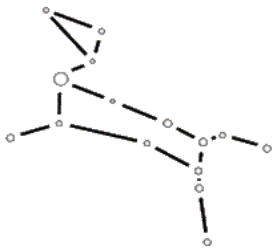
NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF MECHANICAL ENGINEERING
DEPARTMENT OF M.D. & C.S.
Control Systems Laboratory

Diploma Thesis

Modeling and Model Predictive Control of Quadruped Robots

Athanasios Avgeris

Supervisor: E.G. Papadopoulos



ATHENS, 2024

Περίληψη

Η παρούσα εργασία πραγματεύεται τον σχεδιασμό και υλοποίηση ενός μοντέλου προσομοίωσης του τετράποδου ρομπότ ARGOS στον προσομοιωτή Gazebo [1] και ενός ελεγκτή που χρησιμοποιεί βάσει μοντέλου προβλεπτικό έλεγχο (MPC) στο ROS (Robot Operating System) [2]. Το ARGOS είναι ένα τετράποδο ρομπότ που σχεδιάστηκε από το Εργαστήριο Αυτομάτου Ελέγχου της Σχολής Μηχανολόγων Μηχανικών, και σκοπός του είναι η επιθεώρηση αμπελώνων. Σχήματα ελέγχου σαν αυτά που παρουσιάζονται σε αυτή την εργασία θα επιτρέπουν στο τετράποδο να εκτελεί κινήσεις στον τρισδιάστατο χώρο στους αμπελώνες. Το ROS χρησιμοποιήθηκε διότι επιτρέπει την υλοποίηση και εκτέλεση νόμων ελέγχου σε πραγματικό χρόνο, είτε σε hardware είτε σε προσομοίωση. Χάρις στην συνεργασία του Gazebo με το ROS καθίσταται δυνατή η προσομοίωση και ο έλεγχος του ARGOS σε πραγματικό χρόνο. Οι μετρήσεις που χρειάζονται σαν ανάδραση στο σύστημα κλειστού βρόχου είναι οι μετρήσεις που παρέχουν ιδιοδεκτικοί αισθητήρες.

Το δεύτερο κεφάλαιο περιλαμβάνει μια αναλυτική επισκόπηση της διαδικασίας μοντελοποίησης ρομπότ στο Gazebo καθώς επίσης και του επιμέρους λογισμικού που διαθέτουν τέτοιου είδους προσομοιωτές ρομπότ. Επικεντρώνεται στην επιλογή και ρύθμιση των διαφόρων παραμέτρων του Gazebo που απαιτούνται για τη δημιουργία μιας ρεαλιστικής προσομοίωσης. Τέλος, πραγματοποιήθηκε μια επαλήθευση της ακρίβεια της προσομοίωσης στο Gazebo χρησιμοποιώντας αναλυτικά μοντέλα του συστήματος που σχεδιάστηκαν στο Simulink. Οι συγκρίσεις ανάμεσα στο μοντέλο στο Gazebo και στο αναλυτικό μοντέλο απέδειξαν ότι η ακρίβεια της μοντελοποίησης στο Gazebo είναι ικανοποιητική.

Το τρίτο κεφάλαιο περιλαμβάνει μια αναλυτική επισκόπηση των μεθόδων βέλτιστου ελέγχου (ανοιχτού/κλειστού βρόχου) και πιο συγκεκριμένα των διαφορετικών διατυπώσεων προβλημάτων βέλτιστου ελέγχου (OCPs) και των μεθόδων επίλυσης που χρησιμοποιούνται για κάθε διατύπωση. Στη συνέχεια, οι διάφορες μέθοδοι βελτιστοποίησης τροχιάς (TO) συγκρίθηκαν μεταξύ τους χρησιμοποιώντας ένα διπλό εκκρεμές σε προσομοίωση προκειμένου να αποφασισθεί ποια είναι η πιο κατάλληλη μέθοδος για τον MPC του τετράποδου. Αυτά τα προβλήματα λύθηκαν χρησιμοποιώντας έτοιμα πακέτα βελτιστοποίησης και επιλύτες καθώς και ειδικά προσαρμοσμένους επιλύτες. Τα αποτελέσματα των συγκρίσεων απέδειξαν ότι η Κυρτή Βελτιστοποίηση (Convex Optimization) είναι η καταλληλότερη μέθοδος καθότι μπορεί να επιλύσει τέτοια προβλήματα πολύ γρήγορα και αξιόπιστα (με εγγυημένες ταχύτητες επίλυσης), γεγονός που την καθιστά ιδανική για εφαρμογές ελέγχου σε πραγματικό χρόνο. Συνεπώς, ο ελεγκτής που θα χρησιμοποιηθεί είναι ένας Convex MPC.

Το τέταρτο κεφάλαιο εστιάζει στην εφαρμογή του Convex MPC συγκεκριμένα σε τετράποδα ρομπότ. Αναλύει τις παραδοχές που έγιναν έτσι ώστε να επιτευχθεί η διατύπωση του Convex MPC, την ίδια την διατύπωση του Convex MPC σε τέτοια συστήματα, συμπεριλαμβανομένης της μορφής της συνάρτηση κόστους/δείκτη απόδοσης και των περιορισμών που σχετίζονται με τη δυναμική του συστήματος και με τις δυνάμεις αλληλεπίδρασης των ποδιών με το έδαφος καθώς και τα επιμέρους συστατικά μέρη του συστήματος ελέγχου (π.χ. προγραμματιστής βηματισμού, σχεδιαστής βημάτων). Η επιθυμητή κίνηση προσδιορίζεται από το είδος του βηματισμού και των χαρακτηριστικών του καθώς και από εντολές υψηλού επιπέδου όπως την επιθυμητή πόζα και ταχύτητα του σώματος του ρομπότ κατά τη διάρκεια της κίνησης. Οι είσοδοι ελέγχου που οδηγούν το σύστημα υπολογίζονται βάσει αυτών των εντολών υψηλού επιπέδου.

Τέλος, στο πέμπτο κεφάλαιο, παρουσιάζονται τα αποτελέσματα των πειραμάτων τα οποία διεξήχθησαν χρησιμοποιώντας τη προαναφερθείσα προσομοίωση και ελεγκτή στο ARGOS. Ο MPC δοκιμάστηκε για δύο διαφορετικά είδη βηματισμών: περπάτημα (walk) και τροχασμό (trot). Το ρομπότ είναι ικανό να παρακολουθεί με ικανοποιητική ακρίβεια την εντολή πόζας και ταχύτητας χωρίς να υπερβαίνονται τα όρια ροπής των κινητήρων κατά τη διάρκεια του βηματισμού.

Abstract

The present thesis focuses on the design and implementation of a simulation framework for the quadruped ARGOS using the real-time physics simulator Gazebo [1] and of a control framework that utilizes Model Predictive Control (MPC) in ROS (Robot Operating System) [2]. ARGOS is a quadruped robot designed by the Legged Robots Team of the Control Systems Lab in NTUA, whose main purpose is the inspection of vineyards. Such control schemes will allow the quadruped to execute 3D locomotion maneuvers in the vineyards. ROS was utilized since it allows the implementation and execution of control laws in real-time, whether in hardware or simulation. Gazebo and its tight integration with ROS make the real-time simulation and control of ARGOS possible. The feedback measurements needed by the closed loop system were measurements that proprioceptive sensors provide.

The second chapter includes a thorough study of Gazebo modelling procedure of robots and an overview of the constituent functionalities of such simulators, (e.g., physics and collision detection engine). It focuses on the selection and tuning of the various Gazebo parameters needed to create a realistic simulation. Finally, a validation of the accuracy of the Gazebo simulation was conducted using analytical models of the system built in Simulink. The comparisons between the Gazebo and the analytical model proved that the accuracy of the Gazebo modeling was satisfactory.

The third chapter includes an overview of the optimal control methods (open/closed loop) and more specifically the different optimal control problem (OCP) formulations and the solution methods utilized for each formulation. Then, the various trajectory optimization (TO) methods are compared using a simulated double pendulum to determine which is the most suitable for the quadrupeds' MPC controller. Such problems were solved using off-the-shelf optimization packages and solvers as well as custom made solvers. The findings of the comparisons proved that Convex Optimization is the most suitable method due to its ability to solve such problems very fast and reliably (at guaranteed rates), which is ideal for real-time control applications. Therefore, a Convex MPC controller has been utilized.

The fourth chapter focuses on the application of Convex MPC specifically on quadruped robots. It elaborates on the assumptions made to accomplish a Convex MPC formulation, on the formulation of the Convex MPC itself for such systems, including the form that the cost function/performance index and the constraints pertinent to the system's dynamics and ground reaction forces have and on the constituent components of the control system (e.g., gait scheduler, footstep planner). The desired locomotion task is specified by the gait type and characteristics and by high-level commands like the desired pose and velocity of the robot's body during that task. The control inputs that drive the system are computed based on these high-level commands.

Finally, in the fifth chapter, the results of experiments conducted using the aforementioned simulation and control frameworks on ARGOS are presented. The MPC controller was tested on two different gait types: walk and trot. The robot is capable of tracking accurately enough the commanded pose and velocity without exceeding the motor torque limits during a gait.

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor, Professor Evangelos G. Papadopoulos. I am thankful to him for giving me the opportunity to become a member of his research team. Also, I genuinely appreciate his guidance and advice throughout the elaboration of this thesis.

Moreover, I would like to thank the Ph.D. candidates of the CSL-EP lab, Konstantinos Koutsoukis, Konstantinos Machairas and especially Athanasios Mastrogeorgiou for introducing me to the fascinating world of robotics, and for their invaluable contribution, feedback, and support they offered to help me address all the challenges that arose throughout the conduction of this study.

Furthermore, I owe much to my family since they were always there for me, believed in me and supported me through my studies. Finally, I would like to thank my friends for their continuous help and encouragement.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Definition
ABA	Articulated Body Algorithm
AD	Automatic/Algorithmic Differentiation
ADMM	Alternating Direction Method of Multipliers
AL	Augmented Lagrangian
BVP	Boundary Value Problem
CFM	Constraint Force Mixing
CoM	Center of Mass
CoP	Center of Pressure
CPU	Central Processing Unit
CRBA	Composite Rigid Body Algorithm
DAE	Differential Algebraic Equation
DARE	Discrete Algebraic Riccati Equation
DDP	Differential Dynamic Programming
DoF	Degrees of Freedom
DP	Dynamic Programming
EKF	Extended Kalman Filter
EoM	Equations of Motion
ERP	Error Reduction Parameter
GN	Gauss-Newton
GNMS	Gauss-Newton Multiple Shooting
GRF	Ground Reaction Force
HJB	Hamilton-Jacobi-Bellman
iLQG	Iterative Linear Quadratic Gaussian Regulator
iLQR	Iterative Linear Quadratic Regulator
IMU	Inertia Measurement Unit
IP	Interior Point
LCP	Linear Complementarity Problem
LIPM	Linear Inverted Pendulum Model
LP	Linear Program
LQR	Linear Quadratic Regulator
LTI	Linear Time-Invariant
LTV	Linear Time-Varying
MoCap	Motion Capture
MPC	Model Predictive Control
NLP	Nonlinear Programming Problem
NMPC	Nonlinear Model Predictive Control
OCP	Optimal Control Problem
ODE	Ordinary Differential Equation
OpenDE	Open Dynamics Engine
PDE	Partial Differential Equation
PGS	Projected Gauss-Seidel
PMP	Pontryagin’s Maximum Principle
PR-MPC	Policy Regularized Model Predictive Control
QCQP	Quadratically Constrained Quadratic Program
QP	Quadratic Program

RBD	Rigid Body Dynamics
RF-MPC	Representation Free Model Predictive Control
RK3	Runge-Kutta 3rd order
RK4	Runge-Kutta 4th order
RL	Reinforcement Learning
RPC	Regularized Predictive Control
SLQ	Sequential Linear Quadratic Programming
SOCP	Second Order Cone Program
SOR	Successive Over-Relaxation
SQP	Sequential Quadratic Programming
SRBD	Single Rigid Body Dynamics
TO	Trajectory Optimization
WBD	Whole Body Dynamics
WBIC	Whole-Body Impulse Control
ZMP	Zero Moment Point

1 Introduction

1.1 Motivation

One of the objectives of the present thesis is to create a realistic simulated model for the quadruped robot ARGOS. Testing and designing software are easier and more effective when using a simulated robot rather than the real-world one. Accurate simulated models of robots do not require access to the real hardware for testing software. Also, such tests provide proof of concept regarding the implemented control algorithms. More specifically, the user can develop software that can be validated using a simulated robot in a variety of environments and for different tasks without damaging the associated physical systems in the process. Additionally, any software flaws that emerged during the validation process will not be transferred to the real-world robot.

The main objective of the present thesis is optimization-based motion planning and control of quadruped robots in real-time. A simple hand-tuned controller cannot achieve a high level of performance or fully exploit the capabilities of the hardware, which leads to the development of model-based optimal controllers. Unlike conventional approaches, optimization-based ones produce motion plans that minimize a performance index/cost function (e.g., energy consumption, task completion time) and satisfy physical constraints of the system (e.g., motor torque limits, leg kinematic limits). Moreover, in the case of Model Predictive Control (MPC), where the motion is replanned frequently online through optimization, the robot becomes extremely robust against disturbances and capable of adapting to a dynamically changing environment. Therefore, the design and implementation of MPC controllers for legged robots has been an active research area in the past decade. Focus has been given to making the optimal control problems (OCPs) more computationally tractable. This includes attention to the formulation of the OCP, the simplification of the model's dynamics, handling contact with the environment and selecting a numerical technique to solve the OCP based on these choices.

1.2 Literature Review

1.2.1 Legged Robots

In recent years, there has been a significant increase in research interest in the broader field of robotics in legged robots. The main incentive behind this interest lies in the potential of legged robots to be extensively utilized for numerous applications due to their versatility compared to wheeled ones. After all, in a world where legs are the most competent locomotion mechanism in nature, legged robots have a significant role to play. More precisely, the value of legged robots lies in their distinct ability to navigate and move through unstructured environments. This trait could be harnessed to assist or even replace humans in many dangerous occupations such as firefighting, search and rescue missions in disaster areas (e.g., post-earthquake locations), space exploration, and monitoring and inspection in locations with hazardous waste. These tasks pose risks to humans, who are often obligated to complete them. By deploying robots in harm's way instead of humans, potentially dangerous situations for human workers can be avoided or even eliminated entirely. Therefore, research conducted on legged robots and their applications can contribute not only to improving people's quality of life but also to potentially saving many human lives.

In the past decade, there has been a rapid advancement in the capabilities of legged robots, with various academic institutions and high-tech companies creating state-of-the-art platforms. This transitioned from a time when only a limited number of research groups had access to capable platforms to the current state where locomotion is widespread in both industry and academic laboratories, addressing fundamental issues and challenges. State of the art attempts, such as Boston Dynamics' Atlas [3], Spot [4], and BigDog [5], along with others such as Anybotics' ANYmal and its recent successor ANYmal C [6], MIT Cheetah [7] and Agility Robotics' most recent biped DIGIT [8], showcase current research trends in the field (Figure 1-1).

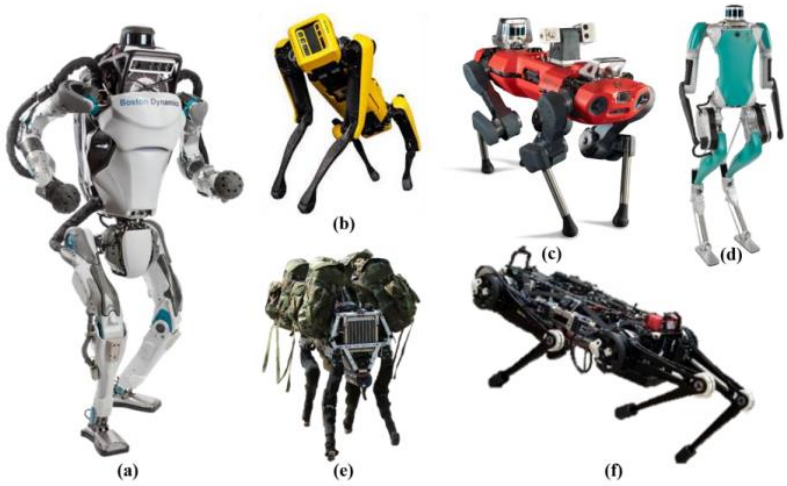


Figure 1-1. (a) Boston Dynamics Atlas. (b) Boston Dynamics Spot. (c) ANYbotics ANYmal-C. (d) Agility Robotics DIGIT. (e) Boston Dynamics BigDog. (f) MIT Cheetah 3.

1.2.2 Motion Planning and Control

MPC's main challenge is to solve finite-horizon OCPs at a real time rate. Especially when using full-body dynamics, it is a nonlinear optimization problem that needs to be solved within a few milliseconds [9]. This process comes with a high computational cost that defies online execution.

Algorithms that plan physically feasible motions for legged robots can subsequently be executed using a tracking controller or embedded into an MPC formulation [10]. In many cases, MPC is used as a higher-level planner rather than as the primary stabilizing controller.

Available solutions for the motion planning problem include heuristic controllers [11], inverse dynamics control [12], and hierarchical operational space control [13]. In [14], [15] the authors optimize the motion-plan in an MPC fashion, resulting in a controller that can react to disturbances and handle model inaccuracies. In [14] a linearized Zero Moment Point (ZMP) planner was used to generate planar Center of Mass (CoM) motion plans, which were executed in an MPC-like fashion via a hierarchical whole-body controller. The desired joint accelerations and contact forces are found by solving a cascade of prioritized Quadratic Programming (QP) problems (or Quadratic Programs). However, it could not generate and track more agile motions, like trotting, dynamics lateral walk and pronking. For modeling and computation of kinematics and dynamics, the open-source Rigid Body Dynamics Library (RBDL) [16] is being used. To numerically solve the QP problems, a custom version of the QuadProg++ [17] library is being used. Experiments were conducted on ANYmal, an accurately torque-controllable quadruped robot.

So, in [15], a nonlinear ZMP planner was used to optimize the CoM position, as well as a parallelized optimization of footholds. Transition between gaits was possible with the addition of a dedicated gait switching module. A Sequential Quadratic Programming (SQP) method was used for the motion optimization which included solving the nonlinear ZMP constraints. The optimization could be evaluated at more than 100 Hz, although this frequency varies depending on the type of gait and optimization parameters. This approach also produced motion plans online and could execute them in MPC fashion by planning online the next motion plan with the most recent state while tracking the current motion plan with a whole-body controller. For modeling and computation of kinematics and dynamics, RBDL is used. The nonlinear optimization problem is solved by using a custom library which implements the SQP framework, which solves the nonlinear program by iterating through a sequence of QP problems. Each QP is numerically solved using the QuadProg++ library. Experiments were conducted on ANYmal.

Multiple attempts of solving motion planning through contacts using a hierarchy of optimization problems have existed [18], [19], [20], [21]. The downside is that the lower hierarchies need to respect the constraints of the higher hierarchies, often leading to heuristics which impair the optimality of the overall solution.

Work by Winkler et al. generates footholds and CoM trajectories for a certain look ahead period and then has a separate controller track the motion in practice [22], [23]. In [22], walking motion were generated without the use of an explicit footstep planner, by simultaneously optimizing over both the CoM trajectory and the footholds. Kinematic constraints between the footholds and the CoM position are explicitly enforced. Also, stability constraints are imposed on the CoM related to the ZMP. In this way, it is ensured that the planned motion is dynamically feasible. This problem is solved online by the Nonlinear Programming solver IPOPT [24]. The performance of this approach was evaluated on the hydraulically actuated quadruped robot HyQ [25].

A downside of [22] is that it can only generate motions for quadruped robots with three or more legs on the ground. This means that it requires 2D-support areas and thus point- and line-contacts can’t be modeled. Typically, such more dynamic motions are generated using Capture Point [26] approaches. However, if a more static gait is desired, the approach has to be switched to the one using ZMP with 2D-support areas. As a result, more complex motions where the gait cannot be precisely categorized, cannot be planned with this method. Therefore, in [23] they introduced a vertex-based representation of the Center of Pressure (CoP) constraint which allows treating arbitrarily oriented point-, line- and area-contacts uniformly. Thus, more complex motions like trotting and bounding can be planned. The motion planning problem could be solved for multiple steps in less than a second to generate walking, trotting, and push-recovery motions. Here the nonlinear programming problem (NLP) was solved with IPOPT and SNOPT [27]. The performance of this locomotion framework was evaluated on the quadruped robot HyQ.

Both [22] and [23] can be executed in a few milliseconds (~3 ms and ~35 ms respectively) but they reach their limits as motions and terrains become more complex, due to the use of the Linear Inverted Pendulum Model (LIPM), which has as input the position of the CoP. Also, not all kinematic plans are dynamically feasible since they may exceed actuation limits or friction cones.

In [28] these issues are tackled by replacing the LIPM model with the 6D Single Rigid Body Dynamics (SRBD) model. Since with this model contact forces can be handled directly

as part of the optimization problem, unilateral and friction constraints can be directly formulated as well as generate motions with flight phases or vertical motions. This framework automatically determines the gait-sequence, step-timings, footholds, swing-leg motions and 6D body motion over non-flat terrain. To enable the algorithm to automatically determine the gait-sequence, step-timings and footholds using only continuous optimization variables, a new phase-based foot parameterization was introduced. After defining all the constraints, the solution to the problem can be obtained by solving an NLP, which can be solved fast (~100 ms). The NLP was solved using the Interior Point Method solver (IPOPT). The entire software used to generate these motions is TOWR, is presented in [28] and is freely available. The software was successfully tested on hardware, more specifically on the quadruped ANYmal. However, the complexity of the formulation, for gait parameterization, makes the problem difficult to solve for real-time applications.

In most state-of-the-art works, motion planning for legged locomotion is often tackled with control frameworks, which typically consist of one or more planner stages that use simplified dynamics models and a reactive tracking controller. Despite the successful results, such planners do not always produce feasible motions and need to plan conservatively to do so. The tracking controllers also do not have enough control authority to e.g., modify footholds or contact timings, and only try to track the planned reference blindly. To address this challenging problem, whole-body optimization approaches are increasingly proposed, yet their runtimes are in most cases a few orders of magnitude away from running in receding horizon or MPC fashion.

Some of the current existing MPC methods for quadrupedal locomotion tackle these challenges of the MPC problem through careful software designs and high-performance, parallel implementations [29], [30], [31]. In addition, they adopt simplified dynamics models to reduce computational burdens: one common simplification is pre-defining the footholds with heuristics-based methods [32], [33], [34]. This, however, restricts the range of achievable motion and the capability to reject external disturbances.

Nonlinear optimization techniques, unlike convex optimization solvers, are not guaranteed to find the global optimum because multiple local optima exist and, depending on the initialization of the method, might get stuck on one of these. They also tend to suffer from numerical issues. Recent progress on convex optimization [35] and applications to model predictive control [36] has led to the development of many open-source solvers such as qpOASES [37], OSQP [38] and OOQP [39] that enable QP-based MPC problems to be solved rapidly and reliably.

In [32] MPC problem is transcribed into a QP, formulating the problem as convex optimization, after linearizing SRBD model. With this simplified model, which still captures the full 3D nature of the system, ground reaction force (GRF) planning problems are formulated for prediction horizons of up to 0.5 seconds and are solved in real time at a rate of 20-30 Hz. Experimental results, on the MIT Cheetah 3 Robot, demonstrate control of gaits including trot, pronk, bound and a full 3D gallop. The qpOASES, a QP solver which uses online active set strategy, was used to solve the convex optimization problem.

Later, in [33], an extension of the framework of [32] was proposed. This controller complements the model predictive control (MPC) implementation of [32] with a whole-body impulse control (WBIC) implementation. The MPC is a QP-based MPC using simplified dynamics, like in [32], that computes an optimal reaction force profile over a longer time horizon. The WBIC computes joint torque, position, and velocity commands based on the

reaction forces computed from MPC. More specifically, the first part of WBC computes joint position, velocity, and acceleration commands, by utilizing an inverse kinematics algorithm that strictly holds task priority. The later part modifies the reaction forces computed by MPC, also including the acceleration command found previously, to incorporate body stabilization and swing leg control. This is accomplished by solving another QP using an open source solver [17], efficient for small problems. The torque commands are computed given the reaction forces, configuration space acceleration and multi-body dynamics. In the experiments conducted on the Mini-Cheetah robot, high speed dynamic locomotion with aerial phases was achieved. They enabled the MPC to run at 30 Hz and WBIC to run at 500 Hz on the computer installed in the robot.

As simple as it may seem, the MPC framework in [32] is reliable and there exist many approaches that try to improve Convex MPC performance [40], [41], [42]. In [40] a representation-free linear model predictive control (RF-MPC) framework was developed, where rotational dynamics were represented using the rotation matrix. As a result, issues associated with the use of Euler angles, like gimbal lock, and quaternion orientation representations were nonexistent. This was possible thanks to a variation-based linearization technique for the SRBD model. These changes enabled the MPC to be transcribed into the standard QP form. The MPC controller was implemented on the quadruped robot Panther and could operate at real-time rates between 160 and 250 Hz. The gaits tested were trot, bound and backflip. The QP was solved by the custom solver qpSWIFT [43] for all the experiments.

In [42], a new linearization of the robot's centroidal dynamics was proposed. By expressing the angular motion with exponential coordinates, more linear terms are identified and retained than in the existing methods to reduce the loss from the model linearization. The proposed MPC was tested on the quadruped robot developed by Tencent Robotics X. The MPC prediction happens at 100 Hz using the QP solver qpOASES.

Approaches based on dynamic programming (DP), such as Differential Dynamic Programming (DDP) [44], Iterative Linear Quadratic Gaussian Regulator (iLQG) [45] or Iterative Linear Quadratic Regulator (iLQR) [46], and Sequential Linear Quadratic Programming (SLQ) [47] algorithms (Riccati-style) can also be used to generate optimal motion-plans.

Application of this work to humanoids has been done by [48], [49] which demonstrated impressive results, but did not present hardware results. In [48], Tassa et al., demonstrated in simulation that it is feasible to deploy MPC based on the robot's full dynamics with using iLQR. This approach employed MuJoCo's smooth and invertible contact model. Also in [49] there weren't presented any dynamic motions. Later work in [50] included hardware results but the MPC horizon was short and only quasi-static motions with slow contact changes and kinematic manipulation tasks were tested. A reason is that MuJoCo's contact model seemed to be not accurate enough for locomotion tasks.

Other approaches based on these algorithms generate motions-plans for switched systems, such as a quadruped robot, quick enough to be used in an MPC control loop [29], [30], [51], [52], [53]. In [54] the problem was formulated as an OCP for switched systems based on SLQ. The centroidal dynamics and the full kinematics [55] were employed in the switched system model where the switching times, the contact forces and the joint velocities were optimized for different tasks such as gap crossing, walking and trotting. Simulation results were presented on HyQ, to validate the approach.

Later, based on [54], Farshidian et al. demonstrate their FastSLQ-MPC for generating a real-time, constrained, whole-body nonlinear MPC to create trotting gaits using the same dynamic model [29]. The performance of the algorithm was improved by introducing a multi-processing scheme for estimating value function in its backward pass. This allowed them to optimize over longer horizons without proportional increase in computation time. For evaluating this MPC approach, experiments were conducted on HyQ that generated dynamic gaits like trotting. This MPC could generate trajectories for the next phases of motion in a few milliseconds. Using the four independent threads of their processor, the MPC loop could run at about 60 Hz. Various other recent approaches, like those presented in [56], [57], [58]. have been developed based on this approach.

A real-time whole-body nonlinear MPC (NMPC) implementation is shown on hardware in [30]. A whole-body 3D model was used and neither contact switching times nor contact events nor contact points were defined a priori but were optimized by the solver. A linear spring-damper contact model was used. Both iLQR and Gauss-Newton Multiple Shooting (GNMS) [59] approaches were used and tested on hardware. Rigid Body Dynamics and kinematics, which are essential for the controller implementation, were modelled using the efficient code generation framework RobCoGen [60]. Also Control Toolbox (CT) was used in this work which is available as open-source software [61]. Hardware experiments were conducted on the quadruped robots HyQ and ANYmal. During trotting experiments on ANYmal, iLQR achieved update rates of around 80 Hz while GNMS reaches almost 190 Hz, for a time horizon of half a second. Although the higher update rate does not lead to significantly better performance, it encourages the use of more extended time horizons**.** Despite all these results, they note that computation and real-world limitations must still be overcome to take full advantage of the nonlinear optimization's capabilities. Also, only simple motions, walking and trotting in flat terrain, were demonstrated. Moreover, tuning the hyperparameters of the contact model is a complex task, because non-zero "phantom" (virtual) forces may be created artificially during flight phases. These "phantom" forces may create unrealistic motions plans, during the flight phases, and consequently agile and complex motions will not be generated, under this assumption. Finally, because of the use of iLQR (unconstrained DDP), actuation limits were not considered during motion planning.

Later works like [52], [53] aim to use the full-body dynamics in predictive control as well. In the MPC approach in [53] actuation limits are taken into consideration for the optimization and tracks the optimal policy while executing agile maneuvers. They demonstrate their Riccati-based, state-feedback predictive controller (BOX-FDDP), which has been open sourced in the Crocoddyl (Contact RObot COntrol by Differential DYnamic programming Library) repository [62]. BOX-FDDP also employs a multiple-shooting approach which enables the parallel computation of analytical derivatives. Crocoddyl also utilizes Pinocchio [63] for fast computation of robot dynamics and of the analytical derivatives of the dynamics and of other functions. The controller was tested on hardware, specifically on ANYmal B and C quadrupeds. In most of their experiments, the predictive controller could compute local feedback policies at 50 Hz, although it could also operate at up to 100 Hz, with the use of the offboard MPC computers, over an optimization horizon of 1.25 seconds.

Motion generation in an MPC fashion using a direct method with collocation and a SRBD model is used in [31], [64], [65], [66] without pre-specified contact locations and with pre-specified contact locations in [67]. In general, direct methods using collocation are not as efficient and fast as DDP based methods due to the expensive factorizations that their linear

solvers do. This limits their practical use to real time control on reduced models [68] or motion planning [28], [69] in robotics.

In [64], they proposed a policy-regularized MPC (PR-MPC), where regularization is added to direct collocation optimization problems that are solved. Regularization is added to penalize the deviation of the solution of the optimization from simple heuristic reference policies that are described in the literature. The solution is biased towards a less complex solution, that is defined by these simple heuristics, while it can also diverge from these references when they are not adequate. PR-MPC behaves better than regularization-free MPC and better than bare heuristic control laws. It is challenging to tune heuristic control laws to accommodate various behaviors and regularization-free MPC may converge in bad local optima or may have slow convergence rates. Reference control inputs (contact forces and foothold positions) are used for policy regularization and also policy regularization is included in the cost function. However, this control framework has been tested on MIT Cheetah in simulation, and not on the real hardware. The NLP was solved online using *fmincon()* function of MATLAB.

In [31], the authors implemented nonlinear optimization-based Regularized Predictive Control (RPC) which is an extension of their previous work, described in [64]. PRC uses almost the same control framework as PR-MPC. The authors focused more on the techniques they employed to implement their controller on hardware. Some of the implementation details that are discussed are the direct transcription method used to derive the NLP, adaptive timestep segmentation, prediction delay compensation, the use of heuristic models, asynchronous solution filtering and controller gain tuning. In the experiments conducted on MIT Cheetah 3, the solution of the optimization problem was found using IPOPT solver at roughly 40-80 Hz.

In [65], the authors extend their work in [31], [64] by extracting more polished, regularizing legged locomotion heuristics with a data-driven method. The heuristics found with the proposed framework can be generalized, and thus they can be used in many different scenarios and are easily tunable. The authors improve the performance of RPC with these heuristics without changing the main controller architecture. Also, the robot that uses RPC and these heuristics can execute motions that could not be performed stably only with RPC. Data is extracted from offline simulations of RPC and is fitted with simple models to produce the heuristics. In the experiments conducted on MIT Mini Cheetah, the solution of the optimization problem was found using IPOPT solver at roughly 100-200 Hz.

In [66], a landing controller was implemented that accommodated a NMPC. The landing controller consists of three parts: an NMPC, a trained neural network, and a whole-body controller. The model predictive controller solves an NLP that was formulated using direct transcription, based on [70], that includes contact complementarity constraints and actuator torque and leg kinematic limits. These types of constraints, (e.g., actuation limits), can be imposed because the joint positions of the legs are part of the decision variables of the optimization problem. This problem is solved while the robot is airborne and its solution is the optimal GRFs, the landing body postures and the joint positions. When touchdown occurs, the optimized trajectory is used as a reference that gets tracked by a whole-body controller. This whole-body controller is a QP that is described in [33]. The neural network, trained using supervised learning, was used to warm start the OCP solver. The quality of the initial estimate given to a nonlinear optimization problem is critical as it can either draw the optimization towards undesirable local minima or even prevent it from finding a viable solution. Experiments on hardware were conducted using this control framework, on MIT Mini Cheetah. The solver

Artelys KNITRO [71] was used to solve the NLP online and was able to find solutions at roughly 10 Hz.

2 Real Time Simulation in Gazebo

2.1 Introduction

The main objective of this chapter, is to build a simulation model for the quadruped robot ARGOS using Gazebo simulator [1]. Gazebo is a robotics simulator that is commonly used in research for mobile ground robots like legged and wheeled robots. It allows the user to import robot models in the simulation environment from the Universal Robot Description Format (URDF) files. It is tightly integrated with ROS (Robot Operating System) [2] an open-source framework that contains libraries that define components of robots like actuators, sensors (e.g., camera, IMU, GPS) and control systems through its packages (e.g., ros_control). Such components can connect and communicate with each other using ROS tools (topics, messages). The interface with ROS provided by Gazebo simplifies the process of testing control software on a simulated robot and then transferring it onto the real-world robot [72]. The simulation of rigid, multibody dynamics is performed using the physics engine called Open Dynamics Engine (ODE or OpenDE). Out of all the physics engines supported by Gazebo, OpenDE was selected because it is the most compatible and smoothly integrated physics engine with Gazebo and ROS.

2.2 Modelling in Gazebo/OpenDE

OpenDE does not solely provide to the user the software library responsible for the simulation of rigid body dynamics in real time, which is called physics engine. It provides two additional software modules or engines that are necessary to conduct any simulation. These are a collision detection engine and a rendering engine. More details about the collision detection engine will be presented in chapter 2.2.7. The rendering engine is called OpenGL, is used by the OpenDE demos, is responsible for the generation of images from a model [73]. In Figure 2-1 the interaction of all these engines is shown during a standard simulation.

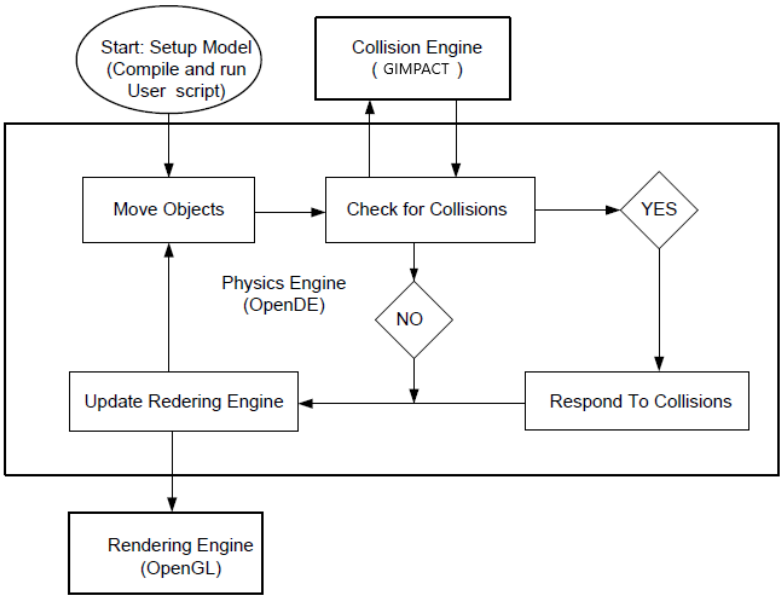


Figure 2-1. Flow diagram of a simulation that depicts the interaction of the engines that are responsible for multibody dynamics, for collision detection and for rendering [63].

2.2.1 Geometric and Inertial Parameters

Building the model of ARGOS in Gazebo demands the values of the geometric and inertial parameters of the robot. These values are contained in Table 2-1. The model of ARGOS in Gazebo is presented in Figure 2-2 and the model of ARGOS's legs in Gazebo is presented in Figure 2-3. To be more precise, the aforementioned values are the masses of the links of the robot, the inertia matrices of the links w.r.t. their CoM and some characteristic dimensions of the robot. The most important dimensions are the length and the width of the body and the lengths of the links of the leg and the manipulator. These values were extracted with the help of Solidworks environment which was utilized for the generation of the CAD files of the robot's components, which are the robot's body and legs. To insert the components of the robot into the simulation environment, they have to be described using the XACRO format [74]. XACRO is an XML macro language. It allows the user to construct shorter and more readable XML files by using macros. All the components of the robot are extracted as STL or DAE files that originate from their corresponding CAD files and are introduced in their XACRO description. The STL files are necessary for the visualization of the components of the robot in Gazebo and for the collision detection between links or between a link and the ground and for the computation of the contact points creates between bodies that come in contact.

Apart from the parameters pertinent to the geometric and inertial attributes of the robot, some additional parameters have to be specified that are associated with other facets of the simulation environment. These parameters are pertinent to the solver utilized for the numerical solution of the equations of motion (EoM), to the modelling of the collision and contact of the feet with the ground, to the joints of the robot, to the ground friction model and to the viscous joint damping model. The values of the aforementioned parameters are located in Table 2-2.

Table 2-1. Geometric and Inertial parameters of ARGOS in Gazebo.

Parameter	Value
Body	
Mass	29kg
Length	0.8m
Width	0.26m
Inertia Matrix	$\{I_{xx}=1.719612,\ I_{yy}=0.421794,\ I_{zz}=1.684144\}\ \text{kgm}^2$
Leg Roll	
Mass	1.1kg
Inertia Matrix	$\{I_{xx}=0.005776,\ I_{xy}=0.000769,\ I_{xz}=0.0,$ $I_{yy}=0.018940,\ I_{yz}=0.0,\ I_{zz}=0.018831\}\ \text{kgm}^2$
Upper Leg	
Distance between hip and knee joint (link length)	0.45m
Mass	0.2kg
Inertia Matrix	$\{I_{xx}=0.014896,\ I_{yy}=0.000647,\ I_{zz}=0.014572\}\ \text{kgm}^2$
Lower Leg	
Distance between knee joint and toe center (link length)	0.5885m
Mass	0.2kg

Inertia Matrix	$\{I_{xx} = 0.042404,\ I_{xy} = 0.0,\ I_{xz} = 0.0,$ $I_{yy} = 0.002461,\ I_{yz} = -0.005774,\ I_{zz} = 0.040486\}\,\mathrm{kgm}^2$
Manipulator	
Joint 1	
Distance between joint 1 and joint 2 (link length)	$0.25\,\mathrm{m}$
Mass	$0.2\,\mathrm{kg}$
Inertia Matrix	$\{I_{xx} = 0.006805,\ I_{yy} = 0.000266,\ I_{zz} = 0.006670\}\,\mathrm{kgm}^2$
Joint 2	
Distance between joint 2 and end effector (link length)	$0.53\,\mathrm{m}$
Mass	$0.5\,\mathrm{kg}$
Inertia Matrix	$\{I_{xx} = 0.020414,\ I_{yy} = 0.000169,\ I_{zz} = 0.020362\}\,\mathrm{kgm}^2$

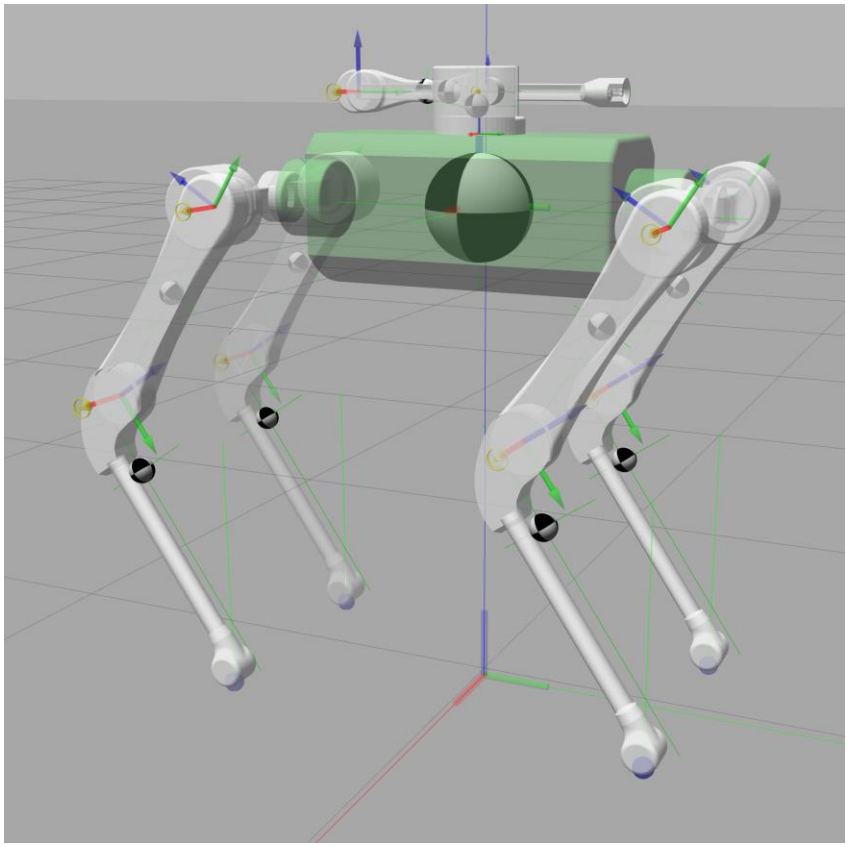


Figure 2-2. Model of ARGOS in Gazebo.

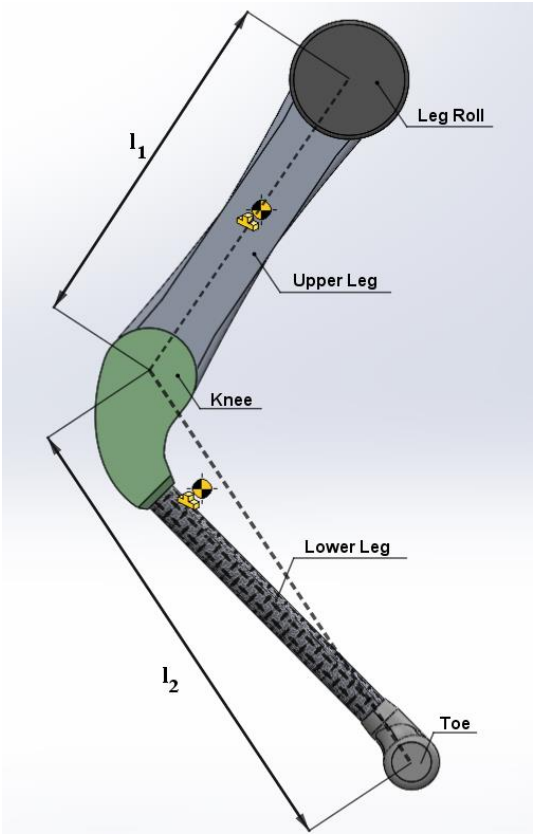


Figure 2-3. Model of ARGOS's leg in Gazebo.

2.2.2 Solver Parameters

Some features of the physics engine utilized by Gazebo are presented in this section. The solver features as well as the parameters of the model of ARGOS in Gazebo that are discussed, are described in detail in [75], [76], [77]. In general, rigid body dynamic simulators attempt to solve the constrained Newton-Euler equation for systems of rigid bodies. OpenDE poses the constrained Newton-Euler equations as a Linear Complementarity Problem (LCP) in the maximal coordinate system with the use of constraints (constraint-based LCP) to ensure that forces satisfy non-interpenetration and joint-constraints. More information about that LPC will be presented in the next paragraph. OpenDE utilizes fixed time step solvers. The selected time step Δt $\left(max\_step\_size\right)$ is equal to 1msec. The reason behind the selection of a time step that lies in the range of milliseconds, is the fact that OpenDE uses Euler integration. Such low-order integration schemes require small time steps to achieve satisfying accuracy. Another parameter necessary in OpenDE is the $real\_time\_update\_rate$, that expresses the frequency at which the simulation time steps are advanced in time. Its selected value is equal to 1kHz. The product $\left(max\_step\_size\right)\cdot\left(real\_time\_update\_rate\right)$ defines the upper bound of $real\_time\_factor$ parameter. If $real\_time\_factor<1$ the simulation is slower than real time. The final physics engine parameter presented is $max\_contacts$, that expresses the maximum possible number of contacts to be generated between two bodies that are in contact. This parameter sets an upper bound to the maximum number of potential contact points between two bodies, useful feature especially for face-to-face collisions, with a potential sacrifice to the accuracy of the computations. This feature is also particularly useful when the bodies that come in contact are modeled using 3D meshes as collision models. Furthermore, this parameter allows the user to allocate a fixed amount of memory at the beginning of the

simulation necessary for contact storage. Therefore, no memory reallocation would be necessary since the memory allocated is adequate for storing the maximum possible number of contacts. In this work, the value selected is equal to 1. More details about the contact points and the collision models will be presented in chapter 2.2.7.

