and point constraints (see Figure 4.5). Path constraints formulate limits that need to be met at every discretized point in

time (similar to output limits). A single constraint between any multiple points in the OCP can be achieved by a point constraint. The latter allows for extremely complex constraint formulation.

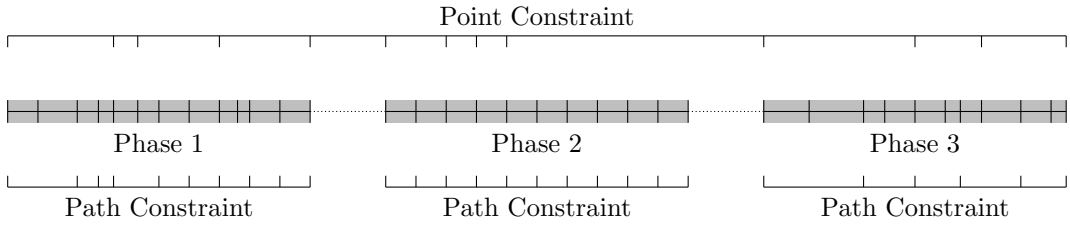


Figure 4.5: Difference between path and point constraints.

Only the variables required for the evaluation enter the user function that calculates the constraints. This reduces the complexity for the user and improves the overall performance in the gradient calculation. In order to achieve this, the constraints store which data they require and communicate this information to *FALCON.m*. Therefore, in the derivative generation step explained in section 4.5, the information required by the constraint is stored in the prepared file that implements the derivatives. This information is communicated to *FALCON.m* using a `struct` interface described in 4.4.2.

The main benefit of this approach is that the constraint formulation becomes independent of the model formulation. Due to the fact that constraints communicate the data they require, the correct information will always be provided by *FALCON.m*. In case an additional state is added to the model or the order of the states is changed, the constraint does not need to be adapted.

In the following, important aspects of the path and point constraints are discussed. Additionally, Mayer and Lagrange cost functions that are based on both constraint types are explained.

Path Constraint

A path constraint is evaluated on the time discretization of the phase but calculates its constraint values w.r.t. a single time step. As mentioned above, only information required by the path constraint user function enters it, thus defining a subset of the phase variables. Therefore, the inputs are the subset outputs $\vec{y}_c \subseteq \vec{y}$, the subsets states $\vec{x}_c \subseteq \vec{x}$, and the subset controls $\vec{u}_c \subseteq \vec{u}$. The parameters of a constraint $\vec{p}_c$ are independent on the model parameters $\vec{p}$, but may have some values in common. Thus, the path constraint interface is given by

$$\vec{g}\left(\vec{y}_c, \vec{x}_c, \vec{u}_c, \vec{p}_c, \vec{c}, \dots\right) \tag{4.17}$$

where $\vec{c}$ represents an arbitrary number of constant inputs. Data that is not required by the path constraint does not appear in the interface (e.g. no parameters $\vec{p}_c = \emptyset$ or controls $\vec{u}_c = \emptyset$). The path constraint is added to the phase using

```
pathcon = phase.addNewPathConstraint(@pathhandle, constraints, tau)
```

where a handle to the prepared path constraint function is given. Additionally, the constraint objects specifying the bounds and the normalized time on which the path constraint is evaluated must be provided.

Point Constraint

A point constraint combines information at different times and phases of an optimal control problem into a single value. As with the path constraints, the prepared user function implementing the point constraint communicates to *FALCON.m* which data it requires. However, in this constraint, values from different phases have to be obtained. Each participating phase k provides a set of model outputs $\vec{y}_k$, states $\vec{x}_k$, and controls $\vec{u}_k$. For each phase, the set of required data by the point constraint may be different. A custom input set is defined for every phase. During run-time, this data is requested from the optimal control problem by the prepared point function. Thus, the requested data from for instance two phases k_1 and k_2 is

$$\vec{y}_{c,k_1} \subseteq \vec{y}_{k,1}, \qquad \vec{x}_{c,k_1} \subseteq \vec{x}_{k,1}, \qquad \vec{u}_{c,k_1} \subseteq \vec{u}_{k,1} \tag{4.18}$$

$$\vec{y}_{c,k_2} \subseteq \vec{y}_{k,2}, \qquad \vec{x}_{c,k_2} \subseteq \vec{x}_{k,2}, \qquad \vec{u}_{c,k_2} \subseteq \vec{u}_{k,2} \tag{4.19}$$

whereas some of the subsets may be empty (e.g. $\vec{x}_{c,k_2} = \emptyset$). Empty sets do not appear in the function interface.

Each phase input may have multiple time instances. Therefore, each set of inputs $\vec{y}_{c,k}, \vec{x}_{c,k}, \vec{u}_{c,k}$ represents a matrix with n_h columns. Additionally to the phase input sets, parameters and an arbitrary number of constants enter the point constraint. Thus, the possible function interface becomes

$$\vec{h}\left(\vec{y}_{c,k_1}, \vec{x}_{c,k_1}, \vec{u}_{c,k_1}, \vec{y}_{c,k_2}, \vec{x}_{c,k_2}, \vec{u}_{c,k_2}, \ldots, \vec{p}_c, \vec{c}, \ldots\right), \tag{4.20}$$

which allows for a highly flexible problem formulation without having to compromise on the performance.

Similar to the path constraint, the point constraint communicates to *FALCON.m* which data it requires. Within the point constraint, the phase input sets (variable types and number of time steps), the required parameters, and constants are stored. However, it is not defined which phase and actual time steps the data originates from. Therefore, when a point constraint is added to the problem

```
pointcon = problem.addNewPointConstraint(@pointhandle,...
    constraints, phase_1, tau_1, phase_2, tau_2, ..)
```

the phase and time discretizations must be provided. Splitting the time information from the function evaluation enables more flexibility and re-usability of point functions.

Cost Functions

There are two types of cost functions that can be implemented in a *FALCON.m* optimal control problem, namely Mayer and Lagrange cost functions.

Regarding the implementation, the Lagrange cost function and the path constraints have similar features. Therefore, the methods to prepare path functions can be used to create a function suitable for a Lagrange cost function as well. The same holds for the Mayer cost function which resembles the point constraint definition.

Lagrange cost functions are phase dependent and are thus created by the phase

```
phase.addNewLagrangeCost(@lagrangehandle, cost, tau).
```

They have a similar interface to the path constraint. The integration for the Lagrange cost functions is carried out using the trapezoidal approach. The Mayer cost function may span over multiple phases and is thus created by the problem

```
problem.addNewMayerCost(@mayerhandle, cost,...
phase_1, tau_1, phase_2, tau_2, ..).
```

The method implements a similar interface to the point constraint. Instead of constraints, the cost functions require cost objects:

```
cost = falcon.Cost(Name, Scaling, Offset).
```

Since the cost function is unbounded, these classes implement the scaling and offset functionality only.

4.2.5 Solving Problems

Once a problem is fully defined, it can be solved. The solution process consists of three steps: building the problem, calling the nonlinear solver, and evaluating the Function Generator. In the following, these steps are explained in more detail. By default, a user is just required to call the `problem.Solve` method, which carries out all relevant steps automatically.

Bake / UnBake

Before the problem can be solved, the optimal control problem formulation must be transformed into a parameter optimization problem. This is achieved by the `Bake` method in the `falcon.Problem` class. After its evaluation, the problem can no longer be changed. Any attempts to do so will cause a run-time error.

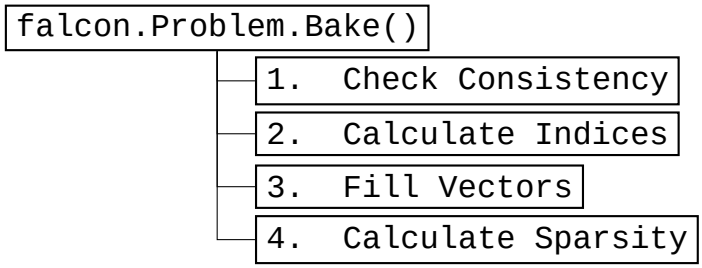


Figure 4.6: *FALCON.m* optimal control problem `Bake` process.

Figure 4.6 shows the principal baking process. It starts with a problem consistency check that shall determine if the problem formulation is valid. These checks include the availability of user functions and data, as well as dimension checks. During run-time, no further checks are required. Thus, performance and code readability are improved.

The Calculate Indices step is carried out after the consistency checks. Since the optimal control problem is discretized into a parameter optimization problem, all optimization variables are stored in a vector $\vec{z}$ and all user function dependent constraints are stored in a vector $\vec{f}$. In order to have simple read and write access to the vectors, indexing matrices are created.

Once the indices are calculated, in the third step, the vectors

$$\vec{z}_{ini}, \vec{z}_{LB}, \vec{z}_{UB}, S_z, \vec{r}_z, \qquad \vec{f}_{LB}, \vec{f}_{UB}, S_f, \vec{r}_f \tag{4.21}$$

are filled with data. The vector $\vec{z}_{ini}$ represents the initial guess provided to the optimizer. Additionally, the lower / upper bounds $\vec{z}_{LB}, \vec{z}_{UB}, \vec{f}_{LB}, \vec{f}_{UB}$, the diagonal scaling matrices S_z, S_f, and the offset vectors $\vec{r}_z, \vec{r}_f$ are created. As the constraint vector $\vec{f}$ is dependent on $\vec{z}$, it requires no initial vector. All data is taken from the variable classes described in section 4.2.1. Before the optimizer is called, the vectors have to be scaled

$$\tilde{\vec{z}}_{ini} = S_z \cdot (\vec{z}_{ini} - \vec{r}_z), \qquad \tilde{\vec{z}}_{LB} = S_z \cdot (\vec{z}_{LB} - \vec{r}_z), \qquad \tilde{\vec{z}}_{UB} = S_z \cdot (\vec{z}_{UB} - \vec{r}_z), \tag{4.22}$$

$$\tilde{\vec{f}}_{LB} = S_f \cdot \left(\vec{f}_{LB} - \vec{r}_f\right), \qquad \tilde{\vec{f}}_{UB} = S_f \cdot \left(\vec{f}_{UB} - \vec{r}_f\right), \tag{4.23}$$

where $\tilde{\square}$ denotes the scaled vectors.

In the last step of the building process, the sparsity structure of the Jacobian and optionally of the Hessian is calculated. This step is crucial for the overall optimization performance especially for large derivative matrices where memory consumption has a huge impact. The idea behind the sparsity calculation is explained in section 4.6.

Solvers

FALCON.m uses "off-the-shelf" NLP optimizers to solve the optimal control problems. As every solver has a unique interface, a custom interface wrapper is required that communicates between both is required. At the moment, interfaces for three solvers IPOPT [40], SNOPT [41], and WORHP [42] are implemented. In the following, the *FALCON.m* derivative interface is explained. Thus, additional optimizers can be interfaced.

The Function Generator that evaluates the optimal control problem for a current scaled optimization state $\tilde{\vec{z}}$ is implemented in the `DiscretizationMethod` class. This class is found as property of the `falcon.Problem` instance and has two methods called `OptiFunc` and `OptiFuncHess`. The first method calculates the scaled cost value $\tilde{J}$, the scaled constraint vector $\tilde{\vec{f}}$, and the scaled Jacobian of both w.r.t. the given $\tilde{\vec{z}}$ vector. The function has two return arguments

$$\tilde{\vec{f}}_J = \begin{bmatrix} \tilde{J} \\ \tilde{\vec{f}} \end{bmatrix}, \qquad \nabla_{\tilde{z}} \tilde{\vec{f}}_J = \frac{\partial \tilde{\vec{f}}_J}{\partial \tilde{\vec{z}}} = \begin{bmatrix} \frac{\partial \tilde{J}}{\partial \tilde{\vec{z}}} \\ \\ \frac{\partial \tilde{\vec{f}}}{\partial \tilde{\vec{z}}} \end{bmatrix} \tag{4.24}$$

where the cost value $\tilde{J}$ and the constraint vector $\tilde{\vec{f}}$ are stacked to create the combined vector $\tilde{\vec{f}}_J$. The non-zero elements of the Jacobian $\nabla_{\tilde{z}} \tilde{\vec{f}}_J$ are returned as a column vector. The assignment of the non-zero values is found in the column vectors `problem.iGfun` and `problem.jGvar` whic represent non-zero row and column indices respectively. Thus, the interface of the function `OptiFunc` is the following:

```
[F_scaled, dFdZ_scaled_vec] =...
    problem.DiscretizationMethod.OptiFunc(z_scaled).
```

The function `OptiFuncHess` calculates the scaled Hessian of the Lagrangian

$$\mathcal{L} = l_0 \cdot \tilde{J}(\tilde{\vec{z}}) + \sum_i \lambda_i \cdot \tilde{f}_i \tag{4.25}$$

which is used in e.g. IPOPT to calculate a descent direction with the Newton method. Input argument to the function are the current multipliers for the cost function l_0 and constraints $\vec{\lambda}$ stacked in a single vector:

$$\vec{\sigma} = \begin{bmatrix} l_0 \\ \vec{\lambda} \end{bmatrix}. \tag{4.26}$$

Thus, the interface of the function `OptiFuncHess` is the following:

```
[H_scaled] =
   problem.DiscretizationMethod.OptiFuncHess(sigma_scaled).
```

Please note that `OptiFuncHess` does not expect a current $\tilde{\vec{z}}$ vector. During the evaluation of the `OptiFunc` method, the user functions return their local Hessians. This information is stored and extracted in `OptiFuncHess`. Therefore, `OptiFunc` must always be called beforehand. As with the Jacobian, the Hessian is returned as a vector containing the non-zero elements only. The assignment of the non-zero values is found in the vectors `problem.iHvar` and `problem.jHvar` representing non-zero row and column indices respectively.

4.2.6 Usability Features

Apart from the basic implementation of the optimal control toolbox, *FALCON.m* offers some unique features that enable a wide range of applications and flexibility. Additionally, the usability is enhanced especially for users who are unfamiliar with the toolbox.

Control Fixing and Constraint Deactivation

In *FALCON.m*, controls and parameters can be fixed so that they can no longer be optimized. Additionally, constraints can be deactivated. These options can either be set in the constructor or by using the appropriate set method:

```
control = falcon.Control(Name, 'Fixed', true);
control.setFixed(true);

constraint = falcon.Constraint(Name, LowerBound, UpperBound,
   'Active', false);
constraint.setActive(false);
```

Thus, it is possible to assess the impact of a constraint on the solution without having to adapt the constraints in the source code. Besides, fixing the controls enables the utilization of FALCON.m for parameter identification purposes [81]. Fixed controls and parameters are automatically removed from the $\vec{z}$ vector. Inactive constraints no longer appear in the constraint vector $\vec{f}$. The calculation of the derivatives is adapted accordingly.

Listing 4.1: Invoke automatic function interface creation PT1.

```
x  = falcon.State('value');
u  = falcon.Control('cmd');

problem = falcon.Problem('PT1');
phase = problem.addNewPhase(@dynmodel, x, 101, 0, 10);
phase.addNewControlGrid(u);

problem.Bake();
```

Listing 4.2: Automatic function interface of PT1.

```
function [states_dot] = dynmodel(states, controls)
% model interface created by falcon.m

% Extract states
value = states(1);

% Extract controls
cmd   = controls(1);

% ------------------------ %
% implement the model here %
% ------------------------ %

% implement state derivatives here
value_dot = ;
states_dot = [value_dot];

end
```

Automatic Interface Generation

For problem setup and structure definition in *FALCON.m*, usually a *MATLAB* script is used. Additionally, the user needs to provide *MATLAB* functions which implement the model dynamics, the constraints, and the cost functions. In order to simplify the creation of these user functions, *FALCON.m* creates templates based on the problem definition.

In order to use this feature, e.g. for model interface generation, instead of passing a handle to a prepared model function, a handle to a non-existent function is passed to the `addNewPhase` method (see Listing 4.1). In the `Bake` method, *FALCON.m* automatically detects that the function is not available and prompts a message to the user whether an interface shall be auto-generated. *FALCON.m* creates a function template for the model interface (see Listing 4.2) in the current working directory. This feature works in the same way for path / point constraints and thus for cost functions as well.

Automatic Initial Guess Creation

Any iterative optimization algorithm requires a starting point, called an initial guess, from which it starts its iterations. For the discretized optimal control problem a guess must be provided as well. The *FALCON.m* optimal control toolbox offers a simple automatic initial guess generation. It uses the boundary conditions and bounds of the states and controls to approximate suitable values for an initial guess. This algorithm requires no information regarding the model dynamics. Although in many cases a suitable initial guess is obtained, there is not guarantee that it leads to an optimal solution. Additionally, initial guesses for parameters must always be provided.

The algorithm works as follows: In case a boundary condition for a state exists, the mean value is taken respectively. Otherwise, the lower and upper bounds of the states are used to calculate a mean value. If a lower or upper bound is infinite, the non-infinite bound is used. If both bounds are infinite, the state is initialized with zero. The initial and final state guesses are used for a linear interpolation over time. For the controls, no boundary conditions exists. However, the lower and upper bounds are used to calculate a constant mean over time.

Post Processing

After a successful optimization, it is possible to apply post processing filters to the results. The post processing is designed for measurements and calculations that are required for plotting or further analysis, but not during the optimization. Thus, the calculation overhead is reduced. No derivatives are calculated for the post processing values.

A post processing step is added with the method

```
problem.addPostProcessingStep(funcHandle, inargscell, calcValues)
```

where `funcHandle` is a function handle, `inargscell` an input argument cell array using the *FALCON.m* data objects, and `calcValues` represents an array of value objects (`falcon.Value`) of suitable size for the return values. `falcon.Value` outputs of a post processing step may be used as inputs in a successive step.

The post processing step calculation is applied to all phases automatically. Using the data objects (e.g. `falcon.State`) the relevant data is extracted from the phases. In case data is not available in the phase it is replaced by `nan` (not a number) values. All post processing steps can be removed from a problem using the `problem.clearPostProcessing` method. The results are stored in the phase property `PostProcessingGrid`.

Data Plotting and Export

A visual representation helps to evaluate the quality of a solution obtained by the optimization. In *FALCON.m* a plotting tool is integrated which enables a user to create custom plots. It can be opened using the `problem.PlotGUI`. Further information can be found in the documentation [80].

Apart from plotting features, it is possible to export the data in two ways: The `problem.ToStruct` method creates a struct representation from the problem instance.

This struct can for instance be used in the plotting tool. Additionally, a time series representation of the results can be obtained with the `getTimeSeries` method. It automatically concatenates all phases in order of appearance. The resulting time series object may be used in Simulink for further analysis.

4.3 User Function Derivatives

FALCON.m uses gradient based optimization algorithms to solve the optimal control problems. Therefore, the first order and sometimes the second order derivatives of the user functions are required to build the overall problem derivatives. As these derivatives shall not be provided by the user, automatic methods have to be applied.

All user functions are preprocessed to include analytic derivatives. *FALCON.m* utilizes the *MATLAB*'s Symbolic Math Toolbox for the differentiation requiring no or little user input. Since the model and path constraints are evaluated at multiple points in time, user functions and their derivatives are compiled into MEX files using *MATLAB* Coder. This enables very fast evaluation that supports multi-threading.

In case either the Symbolic Math Toolbox or the *MATLAB* Coder are not available, *FALCON.m* automatically switches to a compatibility mode. Then, finite forward differences or evaluations in *MATLAB* are used respectively. However, these modes are not discussed in this thesis.

All models / constraints and cost functions are created using builder classes in *FALCON.m*:

falcon.SimulationModelBuilder for dynamic models

falcon.PathConstraintBuilder for path constraints and Lagrange cost functions

falcon.PointConstraintBuilder for point constraints and Mayer cost functions

In this section, the derivative calculation is explained from the user's point of view. The implementation is described in section 4.5. Derivatives for all user functions may be generated with one of the following two modes:

Function Mode Calculates the derivatives of a single *MATLAB* source function for the dynamic model, constraint, or cost function. It is the preferred way as the actual user implementation is very simple. However, dependent on the complexity of the source function at hand, the calculation of the derivatives may fail. This is mainly due to computational limits of *MATLAB*'s symbolic math engine. In this case, the subsystem mode must be used.

Subsystem Mode Is used if the Symbolic Math Toolbox cannot calculate the analytic derivatives for a whole user function directly. This situation occurs mainly in the following (but not exclusive) situations:

- multiple matrix vector multiplications
- multiple high order polynomials
- non-continuities such as lookup tables
- derivatives contain imaginary numbers

There is no clear rule when the symbolic differentiation fails. However, difficulties in the source code transformation are recognizable by the required time (e.g. longer than 30 seconds). In this case, the user function can usually be split into smaller subsystems, which can be differentiated locally (see Figure 4.7), giving the subsystem mode its name. The local derivatives are automatically recombined via the chain rule to include the overall user function derivatives.

In the following, both modes are explained in more detail.

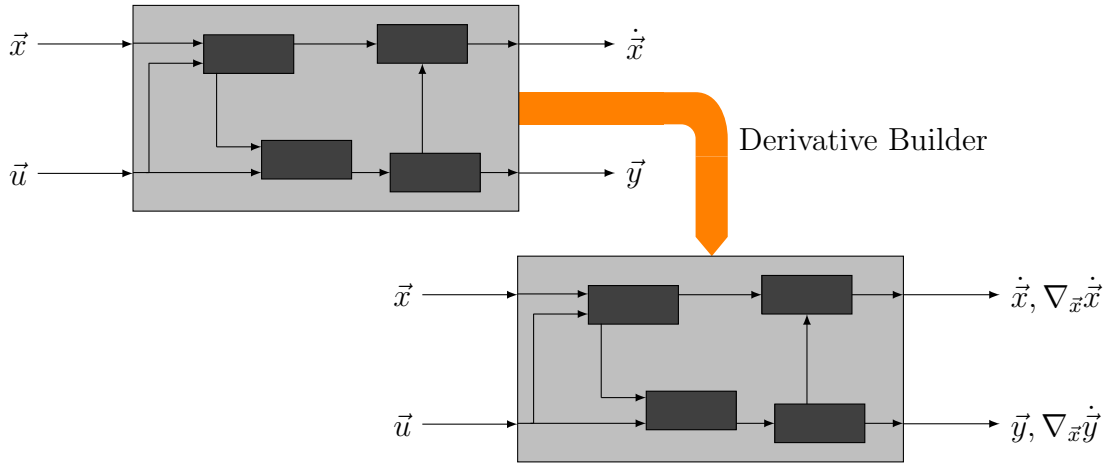


Figure 4.7: Subsystems of a user model function.

4.3.1 Function Mode

The function mode is invoked by passing the function handle to the builder in the constructor. In this case, the builder requires the interface of the input arguments and output arguments. From this information, symbolic variables are created. These are used to call the user function. Thus, the function is available as a symbolic representation in the *MATLAB* workspace. The derivatives are generated and written to *MATLAB* code. The differentiated function is compiled to *MATLAB* MEX file for fast evaluation.

In section 4.2.6 the automatic interface generation is presented. *FALCON.m* creates an interface for a not yet existing user function. Once it is implemented by the user, *FALCON.m* is linked to the source user function which does not implement the derivatives. In this case, *FALCON.m* automatically uses the function mode derivative generation in the background. The resulting file is stored in the current working directory using the name of the function handle preceded by `fm_mex`.

4.3.2 Subsystem Mode

More complicated user functions cannot be differentiated by the symbolic math toolbox directly. Therefore, *FALCON.m* offers the subsystem mode for the derivative generation. The basic idea is that the user splits the dynamic model, constraint, or cost function into simple subsystems which are implemented as *MATLAB* functions. *FALCON.m* automatically creates the derivatives for each and calculates the derivatives for the overall user function by applying the chain rule. In the following, the principles of the subsystem mode and the method provided by the builder classes are described.

Listing 4.3: Minimal subsystem mode example.

```
states   = [falcon.State('x'), falcon.State('y')];
controls = [falcon.Control('V'), falcon.Control('alpha')];

mdl = falcon.SimulationModelBuilder('myModel', states, controls);
mdl.addSubsystem(@myfunc,...          % Subsystem Function Handle
    {'x', 'y', 'V', 'alpha'},...      % Inputs  to Subsystem
    {'xdot', 'ydot'})                 % Outputs of Subsystem
mdl.setStateDerivativeNames({'xdot', 'ydot'});
mdl.Build();
```

For simplicity, the explanation will be given with an application to a dynamic model in mind as it is usually the most complex part of the derivative generation. All principles and methods described can be transfered to constraints and cost functions too.

Principles

Apart from the subsystems themselves, the user needs to define how these are connected. In *FALCON.m* every signal / variable is represented by a unique string. For every variable available in the model, *FALCON.m* stores its size (number of rows and columns).

In Listing 4.3, the states and controls are defined by an array of data objects which are passed to the constructor of the builder instance. Additionally, the name of the final prepared derivative function is given. In the example the name is `myModel`.

Within the builder, all states and controls will be registered as individual scalar variables. Then, an arbitrary number of subsystems can be added to the model. For every subsystem, the source function (first argument), input arguments (second argument), and the output arguments (third argument) must be specified. Input and output arguments of a subsystem are specified using cell arrays of strings. The method `setStateDerivativeNames` sets the variable names that hold the state derivative information. The order must match the states specified in the constructor. Finally, `Build` method invokes the derivative generation process.

In the example above only states and controls enter the model. Additional inputs may be a set of parameters or a series of constants.

Constants

There are three ways how constants can be used in the subsystem mode. As a reminder, no derivatives are calculated w.r.t. constants. In the following, `mdl` represents an arbitrary derivative builder instance.

addConstantInput This method

```
mdl.addConstantInput('name', [m,n])
```

expects the name of the input as well as its size. This way, an additional, constant input is added to the model. This method is also available in the function mode.

addConstant Adds an internal constant to the list of variables that cannot be altered after the construction of the model. The method

```
mdl.addConstant('name', value)
```

requires the name of the constant as a string as well as its value. This approach may be used in case a constant is used multiple times within the model dynamics.

Numeric Value Apart from strings, subsystem inputs can also be numeric variables (see Subsystems section below). This approach may be used instead of the constant that is represented by a string. In case it contains many zero elements, the analytic derivative generation can exploit the structure resulting in faster evaluations during run-time. The derivatives must be recalculated in case the numeric constant is changed.

Subsystems

Subsystems are added to a model using the

```
mdl.addSubsystem(subsystem, inputs, outputs)
```

method. All subsystems are evaluated in the order they are added to the builder. Therefore, it is crucial that all required input data is available. *FALCON.m* checks the feasibility of the subsystem call and throws an error if conditions are not met. A subsystem can be of the following types:

function A function handle to a *MATLAB* function. This is the simplest and recommended option.

matlab.System An instance of a `matlab.System` class. The main benefit of this subsystem is that it can be re-used in the Simulink environment.

anonymous function Ideal for small computations that fit in a single line of code (e.g. a summation).

MATLAB builtin functions, nested functions or local functions (e.g. below class definition) are not supported. However, they can always be included by calling them from another supported subsystem type (e.g. anonymous function).

During the build process *FALCON.m* creates the derivatives of the source functions. For every subsystem source function a hash value is generated that is stored if the differentiation of the subsystem was successful. Thus, already differentiated subsystems are skipped. This speeds up the derivative generation process if a small change was made and a reconstruction is necessary. The hash value is calculated only for the top-level function, meaning any changes in called subfunctions are not detected. In order to force a new generation of all derivatives, the `fm_models` and `fm_constraints` folders in the current working directory can be deleted.

Inputs to the subsystem are defined as a cell array of strings where each entry represents an input. A cell array with one entry expects the subsystem to have a single input. The number of inputs is not limited. Each input is identified by the string name. Additionally, an input entry may be a numeric value. In case all inputs are constants or

numeric values, *FALCON.m* will not create derivatives. The outputs of the subsystem will be regarded as constant as well.

Similar to the inputs, the outputs are defined as a cell array of strings. For each output, a name has to be defined. The output sizes are automatically determined. All outputs are added to a list of variables currently available and can thus be used in following subsystem calls. An output can be omitted by setting a tilde (`'~'`) as the output string. Listings 4.4 shows some example subsystem calls.

Listing 4.4: Input and output examples of a subsystem call.

```
mdl.addSubsystem(@system, {'in1'}, {'out2'})
mdl.addSubsystem(@system, {'in1', value, 'in3'}, {'~', 'out2'})
```

Additionally to the derivatives, *FALCON.m* calculates the sparsity patterns of the outputs Jacobian/Hessian w.r.t. the subsystem inputs. Constant inputs are not considered in the pattern. Let

$$f(x,y,z) = \begin{bmatrix} x^2 + \cos(z) \\ \exp(x) + y \end{bmatrix} \tag{4.27}$$

be an exemplary subsystem function. If all inputs (x, y, z) are derived from optimizable model inputs, the Jacobian and Hessian

$$\frac{\partial f}{\partial [x,y,z]} = \begin{bmatrix} 2\cdot x & 0 & -\sin(z) \\ \exp(x) & 1 & 0 \end{bmatrix}, \quad \frac{\partial^2 f}{\partial [x,y,z]^2} = \begin{bmatrix} 2 & 0 & -\cos(z) \\ 0 & 0 & 0 \\ -\cos(z) & 0 & 0 \\ \exp(x) & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4.28}$$

as well as their sparsity patterns

$$\left\{ \frac{\partial f}{\partial [x,y,z]} \neq 0 \right\} = \begin{Bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{Bmatrix}, \quad \left\{ \frac{\partial^2 f}{\partial [x,y,z]^2} \neq 0 \right\} = \begin{Bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{Bmatrix} \tag{4.29}$$

can be calculated. In case y represents a constant, the calculated derivatives and thus the sparsity patterns are reduced:

$$\frac{\partial f}{\partial [x,z]} = \begin{bmatrix} 2\cdot x & -\sin(z) \\ \exp(x) & 0 \end{bmatrix}, \qquad \left\{ \frac{\partial f}{\partial [x,z]} \neq 0 \right\} = \begin{Bmatrix} 1 & 1 \\ 1 & 0 \end{Bmatrix}, \tag{4.30}$$

$$\frac{\partial^2 f}{\partial [x,z]^2} = \begin{bmatrix} 2 & -\cos(z) \\ -\cos(z) & 0 \\ \exp(x) & 0 \\ 0 & 0 \end{bmatrix}, \qquad \left\{ \frac{\partial^2 f}{\partial [x,z]^2} \neq 0 \right\} = \begin{Bmatrix} 1 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 0 \end{Bmatrix}. \tag{4.31}$$

Similar to the calculation of the overall model derivatives from the local ones, the local sparsity patterns are used to calculate the overall sparsity pattern of the model output arguments w.r.t. the model input arguments. The sparsity pattern is used to calculate the sparsity pattern of the overall optimal control problem. The structure of the Jacobian and Hessian is explained in more detail in section 4.5.

In order to calculate analytic derivatives of the subsystems, all calculations have to be supported by *MATLAB*'s symbolic math engine. Thus, conditions and loops are generally not supported. The optimal control theory states that all functions must be continuously differentiable. In case a function is defined piecewise or a look-up-table is used it may be continuously differentiable but the symbolic math engine is unable to handle the differentiation. The issue can be resolved by using a derivative subsystem.

Derivative Subsystems

Derivative Subsystems are used in case a subsystem cannot be differentiated analytically by the symbolic math engine (e.g. table data, piecewise defined functions). By using the

```
mdl.addDerivativeSubsystem(subsystem, inputs, outputs)
```

method it is possible to add any kind of subsystem to the model. However, in this case, the derivatives need to be supplied by the user. As subsystems are much smaller than whole dynamic models, these can usually be differentiated by the user. In all other cases approximation of derivatives such as finite differences may be used.

The builder method works in the same way as `addSubsystem`, but automatic derivatives will not be calculated. Additionally, only function handles are supported. The output sizes as well as the sparsities of the derivatives are determined automatically using a "not-a-number call" with suitable input sizes to the supplied subsystem handle. In case the subsystem does not support "not-a-number" calls, the output sizes as well as the sparsities can be set manually using *MATLAB* Name-Value pairs.

The return arguments of a differentiated subsystem have a certain order. All output values are returned, followed by a Jacobian for every output w.r.t. the subsystem input variables. In case the Hessians are required one for each output must be returned after the Jacobians. Thus, a subsystem having three inputs x, y, z and two individual outputs a, b has the following interface

```
[a, b, j_a, j_b, h_a, h_b] = subsystem(x,y,z)
```

where the preceding `j` and `h` are used to denote the Jacobian and Hessian. Please note that the names of the Jacobians and the Hessians must not appear in the output cell array of the `addDerivativeSubsystem` method. They are automatically assumed by *FALCON.m*. The derivatives provided by the user must match the input arguments. Additionally, derivatives of constant inputs must not appear in the provided derivatives. The structure of the derivatives is explained in section 4.5.2.

Variable Manipulation

It often occurs that individual values are required, but the variable is only available as a vector. On the other hand, sometimes variables need to be stitched together (e.g. to form a matrix or a vector). For these cases, the subsystem derivative builder in *FALCON.m* offers two methods:

SplitVariable splits a variable into multiple parts.

CombineVariables combines variables to a single new variable.

Multiple variables are combined using the method

```
mdl.CombineVariables('name', cellarr)
```

that requires a name for the new variable. Additionally, a cell array of strings is provided. It holds the names of the variables to be stitched together. The method may be used to concatenate variables to vectors or matrices. Therefore, the layout of the cell matrix is considered. The sizes of the individual variables must support a concatenation to a two dimensional rectangle (see Figure 4.8). Additionally, all variables must either be constant or dependent on the model inputs. Mixing both constant and derivative variables is currently not supported.

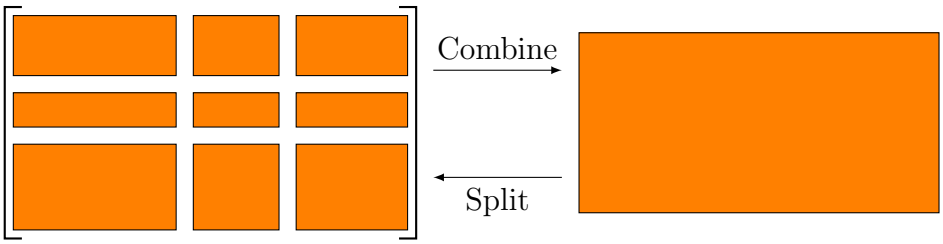


Figure 4.8: Combining and splitting of variables.

A vector or matrix can be split into multiple variables with the

```
mdl.SplitVariable('name', cellarr)
```

method. It requires the name of the variable to be split and a cell array of names for the generated parts. In the split variable method two cases have to be considered. In case the variable is split into scalar parts, the cell array of strings must have the same size as the variable. *FALCON.m* thus creates a new scalar variable for each.

If a variable is split into multiple matrices, the size of the cell array does not match the variable size. In this case, the column and row splitting vectors

```
mdl.SplitVariable('name', cellarr,...
    'RowSplit', rows, 'ColSplit', cols)
```

must be specified. They define how many rows and columns are assigned to each row and column of the cell array matrix. The sum of the splitting vectors must match the size of the original variable. For example, the variable A has the size $[6, 5]$. Using the command

```
mdl.SplitVariable('A', {'a','b','c';'d','e','f'},...
    'RowSplit', [2,4], 'ColSplit', [2,1,2])
```

it is split into 6 new variables

$$A \in \mathbb{R}^{6\times5} \quad \rightarrow \quad \begin{matrix} [a \in \mathbb{R}^{2\times2}] & [b \in \mathbb{R}^{2\times1}] & [c \in \mathbb{R}^{2\times2}] \\ [d \in \mathbb{R}^{4\times2}] & [e \in \mathbb{R}^{4\times1}] & [f \in \mathbb{R}^{4\times2}] \end{matrix} \tag{4.32}$$

of different sizes. The combining and splitting may be compared to mux / demux block in Simulink. Both methods do not remove their source variables as just new onces are created.

In order to avoid cluttering of variables, it is possible to combine and split variables directly in the subsystem call. The combine on input feature enables a user to concatenate variables in the input argument call. Instead of passing a variable string for the

input argument, a cell array of strings is used. The concatenation occurs in the same way as with the combine variables method but no new variable is created.

Similar to combine on input, the split on output feature splits a subsystem return variable into scalar entries. It is invoked by passing a cell array of strings instead of a string for the output considered. The cell array must match the return size. A splitting to a block of matrices is not supported.

The combine on input feature can be applied to both subsystem methods. However, the split on output feature is not supported by the derivative subsystem method, as otherwise the user supplied derivatives must be altered. Listing 4.5 shows an example of both convenience features.

Listing 4.5: Combine on input and split on ouput.

```
mdl.addSubsystem(@system,...
    {'in1', {'a','b';'c','d'}, 'in3'},... % Combine on Input
    {'out1', {'x';'y';'z'}});             % Split on Output
```

Additional Remarks

In the following, some additional remarks are given:

- Global variables may be used in subsystems. However, in case the analytic derivatives are calculated, the current value will be hard coded into the derivative function. Furthermore, this code is compiled to a MEX file. In case finite differences with evaluation in *MATLAB* are used, the original source function is linked. Thus, the global variable will persist in the model. It is recommended to avoid global variables as the behavior may become unpredictable.

- Constant inputs of user functions may have a variable size. A dimension of variable size is specified by setting it to infinity. Thus, e.g. look-up-table data can be replaced without having to regenerate the model. However, as the symbolic math toolbox does not support variable size data, a derivative subsystem must be used.

- In subsystem mode, the names of the return variables of models, constraints, or cost functions must be specified to the builder instance. Each builder instance supplies a method where the variable names can be specified. For example, the builder instance creating the dynamic model supplies the method `setDerivativeNames`, see Listing 4.3.

4.3.3 Derivative Builder Classes

FALCON.m implements specialized derivative builders for every user function type. Thus, derivatives of these can be created with just a few lines of code. In the following, the builder classes are explained in more detail. All of them support both the function and the subsystem mode.

Simulation Model Builder

The `falcon.SimulationModelBuilder` is used to differentiate dynamic models. As described in section 4.1, the dynamic models in *FALCON.m* implement the state derivatives and an optional output. In the constructor

```
mdl = falcon.SimulatioModelBuilder(ProjectName, States, Controls,...
    Parameters, Handle)
```

the states, controls, and parameters are defined. These can either be an array of *FALCON.m* data objects or a number specifying the length of the input vector. The latter is only supported in the function mode. If the model does not implement controls or parameters, the input is set to zero or empty (`[]`). The `ProjectName` input argument specifies the filename of the prepared function that implements the derivatives. In case the optional `Handle` parameter is set the function mode is invoked. In this case, all subsystem mode methods are disabled. Constant inputs are added to the model using the `addConstantInput` method which can be called multiple times.

Model outputs are set by calling the

```
mdl.setOutputs(outputs)
```

method. Inputs to the method can either be an array of `falcon.Constraint` objects or the number of outputs. The latter is only supported in function mode.

In subsystem mode, the variable names of the state derivatives must be specified. This is achieved with the `setStateDerivativeNames` method. Additionally, all other subsystem features are available.

The model is differentiated using the `Build` command. The resulting prepared model function has the following interface

```
[xdot, y, jxdot, jy, hxdot, hy] = model(x,u,p,c1,c2,..)
```

where the values of state derivatives `xdot` and outputs `y` are followed by their Jacobians and Hessians. The input arguments (states `x`, controls, `u`, parameters `p`, and constants `c`) are dependent on the inputs specified in the constructor. If the model does not implement outputs or does not require all inputs (e.g `p=[]`),

```
[xdot, jxdot, hxdot] = model(x,u,c1,c2,..)
```

the interface is reduced accordingly.

Path Constraint Builder

Path constraints and Lagrange cost functions are differentiated using instances of the `falcon.PathConstraintBuilder` class. It is created in a similar manner as the model builder

```
mdl = falcon.PathConstraintBuilder(ProjectName, Outputs, States,...
    Controls, Parameters, Handle)
```

where the model outputs act as an additional input. In case the subsystem mode is used, the `setConstraintValueNames` method must be used to specify the names of the return variables. The possible function interface is given by

```
[v, jv, hv] = pathconstraint(y,x,u,p,c1,c2,..)
```

where compared to the model dynamics an additional input argument for the outputs `y` exists. The constraint value vector `v` is returned together with its Jacobian `jv` and Hessian `hv`.

Point Constraint Builder

Point constraints and Mayer cost functions can span over multiple phases. Thus, the `falcon.PointConstraintBuilder` differs from the other builders. In the constructor

```
mdl = falcon.PointConstraintBuilder(ProjectName, Handle)
```

merely the name of the derivative function as well as the optional handle for the function mode are expected. In this builder, for each phase that participates, a so-called phase input is added using the

```
mdl.addPhaseInput(Outputs, States, Controls, NumTimeSteps)
mdl.addPhaseInput(Outputs, NumTimeSteps)
mdl.addPhaseInput(States, Controls)
```

method. As with the other builders, it expects an array of data objects or the number specifying the size of an input. Only required inputs have to be specified. The `NumTimeSteps` input argument tells the builder the number of time steps that can be expected for all phase inputs. The default value is one time step. In case of multiple time steps the column vectors entering the point function become matrices. Only the information and the expected number of time steps are defined, but not the phase and the time steps which the data actually comes from. The number of phase inputs is not limited.

Additionally to the phase inputs, optimizable parameters can be added to a point constraint using the `setParameters` method. As with the other builders, constant inputs can be added as well. All inputs to a point constraint are optional. Thus, it is possible to define constraints and cost functions that only depend on parameters.

In the interface of the derivative point constraint

```
[v, jv, hv] = pointconstraint(y1,x1,u1,y2,x2,u2,..,p,c1,c2,..)
[v, jv, hv] = pointconstraint(y1,u1,u2,..,p,c1,c2,..)
[v, jv, hv] = pointconstraint(p,c)
```

the phase inputs are expected in the order of definition followed by the constraint parameters and the list of constant inputs. Additionally, other possible interfaces are shown.

In case the subsystem mode is used, the variable names must be associated with a phase input block. As multiple phase inputs may have the same name for e.g. states, variables must be clearly identifiable. For each phase input a group index, starting at 1, is assigned. Every variable of the phase input is suffixed by `_g#` where # represents the group index. Thus, a state named `'speed'` entering with the second phase input block will have the variable name `'speed_g2'`. If the phase input has multiple time steps, the variable will be a row vector of the time step length. As before, the constraint value names have to be specified using `setConstraintValueNames`.

Name-Value Pairs

All builder classes offer an additional set of *MATLAB* Name-Value pairs that can be used to influence the derivative generation process. These options can be set in the constructor or in the build method. In the constructor of a builder, the following options are available:

DerivativeMode Flag that defines whether the derivatives are calculated analytically (`analytic`) or using forward finite differences (`finite_difference`). The latter is automatically selected in case the Symbolic Math Toolbox ist not available. The default setting is `analytic`.

Optimize Sets the code optimization option when writing the analytic derivatives to a *MATLAB* function. This feature is only available in *MATLAB* 2014b or later and if the DerivativeMode is `analytic`. In earlier versions of *MATLAB*, this option is fixed to `true`. Dependent on the subsystem or user function size, code optimization can yield a significant speed improvement during run-time but may take much longer during the derivative generation process. The default option is `false` for function mode and `true` for subsystem mode.

DoDependencyCheck Setting this flag to `true` causes *FALCON.m* to check whether a user function or subsystem depends on any other non-built-in functions. As these will not be considered in the hash value check, potential changes will not be detected. An update of the derivatives does not occur in this case. This option merely displays a warning if dependencies are found. However, since this check requires a significant amount of time in *MATLAB*, this option is set to `false` by default.

DoHessian Specifies whether the Hessian of the user function is calculated. The default value is `false`. This option is not available if the DerivativeMode is set to `finite_difference`.

ParentDirectory The preparation of the derivatives requires the generation of temporary files and data. These are stored in a subfolder which has the same name as the derivative function (`ProjectName`). This option specified where the parent directory of this folder is found. By default, the temporary folders for models and constraints are stored in the `fm_models` and `fm_constraints` respectively. Both folders are found in the current working directory of *MATLAB* and are created automatically during the build process. These folders can be deleted as the information is temporary. However, in this case all derivatives must be regenerated if the project is rebuilt.

With the call of the build method the following options can be set:

EvaluationProvider Flag specifying whether the differentiated user function shall be evaluated in *MATLAB* (`matlab`) or in a compiled MEX file (`mex`). *FALCON.m* automatically switches to *MATLAB* evaluation in case the *MATLAB* Coder is not available. The default option is `mex`.

MultiThreading Model and path constraint user functions are evaluated at multiple points in time. Due to the collocation method, all of these evaluations are independent. Therefore, this process can be highly parallelized. Setting the multi-threading option to `true` enables parallelized evaluation in the compiled MEX files. The OpenMP compiler options are used. This option is deactivated by default as not all compilers support this framework. A parallelized evaluation in the `matlab` EvaluationProvider option is currently not considered.

OutputFolder Specifies where the final prepared user function is stored. The default location is the current work directory.

Discrete Control Support

In this thesis, discrete controls are considered in optimal control problems. The Outer Convexification was introduced in chapter 3. The OC reformulates the discrete controls in continuous form. Thus, these types of problems can be solved with existing methods for continuous optimal control problems.

However, the OC requires the evaluation of all discrete control combinations at every time step. All evaluations are weighted

$$\dot{\vec{x}} = \sum_k w_k \cdot \vec{f_x}\left(\vec{x}(t), \vec{u}(t), \vec{p}, \vec{v}_k, \vec{c}, \dots\right) \tag{4.33}$$

$$\vec{y} = \sum_k w_k \cdot \vec{f_y}\left(\vec{x}(t), \vec{u}(t), \vec{p}, \vec{v}_k, \vec{c}, \dots\right) \tag{4.34}$$

$$\sum_k w_k = 1, \qquad w_k \in [0, 1] \tag{4.35}$$

to create a combined state derivative and output. The original discrete control values becomes a constant in the optimal control problem but changes with the choice of the discrete control. In order to avoid complicated creations of models, the OC approach is implemented directly into the derivative generation tool-chain.

A discrete control is added to the simulation model builder instance with the

```
mdl.addDiscreteControl(Name, Size)
```

method that expects the variable name of the discrete control and its size. As stated above, the discrete controls enter the model dynamics as a constant matrix. Every column is assumed to represent a discrete control choice. The number of columns is variable and depends on the number of discrete choices during run-time. Therefore, the specified size of the discrete control must represent a column vector.

An unlimited number of discrete controls may be added to the model dynamics. Each discrete control acts as separate input. Please note that the simulation model builder does not evaluate all possible discrete control combinations automatically. The differentiated user function merely evaluates the combinations passed to it and applies the OC.

If discrete controls are involved, the function interface of the differentiated model dynamics

```
[xdot,y,jxdot,jy,hxdot,hy] = model(x,u,p,v1,v2,..,c1,c2,..,alpha,w)
```

slightly differs from the nominal case. The discrete control value inputs (`v1,v2,...`) are expected between the model parameters and the constants. The discrete control weights `w` are the last input. Within the optimal control problem they are considered as additional controls. The `alpha` input parameter is an additional input that can be used to augment the derivatives by specifying additional of zero columns and rows (e.g. to account for slack variables not considered in the model dynamics). This is e.g. required for the switching cost approach by [6]. The derivative structure of a model with discrete controls is shown in the next section.

As the *FALCON.m* optimal control problem does not support discrete controls, the prepared discrete control model cannot be used in *FALCON.m* directly. Instead, a wrapper function must be used. Section 4.7 describes how discrete controls and vanishing constraints are considered in the optimal control toolbox.

4.4 Interfaces and Advanced Options

In this section, the interface between the *FALCON.m* optimal control toolbox and the differentiated user functions is explained in more detail. Although this information is not required for the default usage of *FALCON.m*, it is nevertheless a good idea to understand the underlying structure. Furthermore, there may be situations in which a user function cannot be differentiated by the derivative builder classes. For instance, a struct is used within the user function which is not supported as input argument. In this case, a manually written function may be used instead.

4.4.1 Derivative Structure

In the previous section, the derivative builder was introduced. Prepared user functions do not only return values but also the derivatives w.r.t. the inputs. In the following, the structures of the user function derivatives are explained.

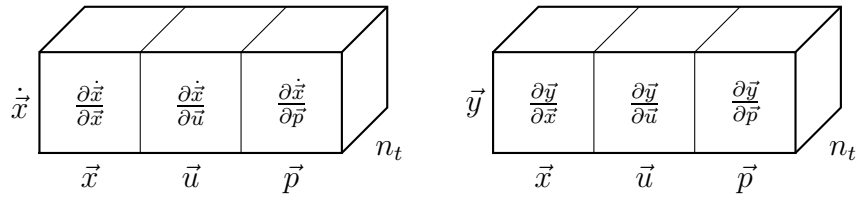
In *FALCON.m*, all derivatives are returned in matrices containing the Jacobian and Hessian w.r.t. all inputs. In case a user function is evaluated at multiple time steps (such as dynamic models and path constraints), each time step is a page in the third dimension. The number of pages in the third dimension equals the number of evaluated time steps n_t.

Dynamic Models

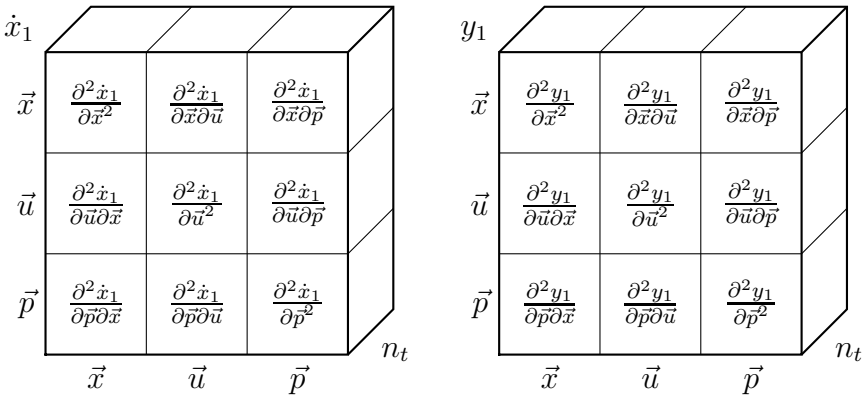
Dynamic models return not only the state derivatives but also additional model outputs. Therefore, a dynamic model in *FALCON.m* may have two return arguments and thus provides a Jacobian and a Hessian for each argument.

Figure 4.9a shows the structure of the Jacobian. The derivatives are stored in input blocks with the order states $\vec{x}$, controls $\vec{u}$, and parameters $\vec{p}$. The third dimension represents the number of time steps n_t.

Similarly, the Hessians of the dynamic model are shown in Figure 4.9b. The Hessians for each entry of the return parameters are stacked vertically. In the figure, the Hessians for the first entries are shown. Each Hessian block can be split into multiple sub-blocks that represent the different derivative combinations.



(a) Jacobian layout of state derivatives and outputs of dynamic models.



(b) Hessian layout of state derivatives and outputs of dynamic models.

Figure 4.9: Dynamic model derivative structure without discrete controls.

Dynamic Models with Outer Convexification

In case the dynamic model implements the OC of the discrete controls, the return derivative structure is expanded. In Figure 4.10, the structures are shown for the state derivatives. It can be seen that additional dependencies are introduced. These dependencies are the discrete control weights $\vec{w}$ and additional slack control variables $\vec{\alpha}$.

It may not be visible directly, but the derivative structure does not match the one expected by *FALCON.m*. As the slack variables and the discrete control weights represent controls in the optimal control problem, the derivative order should be

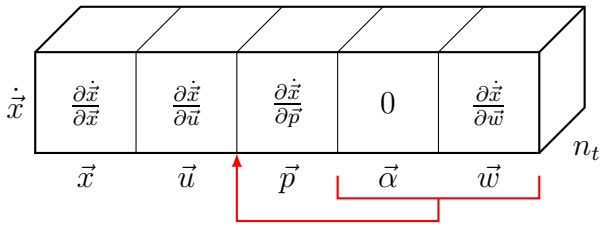
$$\vec{x}, \vec{u}, \vec{\alpha}, \vec{w}, \vec{p}. \tag{4.36}$$

This is not met in case the model implements parameters which is an issue that was not accounted for during development. The C++ code that generates the extended derivatives does not have any information on the order or type of the derivatives. It merely appends the OC to the derivative blocks. In case the model implements parameters, the rows and columns must be shifted (see red arrows in Figure 4.10). This shift is performed by the *FALCON.m* discrete control extension which is described in section 4.7.

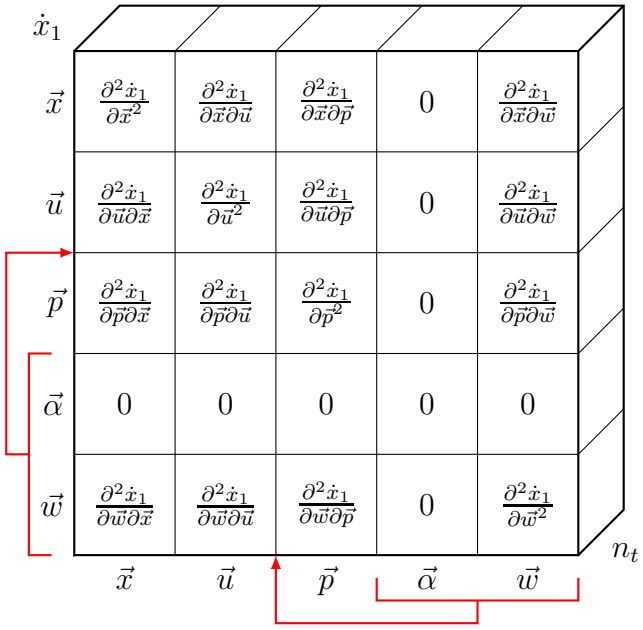
The appended derivatives of the OC are not calculated by the differentiated model but are implemented in the MEX file that wraps the original non-discrete model. The derivatives can be derived from the definition of the OC:

$$\dot{\vec{x}} = \sum_k w_k \cdot \vec{f_x}\left(\vec{x}, \vec{u}, \vec{p}, \vec{v}_k\right). \tag{4.37}$$

As the model state derivatives are weighted by $\vec{w}$, the corresponding Jacobian and



(a) Jacobian layout of outer convexification approach.



(b) Hessian layout of outer convexification approach.

Figure 4.10: Jacobian and Hessian layout of dynamic model derivatives implementing discrete controls and outer convexification.

Hessian blocks

$$\frac{\partial \dot{\vec{x}}}{\partial [\vec{x}, \vec{u}, \vec{p}]} = \sum_k w_k \cdot \frac{\partial \vec{f}_x}{\partial [\vec{x}, \vec{u}, \vec{p}]} (\vec{x}, \vec{u}, \vec{p}, \vec{v}_k) \tag{4.38}$$

$$\frac{\partial^2 \dot{\vec{x}}}{\partial [\vec{x}, \vec{u}, \vec{p}]^2} = \sum_k w_k \cdot \frac{\partial^2 \vec{f}_x}{\partial [\vec{x}, \vec{u}, \vec{p}]^2} (\vec{x}, \vec{u}, \vec{p}, \vec{v}_k) \tag{4.39}$$

are weighted in the same way. The derivatives w.r.t. the discrete control weights $\vec{w}$ are basically a re-sorting of the values and Jacobians of the state derivatives for different discrete control evaluations. In the Jacobian, the derivatives w.r.t. the discrete control weights

$$\frac{\partial \dot{\vec{x}}}{\partial \vec{w}} = \left[\begin{array}{ccc} \vec{f}_x (\vec{x}, \vec{u}, \vec{p}, \vec{v}_1) & \dots & \vec{f}_x (\vec{x}, \vec{u}, \vec{p}, \vec{v}_{n_v}) \end{array} \right] \tag{4.40}$$

are the state derivative columns for each discrete control combination concatenated to

a matrix. Similarly, the Hessian of an entry is created by stacking the Jacobians

$$\frac{\partial^2 \dot{x}_1}{\partial \vec{w} \partial [\vec{x}, \vec{u}, \vec{p}]} = \left(\frac{\partial^2 \dot{x}_1}{\partial [\vec{x}, \vec{u}, \vec{p}] \partial \vec{w}} \right)^T = \begin{bmatrix} \frac{\partial f_{x,1}}{\partial [\vec{x}, \vec{u}, \vec{p}]} (\vec{x}, \vec{u}, \vec{p}, \vec{v}_1) \\ \vdots \\ \frac{\partial f_{x,1}}{\partial [\vec{x}, \vec{u}, \vec{p}]} (\vec{x}, \vec{u}, \vec{p}, \vec{v}_{n_v}) \end{bmatrix} \tag{4.41}$$

of a state derivative entries vertically. It can easily be seen that the Hessian

$$\frac{\partial^2 \dot{x}_1}{\partial \vec{w}^2} = 0 \tag{4.42}$$

is equal to zero. Due to the fact that the slack controls $\vec{\alpha}$ bypass the model, any partial derivatives are zero.

Path Constraints and Lagrange Cost Functions

The structures of the path constraint and Lagrange cost function derivatives are very similar to the dynamic models. Figure 4.11 shows the derivative structure for the Jacobian and Hessian. There exists an additional dependency, which are the model outputs. Thus, the order of derivatives blocks are outputs $\vec{y}_c$, states $\vec{x}_c$, controls $\vec{u}_c$, and parameters $\vec{p}_c$. As stated in section 4.2.4, all constraint inputs may be a subset of the available data in the phase. The parameters $\vec{p}_c$ may be partially common with model input parameters $\vec{p}$.

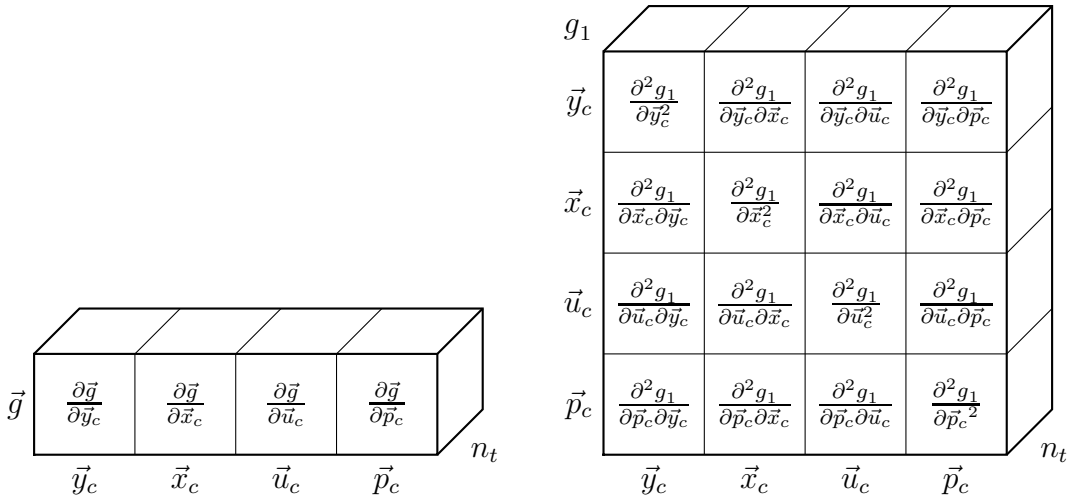


Figure 4.11: Jacobian and Hessian layout of path constraint and Lagrange cost function.

Vanishing Constraints

Vanishing constraints are not supported by the derivative builder as a generalized formulation cannot easily be written in C/C++ code. Therefore, the augmentation of the vanishing constraints is done in *MATLAB* and implemented in the *FALCON.m* discrete control extension (see section 4.7). Here, the vanishing constraint derivative structure of the Jacobian and the Hessian is explained. For simplicity, a scalar vanishing constraint is assumed. A vectorized evaluation is achieved by applying the presented formulas to all entries of the constraint vector.

The vanishing constraint is defined by a function

$$\psi(g, w_c, \kappa) \leq 0, \qquad g \leq 0, \quad w_c \in [0, 1], \quad \kappa \geq 0 \tag{4.43}$$

where g is a user defined path function that shall be used in the vanishing constraint, and where $w_c \subseteq \vec{w}$ is an entry of all discrete control weights. The relaxation parameter $\kappa \subseteq \vec{\alpha}$ is an entry in the slack variable vector. In this example, the relaxation parameter is assumed to be optimizable.

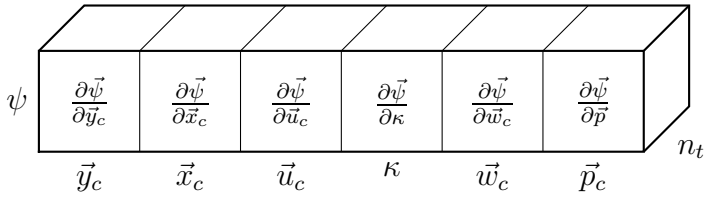
As introduced in section 3.3.1, the vanishing constraint can be relaxed

$$\psi(g, w_c, \kappa) = w_c \cdot (g + \kappa) - \kappa \leq 0 \tag{4.44}$$

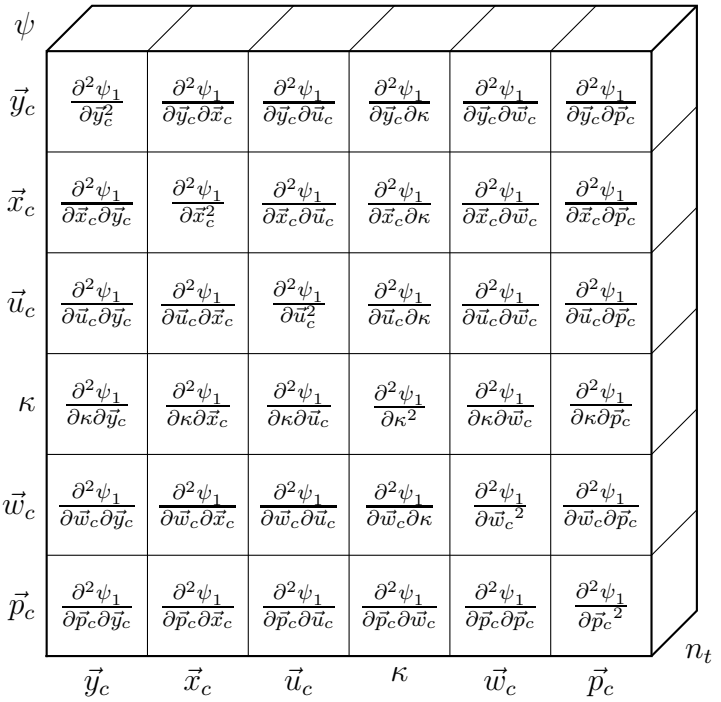
or reformulated

$$\psi(g, w_c, \kappa) = \frac{1}{2}\left(g w_c + \sqrt{g^2 {w_c}^2 + \kappa^2} + \sqrt{{w_c}^2 + \kappa^2} - w_c\right) - \kappa \leq 0 \tag{4.45}$$

to make it numerically easier to solve.



(a) Jacobian layout of vanishing constraint.



(b) Hessian layout of a vanishing constraint.

Figure 4.12: Jacobian and Hessian layout of a vanishing constraint.

Figure 4.12 shows the Jacobian and Hessian structure of the vanishing constraint that needs to be returned to *FALCON.m*. The path constraint that is used in the vanishing constraint formulation is assumed to be an already prepared user function. However, it returns the local values, Jacobian, and Hessian w.r.t. its inputs. For the vanishing constraint, the return data must be augmented.

The chain rule for the Jacobian is applied to the vanishing constraints w.r.t. the path constraint

$$\frac{\partial \psi}{\partial [\vec{y}_c, \vec{x}_c, \vec{u}_c, \vec{p}_c]} = \frac{\partial \psi}{\partial g} \cdot \frac{\partial g}{\partial [\vec{y}_c, \vec{x}_c, \vec{u}_c, \vec{p}_c]}, \tag{4.46}$$

as well as the discrete control weight and the slack variables:

$$\frac{\partial \psi}{\partial \kappa}, \qquad \frac{\partial \psi}{\partial w_c}. \tag{4.47}$$

The latter two do not have any further dependencies. Therefore, the local derivatives can be used in the Jacobian directly.

In the Hessian calculation, the chain rule for the path constraint dependency

$$\frac{\partial^2 \psi}{\partial [\vec{y}_c, \vec{x}_c, \vec{u}_c, \vec{p}_c]^2} = \left(\frac{\partial g}{\partial [\vec{y}_c, \vec{x}_c, \vec{u}_c, \vec{p}_c]} \right)^T \cdot \frac{\partial^2 \psi}{\partial g^2} \cdot \frac{\partial g}{\partial [\vec{y}_c, \vec{x}_c, \vec{u}_c, \vec{p}_c]} + \frac{\partial \psi}{\partial g} \cdot \frac{\partial^2 g}{\partial [\vec{y}_c, \vec{x}_c, \vec{u}_c, \vec{p}_c]^2} \tag{4.48}$$

is given as well as the dependencies to the slack variable and the discrete control weights:

$$\frac{\partial^2 \psi}{\partial \kappa \partial [\vec{y}_c, \vec{x}_c, \vec{u}_c, \vec{p}_c]} = \frac{\partial^2 \psi}{\partial \kappa \partial g} \cdot \frac{\partial g}{\partial [\vec{y}_c, \vec{x}_c, \vec{u}_c, \vec{p}_c]}, \tag{4.49}$$

$$\frac{\partial^2 \psi}{\partial w_c \partial [\vec{y}_c, \vec{x}_c, \vec{u}_c, \vec{p}_c]} = \frac{\partial^2 \psi}{\partial w_c \partial g} \cdot \frac{\partial g}{\partial [\vec{y}_c, \vec{x}_c, \vec{u}_c, \vec{p}_c]}. \tag{4.50}$$

Both the Hessian and the Jacobian calculations include derivatives

$$\frac{\partial^2 \psi}{\partial {w_c}^2}, \qquad \frac{\partial^2 \psi}{\partial \kappa^2}, \qquad \frac{\partial^2 \psi}{\partial g^2}, \tag{4.51}$$

$$\frac{\partial^2 \psi}{\partial w_c \partial \kappa} = \left(\frac{\partial^2 \psi}{\partial \kappa \partial w_c} \right)^T, \qquad \frac{\partial^2 \psi}{\partial w_c \partial g} = \left(\frac{\partial^2 \psi}{\partial g \partial w_c} \right)^T, \qquad \frac{\partial^2 \psi}{\partial \kappa \partial g} = \left(\frac{\partial^2 \psi}{\partial g \partial \kappa} \right)^T \tag{4.52}$$

which are dependent on the vanishing constraint formulation. The derivatives for the relaxation approach (4.44) are given by:

$$\frac{\partial \psi}{\partial g} = w_c, \qquad \frac{\partial \psi}{\partial w_c} = g + \kappa, \qquad \frac{\partial \psi}{\partial \kappa} = w_c - 1, \tag{4.53}$$

$$\frac{\partial^2 \psi}{\partial g^2} = 0, \qquad \frac{\partial^2 \psi}{\partial g^2} = 0, \qquad \frac{\partial^2 \psi}{\partial \kappa^2} = 0, \tag{4.54}$$

$$\frac{\partial^2 \psi}{\partial g \partial w_c} = \frac{\partial^2 \psi}{\partial w_c \partial g} = 1, \qquad \frac{\partial^2 \psi}{\partial g \partial \kappa} = \frac{\partial^2 \psi}{\partial \kappa \partial g} = 0, \qquad \frac{\partial^2 \psi}{\partial w_c \partial \kappa} = \frac{\partial^2 \psi}{\partial \kappa \partial w_c} = 1. \tag{4.55}$$

In case the reformulation approach (4.45) is used, the derivatives are more complicated:

$$\frac{\partial \psi}{\partial g} = \frac{1}{2} \cdot \left(w_c + \frac{g {w_c}^2}{\sqrt{g^2 {w_c}^2 + \kappa^2}} \right), \tag{4.56}$$

$$\frac{\partial \psi}{\partial w_c} = \frac{1}{2} \cdot \left(g + \frac{g^2 w_c}{\sqrt{g^2 {w_c}^2 + \kappa^2}} + \frac{w_c}{\sqrt{{w_c}^2 + \kappa^2}} - 1 \right), \tag{4.57}$$

$$\frac{\partial \psi}{\partial \kappa} = \frac{1}{2} \cdot \left(\frac{\kappa}{\sqrt{g^2 {w_c}^2 + \kappa^2}} + \frac{\kappa}{\sqrt{{w_c}^2 + \kappa^2}} \right) - 1, \tag{4.58}$$

$$\frac{\partial^2 \psi}{\partial g^2} = \frac{1}{2} \cdot \frac{\kappa^2}{\left(g^2 {w_c}^2 + \kappa^2\right)^{3/2}}, \tag{4.59}$$

$$\frac{\partial^2 \psi}{\partial {w_c}^2} = \frac{1}{2} \cdot \left(\frac{\kappa^2}{\left(g^2 {w_c}^2 + \kappa^2\right)^{3/2}} + \frac{\kappa^2}{\left({w_c}^2 + \kappa^2\right)^{3/2}} \right), \tag{4.60}$$

$$\frac{\partial^2 \psi}{\partial \kappa^2} = \frac{1}{2} \cdot \left(\frac{g^2 {w_c}^2}{\left(g^2 {w_c}^2 + \kappa^2\right)^{3/2}} + \frac{{w_c}^2}{\left({w_c}^2 + \kappa^2\right)^{3/2}} \right), \tag{4.61}$$

$$\frac{\partial^2 \psi}{\partial g \partial w_c} = \frac{\partial^2 \psi}{\partial w_c \partial g} = \frac{1}{2} \cdot \left(1 + \frac{g^3 {w_c}^3 + 2 g w_c \kappa^2}{\left(g^2 {w_c}^2 + \kappa^2\right)^{3/2}} \right), \tag{4.62}$$

$$\frac{\partial^2 \psi}{\partial g \partial \kappa} = \frac{\partial^2 \psi}{\partial \kappa \partial g} = -\frac{1}{2} \cdot \frac{g^2 {w_c}^2 \kappa}{\left(g^2 {w_c}^2 + \kappa^2\right)^{3/2}}, \tag{4.63}$$

$$\frac{\partial^2 \psi}{\partial w_c \partial \kappa} = \frac{\partial^2 \psi}{\partial \kappa \partial w_c} = -\frac{1}{2} \cdot \left(\frac{g^2 w_c \kappa}{\left(g^2 {w_c}^2 + \kappa^2\right)^{3/2}} + \frac{w_c \kappa}{\left({w_c}^2 + \kappa^2\right)^{3/2}} \right). \tag{4.64}$$

Point Constraints and Mayer Cost Functions

As point constraints and Mayer cost functions span across multiple phases, the derivative structure must account for any type of dependency. In 4.13 the structures of the Jacobian and of the Hessian are given. As before, the parameters are independent of the phase. It can be seen that in the Jacobian all phase derivatives are independent, whereas in the Hessian they may be coupled in an arbitrary way.