In addition to the physics engine parameters presented before, some features of the solvers available in this physics engine are also presented. OpenDE provides two types of solvers to solve the constraint-based LCP, world and quick. World step uses a direct (pivoting) method, that is based on an extension of Danzig's algorithm [78], which, according to [79], is Lemke's algorithm [80]. Quick step uses an iterative method, called projected Gauss-Seidel (PGS) with Successive Over-Relaxation (SOR) [81], [82]. World step provides more accurate solutions but exhibits convergence difficulties since it is not always able to compute a solution and exhibits slower convergence rate compared to quick. Quick step converges easier, but its accuracy depends on the number of iterations. In general, the higher the number of solver iterations (in every time step), the more accurate the results will be but the slower the convergence will be. The parameter $iters$ that specifies the number of iterations is used only by quick solver. When quick solver is selected, the SOR parameter must also be specified. The Over-Relaxation parameter is a number larger than 1 and is used to increase the convergence rate of an iterative method.

In this work, the world step solver was selected since, after many trials, it was determined that it provides more accurate results compared to quick. To be more precise, simulations were conducted to decide which solver provides the most accurate results. In these simulations, the quadruped fell from a small height to the ground and eventually stopped moving completely. After running these simulations, the following results were recorded: the joint angular velocities computed by world step (order of magnitude $10^{-10}\,\mathrm{rad/s}$) were closer to zero than the ones that quick step computed (order of magnitude $10^{-2}\,\mathrm{rad/s}$). Moreover, the expected outcome of this experiment is that after some seconds of simulation time have passed, the responses of the joint angles and the responses of the torques of the actuators of the joints should converge to steady values. This outcome was observed only when world step was employed. When quick step was employed, even after many minutes of simulation time have passed, the responses of the joint angles and motor torques did not converge to steady values. In general, quick solver exhibits poor accuracy for near-singular systems [76]. Near-singular systems can occur when using high-friction contacts, motors, or certain articulated systems. For instance, a quadruped robot sitting on the ground may be near-singular. There are certain ways that can help tackle the inaccuracy problem from which the quick solver suffers. Some of these are the following:

- Increase CFM.
- Reduce the number of contact points in the system (i.e., by using the minimum possible number of contact points for the feet of the robot).
- Avoid excessive friction in the contacts.
- Avoid closed kinematic loops, which is inevitable in the case of legged robots.
- Increase the number of solver iterations, which in case where the system is near-singular is not significantly helpful.

Since the aforementioned alternatives were tested but the accuracy of quick step solver was not improved, world step solver was the final choice.

2.2.3 Rigid Body Dynamics

OpenDE represents rigid body states with maximal coordinates, in which each rigid body has six degrees of freedom, and articulation and contact constraints are enforced by adding constraint equations. The methodology that OpenDE uses to extract the EoM, is thoroughly described in [79].

Unconstrained Rigid Body Dynamics

All vectors utilized are expressed in the world frame, unless otherwise specified. Each rigid body has a body-fixed coordinate frame attached to its CoM, whose position is denoted as $\mathbf{x}$, and its orientation denoted as $\mathbf{q}$, that is a quaternion (quaternion orientation). These quantities evolve in time according to the rigid body kinematic equations:

$$\mathbf{x}_t = \dot{\mathbf{x}} \tag{2-1}$$

$$\mathbf{q}_t = \frac{1}{2}\boldsymbol{\omega} \otimes \mathbf{q} \tag{2-2}$$

where $\dot{\mathbf{x}}$ is the linear velocity of the body's CoM, $\boldsymbol{\omega}$ is the body angular velocity and where $\otimes$ denotes quaternion product. These velocities evolve in time according to the Newton-Euler EoM:

$$m\dot{\mathbf{x}}_t = \mathbf{f} \tag{2-3}$$

$$\mathbf{L}_t = \boldsymbol{\tau} \tag{2-4}$$

where m is the body mass, $\mathbf{f}, \boldsymbol{\tau}$ are the net force and torque acting on the body and $\mathbf{L}$ is the angular momentum of the body. The angular momentum can be calculated using the expression: $\mathbf{L} = \mathbf{I} \cdot \boldsymbol{\omega}$ where $\mathbf{I}$ is the inertia tensor w.r.t the world frame that is given by the following expression: $\mathbf{I} = \mathbf{R} \cdot \mathbf{D} \cdot \mathbf{R}^T$ where $\mathbf{R}(\mathbf{q})$ is the rotation matrix that describes the orientation of the body-fixed coordinate frame w.r.t the world frame and $\mathbf{D}$ is the body-frame inertia tensor. Therefore, equations (2-3) and (2-4), for a single unconstrained body, can be written in the following form:

$$\begin{bmatrix} m \cdot \boldsymbol{\delta} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{x}} \\ \dot{\boldsymbol{\omega}} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \boldsymbol{\tau} - \boldsymbol{\omega} \times \mathbf{I}\boldsymbol{\omega} \end{bmatrix} \tag{2-5}$$

where $\boldsymbol{\delta}$ is the identity matrix.

In case of a system that is comprised of multiple rigid bodies, the following augmented variables are defined for every rigid body b: the velocity vector $\mathbf{v}_b = \begin{bmatrix} \dot{\mathbf{x}}_b^T & \boldsymbol{\omega}_b^T \end{bmatrix}^T$, the block diagonal mass matrix $\mathbf{m}_b = diag(m_b \boldsymbol{\delta}, \mathbf{I}_b)$ and the vector that contains the forces and torques acting on the bodies (effort vector) $\mathbf{e}_b = \begin{bmatrix} \mathbf{f}_b^T & (\boldsymbol{\tau}_b - \boldsymbol{\omega}_b \times \mathbf{I}_b \boldsymbol{\omega}_b)^T \end{bmatrix}^T$. Consequently, the equation (2-5) can be written in the following form:

$$\mathbf{m}_b \dot{\mathbf{v}}_b = \mathbf{e}_b \tag{2-6}$$

In case of a system that is comprised of N rigid bodies, the following system variables are defined: the velocity states $\mathbf{v} = \begin{bmatrix} \mathbf{v}_1^T & \mathbf{v}_2^T & \dots & \mathbf{v}_N^T \end{bmatrix}^T$, the block diagonal system mass matrix $\mathbf{M} = diag(m_1, m_2, \dots, m_N)$ and the system effort vector $\mathbf{E} = \begin{bmatrix} \mathbf{e}_1^T & \mathbf{e}_2^T & \dots & \mathbf{e}_N^T \end{bmatrix}^T$. Thus, the unconstrained dynamics described in equation (2-6), can be rewritten in the following form:

$$\mathbf{M}\dot{\mathbf{v}} = \mathbf{E} \tag{2-7}$$

Articulation and Contact Constraints

Articulations that connect bodies with each other and contacts between bodies, are modeled using a set of nonlinear constraints on the body position and orientation of the rigid bodies that are part of the system. Articulations make use of equality constraints that have the form $h_e(\mathbf{x},\mathbf{q})=0$ and contacts make use of inequality constraints that have the form $h_i(\mathbf{x},\mathbf{q})\geq 0$, both of which are position constraints.

To avoid nonlinearities, the position constraints are differentiated to extract linear velocity constraints [79] that have the form:

$$\mathbf{J}\mathbf{v}=\mathbf{c} \tag{2-8}$$

where $\mathbf{J}$ is the constraint Jacobian matrix with one row for every degree of freedom the constraint removes from the system, and $\mathbf{c}$ a right-hand side vector. For articulation constraints, that are equalities, $\mathbf{c}$ obeys the following relationship: $\mathbf{c}=\mathbf{c}_e=0$. For contact constraints, that are inequalities, $\mathbf{c}$ obeys the following relationship: $\mathbf{c}=\mathbf{c}_i\geq 0$. The procedure that should be followed to construct these velocity constraints is thoroughly described in [83]. The velocity constraints get appended to the EoM using a vector of Lagrange multipliers $\boldsymbol{\lambda}$ as:

$$\mathbf{M}\dot{\mathbf{v}}=\mathbf{E}+\mathbf{J}^T\boldsymbol{\lambda} \tag{2-9}$$

where, according to [84], vector $\mathbf{E}$ is equal to the sum the external forces applied to the system, $\mathbf{f}_{\text{ext}}$, with the joint viscous damping forces, $\mathbf{f}_{\text{damp}}$. The term $\mathbf{J}^T\boldsymbol{\lambda}$ is equal to the constraint forces applied to the system, $\mathbf{f}_{\text{constraint}}$. So equation (2-9) can also be written in the form:

$$\mathbf{M}\dot{\mathbf{v}}=\mathbf{f}_{\text{ext}}+\mathbf{f}_{\text{damp}}+\mathbf{f}_{\text{constraint}} \tag{2-10}$$

Discretization and Solution of the Dynamics

Equation (2-9) has to be discretized over a time interval Δt using first-order Euler integration method (forward or explicit Euler integration), for the discretization of the time derivatives, which is the following:

$$\dot{\mathbf{v}}=\frac{\mathbf{v}^{n+1}-\mathbf{v}^n}{\Delta t} \tag{2-11}$$

Rearranging the terms of equation (2-11) and combining it with equation (2-9) results in the following expression:

$$\begin{bmatrix}(1/\Delta t)\mathbf{M} & -\mathbf{J}^T\\ \mathbf{J} & 0\end{bmatrix}\begin{bmatrix}\mathbf{v}^{n+1}\\ \boldsymbol{\lambda}\end{bmatrix}=\begin{bmatrix}(1/\Delta t)\mathbf{M}\mathbf{v}^n+\mathbf{E}\\ \mathbf{c}\end{bmatrix} \tag{2-12}$$

where $\boldsymbol{\lambda}\geq\mathbf{0}$ and $\mathbf{c}\geq\mathbf{0}$ for unilateral contacts. Assuming that the constraints are satisfied implicitly at the next time step $(t+\Delta t)$, $\mathbf{J}\mathbf{v}^{n+1}=\mathbf{c}$, the Lagrange multipliers $\boldsymbol{\lambda}$ are calculated using the equation (2-13), that is derived by left multiplying the equation (2-12) with the term $\mathbf{J}\mathbf{M}^{-1}$. This equation has the following form:

$$\left[\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T\right]\boldsymbol{\lambda}=\frac{\mathbf{c}}{\Delta t}-\mathbf{J}\left[\frac{\mathbf{v}^n}{\Delta t}+\mathbf{M}^{-1}\mathbf{E}\right] \tag{2-13}$$

The equation (2-13) is a system of linear equations that can be rewritten in the following form:

$$\mathbf{A}\boldsymbol{\lambda}=\mathbf{b} \tag{2-14}$$

where

$$\mathbf{A} = \mathbf{J}\mathbf{M}^{-1}\mathbf{J}^{T}, \qquad \mathbf{b} = \frac{\mathbf{c}}{\Delta t} - \mathbf{J}\left[\frac{\mathbf{v}^{n}}{\Delta t} + \mathbf{M}^{-1}\mathbf{E}\right] \tag{2-15}$$

Equation (2-13) is solved for the unknown Lagrange multipliers $\boldsymbol{\lambda}$ which are necessary for forward dynamics. This system can be solved using the algorithms that were referred in section 2.2.2, which are either Dantzig or PGS.

Having the vector $\boldsymbol{\lambda}$ acquired, the velocities $\mathbf{v}^{n+1}$ are calculated using equation (2-12). The positions $\mathbf{x}^{n+1}$ and the orientations $\mathbf{q}^{n+1}$ are computed by integrating equations (2-1), (2-2) respectively, using the velocity at the next time step $\left(t+\Delta t\right)$, $\mathbf{v}^{n+1}$ (backward or implicit Euler integration), to give semi-implicit stability [79]. The resulting integrations schemes are the following:

$$\mathbf{x}^{n+1} = \mathbf{x}^{n} + \Delta t\dot{\mathbf{x}}^{n+1} \tag{2-16}$$

$$\mathbf{q}^{n+1} = \mathbf{q}^{n} + \frac{\Delta t}{2}\boldsymbol{\omega}^{n+1} \otimes \mathbf{q}^{n} \tag{2-17}$$

This should be combined with a renormalization $\mathbf{q}^{n+1} := \mathbf{q}^{n+1} / \left\|\mathbf{q}^{n+1}\right\|$ to ensure that the result is a unit quaternion. The entire integration scheme is called semi-implicit Euler integration, due to the combination of both forward and backward Euler integration rules.

The Lagrange multipliers that correspond to inequality constraints, are initially calculated using equation (2-13) and these intermediate values are projected afterwards into their corresponding solution space, depending on the type of constraint that corresponds in each situation. This solution projection [84], is conducted in the case of contact and frictional constraints. For contact constraints, the constraint force that corresponds to the direction normal to the contact surface,

$$\mathbf{f}_{\mathbf{constraint},i} = \mathbf{J}_{i}^{T}\boldsymbol{\lambda}_{i}, \qquad i = 1,\ldots,N_{\text{constraints}} \tag{2-18}$$

is necessary to push the bodies apart from each other (no stiction between the bodies), means that,

$$\boldsymbol{\lambda}_{i} \geq 0 \tag{2-19}$$

where $\mathbf{J}_{i}^{T}$ denotes the transpose of the i-th column of matrix $\mathbf{J}$ and $N_{\text{constraints}}$ is the number of all the constraints present in a system. Frictional constraints will be examined in more detail in section 2.2.5.

2.2.4 Joint Constraints Parameters

When a joint (constraint) attaches two bodies, they are supposed to have certain positions and orientations relative to each other. Nevertheless, it is possible for the two bodies to be in positions where the conditions that the constraints represent are not satisfied. This “joint error” may occur for the following two reasons:

- The user sets the position/orientation of one body without correctly setting the position/orientation of the other body.
- During the simulation, errors can creep in that result in the bodies drifting away from their required positions and not lining up properly, as is shown in Figure 2-4.

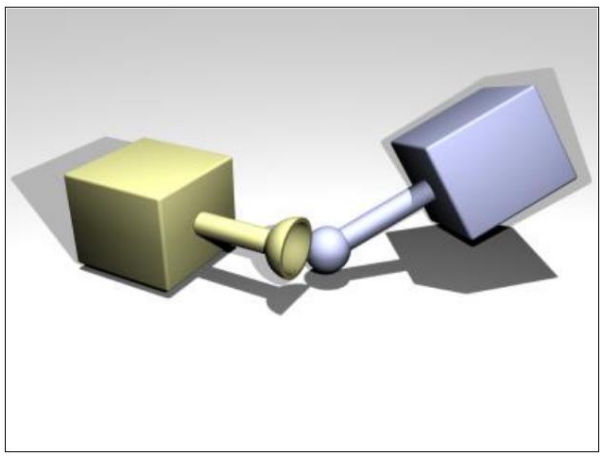


Figure 2-4. An example of "joint error" in a ball and socket joint [76].

It is possible to reduce the joint error. At every simulation step, each joint applies a force to its bodies, that aims to bring them back into correct alignment. This force is controlled by the error reduction parameter (ERP), and it has a value between 0 and 1.

To be more specific, according to [79], OpenDE adds an additional term to the constraint equation, which is a position constraint correction term. The position constraint error at the nth time step $\mathbf{h}^n$, is calculated for every constraint and at every time step. This term is combined with ERP and then added to the velocity constraint equation. Thus, the constraint equation takes the following form:

$$\mathbf{Jv} + \frac{ERP}{\Delta t}\mathbf{h}^n = \mathbf{c} \tag{2-20}$$

This term can be regarded as a form of Baumgarte stabilization [85]. Additionally, the constraint error correction term can be also added in equation (2-13), which takes the form:

$$\left[\mathbf{JM}^{-1}\mathbf{J}^T\right]\boldsymbol{\lambda} = \frac{\mathbf{c}}{\Delta t} - \frac{ERP}{\Delta t^2}\mathbf{h}^n - \mathbf{J}\left[\frac{\mathbf{v}^n}{\Delta t} + \mathbf{M}^{-1}\mathbf{E}\right] \tag{2-21}$$

ERP specifies what portion of joint error will be corrected during the next step of the simulation. If $ERP = 0$, then no correcting force will be applied to the bodies, and they will eventually drift apart. If $ERP = 1$, then the simulation will attempt to correct the entirety of joint error during the next step of the simulation. However, in practice it is not recommended to use $ERP = 1$, since due to various internal approximations joint error will not be entirely corrected. Apart from that, it may also lead some systems to instability. In practice, the recommended value for ERP lies in the interval $\left[0.1 \quad 0.8\right]$.

A second parameter used for the joint constraints called the constraint force mixing parameter (CFM) will be presented here. Most articulation constraints are "hard", meaning that conditions that the constraints represent must never be violated. For example, in case of a ball and socket joint, the ball must always lie inside the socket. Nevertheless, these constraints may be violated in practice because of errors that are inadvertently introduced into the system. However, these errors can be corrected with the use of ERP.

Some constraints are "soft", meaning that they are designed to be violated. For instance, the contact constraint formed by two bodies that come in contact and should not penetrate each other is "hard". However, a contact constraint formed by two bodies that come in contact

and should be able to deform and allow some penetration upon contact, is “soft”. The distinction between “hard” and “soft” constraints is done using ERP and CFM parameters.

As it was mentioned in chapter 2.2.3 the constraint equation of every joint can be written in the form:

$$\mathbf{Jv} = \mathbf{c} \tag{2-22}$$

Ta the next time step, the vector $\boldsymbol{\lambda}$ is calculated, that has the same dimensions as vector $\mathbf{c}$. The vector $\boldsymbol{\lambda}$ is calculated in such a way that the forces $\mathbf{f}_{\text{constraint}}$ acting on the bodies of the joint to enforce the constraint are the following:

$$\mathbf{f}_{\text{constraint}} = \mathbf{J}^T\boldsymbol{\lambda} \tag{2-23}$$

In OpenDE the constraint equation has the following form:

$$\mathbf{Jv} = \mathbf{c} + \mathbf{CFM}\cdot\boldsymbol{\lambda} \tag{2-24}$$

where $\mathbf{CFM}$ is a square, diagonal, and positive definite matrix whose elements are comprised of the parameter CFM. A positive $\mathbf{CFM}$ allows the aforementioned constraint to be violated by an amount proportional to the product $\mathbf{CFM}\cdot\boldsymbol{\lambda}$, where the restoring force $\boldsymbol{\lambda}$ is necessary to enforce the constraint. In the approach that OpenDE adopts, the term $(1/\Delta t)\mathbf{CFM}\cdot\boldsymbol{\lambda}$ is added to the right hand side part of equation (2-21), and by calling the right hand side part of the equation, rhs, equation (2-21) can be rewritten as follows:

$$\left[\mathbf{JM}^{-1}\mathbf{J}^T + \frac{\mathbf{CFM}}{\Delta t}\right]\boldsymbol{\lambda} = rhs \tag{2-25}$$

Thus, $\mathbf{CFM}$ adds additional terms to the diagonal of the original system matrix. If $CFM = 0$, then the constraint will be “hard”. If $CFM > 0$, then the constraint will be “soft” and as CFM increases, so does the softness of the constraint. One additional benefit of positive CFM is that the system moves away from singularities and thus the factorizer accuracy is being improved [76]. The parameters ERP, CFM can be set for each rotational joint of the robot separately, or they can be set simultaneously for all the rotational joints of the robot by using global ERP, CFM parameters. For the articulation constraints of the robot, which are rotational joints, the selected values are $ERP = 0.8$ and $CFM = 10^{-3}$. More information about the “joints” that correspond to the contact points of the feet of the quadruped with the ground will be presented in chapter 2.2.6.

To select the proper ERP, CFM for the articulation constraints simulations were conducted. In these simulations, the quadruped, with legs with 2 Degrees of Freedom (DoF) (abduction angle was fixed and was equal to zero), fell from a small height to the ground and eventually stopped moving completely. Running these simulations, with friction present at the contacts of the feet with the ground, the following were observed: when $CFM = 0$, the robot’s body roll angle was not close enough to zero. To be more precise, for $CFM = 0$, the body roll angle varied, depending on the controllers’ gains, in the order of magnitude of $10^{-4} \sim 10^{-5}\,\text{rad}$. When $CFM > 0$ and more precisely when CFM varied in the order of magnitude of $10^{-3} \sim 10^{-10}$, then the body roll angle varied in the order of magnitude of $10^{-7} \sim 10^{-11}\,\text{rad}$. Therefore, the use of a non-zero CFM significantly improves the accuracy of the computations. Some additional simulations were also conducted. In these simulations, a table fell from a small height to the ground and eventually stopped moving completely. The table was a rigid body with legs that had no joints. When it was standing still on the ground, the body roll angle was almost zero (order of magnitude $10^{-18}\,\text{rad}$) even for $CFM = 0$. Therefore,

the non-zero body roll angle is caused by the joint error, present due to the articulation constraints, in the simulation. If legs with joints were added to the table, then the body roll angle moved away from zero. The body roll angle moved even further from zero when the number of leg joints was increased. Consequently, this problem present in articulated systems can be significantly reduced by tuning appropriately CFM .

2.2.5 Friction Parameters

The friction model utilized to model the friction between two or more bodies that come in contact is the Coulomb friction model, that poses a relationship between normal and the tangential component of the friction force at the contact point. This relationship is the following:

$$\left|\mathbf{f}^t\right| \le m_u \left|\mathbf{f}^n\right| \tag{2-26}$$

where $\mathbf{f}^n, \mathbf{f}^t$ are the normal and tangential components of the contact force, and m_u is the friction coefficient. This equation defines a friction cone with axis parallel to the vector $\mathbf{f}^n$ and with vertex the contact point. This friction cone is depicted in Figure 2-5. If the total friction force vector lies within the friction cone, then the contact is in “sticking mode”, which means that the friction force is enough to prevent the bodies that are in contact from moving w.r.t each other. If the total friction force vector lies on the surface of the friction cone, then the contact is in “sliding mode”, which means that the friction force is typically not large enough to prevent the contacting surface the bodies that are in contact from sliding. Ergo the coefficient m_u specifies the maximum ratio of tangential to normal force that will not cause sliding.

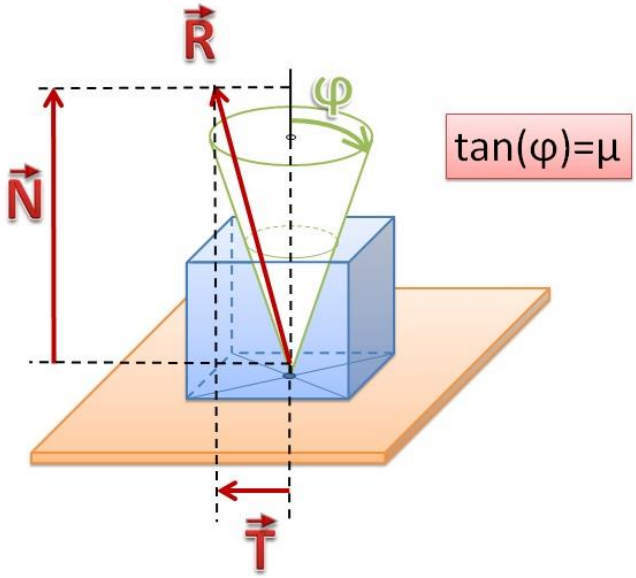


Figure 2-5. Friction Cone [86].

In OpenDE, according to [79], the frictional constraint equations are also written as velocity constraints in the following form:

$$\mathbf{J}_{fric}\mathbf{v} = \mathbf{c}_{fric} \tag{2-27}$$

The direction the tangential friction force component $\mathbf{f}^t$, is referred to as $\mathbf{j}_T$, and should satisfy the maximum dissipation principle [87]. The Lagrange multipliers that correspond to the frictional constraints, $\boldsymbol{\lambda}_T$, are projected into a corresponding solution space based on Coulomb’s friction law of equation (2-26).

OpenDE utilizes friction models that are approximations of the friction cone, for more efficient calculations since they tend to be much faster than the friction cone [79]. In such models the direction of maximum dissipation is not computed, but the frictional constraints are split into two or more vectors that span over the contact surface manifold. In case where two spanning directions are chosen, the friction cone is approximated by a pyramid. The friction pyramid, displayed in Figure 2-6, is utilized in this work. Consequently, friction's behavior will be anisotropic, and thus orientation based artifacts might be introduced [84]. These artifacts do not occur when the friction cone is utilized. In this model, two friction coefficients μ_u, μ_{u2} must be specified that correspond to the two friction directions of the friction pyramid. These two directions coincide with the directions of the axes of the global coordinate system (world frame). When the friction pyramid is used, the frictional constraint equations have the following form:

$$\mathbf{J}_{fric}\mathbf{v} = \begin{bmatrix} \cdots & \mathbf{j_N} & \cdots & \mathbf{j_{T1}} & \cdots \\ \cdots & \mathbf{j_N} & \cdots & \cdots & \mathbf{j_{T2}} \end{bmatrix} \begin{bmatrix} \cdots & \mathbf{v_N}^T & \cdots & \mathbf{v_{T1}}^T & \mathbf{v_{T2}}^T \end{bmatrix}^T = \mathbf{c}_{fric} \tag{2-28}$$

with the Lagrange multipliers $\boldsymbol{\lambda}_{\mathbf{T1}}, \boldsymbol{\lambda}_{\mathbf{T2}}$ that correspond to the two directions of the pyramid, being projected into their corresponding solution spaces, at each solver iteration, based on the expressions:

$$\left|\mathbf{f}^{t1}\right| \le \mu_{u1} \left|\mathbf{f}^{n}\right| \tag{2-29}$$

$$\left|\mathbf{f}^{t2}\right| \le \mu_{u2} \left|\mathbf{f}^{n}\right| \tag{2-30}$$

OpenDE makes an additional approximation to this model which is the following: firstly, OpenDE calculates the normal components of the contact forces assuming that the contacts are frictionless. Then it computes the bounds of the tangential component of the friction force $\mathbf{f_m}$ from the expression:

$$\mathbf{f_m} = \mu_u \left|\mathbf{f}^{n}\right| \tag{2-31}$$

and afterwards solves the equations of the system using these fixed bounds. The value selected for the coefficients μ_u, μ_{u2} is 1.2.

Apart from the coefficients μ_u, μ_{u2}, OpenDE allows the user to specify the direction of the first friction direction of the pyramid at which the friction coefficient μ_u corresponds to. This direction is specified with the unit vector $\mathbf{f_{dir1}}$, that is shown in Figure 2-6. This vector is defined w.r.t. the local reference frame of each of the bodies that come in contact. The vector that specifies the second friction direction of the pyramid $\mathbf{f_{dir2}}$, is equal to the cross product of the unit vector along the normal direction to the contact surface with the vector $\mathbf{f_{dir1}}$. In case where $\mathbf{f_{dir1}} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$ for both bodies that come in contact, then the two directions of the friction pyramid will be aligned with the axes of the world frame. This means that the direction of $\mathbf{f_{dir1}}$ is the same as the direction of the positive semi-axis x, and the direction of $\mathbf{f_{dir2}}$ is the same as the direction of the positive semi-axis y. In case where a nonzero $\mathbf{f_{dir1}}$ is utilized, then its value should be defined in only one of the two bodies that come in contact. If it is set for both bodies, then the behavior of the friction model will be undefined.

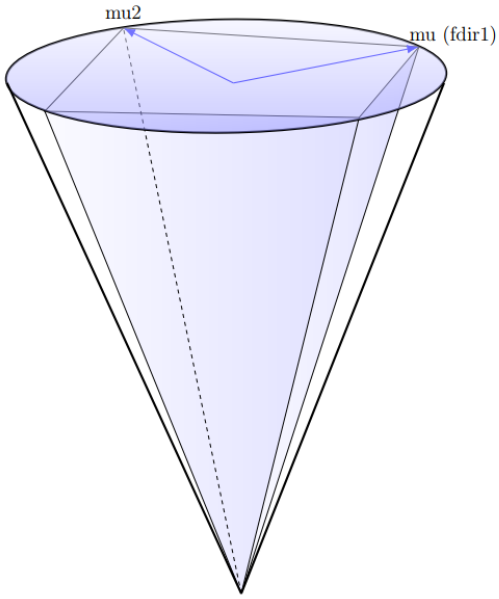


Figure 2-6. Friction cone approximation with square pyramid [75].

Finally, torsional friction [88] must also be included in the model in addition to the translational friction that has already been defined. The importance of torsional friction becomes apparent with the following example: consider a sphere that is in contact with the ground, at only one contact point, and rotates about an axis that passes through that contact point. More information about contact and collision modelling of bodies with the ground in Gazebo will be presented in chapters 2.2.6, 2.2.7. Since the axis of rotation passes through the contact point with the ground, the linear velocity of that point will be equal to zero while its angular velocity will be nonzero. As a result, the translational friction acting on the sphere is zero and thus the sphere will rotate without being decelerated by friction. Therefore, torsional friction must be defined and included in the model. The torsional friction torque T_t that acts on a body and decelerates it, is given by the following expression:

$$T_t = \frac{3\pi}{16} \cdot a \cdot coefficient \cdot \mathbf{f}^n \tag{2-32}$$

where $coefficient$ is the coefficient of torsional friction, which is usually equal to the translational friction coefficients m_u, m_{u2}. In this work, this coefficient is equal to 0.0. The radius a is the radius of the contact patch created between the two surfaces that come in contact as is shown in Figure 2-7. The value of that radius is given by the expression:

$$a = \sqrt{R \cdot d} \tag{2-33}$$

where R is the surface radius at the contact point and d is the contact penetration depth. The procedure that must be followed to derive the equation (2-33) is analyzed in [89].

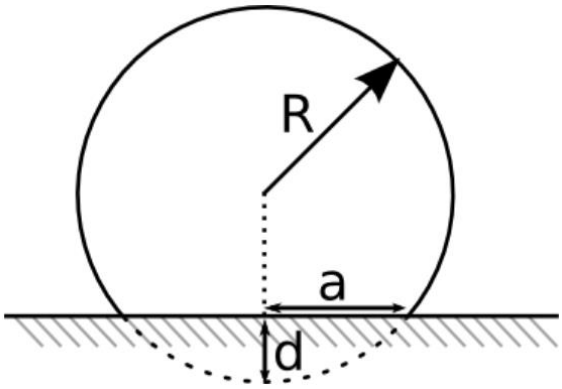


Figure 2-7. Sphere geometry at the contact point with the ground [88].

As seen in equation (2-32), torsional friction, unlike translational friction, does not depend solely on the friction coefficient and the normal contact force component. It also depends on the area of contact formed between two bodies that come in contact. OpenDE supports two different parameterizations of the contact area. With the first one, the user has to define the surface radius a needed for the calculation of T_t. However, with this parameterization, the surface radius a will no longer depend on d, but will always have a constant value and equal to the value that the user has provided. With the second one, the user has to define the surface radius R and thus the patch radius a is computed using equation (2-33) and the contact depth d. The value of R, was set equal to the radius of the cylindrical part of the lower legs of the robot. With the first parameterization, since the user has defined the patch radii a_1, a_2 of the two bodies that are in contact, the equivalent surface radius a is calculated as follows:

$$a = \max(a_1, a_2) \tag{2-34}$$

with the second parameterization, since the user has defined the surface radii R_1, R_2 of the two bodies that are in contact, the equivalent surface radius R is calculated as follows:

$$R = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}} \tag{2-35}$$

In case where one of the two bodies is the ground (plane), then its surface radius is set equal to infinity (inf).

2.2.6 Contact Constraints Parameters

The contacts between bodies are handled by the physics engine as joints (contact joints), as is shown in Figure 2-8. When two bodies or a body and the ground come in contact, then that contact is represented with one or more contact points. A penetration depth corresponds at each contact point that expresses the depth to which the two bodies inter-penetrate each other. Also, a unit vector, that is perpendicular to the contact surface, corresponds at each contact point as well. Representing contact with contact points is just an approximation of the real phenomenon. The use of contact patches or surfaces is a much more accurate approximation, but it is extremely challenging to employ such models in a high-speed simulation software.

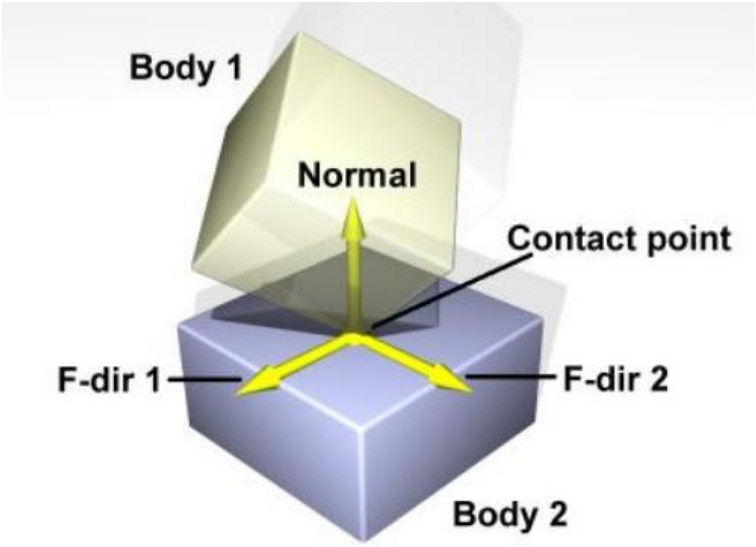


Figure 2-8. Contact Joint [76].

As it was mentioned before, the contact of the feet with the ground can be modeled if the appropriate values of ERP, CFM are selected firstly. The contact modeling between bodies can be implemented by using a spring-damper model. Many variants of this model exist. The spring-damper model utilizes a spring (linear or nonlinear) connected in parallel with a damper (linear or nonlinear), whose constants k_p, k_d must be specified. These constants depend on the materials from which the two bodies that come in contact are constructed. Since the user has defined the spring constants $k_{p,1}, k_{d,1}$ and the damper constants $k_{p,2}, k_{d,2}$ of the two bodies that come in contact, the equivalent spring and damper constants $k_{p,eq}, k_{d,eq}$ are calculated as follows: assume that $k_{p,f}, k_{d,f}$ and $k_{p,g}, k_{d,g}$ are the spring and damper constants of the two bodies that come in contact, which are the foot and the ground respectively. According to the Gazebo source code, the two springs are assumed to be connected in series and the two dampers are assumed to be connected in parallel. Consequently, the equivalent spring and damper constants $k_{p,eq}, k_{d,eq}$ are the following:

$$k_{p,eq} = \frac{1}{\frac{1}{k_{p,f}} + \frac{1}{k_{p,g}}} \tag{2-36}$$

$$k_{d,eq} = k_{d,f} + k_{d,g} \tag{2-37}$$

However, it is possible to map the ERP, CFM parameters into equivalent spring and damper constants k_p, k_d with the following expressions:

$$ERP = \frac{k_p \Delta t}{k_p \Delta t + k_d} \tag{2-38}$$

$$CFM = \frac{1}{k_p \Delta t + k_d} \tag{2-39}$$

The procedure that must be followed to derive the equations (2-38), (2-39) is analyzed in [90]. Any spring-damper system can be implemented by tuning appropriately ERP, CFM. It is important to note that the spring-damper system solution is obtained in this way is obtained implicitly as part of the overall LCP system. Therefore, the numerical stiffness problems that

are present in explicit spring-damper systems are nonexistent here [79]. The constants that were chosen for the material of the robot's foot are the following: $k_{p,f} = 300000\,\text{N/m}$, $k_{d,f} = 1000\,\text{Ns/m}$ the constants that were chosen for the material of the ground are the following: $k_{p,g} = 300000\,\text{N/m}$, $k_{d,g} = 1000\,\text{Ns/m}$. These constants correspond to rubber, which is the material from which the feet and the ground are constructed (CSL lab's treadmill). The equivalent spring and damper constants are calculated using (2-36), (2-37) and are the following: $k_{p,eq} = 150000\,\text{N/m}$, $k_{d,eq} = 2000\,\text{Ns/m}$.

Table 2-2. Solver, articulation constraints, friction and contact constraints parameters utilized for the model of ARGOS in Gazebo.

Parameter	Value
Physics Engine	
type	*ode*
max_step_size	0.001sec
real_time_update_rate	1000Hz
max_contacts	1
Solver	
type	*world*
Constraints	
CFM	10^{-3}
ERP	0.8
Friction	
friction_model	*pyramid_model*
m_u	1.2
m_{u2}	1.2
$\mathbf{f}_{\mathbf{dir}1}$	$[0 \quad 0 \quad 0]$
torsional / coefficient	0.0
Contacts	
k_p	300000N/m
k_d	1000Ns/m

2.2.7 Collision Detection

The collision detection engine checks to find out if any objects have collided. If so, 'virtual' joints (contact joints) are created for every collision point and are added to points of contact containing penetration depth. This is often the costliest step w.r.t the processing time. In OpenDE the collision detection engine GIMPACT [91] is being utilized. To perform collision detection in Gazebo, a collision model has to be associated with each body. This model can be an STL file, which is a 3D unstructured mesh that describes the geometry of a body, or a simple geometric primitive provided by ROS libraries. Some of these simple collision primitives are boxes, cylinders, spheres, convex polyhedra and planes.

Initially, the collision models used for the lower legs were the meshes contained in STL files that were extracted from the CAD files that describe their geometry. The collision model utilized for the ground was a plane. To validate the accuracy of these collision models, simulations were conducted. In these simulations, the quadruped fell from a small height to the ground and eventually stopped moving completely. The resulting contact points were not

stable, meaning that their coordinates w.r.t. the inertia (or world frame) were time varying even though the quadruped was standing still on the ground. This phenomenon is called flickering/jittering contact and is a common artifact of OpenDE that has not been resolved yet [92]. The poor contact handling, especially between unstructured meshes, induces simulation artifacts like jitter, phantom impulses, or even divergence ("blowing up"), as is shown in Figure 2-9. This artifact is even more intense when the bodies that come in contact are both modeled using unstructured meshes [93]. Due to the flickering artifact, high frequency signals that resemble "noise" are superimposed on the low frequency signals of physical quantities like the motor torques. Consequently, the time responses of these physical quantities get distorted. According to [92], one check that can be performed is the following one: run the simulation with the physics engine deactivated. The deactivation of the physics engine does not affect the collision detection and thus the contact points can be computed as usual, for the standing still robot. In this case, the contact points are stable. Therefore, small displacements of the legs of the robot, possibly due to numerical errors, result in the intense flickering of the contact points. Consequently, the method that computes the contact points and is utilized by GIMPACT, is sensitive to small variations of the position of the bodies that come in contact. However, these variations are not present when the physics engine is deactivated. Some modifications to the setup of the Gazebo model were also tested to find the remedy for this problem. Some of them are the following:

- Create denser and coarser meshes for the lower legs, which did not resolve the problem.
- Alter the structure of the manipulator. With this modification, the position of the CoM of the robot changed. This modification resolved the problem. However, this is not an appropriate solution for the problem since the position of the CoM of the robot changes when the configuration of the manipulator changes as well. Also, when the manipulator was removed completely from the robot, the flickering was completely removed. Nevertheless, this solution is not realistic.
- Decrease the sampling frequency of the sensors. Therefore, the "noise" that was present in the time responses was filtered out partially.
- Decrease the $real\_time\_factor$. Therefore, the "noise" that was present in the time responses was also filtered out partially, but not completely.

The remedy for the problem was to change the collision models of the lower legs. Spheres were used as shown in Figure 2-10, instead of 3D meshes, whose center coincides with the center of the toes, and whose radius is equal to the radius of the cylindrical segment of the toes. The selection of simpler collision models is a common practice, to increase the stability of the contact points and thus have more robust contact resolution, in many simulators [93]. The simpler collision primitives are also more preferrable in practice in comparison to the 3D meshes because they exhibit faster collision detection. Spheres are selected instead of cylinders since the contact of a sphere with a plane is modeled with only one contact point while the contact of a cylinder with a plane is modeled with two contact points. These two points are located at the two edges of the cylinder. In general, it is recommended to the minimum possible number of contact points in contact modeling. The cylinder can also be modeled with one contact point but in some cases, instabilities emerge, since the position of the contact point alternated between the two positions, at every time step of the simulation, that correspond to the edges of the cylinder.

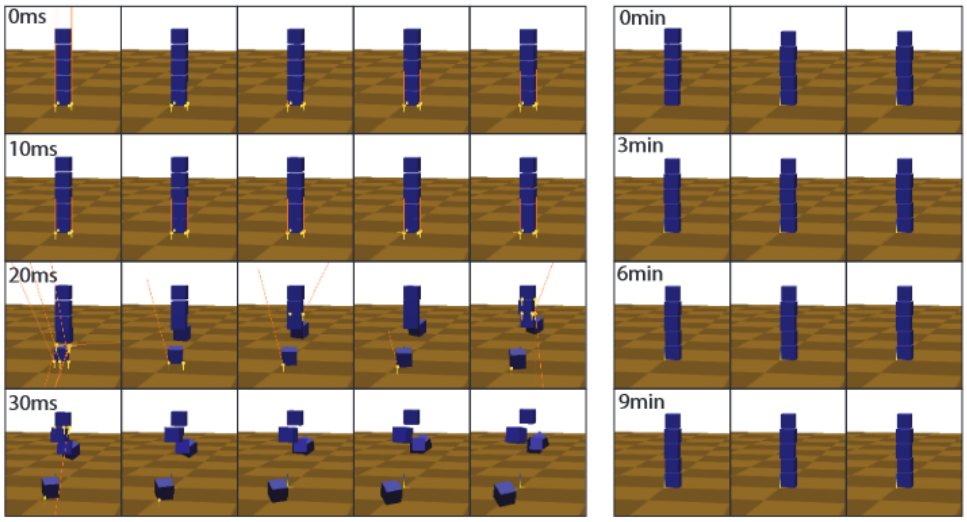


Figure 2-9. A stack of cubes is dropped. Left: with GIMPACT it "blows up" nearly instantly after the second block touches the first. Right: the expected outcome [93].

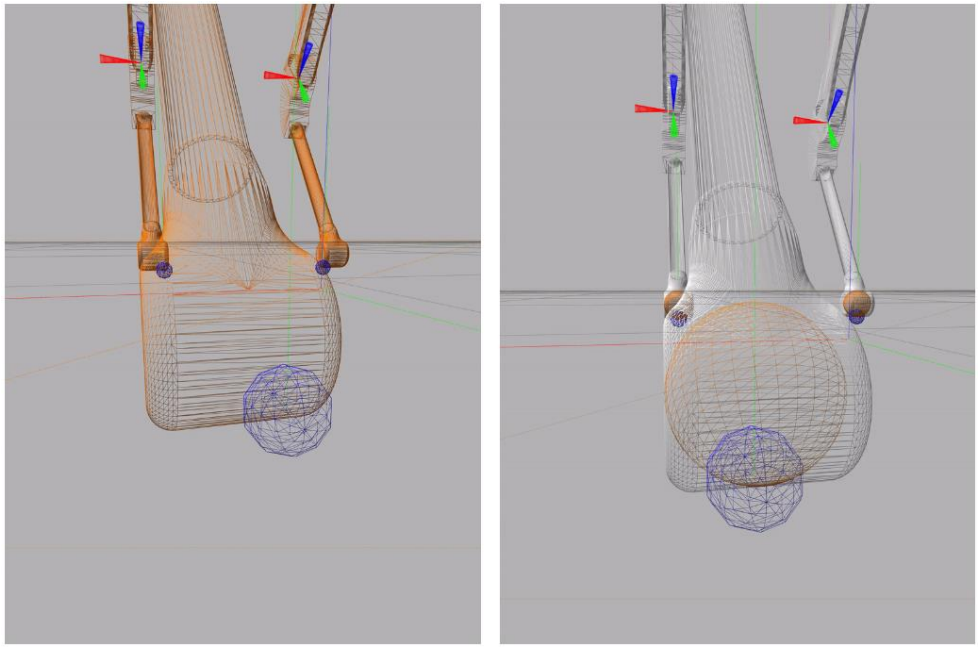


Figure 2-10. Collision primitives for Lower Legs. Left: 3D unstructured mesh. Right: Sphere.