The derivatives w.r.t. a phase are depicted by a phase input

$$\vec{z}_p = \begin{bmatrix} \vec{y}_c \\ \vec{x}_c \\ \vec{u}_c \end{bmatrix} \tag{4.65}$$

that consists of optional outputs $\vec{y}_c$, states $\vec{x}_c$, and controls $\vec{u}_c$. The Jacobian structure of a phase input block is shown in Figure 4.14. Within the point constraint a phase input can have multiple time steps. Thus, a matrix enters the point constraint rather than a vector. In the derivatives, these matrices are considered

$$\vec{y}_c = [\vec{y}_{c,1}, \vec{y}_{c,2}, \dots] \qquad \rightarrow \qquad \vec{y}_c := \begin{bmatrix} \vec{y}_{c,1} \\ \vec{y}_{c,2} \\ \vdots \end{bmatrix} \tag{4.66}$$

by stacking the columns and thus create a single vector.

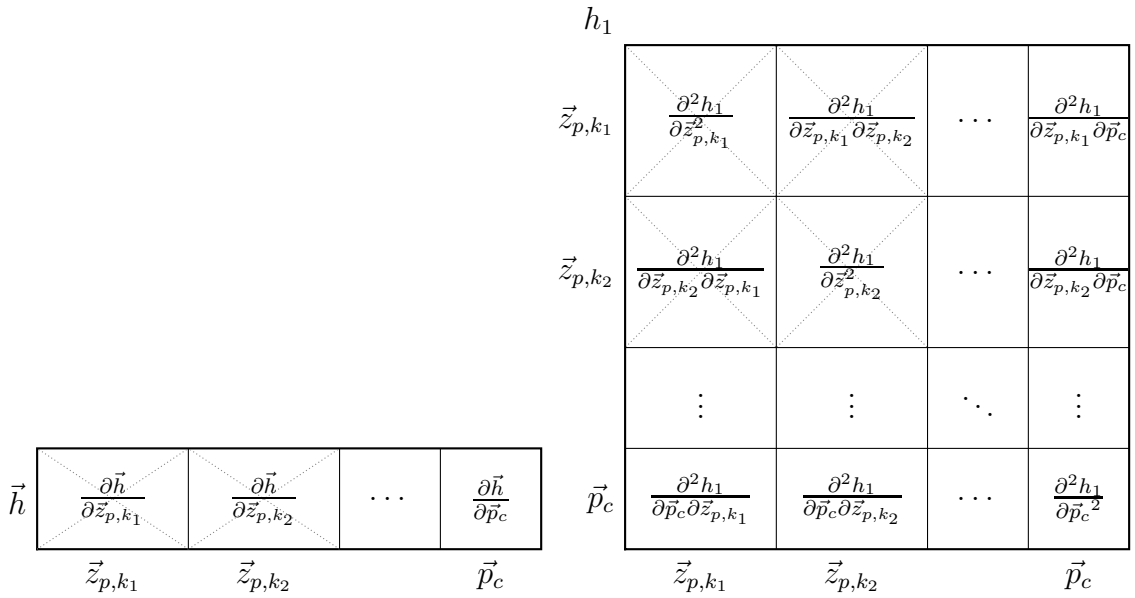


Figure 4.13: Jacobian and Hessian layout of path constraint and Lagrange cost function.

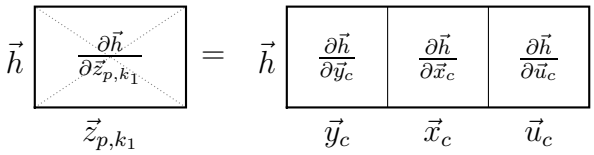


Figure 4.14: Phase input block.

4.4.2 Derivative Builder Info Struct

Every user function of *FALCON.m* which is differentiated by the derivative builder implements an info struct interface. This allows the user function to tell *FALCON.m* which information is required for the evaluation. Thus, the implementation complexity is reduced for the user since only necessary information enter the constraints. Additionally, it can be checked whether the user function and the problem definition coincides.

Figure 4.15 shows the info struct layout. The fields have the following meaning and relevance:

input / output Struct array specifying the input and output information of the user function. An entry in the respective array is made for every input and output.

- **m / n** Dimension of the input where `m` states the number of rows and `n` the number of columns. Vectors that are evaluated at multiple time steps (e.g. state input in a path constraint) return `n` equal to one. *FALCON.m* knows that certain input types require multiple time evaluations.
- **name** Name of the input which is created internally during derivative generation process. It is the name of the input argument of the intermediate *MATLAB* user function that implements the derivatives (see Derivative Function Interface in 4.5.3). No check is made w.r.t. this field.
- **argnames** Names of the *FALCON.m* data objects used to specify the inputs of e.g. a path constraint. Using this cell array of strings, *FALCON.m* is able to

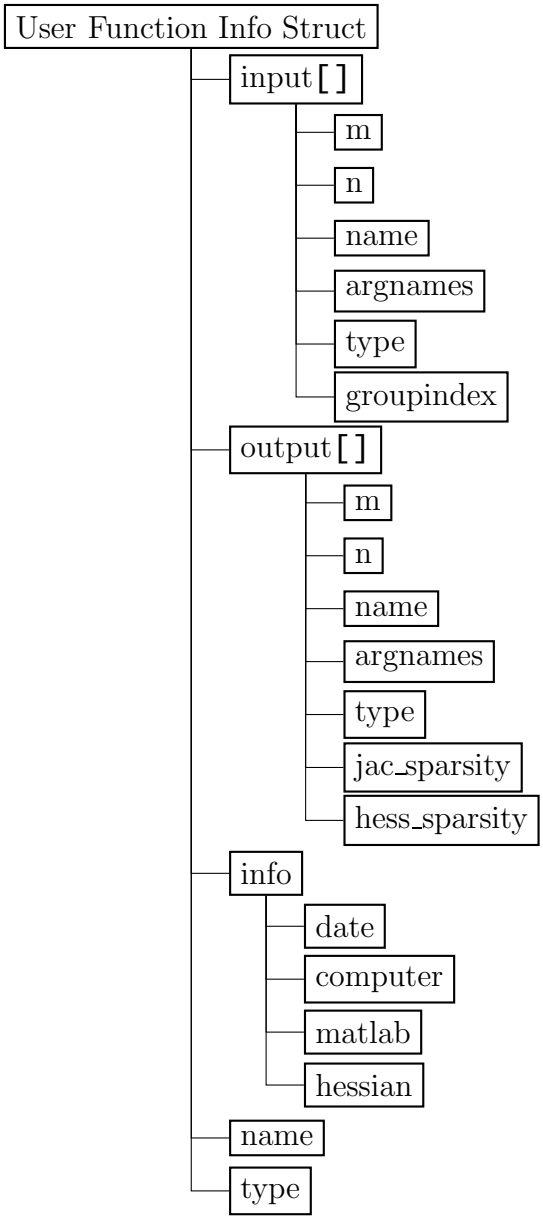


Figure 4.15: User function info struct interface.

extract the relevant information from the optimal control problem. In case the names are unknown, only the size of the input will be checked. In this case, the size has to match the corresponding data in the problem definition.

type Specifies the type of data required or returned. Valid keys for inputs are `OUTPUT`, `STATE`, `CONTROL`, `PARAMETER`, `CONSTANT`, and `DISCRETE`. For outputs, the possible keys are `VALUE` for constraints and cost functions as well as `STATEDOT`, and `OUTPUT` for simulation models.

groupindex Index of a phase input entering a point constraint starting at 1. Parameters and constants have a group index of 0. In all other cases (path constraints, Lagrange cost functions, and dynamic models) this field is irrelevant.

jac_sparsity / hess_sparsity Template sparsity of output calculated by the derivative builder. In case no hessian is supplied by the user function, the corre-

sponding sparsity field is empty.

info Struct containing additional information about the user function.

date Date the user function was created / differentiated.

computer Name of computer the user function was created on.

matlab *MATLAB* version used for the creation.

hessian Logical flag specifying whether the user function supports Hessian calculation.

name Name of the user function that coincide with the file name. Name is used to check whether multiple definitions exist in the *MATLAB* path. In case multiple definitions a warning is printed.

type Type of user function. It is used check correct attachment to the problem. Valid keys are `SIMULATION_MODEL` for dynamic models, `PATH_FUNCTION` for path constraints and Lagrange cost functions, and `POINT_FUNCTION` for point constraints and Mayer cost functions.

The info struct is returned by the user function if a single output argument is expected but no input arguments are passed. In case this evaluation fails but the function exists, *FALCON.m* automatically assumes that the function has not been differentiated yet. The corresponding derivative builder is automatically evaluated in function mode. This way, the automatic function templates are created and differentiated (see section 4.2.6).

4.4.3 Custom User Functions

All builder instances in *FALCON.m* require that the size and number of input optimization variables are fixed. This is necessary, as otherwise the derivatives cannot be calculated. If the input situation changes, a reconstruction of the derivatives is required.

There may be situations where the input situation changes for every optimization. Therefore, a reconstruction in this case may not be desired. For instance, the summation constraint of the outer convexification approach

$$\sum_{k=1}^{n_v} w_k = 1 \tag{4.67}$$

is dependent on the number of discrete choices n_v. If n_v changes, the corresponding builder must be re-evaluated which may be a major drawback. As this is not desired, the issue can be resolved by providing *FALCON.m* with custom user function that accepts variable inputs. In this case, the derivatives and the info struct interface must be custom written. Here, the best practice approach is stated.

In the example Listing 4.6, the summation constraint is implemented. The function expects the discrete control weights `w` as a matrix for all time steps.

Additionally, the input `controls_w` is an array of `falcon.Control` objects representing the weights of the discrete controls expected by the path constraint. The array is used to return the correct input / output sizes and names to *FALCON.m* if the info struct is requested. The first input is hidden from *FALCON.m*

Listing 4.6: Manual written user function that conforms with the *FALCON.m* info struct interface.

```
function [ val, jval, hval ] = pathfunc_dcdc(controls_w, w)
% Calculates the summation constraint of discrete controls

nw = numel(controls_w);

if nargin <= 1
    % Input
    struc.input(1).m                = nw;
    struc.input(1).n                = 1;
    struc.input(1).name             = 'controls';
    struc.input(1).type             = 'CONTROL';
    struc.input(1).groupindex       = 0;
    struc.input(1).argnames         = {controls_w.Name}.';

    % Output
    struc.output(1).m               = 1;
    struc.output(1).n               = 1;
    struc.output(1).name            = 'constraint';
    struc.output(1).argnames        = {};
    struc.output(1).type            = 'VALUE';
    struc.output(1).jac_sparsity    = ones(1, nw);
    struc.output(1).hess_sparsity   = zeros(nw,nw);

    % Other
    struc.info.hessian = true;
    struc.name = 'pathfunc_dcdc';
    struc.type = 'PATH_FUNCTION';

    val = struc;
    return
end

val  = sum(w,1);
jval = ones(1, nw, size(w,2));
hval = zeros(nw, nw, size(w,2));

end
```

```
controls_w = [falcon.Control('dc_1'); falcon.Control('dc_2')];
phase.addNewPathConstraint(...
    @(varargin)pathfunc_dcdc(controls_w, varargin{:}))
```

using an anonymous function. Thus, in case the info struct is requested the user function is called without input arguments. Using the `varargin` keyword, anonymous functions support a variable number of input arguments. In the custom implementation, a single input is provided instead of two. This is detected by the `if` statement which invokes the return of the info struct. During run-time, the values, Jacobian, and Hessian of the constraint are returned with the correct size. The correct implementation of derivatives is checked using derivative check methods in the `falcon.Problem` implementation (`falcon.Problem.CheckGradient/CheckHessian`).

4.5 Subsystem Derivative Builder

In section 4.3, the derivative construction in *FALCON.m* was described from a user point of view. This included the different builder instances for dynamic models, constraints, and cost functions. Additionally in section 4.4, the interface between the user function and the optimal control toolbox was discussed. In this section, the internal structure of the derivative generation toolchain is explained. As the derivatives are constructed using multiple subsystems, this part of *FALCON.m* is referred to as the Subsystem Derivative Builder (SDB) [112].

The SDB algorithm calculates the derivatives of all subsystems and combines them using the chain rule. Additionally, the sparsity (template sparsity) and the info struct interface are created. Internally, the SDB algorithm is generic and therefore interfaced by the user function builder instances of section 4.3.

The section is organized as follows. Section 4.5.1 lists currently available differentiation methods together with requirements for derivatives in direct optimal control methods using collocation. In section 4.5.2, the subsystem derivative method is introduced. Implementation details and a comparison to other derivative approaches are given in sections 4.5.3 and 4.5.4.

4.5.1 Existing Derivative Methods

All differentiation methods may be distributed in two main categories: manual and automatic. Calculating the derivatives by hand is associated to the first category. This approach is error prone by nature and not applicable for any serious / large scale applications. Therefore, it is not discussed further.

Automatic differentiation methods offer many benefits. Only the source function / dynamic model has to be defined. The calculation of the derivatives (1^{st} , 2^{nd} or higher order) is a completely automated process. User interaction during or a validation of the results after differentiation is (ideally speaking) not required.

There are many algorithms or toolboxes that offer automatic differentiation. A detailed list can be found on the web [113] sorted by target programming languages. In the following, different automatic methods are discussed.

Finite Differences

Finite Differences (FD) can be regarded as a "brute force" approach to differentiation which in many situations is not the ideal choice. However, this method is still used in many applications as it is simple to implement. There are several methods available

$$\frac{\partial f}{\partial x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{4.68}$$

$$\frac{\partial f}{\partial x} = \lim_{h \to 0} \frac{f(x) - f(x-h)}{h} \tag{4.69}$$

$$\frac{\partial f}{\partial x} = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h} \tag{4.70}$$

namely Forward Finite Differences (FFD) (4.68), Backward Finite Differences (BFD) (4.69), and Central Finite Differences (CFD) (4.70). f represents a scalar function, x a scalar value and $h > 0$ the deviation used by the finite difference. The main benefit of FD is that they always return a derivative approximation, regardless whether the function is differentiable or not. This benefit is also their greatest weakness since the derivatives will always be just an approximation. Two errors influence the accuracy of the derivative [114]. Truncation error appear due to the Taylor approximation from which the method can be derived from. Additionally, roundoff errors due to machine accuracy influence the resulting derivatives.

In terms of computational cost, FD require additional function evaluations. Let $\vec{x}$ be a vector of size n. In this case, $n + 1$ function evaluations are necessary. According to [114], the central finite differences approach has a higher accuracy at the cost of even more evaluations ($2 \cdot n + 1$).

Additionally to the FD approaches presented above, a numeric gradient can be calculated using a complex step [70]. The definition of the numeric derivative

$$\frac{\partial f}{\partial x} = \lim_{h \to 0} \frac{\mathrm{Im}\left[f(x + ih)\right]}{h}$$

does not involve any subtraction error. In fact, the complex step method converges to machine precision for $h \to 0$ [70]. Therefore, it has a clear advantage compared to the other methods. The only drawback is that f must be analytic (differentiable in the complex plane) which is not supported by all functions (e.g. `atan2` in *MATLAB*).

To compare the FD methods, an arbitrary example function

$$f(x) = sin^2(5 \cdot x)$$

is defined and evaluated at 1000 equidistant points on the interval $x(t) \in [0, 10]$. At every evaluation point the analytic- g_{ana} and the numeric gradient g_{num} are calculated. The maximum relative error

$$e_{rel,max} = \max \frac{|g_{num} - g_{ana}|}{|g_{ana}|} \tag{4.71}$$

is obtained. Figure 4.16 plots the maximum relative error for different step sizes h. It can be seen that the complex step approach produces the best result and converges

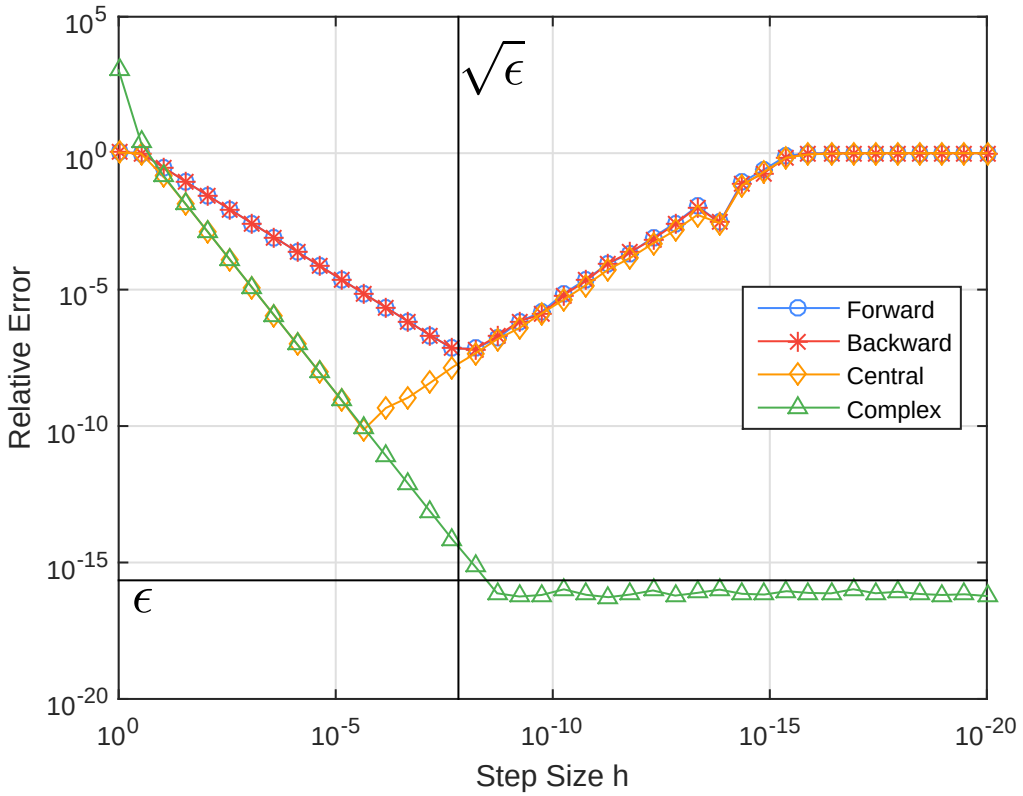


Figure 4.16: Error comparison of numerical derivatives.

to the machine accuracy[1] ϵ. The central differences approach initially produces similar accuracy as the complex step approach. For very small h, the relative error lies at around the same order of magnitude as the forward and backward finite differences approaches. All three converge to $e_{rel} = 1$ due to machine accuracy. The best accuracy for forward and backward finite differences lies at around $h = \sqrt{\epsilon} \cdot |x|$ which is often stated as a good approximation for the optimal step size [114]. In practice, the step size is lower bounded ($h = \sqrt{\epsilon} \cdot \max(|x|, 1)$).

Operator Overloading

Another way to automatically calculate the derivatives of a source function with machine accuracy is by operator overloading. This method follows the assumption, that every calculation performed on a computer can be represented by a set of principal operations ($*, /, +, -, \ldots$). The derivative rules for these operations are known (e.g. chain rule, product rule,...).

Instead of using numeric values for the function call, a class which holds not only the value but also the derivatives w.r.t. the independent variables is used. Within the class, the operators for all principle operations and commonly used math functions (trigonometric, exponential) are implemented.

Operator overloading can be applied to the source code as it is, even loops and conditions can be evaluated. In many cases, only the data type has to be replaced.

[1]Evaluated on Core i5-4260U @ 1.4GHz running Windows 8.1 64bit with a *MATLAB* 2015b given machine accuracy of $\epsilon = 2.22 \cdot 10^{-16}$

However, this method has some drawbacks as well.

First of all, since the original data types are replaced by classes, compilers cannot perform code optimization during compilation. Thus, it can be expected that these algorithms run slower than the source function or analytic implementations of derivatives.

Second of all, operator overloading has to be supported by the programming language it is used in. A list of software packages can be found in [113] for all common programming languages. In this thesis, only *MATLAB* and C++ implementations are relevant. For C++, which can be interfaced to *MATLAB* through MEX files, widely used implementations are Automatic Differentiation using Expression Templates (ADEPT) [71] and Automatic Differentiation by OverLoading in C++ (ADOL-C) [72]. The latter has been successfully applied to flight trajectory optimization [33]. *MATLAB* based approaches use a hybrid approach of operator overloading and source code transformation which is discussed below.

Source Code Transformation

In this method, the original source code is interpreted as a symbolic expression. The derivatives are calculated and written to code in the original programming language. Apart from the generation, there are no dependencies on external libraries during runtime. The code containing no derivatives is transformed to a code which includes them. Since both the function and the derivative calculation are evaluated at the same time, source code transformation has a high code optimization potential. Additionally, as there are no operator overloading classes, the compiler can perform code optimization as well.

MATLAB itself offers a source code transformation tool with the Symbolic Math Toolbox. Instead of numeric values, symbolic variables which can be used to call a *MATLAB* function are created. Thus, the function output becomes a symbolic expression. Afterwards, the first and second order derivatives can be calculated using the *MATLAB* commands `jacobian` and `hessian`. As a result, the original function and its derivatives are available as symbolic expressions in the *MATLAB* workspace. The command `matlabFunction` writes these expressions into a source file. Listing 4.7 gives a code example for a simple case. The resulting *MATLAB* file contains optimized code and is no longer "human friendly" to read.

Although this method offers analytic derivatives in pure *MATLAB* code it has some limitations and drawbacks. The functions to be transformed must be differentiable. This means it must not contain any conditional statements (`if`, `else` or `switch`), loops (`for` or `while`) and other discontinuities. However, these are the same requirements that need to be fulfilled for general optimal control problems.

The actual drawback of this approach is the amount of function complexity that can be handled. While writing the symbolic expression into code *MATLAB* tries to optimize the output. For complex functions, e.g. the aircraft model of section 6.1, this process takes impractically long (approximately 9 hours). The generation time can be significantly reduced to a 10-20 minutes by setting the optimization option to `false` (see Listing 4.7). However, for the aircraft dynamics introduced in section 6.1 the generated code is faulty and produces imaginary outputs where the source function returns the true results. Therefore, the symbolic source code transformation can only be

Listing 4.7: Example Code Analytic Gradients with Symbolic Toolbox

```
% Create symbolic variables
x = sym('x', [5,1]);
y = sym('y', [5,1]);

% Call source function to get symbolic expression
f = func(x,y);

% Create vector of independent variables
z = [x;y];
% ... and create symbolic derivatives
j_z = jacobian(f,z);
h_z = hessian(f,z);

% Write to matlab file
matlabFunction(f, j_z, h_z, 'Vars', {x, y},...
    'File', 'gfunc', 'Optimize', true)
```

applied to functions of manageable complexity.

Hybrid Approaches

In *MATLAB*, the software packages A MATLAB Automatic Differentiation Tool (ADiGator) [75] and Automatisches Differenzieren für Matlab (ADiMat) [76] both use an operator overloading approach to generate a source code transformed output file. It is a *MATLAB* function implementing the derivatives. Whereas ADiGator is open source, ADiMat requires a connection to a code transformation server. Although both approaches implement source code transformation, *MATLAB* features are required that are not supported by code generation (sparse matrices, cell arrays, anonymous functions). Therefore, these approaches currently cannot be sped up through compilation.

Practical Requirements for Optimal Control

All automatic derivative methods presented above have benefits and drawbacks. In the following, a list of requirements to the derivative calculation for optimal control problems in *MATLAB* is given:

- Analytic derivatives of the source functions for first and second order must be created. Additionally, the derivatives shall be transformed into *MATLAB* code and implement the source function and derivatives.
- For fast evaluation, the transformed code shall be compiled to a MEX file. Therefore, any dependencies to code that is not supported by the code generation algorithm of *MATLAB* Coder must not appear.
- Multiple input and output arguments, which may have matrix dimension, must be supported. For each input, it can be defined whether it is an independent variable (w.r.t. which derivatives need to be calculated) or a constant.

- The source function provided by the user (e.g. dynamic model) evaluates a single time step only. Vectorized evaluation of multiple steps is not required. Especially if matrix calculations are involved, vectorized evaluations become three dimensional and thus difficult to handle. However, the generated function with derivatives must be able to handle multiple time steps due to the nature of the collocation method.

- Analytic gradients for high fidelity models must be supported. The generation of the derivatives must be successful on a common consumer PC within reasonable time (e.g. within a few minutes).

- Multi-Threading for multiple time evaluations should be supported since all common consumer PCs have multiple CPUs available.

- The Outer Convexification approach shall be supported by the model generation approach, as it is needed for this thesis.

4.5.2 Subsystem Derivative Method

As discussed in the previous section, it is desired to obtain the analytic derivatives of a source function / model using source code transformation. In *FALCON.m*, this transformation is achieved with the derivative builders introduced in 4.3.3. Complicated user functions can be differentiated using the subsystem mode. In this section, the underlying algorithm is explained in more detail.

As mentioned, every complex system can usually be divided into multiple simple subsystems [77]. Although the overall user-function is non-trivial, local derivatives for individual subsystems are in most cases simple enough to be calculated by the source code transformation. Since the connection of the subsystem is known, the derivatives w.r.t. the user function inputs are calculated by applying the chain rule. Thus, an overall function that calculates the 1^{st} and 2^{nd} order derivatives can be created. Afterwards, the function is compiled to a MEX file for fast execution in *MATLAB*. The derivative generation process consists of two steps:

1. Create Derivative Function
   (a) Division into subsystems (manually by user)
   (b) Subsystem local gradient calculation using source code transformation
   (c) Construction of function derivatives using chain rule (1st and 2nd order)
2. Create Evaluation Function (e.g. support for collocation requirements)
   (a) Wrap derivative function for e.g. multiple time evaluations
   (b) Mex compilation for fast / multi-core execution

This method can be applied to all user supplied functions of an optimal control problem where gradients are required (e.g. models, constraints, and cost functions). In the following, the method is presented with the dynamic model in mind.

Subsystem Derivatives

The first step of the subsystem derivative method is the differentiation of the individual subsystems w.r.t. their inputs. All subsystems must be differentiable in real space. This means that loops, conditions, or any logical assignments are not allowed.

A subsystem may contain multiple inputs and outputs which at the same time can be a scalar, a vector, or a matrix. Thus, the derivatives of the subsystem outputs have to be calculated w.r.t. multiple inputs. At the same time, a matrix differentiated by another matrix is also very common. Therefore, the structure of the derivatives has to be defined. In [115], multiple definitions are discussed. However, it is argued that only one definition is suitable. It is used in this thesis and described in the following.

For the definition of the derivatives, the following notation is used [115]. An $m \times n$ matrix contains m rows and n columns. Thus, the size of a real matrix is given by $A \in \mathbb{R}^{m \times n}$. The transposed of a matrix A is given by A^T. The size of a vector $\vec{v}$ is given by $\vec{v} \in \mathbb{R}^n$ which is analog to $\vec{v} \in \mathbb{R}^{n \times 1}$. The identity matrix of size $n \times n$ is given by I_n or $I_{n \times n}$. Similarly, a zero matrix is given by 0_n or $0_{n \times n}$. Finally, if A is a $m \times n$ matrix, $\mathrm{vec}(A)$ stacks the columns of A to create a $mn \times 1$ vector.

In order to understand the matrix by matrix derivative, a simple example is stated. Assume ϕ is a differentiable scalar function and $\vec{x}$ a vector of size $n \times 1$. Then, the derivative

$$D\phi(\vec{x}) = \frac{\partial \phi(\vec{x})}{\partial \vec{x}^T} \tag{4.72}$$

is a row vector of $1 \times n$. If ϕ is replaced by a vector function $\vec{f}(\vec{x})$ of size $m \times 1$, the derivatives of the elements of $\vec{f}$ can be stacked vertically

$$D\vec{f}(\vec{x}) = \begin{pmatrix} Df_1(\vec{x}) \\ \vdots \\ Df_m(\vec{x}) \end{pmatrix} = \frac{\partial \vec{f}(\vec{x})}{\partial \vec{x}^T} \tag{4.73}$$

to create the Jacobian matrix of $\vec{f}$. This concept can be further generalized

$$F(X), \qquad F \in \mathbb{R}^{m \times p}, \qquad X \in \mathbb{R}^{n \times q} \tag{4.74}$$

for a matrix function F of a matrix variable X. In this case, the derivative

$$DF(X) = \frac{\partial \ \mathrm{vec}\,(F(X))}{\partial \left(\mathrm{vec}\,(X)\right)^T} \tag{4.75}$$

can be expressed by a $mp \times nq$ Jacobian matrix. It can be seen that the $\mathrm{vec}()$ command is used to transform the function (4.74) to a vector function of a vector argument. Thus, the derivative definition of (4.73) can be used.

Apart from the first order derivatives, the second order derivatives (Hessian) have to be defined. Using the definition from the scalar function $\phi(\vec{x})$ above, the Hessian

$$D^2\phi(\vec{x}) = \frac{\partial^2 \phi(\vec{x})}{\partial (\vec{x}^T)^2} \tag{4.76}$$

is a $n \times n$ matrix. In case of the vectorized function $\vec{f}(\vec{x})$ the similar stacking of the derivatives

$$D^2\vec{f}(\vec{x}) = \begin{pmatrix} D^2 f_1(\vec{x}) \\ \vdots \\ D^2 f_m(\vec{x}) \end{pmatrix} = \frac{\partial^2 \vec{f}(\vec{x})}{\partial (\vec{x}^T)^2} \tag{4.77}$$

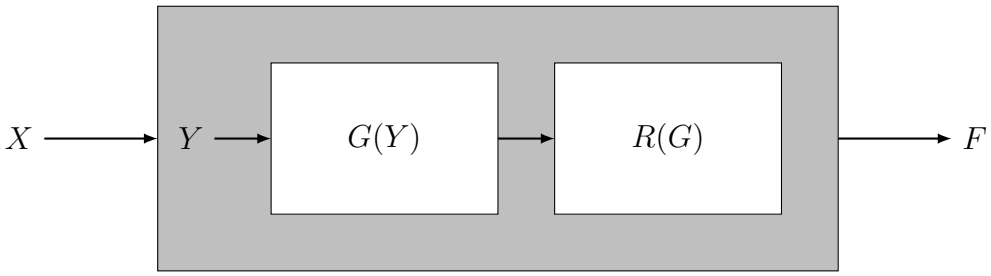


Figure 4.17: Example subsystem used for explanation of chain rule connection.

can be applied resulting in a $mn \times n$ Hessian matrix. Finally, the Hessian of (4.74)

$$D^2F(X) = \frac{\partial^2 \text{ vec}\left(F(X)\right)}{\partial \left(\left(\text{vec}\left(X\right)\right)^T\right)^2} \tag{4.78}$$

is an $mpnq \times nq$ matrix.

The Jacobian (4.75) and Hessian (4.78) above have been defined for functions with a single input and output argument. In general, subsystems may contain multiple input and output arguments which have to be accounted for in the derivative formulation.

In (4.75) and (4.78) the $\text{vec}()$ operator is used to convert matrices or row vectors into column vectors for the derivative definition. In case a function has multiple input arguments, the input column vectors can be stacked vertically to create a function with a single input. Thus, the derivative definitions above can be used. The same approach can be applied in case the subsystem has multiple output arguments. However, outputs of a subsystem may enter different subsystems. In this case, outputs and derivatives need to be split. Therefore, the Jacobian and Hessian are calculated for every output argument individually.

Although the subsystem derivatives can become large, the structure is now well defined. According to [115], this definition retains the mathematical meaning of the Jacobian and Hessian. More importantly, a suitable chain rule, which enables the connection of the derivatives in the next section, exists.

Subsytem Connection

The model is divided into multiple subsystems and the analytic derivatives are calculated w.r.t. the subsystem's input arguments. Therefore, after every call of a subsystem, the chain rule is applied to calculate the subsystem derivatives w.r.t. the model input arguments.

For the description of the chain rule, a matrix function and a single matrix input are assumed for model and subsystems (see Figure 4.17). By applying the definitions above, all matrices are transformed into a vector using the $\text{vec}()$ operator to formulate the derivatives. Multiple inputs can be included in the same way.

Let $F(X)$ define the overall dynamic model to be differentiated with $F(X) \in \mathbb{R}^{m_F \times n_F}$

and $X \in \mathbb{R}^{m_X \times n_X}$. The overall model Jacobian and Hessian are given by:

$$DF(X) = \frac{\partial \text{ vec}\,(F(X))}{\partial \text{ vec}\,(X)^T} \in \mathbb{R}^{m_F n_F \times m_X n_X}, \tag{4.79}$$

$$D^2F(X) = \frac{\partial^2 \text{ vec}\,(F(X))}{\partial \left(\text{ vec}\,(X)^T\right)^2} \in \mathbb{R}^{m_F n_F m_X n_X \times m_X n_X}. \tag{4.80}$$

Let $G(Y)$ define a subsystem within the model with $G(Y) \in \mathbb{R}^{m_G \times n_G}$ and $Y \in \mathbb{R}^{m_Y \times n_Y}$. The Jacobian and Hessian of the subsystem w.r.t. its inputs

$$j_G = DG(Y) = \frac{\partial \text{ vec}\,(G(Y))}{\partial \text{ vec}\,(Y)^T} \in \mathbb{R}^{m_G n_G \times m_Y n_Y} \tag{4.81}$$

$$h_G = D^2G(Y) = \frac{\partial^2 \text{ vec}\,(G(Y))}{\partial(\text{ vec}\,(Y)^T)^2} \in \mathbb{R}^{m_G n_G m_Y n_Y \times m_Y n_Y} \tag{4.82}$$

are represented by j_G and h_G respectively. The derivatives of the subsystem G with respect to the model inputs

$$\tilde{j}_G = DG(X) = \frac{\partial \text{ vec}\,(G(X))}{\partial \text{ vec}\,(X)^T} \in \mathbb{R}^{m_G n_G \times m_X n_X} \tag{4.83}$$

$$\tilde{h}_G = D^2G(X) = \frac{\partial^2 \text{ vec}\,(G(X))}{\partial(\text{ vec}\,(X)^T)^2} \in \mathbb{R}^{m_G n_G m_X n_X \times m_X n_X} \tag{4.84}$$

are assumed to be known and expressed by a $\tilde{\square}$.

Let $R(G) \in \mathbb{R}^{m_R \times n_R}$ be a subsystem with local derivatives

$$j_R = DR(G) = \frac{\partial \text{ vec}\,(R(G))}{\partial \text{ vec}\,(G)^T} \in \mathbb{R}^{m_R n_R \times m_G n_G} \tag{4.85}$$

$$h_R = D^2R(G) = \frac{\partial^2 \text{ vec}\,(R(G))}{\partial(\text{ vec}\,(G)^T)^2} \in \mathbb{R}^{m_R n_R m_G n_G \times m_G n_G} \tag{4.86}$$

that is dependent on the output of subsystem $G(Y)$. Overall, it is assumed that $\tilde{j}_G$, $\tilde{h}_G$, j_R, and h_R are known. From [115], the chain rule for the Jacobian

$$\tilde{j}_R = j_R \cdot \tilde{j}_G \tag{4.87}$$

$$= DR(X) = \frac{\partial \text{ vec}\,(R(X))}{\partial \text{ vec}\,(X)^T} \in \mathbb{R}^{m_R n_R \times m_X n_X} \tag{4.88}$$

maps the derivatives of subsystem R to the model inputs X. To obtain the chain rule for the Hessian

$$\tilde{h}_R = \left(I_{m_R n_R} \otimes \tilde{j}_G^T\right) \cdot h_R \cdot \tilde{j}_G + (j_R \otimes I_{m_X n_X}) \cdot \tilde{h}_G \tag{4.89}$$

$$= D^2R(X) = \frac{\partial^2 \text{ vec}\,(R(X))}{\partial(\text{ vec}\,(X)^T)^2} \in \mathbb{R}^{m_R n_R m_X n_X \times m_X n_X} \tag{4.90}$$

(4.87) is differentiated w.r.t. the model inputs. The derivation of this chain rule is explained in section B.1.

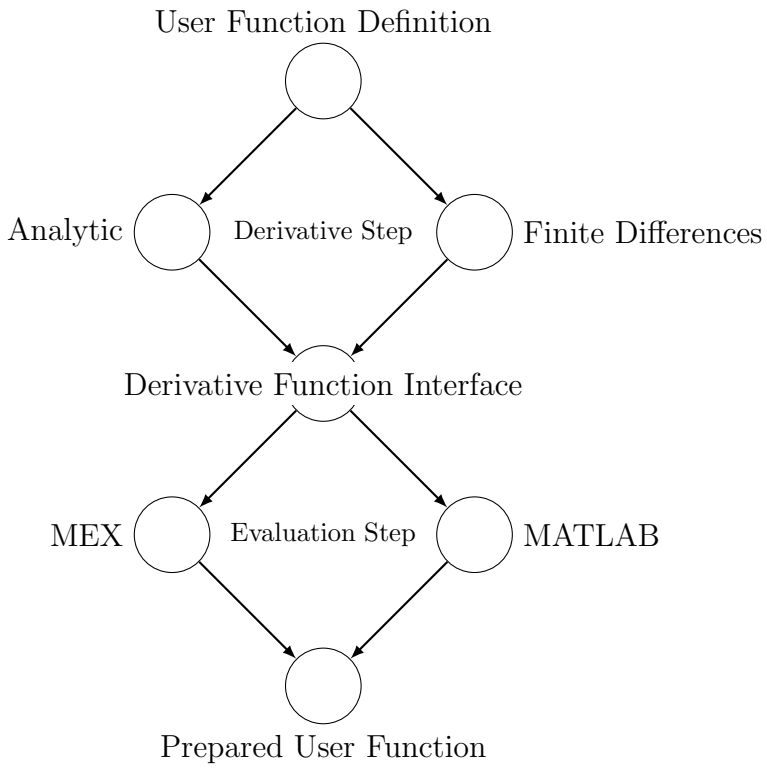


Figure 4.18: Workflow of the subsystem derivative builder.

As can be seen from the chain rule, the derivatives of subsystem G that enter the subsystem R are already given w.r.t. the model inputs. Therefore, after the chain rule update is calculated, the output derivatives of the subsystem R are given w.r.t. the model inputs as well. This ensures that the overall derivative calculation remains relatively simple. If this process is automated, the derivatives of the model inputs w.r.t. themselves are required. These are identity and zero matrices

$$\tilde{j}_X = j_X = I_{m_X n_X}, \qquad \tilde{h}_X = h_X = 0_{m_X n_X \times n_X} \tag{4.91}$$

of suitable size.

4.5.3 Implementation

The subsystem derivative method is implemented in *MATLAB* classes that fully automate the derivative calculation process. The algorithms require the toolboxes *MATLAB*, Symbolic Math Toolbox (for derivative calculation), and *MATLAB* Coder (for MEX file generation). Additionally, a supported C++ Compiler has to be installed on the machine (e.g. Visual Studio). In case a toolbox is missing, a compatibility workflow is provided. It uses finite differences and multiple time evaluation in *MATLAB*.

Fig. 4.18 depicts the schematic work-flow of the Subsystem Derivative Builder. The process can be divided into five main steps. As before, the process can be applied to constraints and cost functions as well. For simplicity, the description is given with a dynamic model in mind.

User Function Definition This is the only step that is visible to the user who has to provide a build script. In *FALCON.m* this step is wrapped by the builder in-

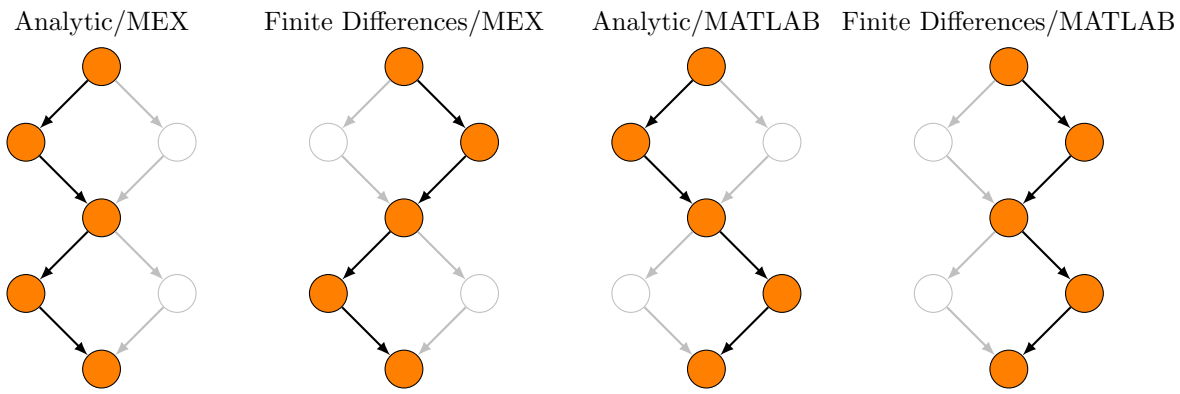


Figure 4.19: Possible workflow paths of the subsystem derivative builder.

stances (see section 4.3). Name, inputs and outputs of the model must be stated. All subsystems are defined in the order of appearance as well as their interconnection. This is referred to as the Build Trace (BT) which is explained further below. The BT does not only store the order of subsystems, but also split and combine commands (see section 4.3.2).

Derivative Step Generates the gradients for the dynamic model. Here, the Subsystem Derivative (SD) method introduced in section 4.5.2 is used. This step requires the Symbolic Math Toolbox. For compatibility, a FFD implementation is provided.

Derivative Function Interface Output of the derivative step is a function that implements the model as well as its derivatives. It is implemented in pure *MATLAB* code without any external dependencies and supports code generation. However, at this state, the values and derivatives are calculated for a single evaluation in time.

Evaluation Step Wraps the derivative function to allow multiple time evaluations for the collocation method. The model is coded to C++ and compiled to MEX using *MATLAB* Coder and a supported compiler. A *MATLAB* wrapper is used in compatibility mode.

Prepared User Function Final result of the subsystem derivative builder work-flow and "ready-to-use" for optimization in *FALCON.m*.

Due to the fact that the Derivative Function Interface is a common intermediate step in the work-flow, there are four possible ways a user function can be prepared for evaluation in *FALCON.m* (see Figure 4.19). A performance benchmark of the different combinations is done in section 4.5.4. In the following, the steps of the derivative generation process are discussed in more detail. All classes and functions described below are found in the namespace `falcon.core.builder`. To allow for a better readability, the namespace is omitted in the following text.

User Function Definition

In *FALCON.m*, the derivative builder classes are used to differentiate the user functions. Figure 4.20 shows the builder classes with their parent class. The usage of the

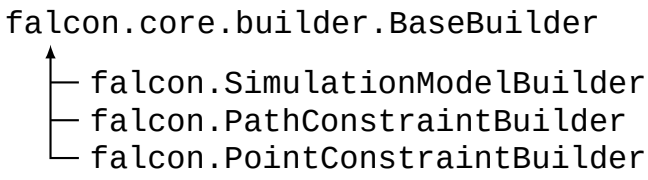


Figure 4.20: User function derivative builder classes and their parent class.

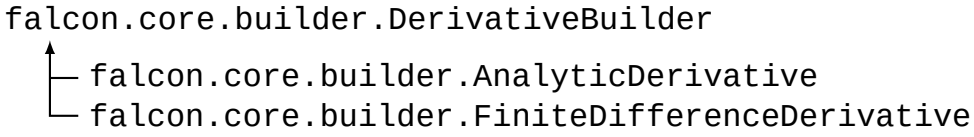


Figure 4.21: Derivative builder classes for analytic and finite differences generation together with parent class.

FALCON.m derivative builders is explained in section 4.3. Internally, the builder classes do not perform any differentiation. Instead they call the SDB instance that carries out the actual calculations. The SDB is written in a generic form and does not require any information concerning the user function involved.

Derivative Step

Figure 4.21 shows the classes that define the derivative generation algorithm. The class `AnalyticDerivative` handles the SD method and transforms the subsystems into *MATLAB* functions with derivatives. Finite differences are provided by the class `FiniteDifferenceDerivative`, which does not generate any derivatives for the subsystems, but ensures that the overall model fulfills code generation requirements. For instance, this requires anonymous functions to be transformed into "normal" *MATLAB* functions. The finite difference are calculate w.r.t. the overall model.

Both classes are inherited from `DerivativeBuilder` which implements shared functionality as inputs, outputs, and variable tracking. Within the class, all inputs, outputs, and intermediate variables are tracked using strings. In the following, an instance of the subsystem derivative builder is used to explain the underlying features. A new instance is created by calling the constructor

```
builder = falcon.core.builder.AnalyticDerivative(ProjectName);
builder = falcon.core.builder.FinitDifferenceDerivative(ProjectName);
```

where `ProjectName` is the name of the prepared function that shall be created.

Model Inputs The interface of the Derivative Function Interface and the final evaluation function is defined by the input and output arguments. Multiple input and output arguments are supported.

A new input is created using the method

```
builder.addInput(Name, VarDim, varargin)
```

which requires the name and the size (`VarDim`) of the input. The size can either be defined by a matrix (e.g. `[5,2]`) or by a cell array of strings (e.g. `{'x'; 'y'; 'z'}`).

The first case is used to specify a constant input, the second to define an input vector of *FALCON.m* data objects.

Setting `inf` in any of the dimensions will create a variable-sized input along the dimension specified. However, some limitations hold true in this case. They are explained below. In the latter case, the size of the cell array is used to determine the input size. Additionally, it is assumed that every entry is a scalar. Other settings for an input are the following:

DoDerivative In case this flag is set to true, the input is regarded as an independent input variable. Derivatives must be calculated w.r.t. this input. Otherwise, the input is regarded as a constant. The default value for this flag is `true`.

MutlipleTimeEval This information is relevant only for the evaluation step and states that this variable varies with time. States and controls have different values for different discretized times. For parameters and constants, the same variable is used for all points in time. The default value for this flag is `true`.

DiscreteControl Flag used in this thesis to specify a discrete control input. The input represents the actual discrete value used in the dynamics. In case this option is set on at least one input, the evaluation function interface is adapted (see section 4.3.3). The default value for this flag is `false`.

EntrySizes In case the input size is specified by a cell array of strings, all entries are assumed to be scalar. This option allows to specify the size of each entry individually. This feature is used by the point constraint builder to account for multiple time step inputs. The default input is empty and assumes scalar entry sizes.

The options allow for a very flexible input definition independent on the function type (simulation model, constraint, cost function). However, concerning the inputs there are some limitations.

- Variable sized data must be a constant and can neither be evaluated at multiple time steps nor be taken into account as a discrete control. The main reason for this is that the number of independent variables must remain constant. Additionally, the Symbolic Math Toolbox does not support variable sized data. Variable-sized constants can only be used with derivative subsystems. This is a drawback compared to other methods (e.g. operator overloading). However, the local derivatives are usually relatively simple and can be provided if required

- Inputs that are evaluated at multiple time steps must be a column vector. In the evaluation function these inputs are treated as a matrix

$$[\vec{v}_{t_1}, \vec{v}_{t_2}, \ldots, \vec{v}_{t_N}]$$

  where each column represents a time slice. The evaluation code can thus be kept more simple. Within the *FALCON.m* optimal control problem, time dependent inputs (e.g. controls) are column vectors anyway.

- Discrete controls must be a column vector since each column represents a discrete choice. Additionally, it must be a constant input and cannot have the multiple time step evaluation flag set to true. As stated above, a discrete control input represents the actual discrete value used in the dynamic model.

Model Outputs A new output is defined by the method

```
builder.addOutput(Name, EntryNames)
```

where only the name of the output (string, e.g. statesdot) and the names of the entries have to be provided (cell array of strings). The entries are concatenated in the two dimensional space. Thus, the overall variable size and derivative sparsity pattern (template sparsity) of the output is determined. Matrix type outputs are supported.

Input and output information is stored in the `InputTable` and the `OutputTable`. Additionally, the `AvailableVariableTable` exists.

Available Variable Table The `AvailableVariableTable` contains information about all variables that are currently available in the derivative generation process. Initially, this table is filled with the information of the model inputs. With every call of a subsystem or a split/combine command, the newly generated outputs are added to the table. The following information about the variables is stored:

Name String identifier of the variable.

Size Two dimensional size of variable `[m,n]`.

DoDerivative Flag that determines if the variable is dependent on any independent input variables. If this is the case, derivatives w.r.t. this variable are calculated as well.

Jacobian/Hessian Sparsity pattern of the Jacobian or Hessian. In case the Hessian is not calculated, the value is set to `nan`.

During the derivative generation, this table fulfills several objectives. Before a subsystem is called for derivative generation, the availability of the required input variables is tested. This check is performed by the `CheckInputs` method. Thus it can be assured that the resulting model remains feasible. Additionally, the size of the inputs is determined. Using the `DoDerivative` flag, the SDB determines the independent input variables of the subsystem. W.r.t. these variables, local derivatives are calculated. Finally, the sparsity patterns of the inputs are used to generate the sparsity patterns of the outputs. For this, the input sparsities and local sparsities are combined using the chain rule (see section 4.5.2).

Build Trace As was mentioned before, all subsystem and variable manipulations (split/combine) are stored in the build trace in the sequence they are added. Definitions of subsystems and variable manipulations on the subsystem derivative builder level work in the same way as described in section 4.3. In fact, the user function builders pass the information directly to the subsystem derivative builder instance without any changes.

Listing 4.8: Example call of differentiated subsystem and implementation of the Jacobian chain rule.

```
%% Call differentiated subsystem "subsys"
[a, b, j_a, j_b] = subsys(x,y,z);

% Gather input derivatives
j_subsys_inputs = vertcat(j_x, j_y, j_z);

% Perform chain rule (override local derivatives)
j_a = j_a * j_subsys_inputs;
j_b = j_b * j_subsys_inputs;
```

Once the build process is invoked, a *MATLAB* file is created. This file implements the derivative function interface. First, the function header as well as the initialization data are written. This includes the extraction of values from the inputs and the initialization of the input Jacobians and Hessians. Additionally, internal constants are written.

Afterwards, all entries of the build trace are evaluated in sequence. Within the derivative function file, the calls to the differentiated subsystems, chain rule calculations, as well as the variable manipulations are written.

Subsystem In case a new subsystem is added to the derivative function by the build trace, the following steps are conducted:

1. Gather subsystem input information and perform input consistency check using the method `CheckInputs`.
2. Generate the local derivatives of the subsystem using the Symbolic Math Toolbox. The differentiation is carried out by the `CreateGradient` function.
3. In the *MATLAB* file that implements the derivative function interface, add a call to the differentiated subsystem.
4. Gather the derivatives of the subsystem inputs in temporary variables. Afterwards, carry out the chain rules for Jacobian and Hessian.
5. Calculate the sparsity of the subsystem outputs.
6. Add the subsystem outputs to the list of available variables.

In Listing 4.8 an exemplary call to a differentiated subsystem as well as the chain rule for the Jacobian are shown. The Hessian chain rule is calculated in a similar manner. In case a derivative subsystem is added to the derivative builder instance, the second step is skipped.

SplitVariable and CombineVariables Variable manipulations do not require the generation of a derivative subsystem. However, as new variables are created, they have to be added to the table of available variables as well as to the *MATLAB* function

Listing 4.9: Call derivative subsystem and implement chain rule

```
%% Combine Variables
x = [a; b; c];

% Combine Jacobians
j_x = vertcat(j_a, j_b, j_c);

%% Split Variable
y = x(1:2,:);
z = x(3,:);

% Split Jacobians
j_y = j_x(1:2,:);
j_z = j_x(3,:);
```

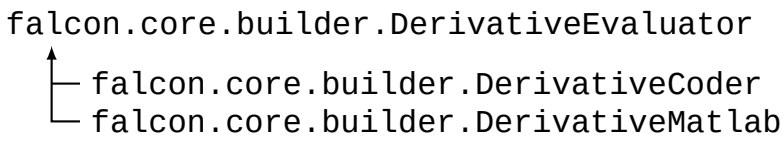


Figure 4.22: Classes for evaluation of user function derivatives.

that implements the derivative function interface. In Listing 4.9 an example implementation of the variable manipulations is given. Although the example is relatively simple, it states the general required steps.

Derivative Function Interface

After the build chain is evaluated, the output values and derivatives are collected. Internally, the `CombineVariables` method is used. The resulting function is written entirely in pure *MATLAB* code and supports code generation.

Listing 4.10: Derivative Model Function Interface

```
[x_dot, y, j_x_dot, j_y, h_x_dot, h_y] = model(x,u)
```

Listing 4.10 shows the interface of the generated derivative function for a model featuring states x, controls u, and additional model outputs y. The interface resembles that of the derivative subsystems. Therefore, it is possible to reuse a model within another subsystem derivative generation process, allowing e.g. to implement a hierarchy.

Evaluation Step

The file implementing the derivative function interface calculates the analytic derivatives for a dynamic model. However, the derivatives are implemented for the evaluation of a single time step. For direct optimal control using the collocation method, multiple independent time evaluations are required.

Since the derivative model function cannot handle vectorized evaluation, a loop has to be used. There are two ways the multiple time evaluation wrapper is generated.

First of all, the *MATLAB* function is coded into C/C++ code which is then compiled into a MEX file. This method is implemented by the `DerivativeCoder` class (see Figure 4.22). In case the *MATLAB* Coder Toolbox is not available, `DerivativeMatlab` implements a *MATLAB* wrapper for the multiple time evaluations. Both classes are derived from `DerivativeEvaluator` which implements common tasks.

For simple user functions, the *MATLAB* loop can be sped up by the just in time compiler of the *MATLAB* engine. In case a more complicated user function is used, this may not work and the performance becomes slow. Therefore, the only viable option is to compile the function to a MEX file.

Additionally to the `DerivativeCoder` class, the C/C++ code workflow path requires the class `DerivativeWrapper` which automatically creates a user function specific MEX file wrapper. It implements common tasks but specific wrapper aspects are implemented by `DerivativeWrapperCol` for the default collocation method and `DerivativeWrapperColDC` in case the Outer Convexification is required. The created MEX file wrapper is written in C++ and implements the multiple time evaluations with optional multi-threading. It is compiled to a MEX file using the `mex` command.

Code Generation

As written above, C/C++ code is generated from a *MATLAB* user function which in turn is wrapped with a custom wrapper for multiple time evaluations. Alternatively, it is possible to write the multiple time wrapper in *MATLAB* and use the `codegen` command with MEX file configuration to compile to MEX directly from *MATLAB*. However, this approach is not used for the following reasons:

- *MATLAB* code can be compiled with multi-threading if the code uses parfor loops and the openMP compiler option is set. However, for large dynamic models, the compilation process sometimes fails. In these cases, the openMP compiler option has to be deactivated. This problem does not present itself if the wrapper is custom build.
- Due to the fact that the prepared user function must handle an unknown number of time steps, certain dimensions (number of columns for states and controls) have to be variable sized. Since the *MATLAB* compiler does not know any conditions between inputs, the code cannot be fully optimized. For instance, the code generation algorithm does not know that the number of columns for states and controls have to be the same. Thus, potential overhead which reduces evaluation performance is generated.
- The number of threads used by openMP can be set individually.

Overall, a custom build MEX wrapper interface is better tailored to the optimal control application. Therefore, the derivative model function is transformed into C++ code using the `codegen` command and the `exe` coder configuration. From the default coder configuration, the following alterations are made. The target language is changed to C++ since the discrete control approach requires dynamically generated intermediate variables. As the `exe` file is not necessary, the generate code only flag is set to `true`. Support of non-finite numbers is set to false since it is not needed. Additionally, the generated code shall be implemented in a single C++ header (`*.h`) and code (`*.cpp`)

file. After the code is generated, a wrapper that implements the MEX interface is written.

Mex File Wrapper

The MEX file wrapper has several purposes. First, it implements the multiple time evaluation for the collocation approach. The framework `openMP` can be used to support multi-threading with minimal effort. Thus, fast evaluation speeds are achieved.

Second, the wrapper performs an input check to ensure that *MATLAB* does not crash due to an invalid call to the MEX file. The number of input arguments and their dimensions are checked. For all inputs that are evaluated at multiple time steps, the number of columns must match. The same holds true for the discrete inputs. In case an incorrect input is found a meaningful error message is thrown.

Finally, the info struct interface is implemented (see section 4.4.2). It is requested by calling the MEX with an output argument but without any input arguments.

After all input checks are passed, the MEX wrapper calls the C++ coded derivative function for all time steps and stores the returned values, Jacobian, and Hessian outputs of appropriate size. If the multi-threading option is set, the subsystem derivative builder will automatically implement the necessary openMP declarations. For the multi-threading evaluation, the number of threads used is set to be one less than available. If all available threads are used for the evaluation, the thread running *MATLAB* is slowed down. This decreases the overall performance.

In case the dynamic model incorporates discrete controls, the MEX wrapper is adapted automatically to include the Outer Convexification. This evaluation requires the `blas.h` library which is linked automatically. The C++ code of the derivative model function does not need to be adapted for this purpose.

After the MEX interface wrapper is written, it is compiled using the `mex` command. The resulting MEX file is ready to be used in *FALCON.m*.

4.5.4 Example and Performance Comparison

In this section, the performance of the subsystem derivative builder is evaluated. The SD approach is compared against other free public available automatic derivative methods. This thesis is concerned with the solution of optimal control methods in *MATLAB*. Therefore, only methods which can be interfaced from *MATLAB* directly are used. These include either *MATLAB* based or C/C++ approaches which can be compiled into a MEX file. Additionally, the different workflow pathes available in *FALCON.m* subsystem derivative builder are compared.

The result of a performance comparison depends on the model complexity involved. Thus, for a very simple model, the performance comparison does not resemble that of a high fidelity model. To address this, three models of increasing complexity are compared. The simple and moderately complex models are introduced in section B.2. The BADA Family 4 model introduced in section 6.1 is used as the high fidelity or large model. Additionally to the performance comparison, the different methods are compared w.r.t. the derivative construction time.

Table 4.1: Derivative generation time of different approaches in seconds.

	Simple [sec]	**Moderate** [sec]	**BADA4** [sec]
SYM OPT	4.16	148.27	-
SYM NO	2.28	126.43	-
ADOLC	2.77	2.96	9.44
ADEPT	2.71	2.96	4.27
FFD	1.74	2.00	2.74
CFD	1.84	1.94	2.66
FALCON.m	6.65	8.40	28.76

Performance Comparison to Other Methods

The following methods are compared

- Direct source code transformation (analytic derivatives) by Symbolic Math Toolbox with optimization (SYM OPT) and without optimization (SYM NO)
- Operator overloading in C++ using Automatic Differentiation by OverLoading in C++ (ADOL-C) and Automatic Differentiation using Expression Templates (ADEPT)
- Numerical derivatives with Forward Finite Differences (FFD) and Central Finite Differences (CFD)
- Subsystem derivatives by *FALCON.m*.

All approaches are compiled to a MEX file since the code runs much faster than any similar *MATLAB* approach. For better comparability, all MEX files run on a single thread. For the approaches SYM OPT and SYM NO, source code transformation is applied to generate the derivatives in *MATLAB*. Similarly, the loops that calculate the finite difference approaches (FFD and CFD) are implemented in *MATLAB*. Afterwards, the *MATLAB* functions are coded to C++ code which is compiled to a MEX file using a custom MEX interface wrapper.

In the operator overloading approaches (ADOLC and ADEPT), the source model function is directly coded to C++ since the derivative calculation is implemented there. Again, the MEX file is compiled using a custom wrapper. Finally, the *FALCON.m* subsystem derivative approach model is generated with the toolbox described above.

Figure 4.23 and Table 4.1 shows the creation time for the different approaches[2]. In the figure the times are normalized to *FALCON.m* and the bars are displayed on a logarithmic scale. In the subsystem derivative approach the dynamic model is split into subsystems. Therefore, the Symbolic Math Toolbox must be called multiple times and the creation of the derivative model function in *MATLAB* requires some time. The finite difference and operator overloading approaches only require the code generation and compilation. Additionally, the compilation of the operator overloading approaches requires a slightly higher time since classes instead of floating point values are involved.

[2]Intel Core i5-4670 CPU @ 3.40GHz, Windows 10 64bit, MATLAB 2015a

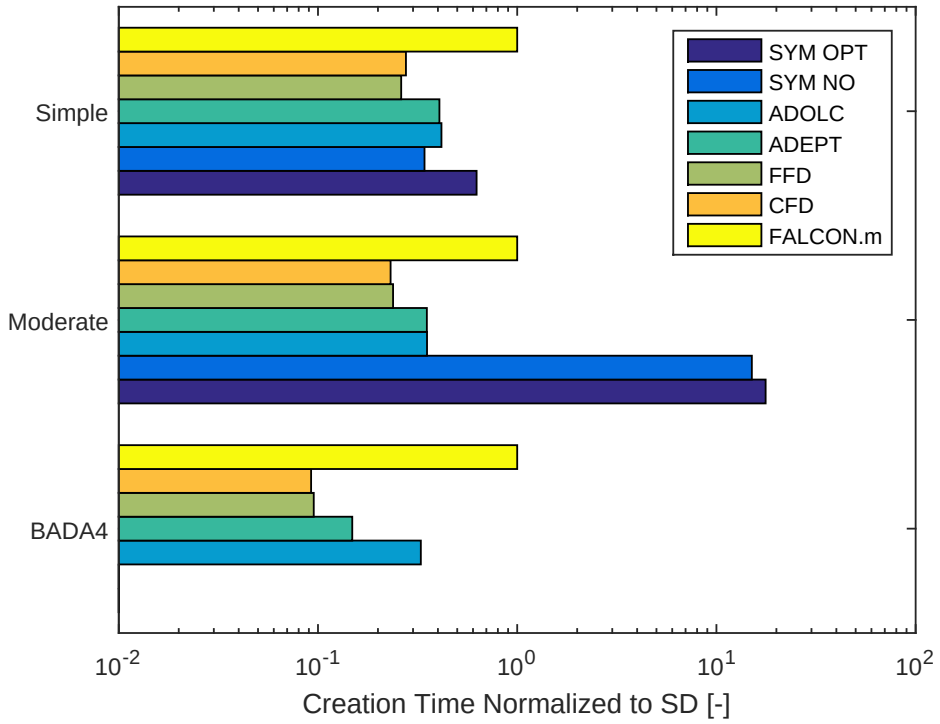


Figure 4.23: Comparison of derivative generation for different model complexities.

Comparing the SYM OPT and SYM NO approaches it can be seen that the code optimization results in a longer creation time. Additionally, with increasing complexity of the model, the derivative creation for the entire model becomes much slower compared to the other approaches. For the BADA4 model, the creation succeeds but the created *MATLAB* function produces complex results. Additionally, the calculation time for the SYM OPT approach is approximately 9 hours. In case of turned off code optimization only 15 minutes are required for the differentiation of the model, but the *MATLAB* file has a size of `15 MB` which is very large for a plain text file. Therefore, these results are not practical thus not shown.

Fig. 4.24 shows the performance comparison of the generated mex files. The test was carried out for 1 million evaluations. Table 4.2 shows the calculation times in seconds. It can be seen that for moderate and complex models *FALCON.m* is the fastest.

Table 4.2: Derivative calculation time for one million evaluations of different approaches in seconds.

	Simple [sec]	**Moderate [sec]**	**BADA4 [sec]**
SYM OPT	0.50	7.43	-
SYM NO	0.73	15.58	-
ADOLC	12.09	15.01	31.40
ADEPT	2.70	5.02	17.94
FFD	1.46	2.20	14.34
CFD	2.48	4.13	26.86
FALCON.m	0.84	1.67	5.27

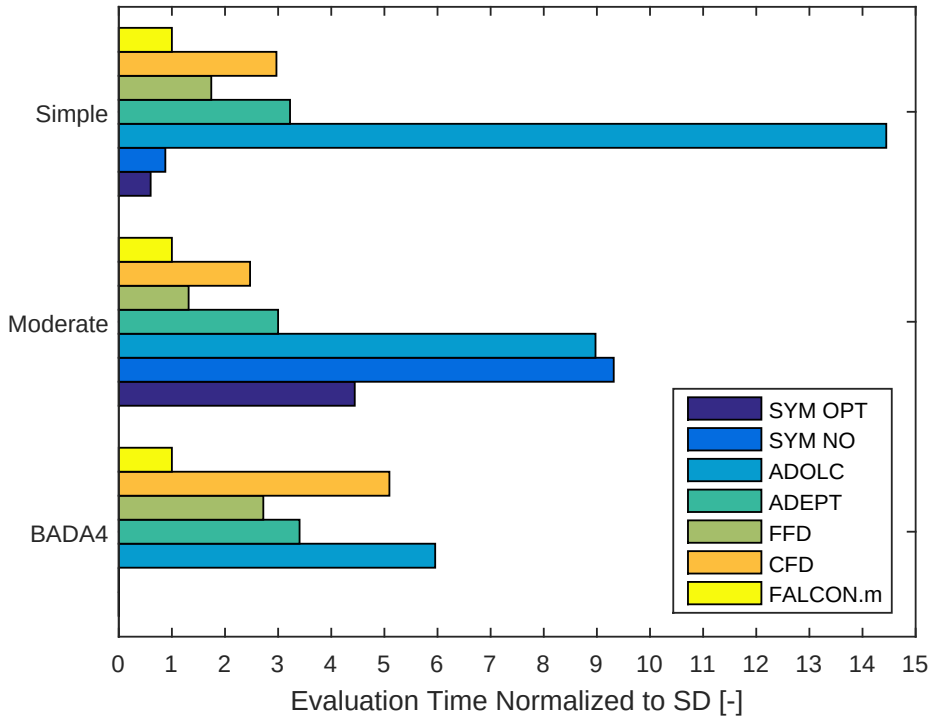


Figure 4.24: Evaluation comparison of different derivative approaches for one million evaluations.

The difference becomes more visible with increasing model complexity. For very simple models, *FALCON.m* is not the fastest approach. Compared to the symbolic approach, the model was divided into multiple subsystems. Therefore, the overhead created in the code reduces the performance. The whole model can be implemented as a single subsystem (function mode).

Comparing the Symbolic Math Toolbox approaches (SYM OPT and SYM NO), it can be seen that the *MATLAB* code optimization drastically improves the performance. However, for more complex models the performance is reduced. In these cases, it seems that *MATLAB* is unable to perform a suitable code optimization. Thus, the computational time increases.

The approaches which implement the operator overloading produce good results. Compared to *FALCON.m*, the computational time of ADEPT is slightly higher. A slower performance of the operator overloading approaches is expected since C++ classes instead of numeric values were used in the computation. Thus, the compiler is not able to perform code optimization. The implementation of ADEPT seems to be much faster than ADOL-C.