2.2.8 Viscous Joint Damping

In OpenDE, according to [84], the modelling that is implemented for the joint-dampening is not adequate enough to model the joints of real robots. In general, the viscous friction joint dampening constant b is not easily determined as a constant before run-time. Moreover, OpenDE is not able to simulate viscous joint damping in the current implementation of the solvers world step and quick step. The user must apply the viscous joint damping force explicitly outside of OpenDE, where to compute the viscous joint damping force at the current time step, the joint velocity from the previous time step should be used,

$$\mathbf{f}_{\mathbf{damp}}{}^{n+1} = b\mathbf{v}_{\mathbf{joint}}{}^{n} \tag{2-40}$$

In this model, viscous friction joint dampening constant selected for all the joints of the robots is the following: $b = 0\,\mathrm{Nm\cdot s/rad}$. The joints of many real robots have gearboxes with high

reduction ratios which leads to high viscous damping coefficients and low efficiency. When large viscous damping forces are introduced in simulations, then the problem suffers from numerical stiffness. Numerically stiff problems demand very small integration time step sizes which leads to poor performance simulations. On the other hand, solving for damping forces implicitly is computationally expensive. In the case of the quick step solver the following approach can be adopted: the forces can be updated within each PGS iteration without significantly impairing the computational efficiency of the method and higher viscous damping coefficients can be utilized for a given time step size.

In Figure 2-11, describes an overview of a simulation architecture in Gazebo, with the use of OpenDE.

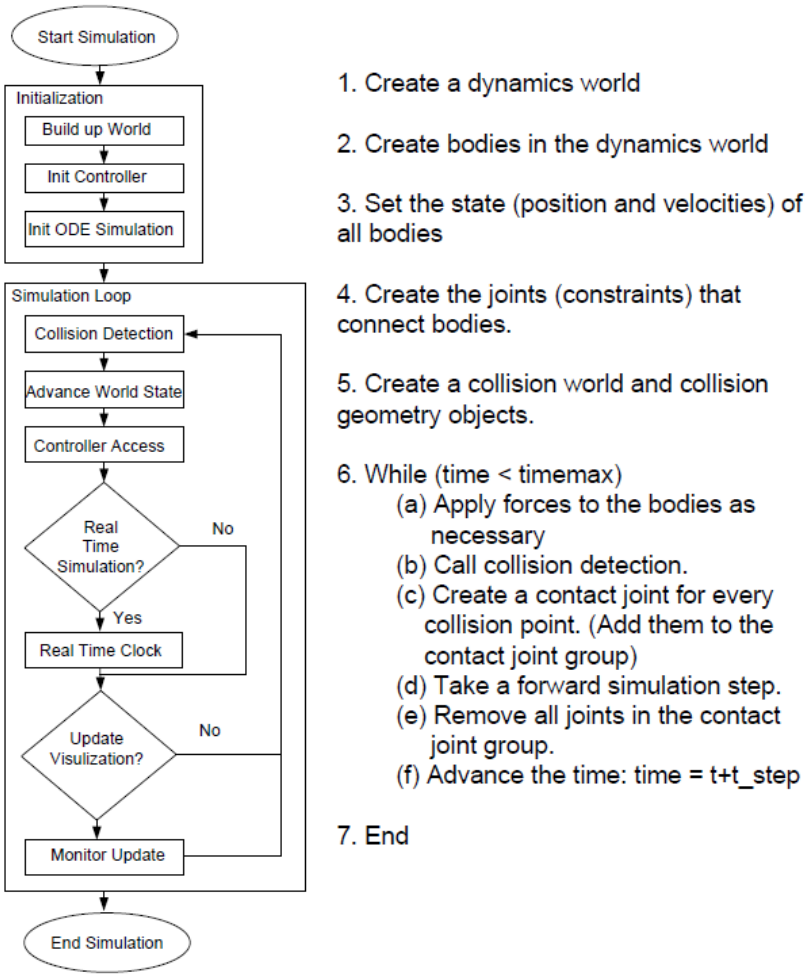


Figure 2-11. Flow chart of simulation architecture in OpenDE [63].

2.3 Validation of Gazebo Modelling

To validate the accuracy of the Gazebo modelling, the results of the Gazebo simulation of a robot have been compared with the results that the analytical equations provide for the same robot. The robot that has been tested is the leg of ARGOS. As long as this comparison was considered, it has been treated as a 2 DoF manipulator. To be more precise, the value of the abduction angle was fixed and was equal to zero. The analytical EoM were extracted using the Euler-Lagrange formulation. However, these equations were solved numerically using a simulation environment in Simulink.

2.3.1 Leg Model in Gazebo

The geometric and inertial parameters used for the model of the leg in Gazebo are the same as the ones used for ARGOS's legs. Also, the solver parameters and the joint constraints parameters used here are the same as the ones used for the model of ARGOS in Gazebo. Additionally, no virtual sensors were mounted on the leg. To compare the results of the Gazebo with the ones given by the analytical EoM, the values of the joint angular positions, angular velocities and torques are necessary. These values are taken from the ROS topic, */joint_states*.

The leg is driven using two PD controllers, one for every leg joint, which are provided by the ros_control package. The proportional and derivative gains of each controller K_p, K_d can be selected separately. These gains are defined in a *.yaml* file. The leg joints accept torque commands as reference inputs (effort joints), and therefore effort controllers are utilized. However, since position control is applied for each leg joint, the reference inputs that are sent to the aforementioned controllers are angular position commands. Thus, the controllers utilized accept position commands (joint_position_controllers) and produce torque commands, given the positions commands, that are applied to the joints by the simulated hip and knee actuators. These position controllers are running at a frequency of 1 KHz. The user can send reference inputs to the topics */joint1_position_controller/command* and */joint2_position_controller/command* that correspond to each controller, and in this case are the angular positions of the joints. To drive the center of the leg's foot along a desired path that lies inside the leg's workspace, defined in the cartesian space, a ROS node (publisher node) must be utilized, that implements the following procedure: at every time step, the desired position of the center of the leg's foot in cartesian space is computed by a trajectory planning algorithm. Then, this position in the leg's workspace is reconstructed to desired joint angles, expressed in the joint space, using an inverse kinematics algorithm. Finally, these angular positions are published to the aforementioned topics.

2.3.2 Trajectory Planning

The center of the foot toe should move along a trajectory that is nearly elliptical. More specifically, a time series of points, in the cartesian space, along an elliptical primitive in the workspace of each leg is generated, which are used as a reference for the leg's joint controllers. This trajectory is generated using the parametric equations of the ellipse which are expressed w.r.t. the fixed frame **O** that is located at the hip joint, as shown in Figure 2-12. These parametric equations are the following:

$$x_{tr} = x_c + b \cdot \sin\left(\omega_{tr} t_{sim}\right) \tag{2-41}$$

$$y_{tr} = y_c + a \cdot \cos\left(\omega_{tr} t_{sim}\right) \tag{2-42}$$

where $\begin{bmatrix} x_{tr} & y_{tr} \end{bmatrix}^T$ is the vector containing the coordinates of the elliptical path along which the center of the foot should move, $\begin{bmatrix} x_c & y_c \end{bmatrix}^T = \begin{bmatrix} 0 & -0.5 \end{bmatrix} \text{m}$ is the vector containing the coordinates of the ellipse center position, $a = 0.06 \text{m}$ and $b = 0.12 \text{m}$ are the lengths of the ellipse semi-axes, $\omega_{tr} = 10 \text{rad/s}$ the angular velocity of the foot's motion along the elliptical path and t_{sim} is the simulation time.

2.3.3 Inverse Kinematics

The leg's joint controllers accept as command the desired position of each leg joint. Therefore, to implement the control law, the time series of the desired foot positions, expressed in the cartesian space, should be reconstructed to a time series of the desired joint angles, expressed in the joint space. This reconstruction is achieved by using an inverse kinematics algorithm. This algorithm takes as input the desired position of the center of the legs foot $\begin{bmatrix} x_E & y_E \end{bmatrix}^T$ and gives the desired joint angles as output $\begin{bmatrix} \theta_1 & \theta_2 \end{bmatrix}^T$.

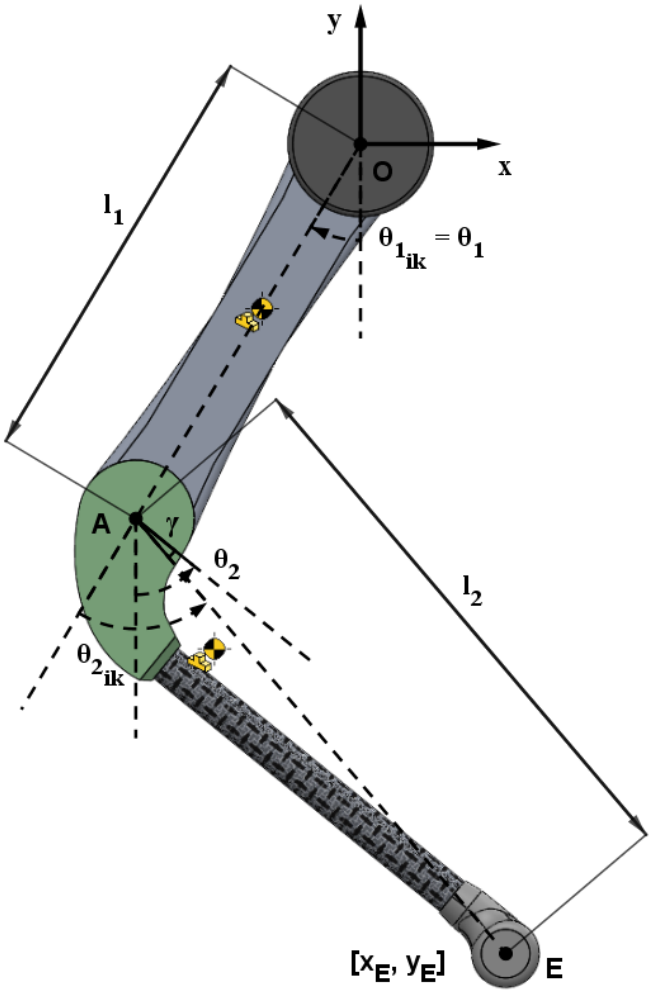


Figure 2-12. Leg dynamic model used for the calculation of the inverse kinematics.

The following relationships are derived by solving the inverse kinematics problem:

$$\cos\left(\theta_{2_{ik}}\right)=\frac{x_E{}^2+y_E{}^2-l_1^2-l_2^2}{2l_1l_2},\quad \sin\left(\theta_{2_{ik}}\right)=\pm\sqrt{1-\cos^2\left(\theta_{2_{ik}}\right)} \tag{2-43}$$

$$\theta_{2_{ik}}=a\tan 2\left(\sin\left(\theta_{2_{ik}}\right),\cos\left(\theta_{2_{ik}}\right)\right) \tag{2-44}$$

$$\theta_{1_{ik}}=a\tan 2\left(-x_E,-y_E\right)-a\tan 2\left(l_2\sin\left(\theta_{2_{ik}}\right),l_1+l_2\cos\left(\theta_{2_{ik}}\right)\right) \tag{2-45}$$

$$\theta_1=\theta_{1_{ik}},\theta_2=\theta_{2_{ik}}+\theta_1+\gamma \tag{2-46}$$

Therefore, two different leg configurations, meaning two different pairs of joint angles θ_1, θ_2, correspond to a single position of the center of the foot in the cartesian space. The one pair corresponds to configuration 1 and the other to configuration 2 as shown in Figure 2-13. The selected solution is the one that corresponds to leg configuration 1. This solution is yielded when the following inequality is satisfied: $\sin\left(\theta_{2_{ik}}\right) > 0$.

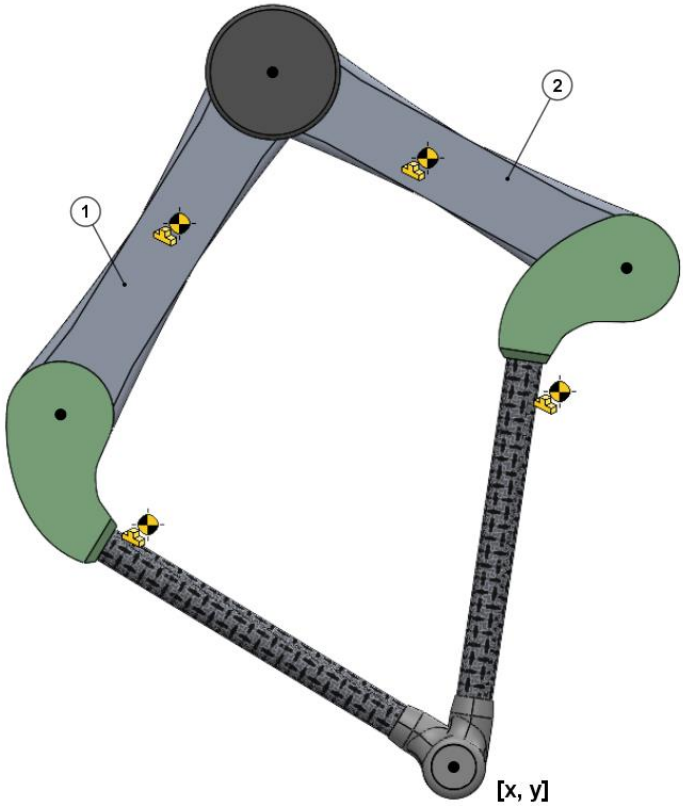


Figure 2-13. Two different configurations of the leg that correspond to the same point in the cartesian space [x, y].

2.3.4 Analytical EoM

To validate the Gazebo simulation results, the EoM should be found using the Euler-Lagrange formulation. As is shown in Figure 2-14, the model of the leg is a 2 DoF planar serial leg and consists of the Leg Roll, the Upper Leg, and the Lower Leg. The mass of each link of the leg is m_i, moment of inertia of each link of the leg about its CoM is I_i and the length of each link of the leg is l_i. The numbering of the subscript i starts from the Leg Roll. The length of link $i=1$, l_1, is defined as the distance between the hip and the knee joint, and the length of the link $i=2$, $l_{2_{eff}}$, is defined as the distance between the knee joint and the contact point of the foot with the ground, as shown in Figure 2-15. Point contact is assumed each time the foot impacts the ground. The CoM of each leg link is at a distance d_i from the respective leg joint. The hip and knee joints are driven by actuators modeled as ideal torque sources τ_i. When the foot impacts the ground, a force acts on it due to contact with the environment. This force consists of a component tangential at the contact surface F_x, due to friction, and of a normal component at the contact surface N.

The length $l_{2_{eff}}$ and the angle ε can be computed by applying the cosine and the sine theorem respectively to the triangle shown in Figure 2-15. The length $l_{2_{eff}}$ is given by:

$$l_{2_{eff}}{}^2 = l_2^2 + r^2 - 2l_2 r\cos\left(\pi - (\theta_2 - \gamma)\right) \Rightarrow$$
$$l_{2_{eff}} = \sqrt{l_2^2 + r^2 + 2l_2 r\cos(\theta_2 - \gamma)} \tag{2-47}$$

The angle ε is given by:

$$\frac{\sin(\varepsilon)}{r} = \frac{\sin\left(\pi - (\theta_2 - \gamma)\right)}{l_{2_{eff}}} \Rightarrow \sin(\varepsilon) = \frac{r\sin(\theta_2 - \gamma)}{l_{2_{eff}}} \Rightarrow$$
$$\varepsilon = \sin^{-1}\left(\frac{r\sin(\theta_2 - \gamma)}{l_{2_{eff}}}\right) \tag{2-48}$$

Therefore, the position of the contact point is a function of the angle θ_2 and not constant during the robot's motion.

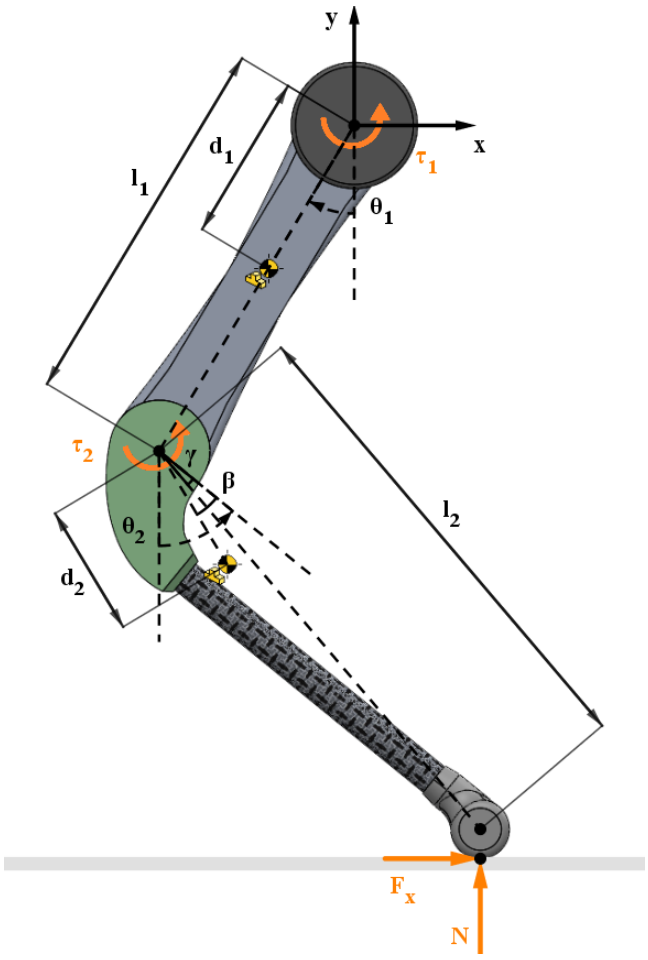


Figure 2-14. Leg dynamic model used for the calculation of the analytical EoM.

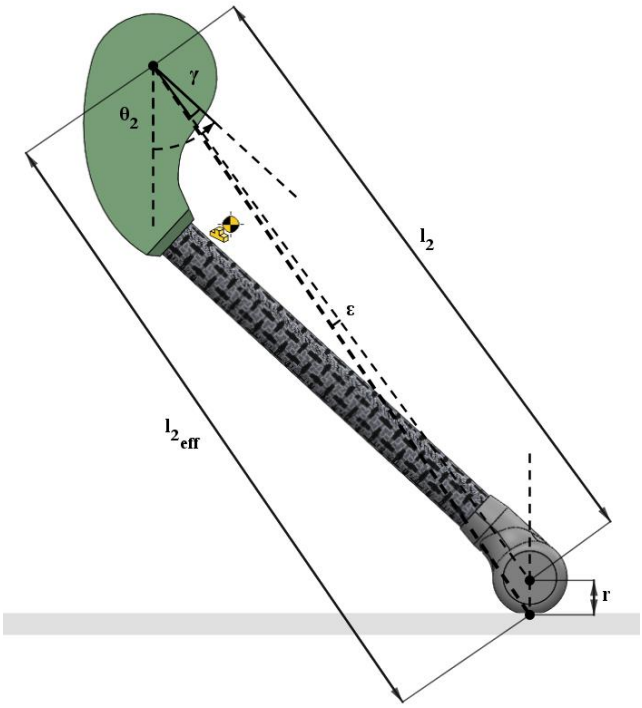


Figure 2-15. Geometry of ARGOS's leg at the contact point of the foot with the ground.

The robot's EoM are computed using the Euler-Lagrange formulation, as it is described in [94]. Firstly, a set of variables $\mathbf{q}_j$, $j=1,\ldots,n$ must be chosen, which are called generalized coordinates and describe the link positions of a robot with $n-\text{DOFs}$. In this case, the selected generalized coordinates $\mathbf{q}_j$, $j=1,\ldots,n$ are the following variables:

$$\mathbf{q}=\begin{bmatrix}\theta_1 & \theta_2\end{bmatrix}^T \tag{2-49}$$

Then the Lagrangian of the mechanical system L can be defined which is a function of the generalized coordinates. The Lagrangian is computed as follows:

$$L=T-U \tag{2-50}$$

where T is the total kinetic energy of the system and U is the total potential energy of the system. The Lagrange equations are expressed by:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\mathbf{q}}_j}\right)-\frac{dL}{\partial \mathbf{q}_j}=\frac{dP_\xi}{\partial \dot{\mathbf{q}}_j},\quad j=1,\ldots,n \tag{2-51}$$

where P_ξ is the power of the nonconservative generalized forces acting on the robot. These generalized forces are the joint actuator torques, the joint friction torques, and the joint torques induced by end-effector forces at the contact with the environment.

To compute T, U and P_ξ, the expressions that describe the cartesian coordinates of the CoM of each leg's link $\begin{bmatrix}x_i & y_i\end{bmatrix}^T$ as well as the coordinates of the foot $\begin{bmatrix}x_f & y_f\end{bmatrix}^T$ must be found. These expressions are the following:

$$x_1=d_1\sin(\theta_1) \tag{2-52}$$

$$y_1=-d_1\cos(\theta_1) \tag{2-53}$$

$$x_2 = l_1 \sin(\theta_1) + d_2 \sin(\theta_1 + \theta_2 - \beta) \tag{2-54}$$

$$y_2 = -l_1 \cos(\theta_1) - d_2 \cos(\theta_1 + \theta_2 - \beta) \tag{2-55}$$

$$x_f = l_1 \sin(\theta_1) + l_{2_{eff}} \sin(\theta_1 + \theta_2 - \gamma - \varepsilon) \tag{2-56}$$

$$y_f = -l_1 \cdot \cos(\theta_1) - l_{2_{eff}} \cdot \cos(\theta_1 + \theta_2 - \gamma - \varepsilon) \tag{2-57}$$

The total kinetic energy T of the system is computed using the following expressions:

$$T_1 = \frac{1}{2} I_1 \dot{\theta}_1^2 + \frac{1}{2} m_1 \left(\dot{x}_1^2 + \dot{y}_1^2 \right) \tag{2-58}$$

$$T_2 = \frac{1}{2} I_2 \dot{\theta}_2^2 + \frac{1}{2} m_2 \left(\dot{x}_2^2 + \dot{y}_2^2 \right) \tag{2-59}$$

$$T = T_1 + T_2 \tag{2-60}$$

The total potential energy U of the system is computed using the following expressions:

$$U_1 = m_1 g y_1 \tag{2-61}$$

$$U_2 = m_2 g y_2 \tag{2-62}$$

$$U = U_1 + U_2 \tag{2-63}$$

The power of the nonconservative generalized forces acting on the robot P_ξ consists of the power of the joint actuator torques, P_τ and the power of the joint torques induced by end-effector forces at the contact with the environment, P_g. In this case, it is assumed that the Lower Leg does not slide on the ground but rotates instantaneously about the contact point of the foot with the ground, whose velocity is equal to zero. These powers are given by the following expressions:

$$P_\tau = (\tau_1 - \tau_2)\dot{\theta}_1 + \tau_2 \dot{\theta}_2 \Rightarrow P_\tau = \tau_1 \dot{\theta}_1 + \tau_2 \left(\dot{\theta}_2 - \dot{\theta}_1 \right) \tag{2-64}$$

$$P_g = F_y \dot{x}_2 + N \dot{y}_2 + M \dot{\theta}_2 \tag{2-65}$$

where $M = (x_f - x_2) N - (y_f - y_2) F_y$

$$P_\xi = P_\tau + P_g \tag{2-66}$$

Taking the derivatives required by Lagrange equations in (2-51), the EoM of the robot can be rewritten in the following compact matrix form which represents the joint space dynamic model:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{c}(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{G}(\mathbf{q}) = \mathbf{S}^T \boldsymbol{\tau} + \mathbf{J}^T(\mathbf{q}) \mathbf{f} \tag{2-67}$$

where $\mathbf{q}$ is the generalized coordinates vector, $\mathbf{M}$ is the Joint Space Inertia matrix, $\mathbf{c}$ are the Coriolis and centripetal terms, $\mathbf{G}$ is the gravity term, $\mathbf{J}$ is the contact (geometric) Jacobian that that maps the external forces/torques to the generalized coordinate space, $\mathbf{f}$ are the

external forces/torques, $\mathbf{S}$ is the selection matrix that maps input forces/torques to joints and $\boldsymbol{\tau}$ are the input forces/torques. The vectors $\boldsymbol{\tau}, \mathbf{F}$ are the following:

$$\boldsymbol{\tau} = \begin{bmatrix} \tau_1 & \tau_2 \end{bmatrix}^T \tag{2-68}$$

$$\mathbf{F} = \begin{bmatrix} F_y & N \end{bmatrix}^T \tag{2-69}$$

The values of the matrices $\mathbf{M}, \mathbf{c}, \mathbf{G}, \mathbf{S}, \mathbf{J}$ may be found in Appendix A.

2.3.5 Integration of Analytical EoM

The Euler-Lagrange equations in (2-67) define the inverse dynamics problem. The inverse dynamics problem consists of determining the joint torques $\boldsymbol{\tau}$ which are needed to generate the motion specified by the joint positions, velocities and accelerations $(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}})$, given the possible end-effector forces $\mathbf{f}$. Solving the inverse dynamics problem is useful for manipulator trajectory planning and control algorithm implementation.

The forward dynamics problem consists of determining the joint accelerations $\ddot{\mathbf{q}}$, as well as the joint velocities and positions $\dot{\mathbf{q}}, \mathbf{q}$ through numerical integration, resulting from given joint torques $\boldsymbol{\tau}$ and the possible end-effector forces $\mathbf{f}$, given also the initial state of the system (i.e., initial joint positions $\mathbf{q_0}$ and initial joint velocities $\dot{\mathbf{q}}_\mathbf{0}$). Solving equation (2-67) for $\ddot{\mathbf{q}}$ yields the following expression:

$$\ddot{\mathbf{q}} = \mathbf{M}(\mathbf{q})^{-1}\left[\mathbf{S}^T\boldsymbol{\tau} - \mathbf{c}(\mathbf{q}, \dot{\mathbf{q}}) - \mathbf{G}(\mathbf{q}) + \mathbf{J}^T(\mathbf{q})\mathbf{f}\right] \tag{2-70}$$

In this equation, the inversion of matrix $\mathbf{M}(\mathbf{q})$ is always possible since it is positive definite. The joint velocities and positions are computed by integrating equation (2-70) two times, given the initial joint positions and velocities $\mathbf{q}(t = t_0) = \mathbf{q_0}, \dot{\mathbf{q}}(t = t_0) = \dot{\mathbf{q}}_\mathbf{0}$. Solving the forward dynamics problem is useful for manipulator simulation. The block diagram that implements equation (2-70), is shown in Figure 2-16.

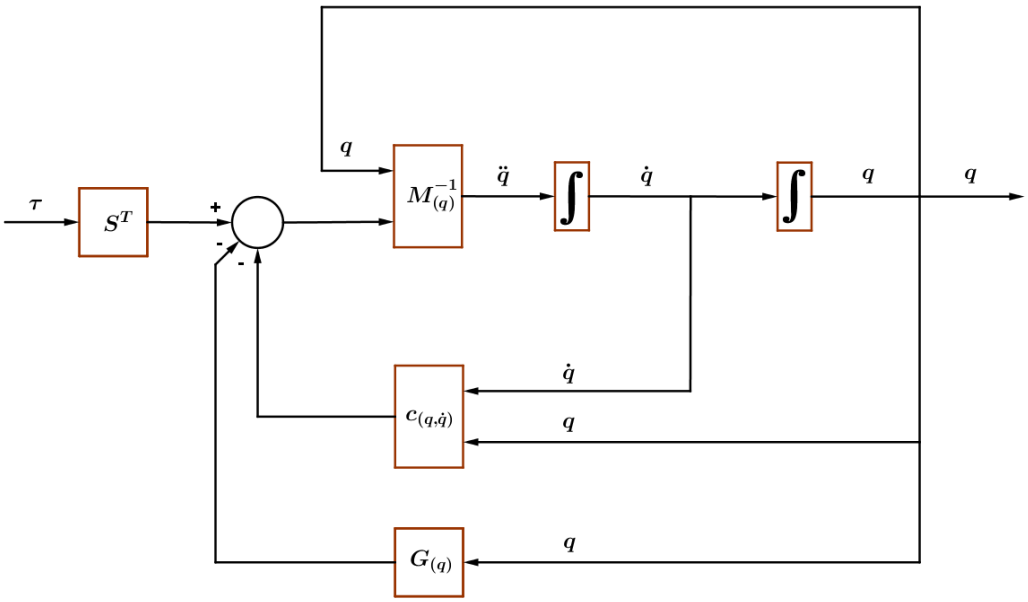


Figure 2-16. Block diagram of forward dynamics.

The joint accelerations and joint velocities are integrated numerically in a simulation environment built in Simulink, in which the variable-step solver ode45 is used, with absolute tolerance 10^{-9}, relative tolerance 10^{-9} and maximum step 10^{-3} s. The closed loop block diagram as it is implemented in Simulink is shown in Figure 2-17.

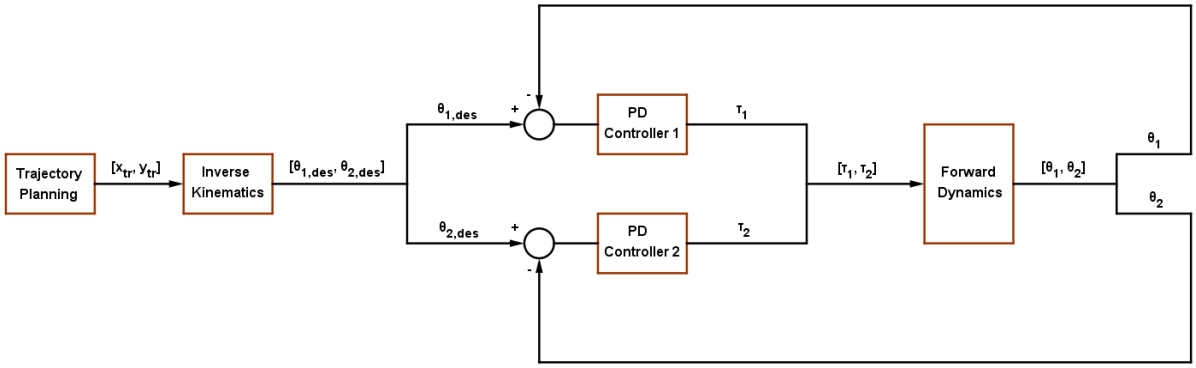


Figure 2-17. Block diagram of the closed loop system.

2.3.6 Comparison

Leg Stabilization

For this simulation, the leg is initially in a specific configuration. The controller's objective is to maintain the leg in this configuration despite the influence of gravity. The initial hip and knee joint angular positions are $\theta_1 = -0.5708\,\text{rad}$ and $\theta_2 = 0.5708\,\text{rad}$. The proportional and derivative gains selected for the hip and knee joint controllers are $K_{p,1} = 10\,\text{Nm/rad}$, $K_{d,1} = 1\,\text{Nm}\cdot\text{s/rad}$ and $K_{p,2} = 10\,\text{Nm/rad}$, $K_{d,2} = 1\,\text{Nm}\cdot\text{s/rad}$ respectively. During its motion, the leg's foot does not impact the ground.

The time responses of the joint angular positions as well as the difference between the responses given by the Gazebo model and the Simulink model are shown in Figure 2-18 and Figure 2-19 respectively. It is obvious that the results given by the Gazebo and the Simulink model do not differ significantly from each other. The steady state errors of the joint angles responses are relatively large. Tuning appropriately the gains K_p, K_d can improve the characteristics of the transient response. Increasing K_p will make the system's response faster, will increase the overshoot and will reduce the steady state error. However, the steady state error will not be eliminated unless an integral gain K_I gets introduced. Also, increasing K_d will limit significantly the oscillations present in the transient response of the system as well as the overshoot, since it increases the dampening of the oscillations, and will make the response more stable. Nevertheless, the purpose of these simulations is not the joint PD controllers tuning but the validation of the Gazebo's model results. The time responses of the joint angular velocities as well as the difference between the responses given by the Gazebo model and the Simulink model are shown in Figure 2-20 and Figure 2-21 respectively. It is also obvious that the results given by the Gazebo and the Simulink model do not differ significantly from each other. The time responses of the joint torques as well as the difference between the responses given by the Gazebo model and the Simulink model are shown in Figure 2-22 and Figure 2-23 respectively. It is obvious, in this case as well, that the results given by the Gazebo and the Simulink model do not differ significantly from each other. In

conclusion, the Gazebo modelling provides a satisfactory approximation of the modelling that utilizes the Euler-Lagrange equations and thus the two models agree with each other.

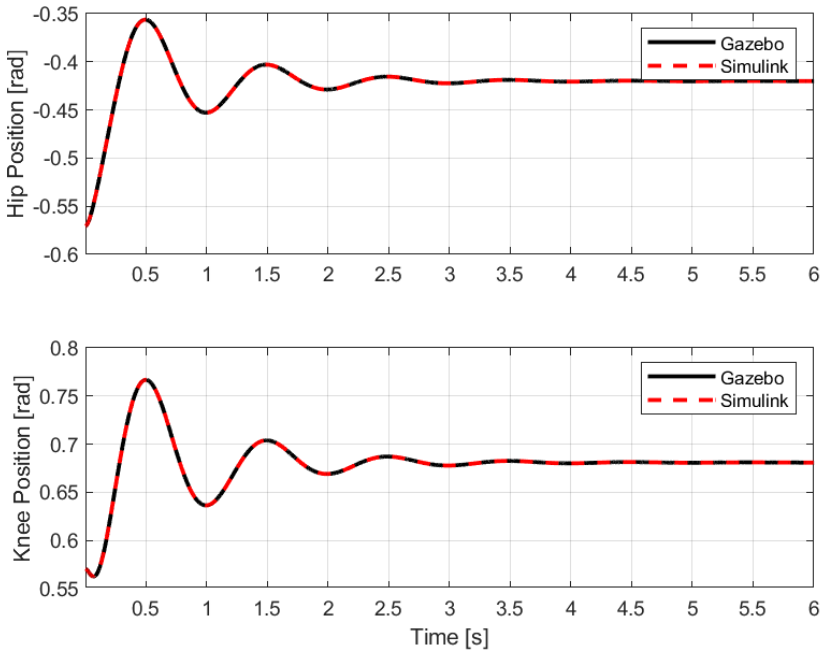


Figure 2-18. Time responses of the angular position of the hip and knee joint computed by the Gazebo model and the analytical model for the leg stabilization.

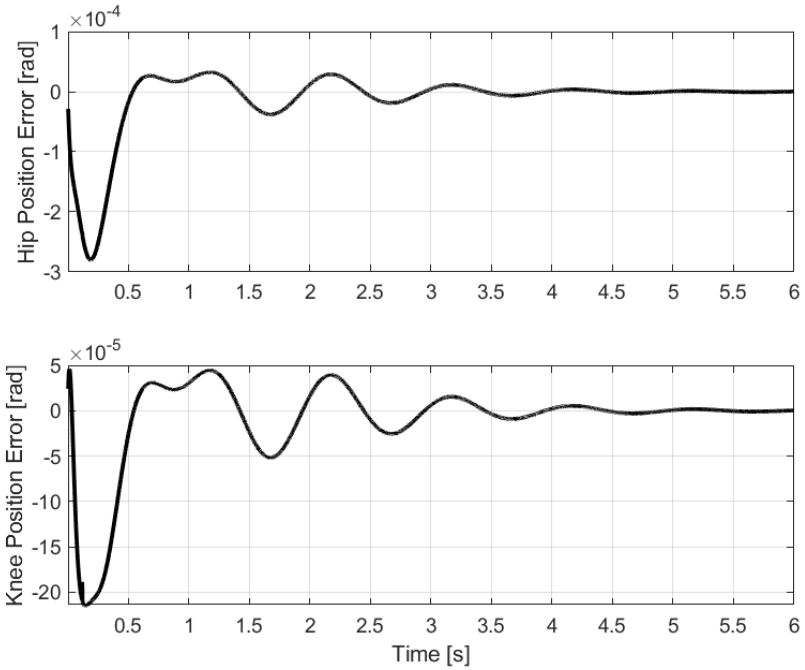


Figure 2-19. Difference between the time responses of the angular position of the hip and knee joint computed by the Gazebo model and the analytical model for the leg stabilization.

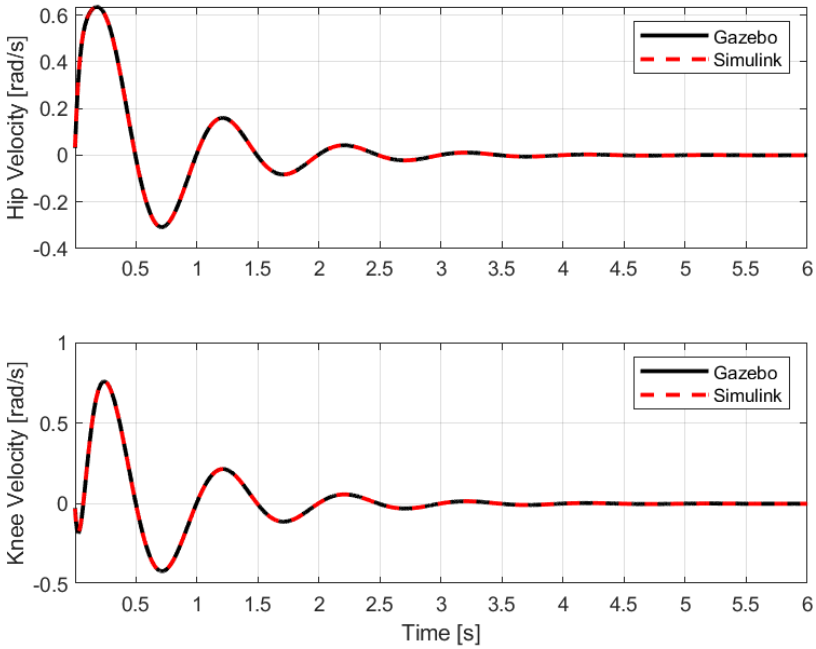


Figure 2-20. Time responses of the angular velocity of the hip and knee joint computed by the Gazebo model and the analytical model for the leg stabilization.

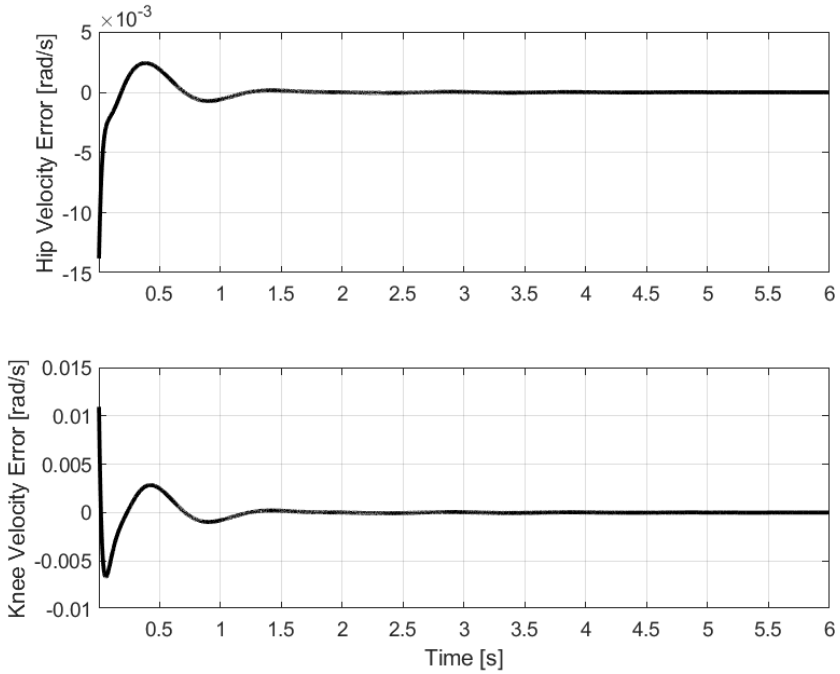


Figure 2-21. Difference between the time responses of the angular velocity of the hip and knee joint computed by the Gazebo model and the analytical model for the leg stabilization.

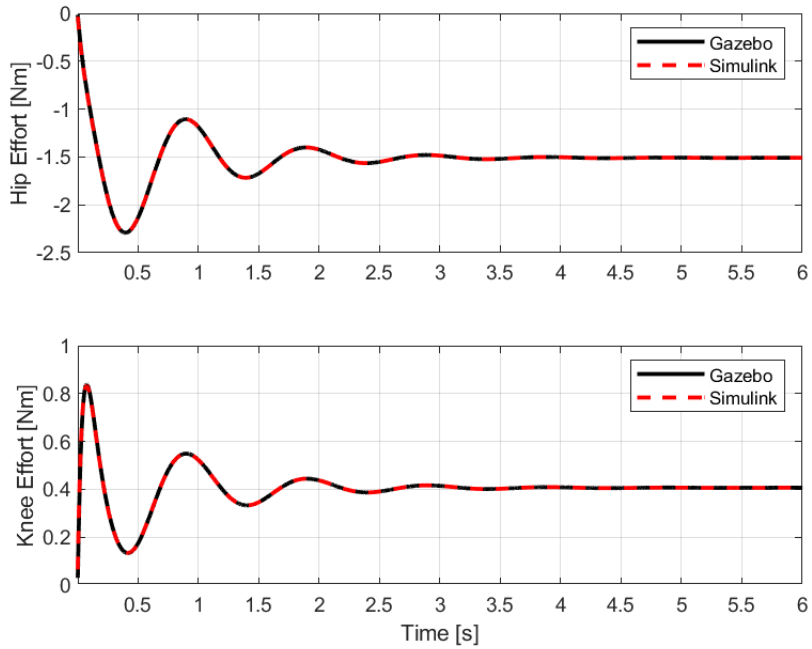


Figure 2-22. Time responses of the torque of the hip and knee joint computed by the Gazebo model and the analytical model for the leg stabilization.

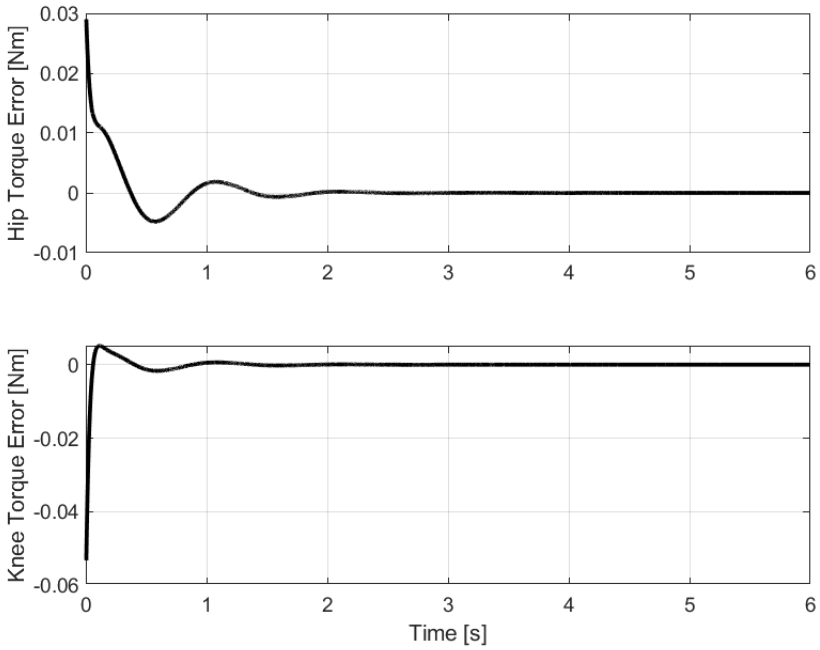


Figure 2-23. Difference between the time responses of the torque of the hip and knee joint computed by the Gazebo model and the analytical model for the leg stabilization.

Elliptical Trajectory

For this simulation, the controller's objective is to drive the center of the leg's foot along an elliptical path. This path is described in the chapter 2.3.2. The initial hip and knee joint angular positions are $\theta_1 = -1.4447\,\text{rad}$ and $\theta_2 = 1.0449\,\text{rad}$. The proportional and derivative gains selected for the hip and knee joint controllers are $K_{p,1} = 100\,\text{Nm/rad}$, $K_{d,1} = 1\,\text{Nm}\cdot\text{s/rad}$ and

$K_{p,2} = 70\,\text{Nm/rad}$, $K_{d,2} = 1\,\text{Nm}\cdot\text{s/rad}$. During its motion, the leg's foot does not impact the ground.

The time responses of the joint angular positions as well as the difference between the responses given by the Gazebo model and the Simulink model are shown in Figure 2-24 and Figure 2-25 respectively. The time responses of the joint angular velocities as well as the difference between the responses given by the Gazebo model and the Simulink model are shown in Figure 2-26 and Figure 2-27 respectively. Finally, the time responses of the joint torques as well as the difference between the responses given by the Gazebo model and the Simulink model are shown in Figure 2-28 and Figure 2-29 respectively. It is obvious, in this case as well, that the results given by the Gazebo and the Simulink model do not differ significantly from each other. In conclusion, the Gazebo modelling provides a satisfactory approximation of the modelling that utilizes the Euler-Lagrange equations and thus the two models agree with each other.

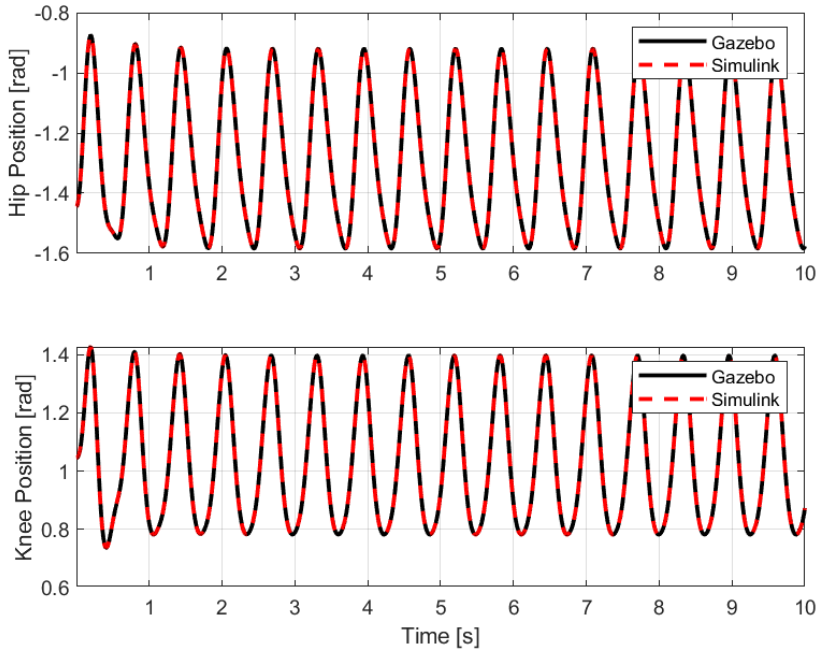


Figure 2-24. Time responses of the angular position of the hip and knee joint computed by the Gazebo model and the analytical model for the elliptical trajectory.

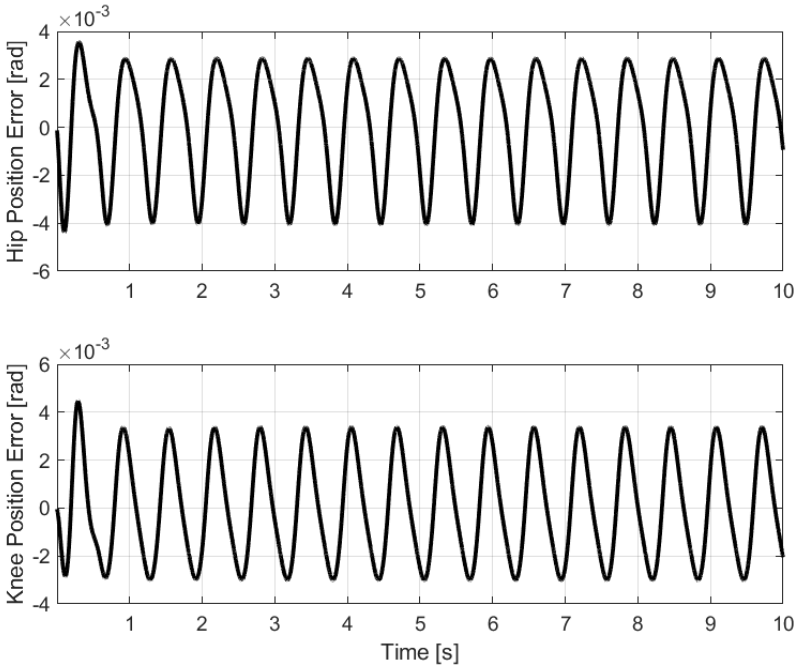


Figure 2-25. Difference between the time responses of the angular position of the hip and knee joint computed by the Gazebo model and the analytical model for the elliptical trajectory.

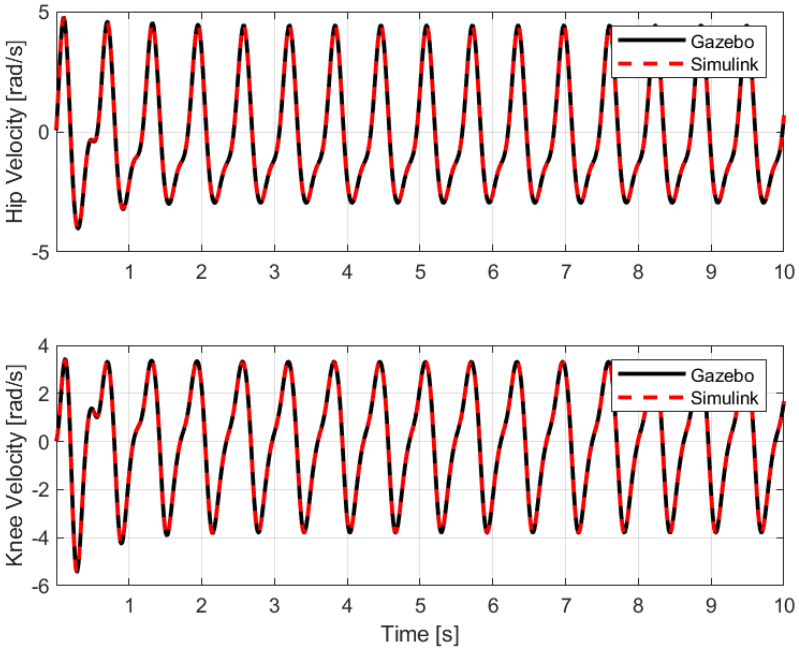


Figure 2-26. Time responses of the angular velocity of the hip and knee joint computed by the Gazebo model and the analytical model for the elliptical trajectory.

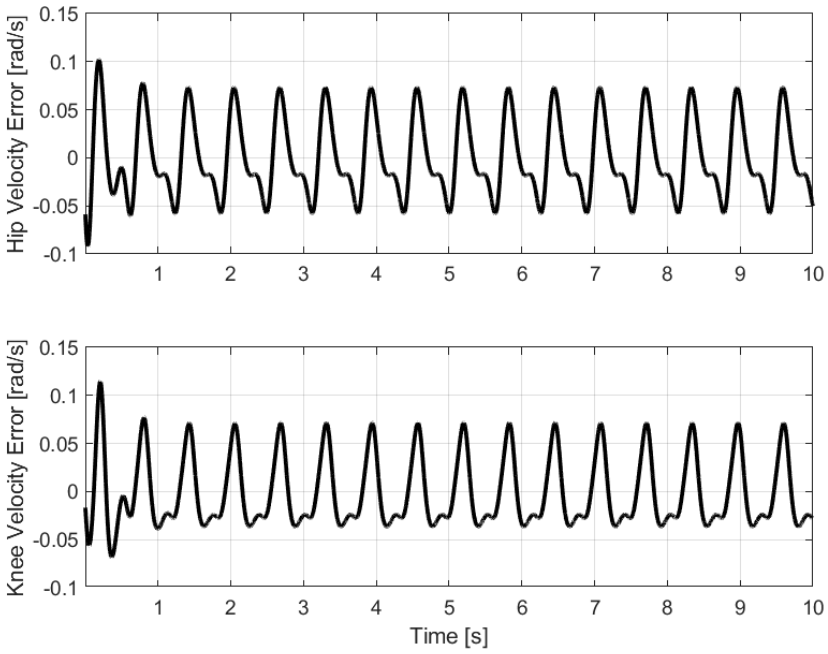


Figure 2-27. Difference between the time responses of the angular velocity of the hip and knee joint computed by the Gazebo model and the analytical model for the elliptical trajectory.

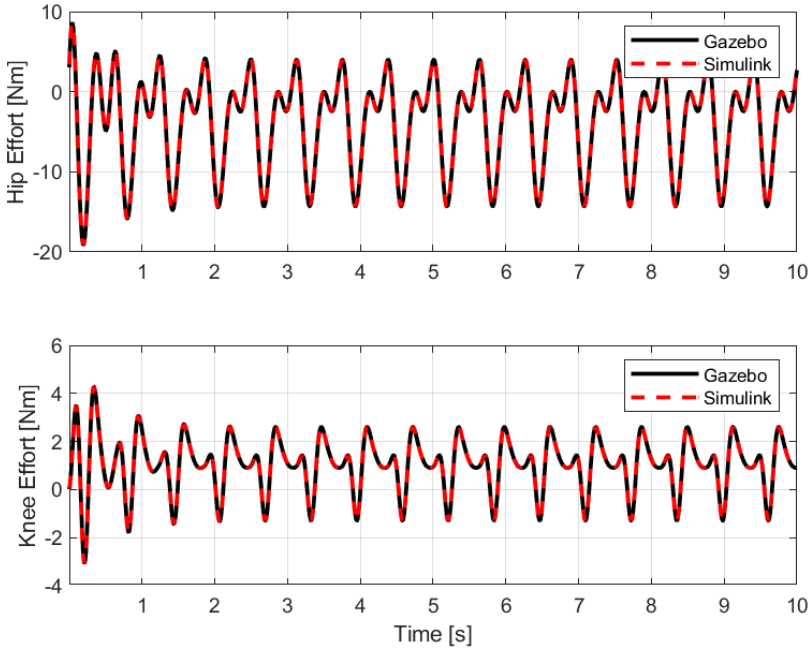


Figure 2-28. Time responses of the torque of the hip and knee joint computed by the Gazebo model and the analytical model for the elliptical trajectory.

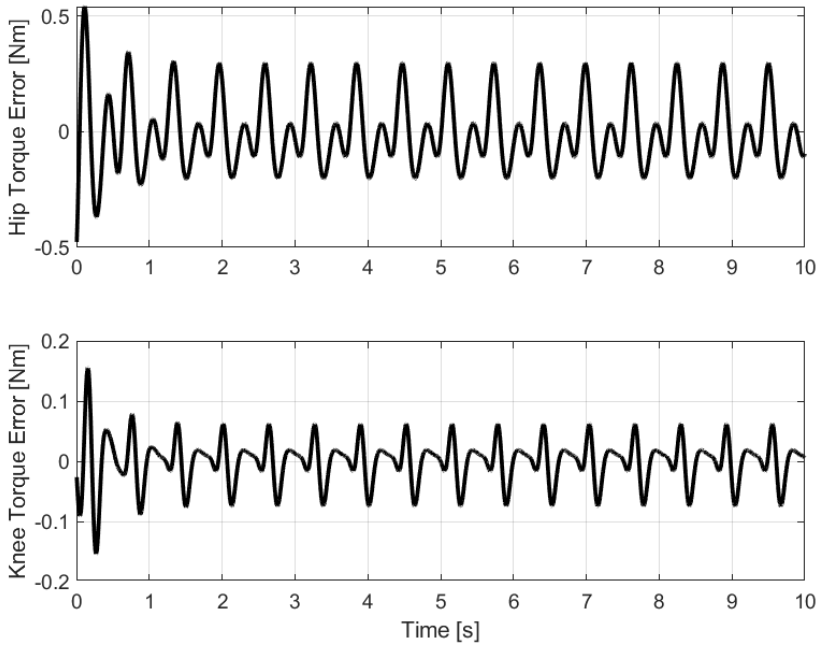


Figure 2-29. Difference between the time responses of the torque of the hip and knee joint computed by the Gazebo model and the analytical model for the elliptical trajectory.

2.4 Conclusion

In this chapter, the simulation framework for ARGOS in Gazebo is described and analyzed thoroughly. The various artifacts that tend to emerge in simulated models of legged robots were addressed through appropriate collision modelling as well as selection and tuning of the various parameters of the solver and the rest constituent functionalities of Gazebo. These modifications led to results that are qualitatively physically correct and realistic. Also, the comparisons with the analytical EoM validated the accuracy of the model and proved that the results of the Gazebo model are also quantitatively correct, since the pertinent errors were insignificant. Thus, it can be utilized as a testbed to evaluate the control frameworks that can be executed in real-time and will be described and analyzed in this work.

3 Optimal Control

3.1 Overview of Optimal Control (Open/Closed Loop)

Optimal control is concerned with finding the optimal choice of controls (motion plan) to achieve some desired goal, while minimizing a given cost function or objective function or performance index (e.g., energy consumption, task completion time) and while satisfying constraints (e.g., on contact forces, actuation limits, path constraints).

Methods for formulating and solving Optimal Control problems (open/closed loop) are divided into three broad categories, as shown in Figure 3-1. A detailed overview can be found in [95], [96].

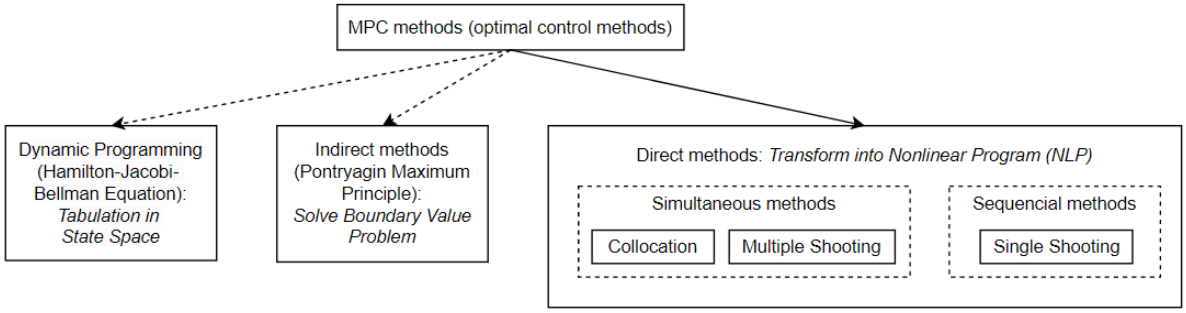


Figure 3-1. Overview of Optimal Control methods.

3.1.1 Dynamic Programming

In Dynamic Programming (DP) [97], a partial differential equation (PDE), known as Hamilton-Jacobi-Bellman (HJB) equation, is discretized and solved over the entire state space. It finds a globally optimal, closed loop solution, also known as optimal policy. An optimal policy, that has the form $\mathbf{u}=\mathbf{u}(\mathbf{x})$, provides globally optimal control for every initial state in the state space, as depicted in Figure 3-2. However, DP does not scale well to high dimensional systems since the computational cost grows exponentially with the state dimension ("curse of dimensionality"). For this reason, DP is trackable only when applied on low dimensional systems. More information about DP will also be presented in chapter 3.3.

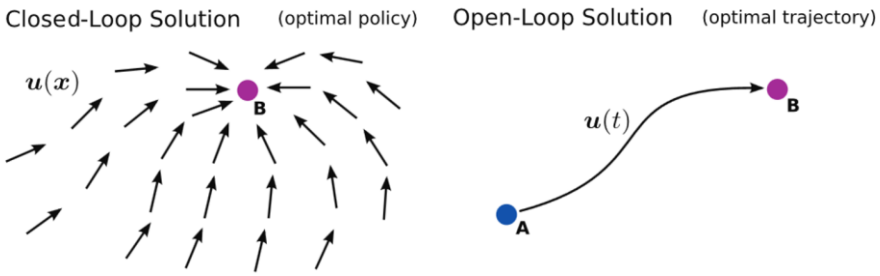


Figure 3-2. Optimal policy vs Optimal trajectory [98].

3.1.2 Indirect Methods

Indirect methods are based on Pontryagin's Maximum Principle (PMP) and derive and solve numerically a Boundary Value Problem (BVP) [99]. Indirect methods analytically construct the necessary and sufficient conditions for optimality, then they discretize these conditions and

solve them numerically using non-linear root finding. This is why these approaches are often referred to as “first optimize, then discretize”. The first-order optimality conditions are determined using the calculus of variations [99]. Generally, indirect methods tend to be more accurate and have a more reliable error estimate compared to the direct methods. These advantages are a result of the analytically derived conditions of optimality [98]. However, constructing these conditions analytically can be challenging [100]. Indirect methods tend to be numerically unstable and difficult to implement and initialize [9]. Thus, they are rarely used in practice.

3.1.3 Direct Methods

Direct methods transcribe the continuous time MPC problem to a finite dimensional Nonlinear Programming Problem (NLP). The transcription happens by discretizing the problem in time, by dividing the time period (horizon in MPC) of length T seconds into $N-1$ segments, typically of equal length of Δt seconds. As a result, N discretization points are created, also referred to as “knot” points, including the initial and final times. Then, they get solved using numerical optimization techniques. Consequently, these approaches are often referred to as “first discretize, then optimize”. In general, direct methods have a larger region of convergence when compared to indirect methods which means that they do not require as good of an initialization as indirect methods do. Finally, direct methods scale well to high dimensional systems but converge a locally optimal, open loop solution, also known as optimal trajectory [101]. An optimal trajectory, that has the form $\mathbf{u}=\mathbf{u}(t)$, is a sequence of control actions, as a function of time, for a single initial state, as depicted in Figure 3-2. In some instances, these methods may even fail to converge to a solution. Despite that downside, direct methods are the most widely used methods in MPC applications on robotic systems [102] and therefore will be examined thoroughly in this work.

3.2 Transcription Methods

Finite-horizon, OCPs, in continuous time have the following form:

$$\begin{aligned} \min_{\mathbf{x}(t),\mathbf{u}(t)} \quad & J=\int_{t_0}^{t_f} l\left(\mathbf{x}(t),\mathbf{u}(t)\right)dt + l_f\left(\mathbf{x}\left(t_f\right)\right) \\ \text{s.t.} \quad & \dot{\mathbf{x}}(t)=\mathbf{f}\left(\mathbf{x}(t),\mathbf{u}(t),t\right), \quad \mathbf{x}(t_0)=\mathbf{x}_0 \\ & \mathbf{g}\left(\mathbf{x}(t),\mathbf{u}(t)\right)\le\mathbf{0}, \qquad \forall t\in\left[t_0,t_f\right] \\ & \mathbf{h}\left(\mathbf{x}(t),\mathbf{u}(t)\right)=\mathbf{0}, \qquad \forall t\in\left[t_0,t_f\right] \end{aligned} \tag{3-1}$$

where the decision variables $\mathbf{x}(t),\mathbf{u}(t)$ are the state and control input trajectories from the initial time t_0 to the final time t_f. The initial and final time are parameters and not decision variables in finite/receding horizon control problems. The decision variables have to be calculated by the solver so that they minimize the cost function J while fulfilling the dynamical system constraints, defined by the continuous dynamics $\dot{\mathbf{x}}(t)=\mathbf{f}\left(\mathbf{x}(t),\mathbf{u}(t),t\right)$, given the initial state of the system $\mathbf{x}_0$, other equality constraints $\mathbf{h}\left(\mathbf{x}(t),\mathbf{u}(t)\right)=\mathbf{0}$ and inequality constraints $\mathbf{g}\left(\mathbf{x}(t),\mathbf{u}(t)\right)\le\mathbf{0}$. The cost function, in general, is comprised of the stage or running cost l and the terminal or final cost l_f.

This problem is also called trajectory optimization (TO) problem. The decision variables here are vector functions and not real numbers and thus the problem is infinite dimensional.

Additionally, the fact that $\mathbf{x},\mathbf{u}$ are vector functions contributes to the fact that J is a functional and not a scalar function. Consequently, this problem is extremely difficult to solve, since the space of functions is much larger than the space of real numbers and thus extremely hard to optimize over. Also, integral terms and differential equations are part of the cost function or the constraints of the problem.

Consequently, transcription (or discretization) is being utilized, by the direct methods, to convert the continuous TO problem into a constrained parameter optimization problem or a finite dimensional NLP. The problem is discretized in time, by dividing the time period (prediction horizon in MPC) of length $T=t_f-t_0$ seconds into $N-1$ segments, typically of equal length of Δt seconds. As a result, N discretization points are created, also referred to as "knot" points, including the initial and final times. Finite-horizon, OCPs, in discrete time have the following form:

$$\begin{aligned}
\min_{\mathbf{x}_{0:N},\mathbf{u}_{0:N-1}} \quad & J=\sum_{k=0}^{N-1} l_k\left(\mathbf{x}_k,\mathbf{u}_k,\Delta t\right)+l_N\left(\mathbf{x}_N\right) \\
\text{s.t.} \quad & \mathbf{x}_{k+1}=\mathbf{f}\left(\mathbf{x}_k,\mathbf{u}_k,\Delta t\right), \quad \mathbf{x}_{(k=0)}=\mathbf{x}_0, \qquad \forall k=\{0,\ldots,N-1\} \\
& \mathbf{g}_k\left(\mathbf{x}_k,\mathbf{u}_k\right)\le \mathbf{0}, \qquad\qquad\qquad \forall k=\{0,\ldots,N-1\} \\
& \mathbf{h}_k\left(\mathbf{x}_k,\mathbf{u}_k\right)=\mathbf{0}, \qquad\qquad\qquad \forall k=\{0,\ldots,N-1\}
\end{aligned} \tag{3-2}$$

where k is the time-step index. Here the decision variables $\mathbf{x}_k=\mathbf{x}(t_k)$ and $\mathbf{u}_k=\mathbf{u}(t_k)$ where $\mathbf{x}_k\in\mathbb{R}^n$ and $\mathbf{u}_k\in\mathbb{R}^m$, are real numbers and thus the problem is finite dimensional. State and control trajectories $\mathbf{X},\mathbf{U}$, where $\mathbf{X}\in\mathbb{R}^{nN}$ and $\mathbf{U}\in\mathbb{R}^{m(N-1)}$, are defined as a sequence of states $\mathbf{X}\triangleq\left[\mathbf{x}_0,\ \mathbf{x}_1\ \ldots,\ \mathbf{x}_N\right]^T$ and a sequence of control inputs $\mathbf{U}\triangleq\left[\mathbf{u}_0,\ \mathbf{u}_1\ \ldots,\ \mathbf{u}_N\right]^T$. Additionally, due to that fact, the objective function J is a scalar function, not a functional. Moreover, the integral terms (i.e., the stage cost) are approximated with discrete sums and the ordinary differential equations ODEs (i.e., the system dynamics in continuous time $\dot{\mathbf{x}}(t)=\mathbf{f}\left(\mathbf{x}(t),\mathbf{u}(t),t\right)$) are transformed into discrete difference equations (i.e., the system dynamics in discrete time $\mathbf{x}_{k+1}=\mathbf{f}\left(\mathbf{x}_k,\mathbf{u}_k,\Delta t\right)$), where $\mathbf{f}\left(\mathbf{x}_k,\mathbf{u}_k,\Delta t\right):\mathbb{R}^n\times\mathbb{R}^m\to\mathbb{R}^n$. Both of these are essentially converted into algebraic expressions. Here the stage and terminal cost of the cost function are denoted as l_k,l_N, where $l_k\left(\mathbf{x}_k,\mathbf{u}_k\right):\mathbb{R}^n\times\mathbb{R}^m\to\mathbb{R}$.

The most common assumptions made to successfully tackle TO problems are the following:

- Cost functions are assumed to be C^2 (have continuous second derivatives) and constraints are assumed to be, at least, C^1 (have continuous first derivatives), but ideally should be also C^2. Also, the derivatives should all remain relatively small. Non smooth (sharp) but continuous corners will still cause problems [101].
- The output of both the cost function and the constraints must be consistent for every function call. This means that the same lines of code must be executed at every function call (functions that include if statements, like absolute value function or minimum function should be avoided).
- TO is generally applied on single-phase problems, as shown in Figure 3-3, in which the dynamics are continuous over the entire time horizon. However, it can also be applied on hybrid systems with multi-phase trajectories, as shown in Figure 3-4, where the system dynamics are piecewise-continuous with discrete jumps between them. In these problems, multiple sequences of continuous-motions phases exist separated by discrete jumps. They can be solved in fashion similar to

solving many single-phase problems parallelly while connecting the boundary constraints between any two phases to couple the two trajectories.

Figure 3-3. Single-phase problems.

Figure 3-4. Multi-phase problems.

Direct transcription methods, as is shown in Figure 3-5, can be divided into sequential and simultaneous methods. A detailed overview of them can be found in [103], [104]. Sequential methods consist of single shooting and simultaneous methods consist of collocation and multiple shooting.

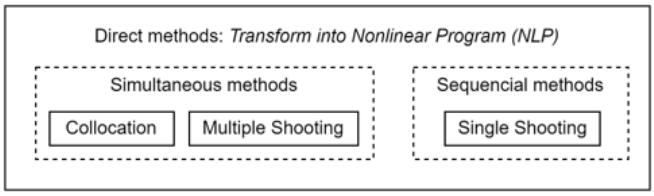


Figure 3-5. Direct methods overview.

3.2.1 Single Shooting

In single shooting, the decision variables are only the control inputs $\mathbf{u}(t)$, which are parameterized (discretized) through a set of discrete variables, $\mathbf{u}_k$. The states are represented using simulation. The system dynamics are integrated forward in time (shooting or rollout) using explicit (forward) integration, as shown in Figure 3-6, while applying the computed control inputs. A single shot is being conducted (single segment trajectory) starting from the initial time t_0 and ending at final time t_f. Therefore, the state trajectory, that is not part of the decision variables, is computed using the control input trajectory and the initial state $\mathbf{x}_0$. Thus, the system dynamics are enforced to the solution of the optimization problem using simulation, and consequently single shooting is referred to as method based on simulation. Also, the resulting state trajectory is always dynamically feasible (satisfies the dynamical system constraint), even before the convergence occurs, due to the forward rollouts that happen in every iteration of the solving procedure.

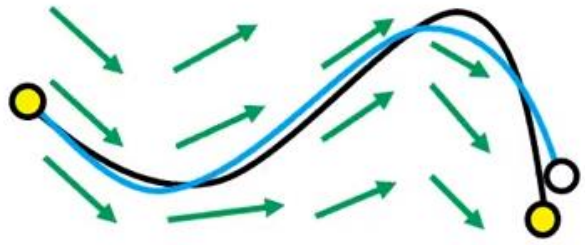


Figure 3-6. Sequential methods [105].

One advantage of single shooting is pertinent to the utilization of adaptive step-size, error-controlled ODE/DAE solvers [103]. These solvers integrate the dynamics of the system with a reduced computational cost compared to fixed step-size integrators [104], thereby reducing the overall computational cost of the optimization method. Moreover, the number of decision variables of this problem is relatively small, since they are only the control inputs. As a result, the size of the derived NLP is relatively small compared to the NLPs created using simultaneous methods, and thus less expensive to solve. Additionally, only an initial guess for the control inputs is necessary for the initialization of the solution.

A downside of single shooting is that it only requires an initial guess for the input trajectory, not for the state trajectory. The guess for the state trajectory is often more intuitive than an initial guess for the input trajectory. As a result, available information about the state trajectory cannot be utilized for the initialization of the method. Moreover, handling unstable systems with single shooting is a demanding task due to the fact that variations in control input affect the entire state trajectory [101], [103], [104]. To be more precise, integrating over a horizon requires calling recursively the explicit integrator function:

$$\begin{aligned}\mathbf{x}_{k+1} &= \mathbf{f}\left(\mathbf{x}_k, \mathbf{u}_k\right) \\ &= \mathbf{f}\left(\mathbf{f}\left(\mathbf{x}_{k-1}, \mathbf{u}_{k-1}\right), \mathbf{u}_k\right) \\ &= \vdots \\ &= \mathbf{f}\left(\mathbf{f}\left(\cdots \mathbf{f}\left(\mathbf{x}_0, \mathbf{u}_0\right), \mathbf{u}_{k-1}\right), \mathbf{u}_k\right)\end{aligned} \tag{3-3}$$

By doing so nonlinearity is increasing along the horizon, leading to inaccurate solutions to the problem [106]. The longer the horizons that are utilized get, the more inaccurate the solution of the problem becomes. This is a well-recognized and documented limitation that solvers based on single shooting have [106]. It is a fundamental property of the problem that was formulated using single shooting, and not merely an artifact of that particular transcription method [107]. Finally, it is difficult to enforce constraints on the state trajectory because the state values are not part of the decision variables of the optimization problem [100].

Solvers dedicated to solving single shooting problems follow the procedure sketched here. Firstly, an initial guess of the control trajectory is made. Secondly, given that initial guess, the system dynamics get integrated forward in time, thus providing a state trajectory. Then checks are conducted to verify whether the constraints are violated or not (e.g., check if a desired goal state has been reached (terminal constraint) as shown in Figure 3-7). If the

constraints are violated, a modification of the control inputs gets calculated and then it gets applied to the control trajectory thus updating its parameterization. This process is repeated until the constraints are satisfied and desired convergence criteria are met. Consequently, in single shooting, simulation and optimization proceed sequentially.

Single shooting problems have special, dense structures [95], [108], also known as Markovian structure. Therefore, these problems are solved more efficiently by algorithms that exploit that structure by solving a sequence of smaller sub-problems [106]. Such algorithms are structure-exploiting Newton-type methods for nonlinear MPC problems. Also, these algorithms are based on Riccati recursion-based Newton's methods, often interpreted as Differential Dynamic Programming (DDP) [44]. More information about DDP will be presented extensively in chapter 3.3.

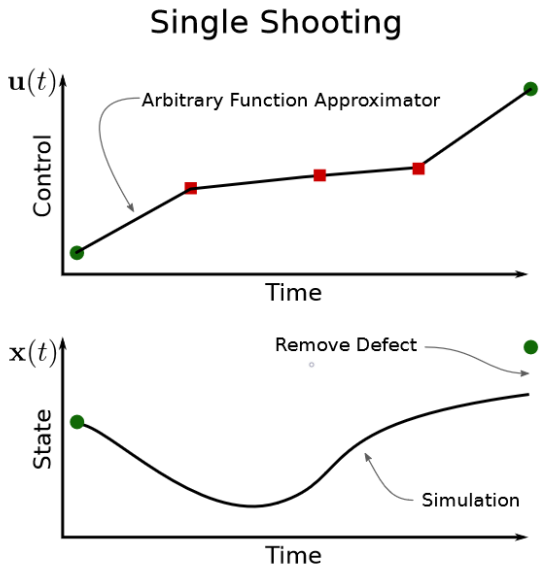


Figure 3-7. Single Shooting [101].

3.2.2 Collocation

In collocation, the decision variables are both the states $\mathbf{x}(t)$ and control inputs $\mathbf{u}(t)$, which are parameterized (discretized) through sets of discrete variables, $\mathbf{x}_k, \mathbf{u}_k$. The state and control input trajectories are represented using function approximations (e.g., polynomial splines), as shown in Figure 3-8, which essentially are implicit integration schemes (e.g., implicit Runge-Kutta methods). The system dynamics are imposed to the solution of the optimization problem as equality constraints between knot points, at pre-specified times on intermediate points, called collocation points. The solver will attempt to vary the state and control input trajectories $\mathbf{x}_k, \mathbf{u}_k$ simultaneously while trying to enforce the dynamical system constraints between them at the collocation points. Consequently, collocation is referred to as method based on function approximation. Also, in collocation methods, simulation and optimization proceed simultaneously and so the solution will satisfy the dynamical system constraint, as well as the rest of the enforced constraints, only at the convergence of the NLP solving procedure. As a result, if the user extracts the values of the decision variables before the convergence of the solver, they will not obey the dynamical system constraints and thus this (sub optimal) trajectory cannot be executed by the robot.

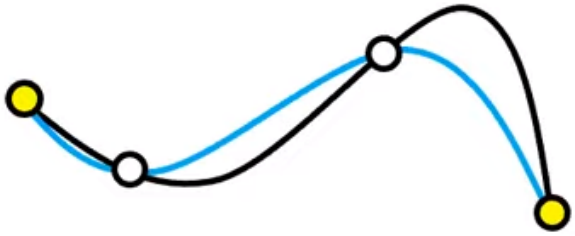


Figure 3-8. Simultaneous methods [105].

One advantage of collocation is that the obtained NLPs are very sparse[1], although larger in size than the NLPs derived using single shooting and show fast local convergence [103]. Another advantage is pertinent to the initialization of the method. To be more precise, knowledge of the initial state trajectory can now be utilized for the initialization of the method, since states are also part of the decision variables of the optimization problem, alongside the control inputs. Therefore, the initial guess of the solution can be easily made. Since the states are part of the decision variables of the optimization problem, collocation can easily handle state and terminal constraints [103]. Finally, collocation is more robust and more suitable for handling unstable systems compared to single shooting. Because of the fact that both the states and the controls inputs are separate, independent decision variables, variations in control input solely impact the state only at that time instance, with no impact on future states [104]. This property is demonstrated in the sparsity patterns within the Jacobians and Hessians of constraint and cost functions [68], [101], [103].

The main downside of collocation is that it is computationally costly to change its accuracy during the runtime of the optimization. To do so, the points (and the times) on which the dynamics constraint is enforced have to change (regridding) during the iterations of the optimization, altering also the dimension of the NLP. Single shooting does not suffer from that issue since adaptive step-size solvers are utilized there [104]. Nevertheless, collocation is extensively used in many OCPs that are presented thoroughly in the literature review.

Collocation problems can be solved using general Newton-type methods for NLPs such as Sequential Quadratic Programming (SQP) methods and (nonlinear) Interior Point (IP) (or Barrier) methods. These numerical optimization techniques have been implemented and developed along with efficient general-purpose off-the-shelf NLP solvers like SNOPT [27], IPOPT [24] and KNITRO [71]. These solvers can tackle collocation as well as single shooting problems. However, it is important to note that these solvers handle sparser problems, with certain structures, more efficiently because they exploit he banded diagonal sparsity patterns within the cost and constraint Jacobians/Hessians [103]. In single shooting, this is not the case, as the aforementioned diagonal sparsity patterns do not exist due to the fact that state trajectories were not optimized alongside control inputs trajectories [64]. Both the constraint Jacobian and the Lagrangian Hessian exhibit a highly sparse, repeated structure, as was shown in [31], which contributes to the flexibility and speed of the SQP/IP algorithms. In

[1] The problem's matrices have a sufficiently large number of zeros compared to non-zero elements, allowing for algorithmic advantages to be leveraged [109].

general, the sparser the problem, the more suitable these solvers will be to solve it. Consequently, SQP and IP methods are ideal for collocation problems and are suboptimal for single shooting. SNOPT and IPOPT can handle any arbitrary NLP, with any arbitrary constraints, and are numerically robust and versatile. However, SNOPT, IPOPT and similar optimizers tend to be slower, when compared with optimizers based on DDP, due to the sparse linear solvers they use (i.e., MA27, MA57, and MA97 [110]). These sparse linear solvers are necessary for performing very large matrix factorizations when computing the search direction required to converge to a solution [106]. The aforementioned solvers can handle every arbitrary NLP if the terms of the NLP described in (3-2) get re-arranged in the following form:

$$\begin{aligned} \min_{\mathbf{z}} \quad & a(\mathbf{z}) \\ \text{s.t.} \quad & \mathbf{b}(\mathbf{z}) = \mathbf{0} \\ & \mathbf{c}(\mathbf{z}) \le \mathbf{0} \\ & \mathbf{z}_{lower} \le \mathbf{z} \le \mathbf{z}_{upper} \end{aligned} \tag{3-4}$$

where $\mathbf{z}$ are decision variables of the optimization problem, $a(\mathbf{z})$ is the cost function, $\mathbf{b}(\mathbf{z}) = \mathbf{0}$ are the equality constraints, $\mathbf{c}(\mathbf{z}) \le \mathbf{0}$ are the inequality constraints and $\mathbf{z}_{lower}, \mathbf{z}_{upper}$ are the lower and upper bounds of the decision variables $\mathbf{z}$. The two most commonly used collocation methods are direct transcription (DIRTRAN) [68], that is also called trapezoid collocation, and direct collocation (DIRCOL) [111], that is also called Hermite-Simpson collocation, as shown in Figure 3-9. In DIRTRAN, the state trajectory is represented as a piecewise linear function of time and the control input trajectory as a piecewise-constant function. In DIRCOL, the state trajectory is represented as a piecewise cubic function of time and the control input trajectory as a piecewise linear function of time.

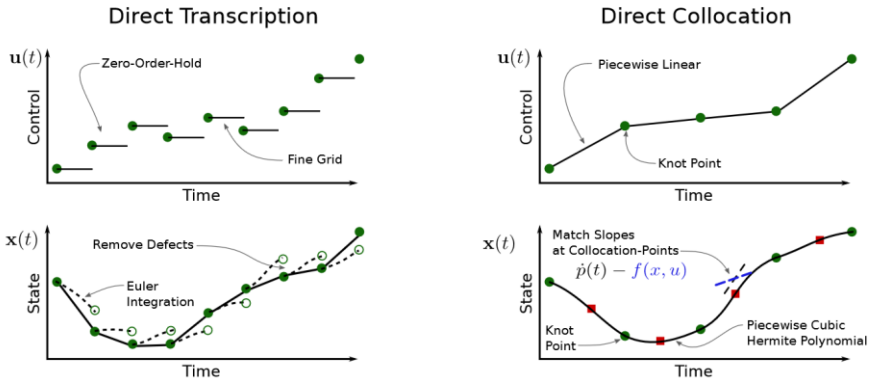


Figure 3-9. Direct Transcription & Direct Collocation [101].

DIRTRAN

The main difference between all collocation methods lies in the approach they adopt to represent the dynamical system constraint, also called defect $\boldsymbol{\Delta}_k \in \mathbb{R}^{(N-1)n}, \forall k = \{0,\ldots,N-1\}$, due to the different representations of state and control trajectories that are used. The simplest dynamic constraint is given by explicit Euler integration scheme. In this scheme, the states trajectories are represented by piecewise linear polynomials and the constraint is defined by the defect:

$$\boldsymbol{\Delta}_k = \mathbf{x}_{k+1} - \mathbf{x}_k - h\mathbf{f}(\mathbf{x}_k, \mathbf{u}_k) = \mathbf{0} \tag{3-5}$$

where $h = t_{k+1} - t_k$ is the constant integration time-step that is being used. In DIRTRAN, an implicit integration scheme is utilized, where the dynamics are integrated using the trapezoidal rule. Here, the defect is defined as follows:

$$\boldsymbol{\Delta}_k = \mathbf{x}_{k+1} - \mathbf{x}_k - \frac{h}{2}\left[\mathbf{f}\left(\mathbf{x}_k, \mathbf{u}_k\right) + \mathbf{f}\left(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}\right)\right] = \mathbf{0} \tag{3-6}$$

The NLP solver will modify the values of $\mathbf{x}_k, \mathbf{u}_k, \mathbf{x}_{k+1}, \mathbf{u}_{k+1}$ to drive the value of $\boldsymbol{\Delta}_k$ to zero. This method is also called trapezoid collocation because equation (3-6) is reminiscent of the trapezoid integration rule. The implicit integration scheme of equation (3-6) has the same order of accuracy as the explicit integration scheme of equation (3-5). In general, implicit integration schemes tend to be more numerically stable than their explicit counterparts [112], and as a result they are chosen frequently in collocation methods. Also, explicit integration schemes are more frequently used in methods where forward rollouts are executed (shooting methods). In collocation methods where the dynamics are enforced as equality constraints between knot points, this attribute is not advantageous.

DIRCOL

The main idea behind the classic DIRCOL is illustrated in Figure 3-10. For each discretization time interval $\left[t_k, t_{k+1}\right]$, of length $h = t_{k+1} - t_k$, the two points at the boundaries of the interval are called “knot” points (blue dots). These points represent the state and control variables $\mathbf{x}_k, \mathbf{u}_k, \mathbf{x}_{k+1}, \mathbf{u}_{k+1}$ that correspond to them. The derivatives of the state variables at the knot points $\dot{\mathbf{x}}_k, \dot{\mathbf{x}}_{k+1}$, can be computed utilizing the continuous system dynamics $\dot{\mathbf{x}} = \mathbf{f}\left(\mathbf{x}, \mathbf{u}\right)$. Having the values of the state and the state derivatives, at the knot points, $\mathbf{x}_k, \mathbf{x}_{k+1}, \dot{\mathbf{x}}_k, \dot{\mathbf{x}}_{k+1}$ available, means that a cubic interpolant polynomial can be constructed. Due to the way that it is constructed, this interpolant polynomial will satisfy the dynamics equations only at the knot points, and not at any time instance within the interval $\left[t_k, t_{k+1}\right]$. In DIRCOL, a point at the middle of the interval $\left[t_k, t_{k+1}\right]$ is defined, at the time instance t_c (also denoted as $t_{k+1/2}$) and with corresponding state and control variables being equal to $\mathbf{x}_c, \mathbf{u}_c$ (also denoted as $\mathbf{x}_{k+1/2}, \mathbf{u}_{k+1/2}$). This point is called “collocation” point (red diamond). Enforcing the constraint $\boldsymbol{\Delta} = \dot{\mathbf{x}}_c - \mathbf{f}\left(\mathbf{x}_c, \mathbf{u}_c\right) = \mathbf{0}$ will result in a polynomial that satisfies the dynamics constraints not only at the knot points but also at the collocation point. The defect $\boldsymbol{\Delta}$ defines the difference between the derivative computed using the polynomial interpolation $\dot{\mathbf{x}}_c$ with the derivative computed using the system dynamics at the collocation point $\mathbf{f}\left(\mathbf{x}_c, \mathbf{u}_c\right)$. The more discretization intervals utilized for optimization, the better the resulting state trajectory will adhere to the system dynamics across the entire interval.

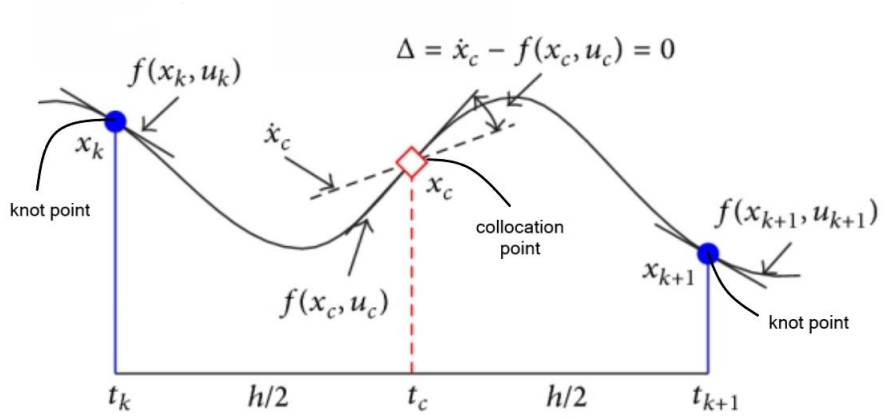


Figure 3-10. Hermite-Simpson collocation method [113].