Finally, the finite difference approaches have a good performance as well. As expected, the forward finite difference approach is around twice as fast as the central finite difference approach.

Workflow Path Comparison

Here, the different model generation options of the *FALCON.m* subsystem derivative builder are compared. User functions can be differentiated either analytically or using

Table 4.3: Time for 100,000 evaluations in seconds of derivative builder workflow pathes.

	Simple [sec]	**Moderate** [sec]	**BADA4** [sec]
Analytic MEX (ANA/MEX)	0.09	0.17	0.46
Analytic MATLAB (ANA/MAT)	21.52	32.33	72.72
Finite Differences MATLAB (FFD/MAT)	80.75	133.17	273.08
Finite Differences MEX (FFD/MEX)	0.15	0.24	1.50

finite differences. The evaluation for multiple time steps can be achieved by using a *MATLAB* loop or by compilingn to a MEX file with a C/C++ loop. Thus, the performance is compared for four possible combinations.

The comparison is carried out with the same dynamic models as in the previous comparison. All prepared user functions are evaluated with 100k time points. Table 4.3 and Figure 4.25 show the evaluation times in seconds. As expected, compiled analytic derivatives are the fastest followed by compiled forward finite differences. The evaluation in *MATLAB* is much slower compared to the MEX file execution.

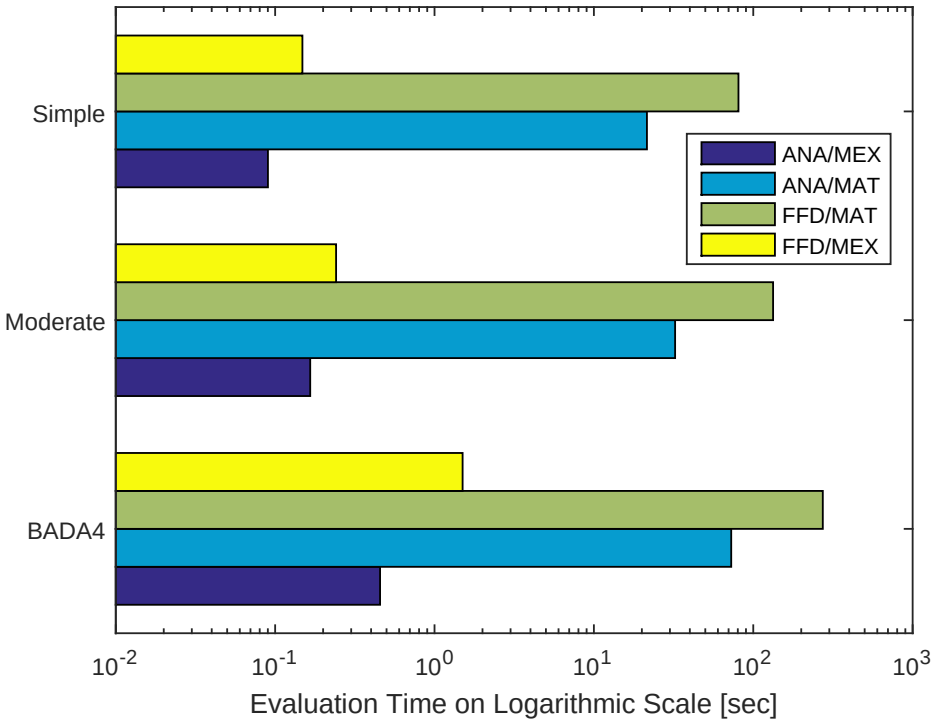


Figure 4.25: Evaluation comparison of derivative builder workflow paths.

4.6 Problem Derivative Calculation

The gradient based optimization algorithms that interface with *FALCON.m* require the derivatives of the optimal control problem. In this section, the algorithm that calculates the OCP Jacobian and Hessian from the user function derivatives is presented.

Both derivatives are calculated in an analytic way by assuming the the user functions provide analytic derivatives themselves. Although the implementation for large scale optimal control problems becomes complicated, the overall speed improvement is significant.

As was mentioned in chapter 2, fast and efficient derivative calculation in numeric optimal control methods is driven by three main factors:

- calculation of analytic derivatives,
- exact knowledge of the non-zero elements (sparsity),
- and implementation of the algorithms.

The calculation of the analytic derivatives is discussed in the previous section with the subsystem derivative builder. In this section, the calculation of the problem sparsity and the implementation are discussed.

As was stated by [24], the full exploitation of the problem sparsity down to the user function level has superior computational performance, especially for large optimal control problems. The subsystem derivative builder provides the user function template sparsity to *FALCON.m*. From the position of constraints and optimization variables in their respective vectors, block structures can be identified in the problem derivatives which are potentially non-zero (block sparsity, see Figure 4.26). Superposing the template sparsity to the block sparsity gives the actual non-zero elements of the constraint.

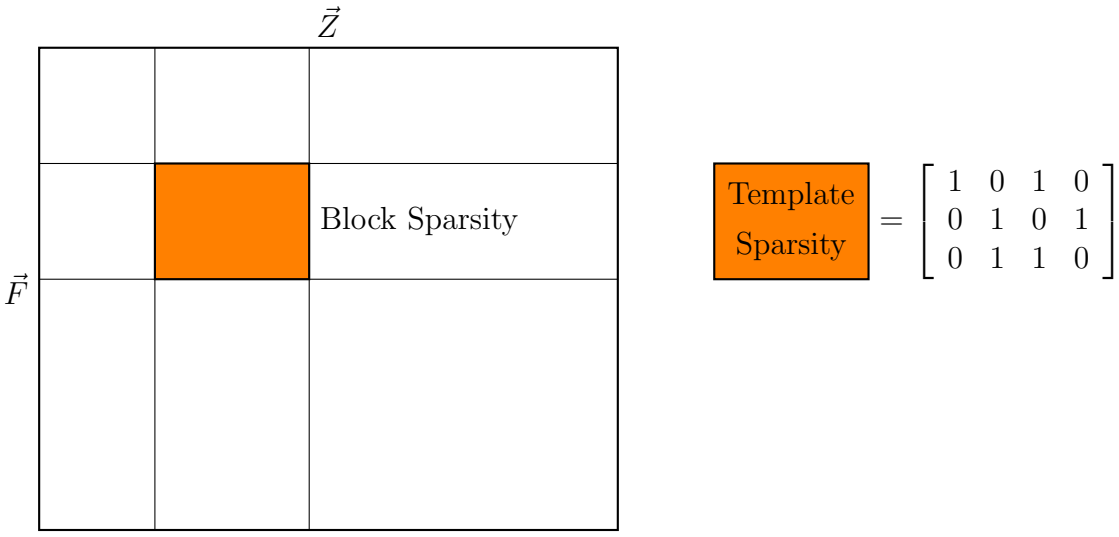


Figure 4.26: Block and template sparsity of Jacobian.

Efficient means are required to calculate the sparsity structure of the OCPs derivatives. This is especially important for large scale optimal control problems, as the sparsity cannot be constructed on a dense matrix. In *FALCON.m*, the non-zero positions in the derivatives are stored as row and column indices which are then used to construct a sparse matrix representation. In the following, the principles of the derivative calculations in *FALCON.m* are explained, followed by implementation aspects on the construction of the problem Jacobian and Hessian.

4.6.1 Direct Sparsity Sorting

The evaluation of the OCP's Jacobian and Hessian is highly time consuming. Since the derivatives have to be evaluated in every iteration at least once, their calculation is a major driver of the overall optimization performance.

All optimizers which currently interface with *FALCON.m* require the derivatives to be returned as a sparse matrix in the row column value format (rcv). In this storage type, non-zero elements are stored in three vectors (row index, column index, value). During optimization, both index vectors remain constant, but the value vector changes with every iteration. An efficient way to fill this value vector is provided by the direct sparsity sorting algorithm.

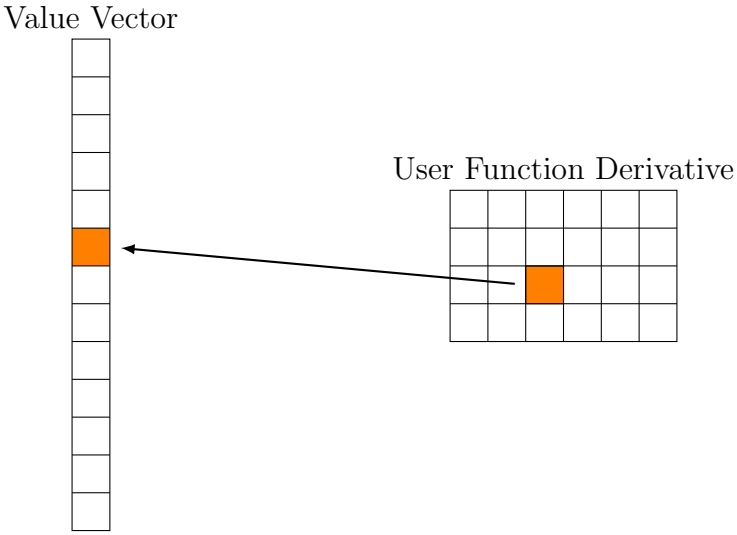


Figure 4.27: Idea of the direct sparsity sorting algorithm.

The idea is to sort the derivative values of the user functions directly at the correct position in the value vector (see Figure 4.27). Every value of the Jacobian and Hessian provided by the user function is indexed linearly. For every value, a second linear index determines where it has to be stored in the value vector. The whole derivative calculation is thus transformed into a linear indexing operation which executes very fast during runtime. This holds true for all constraints of the optimal control problem. In some cases, calculations have to be made (chain rule for model output dependencies, defect calculation). These are carried out in small and dense matrices. Overall, the method avoids large scale matrix evaluations and ensures a minimal memory consumption and computational overhead. For instance, the Jacobian of an optimal control problem solved in this thesis is spanned by $427,056$ optimization variables and $363,003$ constraints. Due to a very high sparsity, the number of non-zero elements is $4,782,909$ which translates to a memory consumption of merely 38.3 megabytes for the value vector (114.8 megabytes for the full rcv representation). In contrast, storing the full matrix would require 1155 gigabytes of memory.

In the following, the linear index construction for the Jacobian and Hessian sparsity sorting is explained. The algorithm is able to handle *FALCON.m* specific features such as:

- user function template sparsity,

- fixed optimization variables and inactive constraints,
- and multiple control grids with independent discretization and interpolation methods.

The calculation is carried out with the path constraint in mind. The principles can be translated to collocation defects and point constraints. Aspects introduced in the Jacobian calculation are not repeated in the Hessian calculation.

4.6.2 Jacobian Calculation

In this section, the Jacobian calculation using the path constraint implementation is explained. As mentioned earlier in section 4.2.4, the inputs entering the path constraint

$$\vec{y}_c \subseteq \vec{y}, \qquad \vec{x}_c \subseteq \vec{x}, \qquad \vec{u}_c \subseteq \vec{u} \tag{4.92}$$

are a subset of the phase data. If a constraint is dependent on model outputs, the chain rule

$$\frac{\partial j_{\vec{g}}}{\partial [\vec{x}, \vec{u}, \vec{p}]} = \frac{\partial j_{\vec{g}}}{\partial \vec{y}_c} \cdot \frac{\partial \vec{y}_c}{\partial [\vec{x}, \vec{u}, \vec{p}]} \tag{4.93}$$

has to be applied. Thus, the path constraint is no longer purely dependent on the subset of the phase inputs but also on the phase data which the model outputs dependent on. Additionally, parameters entering the constraint may overlap with the model parameters or be exclusive. In the Jacobian calculation, the direct dependency on the path constraint inputs and the model dependency through outputs can be superposed. In the following, the direct dependency is shown. The application of the method to the result of the chain rule is analogous.

Linear Indexed Jacobian and Reduction

The implemented and differentiated user path function returns a three-dimensional Jacobian matrix (see. Figure 4.28). For the direct sparsity sorting algorithm, this matrix is indexed linearly. Afterwards, the matrix is split into dependency blocks for states, controls, and parameters entering the constraint. The linear index block for the output dependency can be omitted as the chain rule has to be applied first. For every block, the rows corresponding to inactive constraints are eliminated (see red lines). Dependencies to fixed controls or fixed parameters are removed as well.

Analogous to the linear index matrix, the sparsity structure of the Jacobian returned by the path constraint is split into dependency blocks as well. The sparsity structure of the model output dependency is retained as it is used to calculate the sparsity structure of the chain rule result. Inactive constraint rows and fixed optimization variable columns are removed.

State Dependency

In order to find the non-zero elements of the path constraint state dependencies in the overall problem Jacobian, the discretization indices of the states in the $\vec{z}$ vector and of

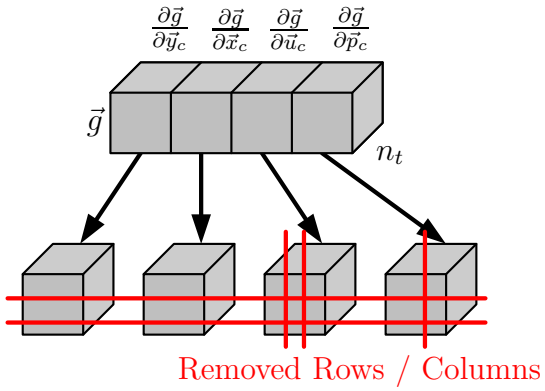


Figure 4.28: Linear indexing of the user function Jacobian.

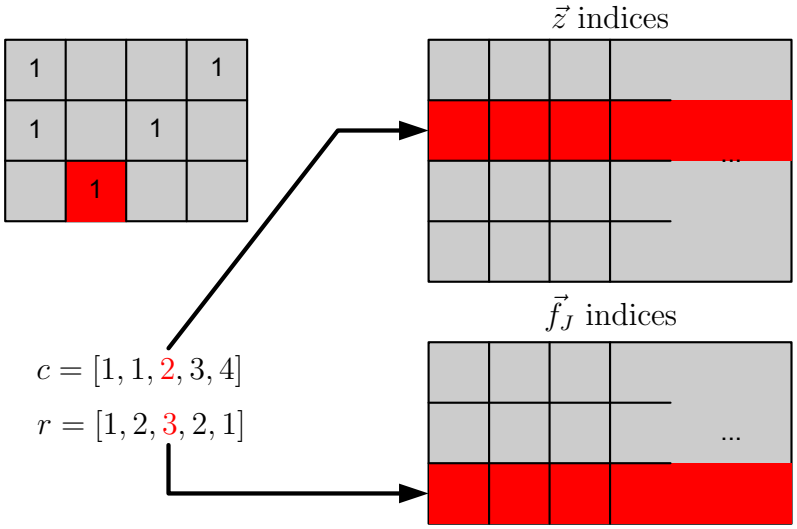


Figure 4.29: Extraction of the non-zero state dependency elements of the path constraint in the problem Jacobian.

the path constraint in the $\vec{f}_J$ vector must be extracted. Both index sets have already been calculated in the indexing step of the problem build process (see section 4.2.5).

Figure 4.29 shows an example sparsity pattern together with the indexing matrices. From the state template sparsity block, the row and column pairs representing non-zero entries are obtained. Using the resulting row and column vectors, the corresponding rows in the $\vec{f}_J$ and $\vec{z}$ index matrices are extracted. Thus, the non-zero entries of the state dependency in the problem Jacobian are calculated.

Additionally, the template sparsity is applied along the third dimension of the linear indexed user function Jacobian to find the indices which contribute to the non-zero elements. Therefore, for each linear index of the user function Jacobian, a row / column pair in the problem Jacobian exists.

Parameter Dependency

The parameter dependency calculation is analogous to the one of the states. The only difference is the fact that parameters are a vector of scalar values and not defined on a grid. Thus, the $\vec{z}$ vector indices have to be copied as often as the path constraint is evaluated.

Control Interpolation

Before the control dependency can be stated, further explanation on the control interpolation is required. As already mentioned, *FALCON.m* allows a user to define multiple control grids which may have a subset discretization w.r.t. the state grid. For the path constraint evaluation, all control grids have to supply control values on the state grid discretization. Thus, an interpolation method must be used. `Linear` or `previous` interpolation methods are currently supported. The interpolation maps the discretized controls in the optimization vector to the interpolated controls used in the constraints and model evaluations. All values, Jacobians, or Hessians returned by the user functions are mapped to the state grid discretization.

Dependent on the used interpolation scheme, the influence of the discretized controls on the interpolated controls changes. Additionally, in linear interpolation, the an interpolated control value is dependent on both neighboring discretized controls.

In order to store the mapping, for each control grid an interpolation gradient

$$\frac{\partial \eta}{\partial \eta_{\vec{z}}} \tag{4.94}$$

is created where η represents the state discretization and $\eta_{\vec{z}}$ the discretized controls. Figure 4.30 shows the control interpolation gradient for an example discretization situation for the linear and previous case. The rows represent the state grid discretization and the columns the discretized controls. The entries of the matrix give the weights each discretized control contributes to the interpolated controls. Other interpolation algorithms may be implemented as well, as long as their interpolation mapping is constant and not dependent on the actual discretized control values.

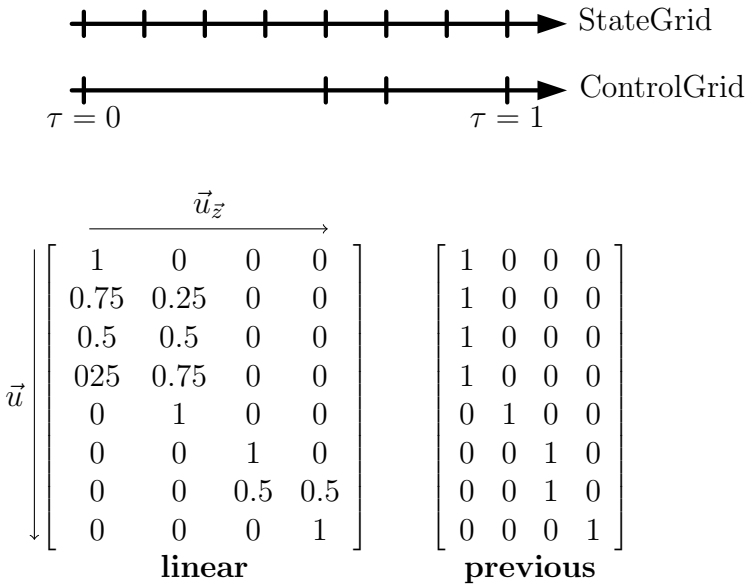


Figure 4.30: Interpolation of control grid values.

In the following, the interpolation gradient is used to extract the discretized controls for every time step in the control dependency calculation.

Control Dependency

The control dependency is calculated applying the chain rule

$$\frac{\partial \vec{g}}{\partial \vec{u}_{\vec{z}}} = \frac{\partial \vec{g}}{\partial \vec{u}} \cdot \frac{\partial \vec{u}}{\partial \vec{u}_{\vec{z}}} \tag{4.95}$$

to map the path constraint evaluations to the actual discretized controls. The matrix

$$\frac{\partial \vec{u}}{\partial \vec{u}_{\vec{z}}} = \begin{bmatrix} \frac{\partial u_1}{\partial u_{1,z}} & 0 & 0 \\ 0 & \frac{\partial u_2}{\partial u_{2,z}} & 0 \\ 0 & 0 & \ddots \end{bmatrix} \tag{4.96}$$

has a block diagonal structure. For each control at a certain time step, the specific non-zero interpolation values of the interpolation gradient are stored. These are the non-zero elements of a row in the interpolation gradient. In case a control is fully discretized or the current time step is a discretization time step of the control, the respective mapping is equal to one. Due to the independent discretization of the controls w.r.t. the states, this matrix cannot be assumed to be constant and has to be created for every discretized time step. As this calculation is done once during the construction of the OCP and not during the optimization loop, the runtime performance is not degraded.

Apart from the interpolation values, the $\vec{z}$ vector indices of the relevant discretized controls must be stored. Additionally, for each control, the number of discretized controls involved in the interpolation is stored.

Multiplying matrix (4.96) with the sparsity matrix of the control dependency gives a matrix which resembles the sparsity structure, but may contain fractional values from the control interpolation gradient. Using the number of discretized controls involved for each control, a block matrix structure can be extracted which has the same size as the original sparsity pattern.

In order to explain the last step in more detail, the example sparsity structure and discretized mapping control matrix

$$\left\{\frac{\partial \vec{g}}{\partial \vec{u}} \neq 0\right\} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}, \qquad \frac{\partial \vec{u}}{\partial \vec{u}_{\vec{z}}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0.2 & 0.8 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0.5 \end{bmatrix} \tag{4.97}$$

are assumed. Both the second and third control depend on two discretized controls. The multiplication results in fractional values

$$\frac{\partial \vec{g}}{\partial \vec{u}_{\vec{z}}} = \begin{bmatrix} 1 & 0.2 & 0.8 & 0.5 & 0.5 \\ 1 & 0 & 0 & 0.5 & 0.5 \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} & \begin{bmatrix} 0.2 & 0.8 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix} \end{bmatrix} \tag{4.98}$$

which can be rewritten as multiple block matrices concatenated together.

Similar to the methods before, the non-zero row and column indices for the template sparsity block are calculated. These are used to extract the respective row and column entries in the overall problem Jacobian. However, there are a few differences

- The index calculation must be carried out for every time step individually.

- As a control at a time step may depend on multiple discretized controls, it gets the same number of $\vec{z}$ indices assigned. Thus, the corresponding linear Jacobian indices and the $\vec{f}_J$ index must be copied.
- The discretized controls may have a partial influence at a current time step. Therefore, the direct sparsity sorting contains weights which are found in the concatenated block matrices (4.98).

Problem Sparsity Structure

Above, the method to calculate the non-zero elements of a path constraint was introduced. A similar method is applied to all other constraints, defects, and cost functions. The results for each are row / column index pairs that specify the non-zero elements in the problem Jacobian.

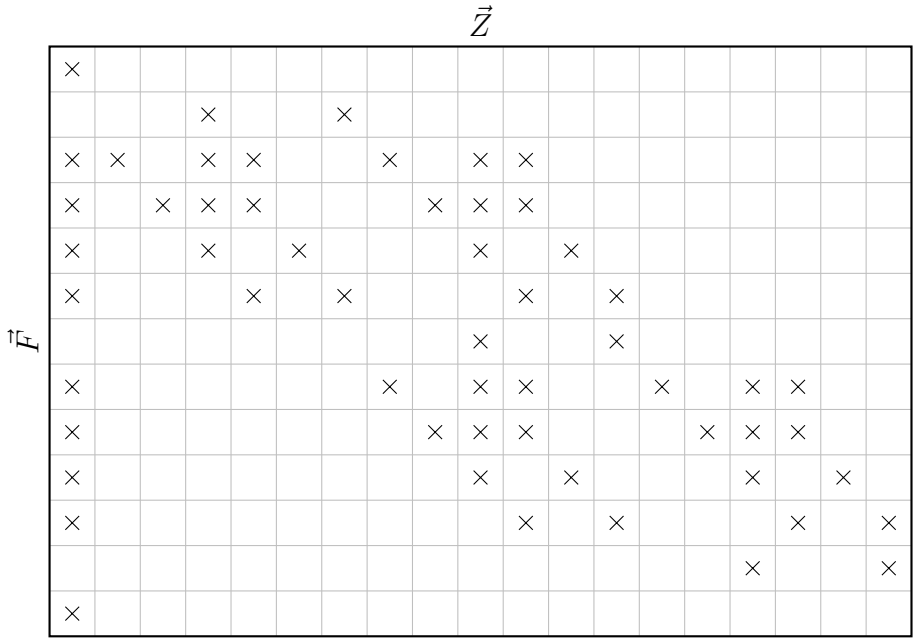


Figure 4.31: Problem Jacobian sparsity structure calculated by *FALCON.m*.

All non-zero entries combined give the overall sparsity structure (see Figure 4.31). Since the structure is known, a sparse matrix can be created. *MATLAB* automatically sorts and combines dublicate entries. From the sparse matrix representation, the sorted constant row and column vectors of the non-zero elements can be extracted. Figure 4.32 shows the extracted vectors for a small example matrix Jacobian. Additionally, all non-zero entries are linearly indexed. Each index represents the linear index in the value vector of the sparse matrix representation. Additionally, the row and column index vectors are extracted.

Value Vector Linear Mapping

At this point, the linear index of the user function Jacobian, the location of the corresponding non-zero elements in the problem Jacobian, and the overall problem sparsity structure are calculated. For the direct sparsity sorting, the linear index in the value

Value Vector Indices $\vec{Z}$

$\vec{F}$																		
1																		
				13			21											
2	11			14	17			23		27	34							
3			12	15	18				25	28	35							
4				16		20				29		40						
5					19		22				36		42					
										30			43					
6								24		31	37			45		47	51	
7									26	32	38				46	48	52	
8										33		41				49		54
9											39		44				53	
																50		
10																		

$$\text{Row-Indices} = \begin{bmatrix} 1 & 3 & 4 & 5 & 6 & 8 & 9 & 10 & 11 & 13 & 3 & 4 & 2 & 3 & 4 & 5 & 3 & 4 & \dots \end{bmatrix}$$
$$\text{Col-Indices} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 3 & 4 & 4 & 4 & 4 & 5 & 5 & \dots \end{bmatrix}$$

Figure 4.32: Problem Jacobian sparsity structure calculated by *FALCON.m*.

vector is still missing. In the global sparsity structure, all non-zero entries are linearly indexed. These indices resemble their linear index position in the value vector.

The indexed sparsity matrix is passed to every constraint and cost function in the *FALCON.m* OCP. Using their non-zero row and column indices, the linear index in the value vector is extracted. Thus, the two index sets required for the direct sparsity sorting method are created.

4.6.3 Hessian Calculation

The Hessian of the Lagrangian (2.37) is calculated using the sparsity sorting method. However, the generation of the indices is more difficult. While in the Jacobian all dependencies (w.r.t. states, controls,...) could be regarded as independent, this is no longer possible in the Hessian calculation. Here, the derivatives may be coupled between any two participating inputs. Additionally, the set of model parameters and constraint parameters must be unified. Finally, *FALCON.m* features such as multiple independent discretized control grids must be considered.

Due to the fact that in the Lagrange Hessian calculation all constraint Hessians are summed up, the scaling of Hessian has to be considered directly in the summation. The scaling cannot be imposed later by a matrix transformation. In order to reduce complexity, scaling is not considered in this explanation.

Mapping of Model Output Dependency Hessian to Common Representation

In case outputs are present in the path constraint, the chain rule for the Hessian has to be applied. For this reason, the chain rule function for the Hessian from 4.5 is used. Both the dynamic model and the path function are regarded as subsystems where results of the model enter the path function. Figure 4.33 depicts the information flow through both subsystems. For the Hessian structure calculation, the chain rule method is applied to the sparsity matrices. The same calculation applies for the actual computation during run-time.

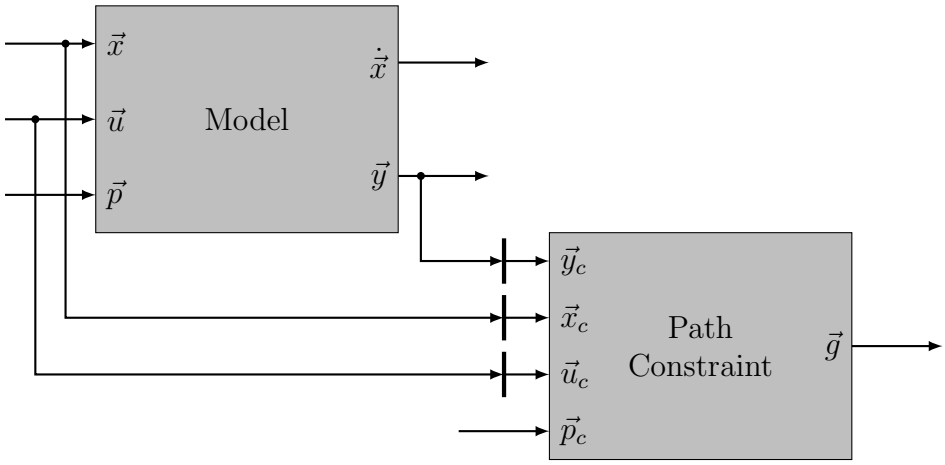


Figure 4.33: Reduced input set and output dependency of path function.

The chain rule formula is given by

$$h_{\vec{g}} = \left(I_{n_{\vec{g}}} \otimes j_{\vec{g}m}^T\right) \cdot h_{\vec{g}} \cdot j_{\vec{g}m} + \left(j_{\vec{g}} \otimes I_{n_{\vec{z}_m}}\right) \cdot h_{\vec{g}m} \tag{4.99}$$

where a $\tilde{\square}$ represents derivatives w.r.t. the model input states $\vec{x}$, controls $\vec{u}$, and the unique parameter set $\vec{p}_{gm}$. The combined vector of the overall input variables is represented by

$$\vec{z}_m = \begin{bmatrix} \vec{x} \\ \vec{u} \\ \vec{p}_{gm} \end{bmatrix}. \tag{4.100}$$

In the chain rule equation, $I_{n_{\vec{g}}}$ and $I_{n_{\vec{z}_m}}$ are identity matrices with the size number of path constraints $n_{\vec{g}}$ and number of combined vector variables $n_{\vec{z}_m}$. The Jacobian and Hessian of the path constraint w.r.t. their own inputs are given by $j_{\vec{g}}$ and $h_{\vec{g}}$ respectively. The Hessian chain rule calculates the Hessian $h_{\vec{g}}$ of the path constraint w.r.t. $\vec{z}_m$. Additionally, the matrices

$$j_{\vec{g}m} = \begin{bmatrix} \frac{\partial \vec{y}_c}{\partial \vec{x}} & \frac{\partial \vec{y}_c}{\partial \vec{u}} & \frac{\partial \vec{y}_c}{\partial \vec{p}_{gm}} \\ \frac{\partial \vec{x}_c}{\partial \vec{x}} & 0 & 0 \\ 0 & \frac{\partial \vec{u}_c}{\partial \vec{u}} & 0 \\ 0 & 0 & \frac{\partial \vec{p}_c}{\partial \vec{p}_{gm}} \end{bmatrix}, \qquad h_{\vec{g}m} = \begin{bmatrix} \frac{\partial^2 \vec{y}_c}{\partial \vec{z}_m^2} \\ \frac{\partial^2 \vec{x}_c}{\partial \vec{z}_m^2} \\ \frac{\partial^2 \vec{u}_c}{\partial \vec{z}_m^2} \\ \frac{\partial^2 \vec{p}_c}{\partial \vec{z}_m^2} \end{bmatrix} = \begin{bmatrix} \frac{\partial^2 \vec{y}_c}{\partial \vec{z}_m^2} \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{4.101}$$

represent the Jacobian and Hessian of the path constraint inputs w.r.t. $\vec{z}_m$. The first row of derivative blocks in both matrices represents the output derivatives of the dynamic

model. As the other constraint inputs (subset states, controls, and parameters) are a subset of the combined vector, the resulting block derivatives

$$\frac{\partial \vec{x}_c}{\partial \vec{x}}, \qquad \frac{\partial \vec{u}_c}{\partial \vec{u}}, \qquad \frac{\partial \vec{p}_c}{\partial \vec{p}_{gm}} \tag{4.102}$$

are trivial. Similar to the Jacobian the matrix $h_{\vec{g}}$ of the chain rule is indexed linearly. Inactive constraints as well as fixed inputs are removed.

Path Constraint Hessian Sparsity Structure

In the following, the unified and reduced Hessian matrix is used to calculate the non-zero row and column indices in the OCP's Lagrange Hessian. This requires the interpolated controls to be mapped to the discretized controls. Therefore, a similar mapping matrix to (4.96) is reused. As inputs may be coupled, the mapping has to be applied on the combined vector $\vec{z}_m$

$$\frac{\partial \vec{z}_m}{\partial \vec{z}_{m\vec{z}}} = \begin{bmatrix} I & 0 & 0 & 0 & 0 \\ 0 & \frac{\partial u_1}{\partial u_{1,z}} & 0 & 0 & 0 \\ 0 & 0 & \frac{\partial u_2}{\partial u_{2,z}} & 0 & 0 \\ 0 & 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & 0 & I \end{bmatrix} \tag{4.103}$$

where $\vec{z}_{m\vec{z}}$ represents the discretized optimization variables. The block diagonal matrix (4.96) is expanded by identity matrices of suitable size for the state and parameter dependencies. This matrix is not constant over time and has to be generated for every time step. Additionally, the indices in the $\vec{z}$ vector and the number of discretized controls must be stored.

Applying the discretized mapping matrix as a transformation to the chain rule sparsity

$$\left(\frac{\partial \vec{z}_m}{\partial \vec{z}_{m\vec{z}}}\right)^T \cdot \{h_{\vec{g}} \neq 0\} \cdot \frac{\partial \vec{z}_m}{\partial \vec{z}_{m\vec{z}}} \tag{4.104}$$

gives a weight matrix for the Hessian. It can be associated to the entries by dividing it into blocks associated to $h_{\vec{g}}$ (see equation (4.98)).

Finally, the non-zero row and column index pairs of the chain rule sparsity Hessian are extracted and used to find the corresponding row and column indices of the path constraint in the Hessian of the Lagrangian. The construction of the problem Hessian sparsity as well as the linear value vector mapping occur in the same way as for the Jacobian.

4.7 Discrete Controls in *FALCON.m*

This section explains the implementation aspects of the discrete controls in the *FALCON.m* optimal control toolbox. As presented in 4.3.3, the model derivative builder class supports the use of discrete control inputs using the outer convexification approach. The necessary discrete control weights w must be introduced as additional controls in the optimal control problem.

In order to reduce the internal complexity of the optimal control toolbox, the necessary augmentation of the optimal control problem is achieved by a discrete control extension during the `Bake` process. This approach ensures that the toolbox core structure remains unaffected. In the following, the extension idea is presented. Afterwards, the discrete control extension that allows simple access to optimal control problems with discrete controls is presented.

4.7.1 *FALCON.m* Extensions

The extension possibility of the *FALCON.m* optimal control toolbox shall allow a user to write custom augmentation of the optimal control problem. The extension possibilities are developed with the discrete controls feature in mind. Other extensions may be written in a similar way. Extensions can be added to a phase or the main problem.

Phase Extension

A phase extension must be inherited from the `falcon.ext.PhaseExtension` class. It is added to a phase using the

```
phase.addPhaseExtension(extension)
```

method. Multiple extensions may be added, but it has to be noted that the extensions are not instantiated by the phase itself. Augmentation changes to a phase must be performed in the `Bake` process before the phase conducts its consistency check. Therefore, the `OnPreCheckConsistency` method of the phase extension is called beforehand.

Problem Extension

A problem extension must be inherited from the `falcon.ext.ProblemExtension` class. It is added to a problem using the

```
problem.addProblemExtension(extension)
```

method. Tshe extension is not created by the problem itself. Multiple extensions may be added. The problem extension can augment the problem during the `Bake` process in the CheckConsistency call. It is called twice with the `OnPreCheckConsistency` and the `OnPostCheckConsistency` methods. In between those methods, the phase extensions perform their augmentation. Both methods are called before the consistency check of the problem.

4.7.2 Discrete Controls and Discrete Control Sets

The discrete control extension implements the Outer Convexification

$$\dot{\vec{x}} = \sum_k w_k \cdot \vec{f}\left(\vec{x}, \vec{u}, \vec{p}, \vec{v}_k\right), \qquad \sum_k w_k = 1 \tag{4.105}$$

approach presented in chapter 3. It supports a theoretical arbitrary number of independent discrete controls. The combinatory set (cartesian product) is calculated automatically (see section 3.5). Invalid combinations specified by the user are automatically removed. In the following, the creation of the discrete controls and their sets in *FALCON.m* is explained.

`falcon.ext.dc.DiscreteControl`

A discrete control is created using

```
dc = falcon.ext.dc.DiscreteControl(Name, CellSet);
dc = falcon.ext.dc.DiscreteControl(.., 'DiscreteSetNames',
   CellSetNames).
```

It requires the name of the discrete control input (acting as identifier) and the discrete control set as a cell array of numeric values. A discrete control must be a column vector. Additionally, the names for the entries of the set can be specified by a cell array of strings using a parameter input. The data specified in the constructor are available in the class as properties (see Figure 4.34).

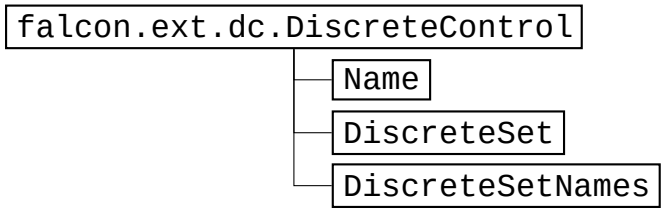


Figure 4.34: *FALCON.m* extension discrete dontrol class.

`falcon.ext.dc.DiscreteControlManager`

Handles mutliple discrete controls by calculating the valid combinatory set. It is instantiated

```
dcManager = falcon.ext.dc.DiscreteControlManager(dc1, dc2, ..);
```

with a list of discrete control objects. s The manager instance has the following prop-

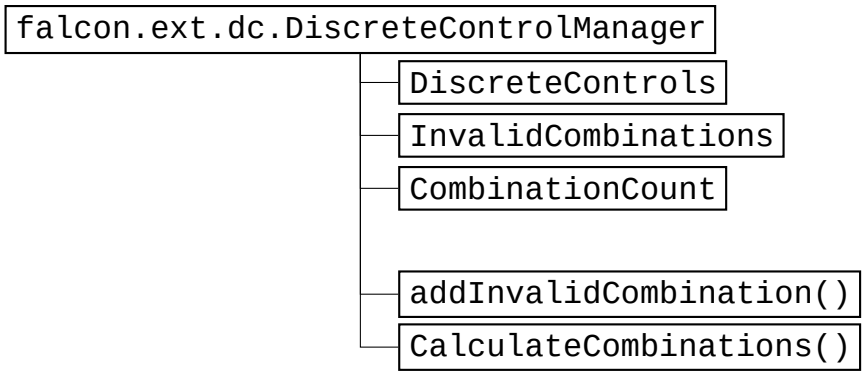


Figure 4.35: *FALCON.m* extension Discrete Control Manager class.

erties and methods (see Figure 4.35):

DiscreteControls Discrete controls instances are added to the manager using the constructor. All instances are stored in the `DiscreteControls` property.

CombinationCount / CalculateCombinations The manager calculates all possible combinations of the discrete controls. The number of valid combinations is stored in the `CombinationCount` property.

InvalidCombinations / addInvalidCombination Invalid discrete control combinations are specified using the `addInvalidCombination` method

```
dcManager.addInvalidCombination(...
    dcName1, dcSet1,...
    dcName2, dcSet2,..).
```

This requires the discrete control names and their relevant subsets that make up the invalid combination. The discrete control is identified by the object instance or the name. Its relevant subset is specified by the set names using a cell array of strings or the index of the discrete control value in the set.

An example helps to better understand the functionality of the discrete control set calculation. Two discrete controls

```
dcFlap = falcon.ext.dc.DiscreteControl('Flap', {0,1,2,3,4,5});
dcGear = falcon.ext.dc.DiscreteControl('Gear', {0,1});
```

are created that specify six flap positions and two landing gear positions. The discrete set values represent placeholders that have to be replaced by the actual values (e.g. aerodynamic parameters).

Both discrete controls are added to the manager class instance:

```
dcManager = falcon.ext.dc.DiscreteControlManager(dcFlap, dcGear);
```

The landing gear may only be deployed in the last two flap positions. Therefore, for all lower flap settings the invalid combination is added to the manager

```
dcManager.addInvalidCombination(dcFlap, 1:4, dcGear, 2)
dcManager.CalculateCombinations();
```

This must be done before all possible combinations are calculated. The invalid sets are automatically excluded. In the example, eight valid combinations are calculated.

4.7.3 Dynamic Model

If a dynamic model with discrete controls is created, the interface of the prepared user function

```
[xdot,y,jxdot,jy,hxdot,hy] = model(x,u,p,v1,v2,..,c1,c2,..,alpha,w)
```

is slightly different to the default case. The interface was already introduced in 4.3.3 and places the discrete control inputs between the parameters and constants. Additionally, two inputs are added at the end. The `alpha` input introduces additional zeros in the Jacobian and Hessian (e.g. to account for slack variables that are not considered by the dynamic model). The last input `w` expects the discrete control weights for the Outer Convexification. The number of columns for each discrete control and the number of rows of the discrete control weights must coincide. The number of columns for the discrete control weights must equal the number of time steps.

In order to consider the discrete controls in *FALCON.m* phase, the model dynamics need to be wrapped. The `falcon.ext.dc.PhaseExtension` class performs this task. It is inherited from the `falcon.ext.PhaseExtension` class. It is instantiated by

```
phaseExtension = falcon.ext.dc.PhaseExtension(dcManager);
```

where the discrete control manager `dcManager` needs to be passed to the constructor. The extension is added to the phase using the

```
phase.addPhaseExtension(phaseExtension);
```

method. The handle to the prepared dynamic model is added to the phase as usual. During the phase consistency check, the extension augments the phase. This changes are explained in the following.

Control Grids for Discrete Control Weights

Base on the information in the discrete control manager class, a new control grid for the discrete control weights is created. It is discretized with the same discretization density as the state grid. The last time step $\tau = 1$ is not included. The interpolation method is set to previous. A custom discretization can be set using the `setDiscretization` method.

An initial guess for the discrete controls can be set using the `setInitialGuess` method

```
phaseExtension.setInitialGuess(dc1, values1, dc2, values2, ..)
phaseExtension.setInitialGuess(tau, dc1, values1, ..)
phaseExtension.setInitialGuess(treal, dc1, values1, ..,...
    'RealTime', true)
```

which expects pairs of discrete control identifiers and their respective discrete value selection. If a time vector is specified, the number of discrete set values must match the time grid. Otherwise, the discrete control is assumed to be constant throughout the whole time interval. The initial guess is interpolated using the previous command. In case no initial guess is provided, all discrete control weights are initialized with a fraction of the discrete control combinations.

Slack Variables

Slack variables can be used by the Vanishing Constraints and the switching cost approaches to relax the overall optimal control problem. These require that additional controls are created, which are bypassed around the model dynamics. This bypassing is achieved by the `alpha` parameter in the prepared dynamic model function.

A new slack variable is requested from the phase extension by calling the

```
idx = phaseExtension.addNewSlackVariable();
```

method. It does not return a *FALCON.m* control object but the index of new control variable in the discrete control stack grid (`DiscreteControlSlack`) which is a property of the phase extension. All slack variables are bounded between 0 and 1.

Summation Constraint of Discrete Control Weights

In order to fulfill the discrete control weight summation constraint

$$\sum_k w_k = 1 \tag{4.106}$$

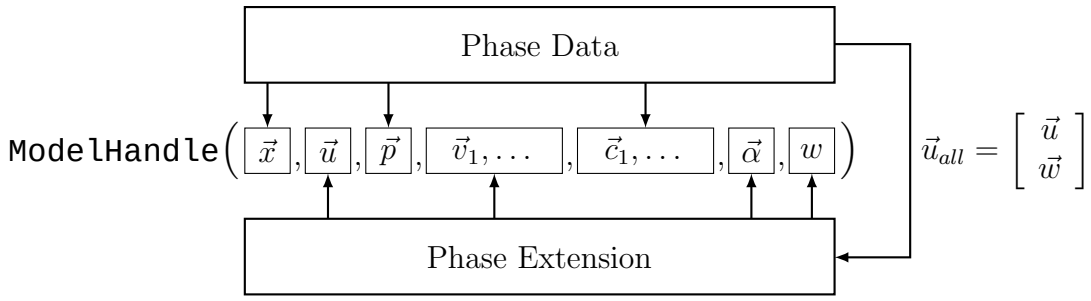


Figure 4.36: Information flow in the discrete control model wrapper.

a path function is added to the phase. It is evaluated on the same discretization grid as the discrete control weights. The "user-function" is handled by the internal `falcon.ext.dc.PhaseExtension.SummationConstraint` function which is similar to the implementation shown in 4.4.3.

Wrapping of Model Handle

In order to wrap the model dynamics for the *FALCON.m* toolbox, the handle to the model in the `Model` class is redirected to the `DiscreteModelHandle` method of the phase extension. The original handle to the prepared user function file of the derivative builder is stored.

During the consistency check, the model info struct is requested by *FALCON.m*. This request is handled by the wrapper function using the following alterations:

1. The number of controls expected by the model is increased by the number of valid discrete control combinations and slack variables.
2. All inputs that have the `DISCRETE` type are removed from the input list of the struct interface. *FALCON.m* is not able to handle the identifier for the discrete controls. Therefore, the discrete inputs are hidden from the toolbox. A consistency check for the discrete controls regarding the name and size is performed by the phase extension.
3. The Jacobian and Hessian sparsity structure is adapted to account for the discrete control weights and slack variables. As the number of valid combinations is unknown during derivative generation, this information cannot be stored in the info struct. The structure of the returned Jacobian and Hessian is explained in 4.4.

During evaluation, the information provided by the phase as well as the discrete control data is sorted and the original model handle is called (see Figure 4.36). The controls provided by the phase $\vec{u}_{all}$ are split into the continuous controls $\vec{u}$ and the discrete control weights w.

4.7.4 Vanishing Constraints

The Vanishing Constraints in *FALCON.m* offer a high flexibility and user friendliness. A path constraint

$$g_{LB} \leq g\left(\vec{y}_c, \vec{x}_c, \vec{u}_c, \vec{p}_c, \vec{c}, \ldots\right) \leq g_{UB} \tag{4.107}$$

is applied, only in case a certain discrete control choice $\vec{v}_k$ is selected (the corresponding weight w_k is greater than zero). From the user side, a simple path constraint has to be defined. Additionally, the active condition as well as the corresponding bounds have to be specified. The definition is transformed

$$w_k \cdot g(\vec{x}, \vec{u}, \vec{p}, \vec{v}_k) \leq 0, \qquad w_k \geq 0, \quad g(\vec{x}, \vec{u}, \vec{p}, \vec{v}_k) \leq 0 \tag{4.108}$$

to the vanishing constraint. Internally, the relaxed

$$w_k \cdot (g(\vec{x}, \vec{u}, \vec{p}, \vec{v}_k) + \kappa) - \kappa \leq 0 \tag{4.109}$$

or the reformulation

$$\frac{1}{2}\left(gw_k + \sqrt{g^2 w_k^2 + \kappa^2} + \sqrt{w_k^2 + \kappa^2} - w_k\right) - \kappa \leq 0, \qquad \kappa \geq 0 \tag{4.110}$$

approach are used. The approach type can be set using the

```
vanishingConstraint.setType(type)
```

method that expects a string (`RELAXED` or `REFORMULATED`).

A new Vanishing Constraint is added to the phase using the

```
vanishingConstraint =
   phaseExtension.addNewVanishingConstraint(handle, normalizedTime);
```

method of the `phaseExtension` instance. It requires a handle to the prepared user function as well as the normalized time steps it is evaluated on. The handle must be a prepared path constraint that implements the derivatives and supports the info struct interface.

Conditions

Once a Vanishing Constraint is added, it has to be specified under which conditions it has to be active. Therefore, a new condition

```
condition = vanishingConstraint.addNewCondition(constraint,...
    dc1, set1, dc2, set2, ..);
```

is created which requires the following inputs:

constraint Array of `falcon.Constraint` objects which is used to specify the lower and upper bounds as well as the scaling of the path constraint. The number of constraints must match the number of constraints calculated by the path constraint handle. Lower and upper bound limits are transformed to fulfill the ≤ 0 condition. Bounds that are `inf` are ignored.

dc/set Pairs of discrete control and set identifiers defining the conditions under which the Vanishing Constraint is active. A `dc` can either be a `DiscreteControl` instance or its name identifier. The set is either an index array or a cell array of names of the set.

With the `addNewCondition` method it is possible to define multiple discrete control sets. The sets define when the path constraint must be considered. A vanishing constraint may be desired to be active if all or at least on of the condition pairs is fulfilled. Therefore, the condition method

```
condition.setLogicalCombine(combineType) % 'AND', 'OR'
```

allows to define how the sets are logically combined. The default setting is `AND`.

For each condition added to the vanishing constraint, custom bounds can be specified via the constraint. Thus, it is possible to change bounds on the discrete control selection. Additionally, it is possible to change constant inputs to the path constraint dependent on the condition. In order to use this feature, the names of the constants that shall be provided by the conditions

```
vanishingConstraint.setConditionConstantNames(name1, name2, ..)
condition.setConstants(c1, c2, ..)
```

must be specified in the vanishing constraint. Additionally, the values of the constants must be set in the condition.The order must coincide with the names provided.

Slack Variables

As mentioned in section 3.3.3, it is possible to extent a vanishing constraint by using slack variables. In this case, the relaxation parameter κ becomes optimizable and is driven to zero by an additional cost function:

$$J_P = \sum_i \kappa_i. \tag{4.111}$$

This feature is disabled by default and can be enabled by passing a `true` flag to the

```
vanishingConstraint.setUseSlackVariable(flag)
```

method. The initial relaxation slack parameter κ (and thus its upper bound) is specified by

```
vanishingConstraint.setRelaxVariable(value)
```

In case the slack variable feature is not used the relaxation parameter acts as a constant. The default value is arbitrarily set to 0.001. Additionally, it is possible to define a lower bound for the relaxation slack variable

```
vanishingConstraint.setRelaxVariableLowerBound(value)
```

which has a default value of zero.

Wrapping of Vanishing Constraint

During the optimization of the optimal control problem, the vanishing constraints are evaluated using the information flow scheme in Figure 4.37. As with the dynamic model, during optimization, the data provided by the phase is sorted and the original path constraint handle is called. The controls are split into the continuous controls, the slack variable, and the discrete control weights. Constants may be provided by the phase or the vanishing constraint. After the evaluation, the results are passed to the conditions which calculate the vanishing constraint.

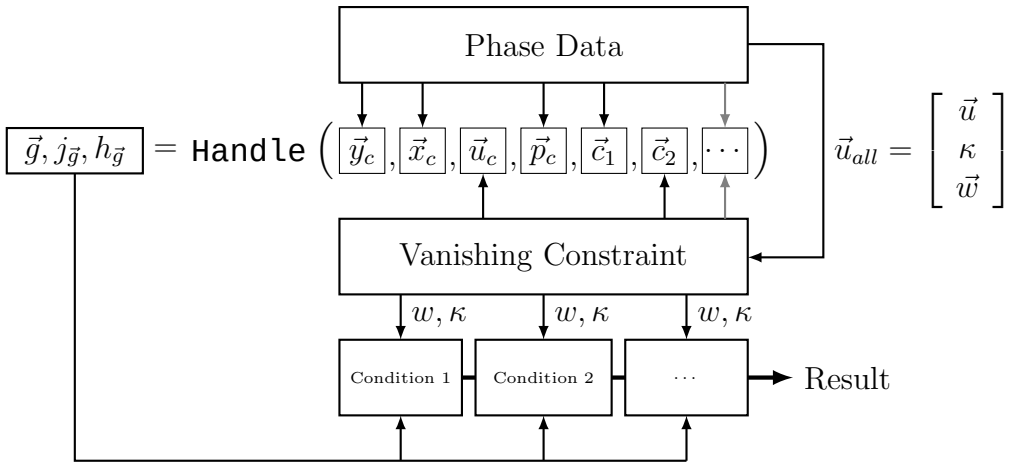


Figure 4.37: Information flow in the vanishing constraint wrapper.

4.7.5 Switching Costs

The switching cost approach spans across multiple phases and hence must be implemented on the problem level. This is achieved with a `ProblemExtension`. This extension is added to the problem with the

```
problemExtension = falcon.ext.dc.ProblemExtension();
problem.addProblemExtension(problemExtension);
```

method. Thus, the problem extension automatically implements switching costs for all discrete controls that are found in the problem.

Bake Augmentation

During the bake process the following augmentations are made to the problem:

1. Find all phases that implement discrete controls. Extract sequences of connected phases using the `ConnectedNextPhase` property.
2. For every set of connected phases:
   (a) Find a unique set of discrete controls.
   (b) For each discrete control found:
      i. Find a unique set of discrete control set values. It may be possible that two phases have the same discrete control but with varying sets.
      ii. For each entry of the unique discrete control set: Find the connected phases and implements the switching cost approach.

Influencing Switching Costs

Although the switching cost functions are created by the problem extension automatically, they can be influenced. By default, all switching costs have a penalty multiplier of 0.05. This scaling is stored in the `DefaultPenaltyScaling` property and can be set using the method:

```
problemExtension.setDefaultSwitchingPenalty(defaultSwitchingPenalty);
```

Individual penalties for certain discrete controls can be set with

```
problemExtension.setSwitchingPenalty(...
    dc1, penalty1, dc2, penalty2, ..);
```

which expects pairs of discrete controls and penalties. In order to disable the switching cost in the first optimization stage, the `PenaltyMultiply` property exists. It is applied to all switching costs and has a default value of zero. Thus, all switching costs are disabled. This is required in the first stage of the two stage solution approach. The value can be changed with the method

```
problemExtension.setPenaltyMultiply(multiplyValue);
```

which expects a value between zero and one. Slack variables can be used for the switching cost approach, too. The use of a slack variable (for all switching cost functions) is enabled using

```
problemExtension.setUseSlackVariable(flag);
```

The upper bound of the slack variable is set with the `setSlackUpperBound` method.

Solve

In order to solve problems with discrete controls at least two optimization stages are required. The two stage solution approach was introduced in 3.6 and implements a first optimization stage without switching cost and a second with switching costs. It is conducted automatically by calling the

```
problemExtension.Solve(solver);
```

method of the problem extension. The solve method expects a *FALCON.m* optimizer instance.

As mentioned in 3.6.3, the intermediate switching structure may be augmented by the box filter. The filter option is disabled by default and is enabled with the

```
problemExtension.setIntermediateAugmentation(...
    dc1, duration1, dc2, duration2, ..)
```

method. It expects pairs of discrete controls and duration parameters. Thus, the minimum duration is specified the discrete control must have. Only specified discrete controls will be augmented.

Chapter 5

Application to Automotive Race Track Optimization

In this chapter, the developed toolbox is applied to calculate the minimal lap time of a car through a race course. The optimal trajectory is subject to the model dynamics. The state and control histories are optimized. Additionally, the gear selection is introduced as a discrete control.

There are several publications that are concerned with time minimal lap times through a race circuit. Good results are obtained by [116] who calculates the minimal lap time under the consideration of the Kinetic Energy Recovery System (KERS). [117] evaluates time minimal cornering of a 180 degree turn for combinations of road surfaces and transmission layouts (rear/front wheel drive). However, in most publications, discrete decisions such as gear changes are not considered. [7] optimizes the full dynamic model with gear changes through a lane changing maneuver. In [6] a similar dynamic model is used to calculate the minimal lap time trajectory through a race course. The results presented in this chapter are comparable to [6] regarding the formulation of the discrete controls using OC. Therefore, the results are primarily used to show the behavior of the novel cost function formulation as well as the solution strategy. A single track model is used for the optimization [7]. Although the results show a realistic driving strategy, the optimal solution for a high fidelity model may differ significantly. The simple model is chosen as it was previously used in other MIOCP and thus can be regarded as an established test model. Early results of this chapter have been published in [111].

The chapter is organized as follows. In section 5.1 the car model, the race track formulation, and the constraints considered in the optimization are presented. Three optimizations are carried out in section 5.2. The first two show the structure of the solution without and with the intermediate spike removal. Additionally, the gas and brake pedal are introduced as discrete controls in the third optimization. In the last section 5.3, the discrete control formulation is tested against various user parameters to show the stability and consistency of the approach.

5.1 Problem Setup

This section discusses the general problem setup of the track optimization. First the formulation of the race track with variable width is introduced, followed by a descrip-

tion of the single-track car model. Additionally, it is stated how the track and other constraints are considered within the car model. Finally, the solution strategy is presented. The problem description is partially taken from the author's publication [111].

5.1.1 Track Model

In order to consider the race track in the OCP, a mathematical representation of it is required. Here, the track is modeled with two cubic splines. The first spline calculates the center line position

$$\vec{\eta}(s) = \begin{bmatrix} x_s(s) \\ y_s(s) \end{bmatrix} \tag{5.1}$$

w.r.t. to the spline parameter s. The second spline calculates the half width of the track

$$\sigma(s) \tag{5.2}$$

w.r.t. to the same spline parameter s. Both splines share the same break parameters

$$s_j = s_1, s_2, \ldots, s_{n_s}, \qquad j = 1, \ldots, n_s \tag{5.3}$$

at which the discretized center line waypoints and track half width

$$x_{sj}, \quad y_{sj}, \quad \sigma_j \tag{5.4}$$

are defined. In between the break points, the spline is interpolated

$$\vec{\eta}_j(s) = \vec{a}_{\eta,j} + \vec{b}_{\eta,j} \cdot (s - s_j) + \vec{c}_{\eta,j} \cdot (s - s_j)^2 + \vec{d}_{\eta,j} \cdot (s - s_j)^3 \tag{5.5}$$

$$\sigma_j(s) = a_{\sigma,j} + b_{\sigma,j} \cdot (s - s_j) + c_{\sigma,j} \cdot (s - s_j)^2 + d_{\sigma,j} \cdot (s - s_j)^3 \tag{5.6}$$

$$s_j \leq s \leq s_{j+1}, \qquad j = 1, \ldots, n_s - 1 \tag{5.7}$$

where the coefficients a, b, c and d are calculated beforehand for each polynomial of the spline. The breaks may be any strict monotone increasing sequence of real numbers. Here, they are chosen

$$s_j = s_{j-1} + \sqrt{\left(x_{sj} - x_{sj-1}\right)^2 + \left(y_{sj} - y_{sj-1}\right)^2}, \qquad s_1 = 0, \qquad j = 1, \ldots, n_s \tag{5.8}$$

to be the accumulated distance over the discretized waypoints.

In the following, the generation of the center line and width of the track is explained. For the race track, the Nürburgring grand prix circuit is chosen, as it has a variety of turns with different radii and straights of various lengths. The method described here can be applied to any other circuit. To generate the splines, the left L and right R boundaries of the track are traced in e.g. Google Earth. The results are two sequences of Global Positioning System (GPS) positions that are transformed into local cartesian coordinates using an arbitrary reference point. For each sequence, a spline is generated

$$\vec{\eta}_L(s_L) = \begin{bmatrix} x_{sL}(s_L) \\ y_{sL}(s_L) \end{bmatrix}, \qquad \vec{\eta}_R(s_R) = \begin{bmatrix} x_{sR}(s_R) \\ y_{sR}(s_R) \end{bmatrix}. \tag{5.9}$$

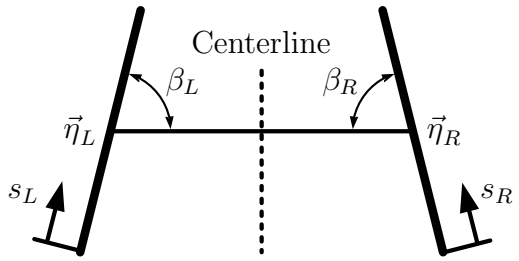


Figure 5.1: Spline opposite side condition $\beta_L = \beta_R$ for left and right track borders in order to determine spline for center line and half width.

In order to generate the center line and half width, the spline breaks of one side are taken as the reference. Then, for each spline break, the corresponding position and spline parameter value of the other spline need to be found. The condition

$$\left(\vec{\eta}_{N,R}(s_R) - \vec{\eta}_{N,L}(s_L)\right)^T \cdot \nabla_{s_L}\vec{\eta}_{N,L}(s_L) = \left(\vec{\eta}_{N,L}(s_L) - \vec{\eta}_{N,R}(s_R)\right)^T \cdot \nabla_{s_R}\vec{\eta}_{N,R}(s_R) \quad (5.10)$$

has to be fulfilled, where all vectors

$$\vec{\eta}_{N,L}(s_L) = \frac{\vec{\eta}_L(s_L)}{\|\vec{\eta}_L(s_L)\|_2} \qquad \nabla_{s_L}\vec{\eta}_{N,L}(s_L) = \frac{\nabla_{s_L}\vec{\eta}_L(s_L)}{\|\nabla_{s_L}\vec{\eta}_L(s_L)\|_2} \quad (5.11)$$

$$\vec{\eta}_{N,R}(s_R) = \frac{\vec{\eta}_R(s_R)}{\|\vec{\eta}_R(s_R)\|_2} \qquad \nabla_{s_R}\vec{\eta}_{N,R}(s_R) = \frac{\nabla_{s_R}\vec{\eta}_R(s_R)}{\|\nabla_{s_R}\vec{\eta}_R(s_R)\|_2} \quad (5.12)$$

are normalized. The angle between the line connecting both spline points and the spline direction (gradient) must have the same angle (see Figure 5.1). The condition can be formulated as an unconstrained optimization problem.

Using the results

$$\hat{\vec{\eta}}_R = (\hat{s}_R), \qquad \hat{\vec{\eta}}_L = (\hat{s}_L), \quad (5.13)$$

the center points

$$\vec{\eta} = \frac{\hat{\vec{\eta}}_R + \hat{\vec{\eta}}_L}{2}, \quad (5.14)$$

and the half widths

$$\sigma = \frac{1}{2} \cdot \left\| \hat{\vec{\eta}}_R - \hat{\vec{\eta}}_L \right\|_2 \quad (5.15)$$

of the race track are calculated. The breaks of the center line are calculated with (5.8) using the obtained center points. Figure 5.2 shows the result for the Nürburgring grand prix circuit.

5.1.2 Car Model

In this section, the car model is described. It is a single track car model taken from [23, 7]. Rolling and pitching behavior are not considered.

Figure 5.3 displays the coordinates, forces, and other entities of the car model. The states and controls are stated in Tables 5.1 and 5.2 respectively. The equations of motion are described by a set of ordinary differential equations which are explained in the following.

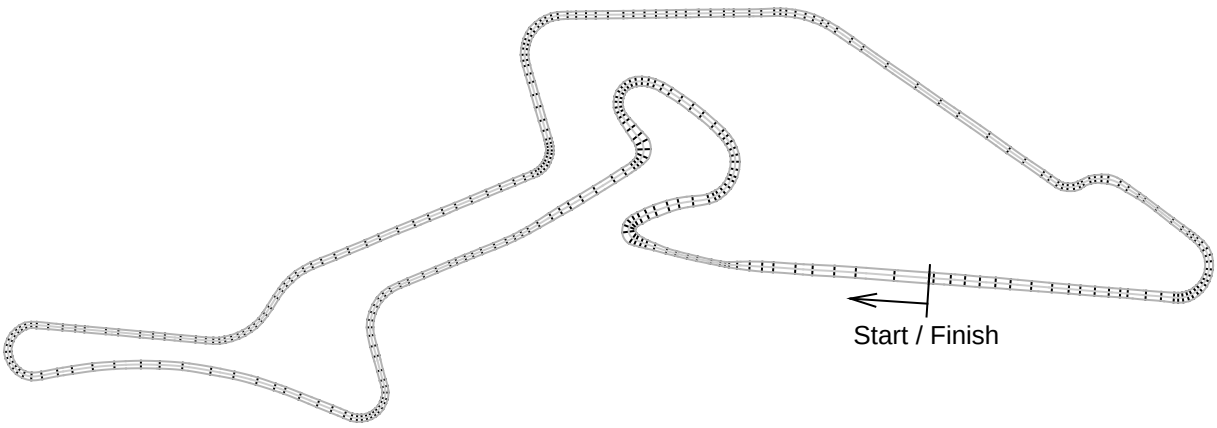


Figure 5.2: Nürburgring track represented as cubic splines. Dark and light gray lines show the track border and centerline. The spline breaks are shown by black lines.

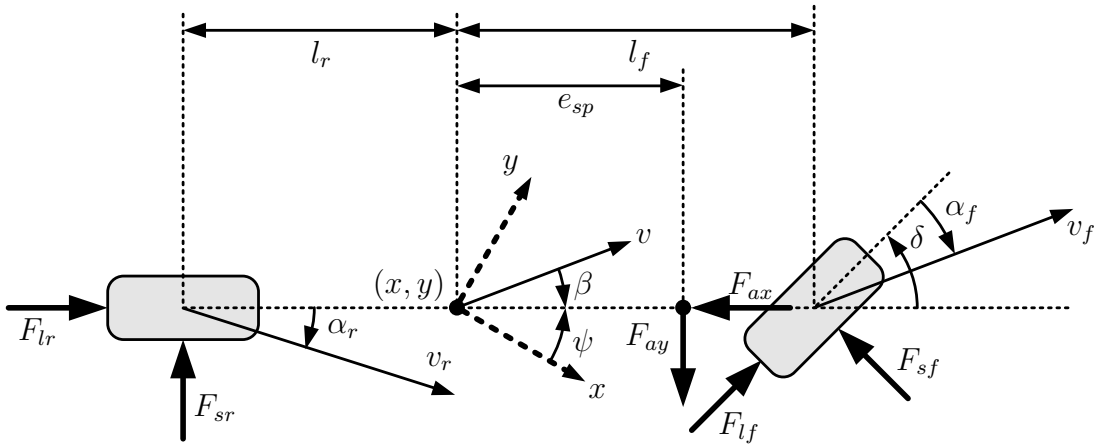


Figure 5.3: Single track car model with distances, angles and forces.

The kinematic state derivatives are given by

$$\dot{x} = v \cdot \cos(\psi - \beta) \tag{5.16}$$

$$\dot{y} = v \cdot \sin(\psi - \beta) \tag{5.17}$$

$$\dot{\delta} = w_\delta \tag{5.18}$$

$$\dot{\psi} = w_z \tag{5.19}$$

which include the position propagation as well as the integration of the steering angle δ and yaw angle rate w_z. The remaining state derivatives depend on the forces acting on the car. The acceleration of the Center of Gravity (CG)

$$\dot{v} = \frac{(F_{lr} - F_{ax}) \cdot \cos\beta + F_{lf} \cdot \cos(\delta + \beta) - (F_{sr} - F_{ay}) \cdot \sin\beta - F_{sf} \cdot \sin(\delta + \beta)}{M} \tag{5.20}$$

is determined using the sum of forces acting in the slip direction. The slip angle time derivative

$$\dot{\beta} = w_z - \frac{\begin{array}{c}(F_{lr} - F_{ax}) \cdot \sin\beta + F_{lf} \cdot \sin(\delta + \beta) \\ + (F_{sr} - F_{ay}) \cdot \cos\beta + F_{sf} \cdot \cos(\delta + \beta)\end{array}}{M \cdot v} \tag{5.21}$$

is influenced by the yaw angle rate w_z as well as the forces acting perpendicularly to

Table 5.1: States of the single track car model.

Name	Description	Unit
x	x-position	$[m]$
y	y-position	$[m]$
v	Speed	$[m/s]$
δ	Steering wheel angle	$[rad]$
β	Side slip angle	$[rad]$
ψ	Yaw angle	$[rad]$
w_z	Yaw angle rate of change	$[rad/s]$

Table 5.2: Controls of the single track car model.

Name	Description	Bounds	Unit
w_δ	Steering angle rate of change	$[-0.5, 0.5]$	$[rad/s]$
ξ	Normalized brake pedal position	$[0, 1]$	$[-]$
ϕ	Normalized accelerator pedal position	$[0, 1]$	$[-]$
i_G	Gear	Discrete	$[-]$

the slip direction. The yaw angle time derivative is given by

$$\dot{w_z} = \frac{F_{sf} \cdot l_f \cdot \cos\delta - F_{sr} \cdot l_r - F_{ay} \cdot e_{sp} + F_{lf} \cdot l_f \cdot \sin\delta}{I_{zz}} \tag{5.22}$$

where l_r and l_f are the rear and front distances to the CG. The moment of inertia is given by I_{zz} and the distance of the aerodynamic drag point to the CG by e_{sp}.

Forces

The forces can be divided into the following groups. Side and longitudinal forces (subscript s and l) act on the front and rear wheel (subscript f and r) in the respective local coordinate system. Additionally, the aerodynamic forces (F_{ax} and F_{ay}) are stated in the car fixed coordinate system. The values for all coefficients and parameters are given in table 5.3 at the end of this section.

The side forces of the front and rear wheel

$$F_{sf} = D_f \cdot \sin\left(C_f \cdot \arctan\left(B_f \cdot \alpha_f - E_f \cdot \left(B_f \cdot \alpha_f - \arctan\left(B_f \cdot \alpha_f\right)\right)\right)\right), \tag{5.23}$$

$$F_{sr} = D_r \cdot \sin\left(C_r \cdot \arctan\left(B_r \cdot \alpha_r - E_r \cdot \left(B_r \cdot \alpha_r - \arctan\left(B_r \cdot \alpha_r\right)\right)\right)\right) \tag{5.24}$$

are calculated using the PACEJKA magic formula [118]. In agreement with the author of [7] the peak values of D_f and D_r are doubled. Otherwise the side friction of the model would be too low. The magic formula is dependent on the tire's slip angle. The slip angles of the front and the rear tire are given by

$$\alpha_f = \delta - \arctan\left(\frac{l_f \cdot w_z - v \cdot \sin\beta}{v \cdot \cos\beta}\right), \tag{5.25}$$

$$\alpha_r = \arctan\left(\frac{l_r \cdot w_z - v \cdot \sin\beta}{v \cdot \cos\beta}\right). \tag{5.26}$$

The forces acting on the tires in their longitudinal direction

$$F_{lf} = -F_{Bf} - F_{Rf}, \tag{5.27}$$
$$F_{lr} = F_{Mr} - F_{Br} - F_{Rr} \tag{5.28}$$

consist of multiple influences. For the front tire, the total force is composed of the brake force as well as the roll friction. As the car model is assumed to be rear wheel drive, the rear tire force has an additional term for the motor force.

The total brake force commanded using the brake pedal is distributed between the front and the rear wheel

$$F_{Bf} = \frac{2}{3} \cdot F_B \cdot \xi, \tag{5.29}$$
$$F_{Br} = \frac{1}{3} \cdot F_B \cdot \xi \tag{5.30}$$

unevenly, since front brakes are usually more effective.

The speed dependent roll friction is given by

$$f_R = 9 \cdot 10^{-3} + 7.2 \cdot 10^{-5} \cdot v + 5.038848 \cdot 10^{-10} \cdot v^4 \tag{5.31}$$

which is distributed between the front and rear wheel

$$F_{Rf} = f_R \cdot M \cdot g \cdot \frac{l_r}{l_f + l_r}, \tag{5.32}$$
$$F_{Rr} = f_R \cdot M \cdot g \cdot \frac{l_f}{l_f + l_r} \tag{5.33}$$

using the loads on the respective axes. The weight of the car is given by Mg and l_f, l_r represent the distances of the front and rear tires to the CG.

The motor force on the rear tire

$$F_{Mr} = \frac{i_G \cdot i_t}{R} \cdot M_{mot} \tag{5.34}$$

is dependent on the wheel radius R, the transmission ratios of gear i_G (which is introduced as a discrete control) and the engine torque transmission ratio i_t. The engine's torque

$$M_{mot} = f_1 \cdot f_2 + (1 - f_1) \cdot f_3 \tag{5.35}$$

is a function of three coefficients. These are calculated using

$$f_1 = 1 - e^{-3 \cdot \phi}, \tag{5.36}$$
$$f_2 = -37.8 + 1.54 \cdot w_{mot} - 0.0019 \cdot {w_{mot}}^2, \tag{5.37}$$
$$f_3 = -34.9 - 0.04775 \cdot w_{mot} \tag{5.38}$$

and are dependent on the accelerator pedal position as well as the engine's rotation speed

$$w_{mot} = \frac{i_G \cdot i_t}{R} \cdot v. \tag{5.39}$$

Table 5.3: Parameters and coefficients of car model. Data taken from [7]. The half car width parameter l_w is arbitrarily chosen to a realistic value. It is used to account for the width of the car in the track constraint.

Symbol	Description	Value	Unit
A	Effective flow surface	1.437895	$[m^2]$
B_f	Front tire stiffness factor	10.96	$[-]$
B_r	Rear tire stiffness factor	12.67	$[-]$
C_f	Front tire shape factor	1.3	$[-]$
C_r	Rear tire shape factor	1.3	$[-]$
D_f	Front tire peak value	9120.80	$[-]$
D_r	Rear tire peak value	7895.62	$[-]$
E_f	Front tire curvature factor	-0.5	$[-]$
E_r	Rear tire curvature factor	-0.5	$[-]$
I_{zz}	Moment of inertia	1752	$[kg \cdot m^2]$
R	Wheel Radius	0.302	$[m]$
c_w	Air drag coefficient	0.3	$[-]$
e_{sp}	Distance drag mount to CG	0.5	$[m]$
g	Gravitational acceleration	9.81	$[m/s^2]$
i_G	Gear transmission ratios *1st to 5th gear*	$\{3.91, 2.002, 1.33, 1.0, 0.805\}$	$[-]$
i_t	Engine transmission ratios	3.91	$[-]$
l_f	Distance front wheel to CG	1.19016	$[m]$
l_r	Distance rear wheel to CG	1.37484	$[m]$
M	Car mass	1239	$[kg]$
ρ	Air density	1.249512	$[kg/m^3]$
F_B	Maximum brake force	$1.5 \cdot 10^4$	$[N]$
l_w	Half car width	0.8	$[m]$

Finally, the aerodynamic forces

$$F_{ax} = \frac{1}{2} \cdot c_w \cdot \rho \cdot A \cdot v^2, \tag{5.40}$$

$$F_{ay} = 0 \tag{5.41}$$

are calculated with a simple drag equation dependent on the drag coefficient c_w, the air density ρ, the projected reference area A, and the car's speed v. The lateral aerodynamic force is assumed to be small and is therefore ignored.

5.1.3 Constraints

In the following, the constraints that are considered in the optimal control problem are explained.

Track Constraint

As mentioned above, the race track is formulated as a cubic spline that gives the track's center line and the half width w.r.t. the spline parameter s. In the optimization, a

track constraint that keeps the car within the bounds of the track must be formulated. Therefore, the spline parameter s that represents the closest point on the spline to the car (foot point) needs to be determined (see Figure 5.4).

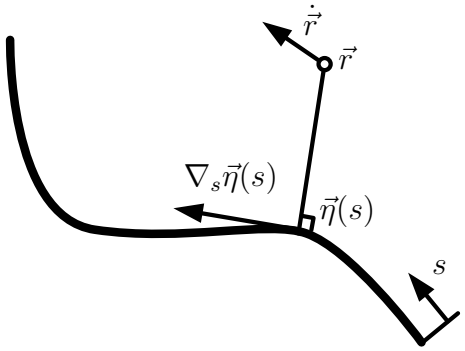


Figure 5.4: Car position with foot point on spline.

The following approach is based on the formulation used by [77] where the spline parameter was introduced as an additional state which is integrated alongside the dynamics. The spline state derivative

$$\dot{s} = \frac{\dot{\vec{r}}^T \cdot \nabla_s \vec{\eta}}{(\nabla_s \vec{\eta})^T \cdot \nabla_s \vec{\eta} - (\vec{r} - \vec{\eta})^T \cdot \nabla^2_{ss} \vec{\eta}}, \qquad \vec{r} = \begin{bmatrix} x \\ y \end{bmatrix}, \quad \vec{\eta} = \begin{bmatrix} x_s \\ y_s \end{bmatrix} \tag{5.42}$$

depends on the car position $\vec{r}$, speed vector $\dot{\vec{r}}$, as well as the spline position $\vec{\eta}$ and the first and second order derivatives w.r.t. to the spline parameter. The differential equation gives the propagation of the spline parameter state s. The spline position of the integrated spline parameter remains a foot point to the car. Although this formulation works well for splines with few nodes, it has some drawbacks.

- The initial spline parameter must be a foot point to the initial position of the dynamic system. Therefore, a perpendicularity constraint for $s(t_0)$ must be considered.
- The breaks of the spline used for the interpolation must be the actual spline lengths from the initial position. Otherwise a drift in the foot point may occur with the result that the perpendicularity is no longer given.
- Although the differential equation provides a simple way to calculate the foot point, the additional dynamics are difficult to fulfill in numerical optimization. The formulation tends to be numerically unstable. Especially on strong changes in the curvature (sharp turns), the discretization density may be insufficient. This results in an integration error that ultimately leads to a drift of the assumed foot point. This drift remains for the rest of the trajectory and cannot be compensated.

Therefore, in this thesis, the spline parameter is introduced as an additional control u_s. Thus, the spline foot point calculation becomes decoupled from the system dynamics. In this case, not only the distance to the spline must be constrained but also the perpendicularity of the foot point.

As the whole car has to stay within the track it is insufficient to consider the center point only. Therefore, the car is expanded by a width parameter l_w which represents

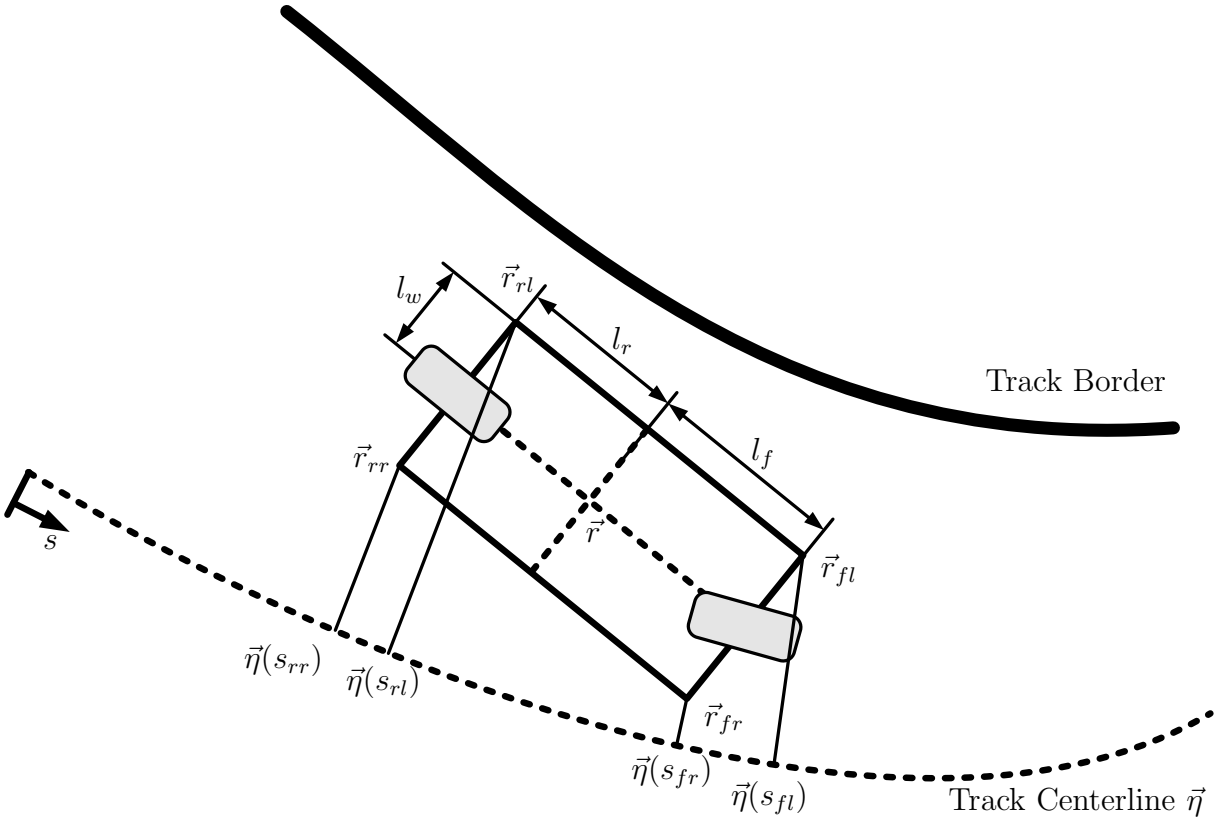


Figure 5.5: Consideration of car corners in optimal control problem.

the half width of the car (see Figure 5.5). Thus, for instance, the position of the front right wheel of the car is calculated by

$$\vec{r}_{fr} = \vec{r} + \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix} \cdot \begin{bmatrix} l_f \\ l_w \end{bmatrix} \tag{5.43}$$

for which the footpoint $\vec{\eta}_{fr}$ w.r.t. the spline parameter s_{fr} needs to be determined. The same holds true for the other wheel $\vec{r}_{fl}, \vec{r}_{rr}, \vec{r}_{rl}$ and spline positions

$$\vec{\eta}_{fl} = \vec{\eta}(s_{fl}), \qquad \vec{\eta}_{rr} = \vec{\eta}(s_{rr}), \qquad \vec{\eta}_{rl} = \vec{\eta}(s_{rl}). \tag{5.44}$$

For simplicity, only the front, right wheel is considered in the following.

In order to determine the correct footpoint $\vec{\eta}_{fr}$ to the wheel position $\vec{r}_{fr}$, the spline control s_{fr} must fulfill the perpendicularity

$$\left(\vec{r}_{fr} - \vec{\eta}(s_{fr})\right)^T \cdot \nabla_s \vec{\eta}(s_{fr}) = 0 \tag{5.45}$$

equality constraint at every time step.

Due to the changing width $\sigma(s_{fr})$ of the race track, the centerline distance constraint

$$0 \leq \frac{\left(\vec{r}_{fr} - \vec{\eta}(s_{fr})\right)^T \cdot \left(\vec{r}_{fr} - \vec{\eta}(s_{fr})\right)}{\sigma^2(s_{fr})} \leq 1 \tag{5.46}$$

is normalized. Please note that the physical constraint is squared as the actual distance (using the square root)

$$\frac{\partial}{\partial s_{fr}} \left(\frac{\sqrt{\left(\vec{r} - \vec{\eta}(s_{fr})\right)^T \cdot \left(\vec{r} - \vec{\eta}(s_{fr})\right)}}{\sigma(s_{fr})} \right) \tag{5.47}$$

would introduce infinite values in the derivatives in case the car is on the center line.

Engine Rotation Speed Constraint

The motor frequency is transformed to revolutions per minute

$$n_{mot} = w_{mot} \cdot \frac{60}{2 \cdot \pi} \tag{5.48}$$

and is limited to a maximum of $6500rpm$. Thus, switches to low gear selections at high speeds are prevented.

Periodic Phase Defect

Finally, the problem is set up as a periodic OCP. The initial and final states

$$\vec{x}_0 - \vec{x}_f = 0 \tag{5.49}$$

must coincide and thus the optimal solution represents the car's time minimal trajectory through the course. Please note that the cars yaw angle may have

$$\psi_0 - \left(\psi_f - k \cdot 2\pi\right) = 0, \qquad k \in \mathbb{N} \tag{5.50}$$

an offset of multiple revolutions.

5.1.4 Solution Strategy

The MIOCP is solved with the solution strategy presented in section 3.6. In the first stage, the optimal control problem is solved without switching costs in order to generate an initial guess for the discrete controls. Afterwards, in the second and final stage, the switching cost is activated and the solution process is warm-started. The switching cost penalty is scaled by 0.05 and the multi-time switching cost function is formulated using a slack variable bounded between $[0, 0.9]$. The feasibility and optimality tolerance are set to $1 \cdot 10^{-5}$. Furthermore, the following topics are considered.

Track Constraint Scaling

Due to the fact that the perpendicularity track constraint is hard to fulfill by the optimization algorithm, the scaling of the corresponding constraints is set to 0.001. This reduces the required accuracy from 10^{-5} to 10^{-2}. The resulting angle error lies at around 0.6 degrees.

Initial Guess

The initial guess is provided in a very simple way. For the states only the position, speed, and orientation of the car are initialized. Using the discretization density of τ, the position of the car is initialized on the centerline for a full lap. Additionally, a constant speed of $25m/s$ ($90km/h$) is assumed. The yaw angle / orientation of the car is calculated from the spline first order derivative. All other states are set to zero.

All physical car controls ξ, ϕ, w_δ are initialized to zero. The spline controls are set to spline parameter values that match the car's position initialization. The gear is assumed to be in the third position. The initial guess for the car is very simple to provide and requires little to no input to the optimal trajectory.

5.2 Single Optimization Results

In this section, single optimization results are shown. In the first optimization the problem is solved with the solution strategy introduced above. The optimal solution is explained. Afterwards, the same OCP is solved using the intermediate spike removal (see section 3.6.3). The differences to the first optimization results are discussed. Finally, the gas and brake pedals are modeled as additional discrete controls. Thus, the problem is solved with three discrete controls.

All problems are discretized in time with 3001 equally distributed points resulting in 60016 optimization variables and 51018 constraints for the first two optimization problems. The sparsity of the Jacobian is 99.9781%. Due to the discrete control modeling in the third optimization, the last problem contains more optimization variables. The scaling of the switching cost is set to 0.05. The main lap time cost function is scaled with 0.01.

5.2.1 Without Spike Removal

Here, the OCP is solved without the intermediate spike removal. The optimal solution is retrieved within 33.9 minutes, where the first optimization stage required 4.0 minutes and the second (with switching cost) 29.9 minutes[1]. Early results of this section have been published in [111].

Figure 5.6: Gear switching solution of Nürburgring Grand Prix circuit (without spike removal).

Figure 5.6 shows the optimal trajectory through the circuit with colors indicating the different gear choices. Low gear choices are found in the turns and high gear choices are found in the straight segments of the track. The minimal lap time obtained by the optimal solution is 132.265 seconds. The full width of the track is exploited

[1]Intel Core i7-930 CPU @ 2.80GHz, Windows 8 64bit, MATLAB 2015a

resulting in a realistic time minimal track for the model used. In Figure 5.7 the time history for various data is displayed. In each plot, the results from the intermediate optimization step without switching cost (orange), and the final solution with switching cost (blue) are shown.

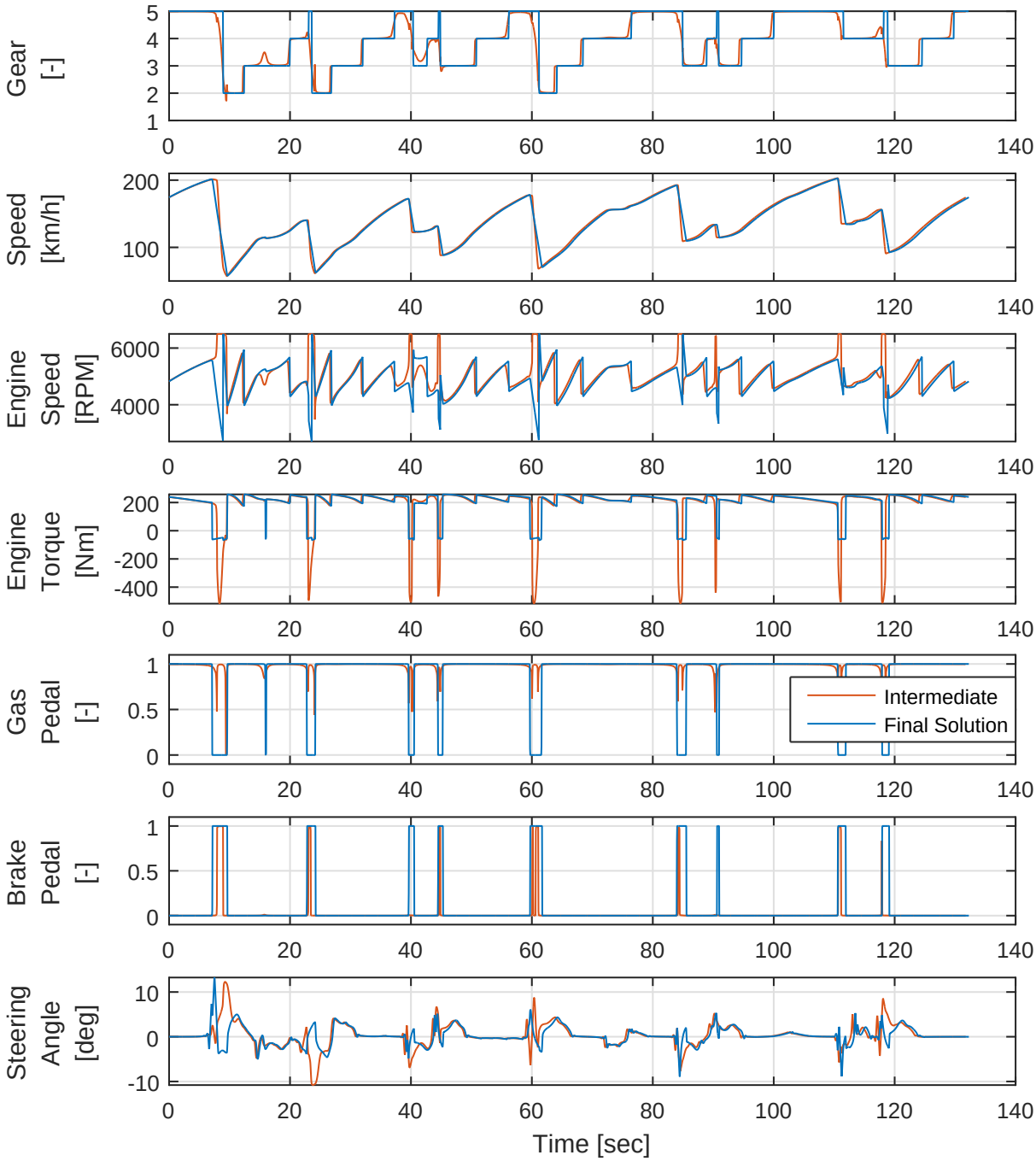


Figure 5.7: Time history of various data for solution of Nürburgring Grand Prix circuit.

In the final solution, the gear selection contains sharp immediate switches whereas in the intermediate solution the results are more continuous. Especially during downshifts before the car enters a turn, fractional values for the discrete controls are found. Additionally, a switch to low gears increases the Revolutions Per Minute (RPM) of the motor drastically (see w_{mot} plot) and enables the car to use the resulting negative engine torque as an additional brake (see correlation between motor torque M_{mot} and the

brake ξ). Due to the fact that the discrete control weights are relaxed and the absence of the switching costs, the optimizer exploits this degree of freedom by introducing fractional values. Thus, the gears become weighted and result in a continuous change. Additionally, the RPM becomes weighted and stays at its upper limit during deceleration. Thus, the magnitude of the deceleration is driven by the RPM limit.

In the second optimization step, switching costs are considered in the OCP. The discrete control weights must become binary feasible. Thus, the optimizer is forced to find an actual switch in the discrete control history. This introduces several changes in the optimal solution.

The minimal lap time duration is slightly increased. Therefore, the time histories of the data displayed drift to later times. Fractional values and high frequent switches are mitigated in the gear switching plot. Especially at downshifts a clear switch to lower gears is found. As mentioned above, the deceleration of the car is driven by the engine's RPM limit. Since the gear changes are no longer continuous, the car has to perform earlier and longer break maneuvers in order to maintain the limit. This change can be seen in the brake and speed plots. Additionally, compared to the intermediate solution, the RPM is reduced until a switch to a lower gear becomes feasible. In many cases, the RPM jumps to the upper bound after the switch.

Overall, this change of the solution requires many iterations and hence explains the high computational time for the second step. Additionally, it can be seen that the optimal gear switching solution contains short spikes to the highest gear selection. This behavior does not seem to be optimal and is addressed in the following.

5.2.2 With Spike Removal

Figure 5.8: Gear switching solution of Nürburgring Grand Prix circuit with spike removal between the optimization stages.

In order to remove the spikes to the fifth gear, the spike removal algorithm from section 3.6.3 is applied. Spikes with a duration of less than two seconds are removed from the intermediate solution. The augmentation of the discrete control weights is written to the optimal $\vec{z}$ vector of the previous optimization stage. The overall optimization required 35 minutes to solve and is thus approximately the same as before.

Figure 5.8 shows the gear switching trajectory on the Nürburging with the intermediate spike removal augmentation. It can be seen that the switches to the fifth gear

are removed. Instead, they are replaced by step-wise downshifting of the gear. This behavior seems to be more realistic. In Figure 5.9, the switching structures of both approaches are compared. The solution with the spike removal algorithm (orange) follows the intermediate / continuous switching structure (gray) very closely. Significant differences to the previous solution (blue, without spike removal) are found only at the downshift points. Otherwise, the gear switching solution is very similar. Finally, the solution without spike removal contains fewer switches.

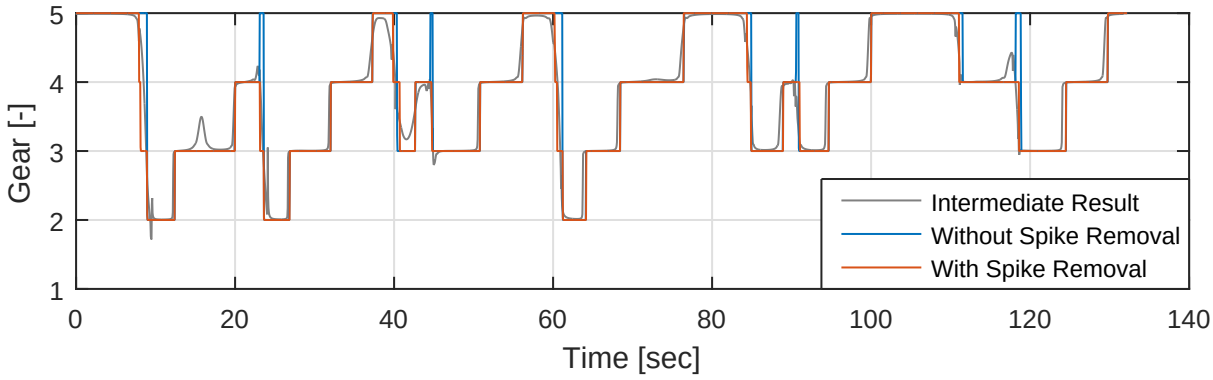


Figure 5.9: Comparison of switching structure without (blue) and with (orange) spike removal. Intermediate Solution shown in gray.

The time histories of the states and the continuous controls remain almost unaffected and are thus not shown. The minimal time for the spike removal solution is 132.2475 seconds and thus even slightly faster than the previous result of 132.2650 seconds. It becomes clear that switching cost approaches introduce many additional local minima in the optimal solution which are very sensitive to the provided solution.

5.2.3 Multiple Discrete Controls

As can be seen in the previous solutions, the gas and the brake pedal have a bang-bang optimal solution structure. Therefore, in this section, they are implemented as discrete controls in order to test the multiple discrete control capability. Thus, three discrete controls enter the dynamic model. The brake and throttle may not be active or inactive at the same time. After removing the invalid cases, 10 discrete control combinations remain. Compared to the previous optimizations, the number of discrete control weights is doubled. Two continuous controls are removed. Thus, the OCP has 69013 optimization variables representing an increase of 15%. The number of constraints and the discretization density remain the same.

Minor adaptations to the dynamic model and the generation of the initial guess are required. The former continuous controls gas pedal ϕ and brake pedal ξ now enter as separate discrete controls

$$\phi = \{0, 1\}, \qquad \xi = \{0, 1\}. \tag{5.51}$$

Furthermore, the initial guess for the discrete control is set to be in the third gear, full throttle and no brakes. The spike removal is applied to gear selection in the intermediate optimization stage. Spikes with a duration shorter than 2 seconds are removed. The gas and brake pedal switching structure is not augmented. For all discrete controls, the switching cost penalty is scaled by 0.05. Otherwise, the OCP remains the same.

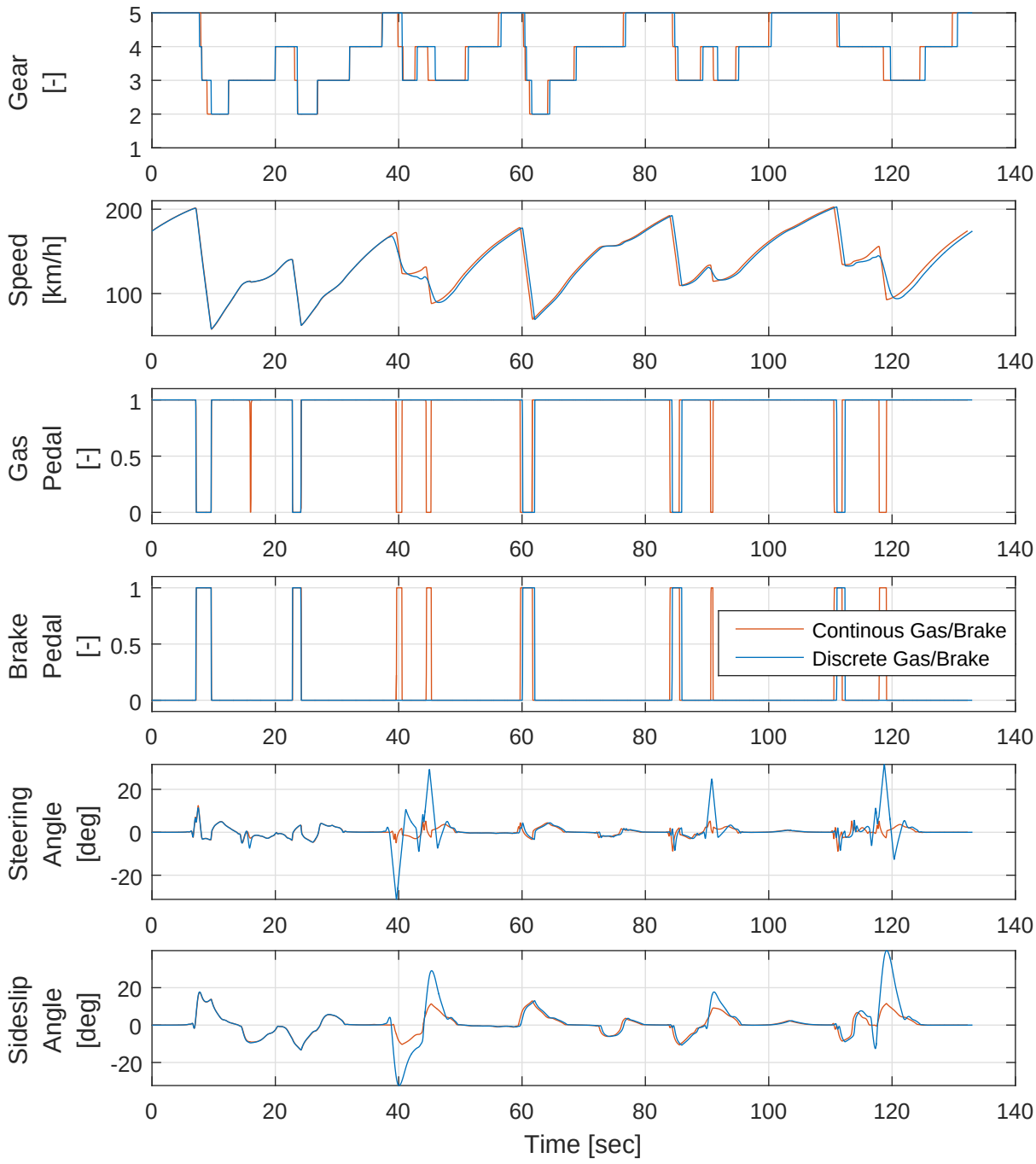


Figure 5.10: Solution comparison of continuous gas/brake pedal to discrete control introduced gas/brake pedal.

The obtained minimal lap time for the discrete gas and brake pedal is 133.0 seconds. In Figure 5.10, the optimization result is shown. It is compared to the solution with continuous throttle and brake from 5.2.2 (with spike removal augmentation). The solution with multiple discrete controls is similar to the continuous case. However, it can be noted that the obtained lap time is slightly slower. In the throttle and brake history it can be seen that the number of brake points is removed. Instead, the maximum throttle is retained. In order to maintain the track, higher side slip angles β are produced through strong steering angles. The strong drifting of the car reduces the car's speed leading to a reduced lap time. Although the results are good, it can be seen

that discrete controls are much harder to consider than continuous ones.

5.3 Switching Cost Formulation Stability

In this section, the robustness of the multi-time switching cost approach and of the solution strategy are discussed. Therefore, the user-selected parameters are varied over a range of values. These include the scaling of the switching penalty, the discretization density, and the initial guess for the discrete controls. Since only a single parameter is varied, all other settings are the same as in the optimizations of the previous section.

5.3.1 Penalty Scaling

The first user parameter that is varied is the scaling of the switching cost penalty α. It shall determined if and how the structure of the gear selection changes with a variation of the penalty scaling. The scaling factor is logarithmically spaced from $1 \cdot 10^{-6}$ to 100 in 81 steps. Thus, the considered range spans over eight magnitudes. All optimizations are started with the same initial guess. As expected, all intermediate solutions are exactly the same. First, the optimizations are executed without augmentation between the optimization stages. Afterwards, the same optimizations are carried out with the spike removal algorithm.

Without Spike Removal

Figure 5.11 displays an overview of the penalty scaling study. The tracks of the car for all 81 optimizations are shown. Below, different data is plotted over the penalty scaling on a logarithmic axis. The scaled lap time is plotted together with the switching cost penalty. Please note that for this plot, the vertical axis is logarithmic as well. Additionally, the real lap times are plotted. Finally, the computational times of the second stage in minutes are shown. The computational time for the first stage is not shown as it is the same for all optimizations.

In the track plot, the ground tracks for the different switching penalties do not match. The optimizations converge to different local minima. The reasons are discussed further below. In the plots below, four different sections on the penalty axis (x-axis) can be identified (indicated by green separation lines). For each of these sections, Figure 5.12 shows a zoomed section of the trajectory (red box in Figure 5.11) as well as the gear switching structures of the optimizations.

At first, the switching penalty scale parameter is very small and has significant impact on the optimal solution. Approximately in the middle of the first subrange, the influence by the optimization algorithm becomes noticeable. This is indicated by the sudden increase in the CPU time. The long computational times are an indicator that the optimization runs into the maximum number of 6000 iterations. Unsuccessful optimizations are indicated by a red cross in the CPU plot. Additionally, it can be seen in Figure 5.12 that the switchings structure is not changed significantly. Merely some minor spikes in the solution have been removed.

In the second subrange, the influence of the switching penalty becomes more prominent and the lap time starts to increase. In the trajectory plot of this subrange, the so-

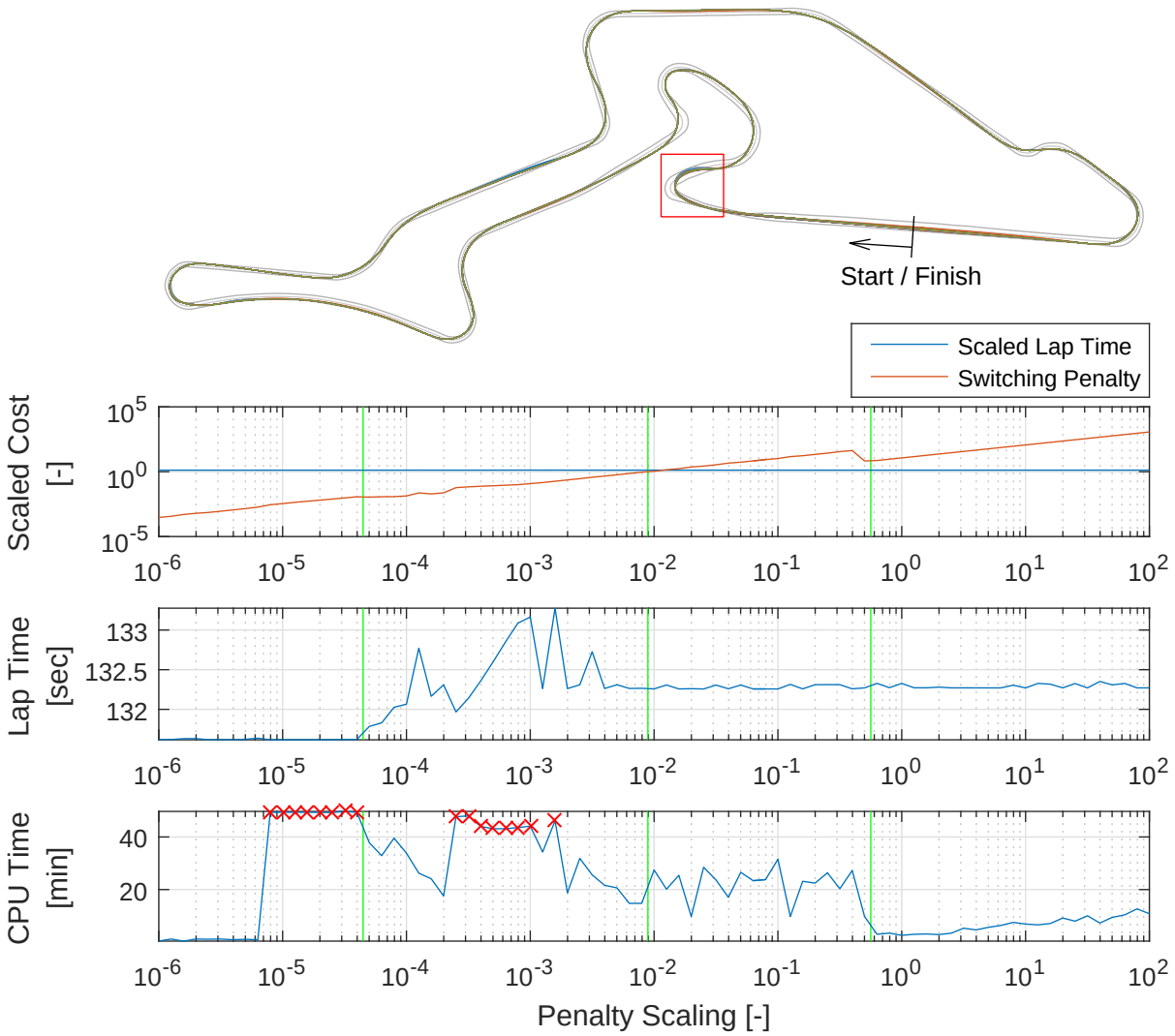


Figure 5.11: Car tracks, scaled lap time, and switching cost functions, real lap time, and computational Central Processing Unit (CPU) time over different switching cost penalties. Optimizations that did not converge are indicated by red crosses.

lution converges to many different minima. This behavior is also visible in the gear switching structures. The increase in the switching cost gradually enforce binary feasibility on the discrete control weights. Some solutions begin to show a clear switching structure. However, the solution seems to be very sensitive to the penalty scale parameter. As a reminder, all solutions from the first stage are exactly the same.

The third subrange starts shortly before the switching cost becomes higher than the scaled lap time. Additionally, a general decrease in CPU becomes visible. The switching structure shows clear switches over the whole subrange. Additionally, markers are used to show the fractional value at the switching instances. For integer feasible values, a marker is not shown (the tolerance is $1 \cdot 10^{-2}$). Although the solutions contain fractional values, the effect is irrelevant for practical applications as the switching times of the discrete control are clearly visible. The tracks for the optimizations of this subrange of penalty scaling parameters is very consistent.

Finally, the last subrange is indicated by the drop in computational time and the

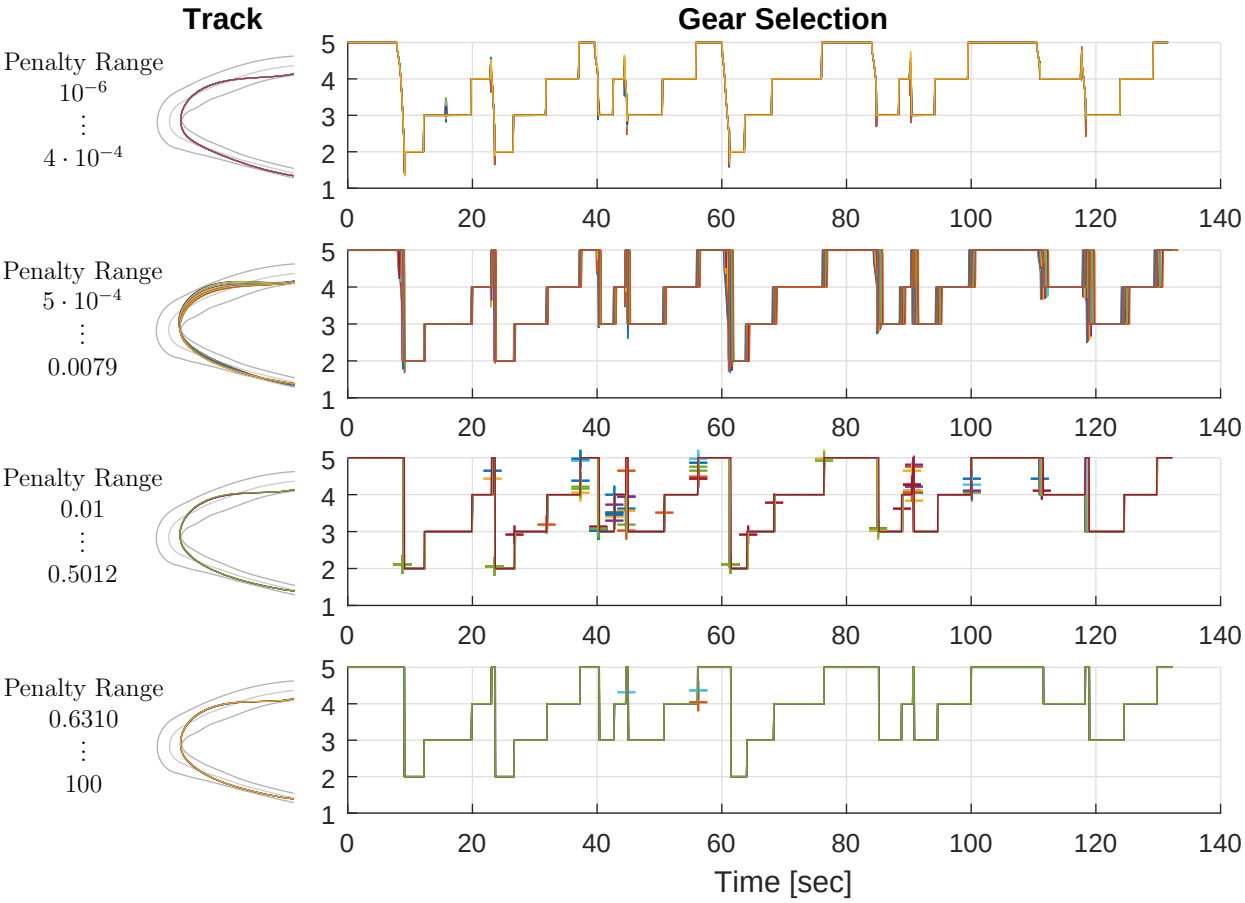


Figure 5.12: Track and switching structures for different subranges of the penalty scale parameter.

drop in the switching cost. In the gear switching structure plots of this subrange, fractional values for the discrete control are mitigated. The binary feasibility for the discrete control weights is much stronger enforced due to the large impact on the overall cost function. Thus, the overall switching cost is reduced.

With Spike Removal

In the following, the spike removal augmentation is enabled between both optimization stages. As can be seen in Figure 5.13, the results of the optimization are much more consistent. Even for very small scalings of the switching penalty, the impact is more gradual and the computational efforts are significantly reduced in some parts. Additionally, the track plot shows a high consistency of the race lines.

In Figure 5.14, the same penalty subranges are plotted as before (see Figure 5.13). The behavior w.r.t. to the switching penalty is very similar to that without the intermediate augmentation. The influences however seem to be more gradual and consistent over the change of the penalty scaling.

As before, in the first subrange, the smallest penalty scaling parameters show no influence in the optimal solution. Additionally, there is no increase in the computational time, since most of the spikes in the solution have already been removed by the intermediate step. At some, point the lap time increased. This indicates that the switching

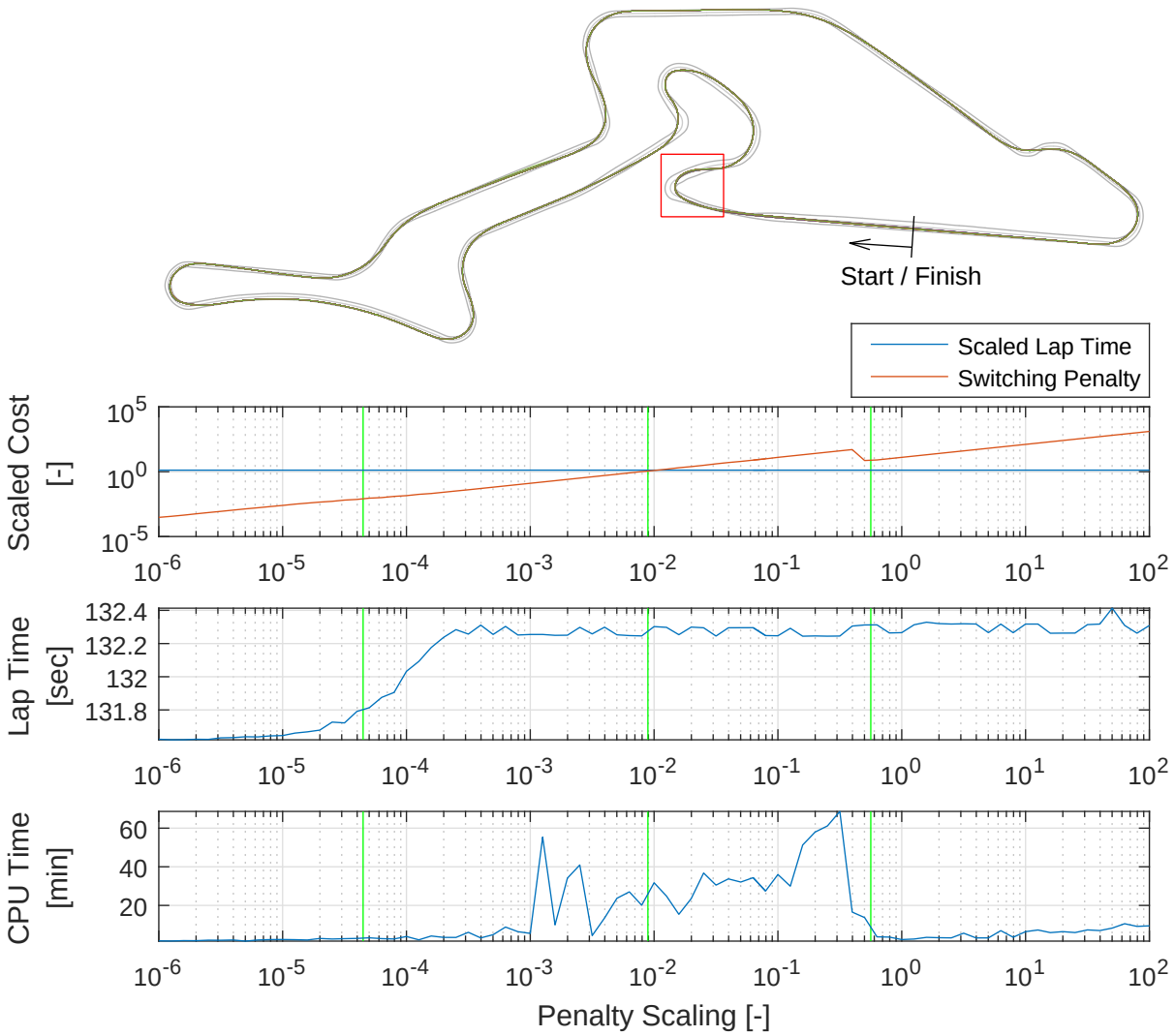


Figure 5.13: Car track, scale lap time and switching cost functions, real lap time and computational CPU time over different switching cost penalties with intermediate step spike removal.

cost begins to have a significant influence. Compared to the previous parameter study, the penalty scaling shows earlier effect. Overall, the CPU seem to be much lower, especially for small scalings. In the first plot row of Figure 5.14, no discrete switching structure is found.

In the second subrange, the discrete value feasibility is gradually enforced. This can be seen in the gradual and steady increase of the lap time. As before, the gear switching structure still varies and converges to different local minima. However, compared to the previous parameter study, the results are much more consistent. This is especially true for the track plots.

In the last to subranges, the behavior is analogous to the case without spike removal. Discrete value feasibility is enforced, except at the switching instances. With higher switching cost, this fractional values are mitigated.

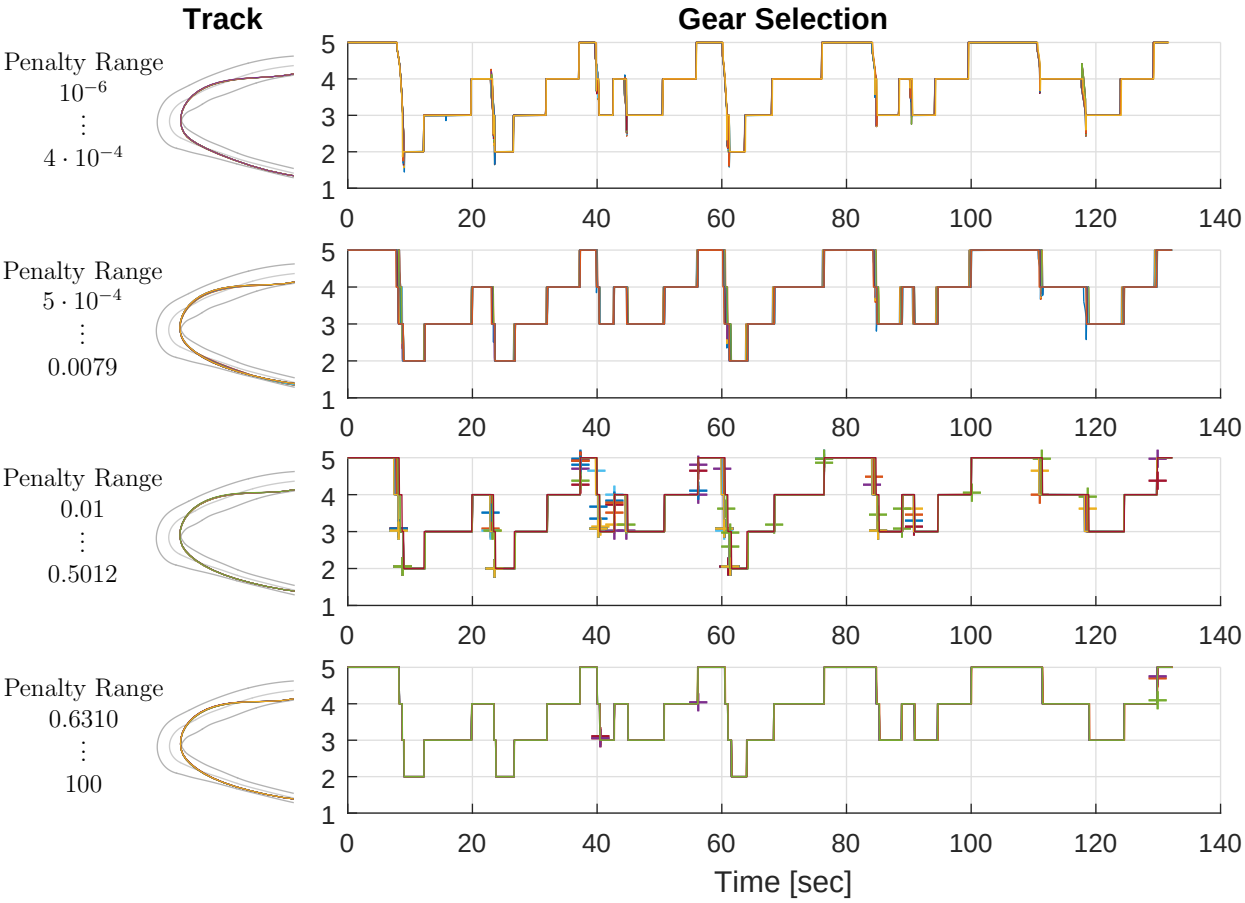


Figure 5.14: Track and switching structures for different subranges of the penalty scale parameter with intermediate step spike removal.

5.3.2 Discretization Density

In this study, the discretization density is varied from $1,001$ up to $30,001$ equidistant points. Thus, the optimization is solved for a relatively coarse discretization up to a very fine discretization. The number of optimization variables ranges from approximately $20,000$ to $600,000$ optimization variables. Similarly, the number of constraints ranges from approximately $17,000$ to $510,000$. All optimizations were solved successfully.

In the following, the optimizations are carried out with switching penalty scaling of 0.05. As before, the optimization is carried out with and without the spike removal augmentation.

Without Spike Removal

Figure 5.15 shows the time minimal lap trajectories, the switching penalties, the lap times and the computational times. The data is plotted w.r.t. the logarithmic discretization density (number of points).

The switching penalty increases slightly with finer discretization. With a higher discretization density, the optimization algorithm is enabled to perform more switches in the optimal solution. Thus, the switching penalty increases slightly. However, this

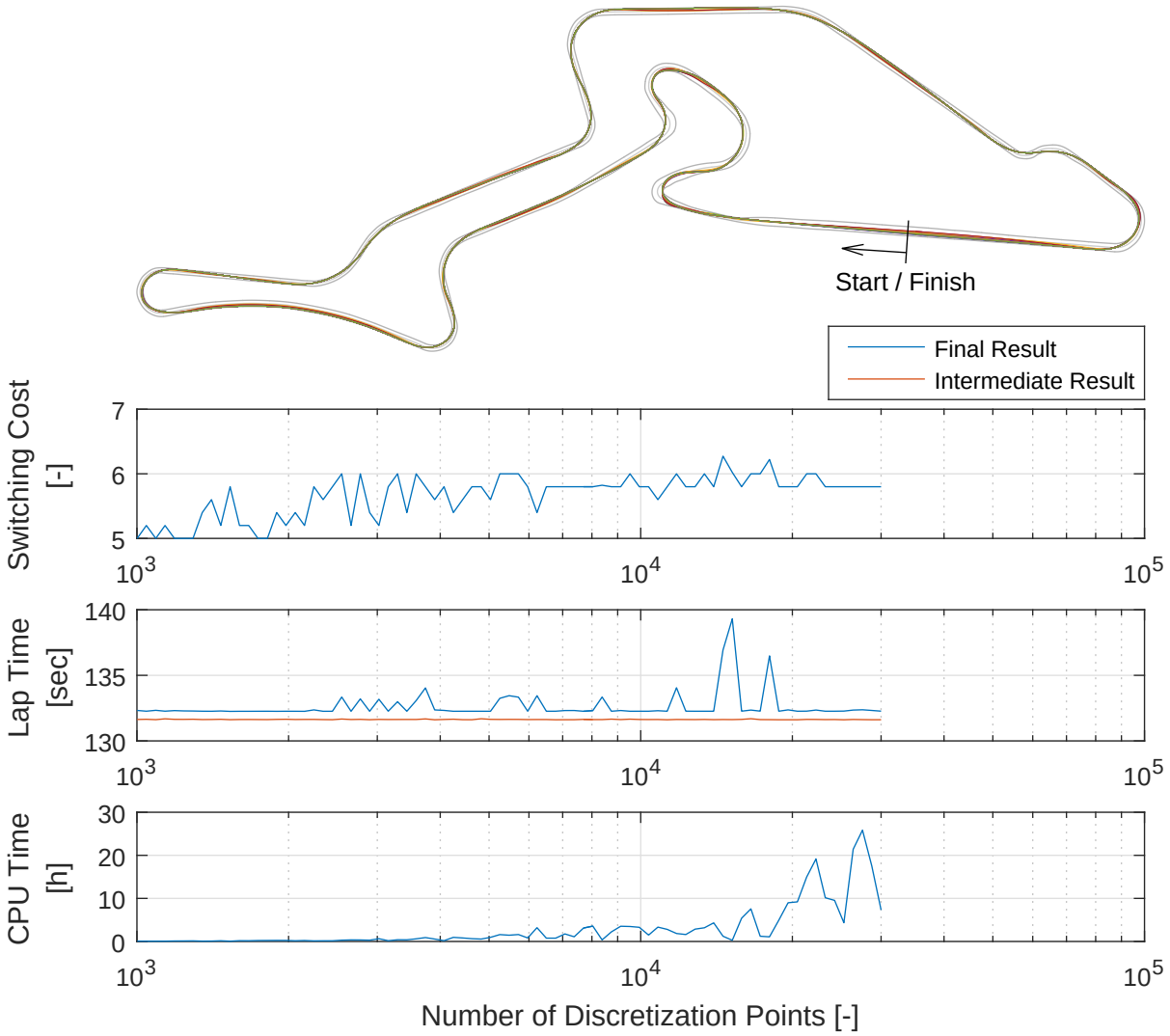


Figure 5.15: Car track, switching cost penalty, lap time and computational time over different discretization densities.

effect is only visible at lower discretization densities. Afterwards, the overall switching cost remains around the same size hinting at a constant number of switches. Therefore, it is not necessary to formulate the switching cost approach as an integral Lagrange function.

In the third plot, the minimal lap time over the discretization density is displayed. The final solution is shown in blue and the intermediate solution is plotted in red. For the final results, the optimal solutions can be divided into two groups. Almost everywhere, a very similar optimal lap time is obtained. However, in a few cases, the optimal solution is worse and shows that gradient based optimization algorithms converge to local minima. In contrast, the intermediate lap time is constant w.r.t. the discretization density. Therefore, it becomes clear that the second optimization stage with switching cost introduces significant local minima in the overall optimal control problem.

In the last plot, the overall computation time in hours w.r.t. discretization density is shown. As expected, the CPU increases with the number of optimization variables. The longest optimization required more than a day to find the optimal solution. Al-

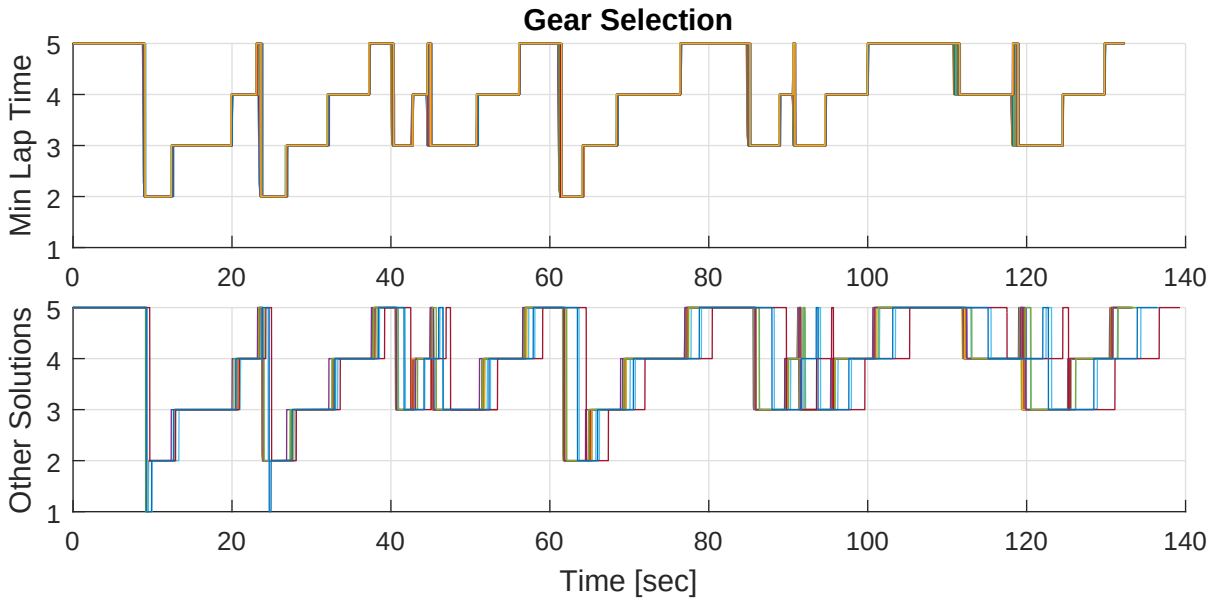


Figure 5.16: Gear selections for the minimal lap time and other solutions.

though the same lap time is obtained in many cases, the require computational time may vary up to 20 hours (for comparable discretization densities).

Figure 5.16 shows the switching sequence for all optimizations. The two plots differentiate between the best local minimum and all other solutions. For the same local minium, the results seem to fit very well. In all other solutions, apart from a few exceptions, almost the same gear switching structure is found. In a few cases, the optimal solution switches to the first gear. The time history is mainly scaled due to longer lap times.

With Spike Removal

In Figure 5.17, the same discretization densities are used but with spike removal augmentation. As before, the same overall behavior regarding the slight increase in the switching penalty as well as longer computational times with finer discretization can be seen. However, due to the spike removal, the results are much more consistent. Also, the number of other minima found is drastically reduced (spikes in the lap time plot). However, the required computational time still varies extremely.

The switching structures for the gear choices with spike removal are shown in Figure 5.18. The switching structure of the gears is very similar for all optimizations that converged to the lowest obtained lap time. Even the optimizations that resulted in longer lap times show the same switching structure. It appears that the switching structure is merely scaled by time. Since the lap times in the intermediate solution show no deviations, the switching costs are the reason for the local minima.

As was seen in the penalty scaling study, the spike removal between both optimization stages increases the consistency of the switching structure. Therefore, for the following optimization study, only the solution with the intermediate augmentation is shown.

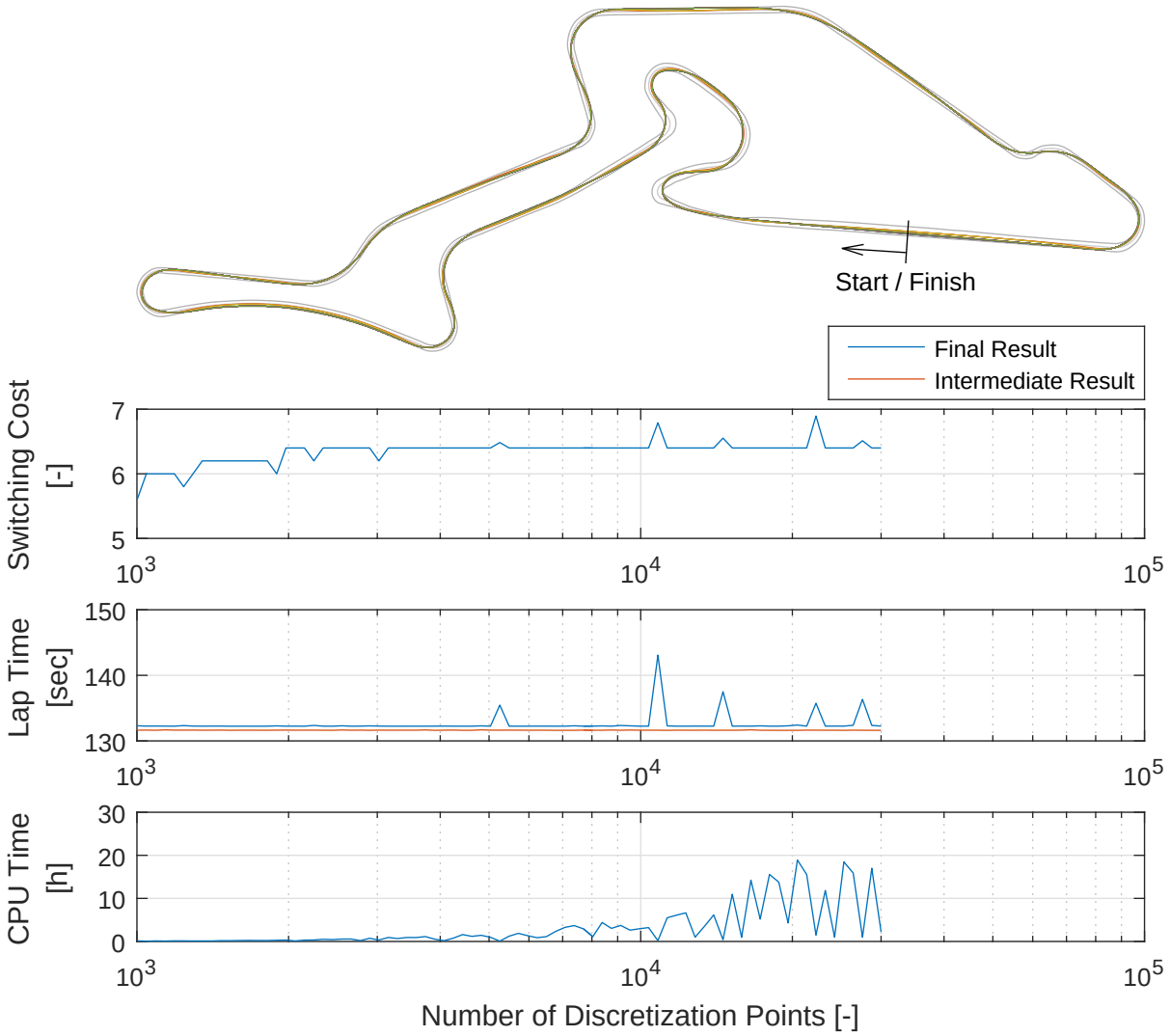


Figure 5.17: Car track, switching cost penalty, lap time and computational time over different discretization densities with spike removal.

5.3.3 Discrete Control Initial Guess

In the previous optimizations, the discrete control is initialized in the third gear. Within this subsection, it shall be evaluated whether the optimized switching structure changes w.r.t. the provided initial guess. Therefore, for each gear selection the optimization is started. Additionally, an optimization is carried out where the discrete control weights are initialized evenly. Since the example consists of five possible gear choices, in the evenly distributed case the weights are initialized with $w_{i,k} = 0.2$. The optimization is carried out with the spike removal augmentation.

Figure 5.19 shows the result of the different initial guesses for the gear selection. The tracks of all six solutions fit almost perfectly. Additionally, the switching penalties, the lap times, as well as the CPU times are plotted w.r.t. the initial guess provided. The integer numbers represent the gear choice. The $1/nv$ tick label indicates evenly initialized weights. In the plots, the results are very consistent. A slightly higher lap time is obtained in case the gear is initialized in the fifth gear. Additionally, the computational time varies significantly.

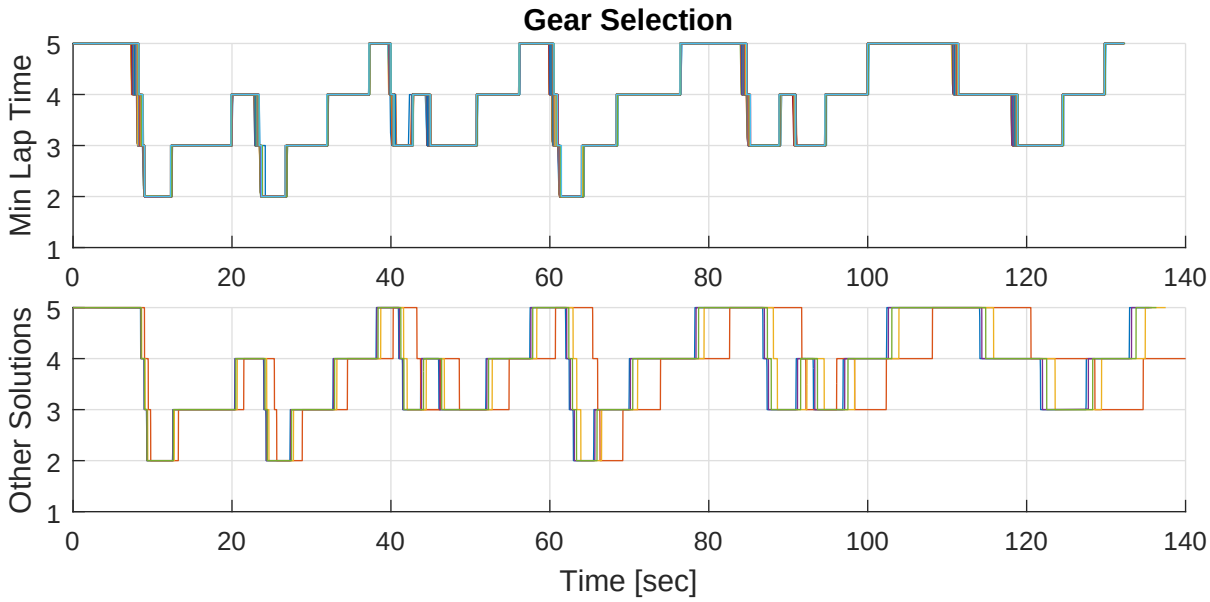


Figure 5.18: Gear selection for the minimal lap time and other solutions with spike removal.

In Figure 5.20, the obtained switching structure after the first optimization stage as well as the final switching structure are shown. As can be seen from the intermediate switching structure which is not yet binary feasible, all first optimization stages are able to find a continuous solution for the gear selection. In general, this assumption cannot be made. Since the intermediate solutions are very similar, it can be expected that the results after the spike removal and the second optimization stage are similar as well.