The procedure followed to construct the defect in DIRCOL can be derived according to [113]. First and foremost, the state $\mathbf{x}(t)$ in each discretization interval $[t_k, t_{k+1}]$ is represented by a third order polynomial of the form:

$$\mathbf{x}(t) = \mathbf{c_0} + \mathbf{c}_1 t + \mathbf{c}_2 t^2 + \mathbf{c}_3 t^3 \tag{3-7}$$

and the state derivative $\dot{\mathbf{x}}(t)$ has the form:

$$\dot{\mathbf{x}}(t) = \mathbf{c}_1 + 2\mathbf{c}_2 t + 3\mathbf{c}_3 t^2 \tag{3-8}$$

where $\mathbf{c_0}, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3$ are the coefficients of the polynomial. The polynomial coefficients can be calculated using the state and state derivative values $\mathbf{x}_k, \mathbf{x}_{k+1}, \dot{\mathbf{x}}_k, \dot{\mathbf{x}}_{k+1}$ at the knot points. For easier computations, the interval is shifted from $[t_k, t_{k+1}]$ to $[0, h]$, so that $t \in [0, h]$, with the values of the states and the state derivatives at the boundaries being the same. The boundary conditions now have the following form:

$$\begin{aligned} \mathbf{x}(0) &= \mathbf{x}_k \\ \mathbf{x}(h) &= \mathbf{x}_{k+1} \\ \dot{\mathbf{x}}(0) &= \dot{\mathbf{x}}_k = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k) \\ \dot{\mathbf{x}}(h) &= \dot{\mathbf{x}}_{k+1} = \mathbf{f}(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}) \end{aligned} \tag{3-9}$$

Evaluating (3-7) and (3-8) at $t = 0$ and at $t = h$ yields the following expression:

$$\begin{bmatrix} \mathbf{x}(0) \\ \dot{\mathbf{x}}(0) \\ \mathbf{x}(h) \\ \dot{\mathbf{x}}(h) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & h & h^2 & h^3 \\ 0 & 1 & 2h & 3h^2 \end{bmatrix} \begin{bmatrix} \mathbf{c_0} \\ \mathbf{c}_1 \\ \mathbf{c}_2 \\ \mathbf{c}_3 \end{bmatrix} \tag{3-10}$$

Inverting the matrix gives the following expression for the polynomial coefficients:

$$\begin{bmatrix} \mathbf{c_0} \\ \mathbf{c}_1 \\ \mathbf{c}_2 \\ \mathbf{c}_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{3}{h^2} & -\frac{2}{h} & \frac{3}{h^2} & -\frac{1}{h} \\ \frac{2}{h^3} & \frac{1}{h^2} & -\frac{2}{h^3} & \frac{1}{h^2} \end{bmatrix} \begin{bmatrix} \mathbf{x}(0) \\ \dot{\mathbf{x}}(0) \\ \mathbf{x}(h) \\ \dot{\mathbf{x}}(h) \end{bmatrix} \tag{3-11}$$

After acquiring the coefficients, they can be substituted in the expressions (3-7), (3-8). Thus, the values of the state and the state derivative at the collocation point $\mathbf{x}_c, \dot{\mathbf{x}}_c$ can be calculated and are the following:

$$\mathbf{x}_c = \mathbf{x}\left(\frac{h}{2}\right) = \frac{1}{2}\left(\mathbf{x}_k + \mathbf{x}_{k+1}\right) + \frac{h}{8}\left[\mathbf{f}\left(\mathbf{x}_k, \mathbf{u}_k\right) - \mathbf{f}\left(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}\right)\right] \tag{3-12}$$

$$\dot{\mathbf{x}}_c = \dot{\mathbf{x}}\left(\frac{h}{2}\right) = -\frac{3}{2h}\left(\mathbf{x}_k - \mathbf{x}_{k+1}\right) - \frac{1}{4}\left[\mathbf{f}\left(\mathbf{x}_k, \mathbf{u}_k\right) + \mathbf{f}\left(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}\right)\right] \tag{3-13}$$

The control input value at the collocation point $\mathbf{u}_c$ is calculated using the piecewise linear interpolation:

$$\mathbf{u}_c = \mathbf{u}\left(\frac{h}{2}\right) = \frac{1}{2}\left(\mathbf{u}_k + \mathbf{u}_{k+1}\right) \tag{3-14}$$

The integration defect $\boldsymbol{\Delta}$ is computed by taking the difference between the interpolated derivative and the derivative calculated using the system dynamics, at the collocation point as follows:

$$\begin{aligned}
\boldsymbol{\Delta} &= \dot{\mathbf{x}}_c - \mathbf{f}\left(\mathbf{x}_c, \mathbf{u}_c\right) \\
&= -\frac{3}{2h}\left(\mathbf{x}_k - \mathbf{x}_{k+1}\right) - \frac{1}{4}\left[\mathbf{f}\left(\mathbf{x}_k, \mathbf{u}_k\right) + \mathbf{f}\left(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}\right)\right] - \mathbf{f}\left(\mathbf{x}_c, \mathbf{u}_c\right) \\
&= \frac{3}{2h}\left\{\mathbf{x}_{k+1} - \mathbf{x}_k - \frac{h}{6}\left[\mathbf{f}\left(\mathbf{x}_k, \mathbf{u}_k\right) + 4\mathbf{f}\left(\mathbf{x}_c, \mathbf{u}_c\right) + \mathbf{f}\left(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}\right)\right]\right\}
\end{aligned} \tag{3-15}$$

The integration defect of equation (3-15) can be rewritten in the following commonly used form:

$$\boldsymbol{\Delta}_k = \mathbf{x}_{k+1} - \mathbf{x}_k - \frac{h}{6}\left[\mathbf{f}\left(\mathbf{x}_k, \mathbf{u}_k\right) + 4\mathbf{f}\left(\mathbf{x}_c, \mathbf{u}_c\right) + \mathbf{f}\left(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}\right)\right] \tag{3-16}$$

The NLP solver will modify the values of $\mathbf{x}_k, \mathbf{u}_k, \mathbf{x}_{k+1}, \mathbf{u}_{k+1}$ to drive the value of $\boldsymbol{\Delta}_k$ to zero. Consequently, the interpolated polynomial will approximate the dynamics of the system according to the accuracy of the numerical integration scheme utilized. This method is also called Hermite-Simpson collocation because the cubic splines used to represent the state trajectory are also called Hermite polynomials and because equation (3-16) is reminiscent of the Simpson's integration rule. Tests conducted in [111] proved that using higher order polynomials to represent the control inputs does not provide any significant advantage.

An additional advantage of implicit integration schemes over their explicit counterparts, in terms of numerical accuracy, is pertinent to their computational cost. The aforementioned approach is equivalent to 3rd order implicit Runge-Kutta integrator (RK3). It is less computationally expensive than the explicit RK3 integrator because it requires less calls of the function that computes the system dynamics, per time step. Invoking the dynamics function and the associated Jacobians constitutes the majority of the computational cost in collocation methods [114]. The explicit RK3 requires 3 calls of dynamics function per time step:

$$
\begin{aligned}
\mathbf{f}_1 &= \mathbf{f}\left(\mathbf{x}_k, \mathbf{u}_k\right) \\
\mathbf{f}_2 &= \mathbf{f}\left(\mathbf{x}_k + \frac{1}{2}h\mathbf{f}_1, \mathbf{u}_k\right) \\
\mathbf{f}_3 &= \mathbf{f}\left(\mathbf{x}_k + 2h\mathbf{f}_1 - h\mathbf{f}_2, \mathbf{u}_k\right) \\
\mathbf{x}_{k+1} &= \mathbf{x}_k + \frac{h}{6}\left(\mathbf{f}_1 + 4\mathbf{f}_2 + \mathbf{f}_3\right) \Rightarrow \\
\boldsymbol{\Delta}_k &= \mathbf{x}_{k+1} - \mathbf{x}_k - \frac{h}{6}\left(\mathbf{f}_1 + 4\mathbf{f}_2 + \mathbf{f}_3\right) = \mathbf{0}
\end{aligned}
\tag{3-17}
$$

However, Hermite-Simpson collocation (equation (3-16)) requires 2 calls of dynamics function per time step, because the end of one time interval, is the beginning of the next one. As a result, the value $\mathbf{f}\left(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}\right)$, that corresponds to the end of a time interval, can be utilized at the next time interval, at the beginning of it as $\mathbf{f}\left(\mathbf{x}_k, \mathbf{u}_k\right)$.

Higher order collocation methods

Finally, higher order polynomials can also be utilized for the representation of state trajectories using the high order Gauss-Lobatto methods [113] or Pseudospectral methods [115]. In general, fewer discretization intervals can be utilized for the optimization, if higher order of implicit integration gets employed. For instance, the fifth-order Gauss-Lobatto method is illustrated in Figure 3-11.

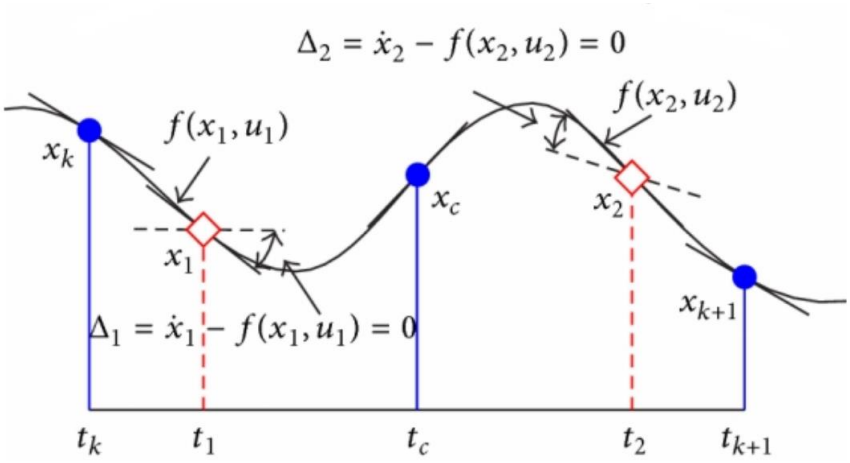


Figure 3-11. Fifth-order Gauss-Lobatto collocation method [113].

Here, three knot points and two collocations points are selected. The values of the state and the state derivative at the three knot points (six values in total), are used to construct the fifth order Hermite polynomial that represents the state trajectory. Then, this interpolant polynomial is used to evaluate the states at the two collocation points. The procedure sketched here is repeated for higher order interpolant polynomials. In general, high-order polynomials represent more accurately the state trajectory but require increased computation time.

However, polynomials used for state trajectory representation that have order higher than three, are not commonly used [101], [116]. In robotics, in general, the system dynamics are not smooth enough (i.e., due to impacts and grasping) to justify the utilization of high order polynomials. Moreover, the models of the system's dynamics might not be accurate enough, as they could be approximations rather than high-fidelity models. Therefore, the loss in accuracy caused by using lower-order polynomials is not of paramount importance. More information about the approximate dynamic models used in legged robotics will be presented

in chapter.4.1 Consequently, lower-order polynomials are encouraged the most. Therefore, these higher order collocation methods will not be examined in this research project.

3.2.3 Multiple Shooting

Multiple shooting [108] (also called parallel shooting) is a hybrid method that combines features of sequential and simultaneous methods, though usually it is considered as a simultaneous method. For instance, in multiple shooting, the decision variables are both the states $\mathbf{x}(t)$ and control inputs $\mathbf{u}(t)$, which are parameterized (discretized) through sets of discrete variables, $\mathbf{x}_k, \mathbf{u}_k$. It is also a simulation-based method, since dynamics are enforced to the solution of the optimization problem using explicit (forward) integration. However, here the trajectory is divided into multiple segments and multiple shots are conducted.

Multiple shooting combines the advantages mostly of collocation but of single shooting as well. For instance, knowledge of the initial state trajectory can be utilized for the initialization of the method, since states are also part of the decision variables of the optimization problem, alongside the control inputs. This integration of states in the decision variables allows collocation to easily handle state and terminal constraints. Moreover, multiple shooting is more robust and more suitable for handling unstable systems compared to single shooting due to the fact that variations in control input affect the state trajectory of the current shot only, and do not affect other shots. Consequently, the nonlinearities that emerge due to the forward rollouts get distributed over the entire horizon and do not increase along the horizon [106]. The more segments the horizon gets divided into, the less the nonlinearities will increase at each segment. However, as the number of segments that the horizon gets divided into increases, the problem size grows, since more decision variables are added to the problem. This effect will become apparent once the downsides of multiple shooting are presented. Finally, efficient state-of-the-art ODE/DAE solvers used in single shooting can also be utilized in multiple shooting as well.

A downside of multiple shooting is that, unlike single shooting, the solution will be dynamically feasible only at the convergence of the NLP solving procedure. As a result, if the user extracts the values of the decision variables before the solver converges, they will not obey the dynamical system constraints and thus this (sub optimal) trajectory cannot be executed by the robot. Also, the NLPs that are derived in multiple shooting problems have smaller dimension, when compared to the ones derived in collocation, but are less sparse. As a result, the computational cost of each iteration in multiple shooting might be higher than the cost of each collocation iteration, especially when including the ODE solver computational cost [103]. However, efficient state-of-the-art ODE/DAE solvers utilized in multiple shooting may bridge that gap and result in multiple shooting solvers are as efficient as solvers used in collocation problems, in terms of CPU time per iteration [103].

Optimizers dedicated to solving multiple shooting problems follow the procedure sketched here. Firstly, an initial guess of the control trajectory is made. Secondly, the state trajectory is divided into many segments and given that initial guess, the system dynamics get integrated forward in time, in each segment, as shown in Figure 3-12. Thus, many shots of smaller duration are being conducted instead of a single shot that covers the entire duration (horizon) of the problem. Each of these shots, except the first one, use an artificial state value $\mathbf{s}_i$ (green dots in the state trajectory plot) as initial conditions. These artificial state values are unknowns and will be part of the decision variables of the problem along with the control inputs [103]. To assure that the entire trajectory is continuous, i.e. the terminal state of each shot is the same

as the initial state of the next shot, continuity conditions are imposed to remove these defects. Then checks are conducted to verify if the constraints are violated or not. If the constraints are violated, a modification of the control inputs gets calculated, and then it gets applied to the control trajectory thus updating its parameterization. This process is repeated until the constraints are satisfied and desired convergence criteria are met.

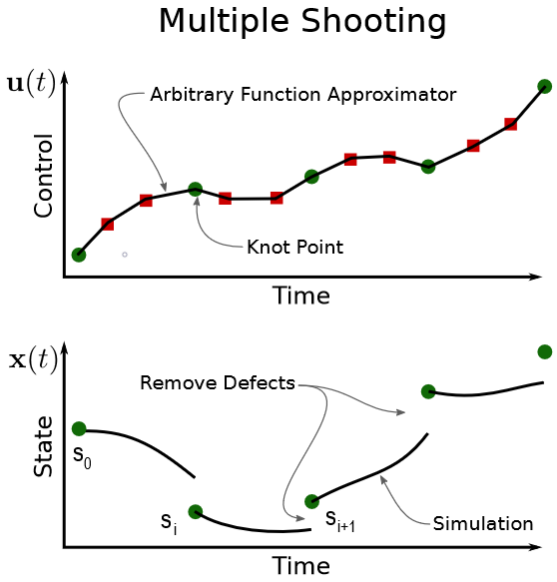


Figure 3-12. Multiple Shooting [101].

In multiple shooting problems, as it was mentioned earlier, the derived NLPs are smaller but less sparse than the ones derived in collocation problems. As a result, numerical optimization techniques, like SQP or IP methods, utilized in collocation are not the ideal choice for multiple shooting. However, these problems have also special, dense structures similar to the structures that single shooting problems have [108]. Therefore, these problems are solved more efficiently by algorithms based on DDP, that are multiple shooting variants of DDP [59], [62], [117], [118].

3.3 Differential Dynamic Programming (DDP)

In DP, a function named the value function or optimal cost-to-go $V_k(\mathbf{x}_k) \in \mathbb{R}, \forall k = \{0,\ldots,N\}$ is defined. It expresses the (minimal) cost J^* incurred, for completing a particular task, when starting from the state $\mathbf{x}_k$ at time k, assuming that the optimal action $\mathbf{u}^*$ is taken. It can be calculated starting backwards from the boundary condition, which is the terminal cost. The cost-to-go $V_k(\mathbf{x})$ is computed recursively, backwards in time, by applying Bellman's Principle of Optimality [48], [119], as follows:

$$V_N = l_N(\mathbf{x}_N) \tag{3-18}$$

$$V_k = \min_{\mathbf{u}_k}\left\{l(\mathbf{x}_k,\mathbf{u}_k) + V_{k+1}(\mathbf{f}(\mathbf{x}_k,\mathbf{u}_k))\right\}, \quad \forall k = \{N-1,\ldots,0\} \tag{3-19}$$

The equation (3-19) is known as the discrete time HJB equation. Also, the action-value function or Q-function $S_k(\mathbf{x}_k,\mathbf{u}_k), \forall k = \{0,\ldots,N-1\}$ is defined as follows:

$$S_k(\mathbf{x}_k,\mathbf{u}_k) = l(\mathbf{x}_k,\mathbf{u}_k) + V_{k+1}(\mathbf{f}(\mathbf{x}_k,\mathbf{u}_k)) \tag{3-20}$$

Action value function is usually denoted as $Q_k(\mathbf{x}_k, \mathbf{u}_k)$. However, in this thesis, the symbol Q is utilized for the quadratic state cost, which is also used in Linear Quadratic Regulator (LQR). Therefore, it will not be used for the action-value function to avoid confusion. Consequently, the expression (3-19) can be rewritten as follows:

$$\begin{aligned} V_k &= \min_{\mathbf{u}_k} \left\{ l(\mathbf{x}_k, \mathbf{u}_k) + V_{k+1}(\mathbf{f}(\mathbf{x}_k, \mathbf{u}_k)) \right\} \\ &= \min_{\mathbf{u}_k} S_k(\mathbf{x}_k, \mathbf{u}_k), \quad \forall k = \{N-1, \ldots, 0\} \end{aligned} \tag{3-21}$$

Acquiring the value function $V_k(\mathbf{x}_k)$, is equivalent to acquiring the desired optimal control law (policy) that is given by the following expression:

$$\begin{aligned} \mathbf{u}_k{}^* &= \arg\min_{\mathbf{u}_k} \left\{ l(\mathbf{x}_k, \mathbf{u}_k) + V_{k+1}(\mathbf{f}(\mathbf{x}_k, \mathbf{u}_k)) \right\} \\ &= \arg\min_{\mathbf{u}_k} S_k(\mathbf{x}_k, \mathbf{u}_k) \end{aligned} \tag{3-22}$$

DP optimality conditions are sufficient for finding a global optimum, in contrast to TO where only locally optimal solutions are guaranteed to be found [45]. Also, in DP, stochastic dynamics can be taken into account, in contrast to the methods based on Pontryagin's Maximum Principle (indirect methods) [45]. As mentioned in chapter 3.1.1, the (exact) DP problem, that involves solving numerically a PDE in continuous time [45], is tractable only for low dimensional systems and simple problems, like LQR. In LQR, the value function can be computed analytically and is quadratic [120], [121]. However, if system dynamics are nonlinear, even if the cost function is quadratic, the value function will not be quadratic and may even be impossible to compute analytically. Also the minimization problem defined by (3-19), will be non-convex and difficult to solve numerically for any arbitrary nonlinear system dynamics.

In approximate DP, function approximations of the value function or the action-value function are used to make the aforementioned problem more solvable. This idea is applied on DDP, a direct single shooting method, where 2nd order Taylor (quadratic) approximations are used for the value and action-value function. This idea is also applied on Reinforcement Learning (RL), particularly in Q-learning, where an action-value function is learned from data sampled from hardware or simulation (using deep neural networks) [122] and then optimal controls are found by solving the minimization problem of (3-22). Learning the action-value function instead of the value function means that the model dynamics will not be needed to solve the optimization problem (model-free RL).

Backward pass

Notation: For a function $l(\mathbf{x}, \mathbf{u}): \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$, the following are defined: $\mathbf{l}_\mathbf{x} \equiv \partial l / \partial \mathbf{x} \in \mathbb{R}^n$, $\mathbf{l}_{\mathbf{xx}} \equiv \partial^2 l / \partial \mathbf{x}^2 \in \mathbb{R}^{n \times n}$ and $\mathbf{l}_{\mathbf{xu}} \equiv \partial^2 l / \partial \mathbf{x} \partial \mathbf{u} \in \mathbb{R}^{n \times m}$.

In DDP [48], a 2nd order Taylor approximation of the value function $V_k(\mathbf{x})$ is used:

$$
\begin{aligned}
V_k + \delta V_k &= V_k\left(\mathbf{x}_k + \delta\mathbf{x}_k\right) \\
&\approx V_k\left(\mathbf{x}_k\right) + \left.\frac{\partial V}{\partial \mathbf{x}}\right|_{\mathbf{x}_k}\left(\mathbf{x}-\mathbf{x}_k\right) + \frac{1}{2}\left(\mathbf{x}-\mathbf{x}_k\right)^T \left.\frac{\partial^2 V}{\partial \mathbf{x}^2}\right|_{\mathbf{x}_k}\left(\mathbf{x}-\mathbf{x}_k\right) \\
&= V_k\left(\mathbf{x}_k\right) + \mathbf{V}_{\mathbf{x}_k}{}^T \delta\mathbf{x}_k + \frac{1}{2}\delta\mathbf{x}_k{}^T \mathbf{V}_{\mathbf{xx}_k}\delta\mathbf{x}_k \Rightarrow \\
\delta V_k\left(\mathbf{x}_k\right) &= \mathbf{V}_{\mathbf{x}_k}{}^T \delta\mathbf{x}_k + \frac{1}{2}\delta\mathbf{x}_k{}^T \mathbf{V}_{\mathbf{xx}_k}\delta\mathbf{x}_k
\end{aligned}
\tag{3-23}
$$

where $\mathbf{V}_{\mathbf{x}_k}, \mathbf{V}_{\mathbf{xx}_k}$ denote the gradient and the Hessian of the value function. Their values at the boundary, which is the terminal state $\mathbf{x}_N$ are known and equal to:

$$
\begin{aligned}
\mathbf{V}_{\mathbf{x}_N} &= \left.\frac{\partial V}{\partial \mathbf{x}}\right|_{\mathbf{x}_N} = \left.\frac{\partial l_N}{\partial \mathbf{x}}\right|_{\mathbf{x}_N} = \mathbf{l}_{\mathbf{f},\mathbf{x}} \\
\mathbf{V}_{\mathbf{xx}_N} &= \left.\frac{\partial^2 V}{\partial \mathbf{x}^2}\right|_{\mathbf{x}_N} = \left.\frac{\partial^2 l_N}{\partial \mathbf{x}^2}\right|_{\mathbf{x}_N} = \mathbf{l}_{\mathbf{f},\mathbf{xx}}
\end{aligned}
\tag{3-24}
$$

Similarly, a 2nd order Taylor approximation of the value function $S_k\left(\mathbf{x},\mathbf{u}\right)$ is used:

$$
\begin{aligned}
S_k + \delta S_k &= S_k\left(\mathbf{x}_k + \delta\mathbf{x}_k, \mathbf{u}_k + \delta\mathbf{u}_k\right) \\
&\approx S_k\left(\mathbf{x}_k,\mathbf{u}_k\right) + \left.\frac{\partial S}{\partial \mathbf{x}}\right|_{\mathbf{x}_k,\mathbf{u}_k}\left(\mathbf{x}-\mathbf{x}_k\right) + \left.\frac{\partial S}{\partial \mathbf{u}}\right|_{\mathbf{x}_k,\mathbf{u}_k}\left(\mathbf{u}-\mathbf{u}_k\right) \\
&\quad + \frac{1}{2}\left(\mathbf{x}-\mathbf{x}_k\right)^T \left.\frac{\partial^2 S}{\partial \mathbf{x}^2}\right|_{\mathbf{x}_k,\mathbf{u}_k}\left(\mathbf{x}-\mathbf{x}_k\right) + \frac{1}{2}\left(\mathbf{u}-\mathbf{u}_k\right)^T \left.\frac{\partial^2 S}{\partial \mathbf{u}^2}\right|_{\mathbf{x}_k,\mathbf{u}_k}\left(\mathbf{u}-\mathbf{u}_k\right) \\
&\quad + \frac{1}{2}\left(\mathbf{u}-\mathbf{u}_k\right)^T \left.\frac{\partial^2 S}{\partial \mathbf{u}\partial \mathbf{x}}\right|_{\mathbf{x}_k,\mathbf{u}_k}\left(\mathbf{x}-\mathbf{x}_k\right) + \frac{1}{2}\left(\mathbf{x}-\mathbf{x}_k\right)^T \left.\frac{\partial^2 S}{\partial \mathbf{x}\partial \mathbf{u}}\right|_{\mathbf{x}_k,\mathbf{u}_k}\left(\mathbf{u}-\mathbf{u}_k\right) \\
&= S_k\left(\mathbf{x}_k,\mathbf{u}_k\right) + \begin{bmatrix}\mathbf{S}_{\mathbf{x}_k}\\ \mathbf{S}_{\mathbf{u}_k}\end{bmatrix}^T \begin{bmatrix}\delta\mathbf{x}_k\\ \delta\mathbf{u}_k\end{bmatrix} + \frac{1}{2}\begin{bmatrix}\delta\mathbf{x}_k\\ \delta\mathbf{u}_k\end{bmatrix}^T \begin{bmatrix}\mathbf{S}_{\mathbf{xx}_k} & \mathbf{S}_{\mathbf{xu}_k}\\ \mathbf{S}_{\mathbf{ux}_k} & \mathbf{S}_{\mathbf{uu}_k}\end{bmatrix}\begin{bmatrix}\delta\mathbf{x}_k\\ \delta\mathbf{u}_k\end{bmatrix} \Rightarrow \\
\delta S_k\left(\mathbf{x}_k,\mathbf{u}_k\right) &= \begin{bmatrix}\mathbf{S}_{\mathbf{x}_k}\\ \mathbf{S}_{\mathbf{u}_k}\end{bmatrix}^T \begin{bmatrix}\delta\mathbf{x}_k\\ \delta\mathbf{u}_k\end{bmatrix} + \frac{1}{2}\begin{bmatrix}\delta\mathbf{x}_k\\ \delta\mathbf{u}_k\end{bmatrix}^T \begin{bmatrix}\mathbf{S}_{\mathbf{xx}_k} & \mathbf{S}_{\mathbf{xu}_k}\\ \mathbf{S}_{\mathbf{ux}_k} & \mathbf{S}_{\mathbf{uu}_k}\end{bmatrix}\begin{bmatrix}\delta\mathbf{x}_k\\ \delta\mathbf{u}_k\end{bmatrix}
\end{aligned}
\tag{3-25}
$$

Combining the Taylor expansions described in equations (3-23), (3-25) with the definition of the action-value function described in equation (3-20) results in the following expressions for the action-value function's Jacobians and Hessians:

$$
\begin{aligned}
\mathbf{S}_{\mathbf{x}_k} &= \mathbf{l}_{\mathbf{x}_k} + \mathbf{f}_{\mathbf{x}_k}{}^T \mathbf{V}_{\mathbf{x}_{k+1}} \\
\mathbf{S}_{\mathbf{u}_k} &= \mathbf{l}_{\mathbf{u}_k} + \mathbf{f}_{\mathbf{u}_k}{}^T \mathbf{V}_{\mathbf{x}_{k+1}} \\
\mathbf{S}_{\mathbf{xx}_k} &= \mathbf{l}_{\mathbf{xx}_k} + \mathbf{f}_{\mathbf{x}_k}{}^T \mathbf{V}_{\mathbf{xx}_{k+1}} \mathbf{f}_{\mathbf{x}_k} + \mathbf{V}_{\mathbf{x}_{k+1}} \cdot \mathbf{f}_{\mathbf{xx}_k} \\
\mathbf{S}_{\mathbf{uu}_k} &= \mathbf{l}_{\mathbf{uu}_k} + \mathbf{f}_{\mathbf{u}_k}{}^T \mathbf{V}_{\mathbf{xx}_{k+1}} \mathbf{f}_{\mathbf{u}_k} + \mathbf{V}_{\mathbf{x}_{k+1}} \cdot \mathbf{f}_{\mathbf{uu}_k} \\
\mathbf{S}_{\mathbf{ux}_k} &= \mathbf{l}_{\mathbf{ux}_k} + \mathbf{f}_{\mathbf{u}_k}{}^T \mathbf{V}_{\mathbf{xx}_{k+1}} \mathbf{f}_{\mathbf{x}_k} + \mathbf{V}_{\mathbf{x}_{k+1}} \cdot \mathbf{f}_{\mathbf{ux}_k} \\
\mathbf{S}_{\mathbf{xu}_k} &= \mathbf{S}_{\mathbf{ux}_k}{}^T
\end{aligned}
\tag{3-26}
$$

where $\mathbf{l}_{\mathbf{x}_k}, \mathbf{l}_{\mathbf{u}_k}$ and $\mathbf{l}_{\mathbf{xx}_k}, \mathbf{l}_{\mathbf{uu}_k}, \mathbf{l}_{\mathbf{ux}_k}$ are the Jacobians and the Hessians of the cost function; and $\mathbf{f}_{\mathbf{x}_k}, \mathbf{f}_{\mathbf{u}_k}$ and $\mathbf{f}_{\mathbf{xx}_k}, \mathbf{f}_{\mathbf{uu}_k}, \mathbf{f}_{\mathbf{ux}_k}$ are the Jacobians and the Hessians of the discretized system dynamics. Also, the last terms of the equations (3-26) denote the product of a vector with a

tensor. The Hessians $\mathbf{f}_{\mathbf{xx}_k}, \mathbf{f}_{\mathbf{uu}_k}, \mathbf{f}_{\mathbf{ux}_k}$ are rank-3 tensors that are often ignored from the expressions of the Hessians of the action-value function, to avoid the computationally expensive tensor-vector multiplications [106]. The version that includes the tensor terms is often called DDP and it corresponds to the Newton method since the step that it computes is nearly identical to the full Newton step [123]. The version that approximates the Hessians of the action-value function, by neglecting the tensor terms (Gauss-Newton (GN) Hessian approximation), is often referred to as Iterative Linear Quadratic Regulator (iLQR) [46] and corresponds to the GN method. It can also be referred to as DDP algorithm with GN approximation instead of iLQR. The DDP method, like Newton's method, has quadratic convergence while the iLQR variant, like GN method, has linear convergence. As a result, DDP can converge to the (local) optimal solution in fewer iterations than iLQR would require. However, iLRQ's iterations are significantly less expensive to compute when compared to the ones in DDP. Consequently, iLQR exhibits a considerably faster convergence rate than DDP. In this work, the GN approximation is adopted. However, it is important to note that there has been some recent work which attempts to compute, in an efficient way, these neglected Hessian terms [124], [125].

Using the quadratic approximation of the Q-function, equation (3-21) is rewritten as follows:

$$\begin{aligned}\delta V_k &= \min_{\delta\mathbf{u}_k} \delta S_k \\ &= \min_{\delta\mathbf{u}_k} \left\{ \mathbf{S}_{\mathbf{x}_k} \delta\mathbf{x}_k + \mathbf{S}_{\mathbf{u}_k} \delta\mathbf{u}_k + \frac{1}{2}\delta\mathbf{x}_k{}^T \mathbf{S}_{\mathbf{xx}_k} \delta\mathbf{x}_k + \frac{1}{2}\delta\mathbf{u}_k{}^T \mathbf{S}_{\mathbf{uu}_k} \delta\mathbf{u}_k + \frac{1}{2}\delta\mathbf{x}_k{}^T \mathbf{S}_{\mathbf{xu}_k} \delta\mathbf{u}_k + \frac{1}{2}\delta\mathbf{u}_k{}^T \mathbf{S}_{\mathbf{ux}_k} \delta\mathbf{x}_k \right\}\end{aligned} \tag{3-27}$$

Minimizing δS_k w.r.t. the control modification $\delta\mathbf{u}_k$, gives optimal control modification $\delta\mathbf{u}^*{}_k$ for some state perturbation $\delta\mathbf{x}_k$. This can be accomplished by calculating the gradient of δS_k w.r.t $\delta\mathbf{u}_k$ and setting it equal to zero.

$$\frac{\partial \delta S_k}{\partial \delta\mathbf{u}_k} = \mathbf{S}_{\mathbf{u}_k} + \mathbf{S}_{\mathbf{uu}_k} \delta\mathbf{u}_k + \frac{1}{2}\mathbf{S}_{\mathbf{xu}_k}{}^T \delta\mathbf{x}_k + \frac{1}{2}\mathbf{S}_{\mathbf{ux}_k} \delta\mathbf{x}_k = \mathbf{0} \tag{3-28}$$

Thus, the optimal control modification $\delta\mathbf{u}^*{}_k$ is given by solving for $\delta\mathbf{u}_k$:

$$\delta\mathbf{u}^*{}_k \left(\delta\mathbf{x}_k\right) = \arg\min_{\delta\mathbf{u}_k} S_k \left(\delta\mathbf{x}_k, \delta\mathbf{u}_k\right) = -\mathbf{S}_{\mathbf{uu}_k}{}^{-1} \left(\mathbf{S}_{\mathbf{u}_k} + \mathbf{S}_{\mathbf{ux}_k} \delta\mathbf{x}_k\right) = \mathbf{k}_k + \mathbf{K}_k \delta\mathbf{x}_k \tag{3-29}$$

This locally linear feedback policy consists of a feed-forward (open-loop) term $\mathbf{k}_k \triangleq -\mathbf{S}_{\mathbf{uu}_k}^{-1} \mathbf{S}_{\mathbf{u}_k}$ and a linear feedback gain term $\mathbf{K}_k \triangleq -\mathbf{S}_{\mathbf{uu}_k}^{-1} \mathbf{S}_{\mathbf{ux}_k}$. The computation of $\mathbf{k}_k$ is called the feed-forward sub-problem, while the computation $\mathbf{K}_k$ of is called the feed-back sub-problem. Substituting the policy $\delta\mathbf{u}^*{}_k$ into the expansion of action-value function in (3-25) gives the following expression:

$$\delta S_k \left(\mathbf{x}_k, \mathbf{u}_k\right) = \begin{bmatrix} \mathbf{S}_{\mathbf{x}_k} \\ \mathbf{S}_{\mathbf{u}_k} \end{bmatrix}^T \begin{bmatrix} \delta\mathbf{x}_k \\ \mathbf{k}_k + \mathbf{K}_k \delta\mathbf{x}_k \end{bmatrix} + \frac{1}{2} \begin{bmatrix} \delta\mathbf{x}_k \\ \mathbf{k}_k + \mathbf{K}_k \delta\mathbf{x}_k \end{bmatrix}^T \begin{bmatrix} \mathbf{S}_{\mathbf{xx}_k} & \mathbf{S}_{\mathbf{xu}_k} \\ \mathbf{S}_{\mathbf{ux}_k} & \mathbf{S}_{\mathbf{uu}_k} \end{bmatrix} \begin{bmatrix} \delta\mathbf{x}_k \\ \mathbf{k}_k + \mathbf{K}_k \delta\mathbf{x}_k \end{bmatrix} \tag{3-30}$$

Equating the result with equation (3-27) and combining it with the expansion of value function in (3-23), provides closed-form expressions of $\mathbf{V}_{\mathbf{x}_k}, \mathbf{V}_{\mathbf{xx}_k}, \Delta V_k$:

$$
\begin{aligned}
\mathbf{V}_{\mathbf{x}_k} &= \mathbf{S}_{\mathbf{x}_k} + \mathbf{K}_k{}^T \mathbf{S}_{\mathbf{uu}_k} \mathbf{k}_k + \mathbf{K}_k{}^T \mathbf{S}_{\mathbf{u}_k} + \mathbf{S}_{\mathbf{ux}_k}{}^T \mathbf{k}_k \\
\mathbf{V}_{\mathbf{xx}_k} &= \mathbf{S}_{\mathbf{xx}_k} + \mathbf{K}_k{}^T \mathbf{S}_{\mathbf{uu}_k} \mathbf{K}_k + \mathbf{K}_k{}^T \mathbf{S}_{\mathbf{ux}_k} + \mathbf{S}_{\mathbf{ux}_k}{}^T \mathbf{K}_k \\
\Delta V_k &= \mathbf{k}_k{}^T \mathbf{S}_{\mathbf{u}_k} + \frac{1}{2} \mathbf{k}_k{}^T \mathbf{S}_{\mathbf{uu}_k} \mathbf{k}_k
\end{aligned}
\tag{3-31}
$$

An iterative recursive procedure is conducted in DDP, backwards in time, to calculate the terms of the optimal feedback policy $\mathbf{k}_k, \mathbf{K}_k$ as well as the terms of the quadratic model of the value function $\mathbf{V}_{\mathbf{x}_k}, \mathbf{V}_{\mathbf{xx}_k}$, and the expected cost change ΔV_k for every time step k. This procedure is called backward pass. The backward pass begins at time step $k = N$, where the values of $V_N, \mathbf{V}_{\mathbf{x}_N}, \mathbf{V}_{\mathbf{xx}_N}$ are known. Then equations (3-26), (3-29), (3-31) are recursively computed for decreasing $k = N-1, \ldots, 1$.

It is important to note that, as it was also mentioned about single shooting in chapter 3.2.1, DDP/iLQR suffers from numerical ill-conditioning on long horizons, due to the error accumulation that occurs during the backward pass. Consequently, the optimal control modification calculated on long trajectories will be inaccurate and the solver will eventually diverge. This problem is also present in machine learning when performing backpropagation on a very deep neural network with many stages. In machine learning this problem is often referred to as “vanishing/exploding gradient” problem and in control theory is often called as “tail wagging the dog” problem [107].

Forward pass

When the backward pass gets completed for the time step $k = N$, a procedure called forward pass get initiated. The forward pass computes the new, updated state and control trajectory $\hat{\mathbf{X}}, \hat{\mathbf{U}}$ by applying the control modification to the nominal control trajectory $\mathbf{U}$.

$$
\begin{aligned}
\hat{\mathbf{x}}_0 &= \mathbf{x}_0 \\
\delta \mathbf{x}_k &= \hat{\mathbf{x}}_k - \mathbf{x}_k \\
\delta \mathbf{u}_k &= \mathbf{k}_k + \mathbf{K}_k \delta \mathbf{x}_k \\
\hat{\mathbf{u}}_k &= \mathbf{u}_k + \delta \mathbf{u}_k \\
\hat{\mathbf{x}}_{k+1} &= \mathbf{f}\left(\hat{\mathbf{x}}_k, \hat{\mathbf{u}}_k\right), \quad \forall k = \{0, \ldots, N-1\}
\end{aligned}
\tag{3-32}
$$

where $\mathbf{x}_{k+1} = \mathbf{f}\left(\mathbf{x}_k, \mathbf{u}_k\right)$ are the system dynamics in discrete time. This backward-forward process is repeated until convergence to the locally optimal trajectory. Some convergence criteria that are commonly used in practice are presented in [121]. Modified versions of the forward and backward pass, that exhibit better numerical properties, are often utilized. These versions incorporate two techniques common in nonlinear optimization: line-search and regularization.

Line search

Line search, which is performed on $\mathbf{k}_k$, along the descent direction, is added to ensure an adequate reduction in cost or even avoid divergence [121]. This happens because in an arbitrary nonlinear system, the updated trajectory may lay too far from the region where the model is valid [48]. Thus, equation (3-32) is modified as follows:

$$
\begin{aligned}
\hat{\mathbf{x}}_0 &= \mathbf{x}_0 \\
\delta\mathbf{x}_k &= \hat{\mathbf{x}}_k - \mathbf{x}_k \\
\delta\mathbf{u}_k &= a\mathbf{k}_k + \mathbf{K}_k\delta\mathbf{x}_k \\
\hat{\mathbf{u}}_k &= \mathbf{u}_k + \delta\mathbf{u}_k \\
\hat{\mathbf{x}}_{k+1} &= \mathbf{f}\left(\hat{\mathbf{x}}_k, \hat{\mathbf{u}}_k\right), \quad \forall k = \{0,\ldots,N-1\}
\end{aligned}
\tag{3-33}
$$

where $a,\ a \in [0,1]$ is a backtracking line-search parameter. This parameter is initially set equal to 1, gets iteratively reduced and the forward pass gets restarted until the adequate reduction in cost is achieved. More specifically, after applying equation (3-33), and thus obtaining candidate state and control trajectory $\hat{\mathbf{X}}, \hat{\mathbf{U}}$, the ration of the actual cost reduction to the expected cost reduction z gets calculated:

$$
z = \frac{J(\mathbf{X},\mathbf{U}) - J\left(\hat{\mathbf{X}},\hat{\mathbf{U}}\right)}{-\Delta V(a)} \tag{3-34}
$$

where

$$
\Delta V(a) = a\sum_{k=0}^{N-1}\mathbf{k}_k{}^T\mathbf{S}_{\mathbf{u}_k} + \frac{a^2}{2}\sum_{k=0}^{N-1}\mathbf{k}_k{}^T\mathbf{S}_{\mathbf{uu}_k}\mathbf{k}_k = a\Delta V_1 + \frac{a^2}{2}\Delta V_2 \tag{3-35}
$$

is the expected cost reduction calculated using $a\mathbf{k}_k$ instead of $\mathbf{k}_k$ in the equation (3-31). If $z \in [\beta_1, \beta_2],\ \beta_1 > 0$, then the iteration is accepted, and the line search has converged. If z does not lie in this interval, parameter a gets updated, using the expression $a = \gamma\alpha$, where $\gamma \in [0,1]$ is a backtracking scaling parameter. The value $\gamma = 0.5$ is used often in practice. Some typical values for the aforementioned hyperparameters can be found in [121].

Regularization

Regularization is added to ensure the invertibility of the Hessian $\mathbf{S}_{\mathbf{uu}_k}$ .To be more precise, as in Newton's method, to ensure a descent direction, regularization must be used so that the Hessian stays always positive definite (pd) [49]. There are two options available for regularization. The first option, as it is analyzed in [126], penalizes deviations from a control trajectory. It adds a diagonal matrix to the control-cost Hessian $\mathbf{S}_{\mathbf{uu}_k}$.

$$
\tilde{\mathbf{S}}_{\mathbf{uu}_k} = \mathbf{l}_{\mathbf{uu}_k} + \mathbf{f}_{\mathbf{u}_k}{}^T\mathbf{V}_{\mathbf{xx}_{k+1}}\mathbf{f}_{\mathbf{u}_k} + \mathbf{V}_{\mathbf{x}_{k+1}}\cdot\mathbf{f}_{\mathbf{uu}_k} + \rho\mathbf{I}_m = \mathbf{S}_{\mathbf{uu}_k} + \rho\mathbf{I}_m \tag{3-36}
$$

where $\rho,\ \rho \geq 0$, also known as Levenberg-Marquardt parameter, acts as a damping parameter, that gets modified when feed-forward sub-problems fails to get solved and $\mathbf{I}_m$ is the $m \times m$ identity matrix. The second option, as it is analyzed in [48], penalizes deviations from the state trajectory.

$$
\begin{aligned}
\tilde{\mathbf{S}}_{\mathbf{uu}_k} &= \mathbf{l}_{\mathbf{uu}_k} + \mathbf{f}_{\mathbf{u}_k}{}^T\left(\mathbf{V}_{\mathbf{xx}_{k+1}} + \rho\mathbf{I}_n\right)\mathbf{f}_{\mathbf{u}_k} + \mathbf{V}_{\mathbf{x}_{k+1}}\cdot\mathbf{f}_{\mathbf{uu}_k} \\
\tilde{\mathbf{S}}_{\mathbf{ux}_k} &= \mathbf{l}_{\mathbf{ux}_k} + \mathbf{f}_{\mathbf{u}_k}{}^T\left(\mathbf{V}_{\mathbf{xx}_{k+1}} + \rho\mathbf{I}_n\right)\mathbf{f}_{\mathbf{x}_k} + \mathbf{V}_{\mathbf{x}_{k+1}}\cdot\mathbf{f}_{\mathbf{ux}_k}
\end{aligned}
\tag{3-37}
$$

Where $\mathbf{I}_n$ is the $n \times n$ identity matrix. In both options, the value of ρ gets increased if a non-positive definite $\mathbf{S}_{\mathbf{uu}_k}$ is encountered and the backward pass gets repeated. If $\mathbf{S}_{\mathbf{uu}_k}$ is positive definite, the value of ρ gets decreased. An important note is that, while the regularized Hessian $\tilde{\mathbf{S}}_{\mathbf{uu}_k}$ (and $\tilde{\mathbf{S}}_{\mathbf{ux}_k}$) is used for the computation of the terms of the optimal feedback policy $\mathbf{k}_k, \mathbf{K}_k$, the Hessian $\mathbf{S}_{\mathbf{uu}_k}$ is utilized for the computation of the quadratic model of the value

function, i.e., $\mathbf{V}_{\mathbf{x}_k}, \mathbf{V}_{\mathbf{x}\mathbf{x}_k}, \Delta V_k$, during the backward pass. This strategy is more robust and more preferable in general [127].