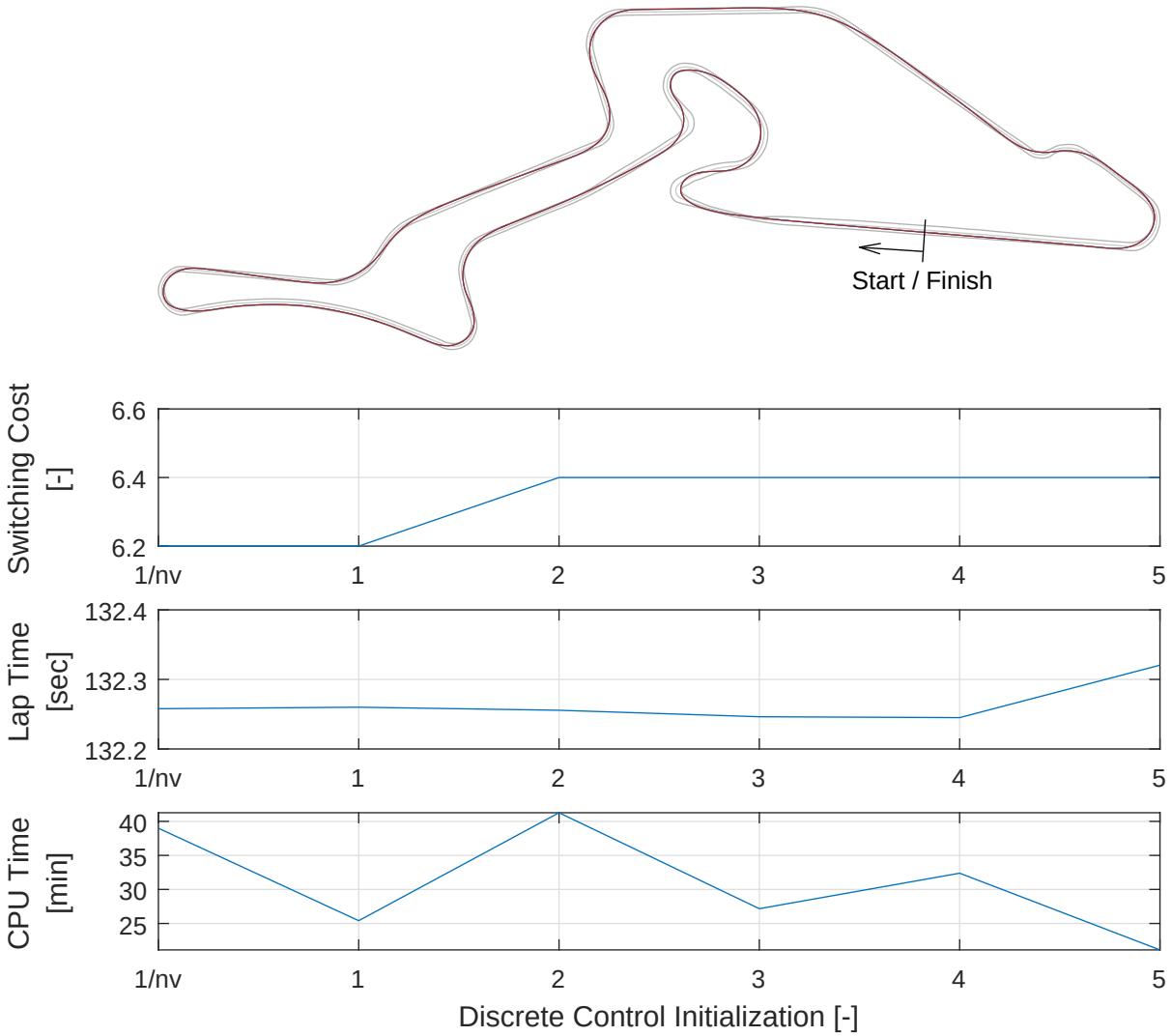


Figure 5.19: Car track, switching cost penalty, lap time, and computational time over different discrete control initializations with spike removal.

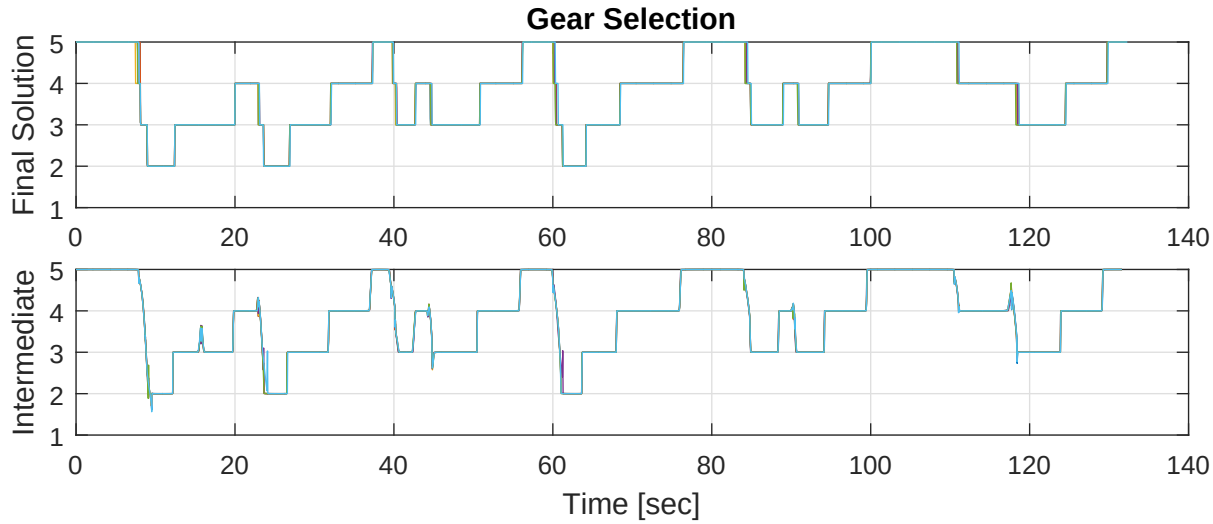


Figure 5.20: Gear selection of intermediate and final solutions for different discrete control initializations with spike removal.

Chapter 6

Application to Aircraft Approach Optimization

In this chapter, the discrete high lift devices as well as the landing gear of a civil aircraft are optimized in an approach scenario. As before, the full state and control histories are optimized and the switching sequences of the discrete controls are subject to optimization. Additionally, the discrete control dependent constraints are considered. The maximum speed limit is dependent on the high lift configuration. Furthermore, the minimum speed limit is influenced by both the high lift and the landing gear selection.

The chapter is organized as follows. In section 6.1, the aircraft model is introduced. The kinematic equations of motions for a Three-Degree Of Freedom (3DOF) point mass model are derived. Additionally, the forces acting on the aircraft are introduced. Constraints that are considered in the optimization are described in section 6.2. These include continuous and discrete constraints as well as operational constraints for a safe approach. The actual aircraft approach problem on runway 26R of Munich airport and the solution strategy are explained in section 6.3. Afterwards, in section 6.4, a single approach optimization is solved. Its solution is used to explain the structure of the time histories. In section 6.5, several parameter studies are carried out. They shall determine the influence of aircraft initial approach parameters (mass, altitude, and speed) , wind (speed, direction), and offsets from the standard atmosphere (temperature, pressure) on the solution of the switching structure. The chapter closes with a comparison to a real flight in section 6.6.

6.1 Aircraft Dynamics

The aircraft model used is taken from the Base of Aircraft Data (BADA) Family 4 by Eurocontrol [19]. It is a Three-Degree Of Freedom (3DOF) point mass dynamic model with sophisticated aerodynamic (flap and landing gear dependent), propulsion, and fuel flow characteristics. Additionally, it contains constraints for various configurations (flaps, landing gear).

Table 6.1 defines the states, controls, and outputs that are used by the BADA 4.0 Family model. Additionally to the states and controls entering the dynamics

$$\dot{\vec{x}} = \vec{f_x}\left(\vec{x}, \vec{u}, \vec{v}_{BADA}\right) \tag{6.1}$$

$$\vec{y} = \vec{f_y}\left(\vec{x}, \vec{u}, \vec{v}_{BADA}\right) \tag{6.2}$$

Table 6.1: States, Controls and Outputs for BADA 4.0 Family Model. Limits considered through a vanishing constraint do not have constant bounds and are indicated by VC.

Name	**Symbol**	**Unit**	**Limit**
	States		
Latitude (WGS84)	ϕ^G	$[rad]$	
Longitude (WGS84)	λ^G	$[rad]$	
Altitude (WGS84)	h^G	$[m]$	
Speed (kinematic)	V_K^G	$[m/s]$	
Course Angle (kinematic)	χ_K^G	$[rad]$	
Climb Angle (kinematic)	γ_K^G	$[rad]$	
Mass	m	$[kg]$	
Thrust Lever Position	δ_T	$[-]$	$[0, 1]$
	Controls		
Lift Coefficient	C_L	$[-]$	VC
Bank Angle (aerodynamic)	μ_A	$[rad]$	$[-30, 30] \cdot \pi/180$
Thrust Lever Position Command	$\delta_{T,CMD}$	$[-]$	$[0, 1]$
	Outputs		
Mach Number	M	$[-]$	
Load Factor (vertical)	n_z	$[-]$	
Aerodynamic Speed	V_A	$[m/s]$	VC
Calibrated Air Speed	V_{CAS}	$[m/s]$	VC
Time Derivative Calibrated Air Speed	$\dot{V}_{CAS}$	$[m/s^2]$	$[-\infty, 0]$

the vector $\vec{v}_{BADA}$ represents aircraft specific data (aerodynamics, propulsion, and fuel flow). By exchanging the vector, different aircraft types can be simulated. In this thesis, the $\vec{v}_{BADA}$ vector is used as a discrete control input.

6.1.1 Coordinate Systems

To describe the motion of the aircraft in a correct manner, multiple coordinate systems have to be considered. In the following, the coordinate systems required for this work are presented together with transformation between them. As a 3DOF model is used, e.g. the body fixed coordinate system is not considered. Coordinate systems and nomenclature are adapted from [119].

Mathematical Preliminaries

A position is given by the vector

$$\left(\vec{r}^{\,P}\right)_A \tag{6.3}$$

where P represents a point and A the coordinate system the position is given relative to. In case the position is differentiated in time w.r.t. another coordinate system B, then

the speed is given by

$$\left(\frac{d}{dt}\right)^{B} \left(\vec{r}^{P}\right)_{A} = \left(\vec{v}_{K}^{P}\right)_{A}^{B} \tag{6.4}$$

where the coordinates are given in the A frame. Additionally, K indicates the value type, in this case a kinematic speed. Other type indicators may be A for aerodynamic data, P for propulsion, G for gravitational, and T for total. A derivative may be w.r.t. multiple coordinate systems. For instance, the acceleration is given by

$$\left(\frac{d}{dt}\right)^{C} \left(\vec{v}_{K}^{P}\right)_{A}^{B} = \left(\vec{a}_{K}^{P}\right)_{A}^{BC} \tag{6.5}$$

where the speed is relative to the B frame and the acceleration relative to the C frame.

The angular speed

$$\left(\vec{\omega}^{AB}\right)_{C} \tag{6.6}$$

states the rotation of the B frame relative to the A frame. The coordinates are given in the C frame. The aerodynamic (A) Force

$$\left(\vec{F}_{A}^{P}\right)_{B} \tag{6.7}$$

acts in the point P and is given in the B frame. Finally, the transformation matrix

$$\left(\vec{r}^{P}\right)_{A} = M_{AB} \cdot \left(\vec{r}^{P}\right)_{B} \tag{6.8}$$

transforms a vector from the B frame to the A frame.

Earth Centered Inertial (ECI)

The aircraft dynamics are derived by the conversion of momentum principle. It can only be applied in a non-accelerated inertial reference coordinate system. Here, the Earth Centered Inertial (ECI) frame is used. This frame's origin is fixed to the earth's center but does not rotate with it (see Fig. 6.1). The x_I axis points to the vernal equinox, the z_I axis lies in the earth's rotational axis with positive direction to geographic north, and the y_I axis completes the right hand coordinate system. Strictly speaking, this frame moves around the sun and with the solar system. However, these effects are very small and thus neglected. All variables given w.r.t. the ECI frame are denoted by the index I.

Earth Centered Earth Fixed (ECEF)

The Earth Centered Earth Fixed (ECEF) purpose is for positioning w.r.t to the earth. This coordinate system has the same origin as the ECI frame but is fixed to the earth and thus rotates with it (see Fig. 6.1). The x_E axis lies in the equatorial plane and points to the Greenwich meridian. The z_I,z_E axes of ECI and ECEF are identical. As before, the y_E axis completes the right hand coordinate system. The angular speed is given by

$$\left(\vec{\omega}^{IE}\right) = \begin{pmatrix} 0 \\ 0 \\ 7292155.0 \cdot 10^{-11} \end{pmatrix} \frac{rad}{s} \approx \begin{pmatrix} 0 \\ 0 \\ \frac{2 \cdot \pi}{24 \cdot 3600} \end{pmatrix} \frac{rad}{s} \tag{6.9}$$

and results in one revolution in 24 hours. The ECEF frame is denoted by the index E.

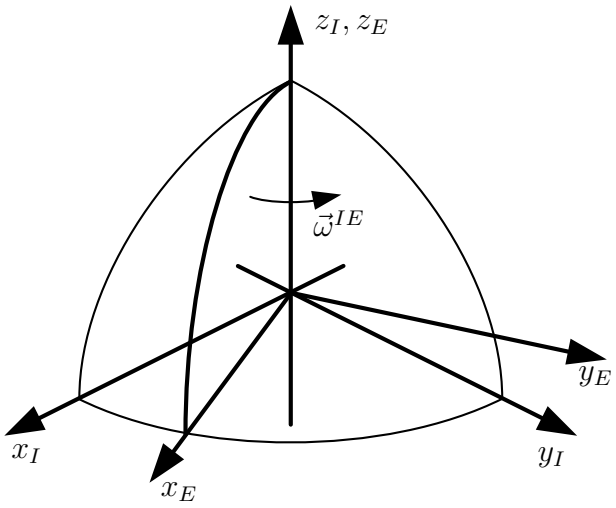


Figure 6.1: Earth centered inertial and earth centered earth fixed frames.

Table 6.2: Defining parameters of the World Geodetic System 1984 [120].

Parameter	**Symbol**	**Value**	**Unit**
Semi-major Axis	a	6378137.0	m
Reciprocal of Flattening	$1/f$	298.257223563	$-$
Angular Velocity of the Earth	$\vec{\omega}^{IE}$	$7292115.0 \times 10^{-11}$	rad/s
Earth's Gravitational Constant	GM	$3986004.418 \times 10^{8}$	m^3/s^2

World Geodetic System 1984 (WGS84)

The World Geodetic System 1984 (WGS84) was introduced as a common position system on earth published by National Imagery and Mapping Ageny (NIMA) [120]. Apart from gravitational information, WGS84 defines a reference ellipsoid used for navigation (e.g. Global Positioning System (GPS)). The WGS84 position system is defined by four parameters as shown in Table 6.2.

From the basic parameters, others can be derived. The semi-minor axis

$$b = a \cdot (1 - f) \tag{6.10}$$

defines the radius of the ellipsoid at the poles. Another important constant is the eccentricity

$$e = \sqrt{1 - \frac{b^2}{a^2}} \tag{6.11}$$

which defines the distance of the ellipse's focal points from the center. Since for planetary examples b is always smaller than a the eccentricity is a real value.

A position in the WGS84 frame is given by the latitude ϕ^G, the longitude λ^G, and the altitude h^G. It is defined relative to the ECEF frame. Figure 6.2 shows the ECEF frame together with the WGS84 position. The longitude λ^G is the angle between the zero-meridian plane and the meridian plane of a point P measured in the equatorial plane. The latitude is measured in the meridian plane as the angle between the equatorial plane and the surface normal of a point P. Due to the fact that the ellipsoid is not a

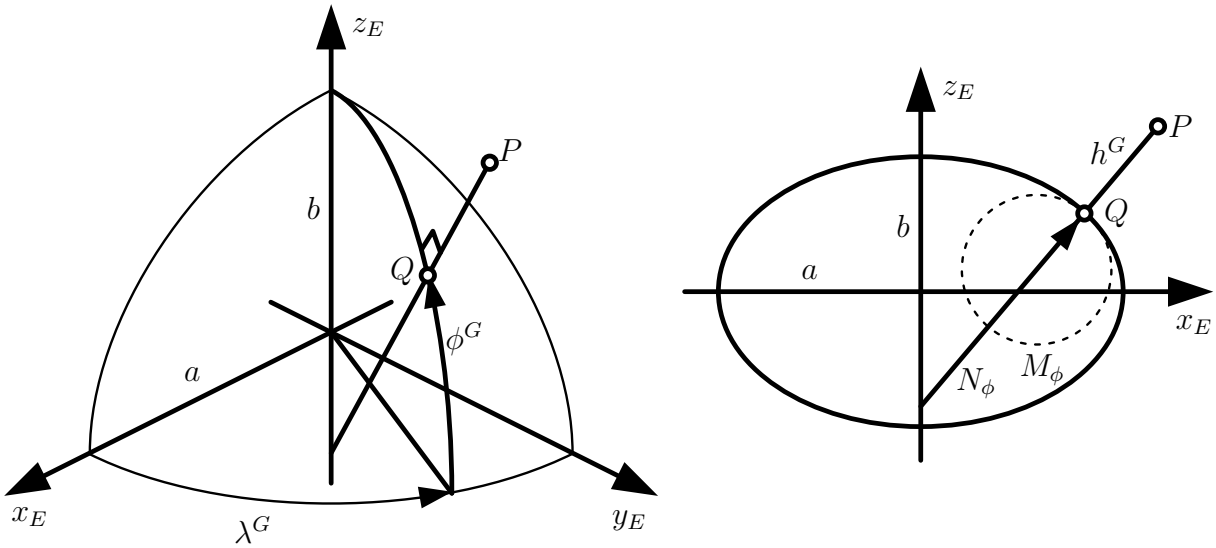


Figure 6.2: Reference frame of the World Geodetic System 1984.

sphere, the surface normal does not necessarily pass through the ellipsoid's center. The altitude is given as the height above the reference ellipsoid along the surface normal.

The side-view of Figure 6.2 shows two additional measurements. The distance perpendicular from the ellipsoids surface (point Q) to the interception with the polar axis z_E is given by

$$N_\phi = \frac{a}{\sqrt{1 - e^2 \cdot \sin^2 \phi^G}} \tag{6.12}$$

which is dependent on the latitude. The radius

$$M_\phi = N_\phi \cdot \frac{1 - e^2}{1 - e^2 \cdot \sin^2 \phi^G} \tag{6.13}$$

describes a circle that has the same curvature as the ellipsoid in point Q in the meridian plane. These two distances are relevant for position propagation in the WGS84 frame (see section 6.1.4)

North-East-Down (NED)

The North-East-Down (NED) frame is used to determine the orientation of the aircraft and is denoted by the index O. The frame's origin is attached to the reference point of the aircraft. Since the aircraft model considered in this thesis is a 3DOF model, the reference point is the center of gravity G. As the name suspects, the frame axes are oriented w.r.t. the directions of the reference ellipsoid where the x_O axis points to geographical north, the y_O to the east, and the z_O axis points perpendicular to the ground. Figure 6.3 shows the alignment of the NED frame. To maintain the NED orientation while the aircraft position changes, this reference frame rotates with the so-called transport rate

$$\left(\vec{\omega}^{EO}\right)_O = \begin{pmatrix} \dot{\lambda}^G \cdot \cos \phi^G \\ -\dot{\phi}^G \\ -\dot{\lambda}^G \cdot \sin \phi^G \end{pmatrix}_O . \tag{6.14}$$

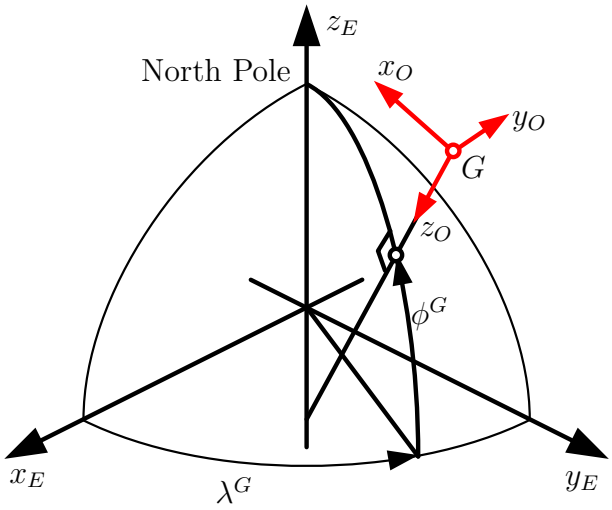


Figure 6.3: North east down reference frame w.r.t. the ECEF frame.

The orientation of the NED frame is dependent on the current latitude ϕ^G and longitude λ^G. Therefore, the axis transformation from the ECEF frame to the NED frame

$$M_{OE} = \begin{pmatrix} -\sin\phi^G \cdot \cos\lambda^G & -\sin\phi^G \cdot \sin\lambda^G & \cos\phi^G \\ -\sin\lambda^G & \cos\lambda^G & 0 \\ -\cos\phi^G \cdot \cos\lambda^G & -\cos\phi^G \cdot \sin\lambda^G & -\sin\phi^G \end{pmatrix} \tag{6.15}$$

is calculate using these angles. The first rotation occurs around the z_E axis with the angle λ^G. The second rotation is achieved around the y_O axis with the angle $\phi^G + \frac{\pi}{2}$.

Kinematic Frame

The kinematic frame (denoted by K) is used to describe the aircraft's velocity and has its origin in the center of gravity G of the points mass. This frame is aligned with the velocity vector of the aircraft (see Figure 6.4) where the x_K axis points in the direction of the current velocity, z_K points downward perpendicular to x_K in the plane that is spanned by the x_K axis, and the z_O axis. y_K fulfills the right hand rule.

The orientation of the kinematic frame w.r.t. the NED frame is described by two angles. The kinematic course angle χ_K^G states the direction the aircraft is traveling. The inclination of the flight path is given by the kinematic climb angle γ_K^G. Thus, the transformation of the NED frame (O) to the K frame is given by

$$M_{KO} = \begin{pmatrix} \cos\chi_K^G \cdot \cos\gamma_K^G & \sin\chi_K^G \cdot \cos\gamma_K^G & -\sin\gamma_K^G \\ -\sin\chi_K^G & \cos\chi_K^G & 0 \\ \cos\chi_K^G \cdot \sin\gamma_K^G & \sin\chi_K^G \cdot \sin\gamma_K^G & \cos\gamma_K^G \end{pmatrix}. \tag{6.16}$$

It combines the first rotation around the z_O axis by χ_K^G and the second rotation around the y_K axis by γ_K^G. The rotation of the kinematic frame w.r.t. the NED frame is stated by

$$\left(\vec{\omega}^{EO}\right)_K = \begin{bmatrix} -\dot{\chi}_K^G \cdot \sin\gamma_K^G \\ \dot{\gamma}_K^G \\ \dot{\chi}_K^G \cdot \cos\gamma_K^G \end{bmatrix}. \tag{6.17}$$

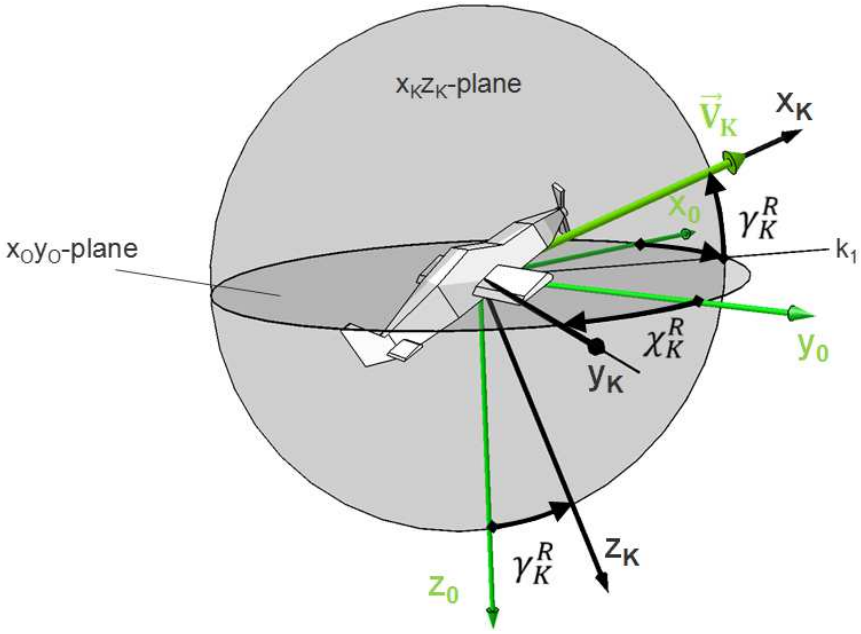


Figure 6.4: Kinematic reference frame w.r.t. the NED frame [119].

Aerodynamic Frame

This frame (denoted by A) has its origin in the center of gravity G and is used to calculate aerodynamic forces. The orientation is given relative to the NED frame as shown in Figure 6.5. The x_A axis points in the direction of the aerodynamic flow, the z_A axis points downwards perpendicular to x_A in the plane that is spanned by the x_A and the z_O axes. y_A fulfills the right hand rule. It has to be noted that the directions of y_A and z_A are only correct for an aerodynamic bank angle of $\mu_A = 0$. In case of $\mu_A \neq 0$, the aerodynamic frame is rotated a third time around the axis x_A by the angle μ_A.

Thus, the transformation from the NED frame into the A is similar to the kinematic case but with an additional rotation by μ_A around the x_A axis. The transformation is given by

$$M_{AO} = \begin{pmatrix} c_\chi \cdot c_\gamma & s_\chi \cdot c_\gamma & -s_\gamma \\ c_\chi \cdot s_\gamma \cdot s_\mu - s_\chi \cdot c_\mu & s_\chi \cdot s_\gamma \cdot s_\mu + c_\chi \cdot c_\mu & c_\gamma \cdot s_\mu \\ c_\chi \cdot s_\gamma \cdot c_\mu + s_\chi \cdot s_\mu & s_\chi \cdot s_\gamma \cdot c_\mu - c_\chi \cdot s_\mu & c_\gamma \cdot c_\mu \end{pmatrix} \tag{6.18}$$

where placeholders

$$\begin{aligned} s_\chi &= \sin\chi_A & s_\gamma &= \sin\gamma_A & s_\mu &= \sin\mu_A \\ c_\chi &= \cos\chi_A & c_\gamma &= \cos\gamma_A & c_\mu &= \cos\mu_A \end{aligned}$$

are used to bring the matrix into a single line. The coordinate systems and their rotations are now used to derive the point mass equations of motion.

6.1.2 Equations of Motion

In this section, the translation equations of motion of the point mass model are derived. Detailed information can be found for instance in [119, 121, 122]. The derivation of the equations of motion begins with NEWTON's second law [123]. It describes the formula

$$\vec{F} = m \cdot \vec{a} \tag{6.19}$$

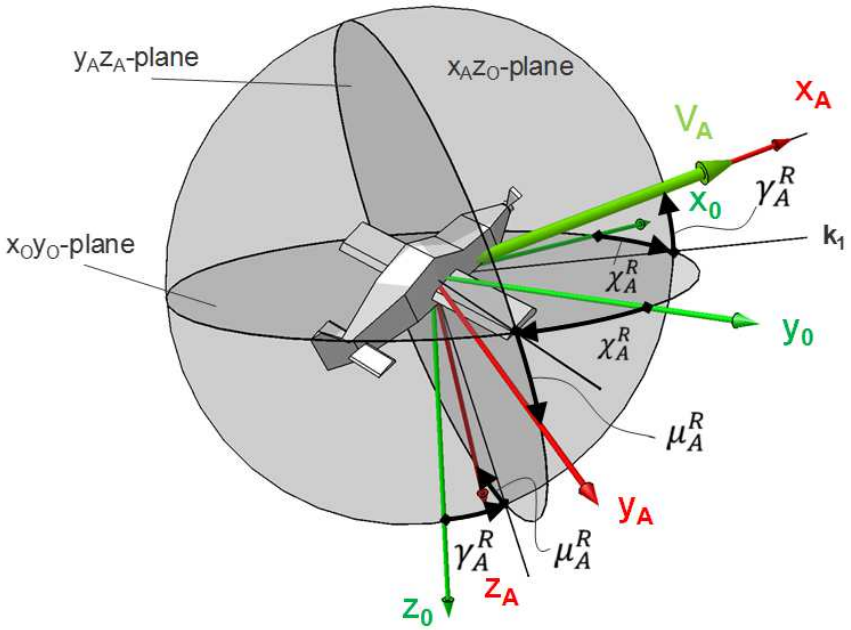


Figure 6.5: Aerodynamic reference frame w.r.t. the NED frame [119].

which states that the acceleration $\vec{a}$ of an object in the inertial frame is equal to the sum of forces $\vec{F}$ acting on the object divided by its mass m. In more general terms, this equation can be expanded to include mass changes [124] . This leads to the time derivative of the momentum p of a mass

$$\vec{F} = \left(\frac{d}{dt}\right)^I (p)^I = \left(\frac{d}{dt}\right)^I \left[m \cdot \left(\vec{v}_K^G\right)^I\right] \tag{6.20}$$

where the derivative is taken w.r.t. the inertial frame I (here ECI). $\left(\vec{v}_K^G\right)^I$ states the kinematic velocity of the mass in I. For aircraft, the change of mass is usually very small. Additionally, the mass flow leaving the aircraft is included into the propulsion force. Thus, it can be omitted from (6.20)

$$\sum \left(\vec{F}_T^G\right) = m \cdot \left(\dot{\vec{v}}_K^G\right)^{II} \tag{6.21}$$

leading back to (6.19). Two variables need to be determined. The sum of forces acting on the aircraft will be discussed further below (see section 6.1.3). The derivation of the acceleration w.r.t. the inertial ECI frame is subject of this section.

Since the derivation incorporates different coordinate systems, the EULER differentiation is used [119]. Assuming a time derivative of a vector $\vec{r}$ w.r.t. a coordinate system A shall be computed

$$\left(\frac{d}{dt}\right)^A (\vec{r})_B \tag{6.22}$$

but the coordinates are given w.r.t the coordinate system B. Then, the derivative

$$\left(\dot{\vec{r}}\right)_B^A = \left(\dot{\vec{r}}\right)_B^B + \left(\vec{\omega}^{AB}\right)_B \times (\vec{r})_B \tag{6.23}$$

is calculated by dividing it into two parts. The original vector $\vec{r}$ is differentiated w.r.t. the coordinate system B. Additionally, a cross product is added where $\left(\vec{\omega}^{AB}\right)_B$ states

the rotation of the B frame w.r.t. the A frame with coordinates given in the B frame. The equation (6.23) is derived in C.2.

To obtain the point mass acceleration in the inertial frame, the position has to be differentiated w.r.t. time twice. The equation of motion itself is independent from the aircraft type and can be used for other applications as well. All forces are aircraft / application dependent.

It is assumed that the point mass position

$$\left(\vec{r}^{G}\right)_{E} \tag{6.24}$$

is given in the ECEF frame. The translation equation of motion can be derived without a reference to a coordinate system in which the variables are stated. This assumption holds due to the fact that the derived formulas can always be transformed into any desired coordinate system. However, since the aircraft position is normally given w.r.t. the earth coordinate system (6.24) this is a plausible assumption. Additionally, it is easier to follow the derivations since the EULER differentiation approach is strictly applied. A similar derivation can be found in [119].

The speed of the point mass is calculated by taking the first time derivative of the position within the inertial reference frame ECI

$$\left(\frac{d}{dt}\right)^{I}\left(\vec{r}^{G}\right)_{E} = \left(\dot{\vec{r}}_{K}^{G}\right)_{E}^{I} = \left(\vec{v}_{K}^{G}\right)_{E}^{I} = \left(\vec{v}_{K}^{G}\right)_{E}^{E} + \left(\vec{\omega}^{IE}\right)_{E} \times \left(\vec{r}^{G}\right)_{E}. \tag{6.25}$$

The first term gives the velocity vector w.r.t. the earth fixed coordinated system. The EULER part accounts for the velocity induced due to the earth rotation.

Since the second derivative (acceleration) in the inertial reference frame is required, (6.25) is differentiated a second time:

$$\left(\frac{d}{dt}\right)^{I}\left(\vec{v}_{K}^{G}\right)_{E}^{I} = \left(\dot{\vec{v}}_{K}^{G}\right)_{E}^{II} = \left(\frac{d}{dt}\right)^{I}\left[\left(\vec{v}_{K}^{G}\right)_{E}^{E} + \left(\vec{\omega}^{IE}\right)_{E} \times \left(\vec{r}^{G}\right)_{E}\right]. \tag{6.26}$$

The derivative of (6.26) can be split into three parts

$$\left(\frac{d}{dt}\right)^{I}\left(\vec{v}_{K}^{G}\right)_{E}^{E} = \left(\dot{\vec{v}}_{K}^{G}\right)_{E}^{EE} + \left(\vec{\omega}^{IE}\right)_{E} \times \left(\vec{v}_{K}^{G}\right)_{E}^{E} \tag{6.27}$$

$$\left[\left(\frac{d}{dt}\right)^{I}\left(\vec{\omega}^{IE}\right)_{E}\right] \times \left(\vec{r}^{G}\right)_{E} = \left[\left(\dot{\vec{\omega}}^{IE}\right)_{E}^{E} + \left(\vec{\omega}^{IE}\right)_{E} \times \left(\vec{\omega}^{IE}\right)_{E}\right] \times \left(\vec{r}^{G}\right)_{E} \tag{6.28}$$

$$\left(\vec{\omega}^{IE}\right)_{E} \times \left[\left(\frac{d}{dt}\right)^{I}\left(\vec{r}^{G}\right)_{E}\right] = \left(\vec{\omega}^{IE}\right)_{E} \times \left(\vec{v}_{K}^{G}\right)_{E}^{E} + \left(\vec{\omega}^{IE}\right)_{E} \times \left[\left(\vec{\omega}^{IE}\right)_{E} \times \left(\vec{r}^{G}\right)_{E}\right] \tag{6.29}$$

if the product rule is applied for the cross product. Please note that the cross product does not fulfill the associative property. For the description of the aircraft dynamics it can be assumed that the rate of change of the earth rotation

$$\left(\dot{\vec{\omega}}^{IE}\right)_{E}^{E} = \vec{0} \tag{6.30}$$

is equal to zero. Additionally, the cross product of $\left(\vec{\omega}^{IE}\right)_{E} \times \left(\vec{\omega}^{IE}\right)_{E}$ equals zero as well. Thus, (6.26) becomes

$$\left(\dot{\vec{v}}_{K}^{G}\right)_{E}^{II} = \left(\dot{\vec{v}}_{K}^{G}\right)_{E}^{EE} + 2 \cdot \left(\vec{\omega}^{IE}\right)_{E} \times \left(\vec{v}_{K}^{G}\right)_{E}^{E} + \left(\vec{\omega}^{IE}\right)_{E} \times \left[\left(\vec{\omega}^{IE}\right)_{E} \times \left(\vec{r}^{G}\right)_{E}\right] \tag{6.31}$$

which includes the formulation of the local acceleration, the CORIOLIS acceleration, and the centripetal acceleration.

In (6.25) the speed was derived w.r.t. the ECEF frame. The acceleration of the aircraft shall be given w.r.t. to the kinematic frame. This makes it easier to formulate the differential equation for the kinematic variables. To achieve this, (6.31) has to be transformed

$$\left(\dot{\vec{v}}_K^G\right)_K^{II} = \left(\dot{\vec{v}}_K^G\right)_K^{EE} + M_{KE}\left[2\cdot\left(\vec{\omega}^{IE}\right)_E \times \left(\vec{v}_K^G\right)_E^E + \left(\vec{\omega}^{IE}\right)_E \times \left[\left(\vec{\omega}^{IE}\right)_E \times \left(\vec{r}^G\right)_E\right]\right] \tag{6.32}$$

into the kinematic frame using the transformation matrix

$$M_{KE} = M_{KO}\cdot M_{OE}. \tag{6.33}$$

Additionally, the acceleration w.r.t. the earth frame is reformulated

$$\left(\dot{\vec{v}}_K^G\right)_K^{EE} = \left(\dot{\vec{v}}_K^G\right)_K^{EK} + \left(\vec{\omega}^{EK}\right)_K \times \left(\vec{v}_K^G\right)_K^E \tag{6.34}$$

by applying the EULER differentiation. Furthermore, the rotation of the K frame relative to the E frame

$$\left(\vec{\omega}^{EK}\right)_K = \left(\vec{\omega}^{EO}\right)_K + \left(\vec{\omega}^{OK}\right)_K \tag{6.35}$$

is split into two rotations. The local acceleration is reformulated to

$$\left(\dot{\vec{v}}_K^G\right)_K^{EE} = \left(\dot{\vec{v}}_K^G\right)_K^{EK} + \left(\vec{\omega}^{EO}\right)_K \times \left(\vec{v}_K^G\right)_K^E + \left(\vec{\omega}^{OK}\right)_K \times \left(\vec{v}_K^G\right)_K^E \tag{6.36}$$

which is resubstituted into (6.32):

$$\begin{aligned}\left(\dot{\vec{v}}_K^G\right)_K^{II} &= \left(\dot{\vec{v}}_K^G\right)_K^{EE} + M_{KE}\left[2\cdot\left(\vec{\omega}^{IE}\right)_E \times \left(\vec{v}_K^G\right)_E^E + \left(\vec{\omega}^{IE}\right)_E \times \left[\left(\vec{\omega}^{IE}\right)_E \times \left(\vec{r}^G\right)_E\right]\right] \\ &= \left(\dot{\vec{v}}_K^G\right)_K^{EK} + \left(\vec{\omega}^{EO}\right)_K \times \left(\vec{v}_K^G\right)_K^E + \left(\vec{\omega}^{OK}\right)_K \times \left(\vec{v}_K^G\right)_K^E \\ &\quad + M_{KE}\left[2\cdot\left(\vec{\omega}^{IE}\right)_E \times \left(\vec{v}_K^G\right)_E^E + \left(\vec{\omega}^{IE}\right)_E \times \left[\left(\vec{\omega}^{IE}\right)_E \times \left(\vec{r}^G\right)_E\right]\right].\end{aligned} \tag{6.37}$$

Thus, a relation between the point mass acceleration in the inertial frame I and the acceleration in the kinematic frame K where the speed is given w.r.t. the earth fixed frame E is created.

The derived acceleration (6.37) is inserted into Newton II (6.21)

$$\begin{aligned}\frac{1}{m}\sum\left(\vec{F}_T^G\right)_K &= \left(\dot{\vec{v}}_K^G\right)_K^{II} \\ &= \left(\dot{\vec{v}}_K^G\right)_K^{EE} + M_{KE}\left[2\cdot\left(\vec{\omega}^{IE}\right)_E \times \left(\vec{v}_K^G\right)_E^E + \left(\vec{\omega}^{IE}\right)_E \times \left[\left(\vec{\omega}^{IE}\right)_E \times \left(\vec{r}^G\right)_E\right]\right] \\ &= \left(\dot{\vec{v}}_K^G\right)_K^{EK} + \left(\vec{\omega}^{EO}\right)_K \times \left(\vec{v}_K^G\right)_K^E + \left(\vec{\omega}^{OK}\right)_K \times \left(\vec{v}_K^G\right)_K^E + M_{KE}\left[\dots\right]\end{aligned} \tag{6.38}$$

and the equation is rearranged:

$$\left(\dot{\vec{v}}_K^G\right)_K^{EK} + \left(\vec{\omega}^{OK}\right)_K \times \left(\vec{v}_K^G\right)_K^E = \frac{1}{m}\sum\left(\vec{F}_T^G\right)_K - \left(\vec{\omega}^{EO}\right)_K \times \left(\vec{v}_K^G\right)_K^E - M_{KE}\left[\dots\right]. \tag{6.39}$$

If the left hand side of (6.39) is expanded with the actual entries

$$\begin{aligned}\left(\dot{\vec{v}}_K^G\right)_K^{EK} + \left(\vec{\omega}^{OK}\right)_K \times \left(\vec{v}_K^G\right)_K^E &= \begin{bmatrix} \dot{V}_K^G \\ 0 \\ 0 \end{bmatrix}_K^{EK} + \begin{bmatrix} -\left(\dot{\chi}_K^G\right) \cdot \sin\left(\gamma_K^G\right) \\ \left(\dot{\gamma}_K^G\right) \\ \left(\dot{\chi}_K^G\right) \cdot \cos\left(\gamma_K^G\right) \end{bmatrix}_K \times \begin{bmatrix} V_K^G \\ 0 \\ 0 \end{bmatrix}_K^E \\ &= \begin{bmatrix} \left(\dot{V}_K^G\right)^{EK} \\ \left(\dot{\chi}_K^G\right) \cdot \cos\left(\gamma_K^G\right) \cdot \left(V_K^G\right)^E \\ -\left(\dot{\gamma}_K^G\right) \cdot \left(V_K^G\right)^E \end{bmatrix}_K \end{aligned} \tag{6.40}$$

a formulation which includes the time derivatives for the kinematic speed, course angle, and climb angle is found.

The right hand size of (6.39) states the forces acting on the point mass and the correction due to the transport rate of the NED frame. This correction is reformulated

$$\left(\vec{\omega}^{EO}\right)_K \times \left(\vec{v}_K^G\right)_K^E = M_{KO} \cdot \left[\left(\vec{\omega}^{EO}\right)_O \times \left(\vec{v}_K^G\right)_O^E\right] \tag{6.41}$$

to calculate the cross product in the NED reference frame. Thus, the equations of motion

$$\begin{aligned}\begin{bmatrix} \left(\dot{V}_K^G\right)^{EK} \\ \left(\dot{\chi}_K^G\right)^K \cdot \cos\left(\gamma_K^G\right) \cdot \left(V_K^G\right)^E \\ -\left(\dot{\gamma}_K^G\right)^K \cdot \left(V_K^G\right)^E \end{bmatrix}_K &= \frac{1}{m} \sum \begin{bmatrix} X_T^G \\ Y_T^G \\ Z_T^G \end{bmatrix}_K - M_{KO} \cdot \left[\left(\vec{\omega}^{EO}\right)_O \times \left(\vec{v}_K^G\right)_O^E\right] \\ &+ M_{KE}\left\{2 \cdot \left(\vec{\omega}^{IE}\right)_E \times \left(\vec{v}_K^G\right)_E^E\right\} \\ &+ M_{KE}\left\{\left(\vec{\omega}^{IE}\right)_E \times \left[\left(\vec{\omega}^{IE}\right)_E \times \left(\vec{r}^G\right)_E\right]\right\}\end{aligned} \tag{6.42}$$

are given w.r.t. to the inertial frame I. However, in this thesis some simplifications are made. The time horizon on which the aircraft is simulated is relatively short. Since the earth angular speed is small, the influence in the equation of motion can be neglected. Thus, the point mass translation equations of motions become:

$$\begin{bmatrix} \left(\dot{V}_K^G\right)^{EK} \\ \left(\dot{\chi}_K^G\right)^K \cdot \cos\left(\gamma_K^G\right) \cdot \left(V_K^G\right)^E \\ -\left(\dot{\gamma}_K^G\right)^K \cdot \left(V_K^G\right)^E \end{bmatrix}_K = \frac{1}{m} \sum \begin{bmatrix} X_T^G \\ Y_T^G \\ Z_T^G \end{bmatrix}_K - M_{KO} \cdot \left[\left(\vec{\omega}^{EO}\right)_O \times \left(\vec{v}_K^G\right)_O^E\right]. \tag{6.43}$$

The round earth notation is still used due to the fact that real word applications use WGS84 coordinates. The next section introduces the forces acting on the aircraft.

6.1.3 Aircraft Forces

In the previous section, the equation of motion were derived. These are the same for any type of aircraft. For the resulting equation (6.43), the total aircraft specific force is required. Common forces for a fixed wing aircraft are:

- aerodynamic forces

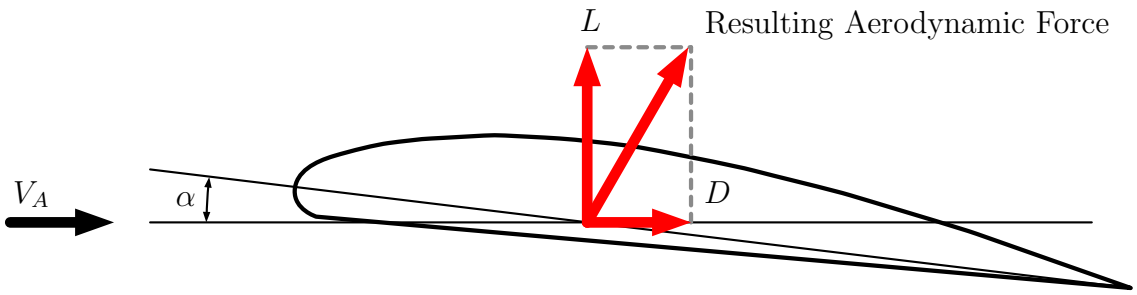


Figure 6.6: Lift and Drag Aerodynamic Forces

- propulsion forces
- gravitation force
- ground forces acting on the landing gear

In course of this thesis the aircraft is regarded to be "in the air". Therefore, ground forces are not considered. The aircraft model used is the Base of Aircraft Data (BADA) Family 4 from Eurocontrol [19]. Although the model is popular, the mathematical formulations of the forces as well as the model parameters are not open to the general public. Therefore, for more information please refer to Eurocontrol.

Aerodynamic Force

As the aircraft moves through the air, the flow of air around the wings and fuselage create forces that act on it. The aerodynamic force is described in the aerodynamic coordinate system A. Since the forces for the equations of motions are required in the kinematic frame, a coordinate transformation

$$\left(\vec{F}_A^G\right)_K = M_{KO} \cdot {M_{AO}}^T \cdot \left(\vec{F}_A^G\right)_A \tag{6.44}$$

is applied. Aerodynamic forces

$$\left(\vec{F}_A^G\right)_A = \begin{bmatrix} -D \\ Q \\ -L \end{bmatrix} = \bar{q} \cdot S_{ref} \cdot \begin{bmatrix} -C_D \\ C_Q \\ -C_L \end{bmatrix}_A \tag{6.45}$$

are usually modeled using force coefficients (C_D, C_Q, C_L) which are unit-less. The actual forces are recalculated with the reference wing area S_{ref} and the dynamic pressure

$$\bar{q} = \frac{1}{2} \cdot \rho \cdot {V_A}^2 \tag{6.46}$$

that depends on the current air density ρ and aerodynamic speed V_A.

Figure 6.6 shows the main aerodynamic forces acting on the aircraft's wing. By manipulation of the wing's pitch angle, the aircraft changes the angle of attack α. It describes the angle between the wing's x-axis and the aerodynamic airflow. The angle of attack influences how the air flows around the wing and thus force acting on it. This force can be distributed into a lift component and a drag component (see Figure 6.6).

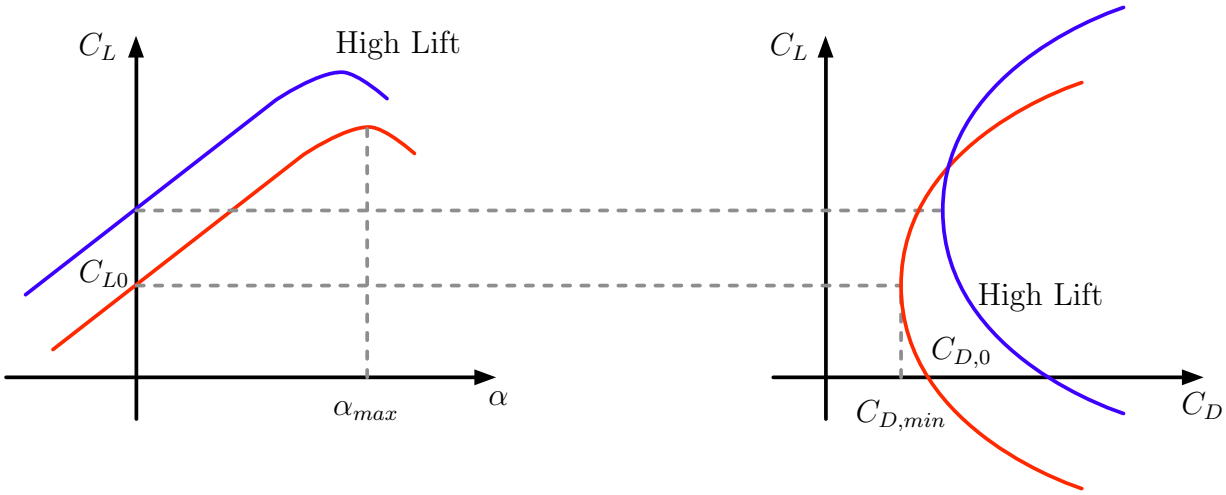


Figure 6.7: Lift and Drag Coefficients [119]

The lift coefficient C_L mainly depends on the angle of attack. For smaller values of α the relationship

$$C_L = C_{L0} + C_{L,\alpha} \cdot \alpha \tag{6.47}$$

can be assumed to be linear. Fig. 6.7 shows the lift coefficient over the angle of attack. For higher angles of attack, the relationship becomes nonlinear. Above α_{max}, the lift decreases resulting in an aircraft stall. The maximum lift coefficient $C_{L,max}$ is increased by deploying the high lift devices. The BADA Family 4 model does not include a formula for (6.47). However, for the different high lift configurations, the maximum lift coefficients are stated. Therefore, in this thesis, the lift coefficient acts as an input to the dynamic model.

The drag coefficient C_D mainly depends on the current lift. A common relationship used in simple aerodynamic models is the quadratic polar

$$C_D = C_{D,0} + C_{D,2} \cdot (C_L - C_{L0})^2 \tag{6.48}$$

that approximates the drag of the aircraft with a parabola as shown in figure 6.7. The drag can be distributed into two main components. The zero lift drag coefficient $C_{D,0}$ accounts for the direct drag of the aircraft due to the cross-section area that faces the airflow. This component is dependent only on the current high lift setting and does not change with the lift. On the other hand, the induced drag coefficient $C_{D,2}$ depends on the current lift coefficient. The BADA Family 4 includes a aerodynamic model for the drag. The drag coefficient

$$C_D = f\left(C_L, M, \delta_{HL}, \delta_{LG}, \delta_{SB}\right) \tag{6.49}$$

is not only dependent on the current lift coefficient C_L but also on the mach number M, the high lift δ_{HL}, the landing gear δ_{LG}, and the speed brakes δ_{SB}.

Propulsion

The engine's thrust is necessary to oppose the drag of the aircraft. For traditional fixed wing aircraft, the engines are aligned with the longitudinal axis of the aircraft. The

thrust vector is approximately parallel to it as well. Since the longitudinal aircraft axis almost aligns with the x-axis of the aerodynamic frame, the thrust vector

$$\left(\vec{F}_P^G\right)_A = \begin{bmatrix} T \\ 0 \\ 0 \end{bmatrix}_A \tag{6.50}$$

is assumed to point in the direction of the aerodynamic speed. The BADA Family 4 model includes a highly sophisticated and complex thrust model

$$T = W_{MTOW} \cdot \eta_p \cdot C_T, \qquad C_T = f(M, \delta_T) \tag{6.51}$$

which is dependent on the maximum takeoff weight W_{MTOW}, the pressure ratio η_p, the Mach number M, and the throttle position δ_T. C_T is a thrust coefficient that is used in the fuel flow calculation as well.

As with the aerodynamic forces, the propulsion force must be transformed into the kinematic frame K. Thus, the transformation via the NED frame

$$\left(\vec{F}_P^G\right)_K = M_{KO} \cdot {M_{AO}}^T \cdot \left(\vec{F}_P^G\right)_A \tag{6.52}$$

is applied.

Gravitation

The third force that acts on the aircraft is the gravitation force. It acts along the z-axis of the NED frame

$$\left(\vec{F}_G^G\right)_O = \begin{bmatrix} 0 \\ 0 \\ m \cdot g \end{bmatrix}_O. \tag{6.53}$$

The gravitational acceleration g is assumed to be constant since location changes of the aircraft in this thesis are assumed to be small. The gravitational force is transformed

$$\left(\vec{F}_G^G\right)_K = M_{KO} \cdot \left(\vec{F}_G^G\right)_O. \tag{6.54}$$

into the kinematic frame using the transformation matrix M_{KO}.

6.1.4 Aircraft Dynamics Subsystems

In this section, the formulas used for the description of the dynamic model are presented. The subsystem derivative approach requires the model to be split into multiple subsystems. Therefore, every subsystem is introduced separately. Some of the subsystems can be combined to a single subsystem. However, they were designed to be reusable for other aircraft models.

Subsystem Kinematic Speed

The speed of the aircraft w.r.t. the earth is given in the Kframe

$$\left(\vec{v}_K^G\right)_K^E = \begin{bmatrix} V_K^G \\ 0 \\ 0 \end{bmatrix}_K^E \tag{6.55}$$

which is transformed to the NED frame

$$\begin{bmatrix} u_K^G \\ v_K^G \\ w_K^G \end{bmatrix}_O^E = \left(\vec{v}_K^G\right)_O^E = {M_{KO}}^T \left(\vec{v}_K^G\right)_K^E \tag{6.56}$$

$$= \begin{bmatrix} V_K^G \cdot \cos\chi_K^G \cdot \cos\gamma_K^G \\ V_K^G \cdot \sin\chi_K^G \cdot \cos\gamma_K^G \\ -V_K^G \cdot \sin\gamma_K^G \end{bmatrix}_O^E \tag{6.57}$$

using the transformation matrix

$$M_{KO} = \begin{pmatrix} \cos\chi_K^G \cdot \cos\gamma_K^G & \sin\chi_K^G \cdot \cos\gamma_K^G & -\sin\gamma_K^G \\ -\sin\chi_K^G & \cos\chi_K^G & 0 \\ \cos\chi_K^G \cdot \sin\gamma_K^G & \sin\chi_K^G \cdot \sin\gamma_K^G & \cos\gamma_K^G \end{pmatrix}. \tag{6.58}$$

Subsystem WGS84 Position Propagation

The position propagation in the WGS84 frame is dependent on the current position and the object's speed relative to the ECEF frame. Figure 6.8 show the frame with the current position. The position G represents the point mass position longitude λ^G, latitude ϕ^G, and altitude h^G. The time derivative of the WGS84 coordinates can be derived by geometrical understanding. Since the z-axis of the NED frame points down to the ellipsoid surface perpendicularly, the time derivative of the altitude

$$\dot{h}^G = -w_K^G \tag{6.59}$$

is given by the negative kinematic speed in z-direction w.r.t. the ECEF frame in NED coordinates.

Both the time derivatives of the latitude $\dot{\phi}^G$ and longitude $\dot{\lambda}^G$ can be derived by simple circular motion

$$v = \omega \cdot r \tag{6.60}$$

using the relationship between the angular rotation speed ω, the radius of a circle r, and the speed at the circle's radius v. In north direction, the aircraft travels along a circle with radius $M_\phi + h^G$ (see Figure 6.8). Using (6.60), the time derivative of the latitude

$$\dot{\phi}^G = \frac{u_K^G}{M_\phi + h^G}. \tag{6.61}$$

is dependent on the kinematic speed in the north direction u_K^G, the current altitude h^G, and M_ϕ (dependent on the latitude ϕ^G). Similarly, the time derivative of the longitude

$$\dot{\lambda}^G = \frac{v_K^G}{(N_\phi + h^G) \cdot \cos\phi} \tag{6.62}$$

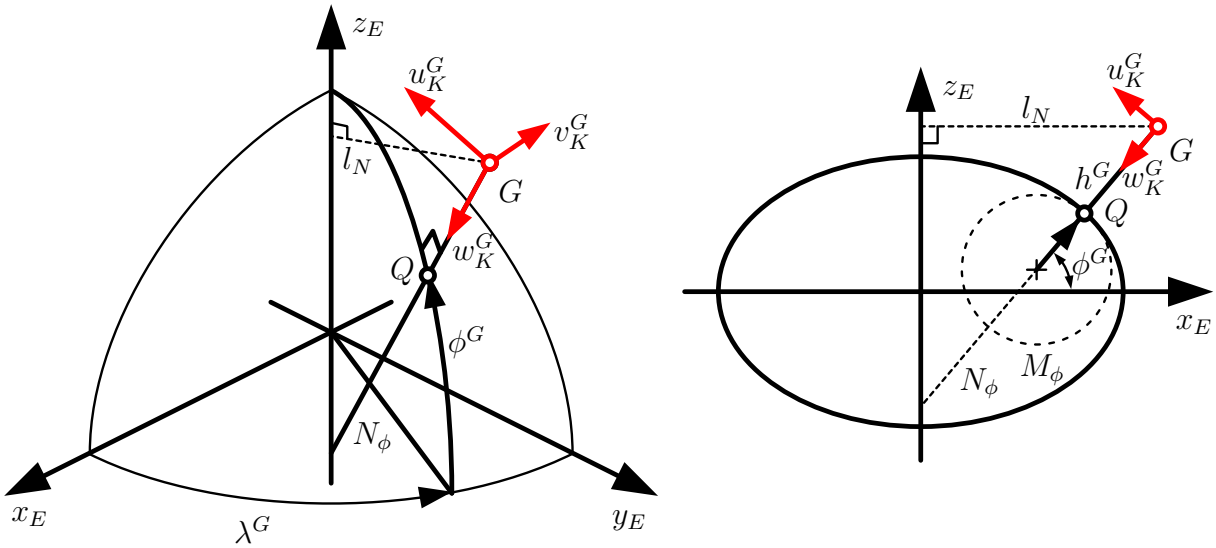


Figure 6.8: Position propagation in WGS84 coordinates.

is defined by a circular motion parallel to the equatorial plane. The radius

$$l_N = \left(N_\phi + h^G\right) \cdot \cos\phi \tag{6.63}$$

of the circle is shown in Figure 6.8. Thus, the subsystem position propagation in WGS84 coordinates is given by

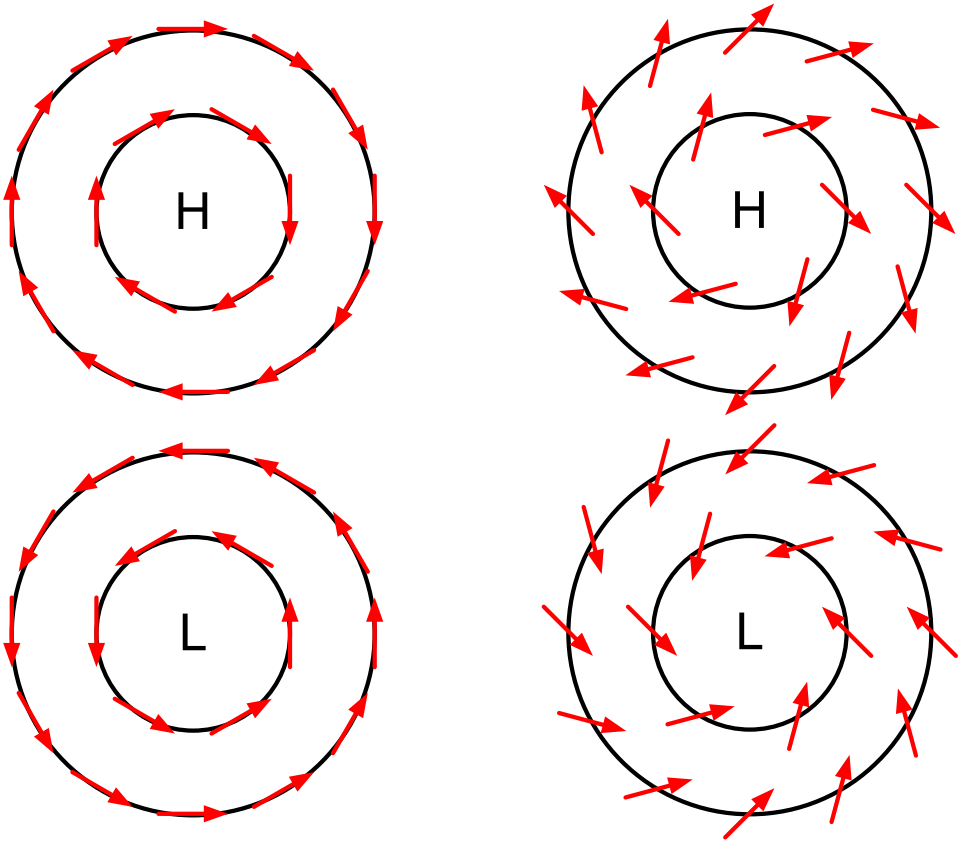
$$\begin{pmatrix} \dot{\lambda}^G \\ \dot{\phi}^G \\ \dot{h}^G \end{pmatrix} = \begin{pmatrix} \frac{v_K^G}{\left(N_\phi + h^G\right) \cdot \cos\phi^G} \\ \frac{u_K^G}{M_\phi + h^G} \\ -w_K^G \end{pmatrix}. \tag{6.64}$$

Subsystem Wind

In order to calculate the aerodynamic speed and angles, the wind acting on the aircraft is required. The wind is influence by three main forces. The difference between areas of high and low air pressure, the Coriolis force, and the friction forces close to ground [125]. The wind speed at the ground is zero (due to friction) and increases with altitude. In the following, the atmospheric boundary layer is discussed.

In high altitudes (e.g. higher than $1000 \ldots 2000m$ above ground level), only the pressure and Coriolis forces are relevant. The resulting force yields a wind parallel to the isobars of the pressure field (see Figure 6.9a) [125]. The flow orientation is dependent on the north or south side of the globe. The wind is called geostrophic and its speed $V_{W,g}$ is approximately constant w.r.t. altitude.

With decreasing altitude, the friction increases acting opposite the wind flow direction. The wind speed is decreased until it reaches zero at the ground altitude. Since the Coriolis force is dependent of the speed, its impact becomes smaller as well. Thus, the wind vector rotates in the direction of the pressure force (see Figure 6.9b). The angle between the wind directions close to the ground and in the geostrophic layer is given by $\alpha_{W,0}$ which is dependent on the ground type. The deflection over water is relatively



(a) In high altitudes, the geostrophic wind flows parallel to the isobars.

(b) Due to influence of ground friction at lower altitudes the wind turns in the direction of the pressure gradient.

Figure 6.9: Wind directions between areas of high (H) and low (L) pressure on northern hemisphere in high an low altitudes. Black lines indicate the areas of equal pressure (isobars) and red arrows depict the wind direction.

small (approx $10°$) and may reach up to $45°$ in mountainous areas [126]. Airport runways, which usually can be represented by a grassy landscape have a deflection of approximately $30°$.

Close to the ground, the speed is much smaller and thus the Coriolis force can be neglected [125]. The wind speed still decreases with decreasing altitude but the direction no longer changes. This so-called Prandtl layer is approximately $20 \ldots 100m$ thick h_P. It is described by the logarithmic speed profile

$$V_{W,Pr} = \frac{u^*}{\kappa_K} \cdot \ln\left(\frac{(h^G - h_{GL})}{h_0}\right) \tag{6.65}$$

where u^* represents the shear velocity, $\kappa_K \approx 0.41$ the Von Kármán constant [125], h_{GL} the ground altitude, and h_0 the surface roughness ($h_0 \approx 0.01$ for short grass).

Above the Prandtl layer the already discussed rotation of the wind occurs. This layer is also called the Ekman layer (or Ekman spiral). It is named after the oceanographer Ekman who first mathematically described the flow direction change of water

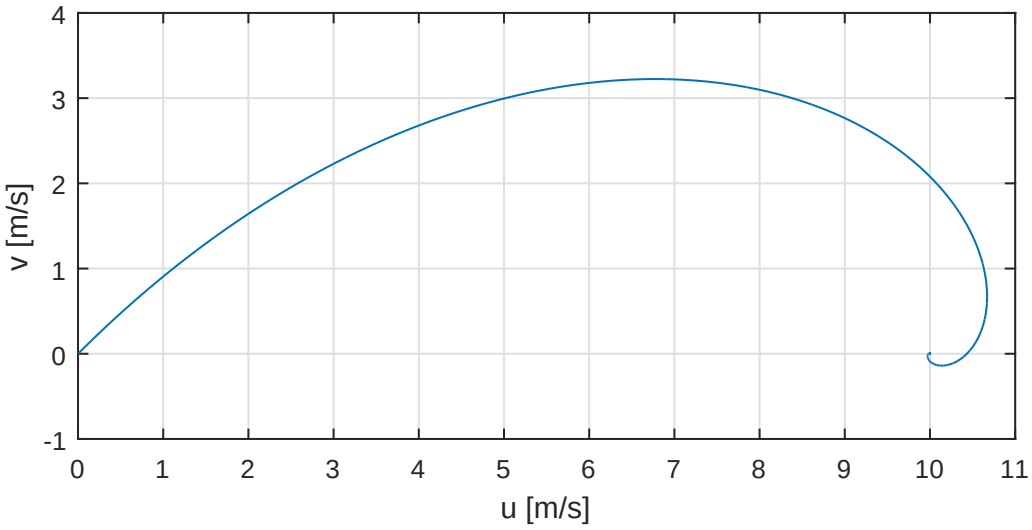


Figure 6.10: Ekman spiral for a geostrophic wind of $10m/s$. The wind deflection in the boundary layer is $45°$.

currents in the boundary layer. The horizontal speed components

$$u(h^G) = V_{W,g} \cdot \left[1 - \exp\left(-\frac{h^G}{D}\right) \cdot \cos\frac{h^G}{D}\right] \tag{6.66}$$

$$v(h^G) = V_{W,g} \cdot \exp\left(-\frac{h^G}{D}\right) \cdot \sin\frac{h^G}{D} \tag{6.67}$$

can be derived from the Navier Stokes equation (see [125]) where

$$D = \sqrt{2 \cdot \frac{K_m}{f}} \tag{6.68}$$

is the Ekman length. K_m represents the diffuse coefficient and f the Coriolis parameter. For both

$$K_m = 5m^2s^{-1}, \qquad f = 1 \cdot 10^{-4}s^{-1} \tag{6.69}$$

usual values are given [125]. Figure 6.10 shows the Ekman spiral for a geostrophic wind speed $V_{W,g} = 10m/s$ by plotting the wind components with altitude. At the ground altitude, the wind speed is zero and converges to the geostrophic wind speed for infinite altitude. The first crossing of the spiral with the $\bar{u}$ axis approximates the height of the Ekman layer

$$h_E = \pi D \tag{6.70}$$

and can be derived by setting (6.67) to zero. The deflection of the wind direction in the Figure is $45°$ and is constant for all Ekman spirals. However, smaller deflection angles occur in reality.

In order to model smaller wind angle defections $\alpha_{W,0}$ [125] combines both the Prandtl and the Ekman layers. In the Prandtl layer ($0 \leq h^G - h_{GL} \leq h_P$) the wind components

are given by

$$u(h^G) = \frac{u^*}{\kappa_K} \cdot \ln\left(\frac{(h^G - h_{GL})}{h_0}\right) \cdot \cos\alpha_{W,0}, \tag{6.71}$$

$$v(h^G) = \frac{u^*}{\kappa_K} \cdot \ln\left(\frac{(h^G - h_{GL})}{h_0}\right) \cdot \sin\alpha_{W,0}, \tag{6.72}$$

and in the Ekman layer ($h^G - h_{GL} \geq h_P$) by

$$u(h^G) = V_{W,g} \cdot \left[1 - \sqrt{2} \cdot \exp\left(-\frac{H_W}{D}\right) \cdot \sin\alpha_{W,0} \cdot \cos\left(\frac{H_W}{D} + \frac{\pi}{4} - \alpha_{W,0}\right)\right], \tag{6.73}$$

$$v(h^G) = V_{W,g} \cdot \sqrt{2} \cdot \exp\left(-\frac{H_W}{D}\right) \cdot \sin\alpha_{W,0} \cdot \sin\left(\frac{H_W}{D} + \frac{\pi}{4} - \alpha_{W,0}\right), \tag{6.74}$$

$$H_W = h^G - h_{GL} - h_P. \tag{6.75}$$

The equations must be adapted to account for the NED frame. Additionally, apart from the wind deflection in the Ekman layer, the general wind direction χ_W can be set. Since the wind direction is usually measured close to the ground (e.g. $10m$), χ_W states the wind direction in the Prandtl layer. Therefore, the components above have to be rotated by the matrix

$$M_W = \begin{bmatrix} \cos\gamma_W & \sin\gamma_W & 0 \\ -\sin\gamma_W & \cos\gamma_W & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{6.76}$$

where

$$\gamma_W = \frac{\pi}{2} - \alpha_{W,0} - \chi_W \tag{6.77}$$

defines the rotation angle. Thus, the wind components in the NED frame are given by

$$\left(\vec{v}_W^G\right)_O^E = M_W \cdot \begin{bmatrix} v(h^G) \\ u(h^G) \\ 0 \end{bmatrix} \tag{6.78}$$

and are displayed in Figure 6.11 For the time derivative of the calibrated air speed, the time derivative of the wind speed $\left(\dot{\vec{v}}_W^G\right)_O^{EE}$ is required. Although the wind field is constant w.r.t. time, an altitude rate induces a perceived time derivative in the wind. It is derived by differentiating (6.78) w.r.t. h^G:

$$\left(\dot{\vec{v}}_W^G\right)_O^{EE} = \frac{\partial\left(\dot{\vec{v}}_W^G\right)_O^{EE}}{\partial h^G} \cdot \dot{h}^G. \tag{6.79}$$

Subsystem Aerodynamic Speed

In the NED frame, the kinematic speed and the wind vector are subtracted using the superposition principle. The aerodynamic speed

$$\left(\vec{v}_A^G\right)_O^E = \begin{bmatrix} u_A^G \\ v_A^G \\ w_A^G \end{bmatrix}_O^E = \begin{bmatrix} u_K^G \\ v_K^G \\ w_K^G \end{bmatrix}_O^E - \begin{bmatrix} u_W^G \\ v_W^G \\ w_W^G \end{bmatrix}_O^E \tag{6.80}$$

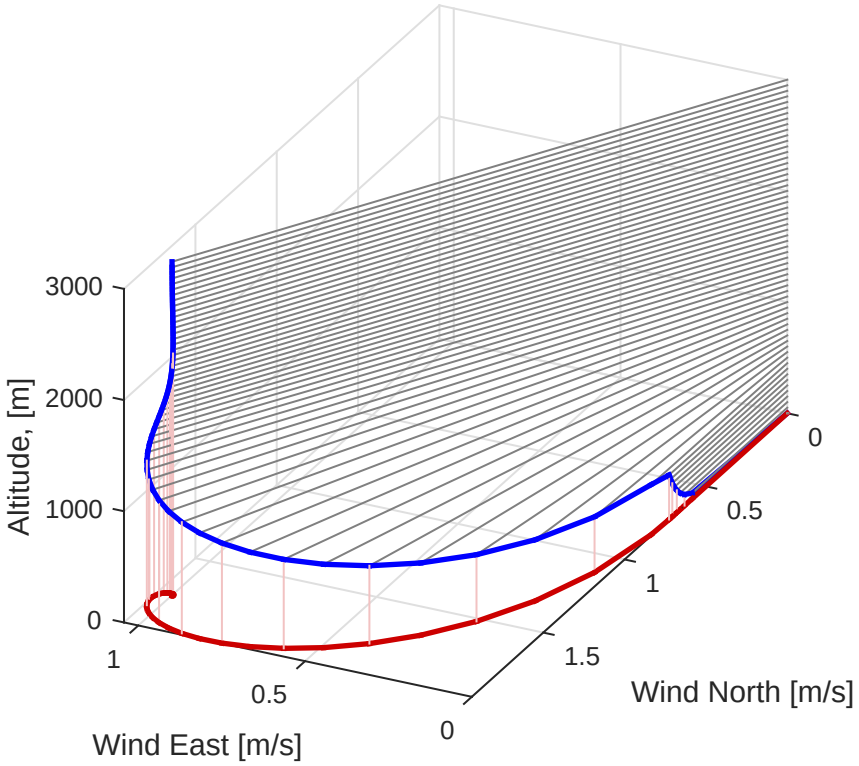


Figure 6.11: Combined Prandtl and Ekman layers for a geostrophic wind of $2m/s$. The wind deflection in the boundary layer is 30°.

is calculated in the NED frame. From the components, the absolute aerodynamic speed

$$V_A = \sqrt{\left(u_A^G\right)^2 + \left(v_A^G\right)^2 + \left(w_A^G\right)^2}, \tag{6.81}$$

the aerodynamic course angle

$$\chi_A = \arctan\left(\frac{v_A^G}{u_A^G}\right), \tag{6.82}$$

and the aerodynamic climb angle

$$\gamma_A = -\arctan\left(\frac{w_A^G}{\sqrt{\left(u_A^G\right)^2 + \left(v_A^G\right)^2}}\right) \tag{6.83}$$

are calculated.

Subsystem Aerodynamic Transformation

The calculated aerodynamic angles (χ_A, γ_A) together with the aerodynamic bank angle μ_A control input are used to create the transformation matrix M_{AO} introduced in (6.18). This matrix is moved into a separate subsystem to ensure that it can be differentiated by the Symbolic Math Toolbox.

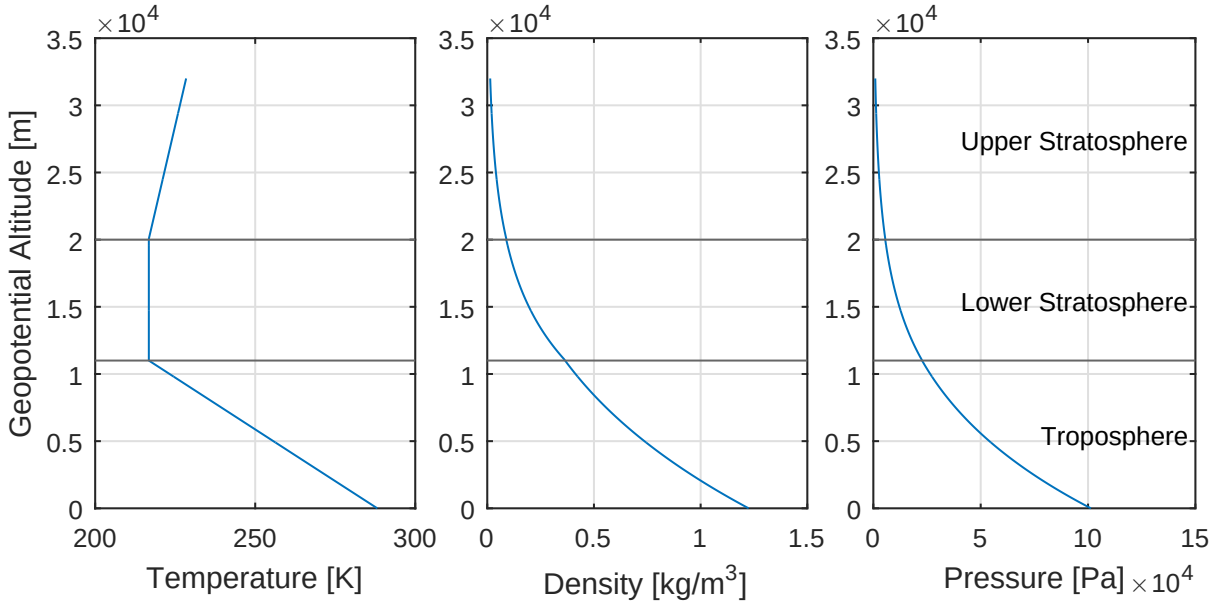


Figure 6.12: International Standard Atmosphere (ISA) defined by ICAO.

Subsystem Atmosphere

The International Standard Atmosphere (ISA) was introduced by the International Civil Aviation Organisation (ICAO) as an idealized atmospheric description model [127]. The ISA atmosphere is defined up to the altitude of $80km$. Figure 6.12 shows the air density and temperature for the troposphere, lower stratosphere and upper stratosphere. Since typical civil aircraft reach around $10km$ altitude in cruise and this thesis covers approach trajectories, the dynamic model implements the troposphere only.

In order to simulate a gravitational decrease with increasing altitude, the ISA uses the geopotential altitude

$$H_G = \frac{h^G \cdot r_E}{h^G + r_E} \tag{6.84}$$

that is calculated from the geometric altitude h^G and the earth radius r_E. The temperature ratio

$$\eta_T = \frac{T}{T_s} = 1 - \frac{n-1}{n} \cdot \frac{g}{R \cdot T_s} \cdot H_G \tag{6.85}$$

states the quotient of the temperature T at a certain altitude and the temperature at sea level T_s. A description of symbols and their values is found in Table 6.3. Similar to this, the ratios for air density

$$\eta_\rho = [\eta_T]^{\frac{1}{n-1}} \tag{6.86}$$

and air pressure

$$\eta_p = [\eta_T]^{\frac{n}{n-1}} \tag{6.87}$$

are calculated. The temperature T, the density ρ, and the pressure p at the altitude h^G are calculated

$$T = \eta_T \cdot T_s, \qquad \rho = \eta_\rho \cdot \rho_s, \qquad p = \eta_p \cdot p_s \tag{6.88}$$

Table 6.3: Symbols International Standard Atmosphere (ISA) defined by ICAO

Description	Symbol	Value	Unit of measurement
Gravitational Acceleration	g	9.80665	m/s^2
Universal Gas Constant	R	287.05287	$J/(K \cdot kg)$
Polytropic Exponent	n	1.235	$-$
Air Pressure at Sea Level	p_s	101.325×10^3	Pa
Air Temperatur at Sea Level	T_s	288.15	K
Air Density at Sea Level	ρ_s	1.225	kg/m^3
Earth Radius	r_E	6371	km

using their ratios and values at sea level. The sea level values for the temperature T_s, the density ρ_s, and the pressure p_s are defined for a reference case. However, with changing weather conditions, the atmosphere model might be unrealistic. Therefore, it is possible to define offsets for the sea level temperature ΔT and pressure Δp. Thus, the sea level values are redefined

$$T_s := T_s + \Delta T \tag{6.89}$$

$$p_s := p_s + \Delta p \tag{6.90}$$

and the value for the air density at sea level is determined using the ideal gas equation

$$\rho_s = \frac{p_s}{R \cdot T_s}. \tag{6.91}$$

Additionally, the atmosphere subsystem returns the speed of sound a and the Mach number M

$$a = \sqrt{\kappa \cdot R \cdot T}, \qquad M = \frac{V_A}{a} \tag{6.92}$$

where $\kappa = 1.4$ is the adiabatic exponent.

Subsystem Propulsion Force

In this thesis, a point mass model is considered. Therefore, it is assumed that the thrust force acts along the x-axis of the kinematic frame. This assumption is fair to make, since for commercial aircraft the angles between the thrust vector and the kinematic velocity vector are very small. Thus, the propulsion subsystem calculates the thrust vector

$$\left(\vec{F}_P^G\right)_A = \begin{bmatrix} T \\ 0 \\ 0 \end{bmatrix}_A \tag{6.93}$$

subject to the thrust formulation in the BADA 4 model (6.51).

Subsystem Propulsion in Kinematic System

The propulsion force cannot be transformed into the K frame directly. Therefore, as an intermediate step, the NED frame

$$\left(\vec{F}_P^G\right)_K = M_{KO} \cdot {M_{AO}}^T \cdot \left(\vec{F}_P^G\right)_A \tag{6.94}$$

is used. Both transformation matrices have already been differentiated in previous subsystems, this multiplication can be handled by the symbolic math toolbox.

Subsystem Thrust Dynamic

In the BADA 4 model no dynamics for the aircraft thrust are considered. In order to simulate a latency in the response, a PT1 element is used

$$T_\delta \cdot \dot{\delta}_T + \delta_T = K_\delta \cdot \delta_{T,CMD} \tag{6.95}$$

where $K_\delta = 1$ is a gain and T_δ a time constant. It is stated in [128] that the time from idle to 95% percent rated takeoff power shall be no more than 5 seconds. Using the analytic solution of the step response

$$\delta_T(t) = 1 - e^{-\frac{t}{T_\delta}} \tag{6.96}$$

the time constant for $\delta_T(5s) = 0.95$ is determined to be

$$T_\delta = -\frac{5}{\ln 0.05} s \approx 1.67s. \tag{6.97}$$

Subsystem Gravitation Force

As mentioned above, the gravitational force in the NED frame

$$\left(\vec{F}_G^G\right)_O = \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix}_O \tag{6.98}$$

is transformed into the K frame

$$\left(\vec{F}_G^G\right)_K = M_{KO} \cdot \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix}_O = \begin{bmatrix} -mg \cdot \sin\gamma_K^G \\ 0 \\ mg \cdot \cos\gamma_K^G \end{bmatrix}_K \tag{6.99}$$

with the already familiar axis transformation M_{KO} (6.58).

Subsystem Aerodynamic Force in Aerodynamic System

In section 6.1.3 the lift and drag are introduced. Thus the aerodynamic force

$$\left(\vec{F}_A^G\right)_A = \begin{bmatrix} -D \\ 0 \\ -L \end{bmatrix}_A \tag{6.100}$$

is formulated. The drag calculation is subject to the BADA 4.0 Family model that defines a detailed formulation. Since the analytic derivative needs to be calculated w.r.t. the subsystem's input variables, the transformation to the kinematic frame is moved into a separate subsystem.

Subsystem Aerodynamic Force in Kinematic System

The aerodynamic force cannot be transformed into the K frame directly. Therefore, as an intermediate step, the NED frame

$$\left(\vec{F}_A^G\right)_K = M_{KO} \cdot {M_{AO}}^T \cdot \left(\vec{F}_A^G\right)_A \tag{6.101}$$

is used. Both transformation matrices have already been differentiated in previous subsystems, this multiplication can be handled by the symbolic math toolbox.

Subsystem Total Force in Kinematic System

As shown in (6.43), the total force acting on the point mass w.r.t. the K frame is required. Since all forces are transformed into the kinematic frame, the total force

$$\left(\vec{F}_T^G\right)_K = \left(\vec{F}_A^G\right)_K + \left(\vec{F}_P^G\right)_K + \left(\vec{F}_G^G\right)_K \tag{6.102}$$

is a simple superposition.

Subsystem Fuel Flow

During flight the aircraft's mass is gradually decreased due to fuel burn. There are many different ways to approximate the fuel consumption. In this thesis, the fuel flow model from the BADA 4.0 Family is used [19]. The change of mass is equivalent to the negative fuel flow

$$\dot{m} = -f_{flow} = -\frac{W_{MTOW} \cdot a_0}{L_{HV}} \cdot \eta_p \sqrt{\eta_T} \cdot C_F \tag{6.103}$$

where W_{MTOW} specifies the maximum takeoff weight and a_0 the speed of sound at sea level. The fuel flow coefficient

$$C_F = f(M, C_T) \tag{6.104}$$

is subject to the thrust coefficient C_T from the propulsion calculation and the current Mach number M. The equation of C_F in the BADA 4.0 Family is not subject to public domain.

Subsystem Load Factor

Load factor limits have to be met in order to meet structural limitations and passenger comfort. Therefore, the vertical load factor

$$n_z = \frac{L}{mg} \tag{6.105}$$

is calculated. It is defined as lift L divided by weight mg.

Subsystem Translation Round Earth

The translation equation of motions for a point mass model on a round earth was introduced in (6.43). Now that all forces are calculated, the state derivatives of the kinematic speed

$$\dot{V}_K^G = \frac{\left(X_T^G\right)_K}{m} - t_{rx}, \tag{6.106}$$

the kinematic course angle

$$\dot{\chi}_K^G = \frac{\left(Y_T^G\right)_K}{m \cdot \cos\gamma_K^G \cdot V_K^G} - \frac{t_{ry}}{\cos\gamma_K^G \cdot V_K^G}, \tag{6.107}$$

and the kinematic climb angle

$$\dot{\gamma}_K^G = -\frac{\left(Z_T^G\right)_K}{m \cdot V_K^G} + \frac{t_{rz}}{V_K^G} \tag{6.108}$$

are calculated in this subsystem. In the equations above,

$$\begin{bmatrix} t_{rx} \\ t_{ry} \\ t_{rz} \end{bmatrix}_K = M_{KO} \cdot \left[\left(\vec{\omega}^{EO}\right)_O \times \left(\vec{v}_K^G\right)_O^E\right] \tag{6.109}$$

represent the components of the transport rate correction in the kinematic frame.

Subsystem Kinematic Speed Derivative

The time derivative of the kinematic speed is required for a subsystem below that calculates the calibrated air speed. The differentiation of the kinematic speed

$$\frac{d}{dt}\left(\vec{v}_K^G\right)_O^E = \left(\dot{\vec{v}}_K^G\right)_O^{EE} = \left(\dot{\vec{v}}_K^G\right)_O^{EO} + \left(\vec{\omega}^{EO}\right)_O \times \left(\vec{v}_K^G\right)_O^E \tag{6.110}$$

is done in the NED-frame using the Euler differentiation. The direct derivative of the kinematic speed w.r.t. the NED-frame

$$\left(\dot{\vec{v}}_K^G\right)_O^{EO} = \begin{bmatrix} \dot{u}_K^G \\ \dot{v}_K^G \\ \dot{w}_K^G \end{bmatrix}_O^{EO} \tag{6.111}$$

is obtained

$$\begin{aligned} \dot{u}_K^G = {} & \dot{V}_K^G \cdot \cos\chi_K^G \cdot \cos\gamma_K^G \\ & - V_K^G \cdot \sin\chi_K^G \cdot \cos\gamma_K^G \cdot \dot{\chi}_K^G \\ & - V_K^G \cdot \cos\chi_K^G \cdot \sin\gamma_K^G \cdot \dot{\gamma}_K^G \end{aligned} \tag{6.112}$$

$$\begin{aligned} \dot{v}_K^G = {} & \dot{V}_K^G \cdot \sin\chi_K^G \cdot \cos\gamma_K^G \\ & + V_K^G \cdot \cos\chi_K^G \cdot \cos\gamma_K^G \cdot \dot{\chi}_K^G \\ & - V_K^G \cdot \sin\chi_K^G \cdot \sin\gamma_K^G \cdot \dot{\gamma}_K^G \end{aligned} \tag{6.113}$$

$$\begin{aligned} \dot{w}_K^G = {} & -\dot{V}_K^G \cdot \sin\gamma_K^G \\ & - V_K^G \cdot \cos\gamma_K^G \cdot \dot{\gamma}_K^G \end{aligned} \tag{6.114}$$

by differentiating (6.57).

Subsystem Aerodynamic Speed Derivative

Additionally to the kinematic speed derivative, the time derivative of the calibrated air speed requires the time derivative of the aerodynamic speed as well. To calculate it, the change of wind needs to be taken into account. This derivative is zero only for a constant wind field (location and time). A static but position dependent wind field introduces a time dependency due to the aircraft motion. Both the current wind speed vector and its time derivative are calculated by the wind subsystem above.

The aerodynamic speed of the aircraft in the NED frame is given by

$$\left(\vec{v}_A^G\right)_O^E = \left(\vec{v}_K^G\right)_O^E - \left(\vec{v}_W^G\right)_O^E. \tag{6.115}$$

Thus, the time derivative of the aerodynamic speed vector

$$\begin{bmatrix} \dot{u}_A^G \\ \dot{v}_A^G \\ \dot{w}_A^G \end{bmatrix}_O^{EE} = \left(\dot{\vec{v}}_A^G\right)_O^{EE} = \left(\dot{\vec{v}}_K^G\right)_O^{EE} - \left(\dot{\vec{v}}_W^G\right)_O^{EE} \tag{6.116}$$

can be calculated. The absolute aerodynamic time derivative is calculated using

$$\dot{V}_A = \frac{u_A^G \cdot \dot{u}_A^G + v_A^G \cdot \dot{v}_A^G + w_A^G \cdot \dot{w}_A^G}{V_A}. \tag{6.117}$$

Subsystem Calibrated Air Speed

The calibrated air speed

$$V_{CAS} = V_A \cdot \sqrt{\eta_\rho} \tag{6.118}$$

is calculated using the square root of the density ratio η_ρ. For the optimization, the time derivative of the calibrated air speed is required. Therefore, (6.118) needs to be differentiated

$$\left(\frac{d}{dt}\right) V_{CAS} = \dot{V}_A \cdot \sqrt{\frac{\rho}{\rho_s}} + V_A \cdot \left(\frac{d}{dt}\right) \sqrt{\frac{\rho}{\rho_s}}. \tag{6.119}$$

The time derivative of the aerodynamic speed is taken from the previous subsystem. For the time derivative of the density ratio, the ISA atmosphere equations need to be differentiated. The resulting derivative simplifies to

$$\frac{\dot{\rho}}{\rho_s} = -\left(1 - (n-1) \cdot \frac{r_E \cdot h^G}{r_E + h^G}\right)^{\frac{2-n}{n-1}} \cdot \left(\frac{g}{n \cdot R \cdot T_s}\right)^{\frac{1}{n-1}} \cdot \frac{r_E}{\left(r_E + h^G\right)^2} \cdot \dot{h}^G \tag{6.120}$$

where the used symbols are described in the ISA atmosphere subsystem description.

6.2 Aircraft Constraints

The constraints imposed in the optimal control problem determine the quality of the obtained solution. In this section, the constraints used in the approach optimization are explained. These may be flight envelope constraints of the aircraft or operational constraints ensuring save air traffic. Path constraints may be purely continuous or dependent on the choice of a discrete control. Additionally, event constraints (modeled as point constraints) have to be considered. The constraints considered are compliant with the ICAO PANS-OPS rules.

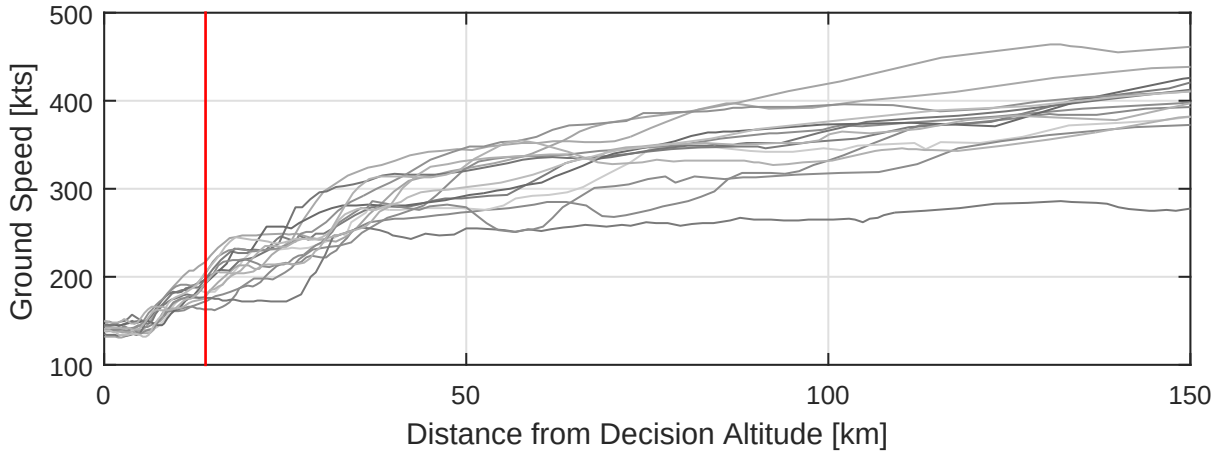


Figure 6.13: Aircraft ground speed in knots w.r.t. track distance from runway decision altitude.

6.2.1 Continuous Path Constraints

The continuous path constraints are not dependent on the flap / landing gear selection and are imposed at every time step.

Approach Deceleration Limit

During approach, the aircraft must reduce the speed continuously until the landing speed, also known as final approach speed, is reached. Usually, the deceleration of the calibrated air speed in knots

$$\dot{V}_{CAS,nm} = \frac{\dot{V}_{CAS} \cdot 3600}{V_{CAS}} \tag{6.121}$$

is given w.r.t. the distance in nautical miles. As the flap positions mainly influence the aircraft speed, the deceleration bounds have a great impact on the optimal solution as well as the optimal switching structure. Therefore, proper bounds need to be determined. [129] states a deceleration of $15kts/nm$. The airbus flight operation briefing notes [130] suggest $10 - 15kts/nm$ in level flight (with approach flaps) and $10 - 20kts/nm$ on the glide slope with flaps to landing and gear down.

Overall, the approach deceleration limit is not strictly defined and mostly subject to the air traffic control. Therefore, radar data from FlightRadar 24 is used to obtain typical approach speeds at Munich airport. Flights with a maximum flight distance of $800km$ are selected. This shall ensure that short-range aircraft are considered only. Otherwise, the flight selection is arbitrary.

Figure 6.13 shows the aircraft ground speed in knots during runway approach. The vertical red line indicates the approximate position of the glide slope intercept. In Figure 6.14, the deceleration in knots per nautical mile w.r.t. the track distance from the runway is shown.The strongest deceleration is achieved during the final approach to the runway.

Table 6.4 shows the deceleration limits used in the approach optimization of this chapter. In the approach section where a clean configuration is used, the aircraft may

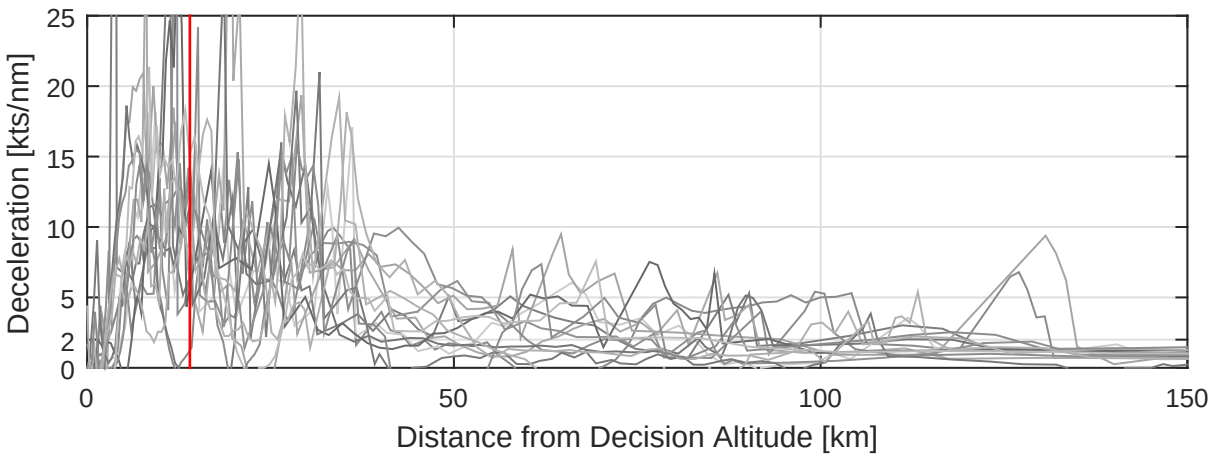


Figure 6.14: Aircraft ground speed deceleration in knots per nautical mile w.r.t. track distance from runway decision altitude. The speed data from Figure 6.13 is smoothed before the differentiation is performed.

Table 6.4: Approach deceleration limits.

Approach Section	**Lower Limit [kts/nm]**	**Upper Limit [kts/nm]**
Clean Approach	-10	0
Glideslope	-15	-2
Fully Stabilized	-3	0

maintain the speed. On the glide slope a minimum deceleration of $2kts/nm$ is enforced. In the final approach, after the aircraft is fully stabilized, the aircraft speed shall be constant. However, in order the have an overlap to the previous limit, a small deceleration is still allowed.

Maximum Rate of Descent

The closer an aircraft comes to the ground, the smaller the rate of descent must be. The rate of descent is limited step-wise w.r.t the Above Ground Level (AGL) (see Table 6.5). However, this limitation is never reached in the optimizations. Additionally, the aircraft must not climb during approach:

$$\dot{h}^G \leq 0. \tag{6.122}$$

6.2.2 Discrete Path Constraints

The aircraft speed limits are mainly driven by the flap and landing gear settings of an aircraft. In the optimization, these constraints are considered by vanishing constraints. For simplicity, the speed constraints are not stated w.r.t. a discrete control. All constraints presented here are reformulated as lower than or equal to zero constraints in order to comply with the vanishing constraints.

Table 6.5: Maximum rate of descent limit during approach.

Condition	**Max. Rate of Descent [ft/min]**
$> 5000ft$	5000
$> 4000ft$	4000
$> 3000ft$	3000
$> 2000ft$	2000
$> 1000ft$	1500
$< 1000ft$	1000

Maximum Speed Limit

The maximum speed is dependent on the current flap setting and shall avoid structural damages to the high lift components. In the BADA Family 4 model, the calibrated air speed is limited:

$$V_{CAS} \leq V_{CAS,max}. \tag{6.123}$$

Minimum Speed Limit

Dependent on the flap and landing gear setting, the BADA Family 4 specifies maximum lift coefficients. Thus, the stall speed for a stationary horizontal flight

$$V_{A,stall} = \sqrt{\frac{2 \cdot m \cdot g}{\rho \cdot S_{ref} \cdot C_{L,max}}} \tag{6.124}$$

can be calculated. However, for save flight operations, a safety margin to the stall limit must be kept in order to motivate the minimal selectable speed V_{LS}. Usually, a $0.3g$ buffet margin has to be kept [131]. Thus, an aircraft stall is avoided in case vertical and lateral corrections during approach have to be made:

$$V_{LS} = \sqrt{\frac{2 \cdot m \cdot g \cdot n_{z,Margin}}{\rho \cdot S_{ref} \cdot C_{L,max}}}, \qquad n_{z,Margin} = 1.3. \tag{6.125}$$

Hence, the speed limit

$$V_A \geq V_{LS} \tag{6.126}$$

is considered in the optimization.

Load Factor Limit

The load factor limit

$$n_z = \frac{L}{m \cdot g} \tag{6.127}$$

is dependent on whether the high lift or the landing gear is extracted [132]. In the clean configuration, the limits are

$$-1 \leq n_z \leq 2.5 \tag{6.128}$$

whereas in all non-clean configurations, the limit becomes

$$0 \leq n_z \leq 2. \tag{6.129}$$

Due to a continuous glide in the approach optimization, this limit is never reached. For passenger comfort, discrete control independent limits

$$0.8 \leq n_z \leq 1.2 \tag{6.130}$$

are used.

6.2.3 Operational Constraints

For save final approach and landing, the aircraft must meet many operational constraints. They influence the trajectory and thus ultimately the switching of the high lift devices.

Final Approach Speed / Landing Speed

The final approach speed V_{APP} is the safest landing speed of the aircraft [133]. It is dependent on multiple factors such as aircraft weight, wind conditions, high/lift configuration, aircraft failure status, icing conditions, and autothrust/autoland usage. Here, only the weight and wind conditions are considered.

Table 6.6: Calculation of lowest selectable speed for A320 family based on the aircraft mass [133].

Mass [1000 kg]	52	56	60	64	68	72	76	80	84	88	92	94
VLS FULL [kts]	116	121	125	129	133	137	141	144	148	151	155	157
VLS 3 [kts]	121	125	130	134	138	142	146	150	154	157	161	163

For the calculation of the final approach speed, first the lower selectable speed must be calculated. Table 6.6 shows how this speed is calculated [133]. Using the maximum lift coefficient of the A320 landing configuration, the table can be approximated using

$$V_{LS} = 1.205 \cdot V_{Stall,1g} = 1.205 \cdot \sqrt{\frac{2 \cdot m \cdot g}{\rho \cdot S_{ref} \cdot C_{L,max,LDG}}} \tag{6.131}$$

where 1.205 is an arbitrary fit parameter. The table data as well as the approximating formula are shown in Figure 6.15.

Apart from the minimum selectable speed, the headwind V_{HW} must be determined. As runway operations are always in the direction the wind originates from, tailwind conditions usually do not appear. Using both values, the final approach speed

$$V_{APP} = V_{LS} + \max\left\{5kts, \min\left\{15kts, \frac{1}{3}V_{HW}\right\}\right\} \tag{6.132}$$

is calculated [133]. This speed is used in the final boundary condition in the last phase.

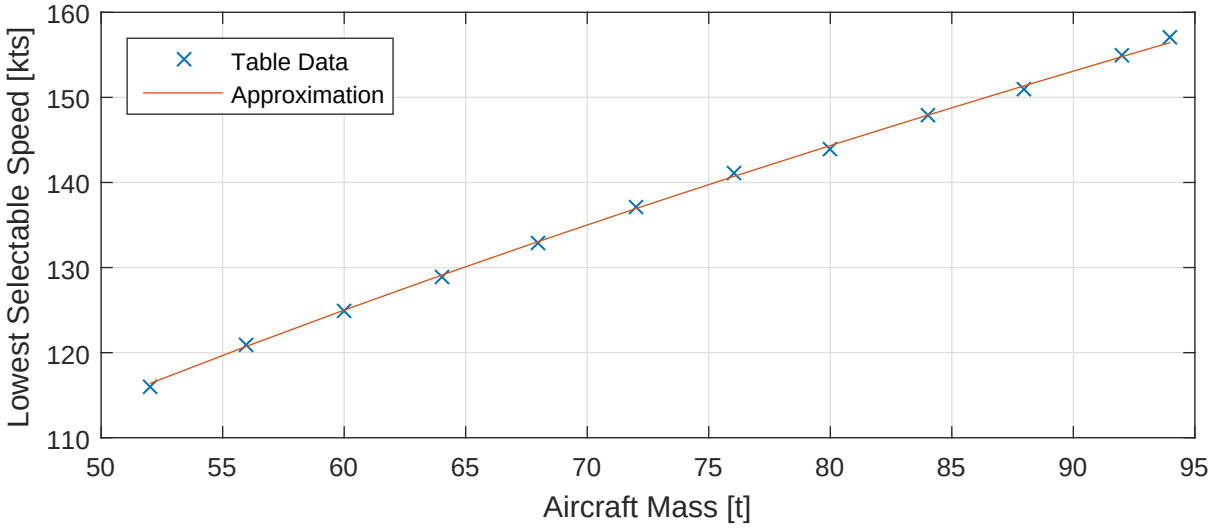


Figure 6.15: Comparison table data and approximation formula for lowest selectable speed.

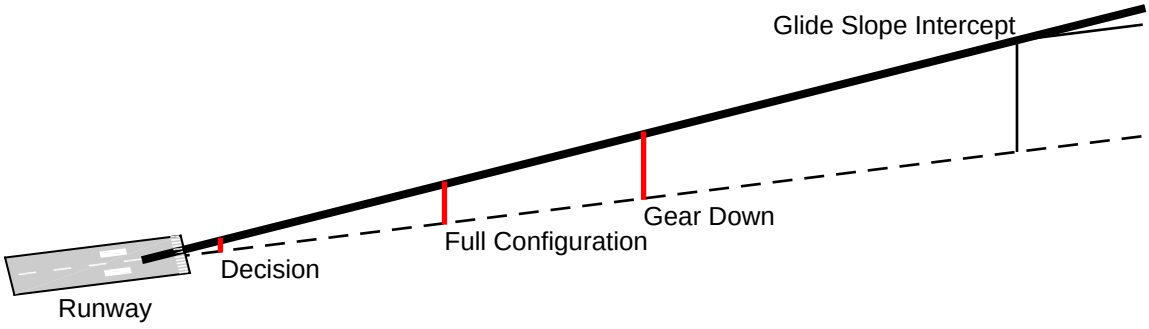


Figure 6.16: Point constraints on the glide slope (not to scale).

Glide Slope Point Constraints

The following constraints have to be met while on the glide slope. The minimum absolute intercept altitude must be $5000ft$. Additionally, the calibrated air speed must be below $160kts$ at the outer marker as otherwise deceleration to the final approach speed may not be possible [134]. The outer marker is $6nm$ from the runway threshold and correlates to $2000ft$ above ground level. At this altitude, the landing gear must be deployed as well. Additionally, the aircraft must be close to approach speed, fully stabilized and in landing configuration (flaps at landing) before the $1000ft$ above ground level is passed [135]. Therefore, in the optimization, the full configuration is enforced at $1200ft$ above ground. Additionally, at this altitude, the aircraft calibrated air speed must be below $V_{APP} + 1kts$.

Glide Slope Constraint

During final approach, the aircraft has to follow the $3°$ glide slope to the runway of Munich Airport. This constraint is imposed on the kinematic climbing angle. However, imposing an equality constraint on a state reduces the degree of freedom of the dynamics. Thus, the optimal control problem becomes hard to solve. Additionally, since the glide slope must be intercepted, the initial glide slope constraint must have wider bounds than close to the runway. Therefore, a box constraint for the kinematic climbing angle is defined. Dependent on the distance from the decision altitude, the lower and upper bounds approach the $3°$ glide slope. The same holds for the aircraft course angle.

In the optimization, the glide slope tolerance is continuously reduced. The bounds on the kinematic angles changes linearly from $\Delta\chi_K^G = \Delta\gamma_K^G = \pm 2°$ at the glide slope intercept down to $\Delta\chi_K^G = \Delta\gamma_K^G = \pm 0.15°$ at the decision altitude. Therefore, the glide slope tolerance

$$\Delta t = \frac{\Delta t_I - \Delta t_D}{d_I - d_D} \cdot (d - d_D) + \Delta t_D \tag{6.133}$$

is dependent on the great arc distance d from the runway threshold. The distance is calculated using the haversine formulation

$$d = 2 \cdot (r_E + h_{RW}) \cdot \arcsin \sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos\phi^G \cdot \cos\phi_{RW} \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right)} \tag{6.134}$$

$$\Delta\phi = \left|\phi^G - \phi_{RW}\right| \tag{6.135}$$

$$\Delta\lambda = \left|\lambda^G - \lambda_{RW}\right| \tag{6.136}$$

which is good conditioned for small distances [136]. In the equation ϕ_{RW}, λ_{RW} represent the latitude and longitude of the runway threshold. The airport altitude is given by h_{RW}.

6.3 Problem Definition

In this section, the approach optimal control problem as well as the solution strategy are presented.

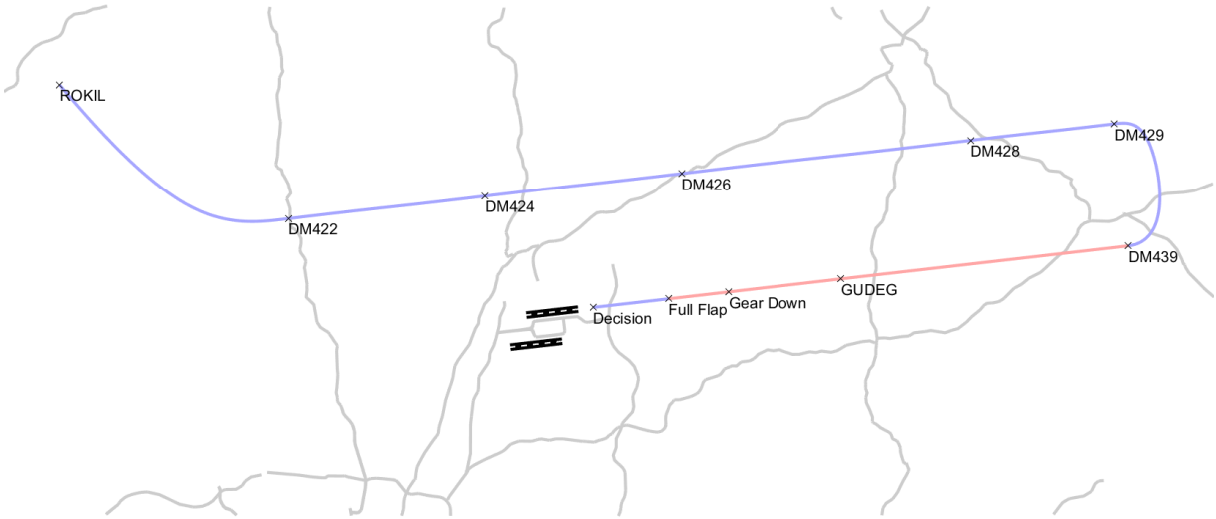


Figure 6.17: Approach optimal control problem mission layout.

6.3.1 Approach Optimal Control Problem

For the fuel minimal approach problem, runway $26R$ from Munich Airport is chosen. The approach starts at the initial approach fix ROKIL and ends on the glide slope at the decision altitude $200ft$ above the ground level of runway $26R$ [137]. In Figure 6.17, the approach mission layout is shown where the aircraft flies a downwind segment and turns right for alignment with the runway and final approach. The aircraft passes through various navigation aids (waypoints) for which the locations are shown [138].

In the optimal control problem, every segment between two navigation aids is represented by a phase. In order to reduce the number of optimization variables and thus the problem size, not all phases are optimized with discrete controls. In Figure 6.17, the blue line indicates the phases where the flaps can be chosen subject to optimization. In the red phases, the flap setting is fixed. Therefore, the initial approach until the alignment with the glideslope is flown in the clean configuration. Furthermore, below $1200ft$, the high lift and gear are fully extracted. Additionally, below $2000ft$, the discrete control selectable options are reduced to the last two high lift settings with the gear extracted.

6.3.2 Solution Strategy

Due to the fact that the aircraft model is significantly more complicated than the car model from chapter 5, the solution strategy is expanded. The two stage approach is still used. However, the generation of the initial guess for the optimization is extended. First, an initial guess for all states and constraints is calculated using an inverse dynamic BADA 4 model. This initial guess is used in a clean optimization where the maximum lift coefficient and other discrete constraints are not considered. From this initial solution, an intial guess for the discrete optimization is constructed. In the following, the solution strategy is explained in more detail:

1. A smooth trajectory through all waypoints is constructed using a B-Spline (see Figure 6.17). For every waypoint, the position in WGS84 coordinates, the flight directions, as well as a guessed speed are provided. The speeds are calculated by

linear interpolation between the initial approach speed (boundary condition) as well as the final approach speed (calculated using (6.132)). The geodetic coordinates are transformed into local NED where the runway threshold represents the origin. Using the B-Spline calculation presented in C.1, time derivatives of the position and kinematic speed are calculated. This information is used to reconstruct kinematic angles χ_K^G, γ_K^G, μ_K as well as the WGS84 position.

2. The kinematic data obtained in the previous step is used to calculate the control inputs to the dynamic model. This is achieved with inverse formulation of the BADA 4 model of section 6.1. Thus, the required lift coefficient C_L and the thrust lever position δ_T are obtained. The thrust lever command control is set to $\delta_{T,CMD} = \delta_T$ and the aerodynamic bank angle μ_A is set to the kinematic bank angle from the previous step.

3. A clean optimization is carried out for the approach. Boundary conditions regarding maximum lift coefficients and other discrete control dependent constraints are ignored. This step shall provide a solution that is better suited than the B-Spline interpolation.

4. The clean optimization solution is used to calculate an initial guess for the discrete OCP. The switching structure of the discrete control is determined dependent on the violation of the discrete constraints. Additionally, regulation constraints such as gear down and full landing configuration are considered.

5. Discrete OCP is solved without switching cost. The slack variables for the vanishing constraint relaxation have an upper bound of 0.01. The resolution of the time discretization is chosen to approximately 0.2 seconds. Additionally, an artificial high lift cost is added to the problem that penalizes higher flap and landing gear selections. The necessity is argued in the beginning of the section 6.5.

6. The solution is augmented by removing spikes shorter than 2 seconds from the solution (see section 3.6.3).

7. Finally, the discrete OCP is solved with switching cost. The slack variable for the switching cost approach has an upper bound of 0.9 (see section 3.4.3) and the penalty scaling is set to 5.

6.4 Single Optimization

In this section, a single approach optimal control problem is solved. It is used as reference for the parameter studies and to explain the structure of the obtained solution. Boundary conditions, environmental factors, and possible discrete control selections are found in Table 6.7. The problem consists of $76,787$ optimization variables, $92,585$ constraints, and has a sparsity of 99.98%. Initial guess generation as well as the clean optimization need just a few seconds. The actual optimization with discrete controls requires approximately 1231 seconds[1]. The first stage is calculated in 1147 seconds

[1]Intel Core i5-4670 CPU @ 3.40GHz, Windows 10 64bit, MATLAB 2015a

Table 6.7: Aircraft settings, initial conditions, and environmental conditions for approach mission.

Name	**Value**	**Unit**
Aircraft	A320.05 -232	
High Lift Position (Flaps)	$[0, 1, 2, 3, 4, 5]$	
Landing Gear Positions	$[0, 1]$	
Initial Speed (ROKIL)	250	$[kts]$
Initial Mass (ROKIL)	63	$[t]$
Initial Alitude (WGS84 at ROKIL)	4000	$[m]$
Wind Speed (at reference Altitude)	0	$[m/s]$
Wind Reference Altitude (above ground level)	10	$[m]$
Wind Direction (delta from aligned runway)	0	$[deg]$

whereas the second optimization stage with switching cost has a CPU time of 84 seconds. The minimum fuel was determined to be $386.2kg$ with additional 6.0 (pseudo-kilogram) as switching cost. All other cost accounted for the slack variables result in 9.9 (pseudo-kilogram).

In Figure 6.18, the obtained optimal solution of various data is shown w.r.t. real time. The switching structure of the flaps δ_{HL} as well as the landing gear δ_{LG} are plotted. Below, the altitude h^G history which contains the ground altitude as well as gear down and full configuration altitudes (black and gray horizontal lines) are shown. In the plot displaying the calibrated air speed V_{CAS}, the maximum speed is shown as a red line. The aerodynamic stall speed and the minimum selectable speed V_{LS} are converted to calibrated air speed. The lines are shown in red and green respectively. Furthermore, the lift coefficient C_L, the thrust lever position δ_T, the aerodynamic bank angle μ_A, and the vertical load factor n_z are displayed.

In the solution, the aircraft remains at the initial approach altitude until a continuous descent begins. This descent is carried out until the decision altitude. At first, the initial speed which requires a certain amount of thrust is kept. However, at some point the thrust is set to idle. This reduces the speed and increases the required lift coefficient until the descent begins. At this point, the calibrated air speed remains constant. Shortly after the turn (visible in the bank angle plot) and the alignment with the glide slope, the aircraft decelerates and the required lift coefficient is increased further. The first flap configuration is selected while the thrust lever remains in the idle position. Once the aircraft reaches $2000ft$ above ground level, the gear is deployed. This is immediately followed by a thrust increase. Dependent on the deceleration, the thrust requirement changes. At $1200ft$ above ground, the full configuration is enforced. Additionally, the aircraft must have almost reached the final approach speed V_{APP}. Therefore, the thrust requirement increases further.

The switching of the flaps to the first non-clean selection is driven by the lift requirement and thus the minimum selectable speed of the current high lift configuration. Both later switches are mainly driven by operational requirements to ensure a safe approach. The deployment of the flaps increases the drag on the aircraft. The operationally required switches in the flaps and landing gear are immediately followed by a thrust increase from idle to approximately 40%. This results in a higher fuel consumption. Therefore, regarding fuel efficiency, the deployment of the flaps is not desired.

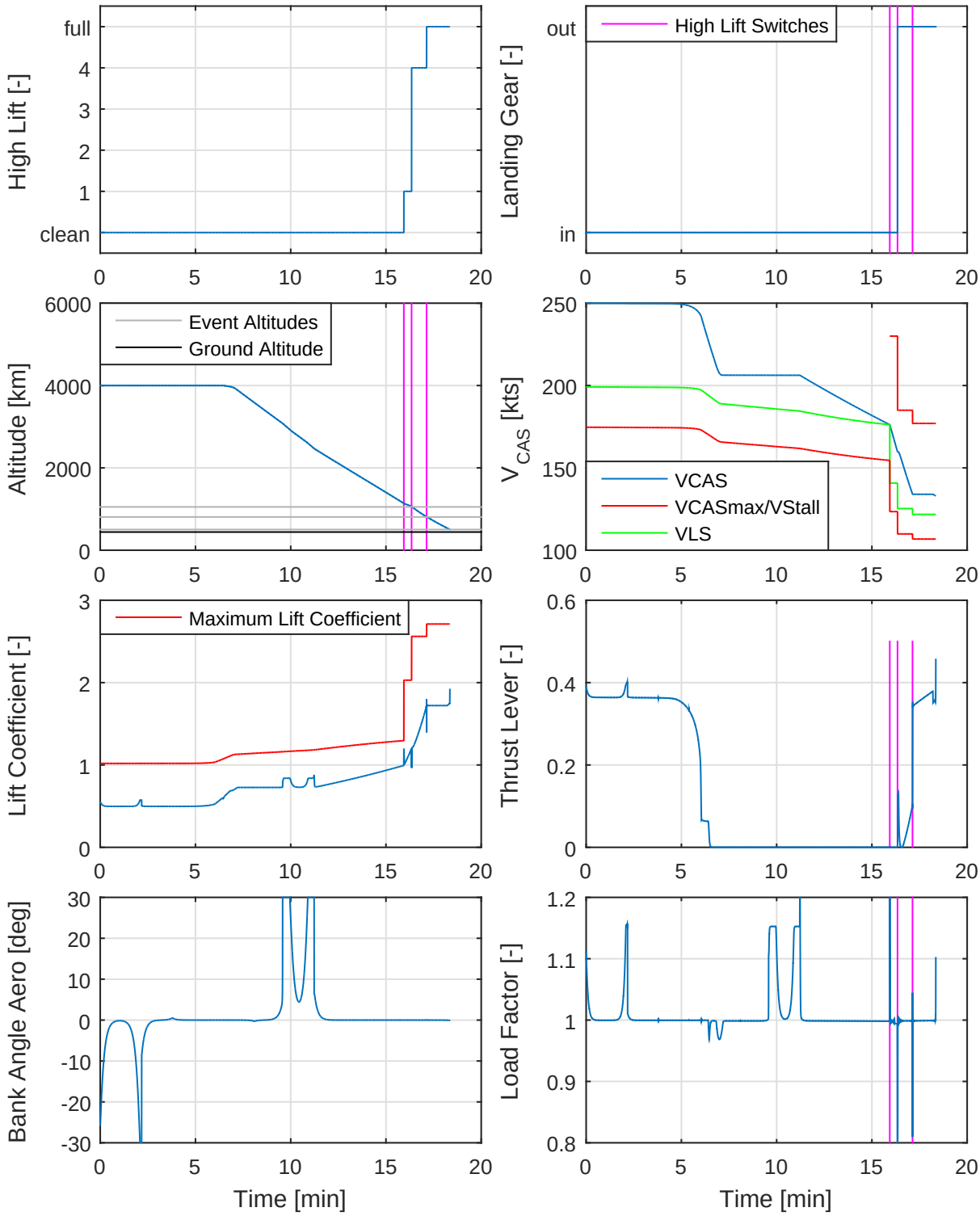


Figure 6.18: Solution of fuel minimal approach optimization under consideration of discrete controls and constraints.

The switch to the first non-clean flap position was mainly used for deceleration and thus a switch subject to the optimization.

6.5 Influence Study

In this section, the initial condition (speed, altitude, mass) as well as the wind condition (speed, direction) are varied. The influence of the parameter on the optimal flap switching structure is evaluated. However, in order to improve the consistency of the solution, an artificial penalty cost must be added to the high lift selection. In the previous section, the switch from the clean configuration to the first non-clean flap setting occurs while the thrust lever is in idle position. Therefore, the change increases the drag which is used for deceleration. However, since the thrust is at idle, the switch does not have a significant influence on the minimal fuel cost function. In case parameter studies are carried out the optimizations converge to different minima with similar cost values.

In order to improve the consistency in the parameter study solutions, an artificial high lift penalty is added. All non-clean high lift selections and landing gear positions are penalized using a simple cost function:

$$J_{p,HL} = 0.05 \cdot \sum_{k} (k-1) \cdot w_k. \tag{6.137}$$

The penalty increases with the aircraft's high lift and ensures that a deployment of the flaps always has a small negative impact on the cost function. A comparison to the case without this additional high lift penalty is given in the appendix C.3. The artificial high lift penalty is used in the previous single optimization as well.

6.5.1 Initial Aircraft Mass

In this section, the influence of the initial approach mass on the trajectory and thus on the flap switching structure is evaluated. Therefore, the aircraft mass is varied from $50t$ to $66t$ in $500kg$ steps. It is considered in the optimal control problem as the initial boundary condition of the mass state. Otherwise, boundary conditions and wind situations remain the same (see Table 6.7). The landing speed is adapted dependent on the considered weight (see section 6.2.3).

In Figure 6.19, multiple plots are shown. The fuel consumption is plotted w.r.t. the initial aircraft mass together with the switching cost and the artificial high lift cost. All penalties are added to the fuel consumption in order to make the plots better comparable. Additionally, the altitude profile as well as the calibrated air speed are displayed w.r.t. time backwards from the decision altitude. Thus, all optimizations are aligned at their final boundary condition. In both plots, the colors range from dark blue to green. The colors represent the initial aircraft masses ranging from $50t$ to $66t$.

In the last subplot, the switching structure dependent on the aircraft mass is shown. Horizontal gray lines indicate the aircraft time frame. As with the plots before, they are aligned w.r.t. their final time. Switches are indicated by colored lines. Additionally, the magenta line indicates the alignment with the runway during approach. It is the point in time where the aircraft completes its final turn and is on direct approach to

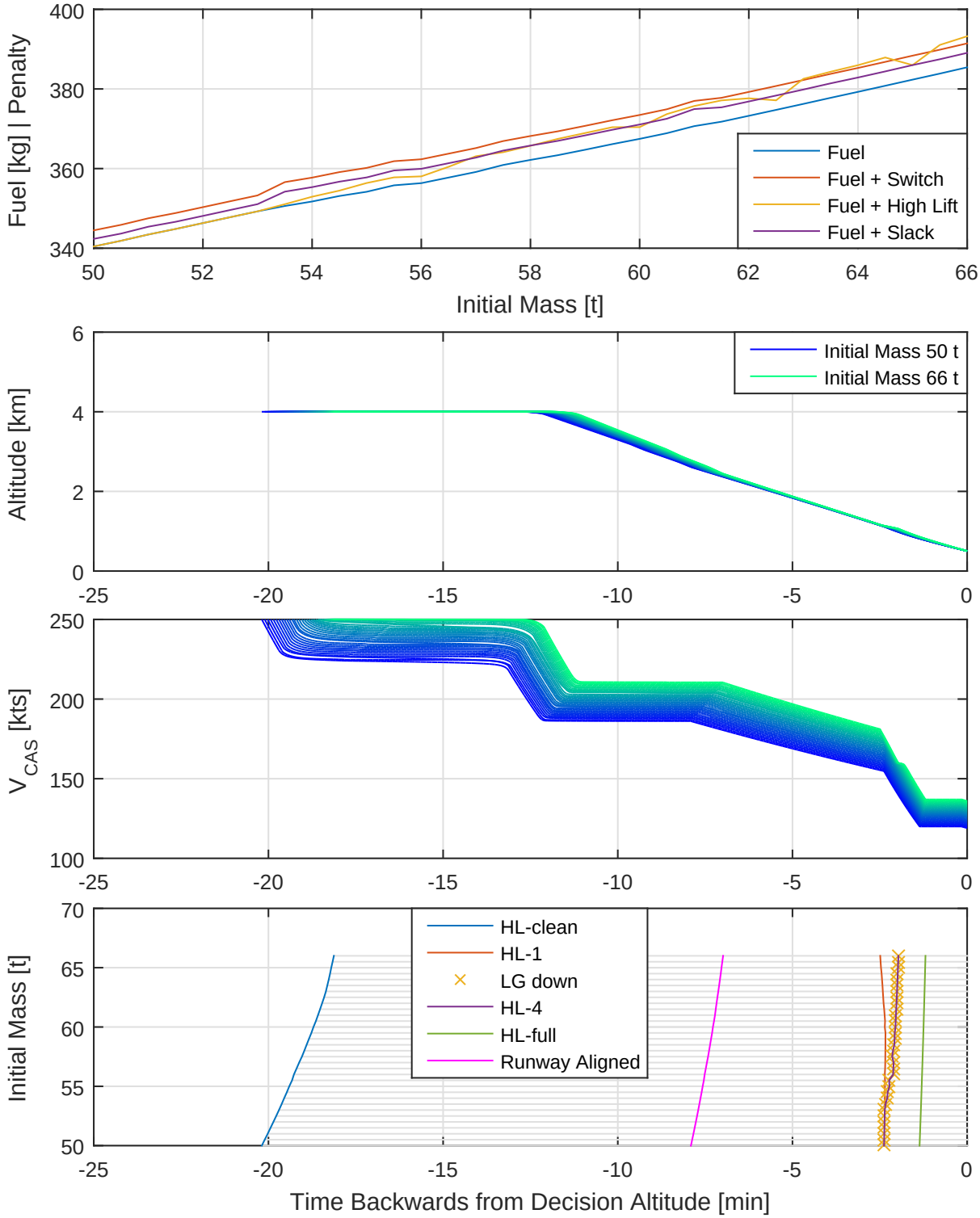


Figure 6.19: Fuel minimal discrete control approach optimization with varying initial aircraft mass.

the runway. Within the optimal control problem, it is the point from which the discrete controls are considered.

The fuel consumption increases almost linearly with the initial mass. Contributions of the penalties to the overall fuel cost are relatively small. Compared to the car optimization, the contribution of the switching cost is much lower. Therefore, it shows that the selection of a suitable penalty scale parameter is dependent on the application involved.

For all aircraft masses, the initial altitude is maintained until the continuous descent begins. Lighter aircraft begin their descent earlier. The speed profile chosen for the approach is dependent on the aircraft mass. However, all profiles are very similar. Heavier aircraft maintain the initial speed whereas lighter aircraft decelerate until an optimal cruising speed is reached. Afterwards, the speed profile follows the same strategy as in the single optimization case described above. Additionally, the aircraft mass dependent final landing speed V_{APP} can be seen.

It the last subplot which contains the switching structure, it can be seen that lighter aircraft require longer approaches. Additionally, lighter aircraft perform two discrete switches. The first deploys the landing gear and switches the flaps to the fourth configuration at $2000ft$ above ground. The full flap configuration is selected once the $1200ft$ above ground level is reached. Therefore, both switches are enforced by operational constraints. As the initial aircraft mass becomes heavier (approximately $54t$), an additional switch which moves to earlier points in time emerges. This switch is subject to the discrete control optimization. Overall, the obtained switching structure and time histories are very consistent within the parameter study.

6.5.2 Initial Altitude

In this parameter study, the influence of the initial approach altitude on the optimal solution is determined. The initial altitude is varied from $3000m$ to $5000m$ altitude in steps of $100m$. The initial aircraft mass is set to $63t$. All other boundary conditions as well as the wind situation remain the same as in the single optimization (see Table 6.7).

In Figure 6.20, the results from the initial altitude optimizations are shown. The layout of the subplots is the same as in the mass study. The initial mass axes are replaced by the initial altitude in kilometers.

The required full consumption decreases with an increase in altitude. This can be explained by the higher potential energy. The relationship is almost linear. As before, the switching cost and the artificial high lift penalty have only a minor contribution to the overall cost value.

In the altitude and speed plots, the colors range from blue ($3000m$) to green ($5000m$). The initial altitude is always maintained until the point for the continuous descent approach is reached. Lower altitudes converge to the same descent profile as higher ones. Before the continuous descent begins, the speed is reduced to an optimal value. It is approximately the same for all optimizations. The final landing speed V_{APP} is not influenced by the initial altitude.

The switching structure is constant throughout the initial altitude parameter study. This can be explained by the fact that all optimizations have the same final approach for altitude and speed. All difference are found in the clean initial approach segment and the flight duration increases slightly with the initial altitude.

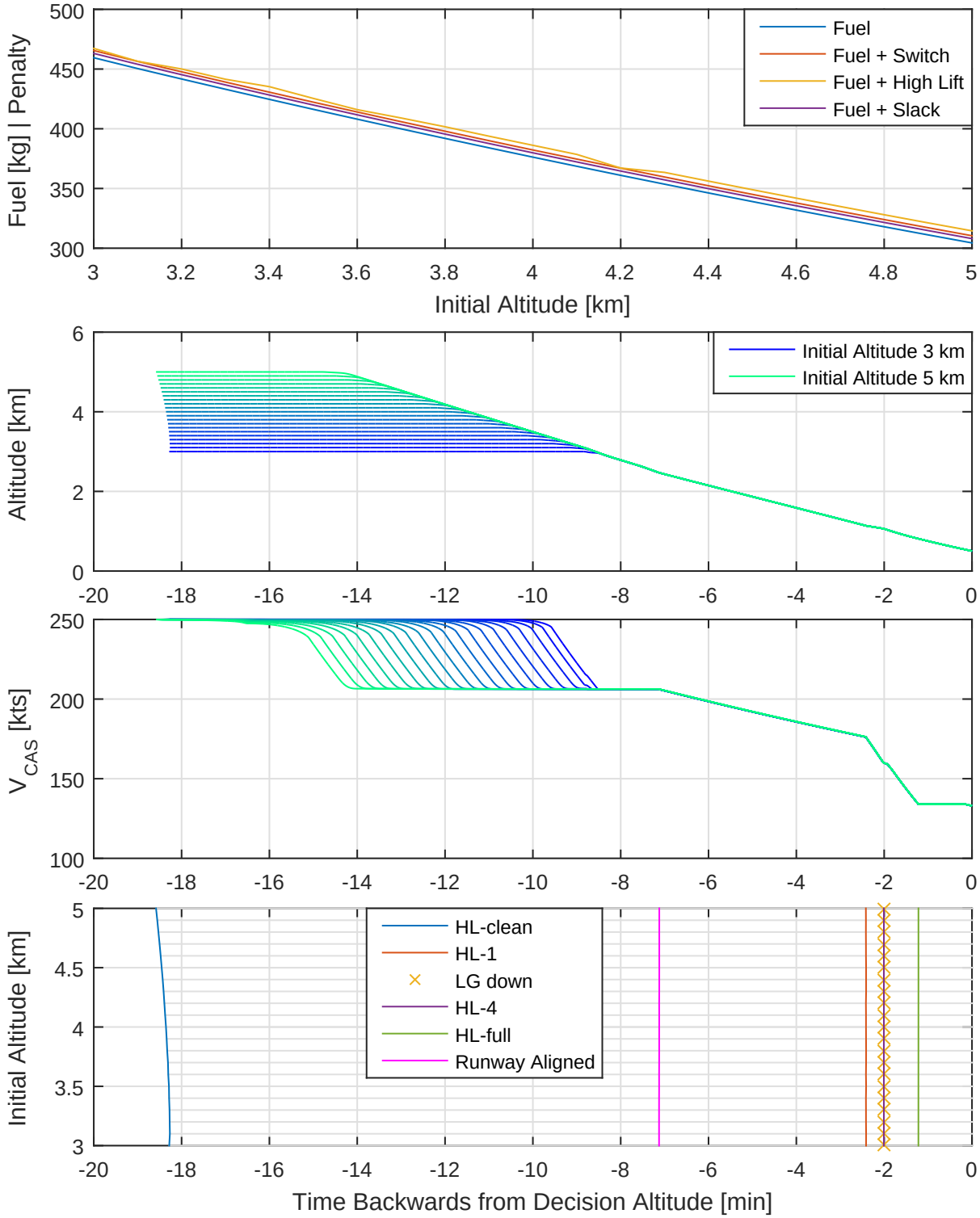


Figure 6.20: Fuel minimal discrete control approach optimization with varying initial altitude.

6.5.3 Initial Speed

In this section, the influence of the initial approach speed on the switching structure is determined. The initial calibrated air speeds range from $200kts$ to $320kts$ in $10kts$ steps. Obviously, some initial speeds violate the regulation that below FL100 a maximum speed of $250kts$ must not be exceeded. Therefore, these optimizations have to be regarded as an extension. The initial altitude is set to $4000m$. All other boundary and wind conditions are the same as in the single optimization (see Table 6.7).

In Figure 6.21, the results of the initial calibrated air speeds are shown. The fuel consumption decreases with an increase in the initial speed as more kinetic energy is available. As before, the influence of the penalties is small compared to the fuel cost. The slowest initial speed $200kts$ results in a higher fuel consumption. This is due to the convergence to another minimum which is explained below. The initial altitude is maintained for all initial speeds until the continuous descent approach begins. All optimizations follow the same altitude profile. The calibrated air speed plot shows that slower aircraft have longer fuel minimal approaches. High initial speeds (above approximately $250kts$) decelerate to an optimal speed for the given altitude. Afterwards, they decelerate further to the optimal descent speed. The fact that the initial altitude is maintained as long as possible shows that the potential energy is a far better energy storage than the kinetic energy.

For an initial speed of $200kts$, the solution is significantly different. This is due to the fact that this initial speed is below the optimal descent speed. An acceleration during the approach is not allowed in the optimization. Therefore, the result is a different speed profile. In the switching structure plot, the flaps are extracted much earlier compared to the other solutions. Since all initial calibrated air speeds above $200kts$ follow the same final approach, the switching structure is constant.

6.5.4 Wind Speed

In this and the following section, the wind influence on the fuel minimal approach solution is evaluated. Therefore, first the influence of the wind speed is discussed. The profile of the Prandtl layer is aligned with the runway (see Figure 6.22). The geostrophic wind speed ranges from $0m/s$ to $30m/s$ in $2m/s$ steps. These wind speeds are equivalent to $0 \ldots 11$ on the BEAUFORT scale [139]. Therefore, calm conditions up to strong high winds are taken into account. Negative winds are not considered as this normally leads to a direction change in runway operations. Otherwise, boundary conditions are the same as in the single optimization (see Table 6.7).

In Figure 6.23, the results from the wind speed study are shown. The structure of the plots is very similar to that of the previous studies. Two additional subplots are added. In the first new subplot, the aerodynamic and kinematic speeds are plotted together. Color coding is used to differentiate low from high wind speeds. The kinematic speed plots range from black to copper colors whereas the aerodynamic speed plots range from blue to green. Additionally, the bank angle over time is shown.

With higher wind speed, the fuel consumption increases significantly. The reason lies in longer flight times at low aerodynamic speed (this is explained below). The impact of the switching cost and artificial high lift cost is relatively small compared to the overall fuel consumption.

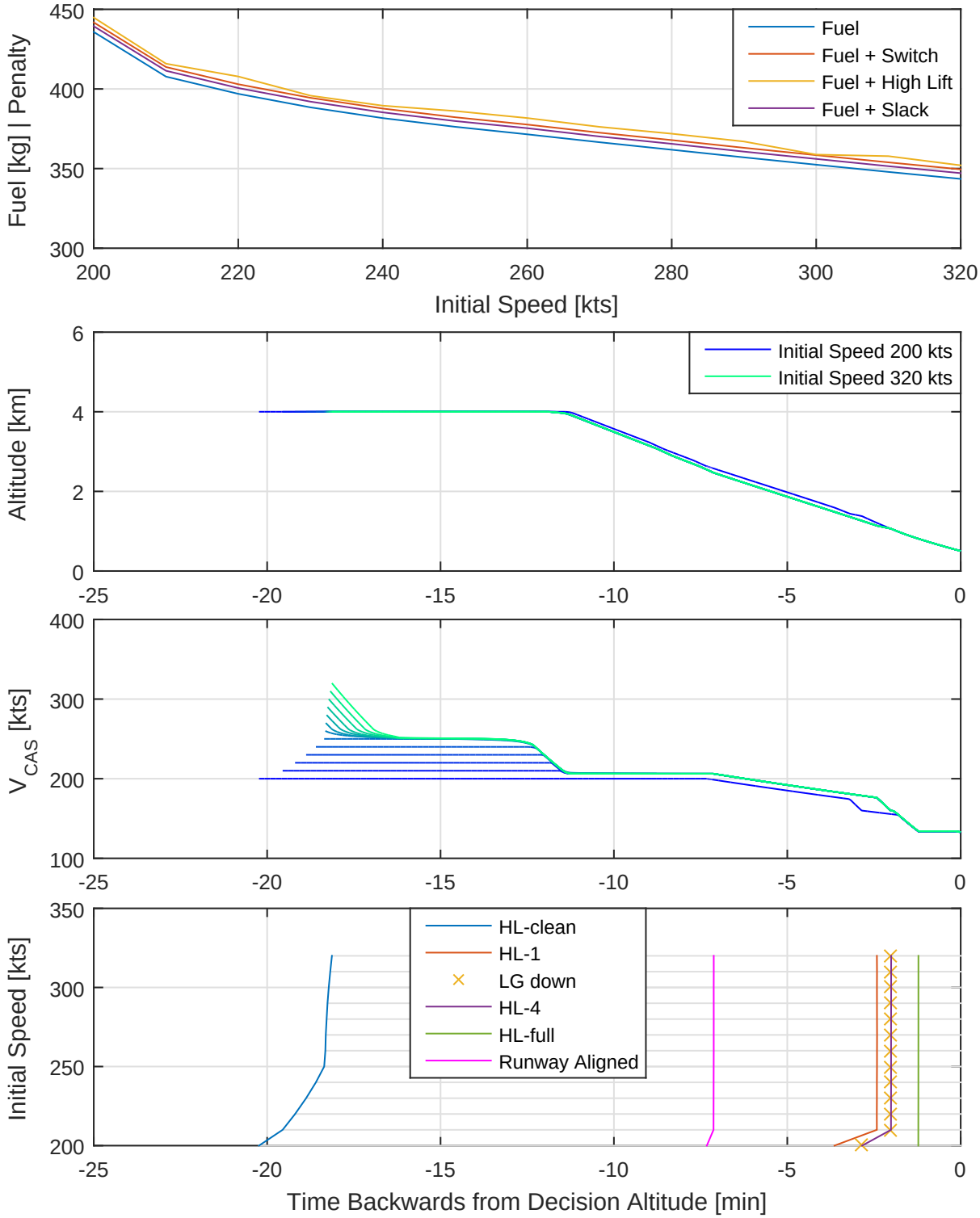


Figure 6.21: Fuel minimal discrete control approach optimization with varying initial calibrated air speed.