Another instance in which regularization is necessary is when the line search procedure does not make progress after a pre-specified number of iterations [121]. In this case, the forward pass gets abandoned and the regularization term ρ is increased prior to starting the backward pass. Increasing ρ, makes the partial Hessian resemble more the identity matrix. It also "moves" the Newton (or GN) step direction closer to the gradient descent direction, that is more reliable when the current value of the decision variables is not close to the local optimum. However, when the current value of the decision variables current is close to the local minimum, ρ should rapidly go to zero to achieve faster convergence. To meet the aforementioned requirements, a quadratic modification scheme of the regularization parameter ρ was adopted, as in [48]. This scheme requires the selection of a minimal value of ρ, $\rho_{\min}$, and a minimal modification factor Δ_0. It has the following form:

$$\begin{aligned}
&\text{Increase } \rho\text{:}\\
&\quad \Delta \leftarrow \max\left(\Delta_0, \Delta \cdot \Delta_0\right)\\
&\quad \rho \leftarrow \max\left(\rho_{\min}, \rho \cdot \Delta\right)\\
&\text{Decrease } \rho\text{:}\\
&\quad \Delta \leftarrow \min\left(\frac{1}{\Delta_0}, \frac{\Delta}{\Delta_0}\right)\\
&\quad \rho \leftarrow \begin{cases} \rho \cdot \Delta & \text{if} \quad \rho \cdot \Delta > \rho_{\min} \\ 0 & \text{if} \quad \rho \cdot \Delta < \rho_{\min} \end{cases}
\end{aligned} \tag{3-38}$$

Handling Constraints

A downside of DDP based methods is that they cannot natively handle constraints on either states or control inputs. They have to be extensively modified to do so. The scenario that is first examined is when only control limits are present. The inequality constraints in that scenario have the form:

$$\underline{\mathbf{u}} \le \mathbf{u}_k \le \bar{\mathbf{u}}, \qquad \forall k = \{0, \ldots, N-1\} \tag{3-39}$$

where $\underline{\mathbf{u}}, \bar{\mathbf{u}}$ are the lower and the upper control input bounds. These kinds of constraints are called box inequality constraints. Box constraints are very common in practice due to the fact that they accurately model the limitations that the vast majority of mechanical actuators has. The first way to enforce box constraints is to clamp the control inputs in the forward pass. Consequently, the control inputs will be modified as follows:

$$\mathbf{u}_k \leftarrow \min\left(\max\left(\mathbf{u}_k, \underline{\mathbf{u}}\right), \bar{\mathbf{u}}\right) \tag{3-40}$$

However, this simple clamping does not take into consideration the clamped direction during the inversion of $\mathbf{S}_{\mathbf{u}\mathbf{u}_k}$, and therefore it does not yield satisfying results [49]. Another way to enforce constraints is to impose a squashing function $\mathbf{s}(\mathbf{u})$ (i.e., a sigmoid) on the control inputs. Consequently, the system dynamics will have the form:

$$\mathbf{x}_{k+1} = \mathbf{f}\left(\mathbf{x}_k, \mathbf{s}\left(\mathbf{u}_k\right)\right) \tag{3-41}$$

where $\mathbf{s}(\cdot)$ is an element-wise sigmoid operator with the following properties:

$$\lim_{\mathbf{u} \to -\infty} \mathbf{s}(\mathbf{u}) = \underline{\mathbf{u}}, \quad \lim_{\mathbf{u} \to \infty} \mathbf{s}(\mathbf{u}) = \bar{\mathbf{u}} \tag{3-42}$$

One example of such sigmoid function that can be utilized is the following:

$$\mathbf{s}(\mathbf{u}) = \frac{\bar{\mathbf{u}} - \underline{\mathbf{u}}}{2} \tanh(\mathbf{u}) + \frac{\bar{\mathbf{u}} + \underline{\mathbf{u}}}{2} \tag{3-43}$$

It has been shown that the simple usage of squashing functions impairs the convergence rate (sublinear convergence), since it introduces non-linearity to the problem [49], [128].

The method adopted in this solver implementation takes into consideration the control input constraints while minimizing the quadratic model of the action-value function. More specifically, with box constraints, the problem of equation (3-29) will have the form:

$$\begin{aligned} \delta\mathbf{u}^*{}_k &= \arg\min_{\delta\mathbf{u}_k} S_k\left(\delta\mathbf{x}_k, \delta\mathbf{u}_k\right) \\ \text{s.t.} \quad & \underline{\mathbf{u}} \le \mathbf{u}_k + \delta\mathbf{u}_k \le \bar{\mathbf{u}} \end{aligned} \tag{3-44}$$

Consequently, a box-constrained QP can be solved in the backward pass, at each time-step. During the backward pass the feed-forward and the feed-back sub-problems have to be solved. The feed-forward term $\mathbf{k}_k$ is obtained by solving the following QP:

$$\begin{aligned} \mathbf{k}_k &= \arg\min_{\delta\mathbf{u}_k} \frac{1}{2} \delta\mathbf{u}_k{}^T \mathbf{S}_{\mathbf{uu}_k} \delta\mathbf{u}_k + \mathbf{S}_{\mathbf{u}_k}{}^T \delta\mathbf{u}_k \\ \text{s.t.} \quad & \underline{\mathbf{u}} \le \mathbf{u}_k + \delta\mathbf{u}_k \le \bar{\mathbf{u}} \end{aligned} \tag{3-45}$$

A custom QP solver is developed to solve the problem of (3-45) that uses the Projected-Newton QP algorithm [129], in similar fashion to the one described in [49]. It is an active set method designed for problems with simple constraints. This algorithm iteratively identifies the active constraints and moves along the free subspace of the Newton step, where the constraints are inactive [106]. It has a similar computational cost to the unconstrained QP if the active set remains unchanged, and thus it has similar performance to the unconstrained DDP algorithm. The Projected-Newton QP algorithm is described in much more detail in [129].

Apart from solving the QP of (3-45), the solver should also provide a decomposition of the free dimensions of $\mathbf{S}_{\mathbf{uu}_k}$ denoted by $\mathbf{S}_{\mathbf{uu},f_k}$ where f_k describes the free dimensions at timestep k, where the bounds are inactive. Similarly, c_k describes the clamped dimensions, where the bounds are active and the decomposition of the clamped dimensions of $\mathbf{S}_{\mathbf{uu}_k}$ is denoted by $\mathbf{S}_{\mathbf{uu},c_k}$. Given the decomposition of $\mathbf{S}_{\mathbf{uu}_k}$, it can be partitioned and rewritten in the following way:

$$\mathbf{S}_{\mathbf{uu}_k} = \begin{bmatrix} \mathbf{S}_{\mathbf{uu},f_k} & \mathbf{S}_{\mathbf{uu},fc_k} \\ \mathbf{S}_{\mathbf{uu},cf_k} & \mathbf{S}_{\mathbf{uu},c_k} \end{bmatrix} \tag{3-46}$$

After acquiring $\mathbf{S}_{\mathbf{uu},f_k}$, while solving the QP, the feedback gain term $\mathbf{K}_k$ can be computed as follows:

$$\mathbf{K}_k = -\hat{\mathbf{S}}^{-1}_{\mathbf{uu},f_k} \mathbf{S}_{\mathbf{ux}_k} \tag{3-47}$$

where $\hat{\mathbf{S}}^{-1}_{\mathbf{uu},f_k}$ is the control Hessian of the free subspace and is computed internally based on the factorization of $\mathbf{S}^{-1}_{\mathbf{uu},f_k}$. It is obvious that the rows of $\mathbf{K}_k$ that correspond to the clamped control inputs, will be equal to zero. This method designed to handle box constraints on control inputs is often referred to as BOX-DDP. More details about the implementation of the code of Projected-Newton QP algorithm can be found in [130].

The scenario in which state constraints are present is more complicated. Many of these approaches can uniformly handle state and control input constraints. The techniques utilized

to handle any arbitrary constraint combine the DDP algorithm with either an active set or a penalization method. All these methods are explained in detail in [131]. In general, active set methods utilize heuristics to guess which constraints are active. Then, they solve an equality constrained problem, which is a straightforward task when compared to the inequality constrained problem. These methods tend to perform better on optimization problems with few decision variables, since the cost of computing the active set becomes significant in large-scale optimization problems. Thus, penalty methods have been examined. These methods replace the inequality constraints with a penalty term, that is usually quadratic, that is being added to the cost function. When the constraint is being violated, this term becomes infinitely large and thus constraint violations get penalized. If the constraint is not violated, the term that is added to the cost function is equal to zero. However, such methods perform poorly in practice since they suffer from numerical ill-conditioning and poor convergence rate.

To tackle these drawbacks, Augmented Lagrangian (AL) methods have been formulated. In AL, a linear term is also added to the cost in addition to the quadratic penalty term. This linear term consists of a Lagrange multiplier multiplied with the constraints vector. Then the DDP algorithm described before tries to minimize this new sum, comprised of the cost function, the penalty term, and the linear term, instead of the cost function solely. Most of the state-of-the-art DDP based solvers utilize this method. One particular solver is ALTRO (Augmented Lagrangian TRajectory Optimizer) [116] while others are included in software libraries like Control Toolbox (CT) [61], OCS2 (Optimal Control for Switched Systems) [132] and Crocoddyl (Contact RObot COntrol by Differential DYnamic Library) [62], namely the FDDP (Feasibility-prone Differential Dynamic Programming) solver. The ability to handle state constraints favors the development of multiple shooting variants of the DDP algorithm (single shooting), which are often referred to as Gauss-Newton Multiple Shooting (GNMS). Consequently, these packages include both single and multiple shooting variants of DDP.

3.4 Convex Optimization

3.4.1 Background & Overview

To explain how convex optimization is utilized in control applications, some background terminology has to be introduced first. A constraint set $\mathcal{D} \subset \mathbb{R}^d$ is characterized as a convex set if:

$$\forall \mathbf{u}, \mathbf{v} \in \mathcal{D},\ t \in [0,1]:\ \mathbf{u} + t(\mathbf{v} - \mathbf{u}) \in \mathcal{D} \tag{3-48}$$

This definition is equivalent to stating that the set is convex if a line segment that connects any two points inside the set, lies inside the set [133], as illustrated in Figure 3-13.

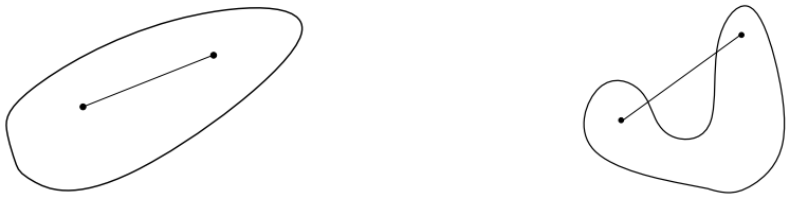


Figure 3-13. Convex and non-convex sets [133].

Some standard examples of convex constraint sets are the following:

$$\begin{aligned}
&\text{Linear Subspace:} && \mathbf{Az}=\mathbf{b}\\
&\text{Half-space/Box/Polytope:} && \mathbf{Az}\le\mathbf{b}\\
&\text{Ellipsoid:} && \mathbf{z}^T\mathbf{Pz}\le 1,\quad \mathbf{P}>0\\
&\text{Cone:} && \left\|\mathbf{z}_{2:d}\right\|_2\le\mathbf{z}_1\quad(\text{Second Order Cone})
\end{aligned} \tag{3-49}$$

A function $f(\mathbf{z}):\mathcal{D}\to\mathbb{R}$ is convex, if $\mathcal{D}$ is a convex set and if:

$$\forall\mathbf{u},\mathbf{v}\in\mathcal{D},\ t\in[0,1]:\ f\left(\mathbf{u}+t(\mathbf{v}-\mathbf{u})\right)\le f(\mathbf{u})+t\left(f(\mathbf{v})-f(\mathbf{u})\right) \tag{3-50}$$

This definition is equivalent to saying that the function is convex is the epigraph of that function (blue colored area) is always a convex set [133], as illustrated in Figure 3-14. It is also equivalent to claiming that a line segment between any two points of the graph of the function (secant) lies above the graph.

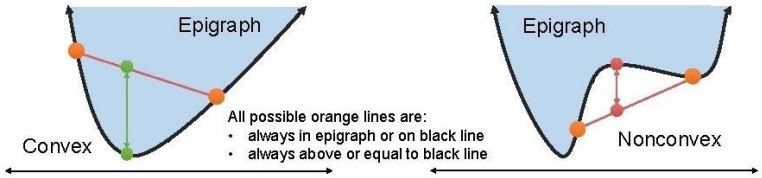


Figure 3-14. Convex and non-convex functions.

Some standard examples of convex cost functions are the following:

$$\begin{aligned}
&\text{Linear:} && f(\mathbf{z})=\mathbf{c}^T\mathbf{z}\\
&\text{Quadratic:} && f(\mathbf{z})=\frac{1}{2}\mathbf{z}^T\mathbf{Qz}+\mathbf{q}^T\mathbf{z},\quad \mathbf{Q}>0\\
&\text{Norm:} && \text{Any vectror norm e.g. } f(\mathbf{z})=\left\|\mathbf{z}\right\|_2
\end{aligned} \tag{3-51}$$

An optimization problem is denoted as a convex optimization problem if a convex cost function $f:\mathcal{D}\to\mathbb{R}$ has to be minimized over a convex constraint set $\mathcal{D}$. In general, a constraint set contains both equality and inequality constraints. Apart from the condition presented before, the set will be convex only if all the equality constraints of the set are linear [133]. In convex optimization problems, any local minimum found is guaranteed to be also the global minimum. In other words, they have a single local optimum which is the global optimum. For non-convex problems, multiple local minima exist and not all of them are global minima, as shown in Figure 3-15.

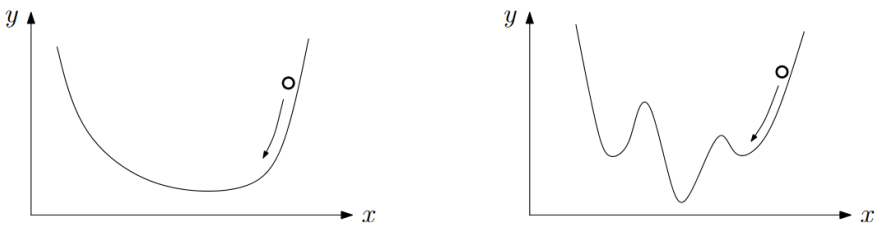


Figure 3-15. Convex and non-convex optimization problems [133].

The most common classes of convex optimization problems are Linear Programs (LPs), Quadratic Programs (QPs), Quadratically Constrained Quadratic Programs (QCQPs) and

Second Order Cone Programs (SOCPs). A LP is an optimization problem with a linear cost function and a constraint set with linear inequality constraints:

$$\begin{aligned} \min_{\mathbf{z}} \quad & f(\mathbf{z}) = \mathbf{c}^T\mathbf{z} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{z} - \mathbf{b} = \mathbf{0} \\ & \mathbf{C}\mathbf{z} - \mathbf{d} \leq \mathbf{0} \end{aligned} \tag{3-52}$$

where $\mathbf{c} \in \mathbb{R}^d$, $\mathbf{A} \in \mathbb{R}^{p \times d}$, $\mathbf{b} \in \mathbb{R}^p$, $\mathbf{C} \in \mathbb{R}^{q \times d}$ and $\mathbf{d} \in \mathbb{R}^q$. The cost function could also include a constant term i.e., $f(\mathbf{z}) = \mathbf{c}^T\mathbf{z} + \mathbf{c}_0$ without affecting the value of the solution to the problem $\mathbf{z}^*$.

A QP is an optimization problem with a quadratic cost function and a constraint set with linear inequality constraints:

$$\begin{aligned} \min_{\mathbf{z}} \quad & f(\mathbf{z}) = \frac{1}{2}\mathbf{z}^T\mathbf{Q}\mathbf{z} + \mathbf{q}^T\mathbf{z}, \quad \mathbf{Q} \geq 0 \\ \text{s.t.} \quad & \mathbf{A}\mathbf{z} - \mathbf{b} = \mathbf{0} \\ & \mathbf{C}\mathbf{z} - \mathbf{d} \leq \mathbf{0} \end{aligned} \tag{3-53}$$

where $\mathbf{Q} \in \mathbb{R}^{d \times d}$, $\mathbf{q} \in \mathbb{R}^d$, $\mathbf{A} \in \mathbb{R}^{p \times d}$, $\mathbf{b} \in \mathbb{R}^p$, $\mathbf{C} \in \mathbb{R}^{q \times d}$ and $\mathbf{d} \in \mathbb{R}^q$. The matrix $\mathbf{Q}$ is the Hessian matrix of the objective function since: $\nabla^2 f(\mathbf{z}) = \mathbf{Q}$ and $\mathbf{q}$ is the gradient of the objective function. Only if the matrix $\mathbf{Q}$ is positive semi-definite (psd) ($\forall \mathbf{z} \in \mathbb{R}^d : \mathbf{z}^T\mathbf{Q}\mathbf{z} \geq 0$ also denoted as $\mathbf{Q} \geq 0$ or $\mathbf{Q} \in \mathbb{S}_+^d$), then the QP will be convex. Otherwise, it will be a non-convex problem and it might have multiple local minima. In general, convex QPs are significantly easier to solve globally that non-convex ones [133]. It is important to note that, according to the majority of the conventions, quadratic programming refers to the convex case, by definition. The cost function may also include a constant term i.e., $f(\mathbf{z}) = (1/2)\mathbf{z}^T\mathbf{Q}\mathbf{z} + \mathbf{q}^T\mathbf{z} + \mathbf{c}_0$ that is usually neglected without affecting the solution $\mathbf{z}^*$. Additionally, every LP can be written as a QP with $\mathbf{Q} = \mathbf{0}$. As it was also mentioned in the literature review, control techniques based on solving QPs have been formulated in the past years. To be more precise, QP-based MPC has been designed and executed on quadrupeds (e.g., MIT Cheetah) with linearized dynamics and friction pyramid model (linearized friction cone model) [32].

A QCQP is an optimization problem with a quadratic cost function and a constraint set with quadratic (ellipsoidal) inequality constraints:

$$\begin{aligned} \min_{\mathbf{z}} \quad & f(\mathbf{z}) = \frac{1}{2}\mathbf{z}^T\mathbf{Q}_0\mathbf{z} + \mathbf{q}_0{}^T\mathbf{z}, \quad \mathbf{Q}_0 > 0 \\ \text{s.t.} \quad & \mathbf{A}\mathbf{z} - \mathbf{b} = \mathbf{0} \\ & \frac{1}{2}\mathbf{z}^T\mathbf{Q}_i\mathbf{z} + \mathbf{q}_i{}^T\mathbf{z} + \mathbf{c}_i \leq \mathbf{0}, \quad \mathbf{Q}_i > 0, \quad \forall i = \{1, \ldots, m\} \end{aligned} \tag{3-54}$$

where $\mathbf{A} \in \mathbb{R}^{p \times d}$, $\mathbf{b} \in \mathbb{R}^p$, $\mathbf{Q}_0, \mathbf{Q}_i \in \mathbb{R}^{d \times d}$, $\mathbf{q}_0, \mathbf{q}_i \in \mathbb{R}^d$, $\mathbf{c}_i \in \mathbb{R}^d$ and m is the number of all the inequality constraints. Similar to the QP case, the difficulty of the problems is dependent on whether or not the matrices $\mathbf{Q}_0, \mathbf{Q}_i$ are psd or not [134]. If $\mathbf{Q}_0, \mathbf{Q}_i$ are psd, then the problem is classified as a convex QCQP Finally, every QP can be written as a QCQP with $\mathbf{Q}_i = \mathbf{0}$.

A SOQP is an optimization problem with a linear cost function and a constraint set with conic inequality constraints:

$$\begin{aligned} \min_{\mathbf{z}} \quad & f(\mathbf{z}) = \mathbf{c}^T \mathbf{z} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{z} - \mathbf{b} = \mathbf{0} \\ & \left\| \mathbf{C}_i \mathbf{z} + \mathbf{d}_i \right\|_2 \leq \mathbf{e}_i^T \mathbf{z} + \mathbf{f}_i, \quad \forall i = \{1, \ldots, m\} \end{aligned} \tag{3-55}$$

where $\mathbf{c} \in \mathbb{R}^d$, $\mathbf{A} \in \mathbb{R}^{p \times d}$, $\mathbf{b} \in \mathbb{R}^p$, $\mathbf{C}_i \in \mathbb{R}^{q_i \times d}$, $\mathbf{d}_i \in \mathbb{R}^{q_i}$, $\mathbf{e}_i \in \mathbb{R}^d$, $\mathbf{f}_i \in \mathbb{R}$ and m is the number of all the inequality constraints. The term SOCP is derived from the inequality constraint, that corresponds to a second order cone. It is also called quadratic, Lorentz, or ice-cream cone. Additionally, every QCQP can be written as a SOCP by reformulating the objective function as a constraint [135]. Finally, control techniques based on solving SOCPs have also been formulated recently, particularly on rocket landing problems [136], [137].

An advantage of convex optimization problems, in contrast to all the nonlinear optimization problems, is that they do not have numerous local optima that minimize the objective function while also satisfying the constraints. Therefore, finding a local optimum in a convex problem is equivalent to finding the global optimum. In general, the numerical optimization techniques utilized to solve arbitrary optimization problems (i.e., Newton's method) only guarantee to find a locally optimal solution to the problem. Consequently, such techniques are well-suited for convex problems since they do not run the risk of getting stuck on a sub-optimal local solution during the runtime of the optimization. Solvers based on Newton's method can solve convex problems very fast and reliably, since the solver cannot not get stuck in a strange local minimum. They can also guarantee the maximum runtime necessary to do so, for a specific problem size. Consequently, solution time can be bounded, which is ideal for real-time online control applications where it is of paramount importance to utilize controllers that can run at guaranteed rates.

3.4.2 Convex MPC

Convex, discrete time, time varying, MPC problems usually have the following form:

$$\begin{aligned} \min_{\mathbf{x}_{0:N}, \mathbf{u}_{0:N-1}} \quad & J = \frac{1}{2} \sum_{k=0}^{N-1} \left(\mathbf{x}_k - \mathbf{x}_{\mathbf{ref}_k} \right)^T \mathbf{Q}_k \left(\mathbf{x}_k - \mathbf{x}_{\mathbf{ref}_k} \right) + \left(\mathbf{u}_k - \mathbf{u}_{\mathbf{ref}_k} \right)^T \mathbf{R}_k \left(\mathbf{u}_k - \mathbf{u}_{\mathbf{ref}_k} \right) \\ & + \frac{1}{2} \left(\mathbf{x}_N - \mathbf{x}_{\mathbf{ref}_N} \right)^T \mathbf{Q}_N \left(\mathbf{x}_N - \mathbf{x}_{\mathbf{ref}_N} \right), \qquad \mathbf{Q}_k \geq 0, \quad \mathbf{R}_k > 0 \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{A}_k \mathbf{x}_k + \mathbf{B}_k \mathbf{u}_k, \quad \mathbf{x}_{(k=0)} = \mathbf{x}_0, \qquad \forall k = \{0, \ldots, N-1\} \\ & \mathbf{x}_k \in \mathcal{X}, \qquad \forall k = \{0, \ldots, N\} \\ & \mathbf{u}_k \in \mathcal{U}, \qquad \forall k = \{0, \ldots, N-1\} \end{aligned} \tag{3-56}$$

where the discrete time system dynamics $\mathbf{x}_{k+1} = \mathbf{A}_k \mathbf{x}_k + \mathbf{B}_k \mathbf{u}_k$, where $\mathbf{A}_k \in \mathbb{R}^{n \times n}$ and $\mathbf{B}_k \in \mathbb{R}^{n \times m}$, are linearized either about an equilibrium point, in equilibrium point stabilization (balancing) problem, or about a nominal trajectory, in a trajectory tracking problem. In the first case $\mathbf{A}_k, \mathbf{B}_k$ will be time invariant ($\mathbf{A}_k = \mathbf{A}, \mathbf{B}_k = \mathbf{B}$) and in the second they will be time varying. In the vast majority of optimal control applications, the cost functions J that are chosen are quadratic, where $\mathbf{Q}_k \in \mathbb{R}^{n \times n}$ is the symmetric stage cost weight matrix corresponding to the states, $\mathbf{R}_k \in \mathbb{R}^{m \times m}$ is the symmetric stage cost weight matrix corresponding to the control inputs and $\mathbf{Q}_N \in \mathbb{R}^{n \times n}$ is the symmetric terminal cost weight matrix. The matrices $\mathbf{Q}_k, \mathbf{Q}_N$ have to be psd and matrix $\mathbf{R}_k$ has to be pd, as in the standard LQR case [120]. Additionally, $\mathbf{x}_{\mathbf{ref}_k} \in \mathbb{R}^n$ and $\mathbf{u}_{\mathbf{ref}_k} \in \mathbb{R}^m$ are the state and control input references that represent either an equilibrium point,

in the time invariant case, or a nominal trajectory, in the time varying case. Finally, this MPC problem is convex only if the constraint sets $\mathcal{X},\mathcal{U}$ are also convex.

In this research project, the convex MPC problems that will be examined include only linear inequality constraints for the states and control inputs. For instance, these constraints can have the form:

$$\begin{aligned} &\underline{\mathbf{x}} \le \mathbf{x}_k \le \bar{\mathbf{x}}, \qquad \forall k = \{0,\ldots,N\} \\ &\underline{\mathbf{u}} \le \mathbf{u}_k \le \bar{\mathbf{u}}, \qquad \forall k = \{0,\ldots,N-1\} \end{aligned} \tag{3-57}$$

where $\underline{\mathbf{u}},\bar{\mathbf{u}}$ are the lower and the upper control input bounds and $\underline{\mathbf{x}},\bar{\mathbf{x}}$ are the lower and the upper state bounds. In that case the problem of equation (3-56) takes the following form:

$$\begin{aligned} \min_{\mathbf{x}_{0:N},\mathbf{u}_{0:N-1}} \quad & J = \frac{1}{2}\sum_{k=0}^{N-1}\left(\mathbf{x}_k - \mathbf{x}_{\mathbf{ref}_k}\right)^T \mathbf{Q}_k \left(\mathbf{x}_k - \mathbf{x}_{\mathbf{ref}_k}\right) + \left(\mathbf{u}_k - \mathbf{u}_{\mathbf{ref}_k}\right)^T \mathbf{R}_k \left(\mathbf{u}_k - \mathbf{u}_{\mathbf{ref}_k}\right) \\ & \quad + \frac{1}{2}\left(\mathbf{x}_N - \mathbf{x}_{\mathbf{ref}_N}\right)^T \mathbf{Q}_N \left(\mathbf{x}_N - \mathbf{x}_{\mathbf{ref}_N}\right), \qquad \mathbf{Q}_k \ge 0, \quad \mathbf{R}_k > 0 \\ \text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{A}_k\mathbf{x}_k + \mathbf{B}_k\mathbf{u}_k, \quad \mathbf{x}_{(k=0)} = \mathbf{x}_0, \qquad \forall k = \{0,\ldots,N-1\} \\ & \underline{\mathbf{x}} \le \mathbf{x}_k \le \bar{\mathbf{x}}, \qquad \forall k = \{0,\ldots,N\} \\ & \underline{\mathbf{u}} \le \mathbf{u}_k \le \bar{\mathbf{u}}, \qquad \forall k = \{0,\ldots,N-1\} \end{aligned} \tag{3-58}$$

If the cost function is quadratic and all the constraints sets are linear, the convex MPC problem of (3-56) or equivalently the NLPs of the direct methods of (3-2) are reduced to QPs. The existing QP solvers include active-set [37], [138], interior point [43], [139], [140] and alternating direction method of multipliers (ADMM)-based methods [38].

3.5 Implementation

The aforementioned optimization techniques have been implemented on a double pendulum using the Gazebo simulator. Thus, the performance of these methods can be reasoned about after the results of those experiments are taken into consideration. First of all, some important information and details about the software, routines and algorithms that were utilized have to be presented. A list of all the function declarations is located in Appendix B.

First and foremost, this entire project consists of four ROS packages. The tree structure of the project is depicted in Figure 3-16. The first one is *plugins* package which is responsible for the interaction of the controllers with the simulation model and environment. The second one is *rr_manipulator_online_control* that consists of four other packages. Each of them implements the controllers that are presented in this section. The third one is *rr_manipulator_online_description* that contains the URDF description of the model that is simulated. The fourth and final package is *rr_manipulator_online_gazebo* that contains a *.launch* file that spawns the model in the simulation environment and a *.world* file that contains the description of the environment in which the model is being simulated. This description includes the ground model, the friction model and global parameters such as physics properties and solver parameters. The entire project can be found at the CSL-Legged Team bitbucket repository[2].

[2] https://bitbucket.org/csl_legged/rr_manipulator_online_optimal_control/src/main/

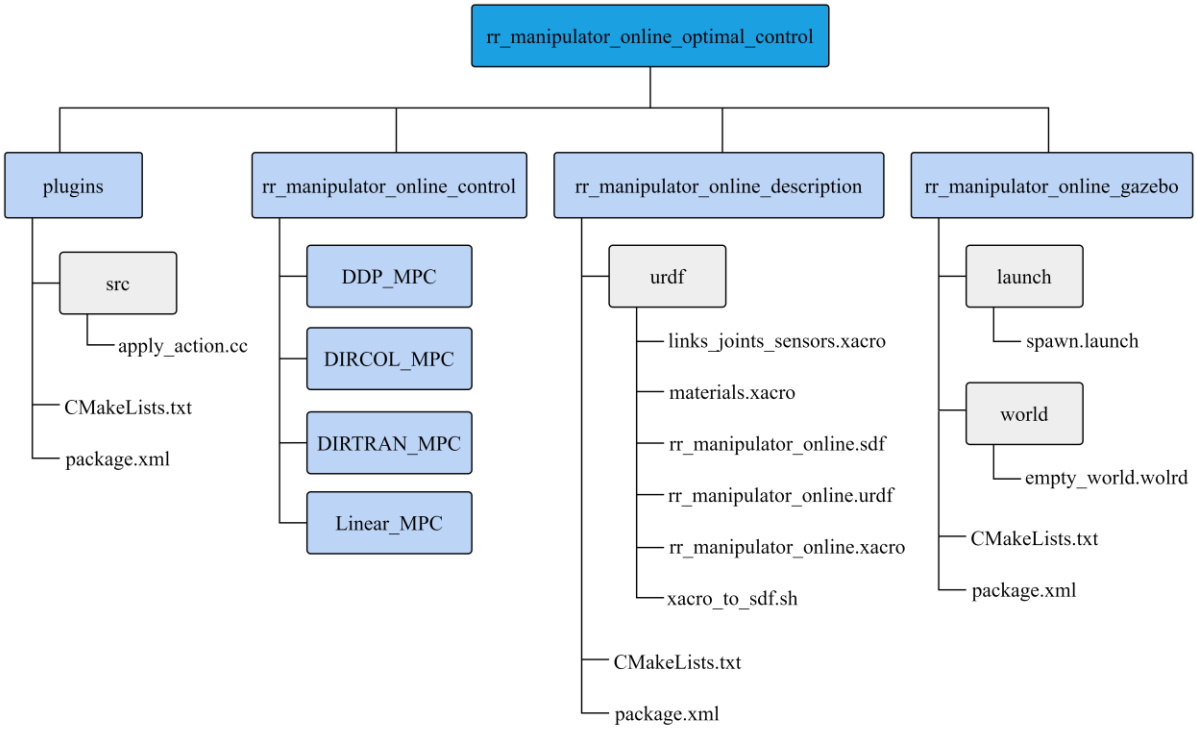


Figure 3-16. Project tree structure of rr_manipulator_online_optimal_control.

Secondly, the C++ linear algebra library Eigen [141] is utilized to represent vectors and matrices and to make use of the algorithms it supports, e.g. for matrix decompositions.

Furthermore, the system dynamics, in continuous time, are necessary for the design of the MPC controller and therefore, they must be provided to the controller. The Rigid Body Dynamics (RBD) of the robot can be expressed in the following compact matrix form, which represents the joint space dynamic model:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}}+\mathbf{c}(\mathbf{q},\dot{\mathbf{q}})+\mathbf{G}(\mathbf{q})=\mathbf{S}^T\boldsymbol{\tau}+\mathbf{J}^T(\mathbf{q})\mathbf{f} \tag{3-59}$$

where $\mathbf{q}$ is the generalized coordinates vector, $\mathbf{M}$ is the Joint Space Inertia matrix, $\mathbf{c}$ are the Coriolis and centripetal terms, $\mathbf{G}$ is the gravity term, $\mathbf{J}$ is the contact (geometric) Jacobian that that maps the external forces/torques to the generalized coordinate space, $\mathbf{f}$ are the external forces/torques, $\mathbf{S}$ is the selection matrix that maps input forces/torques to joints and $\boldsymbol{\tau}$ are the input forces/torques. In case of a fully actuated system with directly driven joints, $\mathbf{S}$ is equal to the identity matrix. Moreover, the RBD of the robot can also be expressed in the state-space model, which is more common in the design of control laws compared to the joint space dynamic model and it has the following form:

$$\dot{\mathbf{x}}=\mathbf{f}(\mathbf{x},\mathbf{u}) \tag{3-60}$$

The relationship between the state variables $\mathbf{x}$ and the generalized coordinates $\mathbf{q}$ as well as the relationship between the control inputs $\mathbf{u}$ and the input forces/torques $\boldsymbol{\tau}$ are the following:

$$\mathbf{x}=\begin{bmatrix}\mathbf{q}\\ \dot{\mathbf{q}}\end{bmatrix},\quad \mathbf{u}=\boldsymbol{\tau} \tag{3-61}$$

Consequently, the state-space dynamics can be computed, given some specific $\mathbf{x},\mathbf{u}$, by calculating $\dot{\mathbf{x}}$ which is equal to:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{\mathbf{q}} \\ \ddot{\mathbf{q}} \end{bmatrix} = \mathbf{f}(\mathbf{x},\mathbf{u}) \tag{3-62}$$

The only unknown variable in the equation (3-62) is the acceleration $\ddot{\mathbf{q}}$ which can be computed using the forward dynamics equation FD:

$$FD(\mathbf{q},\dot{\mathbf{q}},\boldsymbol{\tau}) = \ddot{\mathbf{q}} = \mathbf{M}(\mathbf{q})^{-1}\left[\mathbf{S}^T\boldsymbol{\tau} + \mathbf{J}^T(\mathbf{q})\mathbf{f} - \mathbf{c}(\mathbf{q},\dot{\mathbf{q}}) - \mathbf{G}(\mathbf{q})\right] \tag{3-63}$$

which describes the response of a system to given input forces/torques and external forces/torques, in terms of generalized coordinates. The forward dynamics can be computed using a rigid body dynamics algorithm named Articulated Body Algorithm (ABA) [142], [143]. The ABA and other rigid body algorithms for articulated systems have been implemented in libraries like Pinocchio [63] and RBDL, the Rigid Body Dynamics Library [16]. Both Pinocchio and RBDL are built upon Eigen and are thus compatible with it. In this thesis, both libraries were utilized, for different optimization techniques, for reasons that will become clear in chapter 3.5.3. In Pinocchio the forward dynamics are computed using the function **aba(.)** and in RBDL using the function **ForwardDynamics(.)**. To perform this calculation both libraries need a description format of the robot that is being modeled. Such description is contained in the *.urdf* file, that is also necessary for spawning the robot in the simulation environment. Using this *.urdf* file, they are able to build a Model object of that robot that they make use of for the computation of the forward dynamics as well as for other computations they can perform (i.e., forward and inverse kinematics). Therefore, the Model object contains the physical description of the robot which includes kinematic and inertial parameters defining its structure. Pinocchio apart from the Model object also requires a Data object, necessary for algorithm buffering. In Pinocchio, a Data structure dedicated to the model has to be allocated, similar to how most algorithms need extra storage space for storing internal values. The Data object contains values that are the intermediate or the final results of the computations that are dictated by various algorithms that it provides, like the ABA. Pinocchio relies on a strict separation between constant parameters, in Model, and computation buffer, in Data. The Data objects are built from Model objects. This procedure is conducted in Pinocchio as shown in the following code snippet, using the function **BuildPinocchioModel()**, member of the custom class **RobotDynamics**. This custom function uses the Pinocchio function **buildModel(.)** to create the model given the path of the URDF.

```
// Build pinocchio model of the robot
void RobotDynamics::BuildPinocchioModel()
{
  string urdf_package_filename =
      ros::package::getPath("rr_manipulator_online_description");
  string urdf_filename =
      urdf_package_filename + string("/urdf/rr_manipulator_online.urdf");

  std::cout << "Opening file: " << urdf_filename << std::endl;

  // Load the urdf model
  pinocchio::urdf::buildModel(urdf_filename, model);
  std::cout << "model name: " << model.name << std::endl;
```

```
  // Set model gravity
  model.gravity.linear() << 0.0, 0.0, -9.81;

  // Create data required by the algorithms
  data = Data(model);

  // Model and data needed for CppAD
  ad_model = model.cast<ADScalar>();
  ad_data = ADData(ad_model);
}
```

while in RBDL as shown in the following code snippet, using the function **BuildRBDLModel()**, member of the custom class **SymbolicRobotDynamics**. This custom function uses the RBDL function **URDFReadFromFile(.)** to create the model given the path of the URDF.

```
// Build rbdl model of the robot
void SymbolicRobotDynamics::BuildRBDLModel()
{
  string urdf_package_filename =
      ros::package::getPath("rr_manipulator_online_description");
  string urdf_filename =
      urdf_package_filename + string("/urdf/rr_manipulator_online.urdf");

  std::cout << "Opening file: " << urdf_filename << std::endl;

  model = new RigidBodyDynamics::Model();

  if (!Addons::URDFReadFromFile(urdf_filename.c_str(), model, false,
                                false))
  {
    abort();
  }

  // Set model gravity
  model->gravity = Vector3d(0., 0., -9.81);

  std::cout << "Dofs: " << model->q_size << std::endl;

  //const std::vector<bool> actuated_dofs{ true, true };

  //ConstraintSet C1;
  //C1.Bind(**model);
  //C1.SetActuationMap(**model, actuated_dofs);
}
```

Given this model, the procedure described to compute the continuous time dynamics is implemented using the function **ContinuousDynamics(.)**, member of the custom class **RobotDynamics** located in the file *robot_dynamics.cpp*, for Pinocchio and **ContinousDynamicsRBDL(.)**, member of the custom class **SymbolicRobotDynamics** located in the file *symbolic_robot_dynamics.cpp*, for RBDL. Both functions are illustrated in the following code snippets, the first one corresponds to Pinocchio and the last two correspond to RBDL.

```
// Compute continuous time dynamics
void RobotDynamics::ContinuousDynamics(ADConfigVectorType& ad_x,
                                       ADTangentVectorType& ad_u,
```

```
                                        ADTangentVectorType& ad_xdot)
{
  pinocchio::aba(ad_model, ad_data, ad_x.head(ad_model.nq),
                 ad_x.tail(ad_model.nv), ad_u);
  ad_xdot << ad_x.tail(ad_model.nq), ad_data.ddq;

  // std::cout << "Here is ad_data.ddq:\n" << ad_data.ddq << std::endl;
}
```

```
// Compute continuous time dynamics
casadi::MX SymbolicRobotDynamics::fd(const VectorNd& Q,
                                     const VectorNd& Qdot,
                                     const VectorNd& Tau)
{
  VectorNd Qddot(model->qdot_size);
  ForwardDynamics(*model, Q, Qdot, Tau, Qddot);
  return vertcat(Qdot, Qddot);
}
```

```
void SymbolicRobotDynamics::ContinuousDynamicsRBDL(VectorNd& q,
                                                   VectorNd& q_dot,
                                                   VectorNd& u,
                                                   casadi::MX& x_dot)
{
  rbdl_check_api_version(RBDL_API_VERSION);

  casadi::Function fd_fun =
      casadi::Function("fd_fun", { Q_sym, QDot_sym, Tau_sym },
                       { fd(Q_sym, QDot_sym, Tau_sym) },
                       { "Q", "QDot", "Tau" }, { "xDot" })
          .expand();

  x_dot = fd_fun(casadi::MXDict{
                     { "Q", q }, { "QDot", q_dot }, { "Tau", u } })
              .at("xDot");

  // std::cout << "x_dot :"<< x_dot << std::endl;
}
```

Furthermore, the MPC controller is executed in real-time, and is implemented using a ROS node. The controller uses the joint positions and velocities to calculate the optimal action and afterwards applies it to the model. To do so, it subscribes to the ROS topic */joint_states* to access the joint positions and velocities and publishes the optimal action to the ROS topic */joint_commands* to get it applied to the model.

Also, a Gazebo model plugin, named *apply_action.cc*, is utilized. Model plugins in Gazebo allow the user to access some of the physical properties of the model and its components (i.e., links, joints, collision objects) in the simulation environment as well as enabling the user to interact directly with that model or its components. To be more precise, for the double pendulum control scenario, the position and the velocities of the joint angles are needed. Gazebo offers the functions **Position()** and **GetVelocity(_index)** that provide the joint position and velocity respectively, where **_index** represents index that corresponds to the joint axis, that is equal to zero for 1-DoF joints. These values are then published to the topic */joint_states.* Moreover, this plugin applies the control inputs (actions), which are joint torques calculated by the MPC controller, to the joints of the pendulum. It subscribes to the topic */joint_commands*

to have access to them. These actions then are applied using the Gazebo function **SetForce(_index, _effort)** where **_effort** represents the torque that should be applied.