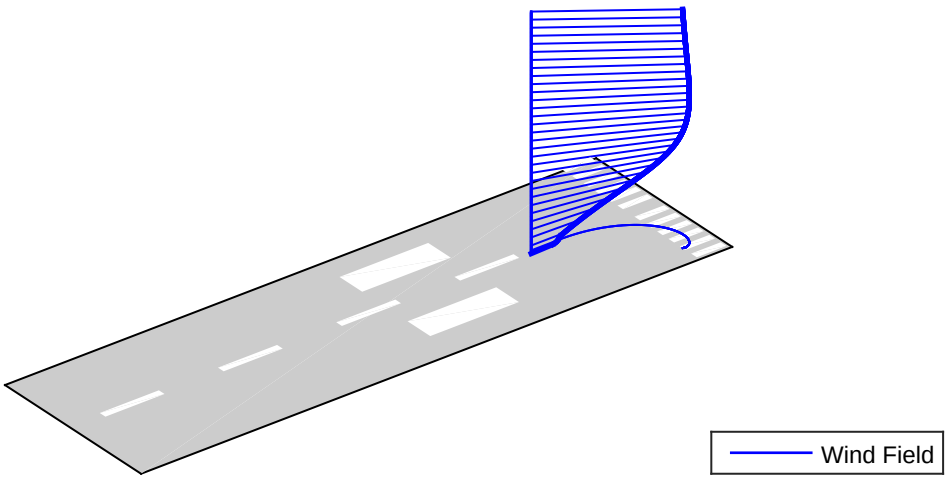


Figure 6.22: Wind speed with respect to runway alignment.

In the altitude plot, the initial altitude is maintained. With increasing wind speed, the beginning of the continuous descent moves to earlier points in time. The calibrated air speed is decreased during the downwind segment. At higher wind speeds, the initial speed is slightly reduced.

Due to the tail wind in the downwind part of the approach, the kinematic speed is increased significantly and therefore the turn occurs much earlier at higher wind speeds. After approximately half of the turn to runway alignment, the aircraft has headwind and thus the kinematic speed drops below the aerodynamic speed. With higher winds, the kinematic speed becomes relatively small. Therefore, these flight segments become much longer compared to the downwind segments. As it was shown in the single optimization, the deployment of landing gear and landing configuration required the thrust to be increased. Together with the longer flight times in the headwind approach segment, the increase in the fuel consumption can be explained.

In the last subplot, the discrete control switching structure is shown. With higher wind speeds, the fuel minimum flight time increases as well. Although the initial approach is flown with tailwind, the final approach has much slower kinematic speeds and thus significantly higher flight times. The discrete control switches and the runway alignment move to earlier points in time.

The ground tracks for the different wind speeds are shown in Figure 6.24. With increased headwind speed, the turn of the aircraft is drawn further outwards. Additionally, it becomes more focused towards the middle. This reflects the stronger drop in the kinematic speed shown in Figure 6.23.

6.5.5 Wind Direction

In this section, the influence of the wind direction on the fuel minimal approach solution is evaluated. The geostrophic wind speed is set to $10m/s$ and the wind angle relative from the runway ranges from $-90°$ to $90°$ in $15°$ steps. The wind direction relative to the runway is shown in the Figure 6.25. The boundary conditions remain the

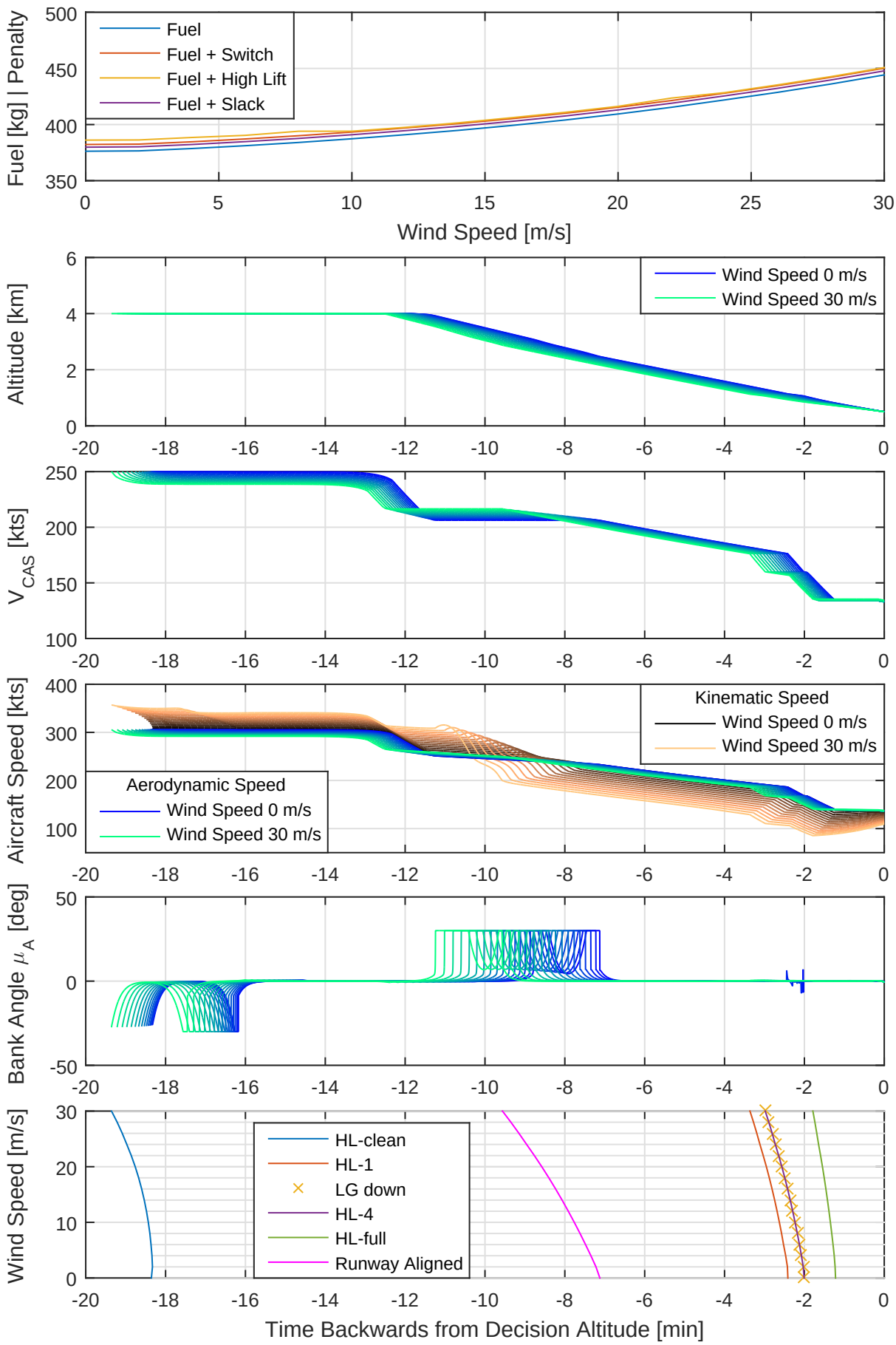


Figure 6.23: Fuel minimal discrete control approach optimization with varying wind speed.

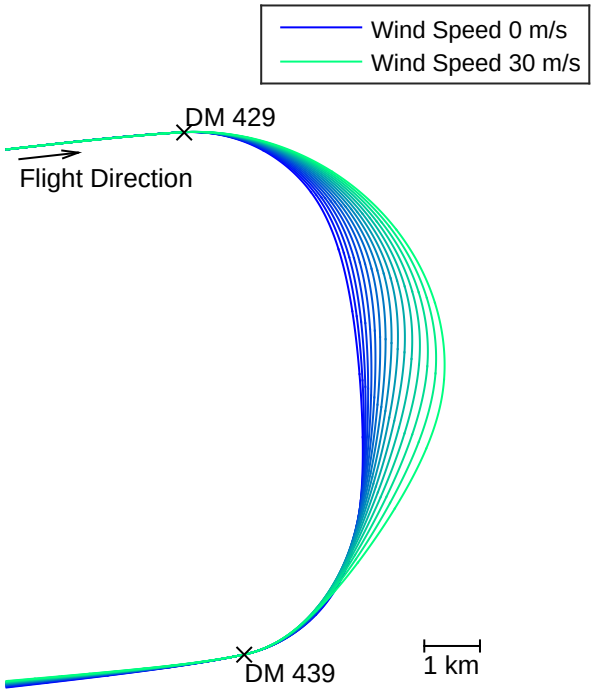


Figure 6.24: Ground track of approach optimimization with varying wind speeds.

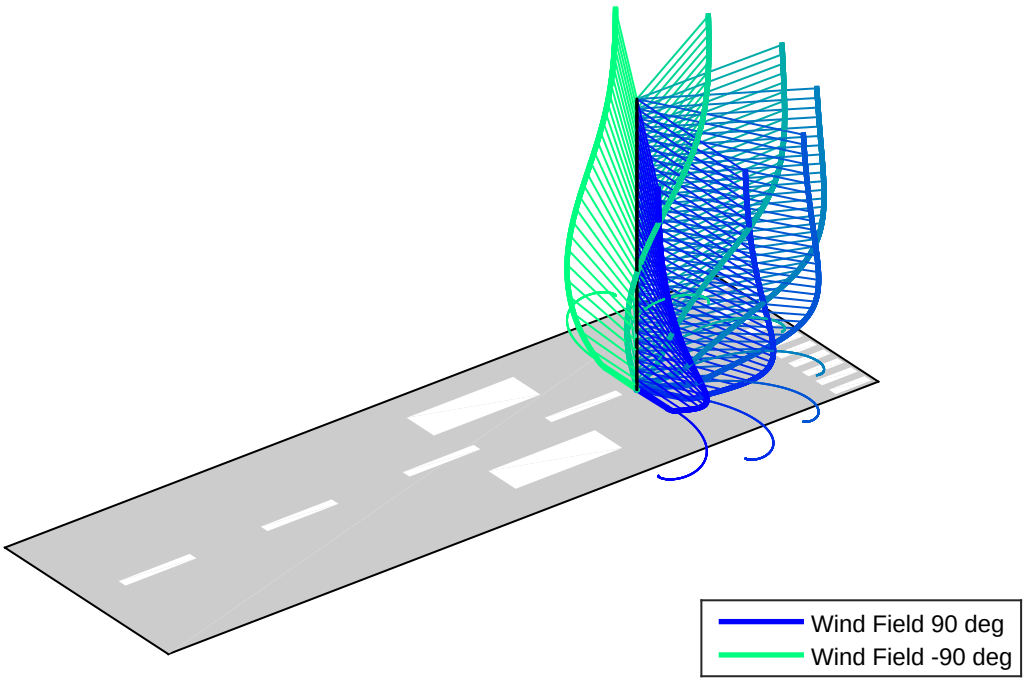


Figure 6.25: Wind direction with respect to runway alignment.

same as in the previous or single optimization case (see Table 6.7).

In Figure 6.26, the solutions of the fuel minimal approaches for different wind directions are shown. As with the wind speed, the plot is expanded by aerodynamic and kinematic speeds as well as the aerodynamic bank angle.

The fuel consumption is lowest for a runway cross wind of $90°$ as the turn is flown in downwind conditions. At a relative wind direction of approximately $-30°$, the fuel consumption is highest. The altitude and speed profiles only show a slight variation. During the turn, dependent on head- or tailwind conditions, the kinematic speed either drops or increases significantly. The calibrated air speed is not allowed to increase in the optimization. Where the wind direction is almost perpendicular to the flight direction, the kinematic speed becomes almost the same as the aerodynamic speed. The kinematic speed is lower than the aerodynamic speed after the turn to the final approach.

In case of $+90°$ relative wind direction the Ekman wind rotation causes the aircraft to fly the beginning of the final approach in tailwind conditions. Therefore, the kinematic speed remains higher than the aerodynamic speed. Once the aircraft has descended further, the directions becomes perpendicular to the runway. Then, the kinematic speed approaches the aerodynamic speed. A similar behavior is visible in the initial approach segment where, due to the Ekman wind rotation, the aircraft has a slight tailwind (relative wind angle $+90°$). Finally, in Figure 6.27, the ground tracks for the different wind directions are shown.

6.5.6 Sea Level Temperature Delta

The ISA atmospheric model defines a reference temperature and pressure at sea level. Therefore, in this and the next section, the influence of temperature and pressure offsets are evaluated. The default ISA temperature is $T = 288.15K$ which is $T = 15°C$. This study ranges the temperature delta ΔT from $-20K$ to $20K$ in $2K$ steps. In Celsius, the temperature range is from $-5°C$ to $35°C$. Otherwise, the settings are as in the single optimization case.

Figure 6.28 shows the results of the study. The layout of the plots is similar to the studies before. An additional subplot showing the aircraft drag is added. In the first subplot, a slight drop of the fuel consumption with increasing temperature is visible. High temperatures lead to a higher optimal approach speed (see V_{CAS} subplot). The drag increases with the temperature as well. However, overall the flight time is reduced which has a higher impact and thus the fuel consumption decreases. In the altitude subplot, no significant differences are visible.

As already mentioned, the flight duration decreases with increased temperature. Additionally, the first flap position moves to earlier points in time. The switch is mainly motivated by the minimal speed limitation in the clean configuration. Due to the fact the a temperature increase decreases the air density, the lift is reduced. Therefore, the aircraft has to deploy the flaps earlier.

6.5.7 Sea Level Pressure Delta

The ISA atmosphere's default pressure at sea level is $1013.25hPa$. According to [140] the sea level pressure approximately ranges from $985hPa$ in low pressure areas up to

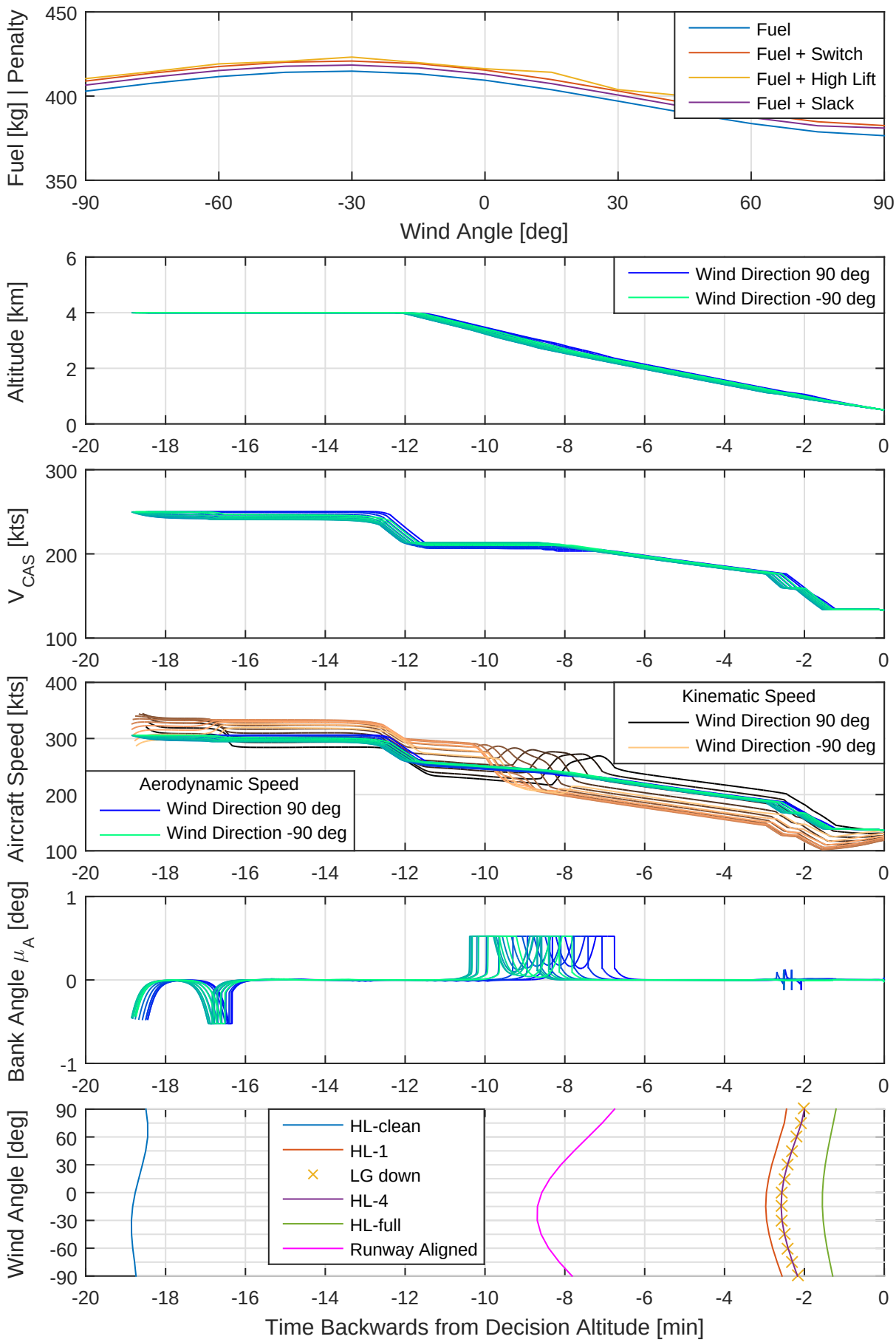


Figure 6.26: Fuel minimal discrete control approach optimization with varying wind direction.

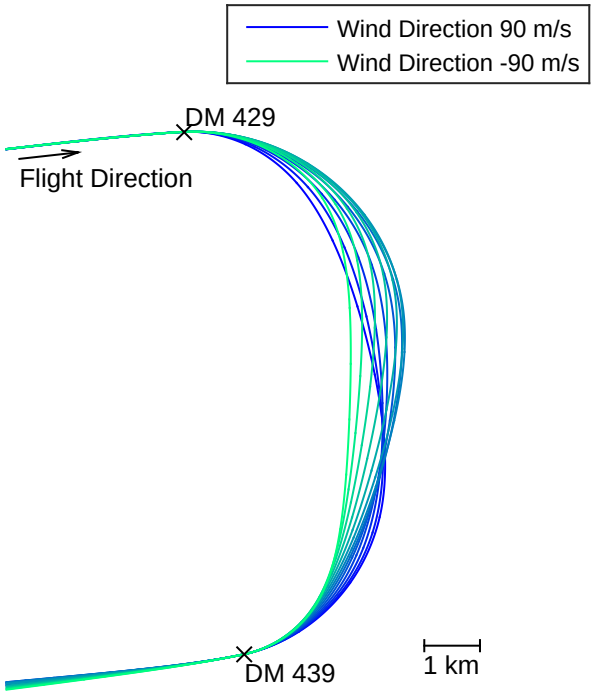


Figure 6.27: Ground track of approach optimimization with varying wind directions.

$1035hPa$ in high pressure areas. This leads to a ISA pressure offsets Δp from $-2825Pa$ to $2175Pa$. The optimizations are carried out in $200Pa$ steps. Otherwise, the settings are as in the single optimization case.

Figure 6.29 shows the results of the optimizations. It can be seen that the fuel consumption remains almost constant with a tendency to lower fuel consumptions with higher air pressures. With an increase in pressure, the air density increases. In the switching structure plot, the first flap position shifts to later points in time. Additionally, the optimal calibrated air speed is reduced. The aircraft drag is slightly reduced with higher air pressures. The flight time increases slightly. Overall, this leads to an almost constant fuel consumption.

6.6 Comparison to Real Approach Trajectory

In this section, the approach optimization is compared against a real flight trajectory. For this purpose, a flight trajectory from a foreign airline is taken. The scenario considers an approach of an A319 aircraft. All data is taken from the Quick Access Recorder (QAR) that stores various data during flight. Before the flight data can be used it has to be preprocessed. Afterwards an optimization is carried out to match the BADA4 simulated trajectory to the real flight. Thus, it can is assessed whether the BADA4 model can reproduce the real trajectory with sufficient detail. Finally, several fuel minimal approaches are carried out. These include free final time and fixed final time conditions.

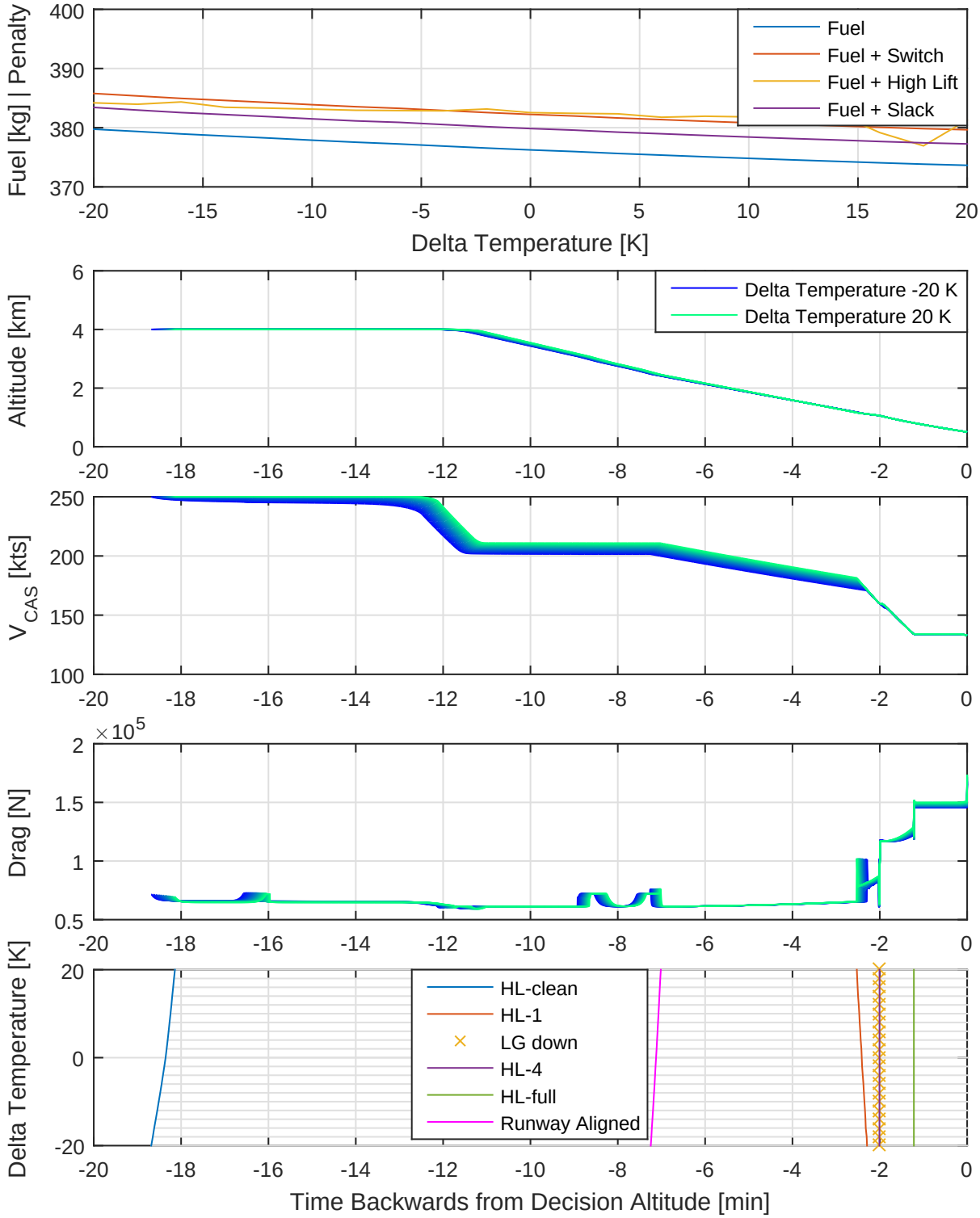


Figure 6.28: Fuel minimal discrete control approach optimization with varying sea level temperature delta.

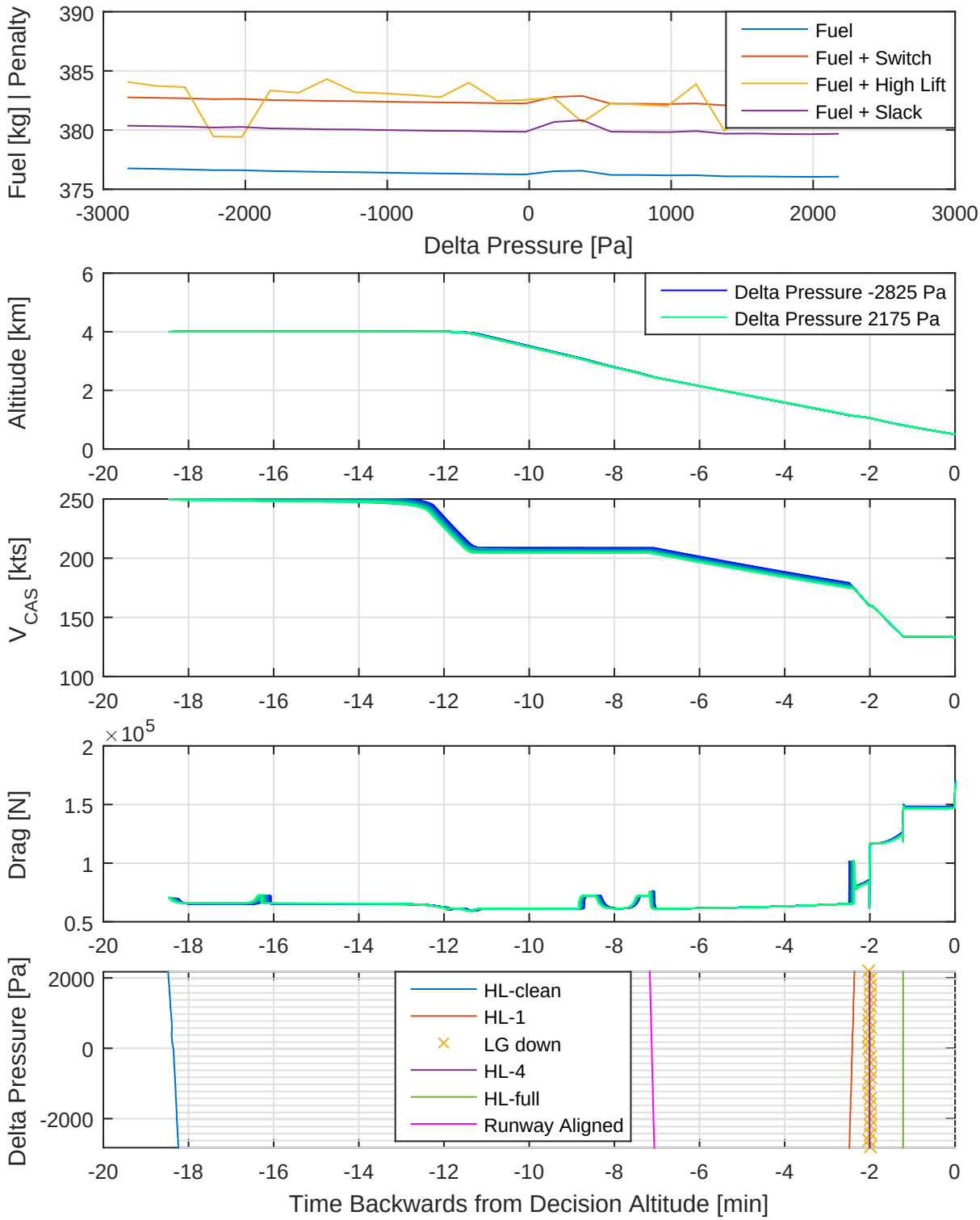


Figure 6.29: Fuel minimal discrete control approach optimization with varying sea level pressure delta.

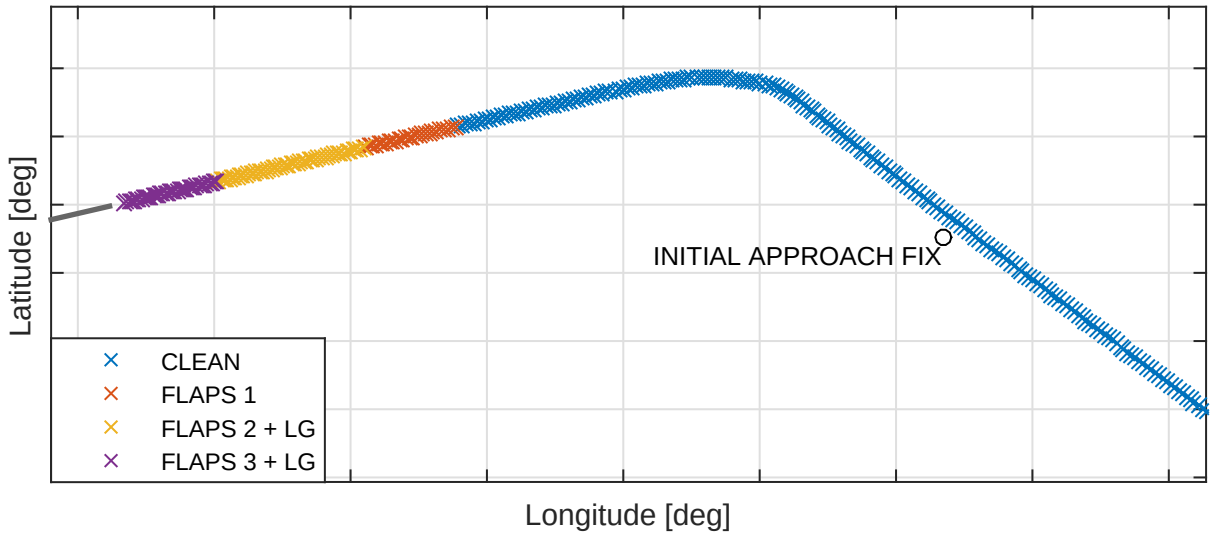


Figure 6.30: GPS track of the approach. Colors indicate the different high lift settings. Additionally, the initial approach fix as well as the runway layout is shown.

6.6.1 Preprocessing of Flight Data

In order to use the QAR data in the further process of this section, it has to be preprocessed. The considered section the flight is shortly before the initial approach fix down to the decision altitude (see Figure 6.30). Actual flight time is approximately 10 minutes.

The QAR data includes the position ϕ^G, λ^G, h^G, the kinematic speed V_K^G, the kinematic angles χ_K^G, γ_K^G, the calibrated air speed V_{CAS}, the aircraft mass m, and the fuel flow f_{flow}. Additionally, the aircraft roll angle is used to approximate the bank angle μ. During flight, the aircraft approximates the wind speed V_W and direction χ_W (where the wind originates from). Finally, the angles of the slats and flaps are stored. In the data at hand, the standard atmosphere offsets ΔT and Δp are not available. Therefore, both values are set to zero.

Data Smooting

In order to save memory, most of the QAR data is stored with low frequency and a reduced number of significant digits. Therefore, high frequent influences such as turbulences are not recorded with sufficient resolution. Additionally, turbulences are not considered in the aircraft model. For these two reasons, the QAR data is smoothed by applying a moving average algorithm. This shall ensure that the noise in the measured data is reduced.

As mentioned, the wind calculated by the aircraft is stored as the wind speed and direction the wind originates from. In Figure 6.31 the both data are plotted w.r.t. the aircraft altitude. The noise is significant and increased with lower altitudes. Additionally, the plots show the smoothed data for both measurements. The smoothed data is used in the optimization. Furthermore a spline for the aerodynamic dataset is created w.r.t. the aircraft altitude. Additionally to the wind data, the kinematic course angle χ_K^G, the kinematic climb angle γ_K^G, and the bank angle μ are smoothed.

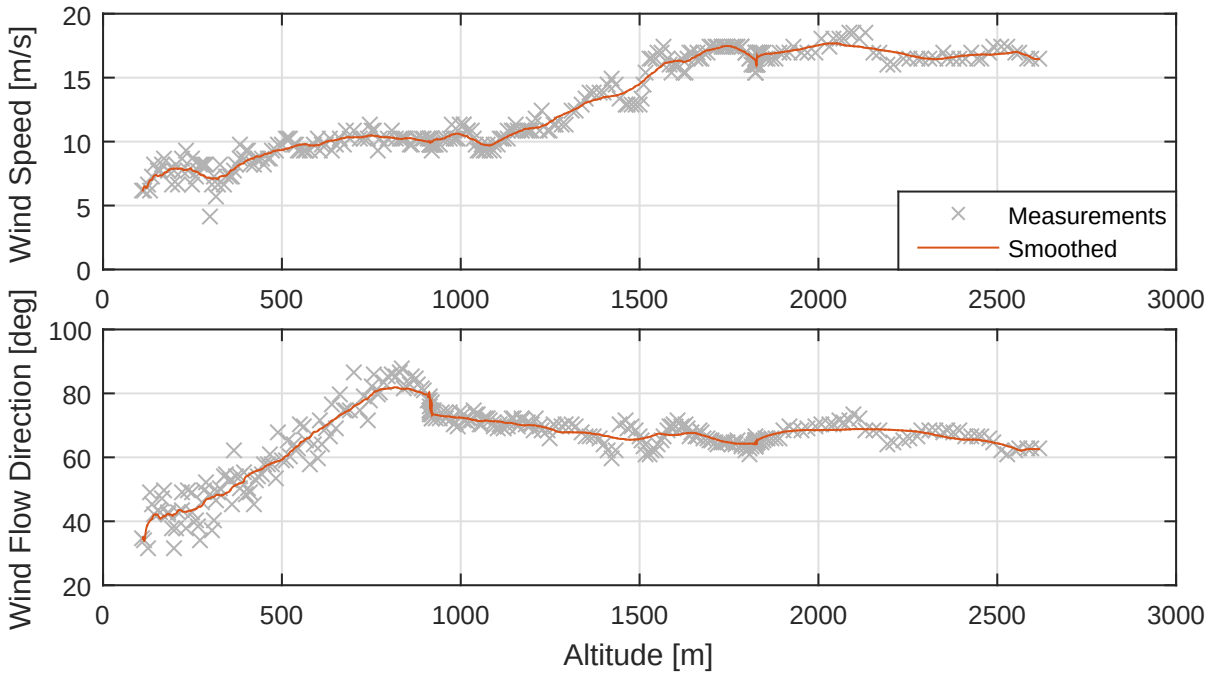


Figure 6.31: Measure wind speed and wind direction w.r.t. altitude and smoothed wind data.

Fuel Flow and Aircraft Mass

In order to fit the BADA4 model to the real flight data, the initial mass of the aircraft is required. Although it is theoretically possible that the aircraft mass is determined through optimization, practically this is difficult. At lower aircraft masses, the controls will always have a better effectiveness on the aircraft movement. Therefore, the optimization algorithm is always motivated to reduce the aircraft mass to the lower possible value.

The QAR data at hand stores two time histories related to the aircraft mass. Fuel flow measurements from both engines are available. Both are combined to obtain the overall fuel flow f_{flow}. Additionally, the current aircraft mass is guessed during flight. Due to the fact that this value is stored with very low frequency, the first mass data point before the initial approach fix is used as starting point for the scenario and as the initial aircraft mass.

In Figure 6.32 the aircraft fuel flow and mass are shown w.r.t. time. Additionally to the aircraft mass, the fuel flow was integrated to generate a mass history with a higher density. The integrated mass as well as the guessed measurements show a relatively good match. However, the integrated mass is always slightly lower than the mass guessed by the aircraft. In the fuel flow plot, a significant increase is located at approximately 100 seconds. This increased fuel consumption is directly visible in the integrated aircraft mass history but not in the QAR data. In the following, the integrated aircraft mass is used as reference. Additionally, it is assumed that the QAR data used as initial aircraft mass is sufficiently accurate.

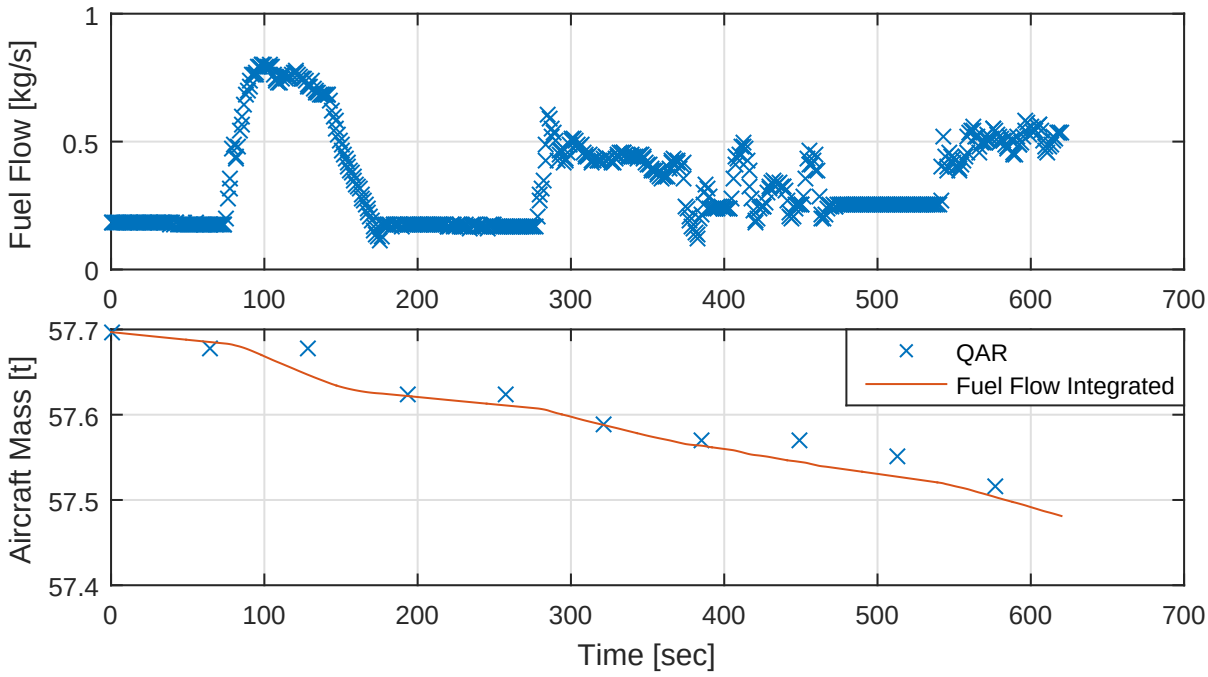


Figure 6.32: Approach aircraft fuel flow and aircraft mass. Fuel flow is integrated to determine more accurate mass history.

Table 6.8: Angles of the high lift configurations for the Airbus A319.

High Lift Setting	**Slat Angle [deg]**	**Flap Angle [deg]**
UP / CLEAN	0°	0°
FLAPS 1	18°	0°
FLAPS 1+F	18°	10°
FLAPS 2	22°	15°
FLAPS 3	22°	20°
FLAPS FULL	27°	35°

Flap Settings

In this chapter, the high lift and gear selections are optimized. Therefore, the data is extracted from the QAR. In the data not the actual pilot selection is stored but the current position of the flaps and slats (high lift devices at the front and the back of the wing). Figure 6.33 shows the time history for the both high lift devices as well as the aircraft altitude above ground. The high lift data should represent the deflection angle in radians. However, there seems to be an scaling factor involved which is currently not known. Since high lift and gear settings are extracted from the changes rather than the actual values this is not an issue.

In Table 6.8 the slat and flap angles for the different high lift selection are shown [134]. Since most of the times only a change either in the slap or the flap angle occurs, the information can be used to extract the actual high lift setting selected by the pilot.

Comparing Figure 6.33 with Table 6.8 gives the following high lift selections. At first, the clean configuration is selected. Afterwards, the slat angle changes but the flap position stays at the same value. This indicates the selection of the FLAPS 1 config-

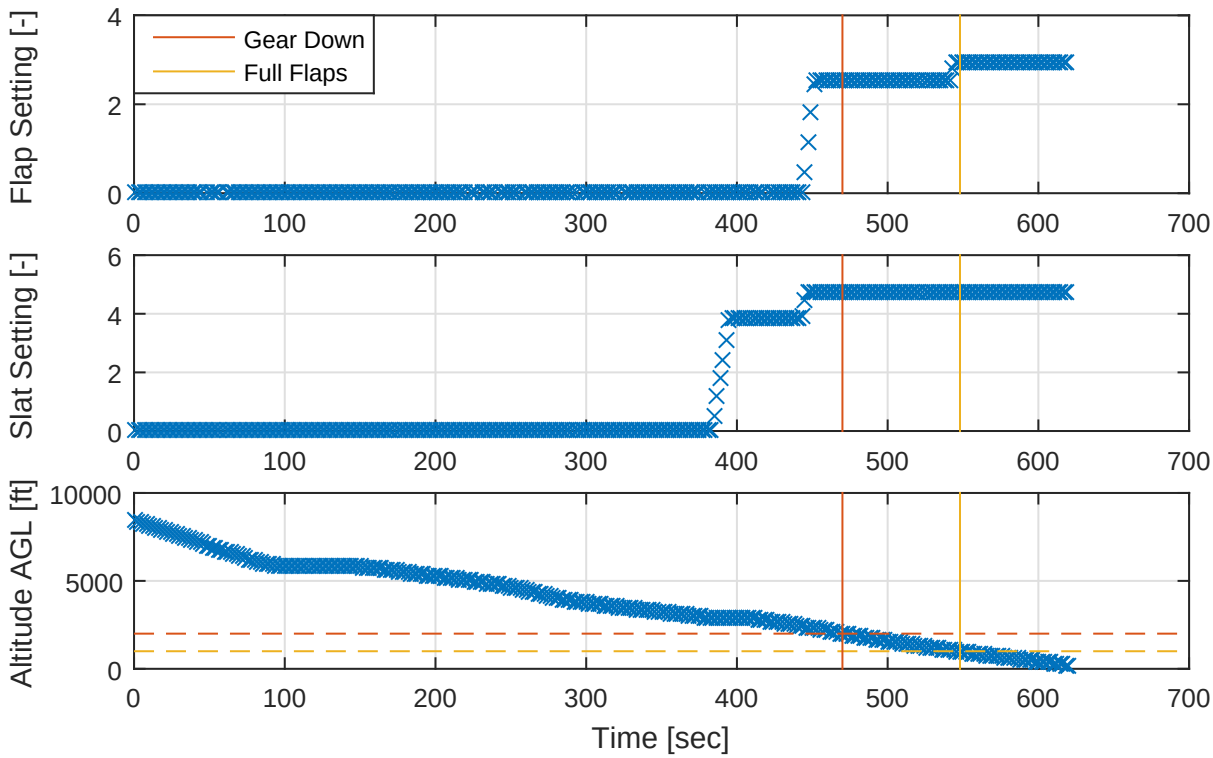


Figure 6.33: Flap and slat setting for approach. Changes in both flaps and slats are used to determine the selected high lift configuration.

uration. The second switch changes the slat and flap position at the same time representing a switch to FLAPS 2. It occurs shortly before the $2000ft$ AGL altitude is passed. Therefore, it can be assumed that the gear is deployed at the same time. The last switch occurs at the $1000ft$ AGL altitude. Since only the flap angle is changed the FLAPS 3 configuration is selected by the pilot. Thus, the aircraft lands in the FLAPS 3 configuration rather than in the full flaps configuration.

From the data above it is determined that the aircraft deploys the landing gear with the selection of the FLAPS 2 configuration. Unfortunately, the BADA 4 Family does not provide aerodynamic data for the FLAPS 2 configuration with extended landing gear. Therefore, the data must be approximated. The following configuration (FLAPS 3) includes aerodynamic data for both the gear down and gear up case. Using the delta of the drag coefficients of this configuration the FLAPS 2 gear down drag coefficients are approximated.

6.6.2 Fitting the BADA4 Model

In order to determine if the BADA4 model reproduces the fuel consumption of the real flight, it has to follow the recorded trajectory. For this reason an optimal control problem is setup in *FALCON.m* that performs a least square fit w.r.t. the flight data.

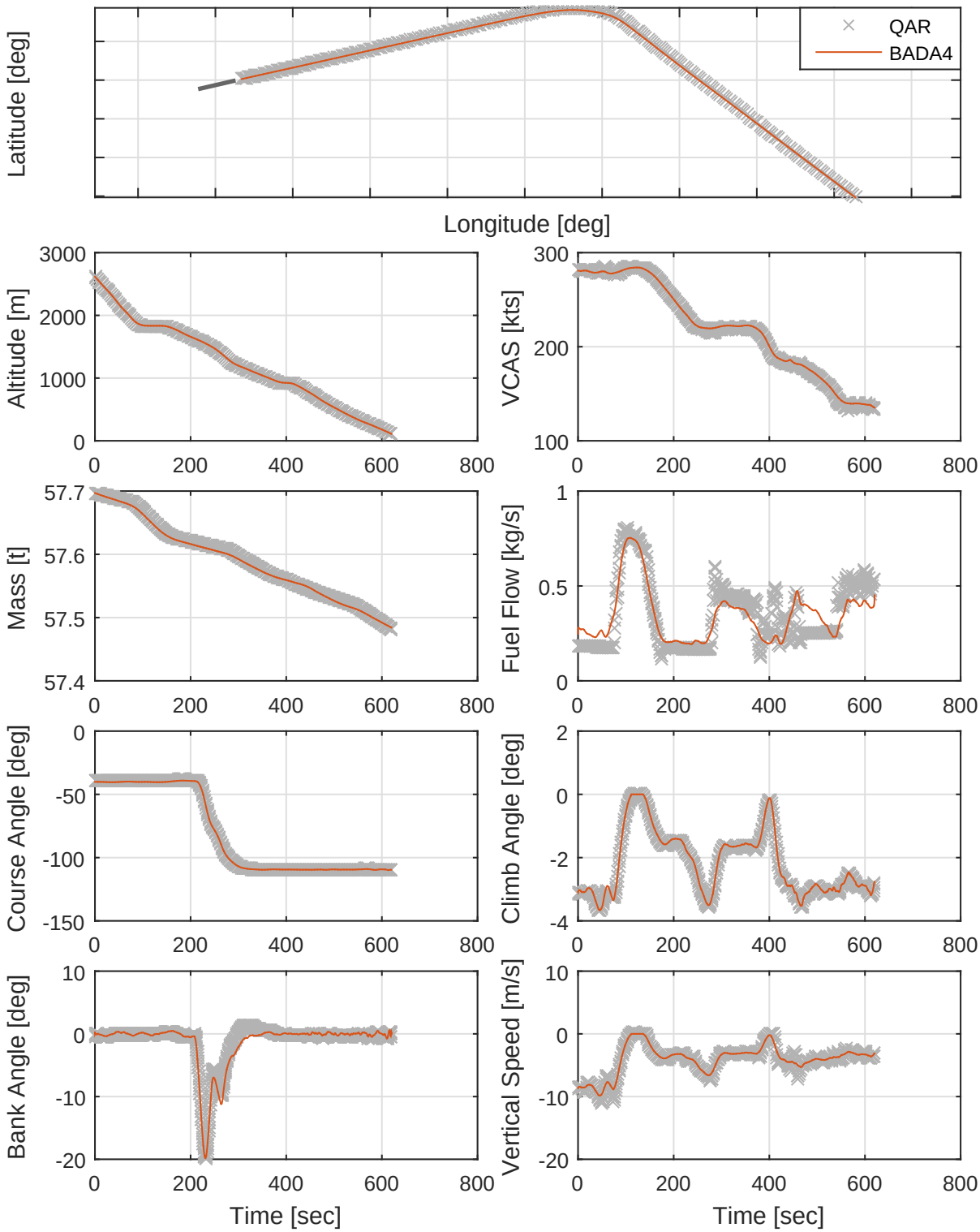


Figure 6.34: Fitted trajectory of BADA4 model with the real flight data using a least square optimization.

The cost function

$$J = \int_{t_0}^{t_f} \begin{bmatrix} 1 \\ 1 \\ 10^{-12} \\ 10^{-8} \\ 10^{-3} \\ 10^{-1} \\ 5 \cdot 10^{-7} \\ 10^{-5} \\ 10^{-3} \end{bmatrix}^T \cdot \begin{bmatrix} \left(\phi^G - \phi^G{}_{REF}\right)^2 \\ \left(\lambda^G - \lambda^G{}_{REF}\right)^2 \\ \left(h^G - h^G{}_{REF}\right)^2 \\ \left(V_K^G - V_{K\,REF}^G\right)^2 \\ \left(\chi_K^G - \chi_{K\,REF}^G\right)^2 \\ \left(\gamma_K^G - \gamma_{K\,REF}^G\right)^2 \\ \left(V_{CAS} - V_{CAS\,REF}\right)^2 S \\ \left(\dot{h}^G - \dot{h}^G{}_{REF}\right)^2 \\ \left(\mu_A - \mu_{A\,REF}\right)^2 \end{bmatrix} dt \tag{6.138}$$

is set up for various aircraft data where $\square_{REF}$ represents the real flight data. Due to the fact that all data have different order of magnitudes, each deviation has to be scaled individually. The scaling data chosen for this optimization are shown in the equation. It is important to note that the fuel flow or the aircraft mass are not considered in the cost function. The optimization is carried out with a fixed final time.

Figure 6.34 shows the fitted result. Both the QAR data as well as the simulated BADA4 data show a very good fit. Although not taken into account in the cost function, both the fuel consumption and the aircraft mass match very well. The final aircraft weight is approximately the same as the integrated aircraft mass from the real flight data. In the following, the fitted BADA4 trajectory is considered as the reference flight trajectory.

6.6.3 Optimized Approach

The approach is now optimized for minimal fuel consumption. As can be seen in Figure 6.35, the aircraft does not exactly passes through the initial approach fix. Therefore, the initial boundary condition for the optimized approach remains the initial real aircraft position. In order to force the aircraft on the same trajectory as before, two additional waypoints are created that define the entry and the exit of the turn. From the runway orientation and Jeppesen flight charts the waypoints for the glide slope intercept (minimum altitude $3000ft$) can be calculated. The gear has to be deployed $2000ft$ AGL and the full flaps stabilized approach must be achieved by $1000ft$. As before, the optimization ends $200ft$ AGL on the glide slope.

All constraints are set up in the same way as in the previous section (see section 6.2). The solution process is shortened as the initial guess for the discrete control optimization is already calculated through the fit of the BADA4 model to the real flight data.

The fuel minimal trajectory may have a different final time. Therefore, all optimizations are carried out with free and fixed final time. Please note that in the fixed final time case some constraints are removed in order to ensure feasibility. The constraint that the aircraft speed at the full flaps point must be $1kts$ above the landing speed is removed. Additionally, the deceleration constraint no longer forces an active deceleration but ensures that in the stabilized configuration phase a maximum deceleration of $2kts/nm$ is not exceeded.

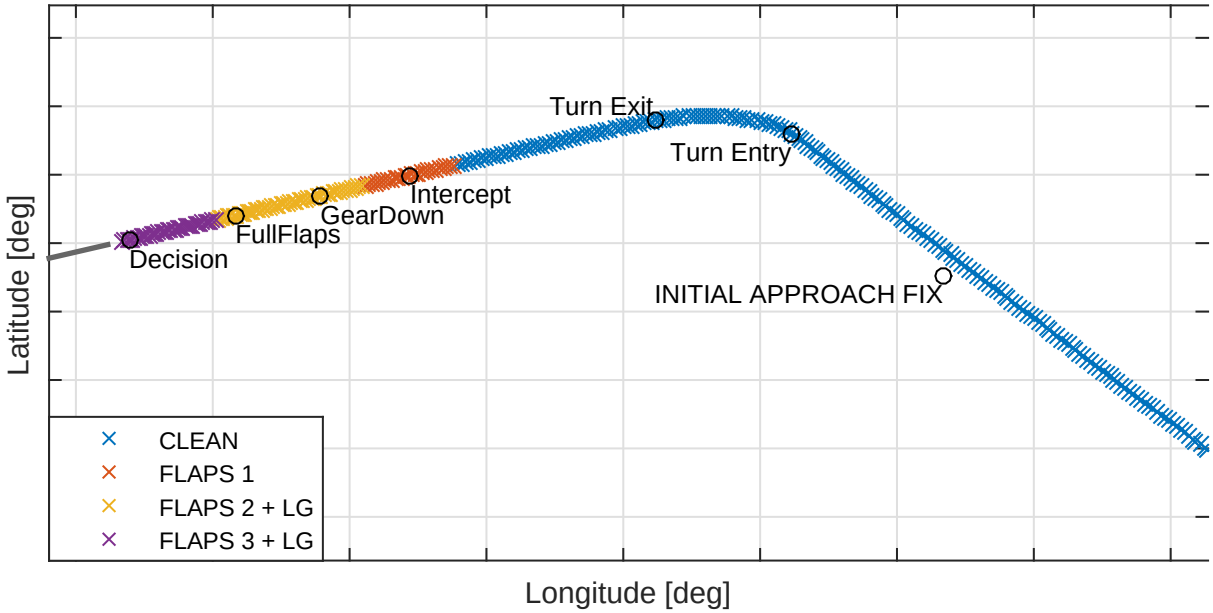


Figure 6.35: Approach optimization scenario with navigation aids and additional waypoints. Colors indicate the high lift and landing gear selection of the reference flight.

Table 6.9: Fuel savings of optimized approach w.r.t. the reference for different final high lift settings as well as fixed and final time. Additionally, the fuel savings for landing in the FLAPS 3 configuration instead of full flaps are shown.

	Flaps 3	**Full Flaps**	**Flaps 3 Compared to Full**
Free Final Time	$15.2\%[32.4kg]$	$11.1\%[23.5kg]$	$4.7\%[8.9kg]$
Fixed Final Time	$12.4\%[26.3kg]$	$8.1\%[17.1kg]$	$4.7\%[9.2kg]$

In the preparation of the flight data it was determined that the aircraft lands in the FLAPS 3 configuration rather than deploying the high lift devices fully. Therefore, the fuel minimal approach optimization is also carried out with both allowed landing configurations. Thus, overall four optimizations are carried out.

Figure 6.36 shows the results of all four optimizations. It can be seen that the track is identical for all solutions. The altitude and calibrated air speed profiles mainly differ for the fixed time and free time cases. This is also visible in the plot displaying the aircraft mass. It also shows the expected result that landing in the FLAPS 3 configuration reduces the fuel consumption further. The fuel savings as well as a comparison of the final high lift configuration are shown in Table 6.9.

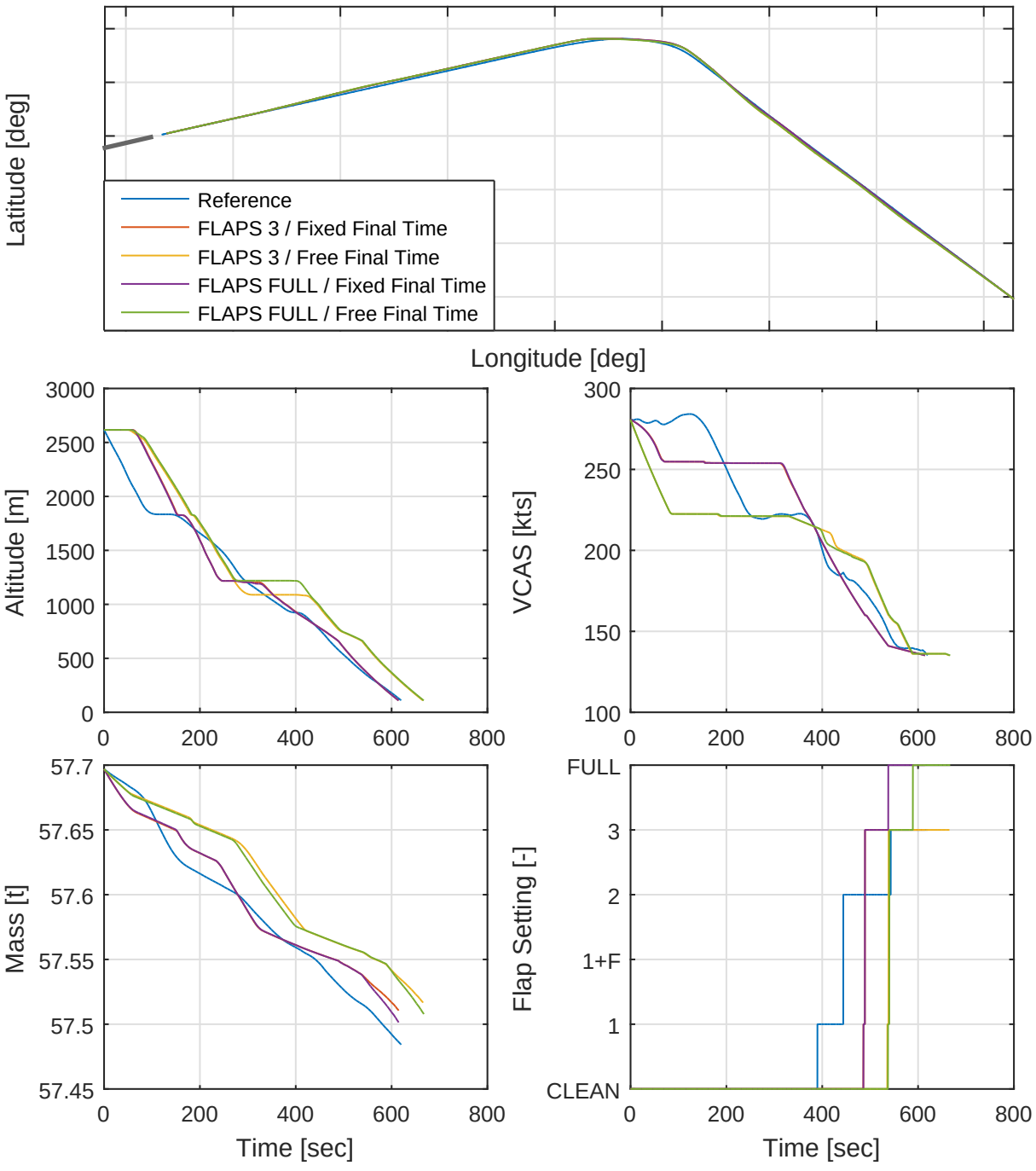


Figure 6.36: Optimal trajectory of BADA4 model for the approach. The fitted flight trajectory is shown as reference.

Chapter 7

Conclusion and Outlook

7.1 Summary and Conclusion

In this thesis, methods to consider discrete controls and discrete constraints in optimal control problems were stated and further developed. The switching sequence of the discrete controls was subject to optimization. The resulting large scale optimal control problems (direct methods) were solved with a newly developed optimal control toolbox *FALCON.m*.

In chapter 2, the continuous optimal control problem was introduced. First, unconstrained and constrained optimization problems were discussed together with numeric gradient based solution approaches. Second, the continuous Optimal Control Problem (OCP) was stated. It was discretized in time and thus transformed into a finite parameter optimization problem, also called Non-Linear Program (NLP). Different discretization methods (single shooting, multiple shooting, collocation) were introduced. The resulting NLP was solved with gradient based approaches. Additionally, implementation aspects such as gradient calculation and scaling were discussed.

Chapter 3 introduced discrete controls into the OCP, which can only take values from a fixed set. These problems are also called Mixed-Integer Optimal Control Problem (MIOCP). Using direct methods, the problem was transfered to a Mixed-Integer Non-Linear Program (MINLP). Discrete controls were considered using Outer Convexification (OC). Weights were introduced for every discrete choice at every discretized point in time making the switching structure subject to optimization. Discrete control dependent constraints were reformulated using Vanishing Constraints. A relaxation method suitable for the OC method was introduced. Thus, the discrete optimal control problem was continuously reformulated. Binary feasibility for the discrete control weights and a minimization of discrete switches were enabled by a novel penalty cost approach. A two stage solution strategy was presented. In the first stage, an optimization without switching cost was carried out to find a suitable initial guess for the switching structure. Afterwards, the second stage enabled the switching cost approach to obtain a discrete value feasible switching structure. In between both steps, the discrete control weights were augmented by pre-eliminating high frequent spikes in the solution. Thus, the stability of the overall process was improved. Multiple independent discrete controls were considered by calculating all feasible permutations. The weights of individual discrete controls were calculated with mapping matrices.

In case realistic dynamic models with discrete controls were considered, large scale

high fidelity OCPs were created. Chapter 4 introduced the *FALCON.m* optimal control framework that is able to calculate the analytic first and second order derivatives (Jacobian and Hessian) of such problems. The problem definition from the user side was explained. It was followed by a description of the derivative generation toolchain that consists of two parts, namely the differentiation of user supplied functions (dynamic models, constraints, cost functions) and the construction of the overall problem derivatives (Jacobian, Hessian). User functions were differentiated using the subsystem derivative builder. This *FALCON.m* sub-toolbox is able to differentiate high fidelity user functions by applying source code transformation. The user functions were divided into more manageable subsystems which were differentiated using the *MATLAB* Symbolic Math toolbox. The chain rule was automatically applied to calculate the overall derivatives. Thus, analytic derivatives for complicated high fidelity models could be created. The interface and communication of the differentiated user functions with the *FALCON.m* optimal control framework were explained. Jacobian and Hessian information was used to create the overall problem derivatives. User function derivative values were directly sorted to the correct position of the sparse row column value vector representation. Thus, high memory consumption was avoided and the problem derivative generation was reduced to a linear index mapping. The algorithms for the Jacobian and the Hessian calculation were explained. Discrete controls in *FALCON.m* were considered through a user friendly toolbox extension. Two selected problems were solved automatically with the methods used in this thesis.

The minimal lap time for a car through a race circuit was determined in chapter 5. As discrete control, the gear transmission was introduced. The Nürburgring race track was modeled as a cubic spline for the center line and the width. All four car wheel positions were considered via constraints. The engine rotation speed was limited as well. First, a few single optimizations were carried out to explain the solution. The problem was solved with and without the spike removal algorithm between the two optimization stages. The latter yields a better lap time. Additionally, gas and brake pedals were introduced as discrete controls. Similar results compared to the continuous case were obtained. Second, the stability of the switching cost formulation was evaluated. It produced consistent results over a wide range of penalty scalings as well as the number of discretization points. It was shown that the intermediate augmentation of the discrete control switching structure improved the consistency between the solutions. Additionally, the minimal lap time problem was solved for different initial guesses for the discrete gear choice.

In chapter 6, aircraft approach trajectories were optimized under the consideration of discrete high lift and landing gear changes. The 3DOF BADA4 aircraft model which supplies discrete control dependent aerodynamics as well as flight envelope limits was used. Additionally, approach relevant constraints were stated and considered. The fuel minimal approach trajectories were calculated for the runway 26R of Munich airport. A single optimization was performed to explain the structure of the solution. Afterwards, parameter studies for initial aircraft mass, initial approach speed, initial approach altitude, wind speed, and wind direction were carried out to determine their influence on the gear switching structure. The obtained results showed a high consistency. The influence of the initial aircraft mass on the discrete control switching structure was particularly prominent.

The methods presented in this thesis were successfully applied to two different applications. It was shown that no initial knowledge on the optimal switching structure

is required. Multiple independent discrete controls can be handled as well. The number of switches was efficiently reduced and discrete value feasibility was ensured using a novel switching cost approach. Its formulation is application independent and can thus be used in other optimal control problems with discrete controls. The resulting large optimal control problems were solved with the *FALCON.m* software. It is able to calculate analytic derivatives for high fidelity models and implements an efficient way to calculate derivatives of large problems. *FALCON.m* is available as a free software from `www.falcon-m.com`.

7.2 Outlook

In this section, potential future work for the consideration of discrete controls and improvements on the *FALCON.m* optimal control toolbox are discussed.

Discrete Controls

The applications of this thesis showed that the used approaches are able to find realistic and consistent solutions for large problem formulations as well as realistic dynamic models. Discrete control optimization problems can usually be solved in two stages. Particularly, augmenting the intermediate discrete control switching structure seems to be a promising approach to achieve consistency in the optimization results. In this thesis, a spike removal augmentation was applied. Additionally, a smoothing of the results as well as rounding approaches may be used.

Dependent on the application, the second optimization stage with switching cost may require a high number of iterations. Especially in the car optimization example many changes were required in the second stage. The high number of iterations lead to long CPU times. Currently, the penalty cost function is formulated between multiple time steps but for each discrete control weight individually. Performance improvements may be achieved by introducing an additional penalty term between all discrete controls weights at each discretized point.

Within interior point methods, a barrier parameter is used to account for inequality constraints. During optimization, it is slowly driven to zero. The slack variables of the vanishing constraints and the penalty scaling of the switching cost may become a function of this parameter. Once the optimization algorithm approaches the minimum, the discrete controls and minimization of switches are gradually enforced. Thus, it may be possible to solve the discrete control optimization problem in a single stage. In case the IPOPT solver is used, a custom *MATLAB* interface would be required as currently only the C++ version has access to the barrier parameter.

Overall, the switching cost approach introduced in this thesis is able to find low frequent switching structures for the discrete controls. Dependent on the application, the required CPU time drops below the time range considered in the OCP. Therefore, the method may also be applicable for close to real time applications.

Optimal Control Toolbox

Although the *FALCON.m* toolbox is able to calculate the cost function, constraints, and their derivatives quickly, certain aspects may be improved in future versions. These enhancements are discussed in the following.

The subsystem derivative builder creates differentiated user functions that return the Jacobian and Hessian as dense 3D matrices. Especially, Hessian matrices may contain many zero elements and thus lead to unnecessary memory consumption. Therefore, it is proposed that differentiated user functions return the potentially non-zero elements only. The struct interface must be adapted accordingly.

The chain rule within the subsystem derivative generation uses dense matrix multiplications as well. Similarly to the output derivatives, these may contain many zero entries. Currently, in the Hessian chain rule, multiplications with full zero matrices (e.g. input Hessians) are not written to code. This approach could be expanded to calculate the chain rules only for the non-zero entries. Thus, only multiplications that will lead to actual non-zero values could be carried out. Initial tests show that the derivative evaluation speed can be improved notably. However, the resulting *MATLAB* code becomes significantly longer and increases code generation and compilation time. Therefore, further tests have to be made before an implementation becomes reasonable.

For most OCP in this thesis, the issues discussed above currently do not impact the performance significantly. However, they may become relevant in case dynamic models with a larger number of states and controls (e.g. higher than 50) are considered.

Currently, most of the OCP in *FALCON.m* are solved with the IPOPT optimizer. The *MATLAB* interface of the solver expects *MATLAB* sparse matrices during runtime. *MATLAB* sparse matrices sort the non-zero elements by row indices and then by column indices. From the C++ interface, it can be determined that IPOPT expects the elements to be sorted the other way round. Additionally, on the IPOPT website it is stated that the sparse matrix implementation in *MATLAB* is difficult to handle. Non-zero elements that are zero during runtime are potentially removed from the internal storage by *MATLAB*. The structure of the vectors therefore changes. Since *FALCON.m* stores the sparse matrices in vectors of fixed structure and length, a custom IPOPT interface may lead to better runtimes.

In the preprocessor step that discretizes and builds the OCP in *FALCON.m*, the index sets for the direct sparsity sorting algorithm are created. During runtime, most of the derivative calculations require linear indexing to copy data between different array elements. Therefore, the information calculated by *FALCON.m* can be used to export C/C++ code that can run within other programs or embedded platforms.

Applications

In the car optimization, a relative simple model was used. More realistic trajectories may be obtained with a two track model implementing full body motion and advanced tire dynamics. In case a realistic model is available, a benchmark against the real race time could be performed. Additionally to the selection of the gear, track specific optimal transmission ratios might be determined by the presented methods. With increasing significance of autonomous driving, the minimal lap time trajectories could

be used as references for a controller in autonomous racing.

In the approach optimization problem, further optimizations can be made for continuous descent approaches from the top of descent as well as for full flight optimization. With increased significance of unmanned aerial vehicles, discrete flap settings may also be used in flight. Additionally, the presented methods could be applied to vertical takeoff and landing systems that switch to a traditional configuration during flight.

Apart from the presented applications, discrete controls may be used for optimizations in other fields. Discrete changes in air traffic control may be modeled as a discrete control. Discrete controls also appear in chemical processes, pipe networks, power distribution, and logistics. Additionally, an adaptation of the switching cost formulation may be used to solve combinatorial problems. An example is the gate scheduling at an airport.

Bibliography

[1] H. G. Bock and Longman R., "Computation of optimal controls on disjoint control sets for minimum energy subway operation," in *American Astronomical Society. Symposium on Engineering Science and Mechanics*, pp. 949–972, 1982.

[2] N. Bedrossian, S. Bhatt, M. Lammers, L. Nguyen, and Y. Zhang, "First ever flight demonstration of zero propellant maneuver(tm) attitute control concept," in *AIAA Guidance, Navigation and Control Conference and Exhibit*, 2007.