Finally, it is wise, especially in convex problems, to set the terminal cost weight matrix $\mathbf{Q}_N$ equal an approximation of the true value function of the OCP being solved. One idea is to set it equal to the solution of the infinite horizon LQR Riccati equation applied to the matrices $\mathbf{A},\mathbf{B},\mathbf{Q},\mathbf{R}$. This solution can be obtained by solving the discrete algebraic Riccati equation (DARE) [38]. In that scenario, the terminal cost will be equal to the LQR optimal value function at the end of the horizon V_N. However, this will only be an approximation of the true value function of the OCP being solved (i.e, convex MPC) since that problem will in general include state and/or control input inequality constraints and LQR does not. In general, it is of paramount importance to equip a good approximation of the true cost-to-go at the tail of MPC horizon, for better performance. Such terminal cost provides information regarding what the tail of the problem is, past the end of the horizon [48], [144], [145]. In other words, this terminal cost encodes information about the future that lies past the end of the horizon that does not get optimized. Convex MPC problems, in particular, match exactly LQR when the state and control input inequality constraints are inactive, for any horizon size N, since both problems use linearized dynamics. Inactive inequality constraints means that the states and the control inputs will lay away from their bounds. Therefore, if these constraints are inactive at the end of the horizon, the LQR value function will be a good approximation of the convex MPC exact value function and consequently will be valid for any future time instances at the tail. Therefore, the horizon must be long enough to guarantee that at its end, states and control inputs will lay away from their bounds. In general, if the system is controllable, it will eventually converge to the equilibrium point or the nominal trajectory by applying a proper control law, where the inequality constraints should be inactive. Better approximations of the true cost-to-go, mean that shorter horizons can be used while simultaneously avoiding myopic behavior. If longer horizons get utilized, then less accurate approximations of the exact value function can be used. An alternative of using the DARE solution for the value function approximation, one such approximation can be learned from data [146]. Here, the algorithm that is executed to solve the DARE is described in detail in [147] and is implemented using the function **DARE(.)**. The definition of that function is contained in the file *solve_dare.cpp* and is presented in the two code snippets below. In the first snippet, all the variables necessary are initialized. The second snippet contains the main body of the function.

```
// Algorithm that computes the discrete-time steady state Riccati-Matrix
// (LQR) by solving the discrete algebraic Riccati equation (DARE)
void DARE(MatrixNxNx& A, MatrixNxNu& B, MatrixNxNx& Q, MatrixNuNu& R,
          MatrixNxNx& Pn)
{
  int const N = A.rows();

  // Dynamic programming Solution for P and K
  Eigen ::MatrixXd Pn_old(N, N);

  Eigen ::MatrixXd Pn_plus_1(N, N);
  Eigen ::MatrixXd Gn_plus_1(N, N);
  Eigen ::MatrixXd Gn(N, N);
  Eigen ::MatrixXd An_plus_1(N, N);
  Eigen ::MatrixXd An(N, N);

  An = A;
```

```
  Gn = B * R.inverse() * B.transpose();
  Pn = Q;

  Pn_old = Eigen ::MatrixXd ::Zero(N, N);
  Pn_plus_1 = Eigen ::MatrixXd ::Zero(N, N);

  Eigen ::MatrixXd I_Nx = 1.0 * Eigen ::MatrixXd ::Identity(N, N);

  int counter = 0;
```

```
  while ((Pn.squaredNorm() - Pn_old.squaredNorm()) / Pn.squaredNorm() >
          1e-05)
  {
    Pn_old = Pn;

    An_plus_1 = An * (I_Nx + Gn * Pn).inverse() * An;
    Gn_plus_1 =
        Gn + An * (I_Nx + Gn * Pn).inverse() * Gn * An.transpose();
    Pn_plus_1 =
        Pn + An.transpose() * Pn * (I_Nx + Gn * Pn).inverse() * An;

    An = An_plus_1;
    Gn = Gn_plus_1;
    Pn = Pn_plus_1;

    std::cout << "Not converged" << std::endl;
    counter++;
    std::cout << "Here is counter:\n" << counter << std::endl;
  }

  std::cout << "Converged" << std::endl;
  std::cout << "Here is the matrix Pn:\n" << Pn << std::endl;
}
```

3.5.1 Convex (QP-based) MPC

For the convex MPC, the continuous time dynamics of the form $\dot{\mathbf{x}}=\mathbf{f}_c(\mathbf{x},\mathbf{u})$, have to be discretized and written in the form $\mathbf{x}_{k+1}=\mathbf{f}_d(\mathbf{x}_k,\mathbf{u}_k)$ and then to be linearized about an equilibrium point or a nominal trajectory and written in the form $\mathbf{x}_{k+1}=\mathbf{A}_k\mathbf{x}_k+\mathbf{B}_k\mathbf{u}_k$. The discretization is conducted using an explicit integration scheme, in particular a 4th order explicit Runge-Kutta integrator (RK4) with zero-order hold (zoh) on the control inputs, that has the following form:

$$\begin{aligned}
\mathbf{f}_1&=\mathbf{f}_c(\mathbf{x}_k,\mathbf{u}_k)\\
\mathbf{f}_2&=\mathbf{f}_c\left(\mathbf{x}_k+\frac{1}{2}h\mathbf{f}_1,\mathbf{u}_k\right)\\
\mathbf{f}_3&=\mathbf{f}_c\left(\mathbf{x}_k+\frac{1}{2}h\mathbf{f}_2,\mathbf{u}_k\right)\\
\mathbf{f}_4&=\mathbf{f}_c(\mathbf{x}_k+h\mathbf{f}_3,\mathbf{u}_k)\\
\mathbf{x}_{k+1}&=\mathbf{x}_k+\frac{h}{6}(\mathbf{f}_1+2\mathbf{f}_2+2\mathbf{f}_3+\mathbf{f}_4)=\mathbf{f}_d(\mathbf{x}_k,\mathbf{u}_k)
\end{aligned} \tag{3-64}$$

where h is the constant time step. This discretization scheme is implemented by the function **DiscreteDynamicsRK4(.)**, member of the custom class **RobotDynamics** located in the file

robot_dynamics.cpp. The continuous time dynamics needed are computed by calling the function **ContinuousDynamics(.)**, located in the same file. The implementation of **DiscreteDynamicsRK4(.)** is illustrated in the code snippet below:

```
// Discretize dynamics (RK4 with zoh on control input u)
void RobotDynamics::DiscreteDynamicsRK4(double& planning_timestep,
                                        VectorNx& x_k, VectorNu& u_k,
                                        ADTangentVectorType& ad_f_k,
                                        VectorNx& f_k)
{
  ConfigVectorType q(ad_model.nq);
  TangentVectorType v(ad_model.nv);
  TangentVectorType tau(u_k.size());

  ADConfigVectorType ad_q(ad_model.nq);
  ADTangentVectorType ad_v(ad_model.nv);
  ADTangentVectorType ad_tau(u_k.size());

  q = x_k.head(ad_model.nq);
  v = x_k.tail(ad_model.nv);
  tau = u_k; // u = [1 1] * tau

  ad_q = q.cast<ADScalar>();
  ad_v = v.cast<ADScalar>();
  ad_tau = tau.cast<ADScalar>();

  // Set independent variable X
  ADTangentVectorType X(ad_model.nq + ad_model.nv + ad_tau.size());
  X << ad_q, ad_v, ad_tau;
  CppAD::Independent(X);

  ADTangentVectorType ad_x(ad_model.nq + ad_model.nv);
  ad_x = X.head(ad_model.nq + ad_model.nv);

  ADTangentVectorType ad_u(ad_tau.size());
  ad_u = X.tail(ad_tau.size());

  ADTangentVectorType ad_x1(ad_model.nq + ad_model.nv);
  ADTangentVectorType ad_x2(ad_model.nq + ad_model.nv);
  ADTangentVectorType ad_x3(ad_model.nq + ad_model.nv);
  ADTangentVectorType ad_x4(ad_model.nq + ad_model.nv);

  ADTangentVectorType ad_k1(ad_model.nq + ad_model.nv);
  ADTangentVectorType ad_k2(ad_model.nq + ad_model.nv);
  ADTangentVectorType ad_k3(ad_model.nq + ad_model.nv);
  ADTangentVectorType ad_k4(ad_model.nq + ad_model.nv);

  ad_f_k(ad_model.nq + ad_model.nv);

  ad_x1 = ad_x;
  continuous_dynamics(ad_model, ad_data, ad_x1, ad_u, ad_k1);

  ad_x2 = ad_x + 0.5 * planning_timestep * ad_k1;
  continuous_dynamics(ad_model, ad_data, ad_x2, ad_u, ad_k2);

  ad_x3 = ad_x + 0.5 * planning_timestep * ad_k2;
  continuous_dynamics(ad_model, ad_data, ad_x3, ad_u, ad_k3);

  ad_x4 = ad_x + planning_timestep * ad_k3;
  continuous_dynamics(ad_model, ad_data, ad_x4, ad_u, ad_k4);
```

```
  ad_f_k = ad_x + (planning_timestep / 6.0) *
                      (ad_k1 + 2 * ad_k2 + 2 * ad_k3 + ad_k4);
  // std::cout << "Here is ad_f_k:\n" << ad_f_k << std::endl;

  f_k = ad_f_k.cast<double>();
  // std::cout << "Here is f_k:\n" << f_k << std::endl;
}
```

Then, the matrices $\mathbf{A}_k, \mathbf{B}_k$ of the discretized linear dynamic model can be computed using the expressions:

$$\mathbf{A}_k = \left.\frac{\partial \mathbf{f_d}}{\partial \mathbf{x}}\right|_{\mathbf{x}_{\mathbf{ref}_k}, \mathbf{u}_{\mathbf{ref}_k}}, \quad \mathbf{B}_k = \left.\frac{\partial \mathbf{f_d}}{\partial \mathbf{u}}\right|_{\mathbf{x}_{\mathbf{ref}_k}, \mathbf{u}_{\mathbf{ref}_k}} \tag{3-65}$$

Matrices $\mathbf{A}_k, \mathbf{B}_k$ are necessary for the dynamical system constraints and for the calculation of the terminal cost weight $\mathbf{Q}_N$, by solving the DARE. These derivatives are computed using Automatic/Algorithmic Differentiation (AD) [148] implemented in the open-source AD framework CppAD that is supported by Pinocchio and is compatible with Eigen. This differentiation is implemented by the function **LinearizedDiscreteDynamics(.),** member of the custom class **RobotDynamics** located in the file *robot_dynamics.cpp*. The implementation of **LinearizedDiscreteDynamics(.)** is illustrated in the code snippet below:

```
// Linearize dynamics about an equilibrium point
void RobotDynamics::LinearizedDiscreteDynamics(double& planning_timestep,
                                               VectorNx& x_k,
                                               VectorNu& u_k,
                                               MatrixNxNx& A_k,
                                               MatrixNxNu& B_k)
{
  ConfigVectorType q(ad_model.nq);
  TangentVectorType v(ad_model.nv);
  TangentVectorType tau(u_k.size());

  ADConfigVectorType ad_q(ad_model.nq);
  ADTangentVectorType ad_v(ad_model.nv);
  ADTangentVectorType ad_tau(u_k.size());

  ADTangentVectorType ad_f_k(ad_model.nq + ad_model.nv);
  VectorNx f_k;

  q = x_k.head(ad_model.nq);
  v = x_k.tail(ad_model.nq);
  tau = u_k; // u = [1 1] * tau

  ad_q = q.cast<ADScalar>();
  ad_v = v.cast<ADScalar>();
  ad_tau = tau.cast<ADScalar>();

  // Set independent variable X
  ADTangentVectorType X(ad_model.nq + ad_model.nv + ad_tau.size());
  X << ad_q, ad_v, ad_tau;
  CppAD::Independent(X);

  // std::cout << "Here is X:\n" << X << std::endl;

  DiscreteDynamicsRK4(planning_timestep, x_k, u_k, ad_f_k, f_k);
```

```
  VectorXAD Y(ad_model.nq + ad_model.nv);

  Eigen::Map<ADData::TangentVectorType>(
      Y.data(), ad_model.nq + ad_model.nv, 1) = ad_f_k;

  // std::cout << "Here is Y:\n" << Y << std::endl;

  // Create the function fund: X -> Y
  CppAD::ADFun<Scalar> fund(X, Y);

  CPPAD_TESTVECTOR(Scalar)
  X_val((size_t)(ad_model.nq + ad_model.nv + ad_tau.size()));

  TangentVectorType X_arg(ad_model.nq + ad_model.nv + ad_tau.size());
  X_arg << x_k, u_k;
  Eigen::Map<Data::TangentVectorType>(
      X_val.data(), ad_model.nq + ad_model.nv + ad_tau.size(), 1) =
      X_arg;

  // std::cout << "Here is X_val:\n" << X_val << std::endl;

  // Compute the derivative of Y w.r.t X using CppAD
  CPPAD_TESTVECTOR(Scalar) df_dX = fund.Jacobian(X_val);
  Data::MatrixXs J =
      Eigen::Map<PINOCCHIO_EIGEN_PLAIN_ROW_MAJOR_TYPE(Data::MatrixXs)>(
          df_dX.data(), ad_model.nq + ad_model.nv + ad_tau.size(),
          ad_model.nq + ad_model.nv + ad_tau.size());

  A_k = J.block(0, 0, ad_model.nq + ad_model.nv,
                ad_model.nq + ad_model.nv);
  B_k = J.block(0, ad_model.nq + ad_model.nv, ad_model.nq + ad_model.nv,
                ad_tau.size());

  // std::cout << "Here is J   :\n" << J   << std::endl;
  // std::cout << "Here is A_k :\n" << A_k << std::endl;
  // std::cout << "Here is B_k :\n" << B_k << std::endl;
}
```

The QP-based MPC controller is implemented in the file *Linear_MPC.cpp*. The solver that is utilized here to solve the QP is OSQP (Operator Splitting Quadratic Program), an ADMM-based solver for quadratic programming. It is also combined with osqp-eigen [149], which is a simple Eigen-C++ wrapper for OSQP library that functions as an interface. The solver OSQP solves (convex) QPs that have the form:

$$\begin{aligned} \min_{\mathbf{z}} \quad & \frac{1}{2}\mathbf{z}^T\mathbf{P}\mathbf{z}+\mathbf{q}^T\mathbf{z} \\ \text{s.t.} \quad & \mathbf{l}_\mathbf{b} \le \mathbf{A}_\mathbf{c}\mathbf{z} \le \mathbf{u}_\mathbf{b} \end{aligned} \tag{3-66}$$

where $\mathbf{z} \in \mathbb{R}^{(n+m)N}$ is the optimization variable comprised of the states and control inputs over the entire horizon. The objective function is defined by the psd matrix $\mathbf{P} \in \mathbb{S}_+^{(n+m)N}$ and a vector $\mathbf{q} \in \mathbb{R}^{(n+m)N}$. The linear equality and inequality constraints are defined by the matrix $\mathbf{A}_\mathbf{c} \in \mathbb{R}^{q \times (n+m)N}$ and vectors $\mathbf{l}_\mathbf{b}, \mathbf{u}_\mathbf{b} \in \mathbb{R}^q$ so that $\mathbf{l}_{\mathbf{b}_i} \in \mathbb{R} \cup \{-\infty\}$ and $\mathbf{u}_{\mathbf{b}_i} \in \mathbb{R} \cup \{+\infty\}$, $\forall i = \{1, \ldots, q\}$ where q is the number of all the constraint equations. Linear equality constraints can be encoded by setting $\mathbf{l}_{\mathbf{b}_i} = \mathbf{u}_{\mathbf{b}_i}$ for some of the elements in vectors $\mathbf{l}_\mathbf{b}, \mathbf{u}_\mathbf{b}$. Consequently, the terms of the problem of equation (3-58) have to be rearranged to match the form of (3-66).

The decision variables vector $\mathbf{z}$ will be equal to: $\mathbf{z}=\begin{bmatrix}\mathbf{u}_0 & \mathbf{x}_1 & \mathbf{u}_1 & \dots & \mathbf{x}_N\end{bmatrix}^T$. According to this arrangement of the decision variables, will the rest of the QP matrices and vectors be selected. The Hessian $\mathbf{P}$ and the gradient $\mathbf{q}$ in the cost function are the following:

$$\mathbf{P}=\begin{bmatrix}\mathbf{R}_0 & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0}\\ \mathbf{0} & \mathbf{Q}_1 & \mathbf{0} & \cdots & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{R}_1 & \cdots & \mathbf{0}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{Q}_N\end{bmatrix} \tag{3-67}$$

$$\mathbf{q}=\begin{bmatrix}-\mathbf{R}_0\mathbf{u}_{\mathbf{ref}_0}\\ -\mathbf{Q}_1\mathbf{x}_{\mathbf{ref}_1}\\ -\mathbf{R}_1\mathbf{u}_{\mathbf{ref}_1}\\ \vdots\\ -\mathbf{Q}_N\mathbf{x}_{\mathbf{ref}_N}\end{bmatrix} \tag{3-68}$$

The linear constraints matrix $\mathbf{A}_\mathbf{c}$ and the constraint bound vectors $\mathbf{l}_\mathbf{b},\mathbf{u}_\mathbf{b}$ are the following:

$$\mathbf{A}_\mathbf{c}=\left[\begin{array}{cccccccc}\mathbf{B}_0 & -\mathbf{I} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{A}_1 & \mathbf{B}_1 & -\mathbf{I} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{A}_2 & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & -\mathbf{I} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{A}_{N-1} & \mathbf{B}_{N-1} & -\mathbf{I}\\ \hline \mathbf{I} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{I} & \mathbf{0}\\ \hline \mathbf{0} & \mathbf{I} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{I}\end{array}\right] \tag{3-69}$$

$$\mathbf{l}_\mathbf{b}=\left[\begin{array}{c}-\mathbf{A}_0\mathbf{x}_0\\ \mathbf{0}\\ \vdots\\ \mathbf{0}\\ \hline \underline{\mathbf{u}}\\ \vdots\\ \underline{\mathbf{u}}\\ \hline \underline{\mathbf{x}}\\ \vdots\\ \underline{\mathbf{x}}\end{array}\right],\quad \mathbf{u}_\mathbf{b}=\left[\begin{array}{c}-\mathbf{A}_0\mathbf{x}_0\\ \mathbf{0}\\ \vdots\\ \mathbf{0}\\ \hline \overline{\mathbf{u}}\\ \vdots\\ \overline{\mathbf{u}}\\ \hline \overline{\mathbf{x}}\\ \vdots\\ \overline{\mathbf{x}}\end{array}\right] \tag{3-70}$$

The aforementioned QP matrices and vectors are filled in after the code in the file *qp_terms.cpp* is executed. The functions that should be executed to do so are **ComputeQPHessian(.)**, **ComputeQPGradient(.)**, **ComputeQPConstraintMatrix(.)** and **ComputeQPConstraintVectors(.)**, members of the custom class **QP**, and are presented here:

```
// Build Hessian matrix H for QP
void QP::ComputeQPHessian(int planning_horizon_steps, MatrixNxNx& Q,
                          MatrixNuNu& R, MatrixNxNx& P, SpMat& H)
{
  int const N = Q.rows();
  int const M = R.rows();

  Eigen::MatrixXd H_dense((N + M) * planning_horizon_steps,
                          (N + M) * planning_horizon_steps);
  H_dense.setZero();
  // H.reserve(
  //     Eigen::VectorXi::Constant((M+N) * planning_horizon_steps, N));

  Eigen::VectorXd qp_hessian_vec(N + M);
  qp_hessian_vec.setZero();

  qp_hessian_vec << R.diagonal(), Q.diagonal();

  H_dense =
      qp_hessian_vec.replicate(planning_horizon_steps, 1).asDiagonal();

  H_dense.block((N + M) * (planning_horizon_steps - 1) + M,
                (N + M) * (planning_horizon_steps - 1) + M, N, N) = P;

  H = H_dense.sparseView();
  // H.makeCompressed();

  // std::cout << "Here is the matrix H:\n" << H << std::endl;
}
```

```
// Build gradient vector q for OP
void QP::ComputeQPGradient(int planning_horizon_steps, MatrixNxNx& Q,
                           MatrixNuNu& R, MatrixNxNx& P, VectorNx& x_ref,
                           VectorNu& u_ref, Vector_sol& q)
{
  int const N = Q.rows();
  int const M = R.rows();

  Eigen::VectorXd qp_grad_vec(N + M);
  qp_grad_vec.setZero();

  qp_grad_vec << -R * u_ref, -Q * x_ref;

  q = qp_grad_vec.replicate(planning_horizon_steps, 1);

  q.tail(N) = -P * x_ref;
}
```

```
// Build Linear Constraints Matrix Ac for QP
void QP::ComputeQPConstraintMatrix(int planning_horizon_steps,
                                   MatrixNxNx& A, MatrixNxNu& B,
                                   SpMat& Ac)
```

```
{
  int const N = A.cols();
  int const M = B.cols();

  Eigen::MatrixXd Ac_dense((N + M + N) * planning_horizon_steps,
                           (N + M) * planning_horizon_steps);
  Ac_dense.setZero();

  ////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////

  Eigen::MatrixXd C_dense(N * planning_horizon_steps,
                          (N + M) * planning_horizon_steps);
  C_dense.setZero();

  int i = 0;
  for (int j = 0; j < (M + N) * planning_horizon_steps; j += (N + M))
  {
    C_dense.block(i, j, N, M) = B;
    i += N;
  }

  i = N;
  for (int j = M; j < (M + N) * (planning_horizon_steps - 1);
       j += (N + M))
  {
    C_dense.block(I, j, N, N) = A;
    i += N;
  }

  i = 0;
  for (int j = M; j < (M + N) * planning_horizon_steps; j += (N + M))
  {
    C_dense.block(i, j, N, N) = -Eigen::MatrixXd::Identity(N, N);
    i += N;
  }

  // std::cout <<""Here is the matrix C:\"" << C_dense << std::endl;

  ////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////

  Eigen::MatrixXd U_dense(M * planning_horizon_steps,
                          planning_horizon_steps * (N + M));
  U_dense.setZero();

  i = 0;
  for (int j = 0; j < (M + N) * planning_horizon_steps; j += (N + M))
  {
    U_dense.block(i, j, M, M) = Eigen::MatrixXd::Identity(M, M);
    i += M;
  }

  // std::cout << "Here is the matrix U:\n" << U_dense << std::endl;

  ////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////
```

```
  Eigen::MatrixXd X_dense(N * planning_horizon_steps,
                          planning_horizon_steps * (N + M));
  X_dense.setZero();

  i = 0;
  for (int j = M; j < (M + N) * planning_horizon_steps; j += (N + M))
  {
    X_dense.block(i, j, N, N) = Eigen::MatrixXd::Identity(N, N);
    i += N;
  }

  // std::cout << "Here is the matrix X:\n" << X_dense << std::endl;

  ////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////

  Ac_dense << C_dense, U_dense, X_dense;

  Ac = Ac_dense.sparseView();
  // Ac.makeCompressed();

  // std::cout << "Here is the matrix Ac:\n" << Ac << std::endl;
}
```

```
// Build lower and upper bound vectors lb, ub for QP
void QP::ComputeQPConstraintVectors(int planning_horizon_steps,
                                    VectorNx& xmin, VectorNx& xmax,
                                    VectorNu& umin, VectorNu& umax,
                                    Vector_constr& lb, Vector_constr& ub)
{
  int const N = xmax.size();
  int const M = umax.size();

  Eigen::MatrixXd b(N * planning_horizon_steps, 1);
  b.setZero();

  // Lower bounds for control inputs
  Eigen ::MatrixXd lb_con_u(M * planning_horizon_steps, 1);
  lb_con_u = umin.replicate(planning_horizon_steps, 1);

  // Upper bounds for control inputs
  Eigen ::MatrixXd ub_con_u(M * planning_horizon_steps, 1);
  ub_con_u = umax.replicate(planning_horizon_steps, 1);

  // Lower bounds for states
  Eigen ::MatrixXd lb_con_x(N * planning_horizon_steps, 1);
  lb_con_x = xmin.replicate(planning_horizon_steps, 1);

  // Upper bounds for states
  Eigen ::MatrixXd ub_con_x(N * planning_horizon_steps, 1);
  ub_con_x = xmax.replicate(planning_horizon_steps, 1);

  lb << b, lb_con_u, lb_con_x;
  ub << b, ub_con_u, ub_con_x;
}
```

After each control loop, the values of the bounds $\mathbf{l_b}, \mathbf{u_b}$ should be updated with the new values of the initial conditions $\mathbf{x}_0$. If the problem is time varying, meaning that

$\mathbf{A}=\mathbf{A}_k, \mathbf{B}=\mathbf{B}_k, \mathbf{Q}=\mathbf{Q}_k, \mathbf{R}=\mathbf{R}_k$, then the values of the matrices $\mathbf{P}, \mathbf{A}_c$ and of the vector $\mathbf{q}$, will also have to be updated, after each control loop.

The procedure followed to instantiate and initialize the solver and also to initialize or update the values of the QP matrices and vectors is described in the code snippet below. If the solver is not initialized some initial parameters of the solver must be specified like the tolerances that must be achieved to converge to a solution, or the number of decision variables and constraints of the problem. Moreover, an initial guess of the solution of the problem must be provided if the solver is not initialized. If the solver is already initialized, this initial guess of the decision variables (primal solution) as well as the guess for the Lagrange multipliers (dual solution) should also be updated. Lagrange multipliers are variables that are needed to force the solution to not only minimize the cost function but also to obey the constraints of the problem. A vector of Lagrange multipliers corresponds for each constraint set. The technique of using the previous solution as an initial guess for the decision variables is known as warm starting and significantly accelerates the convergence of the solver. A common technique is to use the primal and dual solution acquired in the previous control loop as an initial guess for the primal and dual variables. This technique is known as warm starting and significantly accelerates the convergence of the solver. This is done automatically in OSQP, as it is written in the snippet.

```
if (!solver.isInitialized())
{
  std::cout << "It IS NOT initialized \n" << std::endl;

  // solver settings
  solver.settings()->setVerbosity(true);
  solver.settings()->setWarmStart(true);
  solver.settings()->setPolish(true);
  solver.settings()->setAbsoluteTolerance(1e-03);
  solver.settings()->setRelativeTolerance(1e-03);
  solver.settings()->setCheckTermination(1);
  // solver.settings()->setScaledTerimination(1);

  // set the initial data of the QP solver
  solver.data()->setNumberOfVariables(Ac.cols());
  solver.data()->setNumberOfConstraints(Ac.rows());

  solver.data()->setHessianMatrix(P);
  solver.data()->setGradient(q);
  solver.data()->setLinearConstraintsMatrix(Ac);
  solver.data()->setLowerBound(lb);
  solver.data()->setUpperBound(ub);

  // instantiate the solver
  solver.initSolver();
}
else
{
  std::cout << "It IS initialized \n" << std::endl;

  // update the QP matrices and vectors
  // solver.updateHessianMatrix(P);
  // solver.updateGradient(q);
  // solver.updateLinearConstraintsMatrix(Ac);
  solver.updateBounds(lb, ub);
```

```
}
```

Then the QP can be solved by calling the function **solveProblem(.)** provided by the solver. After solving it, the user can extract the optimal values of the decision variables $\mathbf{z}^*$ by calling the function **getSolution(.)**. The first m elements of the vector $\mathbf{z}^*$ are equal to the control input $\mathbf{u}_0$. Then, this control input is applied to the system. Afterwards, the system responds to that control action and the current state of the system gets measured $\mathbf{x}_{\text{meas}}$. In general, $\mathbf{x}_{\text{meas}}$ will not be equal to the expected (predicted) state response $\mathbf{x}_1$. So, the new initial state $\mathbf{x}_0$ is set equal to $\mathbf{x}_{\text{meas}}$, the QP matrices and vectors get updated, and the entire iterative procedure gets repeated at every simulation step. This entire procedure is sketched in detail in the flow chart of Figure 3-17. Also, the directory tree of this package is depicted in Figure 3-18.

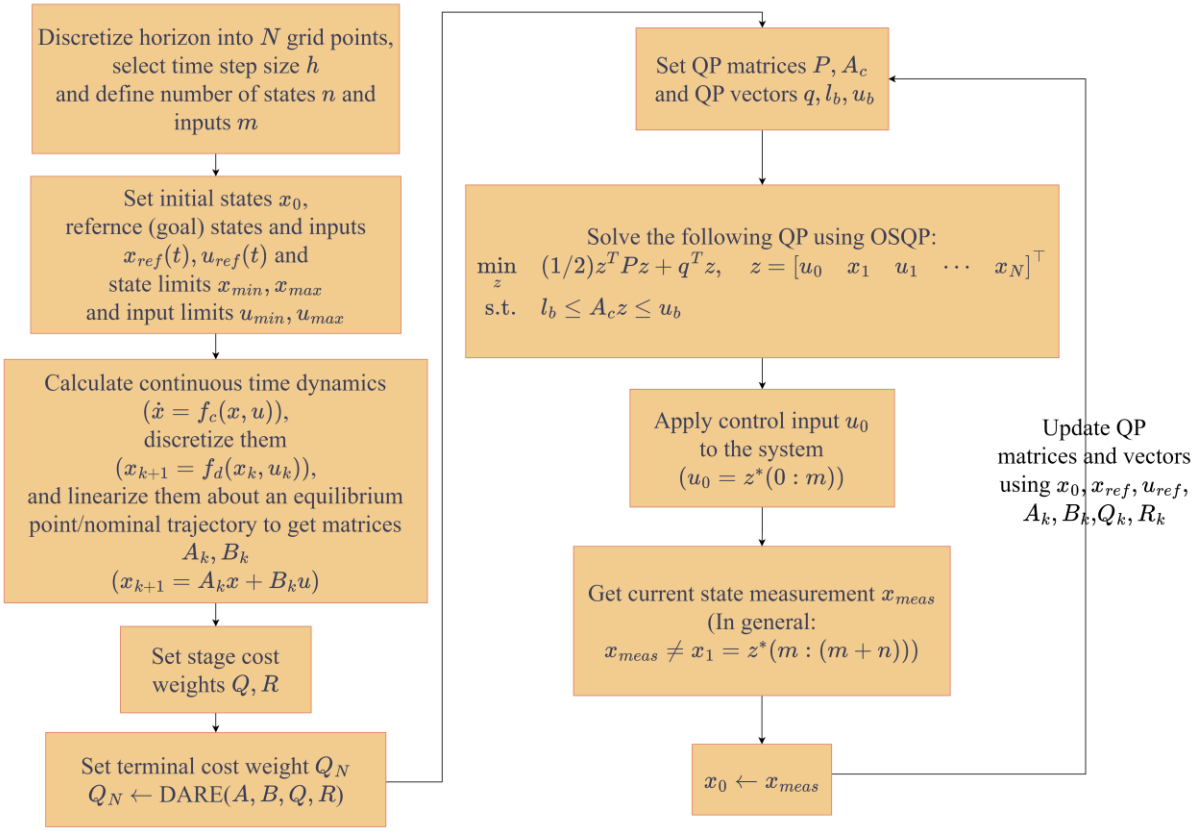


Figure 3-17. QP-based MPC flow chart.

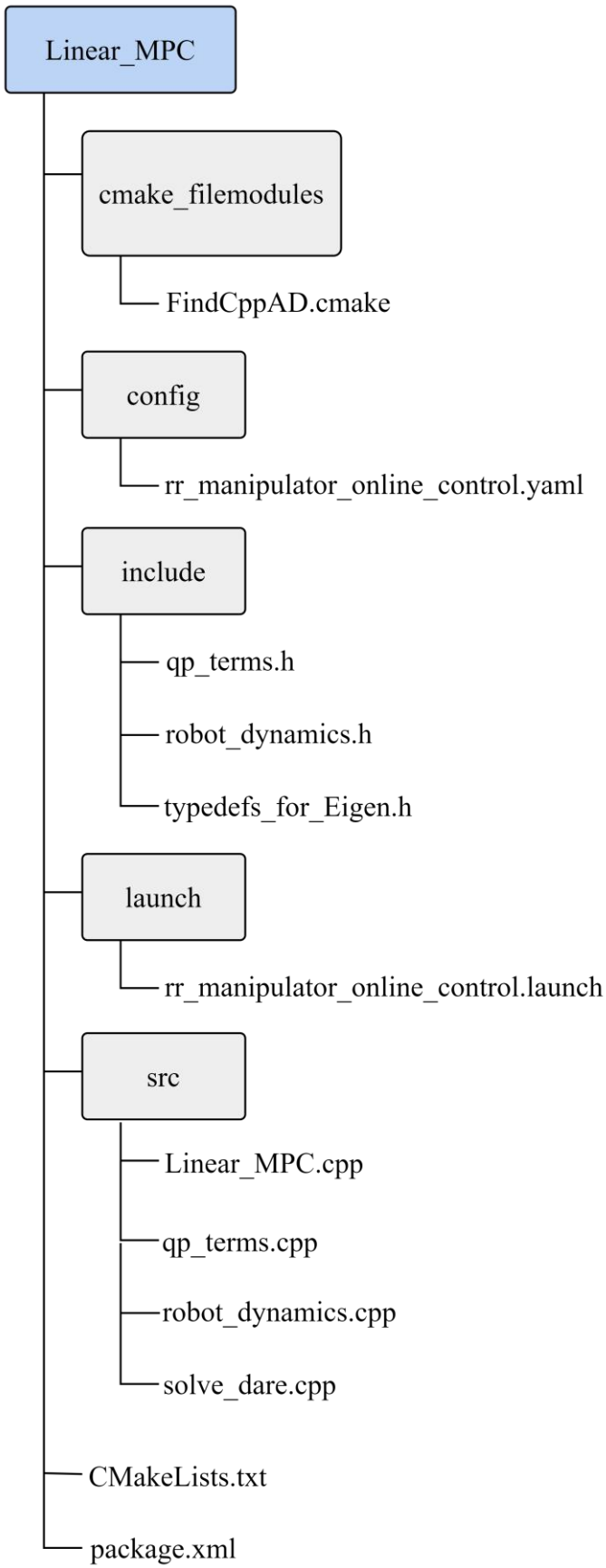


Figure 3-18. Directory tree of Linear_MPC package.

3.5.2 BOX-DDP MPC

For the BOX-DDP MPC, the continuous time dynamics of the form $\dot{\mathbf{x}} = \mathbf{f}_c(\mathbf{x}, \mathbf{u})$, also need to be discretized and written in the form $\mathbf{x}_{k+1} = \mathbf{f}_d(\mathbf{x}_k, \mathbf{u}_k)$, to perform forward rollouts of the dynamics. The discretization is also conducted using an explicit RK4 integrator with zero-order hold (zoh) on the control inputs. The continuous time dynamics are computed by calling the function **ContinuousDynamics(.),** located in the file *robot_dynamics.cpp*. Also, this discretization scheme is implemented by the function **DiscreteDynamicsRK4(.)**, located in the same file. Moreover, the derivatives of the discretized dynamics $\mathbf{f}_{\mathbf{x}_k}, \mathbf{f}_{\mathbf{u}_k}$ are needed for the backward pass and for the calculation of the terminal cost weight $\mathbf{Q}_N$, by solving the DARE. They are defined by the following expressions:

$$\mathbf{A}_k = \mathbf{f}_{\mathbf{x}_k} = \frac{\partial \mathbf{f}_d}{\partial \mathbf{x}}\bigg|_{\mathbf{x}_k, \mathbf{u}_k}, \quad \mathbf{B}_k = \mathbf{f}_{\mathbf{u}_k} = \frac{\partial \mathbf{f}_d}{\partial \mathbf{u}}\bigg|_{\mathbf{x}_k, \mathbf{u}_k} \tag{3-71}$$

These derivatives are also computed using AD and the library CppAD. This differentiation is implemented by the function **LinearizedDiscreteDynamics(.),** located in the file *robot_dynamics.cpp*.

The cost function selected is a quadratic that has the same form as the objective utilized in the Convex MPC case. This objective, in discrete time, is given by the following expression:

$$\begin{aligned} J = \frac{1}{2}\sum_{k=0}^{N-1}\left(\mathbf{x}_k - \mathbf{x}_{\mathbf{ref}_k}\right)^T \mathbf{Q}_k \left(\mathbf{x}_k - \mathbf{x}_{\mathbf{ref}_k}\right) + \left(\mathbf{u}_k - \mathbf{u}_{\mathbf{ref}_k}\right)^T \mathbf{R}_k \left(\mathbf{u}_k - \mathbf{u}_{\mathbf{ref}_k}\right) \\ + \frac{1}{2}\left(\mathbf{x}_N - \mathbf{x}_{\mathbf{ref}_N}\right)^T \mathbf{Q}_N \left(\mathbf{x}_N - \mathbf{x}_{\mathbf{ref}_N}\right), \qquad \mathbf{Q}_k \geq 0, \quad \mathbf{R}_k > 0 \end{aligned} \tag{3-72}$$

This cost is computed by calling the function **ComputeCost(.)**, member of the custom class **BoxDDP** located in the file *box_ddp.cpp.* It is described in the code snippet below.

```
void BoxDDP::ComputeCost(MatrixNxNh& X_traj, MatrixNuNh_1& U_traj,
                         EigenDouble& J)
{
  J.setZero();
  EigenDouble l = EigenDouble::Zero();

  for (int i = 0; i < planning_horizon_steps - 1; i++)
  {
    l += 0.5 * ((X_traj.col(i) - x_goal).transpose() * Q *
                (X_traj.col(i) - x_goal)) +
         0.5 * ((U_traj.col(i)).transpose() * R * (U_traj.col(i)));
  }

  EigenDouble lf =
      0.5 *
      ((X_traj.col(planning_horizon_steps - 1) - x_goal).transpose() *
       Qn  * (X_traj.col(planning_horizon_steps - 1) - x_goal));

  J = l + lf;
}
```

Furthermore, the code that implements the backward pass is presented. It is implemented by the function **BackwardPass(.)**, member of the custom class **BoxDDP** that is located in the file *box_ddp*. It is described using the two code snippets below. In the first snippet, all the variables

necessary are initialized, like the gradient and the Hessian of the value function at the boundary $\mathbf{V}_{\mathbf{x}_N}, \mathbf{V}_{\mathbf{x}\mathbf{x}_N}$, which is the terminal state $\mathbf{x}_N$, using equation (3-24).

```
void BoxDDP::BackwardPass(MatrixNxNh& X_traj, MatrixNuNh_1& U_traj,
                          MatrixNuNh_1& d, MatrixNuNxNh_1& K)
{
  std::cout << "Starting backward pass" << std::endl;

  // The cost function must be quadratic (otherwise it must be
  // quadratized firstly)

  d.setZero();
  K.setZero();
  MatrixNxNh p = MatrixNxNh::Zero();
  MatrixNxNxNh P = MatrixNxNxNh::Zero();
  MatrixNxNxNh P_reg = MatrixNxNxNh::Zero();
  dJ1.setZero();
  dJ2.setZero();

  // Value function's first and second derivatives (p, P respectively)
  // calculated at the end of the horizon
  p.col(planning_horizon_steps - 1) =
      Qn * (X_traj.col(planning_horizon_steps - 1) - x_goal);
  P.block<Nx, Nx>(0, Nx * (planning_horizon_steps - 1)) = Qn;
```

The second snippet contains the main body of the function. Firstly, the Jacobians and the Hessians of the cost function $\mathbf{l}_{\mathbf{x}_k}, \mathbf{l}_{\mathbf{u}_k}, \mathbf{l}_{\mathbf{x}\mathbf{x}_k}, \mathbf{l}_{\mathbf{u}\mathbf{u}_k}, \mathbf{l}_{\mathbf{u}\mathbf{x}_k}$ are calculated. Secondly, the Jacobians and Hessians of the action-value function are computed using equation (3-26) and the GN approximation as well as their regularized versions $\tilde{\mathbf{S}}_{\mathbf{u}\mathbf{u}_k}, \tilde{\mathbf{S}}_{\mathbf{u}\mathbf{x}_k}$ using equation (3-37). In this snippet also, a function named **BoxQPSolver(.),** member of the custom class **BoxQuadProg** whose description is located in the file *box_quad_prog.cpp*, is also called. The purpose of this function is to solve the QP of equation (3-45). To be more specific, it calculates the feed-forward term $\mathbf{k}_k$, the set of the free dimensions f_k, the decomposition of the free dimensions of $\mathbf{S}_{\mathbf{u}\mathbf{u}_k}$, $\mathbf{S}_{\mathbf{u}\mathbf{u},f_k}$ and a flag named *result*. If $result = -1$, then a non-positive definite $\mathbf{S}_{\mathbf{u}\mathbf{u}_k}$ is encountered and regularization begins. During this procedure, the parameter ρ gets increased, using the modification schedule of (3-38), the Hessians $\tilde{\mathbf{S}}_{\mathbf{u}\mathbf{u}_k}, \tilde{\mathbf{S}}_{\mathbf{u}\mathbf{x}_k}$ are computed again and the QP of equation (3-45) gets solved again. This process gets repeated while $result = -1$. Afterwards, the feedback gain term $\mathbf{K}_k$ can be calculated using (3-47). Finally, the quadratic model of the value function can be completed by computing the terms $\mathbf{V}_{\mathbf{x}_k}, \mathbf{V}_{\mathbf{x}\mathbf{x}_k}, \Delta V_k$ using equations (3-31) and (3-35). This process is repeated fo $\mathbf{S}_{\mathbf{x}_k}, \mathbf{S}_{\mathbf{u}_k}, \mathbf{S}_{\mathbf{x}\mathbf{x}_k}, \mathbf{S}_{\mathbf{u}\mathbf{u}_k}, \mathbf{S}_{\mathbf{u}\mathbf{x}_k}$ r every timestep k.

```
  for (int k = planning_horizon_steps - 2; k > -1; k--)
  {
    VectorNx x_k = X_traj.col(k);
    VectorNu u_k = U_traj.col(k);

    // Calculate derivatives
    VectorNx lx = Q * (x_k - x_goal);
    VectorNu lu = R * (u_k);
    MatrixNxNx lxx = Q;
    MatrixNuNu luu = R;
    MatrixNuNx lux = MatrixNuNx::Zero();
```

```
    MatrixNxNx Ad;
    MatrixNxNu Bd;
    robotdynamics->LinearizedDiscreteDynamics(planning_timestep, x_k,
                                              u_k, Ad, Bd);

    // iLQR (Gauss - Newton) version
    // Calculate action value function's S first and second derivatives
    VectorNx Sx = lx + Ad.transpose() * p.col(k + 1);
    VectorNu Su = lu + Bd.transpose() * p.col(k + 1);
    MatrixNxNx Sxx =
        lxx + Ad.transpose() * P.block<Nx, Nx>(0, Nx * (k + 1)) * Ad;
    MatrixNuNu Suu =
        luu + Bd.transpose() * P.block<Nx, Nx>(0, Nx * (k + 1)) * Bd;
    MatrixNuNx Sux =
        lux + Bd.transpose() * P.block<Nx, Nx>(0, Nx * (k + 1)) * Ad;

    P_reg.block<Nx, Nx>(0, Nx * (k + 1)) =
        P.block<Nx, Nx>(0, Nx * (k + 1)) +
        lambda * Eigen ::MatrixXd ::Identity(Nx, Nx);

    MatrixNuNx Sux_reg = lux + Bd.transpose() *
                                   P_reg.block<Nx, Nx>(0, Nx * (k + 1)) *
                                   Ad;
    MatrixNuNu SuuF = luu + Bd.transpose() *
                               P_reg.block<Nx, Nx>(0, Nx * (k + 1)) *
                               Bd;

    // Solve Quadratic Program (QP)
    VectorNu lower = umin - u_k;
    VectorNu upper = umax - u_k;

    VectorNu d_k = d.col(min(k + 1, planning_horizon_steps - 2));

    VectorNu d_i;
    MatrixDD SuuF_free;
    double result;
    Eigen ::LLT<Eigen::MatrixXd> cholesky;
    VectorNuBool free_dir;
    boxQuadProg->BoxQPSolver(SuuF, Su, lower, upper, d_k, d_i, result,
                             cholesky, free_dir);

    // Regalarization of SuuF, Sux_reg
    while (result == -1.0)
    {
      std::cout << "Possibly non positive definitie matrix!"
                << std::endl;
      std::cout << "Regularizing SuuF" << std::endl;

      dlambda = max(lambda_factor, dlambda * lambda_factor);
      lambda = max(lambda_min, lambda * dlambda);

      // std::cout << "lambda :\n" << lambda << std::endl;
      // std::cout << "dlambda :\n" << dlambda << std::endl;

      if (lambda > lambda_max)
      {
        std::cout << "lambda > lambda_max \n" << std::endl;
        break;
      }

      P_reg.block<Nx, Nx>(0, Nx * (k + 1)) =
```

```
          P.block<Nx, Nx>(0, Nx * (k + 1)) +
          lambda * Eigen ::MatrixXd ::Identity(Nx, Nx);

      Sux_reg = lux + Bd.transpose() *
                          P_reg.block<Nx, Nx>(0, Nx * (k + 1)) * Ad;
      SuuF = luu + Bd.transpose() *
                       P_reg.block<Nx, Nx>(0, Nx * (k + 1)) * Bd;

      // Solve Quadratic Program (QP)
      lower = umin - u_k;
      upper = umax - u_k;

      d_k = d.col(min(k + 1, planning_horizon_steps - 2));
      boxQuadProg->BoxQPSolver(SuuF, Su, lower, upper, d_k, d_i, result,
                               cholesky, free_dir);
    }

    MatrixDD Sux_reg_free;
    boxQuadProg->logical_slicing_matrix_asym(Sux_reg, free_dir,
                                             ones_bool, Sux_reg_free);

    Eigen ::MatrixXd Lfree(Sux_reg_free.rows(), Sux_reg_free.cols());

    MatrixNuNx K_i = MatrixNuNx::Zero();

    if (free_dir.any())
    {
      Lfree = -cholesky.solve(
          Sux_reg_free); // -SuuF_free.solve(Sux_reg_free);
      boxQuadProg->LogicalFillingMatrixAsym(Lfree, free_dir,
                                            ones_bool, K_i);
    }

    // Save controls/gains
    d.col(k) = d_i;
    K.block<Nu, Nx>(0, Nx * (k)) = K_i;

    // Update cost-to-go approximation
    p.col(k) =
        Sx +
        (K.block<Nu, Nx>(0, Nx * (k))).transpose() * Suu * d.col(k) +
        (K.block<Nu, Nx>(0, Nx * (k))).transpose() * Su +
        Sux.transpose() * d.col(k);
    P.block<Nx, Nx>(0, Nx * (k)) =
        Sxx +
        (K.block<Nu, Nx>(0, Nx * (k))).transpose() * Suu *
            K.block<Nu, Nx>(0, Nx * (k)) +
        (K.block<Nu, Nx>(0, Nx * (k))).transpose() * Sux +
        Sux.transpose() * K.block<Nu, Nx>(0, Nx * (k));

    dJ1 += d.col(k).transpose() * Su;
    dJ2 += 0.5 * d.col(k).transpose() * Suu * d.col(k);
  }
}
```

Moreover, the code that implements the forward pass is presented. It is implemented by the function **ForwardPass(.)** member of the custom class **BoxDDP** located in the file *box_ddp.cpp.* It is described using the two code snippets below. In the first snippet all the variables necessary are initialized, like the maximum number of line-search iterations. Also,

the first forward rollout is executed using equation (3-33) with $a=1.0$. This rollout evaluates the new trajectory $\hat{\mathbf{X}},\hat{\mathbf{U}}$ by applying the optimal control modification, given by the backward pass, to the old trajectory $\mathbf{X},\mathbf{U}$. Afterwards, the value of the new cost function gets computed using the function **cost(.)** as well as the value of the expected cost reduction $\Delta V(a)$ using equation (3-35). Then, if $\Delta V(a)>0$, the ratio z gets computed using equation (3-34).

```
void BoxDDP::ForwardPass(MatrixNxNh& X_old, MatrixNuNh_1& U_old,
                         EigenDouble& J_old, MatrixNuNh_1& d,
                         MatrixNuNxNh_1& K, MatrixNxNh& X_new,
                         MatrixNuNh_1& U_new, EigenDouble& J_new)
{
  std::cout << "Starting forward pass" << std::endl;

  // Line search iterations
  int iter2 = 0;

  // Forward rollout (a = 1)
  X_new.col(0) = X_old.col(0);

  // Backtracking line search parameter
  double a = 1.0;

  ADTangentVectorType f;
  VectorNx f_k;

  for (int k = 0; k < planning_horizon_steps - 1; k++)
  {
    U_new.col(k) =
        U_old.col(k) + a * d.col(k) +
        K.block<Nu, Nx>(0, Nx * (k)) * (X_new.col(k) - X_old.col(k));

    VectorNx x_k = X_new.col(k);
    VectorNu u_k = U_new.col(k);
    robotdynamics->DiscreteDynamicsRK4(planning_timestep, x_k, u_k, f,
                                       f_k);
    X_new.col(k + 1) = f_k;
  }

  // Cost Function
  ComputeCost(X_new, U_new, J_new);

  // Expected cost decrease
  EigenDouble dJ = -a * (dJ1 + a * dJ2);

  // Convergence criterion
  if (dJ(0, 0) > 0)
  {
    z = ((J_old(0, 0) - J_new(0, 0)) / (dJ(0, 0)));
  }
  else
  {
    z = ((J_old(0, 0) - J_new(0, 0)) > 0) -
        ((J_old(0, 0) - J_new(0, 0)) < 0);
    std::cout << "Warning: non-positive expected reduction: should not "
                 "occur"
              << std::endl;
    return;
  }
```

The second snippet contains the main body of the function, which is the line-search procedure. While the value of the ratio z lies within the interval $[\beta_1, \beta_2]$ or the number of the line-search iterations does not exceed the maximum, the line-search procedure gets repeated. During this procedure the parameter a gets iteratively reduced, using the expression $a = \gamma\alpha$, where $\gamma = 0.5$ is a backtracking scaling parameter. Given this new value for a, a forward rollout is executed again, the cost function gets computed again as well as the value of the expected cost reduction $\Delta V(a)$ and the ratio z, if $\Delta V(a) > 0$.

```
  // Line Search
  while ((z < z_min || z > z_max) && (iter2 < iter2_max))
  {
    a *= 0.5;

    // Forward rollout
    for (int k = 0; k < planning_horizon_steps - 1; k++)
    {
      U_new.col(k) =
          U_old.col(k) + a * d.col(k) +
          K.block<Nu, Nx>(0, Nx * (k)) * (X_new.col(k) - X_old.col(k));

      VectorNx x_k = X_new.col(k);
      VectorNu u_k = U_new.col(k);
      robotdynamics->DiscreteDynamicsRK4(planning_timestep, x_k, u_k, f,
                                         f_k);
      X_new.col(k + 1) = f_k;
    }

    // Cost Function
    ComputeCost(X_new, U_new, J_new);

    // Expected cost decrease
    dJ = -a * (dJ1 + a * dJ2);

    // Convergence criterion
    if (dJ(0, 0) > 0)
    {
      z = ((J_old(0, 0) - J_new(0, 0)) / (dJ(0, 0)));
    }
    else
    {
      z = ((J_old(0, 0) - J_new(0, 0)) > 0) -
          ((J_old(0, 0) - J_new(0, 0)) < 0);
      std::cout << "Warning: non-positive expected reduction: should "
                   "not occur"
                << std::endl;
    }

    iter2++;
  }
}
```

The main body of the control loop is presented next and is implemented in the file *DDP_MPC.cpp*. The following three termination conditions, that are also described in [121], are checked at every iteration of the loop (internal iteration).