[3] N. Bedrossian and S. Bhatt, "Space station zero-propellant maneuver guidance trajectories compared to eigenaxis," in *American Control Conference (ACC '08)*, pp. 4833–4838, 2008.

[4] J. T. Olympio, "Optimal control problem for low-thrust multiple asteroid tour missions," *Journal of Guidance, Control, and Dynamics*, vol. 34, no. 6, pp. 1709–1720, 2011.

[5] F. Fisch, J. Lenz, F. Holzapfel, and G. Sachs, "Trajectory optimization applied to air races," in *AIAA Atmospheric Flight Mechanics Conference*, Guidance, Navigation, and Control and Co-located Conferences, American Institute of Aeronautics and Astronautics, 2009.

[6] C. Kirches, S. Sager, H. G. Bock, and J. P. Schlöder, "Time-optimal control of automobile test drives with gear shifts," *Optimal Control Applications and Methods*, vol. 31, no. 2, pp. 137–153, 2010.

[7] M. Gerdts, "Solving mixed-integer optimal control problems by branch&bound: a case study from automobile test-driving with gear shift," *Optimal Control Applications and Methods*, vol. 26, no. 1, pp. 1–18, 2005.

[8] E. Hellström, M. Ivarsson, J. Åslund, and L. Nielsen, "Look-ahead control for heavy trucks to minimize trip time and fuel consumption," *Control Engineering Practice*, vol. 17, no. 2, pp. 245–254, 2009.

[9] Delgado San Martín, Juan Antonio, M. N. Cruz Bournazou, P. Neubauer, and T. Barz, "Mixed integer optimal control of an intermittently aerated sequencing batch reactor for wastewater treatment," *Computers & Chemical Engineering*, vol. 71, pp. 298–306, 2014.

[10] D. Lebiedz, S. Sager, H. G. Bock, and P. Lebiedz, "Annihilation of limit-cycle oscillations by identification of critical perturbing stimuli via mixed-integer optimal control," *Physical review letters*, vol. 95, no. 10, p. 108303, 2005.

[11] S. Sager, *Numerical methods for mixed–integer optimal control problems*. PhD thesis, University of Heidelberg, Heidelberg, 2005.

[12] I. E. Grossmann, P. A. Aguirre, and M. Barttfeld, "Optimal synthesis of complex distillation columns using rigorous models," *Computers & Chemical Engineering*, vol. 29, no. 6, pp. 1203–1215, 2005.

[13] J. Oldenburg, W. Marquardt, D. Heinz, and D. B. Leineweber, "Mixed-logic dynamic optimization applied to batch distillation process design," *AIChE Journal*, vol. 49, no. 11, pp. 2900–2917, 2003.

[14] S. Sager, M. Diehl, G. Singh, A. Küpper, and S. Engell, "Determining smb superstructures by mixed-integer optimal control," in *Operations Research Proceedings 2006* (K.-H. Waldmann and U. M. Stocker, eds.), vol. 2006 of *Operations Research Proceedings*, pp. 37–42, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.

[15] M. Soler, M. Kamgarpour, J. Lloret, and J. Lygeros, "A hybrid optimal control approach to fuel-efficient aircraft conflict avoidance," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–13, 2016.

[16] M. Rieck, M. Bittner, and F. Holzapfel, "Discrete control dependent constraints in multiple shooting optimal control problems," in *AIAA Guidance, Navigation, and Control (GNC) Conference*, 2013.

[17] F. Fisch, J. Lenz, and F. Holzapfel, "Aircraft configuration settings within the optimization of approach trajectories," in *AIAA Guidance, Navigation and Control Conference*, 2012.

[18] M. Rieck, M. Richter, M. Bittner, and F. Holzapfel, "Optimal trajectories for rpas with discrete controls and discrete constraints," in *Aerospace Electronics and Remote Sensing Technology (ICARES), 2014 IEEE International*, pp. 34–38, IEEE, 2014.

[19] E. Gallo, F. Navarro, A. Nuic, and M. Iagaru, "Advanced aircraft performance modeling for atm: Bada 4.0 results," in *2006 ieee/aiaa 25TH Digital Avionics Systems Conference*, pp. 1–12, 2006.

[20] M. Richter, M. Hochstrasser, M. Bittner, L. Walter, and F. Holzapfel, "Application of minlp techniques to conflict resolution of multiple aircraft," in *AIAA GNC and Co- located Conferences*, 2014.

[21] C. Kirches, H. G. Bock, J. P. Schloder, and S. Sager, "Mixed-integer nmpc for predictive cruise control of heavy-duty trucks," in *Control Conference (ECC), 2013 European*, pp. 4118–4123, 2013.

[22] A. T. FULLER, "Study of an optimum non-linear control system†," *Journal of Electronics and Control*, vol. 15, no. 1, pp. 63–71, 1963.

[23] C. Kirches, *Fast Numerical Methods for Mixed-Integer Nonlinear Model-Predictive Control*. PhD thesis, University of Heidelberg, Heidelberg, 2010.

[24] J. T. Betts, *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. Society for Industrial and Applied Mathematics, 2010.

[25] D. E. Kirk, *Optimal control theory; An introduction*. Prentice-Hall networks series, Englewood Cliffs, N.J: Prentice-Hall, 1970.

[26] M. Gerdts, *Optimal Control of ODEs and DAEs*. De Gruyter textbook, Berlin: De Gruyter, 2012.

[27] L. T. Biegler, "An overview of simultaneous strategies for dynamic optimization," *Chemical Engineering and Processing: Process Intensification*, vol. 46, no. 11, pp. 1043–1053, 2007.

[28] S. Kameswaran and L. T. Biegler, "Simultaneous dynamic optimization strategies: Recent advances and challenges," *Computers & Chemical Engineering*, vol. 30, no. 10-12, pp. 1560–1575, 2006.

[29] H. G. Bock, K. J. Plitt, and Sonderforschungsbereich Approximation und Optimierung in einer Anwendungsbezogenen Mathematik, *A multiple shooting algorithm for direct solution of optimal control problems*. Preprint / Sonderforschungsbereich 72, Approximation und Optimierung, Universität Bonn, Sonderforschungsbereich 72, Approximation u. Optimierung, Univ. Bonn, 1983.

[30] H. G. Bock and K. J. Plitt, "A multiple shooting algorithm for direct solution of optimal control problems," in *9th IFAC World Congress Budapest*, pp. 243–247, Pergamon Press, 1984.

[31] D. B. Leineweber, I. Bauer, H. G. Bock, and J. P. Schlöder, "An efficient multiple shooting based reduced sqp strategy for large-scale dynamic process optimization. part 1: Theoretical aspects," *Computers & Chemical Engineering*, vol. 27, no. 2, pp. 157–166, 2003.

[32] C. R. HARGRAVES and S. W. PARIS, "Direct trajectory optimization using nonlinear programming and collocation," *Journal of Guidance, Control, and Dynamics*, vol. 10, no. 4, pp. 338–342, 1987.

[33] M. Rieck, M. Richter, and F. Holzapfel, "Generation of initial guesses for optimal control problems with mixed integer dependent constraints," in *ICAS 29th International Conference*, 2014.

[34] Ruiyan Zhao and Shurong Li, "Time-optimal control based on hybrid genetic algorithm," in *8th World Congress on Intelligent Control and Automation (WCICA 2010)*, pp. 4766–4769, 2010.

[35] S.-P. Han, "Superlinearly convergent variable metric algorithms for general nonlinear programming problems," *Mathematical Programming*, vol. 11, no. 1, pp. 263–282, 1976.

[36] M. J. D. Powell, "Algorithms for nonlinear constraints that use lagrangian functions," *Mathematical Programming*, vol. 14, no. 1, pp. 224–248, 1978.

[37] J. Gondzio, "Multiple centrality corrections in a primal-dual method for linear programming," *Computational Optimization and Applications*, vol. 6, no. 2, pp. 137–156, 1996.

[38] A. Wächter, *An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2002.

[39] S. Mehrotra, "On the implementation of a primal-dual interior point method," *SIAM Journal on Optimization*, vol. 2, no. 4, pp. 575–601, 1992.

[40] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, 2006.

[41] P. E. Gill, W. Murray, and M. A. Saunders, "Snopt: An sqp algorithm for large-scale constrained optimization," *SIAM Review*, vol. 47, no. 1, pp. 99–131, 2005.

[42] T. Nikolayzik, C. Büskens, and M. Gerdts, "Nonlinear large-scale optimization with worhp," in *13th AIAA/ISSMO Multidisciplinary Analysis Optimization Conference*, 2010.

[43] P. I. Barton and C. K. Lee, "Design of process operations using hybrid dynamic optimization," *Computers & Chemical Engineering*, vol. 28, no. 6-7, pp. 955–969, 2004.

[44] U. Brandt-Pollmann, *Numerical solution of optimal control problems with implicitly defined discontinuities with applications in engineering*. PhD thesis, University of Heidelberg, Heidelberg, 2004.

[45] P. Howlett, "Optimal strategies for the control of a train," *Automatica*, vol. 32, no. 4, pp. 519–532, 1996.

[46] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP - completeness*. A series of books in the mathematical sciences, New York: W.H. Freeman and Co, 24th print ed., 2003.

[47] W. R. Esposito, "Deterministic global optimization in nonlinear optimal control problems," *Journal of Global Optimization*, vol. 17, no. 1/4, pp. 97–126, 2000.

[48] I. Papamichail and C. S. Adjiman, "Global optimization of dynamic systems," *Computers & Chemical Engineering*, vol. 28, no. 3, pp. 403–415, 2004.

[49] R. Li, K. L. Teo, K. H. Wong, and G. R. Duan, "Control parameterization enhancing transform for optimal control of switched systems," *Mathematical and Computer Modelling*, vol. 43, no. 11-12, pp. 1393–1403, 2006.

[50] A. Richards and J. P. How, "Model predictive control of vehicle maneuvers with guaranteed completion time and robust feasibility," in *2003 American Control Conference*, pp. 4034–4040, 4-6 June 2003.

[51] M. Gerdts, "A variable time transformation method for mixed-integer optimal control problems," *Optimal Control Applications and Methods*, vol. 27, no. 3, pp. 169–182, 2006.

[52] K. Palagachev, M. Rieck, and M. Gerdts, "A mixed-integer optimal control approach for aircraft landing model," in *Models & Technologies for Intelligent Transportation Systems*, Technische Universität Dresden, 2013.

[53] H. Lee, K. L. Teo, and X. Q. Cai, "An optimal control approach to nonlinear mixed integer programming problems," *Computers & Mathematics with Applications*, vol. 36, no. 3, pp. 87–105, 1998.

[54] H. Lee, K. L. Teo, V. Rehbock, and L. S. Jennings, "Control parametrization enhancing technique for optimal discrete-valued control problems," *Automatica*, vol. 35, no. 8, pp. 1401–1407, 1999.

[55] S. Sager, "Reformulations and algorithms for the optimization of switching decisions in nonlinear optimal control," *Journal of Process Control*, vol. 19, no. 8, pp. 1238–1247, 2009.

[56] F. Logist, S. Sager, C. Kirches, and J. F. van Impe, "Efficient multiple objective optimal control of dynamic systems with integer controls," *Journal of Process Control*, vol. 20, no. 7, pp. 810–822, 2010.

[57] F. M. Hante and S. Sager, "Relaxation methods for mixed-integer optimal control of partial differential equations," *Computational Optimization and Applications*, vol. 55, no. 1, pp. 197–225, 2013.

[58] W. Achtziger and C. Kanzow, "Mathematical programs with vanishing constraints: optimality conditions and constraint qualifications," *Mathematical Programming*, vol. 114, no. 1, pp. 69–99, 2007.

[59] T. Hoheisel, *Mathematical Programs with Vanishing Constraints*. PhD thesis, University of Würzburg, Würzburg, 2009.

[60] T. Hoheisel and C. Kanzow, "Stationary conditions for mathematical programs with vanishing constraints using weak constraint qualifications," *Journal of Mathematical Analysis and Applications*, vol. 337, no. 1, pp. 292–310, 2008.

[61] T. Hoheisel and C. Kanzow, "First-and second-order optimality conditions for mathematical programs with vanishing constraints," *Applications of Mathematics*, vol. 52, no. 6, pp. 495–514, 2007.

[62] A. F. Izmailov and M. V. Solodov, "Mathematical programs with vanishing constraints: Optimality conditions, sensitivity, and a relaxation method," *Journal of Optimization Theory and Applications*, vol. 142, no. 3, pp. 501–532, 2009.

[63] W. Achtziger, T. Hoheisel, and C. Kanzow, "A smoothing-regularization approach to mathematical programs with vanishing constraints," *Computational Optimization and Applications*, vol. 55, no. 3, pp. 733–767, 2013.

[64] T. Hoheisel, C. Kanzow, and J. V. Outrata, "Exact penalty results for mathematical programs with vanishing constraints," *Nonlinear Analysis: Theory, Methods & Applications*, vol. 72, no. 5, pp. 2514–2526, 2010.

[65] K. Palagachev and M. Gerdts, "Mathematical programs with blocks of vanishing constraints arising in discretized mixed-integer optimal control problems," *Set-Valued and Variational Analysis*, vol. 23, no. 1, pp. 149–167, 2015.

[66] T. Hoheisel and C. Kanzow, "On the abadie and guignard constraint qualifications for mathematical programmes with vanishing constraints," *Optimization*, vol. 58, no. 4, pp. 431–448, 2009.

[67] P. Pardalos, R. Horst, J. Outrata, M. Kočvara, and J. Zowe, *Nonsmooth Approach to Optimization Problems with Equilibrium Constraints*, vol. 28. Boston, MA: Springer US, 1998.

[68] H.-L. Li, "An approximate method for local optima for nonlinear mixed integer programming problems," *Computers & Operations Research*, vol. 19, no. 5, pp. 435–444, 1992.

[69] H.-L. Li and C.-T. Chou, "A global approach for nonlinear mixed discrete programming in design optimization," *Engineering Optimization*, vol. 22, no. 2, pp. 109–122, 1993.

[70] J. Martins, P. Sturdza, and J. Alonso, "The complex-step derivative approximation," *ACM Transactions on Mathematical Software*, vol. 29, pp. 245–262, 2003.

[71] R. J. Hogan, "Fast reverse-mode automatic differentiation using expression templates in c++," *ACM Trans. Math. Softw.*, vol. 40, no. 4, pp. 26:1–26:16, 2014.

[72] A. Walther, A. Griewank, and O. Vogel, "Adol-c: Automatic differentiation using operator overloading in c++," *PAMM*, vol. 2, no. 1, pp. 41–44, 2003.

[73] Mathworks, "Symbolic math toolbox (tm) user's guide," 2016.

[74] J. Lenz, *Optimisation of Periodic Flight Trajectories*. PhD thesis, Technische Universität München, München, 2015.

[75] M. J. Weinstein, M. A. Patterson, and A. V. Rao, "Utilizing the algorithmic differentiation package adigator for solving optimal control problems using direct collocation," in *AIAA Guidance, Navigation, and Control Conference*.

[76] C. H. Bischof, H. M. Bücker, B. Lang, A. Rasch, and A. Vehreschild, "Combining source transformation and operator overloading techniques to compute derivatives for matlab programs," in *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, (Los Alamitos, CA, USA), pp. 65–72, IEEE Computer Society, 2002.

[77] F. Fisch, *Development of a Framework for the Solution of High-Fidelity Trajectory Optimization Problems and Bilevel Optimal Control Problems*. PhD thesis, Technische Universität München, München, 2011.

[78] I. M. Ross, "A beginner's guide to dido: A matlab application package for solving optimal control problems," 2007.

[79] M. A. Patterson and A. V. Rao, "Gpops-ii," *ACM Transactions on Mathematical Software*, vol. 41, no. 1, pp. 1–37, 2014.

[80] M. Rieck, M. Bittner, B. Grüter, and J. Diepolder, "Falcon.m," 2016.

[81] C. Göttlicher, M. Gnoth, M. Bittner, and F. Holzapfel, "Aircraft parameter estimation using optimal control methods," in *AIAA Guidance, Navigation, and Control Conference*, 2016.

[82] J. Z. Ben-Asher, *Optimal control theory with aerospace applications*. AIAA education series, Reston, VA: American Institute of Aeronautics and Astronautics, 2010.

[83] A. E. Bryson and Y.-C. Ho, *Applied optimal control*. Hemisphere, rev. print ed., 1975.

[84] D. G. Zill and P. D. Shanahan, *Complex analysis: A first course with applications*. The Jones & Barlett learning series in mathematics, Burlington, MA: Jones & Bartlett Learning, third edition ed., 2015.

[85] M. Gerdts, "Optimierung," 2008.

[86] D. W. Zingg, M. Nemec, and T. H. Pulliam, "A comparative evaluation of genetic and gradient-based algorithms applied to aerodynamic optimization," *Revue européenne de mécanique numérique*, vol. 17, no. 1-2, pp. 103–126, 2008.

[87] C. G. BROYDEN, "The convergence of a class of double-rank minimization algorithms 1. general considerations," *IMA Journal of Applied Mathematics*, vol. 6, no. 1, pp. 76–90, 1970.

[88] R. Fletcher, "A new approach to variable metric algorithms," *The Computer Journal*, vol. 13, no. 3, pp. 317–322, 1970.

[89] D. Goldfarb, "A family of variable-metric methods derived by variational means," *Mathematics of Computation*, vol. 24, no. 109, p. 23, 1970.

[90] D. F. Shanno, "Conditioning of quasi-newton methods for function minimization," *Mathematics of Computation*, vol. 24, no. 111, p. 647, 1970.

[91] W. C. Davidon, "Variable metric method for minimization," *SIAM Journal on Optimization*, vol. 1, no. 1, pp. 1–17, 1991.

[92] A. R. Conn, N. I. M. Gould, and P. L. Toint, "Convergence of quasi-newton matrices generated by the symmetric rank one update," *Mathematical Programming*, vol. 50, no. 1-3, pp. 177–195, 1991.

[93] P. Wolfe, "Convergence conditions for ascent methods," *SIAM Review*, vol. 11, no. 2, pp. 226–235, 1969.

[94] A. A. Goldstein, "On steepest descent," *Journal of the Society for Industrial and Applied Mathematics Series A Control*, vol. 3, no. 1, pp. 147–151, 1965.

[95] W. Alt, *Nichtlineare Optimierung: Eine Einführung in Theorie, Verfahren und Anwendungen*. Vieweg Studium : Aufbaukurs Mathematik, Braunschweig and Wiesbaden: Vieweg, 1. aufl. ed., 2002.

[96] P. Spellucci, *Numerische Verfahren der nichtlinearen Optimierung*. Basel: Birkhäuser Basel, 1993.

[97] M. Gerdts, S. Karrenberg, B. Müller-Beßler, and G. Stock, "Generating locally optimal trajectories for an automatically driven car," *Optimization and Engineering*, vol. 10, no. 4, pp. 439–463, 2008.

[98] P. E. Gill and E. Wong, "Sequential quadratic programming methods," in *Mixed Integer Nonlinear Programming* (J. Lee and S. Leyffer, eds.), vol. 154 of *The IMA Volumes in Mathematics and its Applications*, pp. 147–224, New York, NY: Springer New York, 2012.

[99] C. Büskens, *Optimierungmethoden und Sensitivitätsarialyse ftir optimale Steuerprozesse mit Steuer- und Zustands-Beschränkungen*. PhD thesis, Westfälische Wilhelms-Universität, Münster, 1998.

[100] M. Gerdts, "Optimale steuerung," 2009.

[101] S. Subchan and R. Żbikowski, *Computational optimal control: Tools and practice*. Chichester, U.K.: J. Wiley, 2009.

[102] P. Albrecht, "The runge-kutta theory in a nutshell," *SIAM Journal on Numerical Analysis*, vol. 33, no. 5, pp. 1712–1735, 1996.

[103] J. C. Butcher, "On the attainable order of runge-kutta methods," *Mathematics of Computation*, vol. 19, no. 91, p. 408, 1965.

[104] T. H. TSANG, D. M. HIMMELBLAU, and T. F. EDGAR, "Optimal control via collocation and non-linear programming," *International Journal of Control*, vol. 21, no. 5, pp. 763–768, 1975.

[105] L. T. Biegler, "Solution of dynamic optimization problems by successive quadratic programming and orthogonal collocation," *Computers & Chemical Engineering*, vol. 8, no. 3-4, pp. 243–247, 1984.

[106] C. Y. Kaya and J. L. Noakes, "Computations and time-optimal controls," *Optimal Control Applications and Methods*, vol. 17, no. 3, pp. 171–185, 1996.

[107] C. Y. Kaya and J. L. Noakes, "Computational method for time-optimal switching control," *Journal of Optimization Theory and Applications*, vol. 117, no. 1, pp. 69–92, 2003.

[108] V. Rehbock and L. Caccetta, "Two defence applications involving discrete valued optimal control," *ANZIAM J*, vol. 44, no. E, pp. E33–E54, 2002.

[109] M. N. Jung, C. Kirches, and S. Sager, "On perspective functions and vanishing constraints in mixed-integer nonlinear optimal control," in *Facets of Combinatorial Optimization* (M. Jünger and G. Reinelt, eds.), pp. 387–417, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[110] C. Kirches and F. Lenders, "Approximation properties and tight bounds for constrained mixed-integer optimal control," *Mathematical Programming*, April 2016.

[111] M. Rieck, G. P. Falconí, M. Gerdts, and F. Holzapfel, "Periodic full circuit race line optimization under consideration of a dynamic model with gear changes," in *IEEE Multi-Conference on Systems and Control 2016*, 2016.

[112] M. Rieck, M. Bittner, B. Grüter, and F. Holzapfel, "Generation of dynamic models with automatically generated derivatives for atm optimal control in matlab," in *ENRI Int. Workshop on ATM/CNS (EIWAC)*, 2015.

[113] M. Bückner, P. Hovland, C. Wente, and H. Bach, "Community portal for automatic differentiation."

[114] W. H. Press, *Numerical recipes: The art of scientific computing*. Cambridge, UK and New York: Cambridge University Press, 3rd ed ed., 2007.

[115] J. R. Magnus and H. Neudecker, "Matrix differential calculus with applications to simple, hadamard, and kronecker products," *Journal of Mathematical Psychology*, vol. 29, no. 4, pp. 474–492, 1985.

[116] D. Limebeer, G. Perantoni, and A. V. Rao, "Optimal control of formula one car energy recovery systems," *International Journal of Control*, pp. 1–16, 2014.

[117] D. Tavernini, M. Massaro, E. Velenis, D. I. Katzourakis, and R. Lot, "Minimum time cornering: the effect of road surface and car transmission layout," *Vehicle System Dynamics*, vol. 51, no. 10, pp. 1533–1547, 2013.

[118] H. B. Pacejka, *Tyre and vehicle dynamics*. Oxford: Butterworth-Heinemann, 2nd ed. ed., 2006.

[119] F. Holzapfel, "Flugsystemdynamik 1," 2015.

[120] National Imagery and Mapping Agency, "World geodetic system 1984."

[121] B. L. Stevens and F. L. Lewis, *Aircraft control and simulation*. Hoboken, N.J.: J. Wiley, 2nd ed. ed., 2003.

[122] R. Brockhaus, W. Alles, and R. Luckner, *Flugregelung*. Berlin: Springer Berlin, 3., neu bearbeitete aufl. ed., 2010.

[123] I. Newton and J. Machin, *The Mathematical Principles of Natural Philosophy*, vol. 1. 1686.

[124] A. R. Plastino and J. C. Muzzio, "On the use and abuse of newton's second law for variable mass problems," *Celestial Mechanics and Dynamical Astronomy*, vol. 53, no. 3, pp. 227–232, 1992.

[125] D. Etling, *Theoretische Meteorologie: Eine Einführung*. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 3. erw. und aktualisierte aufl. ed., 2008.

[126] W. Kühr, *Grundlagen der Flugwetterkunde*, vol. Bd. 2 of *Der Privatflugzeugführer / Wolfgang Kühr*. Bergisch Gladbach: Schiffmann, neue, verb. aufl. ed., 1989.

[127] *Manual of the ICAO standard atmosphere: Extended to 80 kilometres (262 500 feet) = Manuel de l'atmosphère type OACI : élargie jusqu'à 80 kilomètres (262 500 pieds) = Manual de la átmosfera tipo de la OACI : ampliada hasta 80 kilómetros (262 500 pies)*. Montreal: ICAO, 3rd ed. ed., 2002.

[128] U.S. National Archives and Records Administration, "Code of federal regulations: Title 14. aeronautics and space: Part 33 airworthiness standards: aircraft engines," 1971.

[129] L. Wynnyk, C. R. Lunsford, J. A. Tittsworth, and S. Pressley, "Development of approach and departure aircraft speed profiles," *Journal of Aircraft*, pp. 1–10, 2016.

[130] Airbus, "Aircraft energy management during approach," 2005.

[131] Airbus, "Safety first: The airbus safety magazine," 2015.

[132] E. Parks, "Training notes for a319/320/321," 2016.

[133] N. Vitu-Barbier, "New vapp calculation process," 2007.

[134] Airbus, "A318/a319/a320/a321 performance training manual," 2005.

[135] SmartCockpit, "Airbus a320: ils approach landing."

[136] R. W. Sinnott, "Virtues of the haversine," *Sky and Telescope 68*, p. 159, 1984.

[137] Jeppesen, "Airport information eddm," 2008.

[138] J. Link, "Opennav website."

[139] S. Huler, *Defining the wind: The Beaufort scale, and how a nineteenth-century admiral turned science into poetry*. New York: Crown Publishers, 1st ed. ed., 2004.

[140] S. Kerschhofer, "Alpenwetterfibel für hänge- und paragleider."

[141] H. Prautzsch, W. Boehm, and M. Paluszny, *Bézier and B-spline techniques*. Mathematics and visualization, Berlin and New York: Springer, 2002.

Appendix A

Continuous and Mixed-Integer Optimal Control

This appendix chapter contains extensions to chapters 2 and 3.

A.1 Sensitivity Equation Example

Here, an example for the sensitivity equation

$$\dot{S}(\tau) = \vec{f}(\vec{x}, \vec{u}) \cdot S_{t_f} + t_f \cdot J_x \cdot S(\tau) + t_f \cdot J_u \cdot S_u(t\tau), \qquad S(\tau_0) = S_0. \tag{A.1}$$

introduced in section 2.5 is presented. For the example, it is assumed that there are

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \qquad \vec{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, \tag{A.2}$$

three states ($n_x = 3$), two controls ($n_u = 2$), and the time interval $[0, t_f]$ contains 6 discretized points ($n_h = 5$). Assuming the single shooting discretization methods gives the following optimization vector

$$\vec{Z} = [t_f, x_{1,0}, x_{2,0}, x_{3,0}, u_{1,0}, u_{2,0}, u_{1,1}, u_{2,1}, u_{1,2}, u_{2,2}, u_{1,3}, u_{2,3}, u_{1,4}, u_{2,4}, u_{1,5}, u_{2,5}]^T \tag{A.3}$$

where the first index is the determines the state or control and the second the time step. Overall there are $n_{\vec{Z}} = 16$ optimization variables. The variables in equation $(\mathrm{A}.1)$ have the following dimensions

$$J_x \in \mathbb{R}^{n_x \times n_x}, \quad J_u \in \mathbb{R}^{n_x \times n_u}, \qquad S_{t_f} \in \mathbb{R}^{1 \times n_{\vec{Z}}}, \quad S_0 \in \mathbb{R}^{n_x \times n_{\vec{Z}}}, \quad S_u \in \mathbb{R}^{n_u \times n_{\vec{Z}}}, \tag{A.4}$$

and the final time t_f is scalar. As already mentioned

$$S_{t_f} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{A.5}$$

is a constant row vector which has a single entry of one and zero otherwise. The initial state sensitivity

$$S_0 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{A.6}$$

is the identity matrix at the point where the initial state is discretized in the $\vec{Z}$ vector. Finally, the control sensitivity matrix

$$S_{u,0} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{A.7}$$

$$S_{u,1} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{A.8}$$

$$S_{u,2} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{A.9}$$

$$S_{u,3} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{A.10}$$

$$S_{u,4} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{A.11}$$

$$S_{u,5} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.12}$$

is time dependent. In (A.7) through (A.12), the matrix is given at the discretization points where the identity matrix shifts to the control of the current discretized time. In between the same interpolation methods as for the controls $\vec{u}$ is used.

A.2 Hyperbolic Tangent Discrete Constraint

The hyperbolic tangent approach is a viable choice if the switching structure of the discrete controls is known. In case constraints are dependent on the discrete choice, the constraints can be formulated in a similar manner as the discrete controls itself.

Assume the box constraint

$$g_{lb}\left(\vec{v}(t)\right) \leq g\left(\vec{x}(t), \vec{u}(t), t, \vec{v}(t)\right) \leq g_{ub}\left(\vec{v}(t)\right) \tag{A.13}$$

where the lower and upper bound is dependent on the discrete choice $n_v(t)$. Due to the fact that the constraint bounds in numeric optimization algorithms must not change during optimization, the constraint is normalized

$$0 \leq \frac{g - g_{lb}}{g_{ub} - g_{lb}} \leq 1. \tag{A.14}$$

Both the lower and upper bound are transformed to a function w.r.t. time

$$g_{lb}(t) = g_{lb,0} + \sum_{k=1}^{n-1} \left(g_{lb,k} - g_{lb,k-1}\right) \cdot \left[\tanh\left(a \cdot (t - t_k)\right) + 1\right]/2 \tag{A.15}$$

$$g_{ub}(t) = g_{ub,0} + \sum_{k=1}^{n-1} \left(g_{ub,k} - g_{ub,k-1}\right) \cdot \left[\tanh\left(a \cdot (t - t_k)\right) + 1\right]/2 \tag{A.16}$$

with the number of steps n, switching times t_k, and the steepness factor a. This formulation was used successfully in [16].

Appendix B

FALCON.m Optimal Control Framework

B.1 Derivation Hessian Chain Rule Equation

In the following, the Hessian chain rule formula is derived. Assume the two functions

$$R(Y) \in \mathbb{R}^{n_R \times 1}, \qquad Y(X) \in \mathbb{R}^{n_Y \times 1}, \qquad X \in \mathbb{R}^{n_X \times 1} \tag{B.1}$$

where X is an independent variable. Applying the Hessian chain rule equation from 4.5.2 gives

$$\frac{\partial^2 R}{\partial X^2} = \left(I_{n_R} \otimes \left(\frac{\partial Y}{\partial X} \right)^T \right) \cdot \frac{\partial^2 R}{\partial Y^2} \cdot \frac{\partial Y}{\partial X} + \left(\frac{\partial R}{\partial Y} \otimes I_{n_X} \right) \cdot \frac{\partial^2 Y}{\partial X^2} \tag{B.2}$$

where I_x represents an identity matrix of size x. The equation can easily be derived for the scalar case

$$r(y) \in \mathbb{R}, \qquad y(x) \in \mathbb{R}, \qquad x \in \mathbb{R} \tag{B.3}$$

by differentiation of the Jacobian Chain rule equation:

$$\frac{\partial}{\partial x} \left(\frac{\partial r}{\partial x} \right) = \frac{\partial}{\partial x} \left(\frac{\partial r}{\partial y} \cdot \frac{\partial y}{\partial x} \right) \tag{B.4}$$

$$\frac{\partial^2 r}{\partial x^2} = \frac{\partial^2 r}{\partial y^2} \cdot \left(\frac{\partial y}{\partial x} \right)^2 + \frac{\partial r}{\partial y} \cdot \frac{\partial^2 y}{\partial x^2}. \tag{B.5}$$

In the general case, the chain rule requires Kronecker products that expand the derivatives to match the required dimensions. In the following, their appearance is explained through logical conclusion. As all matrices are vectorized for derivative formulation using the vec transformation (see section 4.5.2), the formulation is general enough for the purpose of this thesis. As stated by [115], the Hessian chain rule can be derived. However, to the best of knowledge, a derivation or the actual formula was not found in any public literature.

Assume the case

$$R(Y) \in \mathbb{R}^{4 \times 1}, \qquad Y(X) \in \mathbb{R}^{2 \times 1}, \qquad X \in \mathbb{R}^{3 \times 1} \tag{B.6}$$

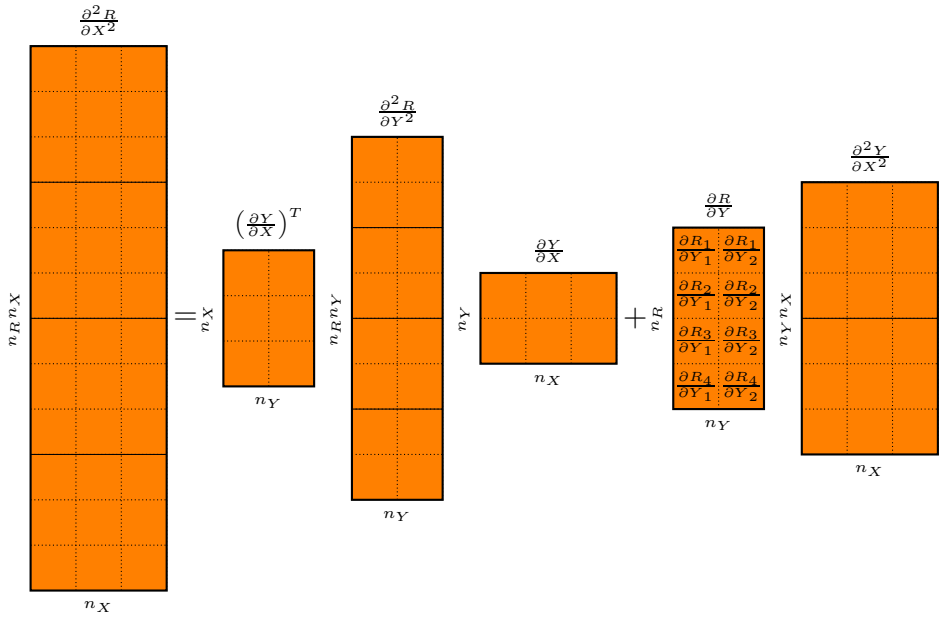


Figure B.1: Hessian chain rule layout without Kronecker products.

which is general enough to show the necessity of the Kronecker products. The derived chain rule without the products

$$\frac{\partial^2 R}{\partial X^2} = \left(\frac{\partial Y}{\partial X}\right)^T \cdot \frac{\partial^2 R}{\partial Y^2} \cdot \frac{\partial Y}{\partial X} + \frac{\partial R}{\partial Y} \cdot \frac{\partial^2 Y}{\partial X^2} \tag{B.7}$$

is visualized in Figure B.1. For each block, the derivative and its size are specified. Solid lines divide the square Hessian blocks and the dotted lines show the actual size of each derivative. A dimension mismatch appears at two points of the equation, namely

$$\left(\frac{\partial Y}{\partial X}\right)^T \cdot \frac{\partial^2 R}{\partial Y^2}, \qquad \text{and} \qquad \frac{\partial R}{\partial Y} \cdot \frac{\partial^2 Y}{\partial X^2}. \tag{B.8}$$

The first mismatch can be resolved easily. In case R is scalar, the mismatch disappears. As each entry of R is differentiated w.r.t. all Y entries, this specific part of the chain rule

$$\left(\frac{\partial Y}{\partial X}\right)^T \cdot \frac{\partial^2 R_j}{\partial Y^2} \cdot \frac{\partial Y}{\partial X}, \qquad j = 1, \ldots, n_R \tag{B.9}$$

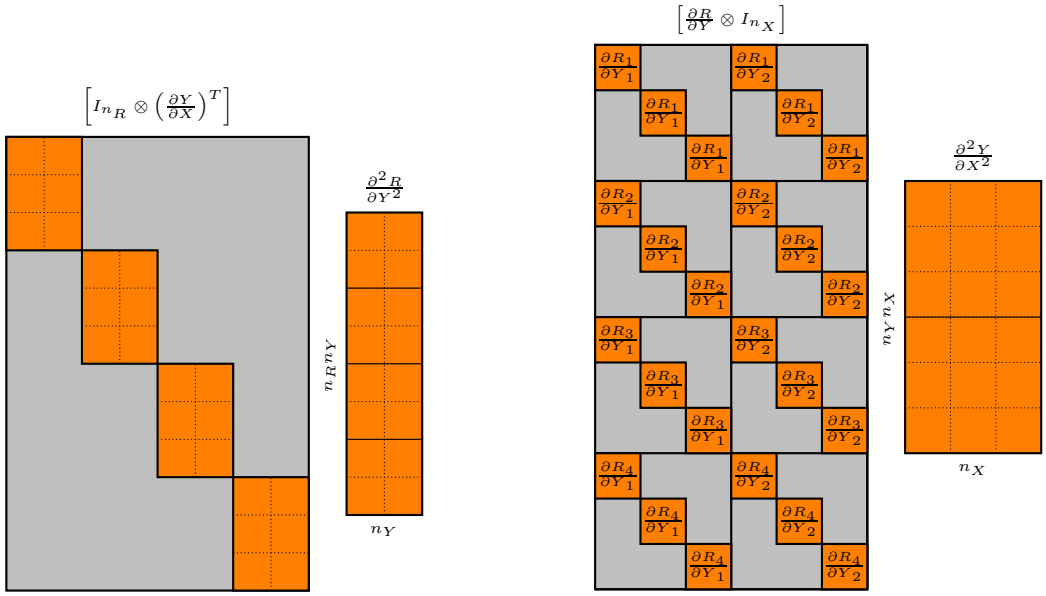
has to be applied block-wise. Using the Kronecker product,

$$\left[I_{n_R} \otimes \left(\frac{\partial Y}{\partial X}\right)^T\right] \cdot \frac{\partial^2 R}{\partial Y^2} \cdot \frac{\partial Y}{\partial X} \tag{B.10}$$

the calculation is generalized for vector cases of R. The brackets create a block diagonal matrix with copies of the transpose Jacobian (see Figure B.2a).

The second mismatch disappears in case the X is scalar. In order to fulfill the chain rule correctly, each entry of $\frac{\partial R}{\partial Y}$ must be multiplied with the corresponding Y Hessian block of $\frac{\partial^2 Y}{\partial X^2}$. Thus, the entries of the Jacobian act as a scaling to the Hessian blocks. In the general case this can be achieved by

$$\left[\frac{\partial R}{\partial Y} \otimes I_{n_X}\right] \cdot \frac{\partial^2 Y}{\partial X^2} \tag{B.11}$$



(a) Multiplying transposed Jacobian block-wise to the Hessian.

(b) Scaling Hessian for every element of the Jacobian.

Figure B.2: Kronecker adaptations of chain rule.

where the Kronecker product constructs a scaling matrix with size $[n_X, n_X]$ for each entry of the Jacobian (see B.2b).

The Hessian chain rule equation was successfully implemented in the subsystem derivative builder. However, due to the Kronecker products, many zero entries are created which reduce the performance significantly (approx. 50%) compared to the element-wise explanation above.

B.2 Simplified Aircraft Dynamics

Section 6.1 introduced the dynamic model used in the optimization. This model includes sophisticated aerodynamics, propulsion, and fuel flow characteristics which cannot be handled by the *MATLAB* Symbolic Math Toolbox. Therefore, a much simpler Three Degree of Freedom (3DOF) dynamic model is introduced which uses simple aerodynamics and propulsion. The earth is regarded flat and the mass is assumed to be constant. This model is used in the simple and moderate case. The latter takes into account wind influences and the model receives an additional input for the wind. This input is regarded to be constant and derivatives w.r.t. it are not calculated. On the other hand the simple model ignores wind influences thus making the dynamics much easier to differentiate.

The states and controls used in this section are show in Table B.1. It can be seen that the position is given in Cartesian coordinates of a local NED frame. The kinematics are stated in the kinematic frame K. Additionally, the model has three outputs. It is noted that for the simple case without wind influence, the aerodynamic bank angle μ_A becomes the kinematic bank angle μ_K. Since the subsystem derivative method splits the dynamic model into multiple subsystems which is resembled in the equations below.

Table B.1: States, Controls and Outputs for Simple Aircraft Dynamics

Name	**Symbol**	**Unit**
States		
x-Position (ned)	x^G	$[m]$
y-Positon (ned)	y^G	$[m]$
z-Position (ned)	z^G	$[m]$
Speed (kinematic)	V_K^G	$[\frac{m}{s}]$
Course Angle (kinematic)	χ_K^G	$[rad]$
Climb Angle (kinematic)	γ_K^G	$[rad]$
Controls		
Lift Coefficient	C_L	$[-]$
Bank Angle (aerodynamic)	μ_A	$[rad]$
Thrust Lever Position	δ_T	$[-]$
Outputs		
Mach Number	M	$[-]$
Load Factor (vertical)	n_z	$[-]$
Calibrated Air Speed	V_{CAS}	$[\frac{m}{s}]$

Subsystem M_KO and Position Propagation

The coordinate transformation form the NED frame O to the kinematic frame K is given by

$$M_{KO} = \begin{pmatrix} \cos\chi_K^G \cdot \cos\gamma_K^G & \sin\chi_K^G \cdot \cos\gamma_K^G & -\sin\gamma_K^G \\ -\sin\chi_K^G & \cos\chi_K^G & 0 \\ \cos\chi_K^G \cdot \sin\gamma_K^G & \sin\chi_K^G \cdot \sin\gamma_K^G & \cos\gamma_K^G \end{pmatrix}. \tag{B.12}$$

Thus the position propagation

$$\begin{bmatrix} u_K^G \\ v_K^G \\ w_K^G \end{bmatrix}_O = \left(\vec{v}_K^G\right)_O^O = {M_{KO}}^T \cdot \begin{bmatrix} V_K^G \\ 0 \\ 0 \end{bmatrix}_K \tag{B.13}$$

in NED can be calculated. Please note that the kinematic speeds in the NED frame are equivalent

$$\begin{bmatrix} \dot{x}^G \\ \dot{y}^G \\ \dot{z}^G \end{bmatrix} \equiv \begin{bmatrix} u_K^G \\ v_K^G \\ w_K^G \end{bmatrix} \tag{B.14}$$

to the state derivative for the position.

Subsystem Aerodynamic Speed and M_AO

In case of the moderately complex model the wind influences needs to be taken into account. The superposition of the wind

$$\begin{bmatrix} u_A^G \\ v_A^G \\ w_A^G \end{bmatrix}_O = \left(\vec{v}_A^G\right)_O^O = \left(\vec{v}_K^G\right)_O^O - \left(\vec{v}_W^G\right)_O^O = \begin{bmatrix} u_K^G \\ v_K^G \\ w_K^G \end{bmatrix}_O - \begin{bmatrix} u_W^G \\ v_W^G \\ w_W^G \end{bmatrix}_O \tag{B.15}$$

gives the aerodynamic speed components in the NED frame. From these, aerodynamic value such as speed, course angle and climb angle

$$V_A = \sqrt{\left(u_A^G\right)^2 + \left(v_A^G\right)^2 + \left(w_A^G\right)^2} \tag{B.16}$$

$$\chi_A = \arctan\left(\frac{v_A^G}{u_A^G}\right) \tag{B.17}$$

$$\gamma_A = -\arctan\left(\frac{w_A^G}{\sqrt{\left(u_A^G\right)^2 + \left(v_A^G\right)^2}}\right) \tag{B.18}$$

are calculation. Additionally, the matrix M_{AO} (see 6.18) gives the transformation form the aerodynamic frame A to the NED frame O.

Subsystem Atmosphere

In order to calculate the aerodynamic forces the current air density (dependent on the altitude)

$$\rho = \rho_s \cdot \left[1 - \frac{n-1}{n} \cdot \frac{g}{R \cdot T_s} \cdot H_G\right]^{\frac{1}{n-1}} \qquad H_G = \frac{r_E \cdot h^G}{r_E + h^G} \qquad h = -z \tag{B.19}$$

is required. It is calculated using ISA where n is the polytropic exponent, T_s, ρ_s the air temperature and density at sealevel, R the universal gas constant and r_E the radius of the earth. Additionally to the density the mach numer M and the calibrated air speed V_{CAS} are calculated.

Subsystem Aerodynamic Forces

The aerodynamic forces

$$\left(\vec{F}_A^G\right)_A = \begin{bmatrix} -\frac{\rho}{2} {V_A}^2 \cdot S_{ref} \cdot C_D \\ 0 \\ -\frac{\rho}{2} {V_A}^2 \cdot S_{ref} \cdot C_L \end{bmatrix} \tag{B.20}$$

are calculated using a simple quadratic drag polar

$$C_D = C_{D,0} + C_{D,2} \cdot {C_L}^2 \tag{B.21}$$

where S represents the surface area of the wing. Please note for the simple model the aerodynamic speed V_A is equivalent to the kinematic speed V_K^G.

Subsystem Gravity and Propulsion

The gravitational force is transformed into the K-frame using M_{KO}

$$\left(\vec{F}_P^G\right)_K = \begin{bmatrix} T_{max} \cdot \delta_T \\ 0 \\ 0 \end{bmatrix}_K \qquad \left(\vec{F}_G^G\right)_K = M_{KO} \cdot \begin{bmatrix} 0 \\ 0 \\ m \cdot g \end{bmatrix}_O \tag{B.22}$$

where m represents the aircraft mass and g the gravity constant. Additionally, the propulsion is assumed to act along the x-axis of the kinematic frame.

Subsystem Total Force

The total sum of force acting on the point mass is given by

$$\left(\vec{F}_T^G\right)_K = \begin{bmatrix} \left(X_T^G\right)_K \\ \left(Y_T^G\right)_K \\ \left(Z_T^G\right)_K \end{bmatrix}_K = \left(\vec{F}_A^G\right)_K + \left(\vec{F}_P^G\right)_K + \left(\vec{F}_G^G\right)_K. \tag{B.23}$$

If wind is considered the aerodynamic force in the K-frame is calculated

$$\left(\vec{F}_A^G\right)_K = M_{KO} \cdot {M_{AO}}^T \cdot \left(\vec{F}_A^G\right)_A \tag{B.24}$$

using the transformations matrices M_{KO} and M_{AO} (see 6.18). In the wind free case the aerodynamic force needs to be transformed from the A to the $\bar{A}$

$$\left(\vec{F}_A^G\right)_K = M_{\bar{A}A} \cdot \left(\vec{F}_A^G\right)_A \tag{B.25}$$

where the rotation matrix is given by

$$M_{\bar{A}A} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\mu_A & -\sin\mu_A \\ 0 & \sin\mu_A & \cos\mu_A \end{pmatrix}. \tag{B.26}$$

Subsystem Translation Equation of Motion

Thus, the translation equations of motion

$$\dot{V}_K^G = \frac{\left(X_T^G\right)_K}{m} \tag{B.27}$$

$$\dot{\chi}_K^G = \frac{\left(Y_T^G\right)_K}{m \cdot V_K^G \cdot \gamma_K^G} \tag{B.28}$$

$$\dot{\gamma}_K^G = \frac{\left(Z_T^G\right)_K}{m \cdot V_K^G} \tag{B.29}$$

can be calculated.

Appendix C

Optimization

In this chapter of the appendix additional information is given regarding chapters 5 and 6. The aircraft initial guess generation as well as B-Spline calculation are explained. Regarding the aircraft kinematics, the EULER differentiation is derived. Finally, the initial mass influence study of 6.5.1 is carried out with and without the artificial high lift penalty.

C.1 B Spline Initial Guess

The initial guess in aircraft optimization chapter is generated using a B-Spline. The primary goal is the creation of a smooth trajectory. Feasibility is not guaranteed and will be ensured by the optimization algorithm. In the following, the B-Spline is introduced. For detailed information please refer to detailed literature (e.g. [141]).

In a B-Spline there exist $n+1$ control points

$$B_1, B_2, \ldots, B_{n+1} \tag{C.1}$$

where each control point is multiplied with a basis function $N_{i,k}(s)$

$$B(s) = \sum_{i=1}^{n+1} B_i \cdot N_{i,k}(s) \tag{C.2}$$

giving the overall interpolate. The order of a spline is k resulting in a degree of $k-1$. The minimum value for the order is $k=2$ which represents a linear interpolation. The basis function is defined recursively

$$N_{i,k}(s) = \frac{s - r_i}{r_{i+k-1} - r_i} N_{i,k-1}(s) + \frac{r_{i+k} - s}{r_{i+k} - r_{i+1}} N_{i+1,k-1}(s) \tag{C.3}$$

$$N_{i,1}(s) = \begin{cases} 1, & r_i \leq s < r_{i+1} \\ 0, & \text{otherwise} \end{cases} \tag{C.4}$$

and can be interpreted as a linear interpolation of a linear interpolation a.s.o. It is dependent on the current order k and a knot vector

$$r_1, r_2, \ldots, r_{k+(n+1)}, \qquad r_i \leq r_{i+1} \tag{C.5}$$

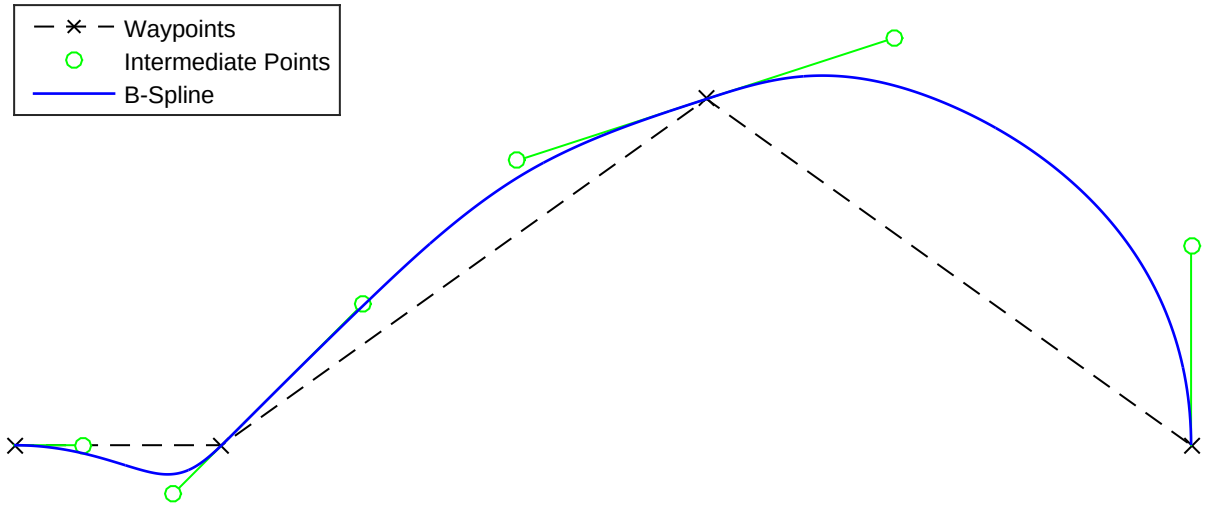


Figure C.1: Intermediate positions added to spline.

that defined a monotone increasing sequence. In this thesis, the spline shall pass through the first and last point. Therefore, an open uniform knot vector is used.

$$r_i = r_1, \qquad i \leq k \tag{C.6}$$
$$r_{i+1} - r_i = constant, \qquad k \leq i < n+2 \tag{C.7}$$
$$r_i = r_{k+n+1}, \qquad i \geq n+2. \tag{C.8}$$

In the following, the required input data to the initial guess calculation is explained and how the spline is created from it. Afterwards the 1st, 2nd and 3rd time derivative of the spline is calculation. This information is used to reconstruct kinematic values such as climb or bank angle.

C.1.1 Input Data

In order to generate an initial guess, the Cartesian position, the kinematic speed and the kinematic course / climb angle at certain waypoints are required. This information is used to generate a sequence of points that are fitted with a B-Spline. The interpolation order for the position interpolation and speed interpolation are given by

$$k_{\vec{x}} = 5, \qquad k_V = 3 \tag{C.9}$$

but other values may be chosen. To account for the flight direction and climbing angle, intermediate positions are added (see Figure C.1). These are added in direction and inclination of the provided point with a distance of $1/3$ of the connected arc length. The speed is weighted in the same way.

Using the intermediate points for the position and speed, the B-Spline is calculated. The final result are a spline for the position and the speed

$$\vec{\eta} = \begin{bmatrix} x(s) \\ y(s) \\ z(s) \end{bmatrix}, \qquad v(s) \tag{C.10}$$

formulating a smooth trajectory that can be used to calculate kinematic values. A differentiation w.r.t. time is required which will be presented in the next sections together with the kinematic value calculation.

C.1.2 Derivative Calculation

The kinematic value reconstruction requires the 1st 2nd and 3rd time derivative of the position. Additionally, the acceleration along the spline is required. In this section, the values are derived from the spline representation. Therefore, the following derivatives w.r.t. the spline parameter are created

$$x, \qquad \frac{\partial x}{\partial s} = x', \qquad \frac{\partial^2 x}{\partial s^2} = x'', \qquad \frac{\partial^3 x}{\partial s^3} = x''' \tag{C.11}$$

$$y, \qquad \frac{\partial y}{\partial s} = y', \qquad \frac{\partial^2 y}{\partial s^2} = y'', \qquad \frac{\partial^3 y}{\partial s^3} = y''' \tag{C.12}$$

$$z, \qquad \frac{\partial z}{\partial s} = z', \qquad \frac{\partial^2 z}{\partial s^2} = z'', \qquad \frac{\partial^3 z}{\partial s^3} = z''' \tag{C.13}$$

$$v, \qquad \frac{\partial v}{\partial s} = v', \qquad \frac{\partial^2 v}{\partial s^2} = v'' \tag{C.14}$$

where the spline distance derivative

$$\frac{\partial d}{\partial s} = \sqrt{x'^2 + y'^2 + z'^2} \tag{C.15}$$

is obtained by the position derivatives. Additionally, the distance derivative w.r.t. time

$$\frac{\partial d}{\partial t} = v \tag{C.16}$$

gives the speed along the trajectory.

1st Time Derivatives

The first order time derivative of the speed

$$\dot{v} = \frac{\partial v}{\partial t} = \frac{\partial v}{\partial s} \cdot \frac{\partial s}{\partial d} \cdot \frac{\partial d}{\partial t} \tag{C.17}$$

is expanded with the spline parameter s and the distance on the spline d. Using the derivatives above

$$\dot{v} = v' \frac{v}{\sqrt{x'^2 + y'^2 + z'^2}} \tag{C.18}$$

the time derivative is calculated. The position rates represent the velocity components

$$\dot{x} = x' \frac{v}{\sqrt{x'^2 + y'^2 + z'^2}}, \qquad \dot{y} = y' \frac{v}{\sqrt{x'^2 + y'^2 + z'^2}}, \qquad \dot{z} = z' \frac{v}{\sqrt{x'^2 + y'^2 + z'^2}} \tag{C.19}$$

are calculated analogous. Obviously, the absolute of the velocity vector gives the speed v.

2nd Time Derivatives

The second time derivative is analogous for all components. The derivation is given for the x component. The final result is stated for all components below.

To obtain the second time derivative (acceleration) of the position, (C.19) is differentiated w.r.t. time

$$\ddot{x} = \frac{\partial^2 x}{\partial t^2} = \frac{\partial}{\partial t} \cdot \frac{\partial x}{\partial t} = \frac{\partial}{\partial t} \left(\frac{x' \cdot v}{\sqrt{x'^2 + y'^2 + z'^2}} \right). \tag{C.20}$$

Expanding the time derivative with the spline distance d and the spline parameter s

$$\ddot{x} = \frac{\partial}{\partial s} \left(\frac{x' \cdot v}{\sqrt{x'^2 + y'^2 + z'^2}} \right) \cdot \frac{\partial s}{\partial d} \cdot \frac{\partial d}{\partial t} \tag{C.21}$$

and conducting the differentiation gives the result:

$$\ddot{x} = \frac{(x''v + x'v') \cdot \sqrt{x'^2 + y'^2 + z'^2} - x'v \cdot \frac{x'x'' + y'y'' + z'z''}{\sqrt{x'^2 + y'^2 + z'^2}}}{x'^2 + y'^2 + z'^2} \cdot \frac{v}{\sqrt{x'^2 + y'^2 + z'^2}}. \tag{C.22}$$

This result is further simpified to give the acceleration is all three components:

$$\ddot{x} = \frac{x''v^2 + x'v'v}{x'^2 + y'^2 + z'^2} - \frac{x'v^2 \cdot (x'x'' + y'y'' + z'z'')}{(x'^2 + y'^2 + z'^2)^2} \tag{C.23}$$

$$\ddot{y} = \frac{y''v^2 + y'v'v}{x'^2 + y'^2 + z'^2} - \frac{y'v^2 \cdot (x'x'' + y'y'' + z'z'')}{(x'^2 + y'^2 + z'^2)^2} \tag{C.24}$$

$$\ddot{z} = \frac{z''v^2 + z'v'v}{x'^2 + y'^2 + z'^2} - \frac{z'v^2 \cdot (x'x'' + y'y'' + z'z'')}{(x'^2 + y'^2 + z'^2)^2} \tag{C.25}$$

3rd Time Derivative

The third time derivative is not absolutely required to compute the basic kinematic values. However, due to completeness purposes and for future applications, the derivative is stated. As before, the 3rd time derivative is analogous for all components x, y and z. The derivation is given for the x component.

The second time derivative is differentiated w.r.t. time

$$\dddot{x} = \frac{\partial \ddot{x}}{\partial t} = \frac{\partial \ddot{x}}{\partial s} \cdot \frac{\partial s}{\partial d} \cdot \frac{\partial d}{\partial t} \tag{C.26}$$

and expanded by the spline distance d and spline parameter s. It can be seen that the derivative

$$\dddot{x} = \left[\frac{\partial}{\partial s} \left(\frac{(x''v^2 + x'v'v)}{x'^2 + y'^2 + z'^2} \right) - \frac{\partial}{\partial s} \left(\frac{x'v^2 \cdot (x'x'' + y'y'' + z'z'')}{(x'^2 + y'^2 + z'^2)^2} \right) \right] \cdot \frac{\partial s}{\partial d} \cdot \frac{\partial d}{\partial t} \tag{C.27}$$

will no longer fit on paper in a readable format. In the following, the derivative is split into two parts which are differentiated individually. For simplicity, only the derivative for the x component is stated. In order to generate the third derivatives for the other components y and z the bold written symbols shall be replaces appropriately.

For the first part

$$\frac{\partial}{\partial s}\left(\frac{\boldsymbol{x}''v^2+\boldsymbol{x}'v'v}{x'^2+y'^2+z'^2}\right)=\frac{\partial}{\partial s}\left(\frac{A}{B}\right) \tag{C.28}$$

can be reformulated as a quotient with

$$A=\boldsymbol{x}''v^2+\boldsymbol{x}'v'v, \qquad B=x'^2+y'^2+z'^2. \tag{C.29}$$

Using the differentiation rules the result is

$$\frac{\partial}{\partial s}\left(\frac{A}{B}\right)=\frac{\frac{\partial A}{\partial s}\cdot B-A\cdot\frac{\partial B}{\partial s}}{B^2} \tag{C.30}$$

with the parts

$$\frac{\partial A}{\partial s}=\boldsymbol{x}'''v^2+3\boldsymbol{x}''v'v+\boldsymbol{x}'v''v+\boldsymbol{x}'v'^2 \tag{C.31}$$

$$\frac{\partial B}{\partial s}=2\cdot\left(x'x''+y'y''+z'z''\right). \tag{C.32}$$

For the second part of the derivative, the term is divided into three parts

$$\frac{\partial}{\partial s}\left(\frac{\boldsymbol{x}'v^2\cdot\left(x'x''+y'y''+z'z''\right)}{\left(x'^2+y'^2+z'^2\right)^2}\right)=\frac{\partial}{\partial s}\left(\frac{A\cdot C}{B}\right) \tag{C.33}$$

with

$$A=\boldsymbol{x}'v^2, \qquad B=x'x''+y'y''+z'z'', \qquad C=x'^2+y'^2+z'^2. \tag{C.34}$$

The resulting derivative is

$$\frac{\partial}{\partial s}\left(\frac{A\cdot C}{B}\right)=\frac{\left(\frac{\partial A}{\partial s}\cdot C+A\cdot\frac{\partial C}{\partial s}\right)\cdot B-A\cdot C\cdot\frac{\partial B}{\partial s}}{B^2} \tag{C.35}$$

with

$$\frac{\partial A}{\partial s}=\boldsymbol{x}''v^2+2\boldsymbol{x}'v'v \tag{C.36}$$

$$\frac{\partial B}{\partial s}=x''^2+x'x'''+y''^2+y'y'''+z''^2+z'z''' \tag{C.37}$$

$$\frac{\partial C}{\partial s}=4\cdot\left(x'^2+y'^2+z'^2\right)\cdot\left(x'x''+y'y''+z'z''\right). \tag{C.38}$$

C.1.3 Kinematic States Reconstruction

The time derivatives of the B-Spline can be used to calculate kinematic states of the aircraft. The kinematic climb and course angle as well as their derivatives

$$\gamma_K^G=-\arctan\left(\frac{\dot{z}}{\sqrt{\dot{x}^2+\dot{y}^2}}\right) \qquad \dot{\gamma}_K^G=-\frac{\ddot{z}\sqrt{\dot{x}^2+\dot{y}^2}-\dot{z}\frac{\dot{x}\ddot{x}+\dot{y}\ddot{y}}{\sqrt{\dot{x}^2+\dot{y}^2}}}{\dot{x}^2+\dot{y}^2+\dot{z}^2} \tag{C.39}$$

$$\chi_K^G=\arctan\left(\frac{\dot{y}}{\dot{x}}\right) \qquad \dot{\chi}_K^G=\frac{\ddot{y}\dot{x}-\dot{y}\ddot{x}}{\dot{x}^2+\dot{y}^2} \tag{C.40}$$

are calculated using simple geometry. The kinematic bank angle is calculated using:

$$\mu_A=\arctan\left(\frac{n_y}{n_z}\right), \qquad n_y=\frac{V_K^G\dot{\chi}_K^G\cos\gamma_K^G}{g}, \quad n_z=\frac{V_K^G\dot{\gamma}_K^G}{g}+\cos\gamma_K^G. \tag{C.41}$$

C.2 Euler Differentiation

As can be seen in the section, the description of the aircraft dynamics require different coordinate system. In order to derive the differential equation for the dynamic model, vectors and matrices have to be differentiated w.r.t. certain coordinates, but their values given in other coordinates. For this reason the Euler Differentiation [119] is introduced. Assuming the a vector r given in the B

$$(\vec{r})_A = M_{AB} \cdot (\vec{r})_B \tag{C.42}$$

is transformed into the A frame using the transformation matrix M_{AB}. Taking the time derivative on both sides w.r.t. A

$$\left(\frac{d}{dt}\right)^A (\vec{r})_A = \left(\frac{d}{dt}\right)^A [M_{AB} \cdot (\vec{r})_B] \tag{C.43}$$

results in the following

$$\left(\dot{\vec{r}}\right)_A^A = \dot{M}_{AB}^A \cdot (\vec{r})_B + M_{AB} \cdot \left(\dot{\vec{r}}\right)_B^B. \tag{C.44}$$

If the time derivative w.r.t. A is desired in the B frame, a multiplication with the matrix M_{BA}

$$M_{BA} \cdot \left(\dot{\vec{r}}\right)_A^A = M_{BA} \cdot \dot{M}_{AB}^A \cdot (\vec{r})_B + \underbrace{M_{BA} \cdot M_{AB}}_{=I} \cdot \left(\dot{\vec{r}}\right)_B^B \tag{C.45}$$

results in the transformed derivative

$$\left(\dot{\vec{r}}\right)_B^A = \left(\dot{\vec{r}}\right)_B^B + \left(\Omega^{AB}\right)_{BB} \cdot (\vec{r})_B \tag{C.46}$$

where

$$M_{BA} \cdot \dot{M}_{AB}^A = \left(\Omega^{AB}\right)_{BB}. \tag{C.47}$$

The formulation

$$\left(\Omega^{AB}\right)_{BB} \cdot (\vec{r})_B = \left(\vec{\omega}^{AB}\right)_B \times (\vec{r})_B \tag{C.48}$$

is equivalent to a cross product which yields the

$$\left(\dot{\vec{r}}\right)_B^A = \left(\dot{\vec{r}}\right)_B^B + \left(\vec{\omega}^{AB}\right)_B \times (\vec{r})_B \tag{C.49}$$

Euler vector differentiation formula.

C.3 Artificial High Lift Penalty Cost Comparison

In this section, the results of the initial aircraft mass influence study of section 6.5.1 re-generated. However, this time, the artificial high lift penalty is not taken into consideration.

In Figure C.2 the high lift and landing gear switching structure are stated for both cases. It clearly can be seen that the results become inconsistent in case the high lift penalty is not taken into consideration. Additionally, the fuel consumptions and switching costs for both cases are displayed. Both costs without high lift penalty are always higher or equal to the high lift penalty case.

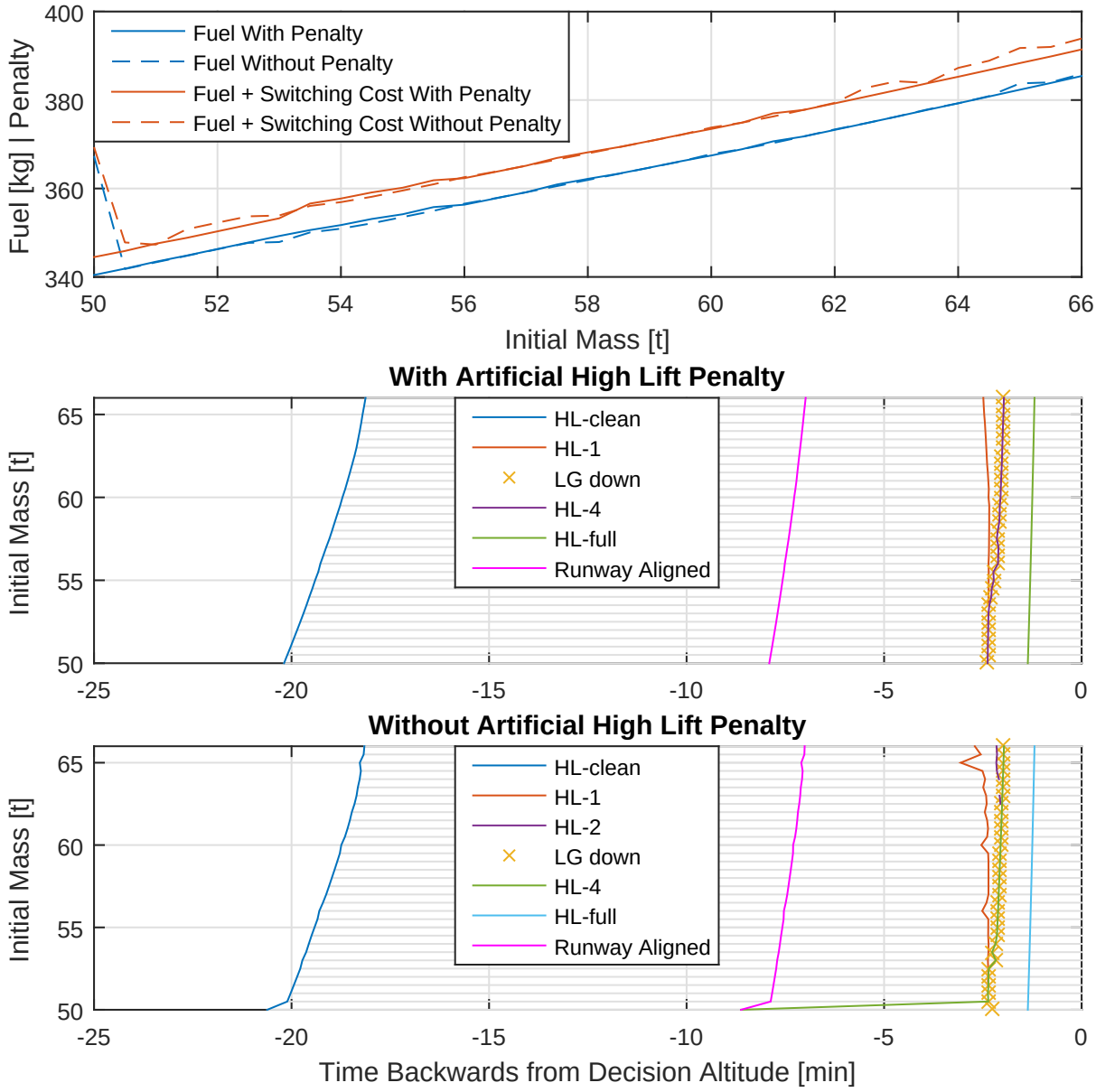


Figure C.2: Comparison of mass influence parameter study with and without the artificial high lift penalty.