- The first condition checks if the cost decrease between two consecutive iterations $J(\mathbf{X}, \mathbf{U}) - J\left(\hat{\mathbf{X}}, \hat{\mathbf{U}}\right)$ is less than the tolerance *tolFun*.

- The second condition checks if the feedforward gains $\mathbf{k}_k$ converge to zero. To do so, the norm $\mathbf{k}_{\text{norm}}$ must be computed which is equal to the average maximum of the normalized gains:

$$\mathbf{k}_{\text{norm}} = \frac{1}{N-1}\sum_{k=0}^{N-1}\frac{\|\mathbf{k}_k\|_\infty}{|\mathbf{U}_k|+1} \tag{3-73}$$

  Convergence occurs when this norm gets lower than the pre-specified tolerance *tolGrad* .

- The third condition checks if the number of solver iterations becomes equal to the maximum number of iterations.

In the code snippet below, while the number of solver iterations is lower than the tolerance, the loop gets executed. Then, the backward pass gets initiated by calling the function **BackwardPass(.)**. Afterwards, $\mathbf{k}_k$ gets computed, using the feedforward gains $\mathbf{k}_k$ given by the backward pass, and the code check if the second termination condition, that is pertinent to the feedforward gains, is met. If it is met, the value of the parameter ρ gets decreased, using the modification schedule of (3-38), and the loop breaks. Afterwards, the forward pass gets initiated by calling the function **ForwardPass(.)**, and the cost reduction $J(\mathbf{X},\mathbf{U})-J(\hat{\mathbf{X}},\hat{\mathbf{U}})$ gets computed. If the line-search procedure succeeds, meaning that $z\in[\beta_1,\beta_2]$, then the value of ρ gets decreased, using the modification schedule of (3-38), and the old trajectories and cost $\mathbf{X},\mathbf{U},J(\mathbf{X},\mathbf{U})$ are set equal to the new ones $\hat{\mathbf{X}},\hat{\mathbf{U}},J(\hat{\mathbf{X}},\hat{\mathbf{U}})$. Also, if the first termination condition, that is pertinent to the cost reduction, is met, then the loop breaks. However, if $z\notin[\beta_1,\beta_2]$, then the value of ρ gets increased, using the modification schedule of (3-38). Also, if ρ hits a maximum threshold, then the loop breaks. The aforementioned procedure is implemented by the function **BoxDDPSolver(.)**, member of the class **BoxDDP**, that is located in the file *box_ddp.cpp.*

```
// BOX-DDP Algorithm
void BoxDDP::BoxDDPSolver(MatrixNxNh& X_old, MatrixNuNh_1& U_old,
                          EigenDouble& J_old, MatrixNxNh& X_new,
                          MatrixNuNh_1& U_new, EigenDouble& J_new)
{
  std::cout << "Start of DDP iterative process" << std::endl;

  // DDP algorithm iterations
  int iter1 = 0;

  z = 0.0;
  lambda = lambda_ini;
  dlambda = dlambda_ini;

  MatrixNuNh_1 k;
  MatrixNuNxNh_1 K;

  // BOX-DDP Algorithm
  while ((iter1 < iter1_max))
  {
    // Backward pass
    BackwardPass(X_old, U_old, k, K);

    // Check for termination due to small gradient
    Matrix1Nh_1 g =
        ((k.cwiseAbs())
```

```
            .cwiseProduct(
                (U_old.cwiseAbs() + Eigen ::MatrixXd ::Ones(
                                        Nu, planning_horizon_steps - 1))
                    .cwiseInverse()))
           .colwise()
           .maxCoeff();
    double g_norm = g.mean();

    if (g_norm < tolGrad && lambda < 1e-5)
    {
      dlambda = min(dlambda / lambda_factor, 1 / lambda_factor);
      lambda = lambda * dlambda * (lambda > lambda_min);
      std::cout << "Decrease Lambda" << std::endl;
      std::cout << "lambda :\n" << lambda << std::endl;
      std::cout << "dlambda :\n" << dlambda << std::endl;

      std::cout << "SUCCESS: gradient norm < tolGrad \n" << std::endl;
      break;
    }

    // Forward pass
    ForwardPass(X_old, U_old, J_old, k, K, X_new, U_new, J_new);
    double dcost = J_old(0, 0) - J_new(0, 0);

    // Improved Regularization
    if (z > z_min && z < z_max)
    {
      // decrease lambda
      dlambda = min(dlambda / lambda_factor, 1 / lambda_factor);
      lambda = lambda * dlambda * (lambda > lambda_min);
      std::cout << "Decrease Lambda" << std::endl;
      std::cout << "lambda :\n" << lambda << std::endl;
      std::cout << "dlambda :\n" << dlambda << std::endl;

      // accept changes
      J_old = J_new;
      X_old = X_new;
      U_old = U_new;

      // terminate ?
      if (dcost < tolFun)
      {
        std::cout << "SUCCESS: cost change < tolFun \n" << std::endl;
        break;
      }
    }
    else
    {
      // no cost improvement
      // increase lambda
      dlambda = max(lambda_factor, dlambda * lambda_factor);
      lambda = max(lambda_min, lambda * dlambda);
      std::cout << "Increase Lambda" << std::endl;
      std::cout << "lambda :\n" << lambda << std::endl;
      std::cout << "dlambda :\n" << dlambda << std::endl;

      // terminate ?
      if (lambda > lambda_max)
      {
        std::cout << "EXIT: lambda > lambda_max\n" << std::endl;
        break;
```

```
        }
      }

      std::cout << "Not converged" << std::endl;
      std::cout << "Cost Difference :\n" << dcost << std::endl;
      std::cout << "Feedforward gain :\n" << g_norm << std::endl;

      iter1++;
    }

    std::cout << "Done DDP completely" << std::endl;
    std::cout << "Here is final number of iterations:\n"
              << iter1 << std::endl;
}
```

When the BOX-DDP algorithm has converged, the updated trajectory $\hat{\mathbf{X}},\hat{\mathbf{U}}$ has been calculated. The first m elements of the vector $\hat{\mathbf{U}}$ are equal to the control input $\hat{\mathbf{u}}_0$. Then, this control input is applied to the system. Afterwards, the system responds to that control action and the current state of the system gets measured $\mathbf{x}_{\text{meas}}$. In general, $\mathbf{x}_{\text{meas}}$ will not be equal to the expected (predicted) state response $\hat{\mathbf{x}}_1$. So, the new initial state $\mathbf{x}_0$ is set equal to $\mathbf{x}_{\text{meas}}$. Also, a new initial control input trajectory $\mathbf{U}$ must be selected. The updated trajectory $\hat{\mathbf{U}}$ is utilized for this initialization. In this way, the solver gets warm started. The entire iterative procedure gets repeated at every simulation step. This entire procedure is sketched in detail in the flow chart of Figure 3-19. Also, the directory tree of this package is depicted in Figure 3-20.

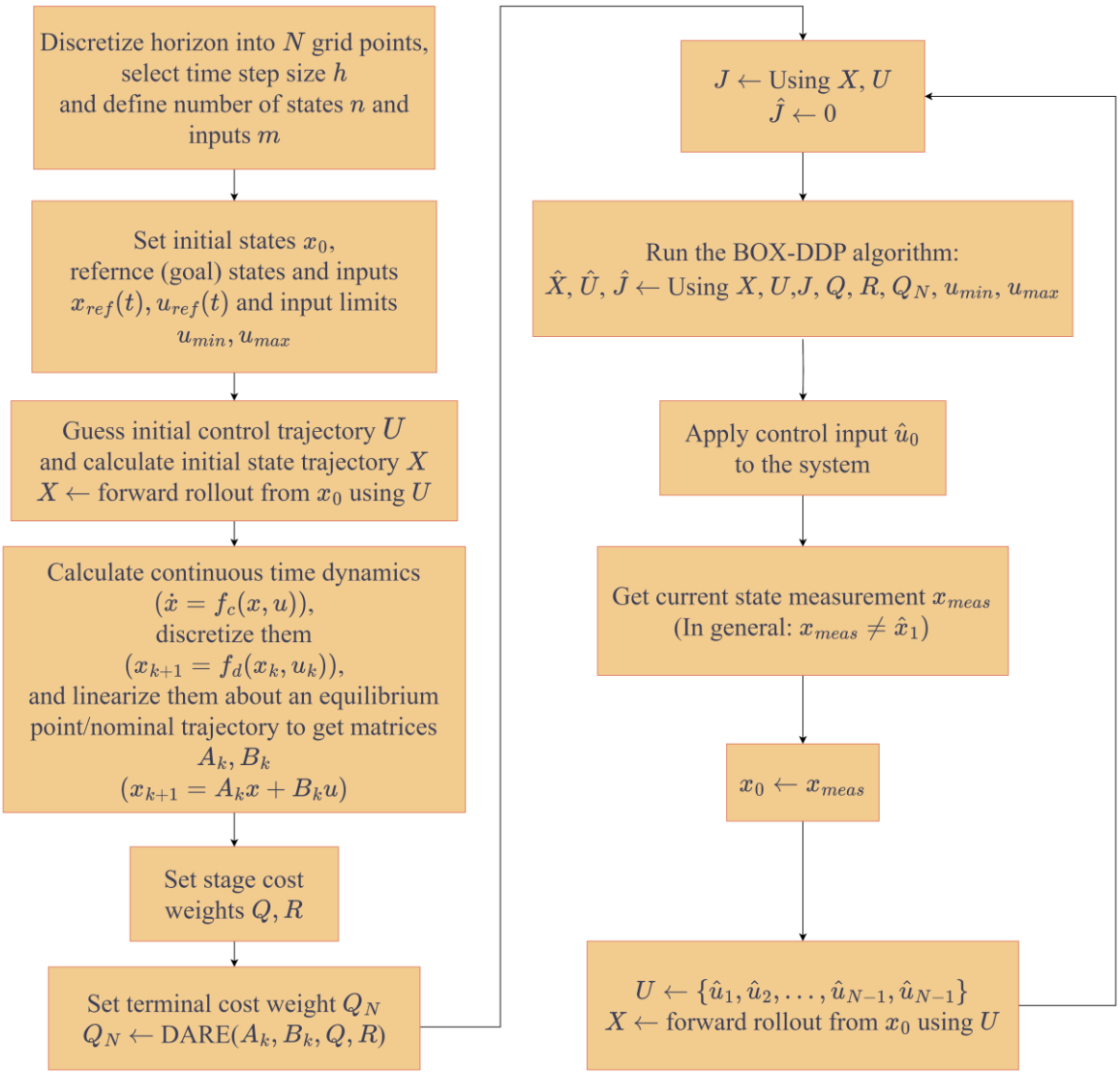


Figure 3-19. BOX-DDP MPC flow chart.

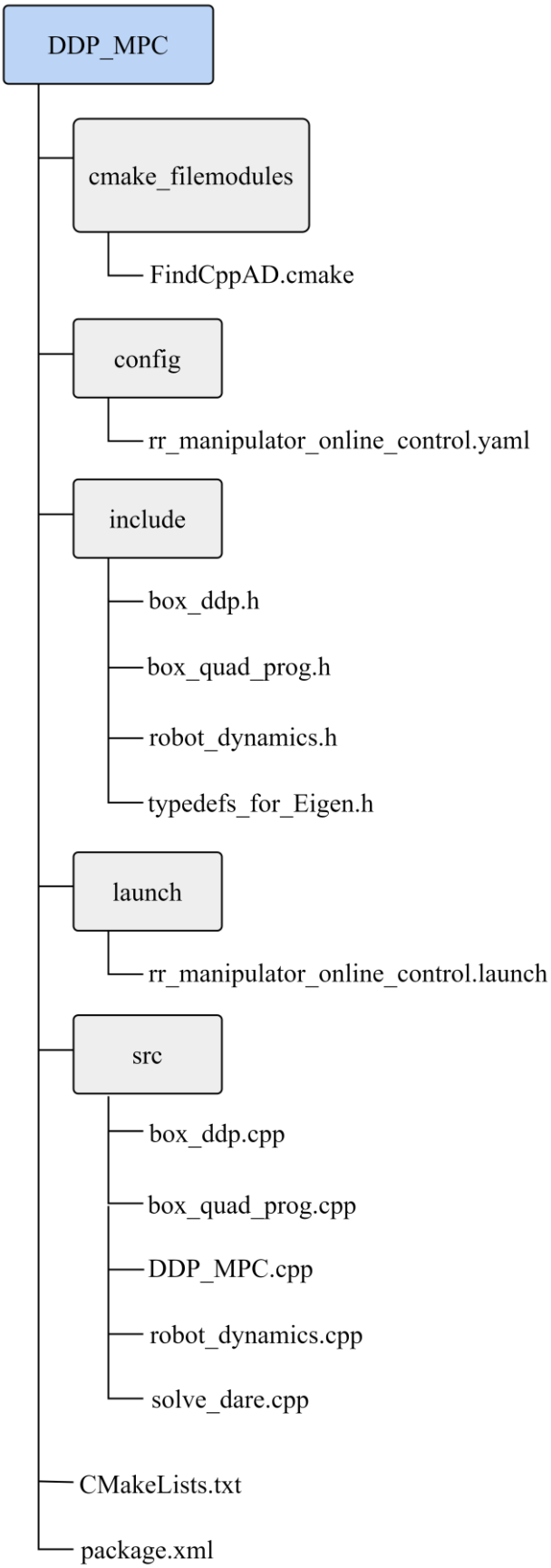


Figure 3-20. Directory tree of DDP_MPC package.

3.5.3 DIRTRAN/DIRCOL MPC

For the DIRTRAN/DIRCOL MPC, the continuous time dynamics of the form $\dot{\mathbf{x}} = \mathbf{f}_c(\mathbf{x}, \mathbf{u})$, are also needed, to formulate the dynamical system constraint, defined by the defects $\boldsymbol{\Delta}_k$ of equations (3-6), (3-16) for DIRTRAN/DIRCOL respectively. The continuous time dynamics needed are computed by calling the function **ContinuousDynamicsRBDL(.)**.

The defect constraint $\boldsymbol{\Delta}_k$ of DIRTRAN is implemented by the function **ComputeNLPDefectConstraintTrapezoidal(.)**, member of the custom class **NLP** located in the file *nlp_terms.cpp*. This function is presented in the two code snippets below. In the first snippet, all the variables necessary are initialized. This snippet also calculates the defect $\boldsymbol{\Delta}_0$. The second snippet contains the main body of the function, where the defects $\boldsymbol{\Delta}_k$ are calculated $\forall k = \{1, \ldots, N-1\}$.

```
// Trapezoidal integration with zero-order hold on u
void NLP::ComputeNLPDefectConstraintTrapezoidal(
    int planning_horizon_steps, double& planning_timestep, casadi::MX& X,
    casadi::MX& U, casadi::MX& constraint)
{
  int const N = X.size1() / planning_horizon_steps;
  int const M = U.size1() / planning_horizon_steps;

  MX x1 = MX::sym("x1", N);
  MX u1 = MX::sym("u1", M);

  MX x2 = MX::sym("x2", N);
  MX u2 = MX::sym("u2", M);

  MX f1 = MX::sym("f1", N);
  MX f2 = MX::sym("f2", N);

  VectorNd q;
  VectorNd q_dot;
  VectorNd tau;

  MX constraint_k = MX::sym("constraint_k", N);

  // Formulate dynamics constraints

  int i = 0;
  int j = 0;

  x1 = X(Slice(i, i + N, 1));
  u1 = U(Slice(j, j + M, 1));
  q = VectorNd(x1(Slice(0, int(0.5 * double(N)), 1), 0));
  q_dot = VectorNd(x1(Slice(int(0.5 * double(N)), N, 1), 0));
  tau = VectorNd(u1(Slice(0, M, 1), 0));
  symbolicrobotdynamics->ContinuousDynamicsRBDL(q, q_dot, tau, f1);
  std::cout << "f1 :\n " << f1 << std::endl;

  x2 = X(Slice(i + N, i + 2 * N, 1));
  u2 = U(Slice(j + M, j + 2 * M, 1));
  q = VectorNd(x2(Slice(0, int(0.5 * double(N)), 1), 0));
  q_dot = VectorNd(x2(Slice(int(0.5 * double(N)), N, 1), 0));
  tau = VectorNd(u2(Slice(0, M, 1), 0));
  symbolicrobotdynamics->ContinuousDynamicsRBDL(q, q_dot, tau, f2);
  std::cout << "f2 :\n " << f2 << std::endl;

  constraint_k = x2 - (x1 + DM(0.5 * planning_timestep) * (f1 + f2));
```

```
  constraint = constraint_k;
```

```
  x1 = x2;
  f1 = f2;

  i += N, j += M;

  for (int k = 1; k < planning_horizon_steps - 1; k++)
  {
    x2 = X(Slice(i + N, i + 2 * N, 1));
    u2 = U(Slice(j + M, j + 2 * M, 1));
    q = VectorNd(x2(Slice(0, int(0.5 * double(N)), 1), 0));
    q_dot = VectorNd(x2(Slice(int(0.5 * double(N)), N, 1), 0));
    tau = VectorNd(u2(Slice(0, M, 1), 0));
    symbolicrobotdynamics->ContinuousDynamicsRBDL(q, q_dot, tau, f2);
    std::cout << "f2 :\n " << f2 << std::endl;

    constraint_k = x2 - (x1 + DM(0.5 * planning_timestep) * (f1 + f2));
    constraint = vertcat(constraint, constraint_k);

    x1 = x2;
    f1 = f2;

    i += N, j += M;
  }

  std::cout << "constraint :\n " << constraint << std::endl;
}
```

The defect constraint Δ_k of DIRCOL is implemented by the function **ComputeNLPDefectConstraintHermiteSimpson(.)**, member of the custom class **NLP** located in the file *nlp_terms.cpp*. This function is presented in the two code snippets below. In the first snippet, all the variables necessary are initialized. This snippet also calculates the defect Δ_0. The second snippet contains the main body of the function, where the defects Δ_k are calculated $\forall k = \{1,\ldots,N-1\}$.

```
// Hermite-Simpson integration with first-order hold on u
void NLP::ComputeNLPDefectConstraintHermiteSimpson(
    int planning_horizon_steps, double& planning_timestep, casadi::MX& X,
    casadi::MX& U, casadi::MX& constraint)
{
  int const N = X.size1() / planning_horizon_steps;
  int const M = U.size1() / planning_horizon_steps;

  MX x1 = MX::sym("x1", N);
  MX u1 = MX::sym("u1", M);

  MX x2 = MX::sym("x2", N);
  MX u2 = MX::sym("u2", M);

  MX f1 = MX::sym("f1", N);
  MX f2 = MX::sym("f2", N);

  MX xm = MX::sym("xm", N);
  MX um = MX::sym("um", M);

  MX fm = MX::sym("fm", N);
  MX xm_dot = MX::sym("xm_dot", N)
```

```
VectorNd q;
VectorNd q_dot;
VectorNd tau;

MX constraint_k = MX::sym("constraint_k", N);

// Formulate dynamics constraints

int i = 0;
int j = 0;

x1 = X(Slice(i, i + N, 1));
u1 = U(Slice(j, j + M, 1));
q = VectorNd(x1(Slice(0, int(0.5 * double(N)), 1), 0));
q_dot = VectorNd(x1(Slice(int(0.5 * double(N)), N, 1), 0));
tau = VectorNd(u1);
symbolicrobotdynamics->ContinuousDynamicsRBDL(q, q_dot, tau, f1);
std::cout << "f1 :\n " << f1 << std::endl;

x2 = X(Slice(i + N, i + 2 * N, 1));
u2 = U(Slice(j + M, j + 2 * M, 1));
q = VectorNd(x2(Slice(0, int(0.5 * double(N)), 1), 0));
q_dot = VectorNd(x2(Slice(int(0.5 * double(N)), N, 1), 0));
tau = VectorNd(u2(Slice(0, M, 1), 0));
symbolicrobotdynamics->ContinuousDynamicsRBDL(q, q_dot, tau, f2);
std::cout << "f2 :\n " << f2 << std::endl;

// xm_dot = DM( -3.0/(2.0*h) )*(x1 - x2) - DM( 0.25 )*(f1 + f2) ;

xm = DM(0.5) * (x1 + x2) + DM(h / 8.0) * (f1 - f2);
um = DM(0.5) * (u1 + u2);
q = VectorNd(xm(Slice(0, int(0.5 * double(N)), 1), 0));
q_dot = VectorNd(xm(Slice(int(0.5 * double(N)), N, 1), 0));
tau = VectorNd(um);
symbolicrobotdynamics->ContinuousDynamicsRBDL(q, q_dot, tau, fm);
std::cout << "fm :\n " << fm << std::endl;

constraint_k =
    x2 - x1 - DM(planning_timestep / 6.0) *
                  (f1 + DM(4.0) * fm + f2); // xm_dot - fm;
constraint = constraint_k;
```

```
x1 = x2;
f1 = f2;

i += N, j += M;

for (int k = 1; k < planning_horizon_steps - 1; k++)
{
  x2 = X(Slice(i + N, i + 2 * N, 1));
  u2 = U(Slice(j + M, j + 2 * M, 1));
  q = VectorNd(x2(Slice(0, int(0.5 * double(N)), 1), 0));
  q_dot = VectorNd(x2(Slice(int(0.5 * double(N)), N, 1), 0));
  tau = VectorNd(u2(Slice(0, M, 1), 0));
  symbolicrobotdynamics->ContinuousDynamicsRBDL(q, q_dot, tau, f2);
  std::cout << "f2 :\n " << f2 << std::endl;

  // xm_dot = DM( -3.0/(2.0*h) )*(x1 - x2) - DM( 0.25 )*(f1 + f2) ;

  xm = DM(0.5) * (x1 + x2) + DM(planning_timestep / 8.0) * (f1 - f2);
```

```
    um = DM(0.5) * (u1 + u2);
    q = VectorNd(xm(Slice(0, int(0.5 * double(N)), 1), 0));
    q_dot = VectorNd(xm(Slice(int(0.5 * double(N)), N, 1), 0));
    tau = VectorNd(um);
    symbolicrobotdynamics->ContinuousDynamicsRBDL(q, q_dot, tau, fm);
    std::cout << "fm :\n " << fm << std::endl;

    constraint_k =
        x2 - x1 -
        DM(planning_timestep / 6.0) *
            (f1 + DM(4.0) * fm + f2); // xm_dot - fm;
    constraint = constraint_k;

    x1 = x2;
    f1 = f2;

    i += N, j += M;
  }

  std::cout << "constraint :\n " << constraint << std::endl;
}
```

Moreover, the cost function, in continuous time, should be discretized using an integration or quadrature scheme. In general, continuous time cost functions in OCPs have the following form:

$$J=\int_{t_0}^{t_f} l\left(\mathbf{x}(t),\mathbf{u}(t)\right)dt+l_f\left(\mathbf{x}\left(t_f\right)\right) \tag{3-74}$$

The cost function can be integrated using the trapezoid rule. Applying this integration scheme to the cost function yields the following:

$$J=\frac{1}{2}\sum_{k=0}^{N-1}\left[l_k\left(\mathbf{x}_k,\mathbf{u}_k\right)+l_{k+1}\left(\mathbf{x}_{k+1},\mathbf{u}_{k+1}\right)\right]\left(t_{k+1}-t_k\right)+l_N\left(\mathbf{x}_N\right) \tag{3-75}$$

For this work, the time-step $h=t_{k+1}-t_k$ is constant. Also, the cost function selected is a quadratic that has the same form as the objective utilized in the Convex MPC case. Therefore, the cost function in discrete time has the following form:

$$\begin{aligned} J=&\frac{h}{2}\sum_{k=0}^{N-1}\left(\mathbf{x}_k-\mathbf{x}_{\mathbf{ref}_k}\right)^T\mathbf{Q}_k\left(\mathbf{x}_k-\mathbf{x}_{\mathbf{ref}_k}\right)+\left(\mathbf{u}_k-\mathbf{u}_{\mathbf{ref}_k}\right)^T\mathbf{R}_k\left(\mathbf{u}_k-\mathbf{u}_{\mathbf{ref}_k}\right)\\ &+\left(\mathbf{x}_{k+1}-\mathbf{x}_{\mathbf{ref}_{k+1}}\right)^T\mathbf{Q}_{k+1}\left(\mathbf{x}_{k+1}-\mathbf{x}_{\mathbf{ref}_{k+1}}\right)+\left(\mathbf{u}_{k+1}-\mathbf{u}_{\mathbf{ref}_{k+1}}\right)^T\mathbf{R}_{k+1}\left(\mathbf{u}_{k+1}-\mathbf{u}_{\mathbf{ref}_{k+1}}\right)\\ &+\left(\mathbf{x}_N-\mathbf{x}_{\mathbf{ref}_N}\right)^T\mathbf{Q}_N\left(\mathbf{x}_N-\mathbf{x}_{\mathbf{ref}_N}\right), \qquad \mathbf{Q}_k\geq 0,\ \ \mathbf{R}_k>0 \end{aligned} \tag{3-76}$$

This trapezoid integration scheme is utilized for the DIRTRAN MPC. It is implemented by the function **ComputeNLPCostTrapezoidal(.)**, member of the custom class **NLP** located in the file *nlp_terms.cpp*. This function is presented in the code snippet below.

```
void NLP::ComputeNLPCostTrapezoidal(int planning_horizon_steps,
                                    double& planning_timestep,
                                    casadi::MX& X, casadi::MX& U,
                                    casadi::MX& x_goal, casadi::MX& Q,
                                    casadi::MX& R, casadi::MX& Qn,
                                    casadi::MX& J)
{
```

```
  int const N = X.size1() / planning_horizon_steps;
  int const M = U.size1() / plnning_horizon_steps;

  MX x_k = MX::sym("x_k", N);
  MX u_k = MX::sym("u_k", M);

  MX l = MX::sym("l");
  MX lf = MX::sym("lf");

  int i = 0;
  int j = 0;
  J = 0.0;

  // Stage Cost
  // idx = 0
  int idx = 0;
  x_k = X(Slice(i, i + N, 1));
  u_k = U(Slice(j, j + M, 1));
  l = mtimes(mtimes((x_k - x_goal).T(), Q), (x_k - x_goal)) +
      mtimes(mtimes(u_k.T(), R), u_k);
  J += DM(1.0) * l;
  i += N, j += M;
  idx++;

  // 1 < idx < horizon - 2
  while (idx < planning_horizon_steps - 2)
  {
    x_k = X(Slice(i, i + N, 1));
    u_k = U(Slice(j, j + M, 1));
    l = mtimes(mtimes((x_k - x_goal).T(), Q), (x_k - x_goal)) +
        mtimes(mtimes(u_k.T(), R), u_k);
    J += DM(2.0) * l;
    i += N, j += M;
    idx++;
  }

  // idx = horizon - 2
  x_k = X(Slice(i, i + N, 1));
  u_k = U(Slice(j, j + M, 1));
  l = mtimes(mtimes((x_k - x_goal).T(), Q), (x_k - x_goal)) +
      mtimes(mtimes(u_k.T(), R), u_k);
  J += DM(1.0) * l;
  i += N, j += M;

  // Terminal Cost
  // idx = horizon - 1
  x_k = X(Slice(i, i + N, 1), 0);

  lf = mtimes(mtimes((x_k - x_goal).T(), Qn), (x_k - x_goal));
  std::cout << "lf :\n " << lf << std::endl;

  // Cost function
  J = DM(0.5 * planning_timestep) * J + lf;
  std::cout << "J :\n " << J << std::endl;
}
```

One other option that can be utilized in DIRCOL is to integrate the cost function using Simpson's integration rule. Applying this integration scheme to the cost function yields the following:

$$J=\frac{1}{6}\sum_{k=0}^{N-1}\left[l_k\left(\mathbf{x}_k,\mathbf{u}_k\right)+4l_{k+\frac{1}{2}}\left(\mathbf{x}_{k+\frac{1}{2}},\mathbf{u}_{k+\frac{1}{2}}\right)+l_{k+1}\left(\mathbf{x}_{k+1},\mathbf{u}_{k+1}\right)\right]\left(t_{k+1}-t_k\right)+l_N\left(\mathbf{x}_N\right) \tag{3-77}$$

In case of a time-step $h=t_{k+1}-t_k$ that is constant and of a cost function that is a quadratic, the discretized cost function has the following form:

$$\begin{aligned} J=&\frac{h}{6}\sum_{k=0}^{N-1}\left(\mathbf{x}_k-\mathbf{x}_{\mathbf{ref}_k}\right)^T\mathbf{Q}_k\left(\mathbf{x}_k-\mathbf{x}_{\mathbf{ref}_k}\right)+\left(\mathbf{u}_k-\mathbf{u}_{\mathbf{ref}_k}\right)^T\mathbf{R}_k\left(\mathbf{u}_k-\mathbf{u}_{\mathbf{ref}_k}\right)\\ &+4\left(\mathbf{x}_{k+\frac{1}{2}}-\mathbf{x}_{\mathbf{ref}_{k+\frac{1}{2}}}\right)^T\mathbf{Q}_{k+\frac{1}{2}}\left(\mathbf{x}_{k+\frac{1}{2}}-\mathbf{x}_{\mathbf{ref}_{k+\frac{1}{2}}}\right)+4\left(\mathbf{u}_{k+\frac{1}{2}}-\mathbf{u}_{\mathbf{ref}_{k+\frac{1}{2}}}\right)^T\mathbf{R}_{k+\frac{1}{2}}\left(\mathbf{u}_{k+\frac{1}{2}}-\mathbf{u}_{\mathbf{ref}_{k+\frac{1}{2}}}\right)\\ &+\left(\mathbf{x}_{k+1}-\mathbf{x}_{\mathbf{ref}_{k+1}}\right)^T\mathbf{Q}_{k+1}\left(\mathbf{x}_{k+1}-\mathbf{x}_{\mathbf{ref}_{k+1}}\right)+\left(\mathbf{u}_{k+1}-\mathbf{u}_{\mathbf{ref}_{k+1}}\right)^T\mathbf{R}_{k+1}\left(\mathbf{u}_{k+1}-\mathbf{u}_{\mathbf{ref}_{k+1}}\right)\\ &+\left(\mathbf{x}_N-\mathbf{x}_{\mathbf{ref}_N}\right)^T\mathbf{Q}_N\left(\mathbf{x}_N-\mathbf{x}_{\mathbf{ref}_N}\right), \qquad \mathbf{Q}_k\geq 0,\ \ \mathbf{R}_k>0 \end{aligned} \tag{3-78}$$

This Simpson integration scheme is utilized for the DIRCOL MPC. It is implemented by the function **ComputeNLPCostSimpson(.)**, member of the custom class **NLP** located in the file *nlp_terms.cpp*. This function is presented in the two code snippets below.

```
void NLP::ComputeNLPSimpson(int planning_horizon_steps,
                            double& planning_timestep, casadi::MX& X,
                            casadi::MX& U, casadi::MX& x_goal,
                            casadi::MX& Q, casadi::MX& R,
                            casadi::MX& Qn, casadi::MX& J)
{
  int const N = X.size1() / planning_horizon_steps;
  int const M = U.size1() / planning_horizon_steps;

  MX x1 = MX::sym("x1", N);
  MX u1 = MX::sym("u1", M);

  MX x2 = MX::sym("x2", N);
  MX u2 = MX::sym("u2", M);

  MX x_k = MX::sym("x_k", N);
  MX u_k = MX::sym("u_k", M);

  MX f1 = MX::sym("f1", N);
  MX f2 = MX::sym("f2", N);

  MX xm = MX::sym("xm", N);
  MX um = MX::sym("um", M);

  VectorNd q;
  VectorNd q_dot;
  VectorNd tau;

  MX l = MX::sym("l");
  MX lf = MX::sym("lf");

  int i = 0;
  int j = 0;
  J = 0.0;

  // Stage Cost
  // idx = 0
  int idx = 0;
```

```
x_k = X(Slice(i, i + N, 1));
u_k = U(Slice(j, j + M, 1));
l = mtimes(mtimes((x_k - x_goal).T(), Q), (x_k - x_goal)) +
    mtimes(mtimes(u_k.T(), R), u_k);
J += DM(1.0) * l;
idx++;
```

```
// 1 < idx < horizon - 2
x1 = X(Slice(i, i + N, 1));
u1 = U(Slice(j, j + M, 1));
q = VectorNd(x1(Slice(0, int(0.5 * double(N)), 1), 0));
q_dot = VectorNd(x1(Slice(int(0.5 * double(N)), N, 1), 0));
tau = VectorNd(u1(Slice(0, M, 1), 0));
symbolicrobotdynamics->ContinuousDynamicsRBDL(q, q_dot, tau, f1);

while (idx < planning_horizon_steps - 2)
{
  x2 = X(Slice(i + N, i + 2 * N, 1));
  u2 = U(Slice(j + M, j + 2 * M, 1));
  q = VectorNd(x2(Slice(0, int(0.5 * double(N)), 1), 0));
  q_dot = VectorNd(x2(Slice(int(0.5 * double(N)), N, 1), 0));
  tau = VectorNd(u2(Slice(0, M, 1), 0));
  symbolicrobotdynamics->ContinuousDynamicsRBDL(q, q_dot, tau, f2);

  x_k = DM(0.5) * (x1 + x2) + DM(planning_timestep / 8.0) * (f1 - f2);
  u_k = DM(0.5) * (u1 + u2);

  l = mtimes(mtimes((x_k - x_goal).T(), Q), (x_k - x_goal)) +
      mtimes(mtimes(u_k.T(), R), u_k);
  J += DM(4.0) * l;

  l = mtimes(mtimes((x2 - x_goal).T(), Q), (x2 - x_goal)) +
      mtimes(mtimes(u2.T(), R), u2);
  J += DM(2.0) * l;

  x1 = x2;
  f1 = f2;

  i += N, j += M;
  idx++;
}

// idx = horizon - 2
x2 = X(Slice(i + N, i + 2 * N, 1));
u2 = U(Slice(j + M, j + 2 * M, 1));
q = VectorNd(x2(Slice(0, int(0.5 * double(N)), 1), 0));
q_dot = VectorNd(x2(Slice(int(0.5 * double(N)), N, 1), 0));
tau = VectorNd(u2(Slice(0, M, 1), 0));
symbolicrobotdynamics->ContinuousDynamicsRBDL(q, q_dot, tau, f2);

x_k = DM(0.5) * (x1 + x2) + DM(planning_timestep / 8.0) * (f1 - f2);
u_k = DM(0.5) * (u1 + u2);

l = mtimes(mtimes((x_k - x_goal).T(), Q), (x_k - x_goal)) +
    mtimes(mtimes(u_k.T(), R), u_k);
J += DM(4.0) * l;

l = mtimes(mtimes((x2 - x_goal).T(), Q), (x2 - x_goal)) +
    mtimes(mtimes(u2.T(), R), u2);
J += DM(1.0) * l;
```

```
  i += N, j += M;

  // Terminal Cost
  // idx = horizon - 1
  x_k = X(Slice(i, i + N, 1), 0);

  lf = mtimes(mtimes((x_k - x_goal).T(), Qn), (x_k - x_goal));
  std::cout << "lf :\n " << lf << std::endl;

  // Cost function
  J = DM((planning_timestep / 3.0)) * J + lf;
  std::cout << "J :\n " << J << std::endl;
}
```

The DIRTRAN/DIRCOL MPC controllers are implemented in the files DIRTRAN_MPC.cpp and DIRCOL_MPC.cpp respectively. The solver that is utilized here to solve the NLPs is IPOPT (Interior Point OPTimizer), a software package for large-scale nonlinear optimization. It is also interfaced to CasADi [150], which is an open-source tool for nonlinear optimization and algorithmic differentiation. Also, IPOPT relies on third-party software for some linear algebra routines. Of the biggest significance is the need to solve sparse, symmetric, indefinite linear systems of equations. Such a sparse symmetric indefinite linear solver is provided by the HSL (Harwell Subroutine Library) Mathematical Software Library [110] which is being utilized for this work. To be more specific, the solver MA57 is employed here.

The solver IPOPT is designed to find (local) solutions to NLPs using an IP line search filter method [151]. IP methods replace inequality constraints with a barrier function, that is usually logarithmic, that is being added to the cost function. At the constraint boundary, this term becomes infinitely large and thus constraint violations get penalized, and the constraint boundary is never crossed. To be more precise, barrier function converges to infinity, as the solution of the problem approaches the constraint boundary. Their main difference with penalty methods is that penalty terms are nonzero only when the constraint boundaries are crossed. Only if the constraint boundaries are crossed, do these methods try to force the solution to lay inside the boundaries. For this reason, penalty methods are often referred to as exterior penalty methods. IP methods prevent the solution from ever crossing the constraint boundary, in the first place. The solution always lies in the interior.

The solver IPOPT can handle mathematical optimization problems of the form:

$$\begin{aligned} \min_{\mathbf{z}} \quad & f(\mathbf{z}) \\ \text{s.t.} \quad & \mathbf{z}_{\text{lb}} \le \mathbf{z} \le \mathbf{z}_{\text{ub}} \\ & \mathbf{g}_{\text{lb}} \le \mathbf{g}(\mathbf{z}) \le \mathbf{g}_{\text{ub}} \end{aligned} \tag{3-79}$$

where $\mathbf{z} \in \mathbb{R}^{(n+m)N}$ is the optimization variable comprised of the states and control inputs over the entire horizon, $f(\mathbf{z}): \mathbb{R}^{(n+m)N} \to \mathbb{R}$ denotes the objective function and $\mathbf{g}(\mathbf{z}): \mathbb{R}^{(n+m)N} \to \mathbb{R}^q$ denotes the constraint functions, with q being equal to the number of all the constraint equations. These two functions are in general linear or nonlinear and convex or nonconvex. The vectors $\mathbf{z}_{\text{lb}}, \mathbf{z}_{\text{ub}} \in \mathbb{R}^{(n+m)N}$ denote the lower and upper bounds on the decision variables, so that $\mathbf{z}_{\text{lb}_i} \in \mathbb{R} \cup \{-\infty\}$ and $\mathbf{z}_{\text{ub}_i} \in \mathbb{R} \cup \{+\infty\}$, $\forall i = \{1, \ldots, (n+m)N\}$. Finally, the vectors $\mathbf{g}_{\text{lb}}, \mathbf{g}_{\text{ub}} \in \mathbb{R}^q$ denote the lower and upper bounds on the constraints, so that $\mathbf{g}_{\text{lb}_i} \in \mathbb{R} \cup \{-\infty\}$ and $\mathbf{g}_{\text{ub}_i} \in \mathbb{R} \cup \{+\infty\}$, $\forall i = \{1, \ldots, q\}$. Equality constraints can be encoded by setting $\mathbf{z}_{\text{lb}_i} = \mathbf{z}_{\text{ub}_i}$

or $\mathbf{g}_{\mathbf{lb}_i} = \mathbf{g}_{\mathbf{ub}_i}$ for some of the elements in vectors $\mathbf{z}_{\mathbf{lb}}, \mathbf{z}_{\mathbf{ub}}$ or $\mathbf{g}_{\mathbf{lb}}, \mathbf{g}_{\mathbf{ub}}$. Consequently, the terms of the problem of equation (3-4) have to be rearranged to match the form of (3-79).

The interface with CasADi requires that the decision variables $\mathbf{z}$ and all the terms of the NLP that are dependent on the decision variables, like $f(\mathbf{z})$ and $\mathbf{g}(\mathbf{z})$, must be represented using CasADi's symbolic variables. The symbolic framework of CasADi allows the user to construct symbolic expressions using a syntax similar to the one that MATLAB uses. According to this syntax, matrices, vectors, and scalars are treated uniformly, as if they all were matrices. However, under these conditions, Pinocchio cannot be utilized for the computation of system dynamics since it is not compatible with this symbolic format. Its various functions cannot accept as arguments or process these symbolic variables. On the other hand, RBDL is compatible with it, thanks to the rbdl-casadi version of the RBDL library. The terms that are constant and not dependent on the decision variables, like the bounds $\mathbf{z}_{\mathbf{lb}_i} = \mathbf{z}_{\mathbf{ub}_i}$ or $\mathbf{g}_{\mathbf{lb}_i} = \mathbf{g}_{\mathbf{ub}_i}$ are represented using std vectors, provided by the C++ library std, instead of Eigen vectors. This action has to be taken since CasADi is not compatible with Eigen.

The decision variables vector $\mathbf{z}$ will be equal to $\mathbf{z} = \begin{bmatrix} \mathbf{x}_0 & \dots & \mathbf{x}_N & \mathbf{u}_0 & \dots & \mathbf{u}_N \end{bmatrix}^T$. According to this arrangement of the decision variables, will the rest of the NLP terms be selected. Thus, objective function $f(\mathbf{z})$ and the constraints function $\mathbf{g}(\mathbf{z})$ are the following:

$$f(\mathbf{z}) = J, \;\; \mathbf{g}(\mathbf{z}) = \begin{bmatrix} \boldsymbol{\Delta}_0 \\ \boldsymbol{\Delta}_1 \\ \vdots \\ \boldsymbol{\Delta}_{N-1} \end{bmatrix} \tag{3-80}$$

In this work, it is assumed that the only constraint equations present on the problem are the dynamical system constraints. As a result, $q = (N-1)n$. This assumption justifies the value of the vector $\mathbf{g}(\mathbf{z})$. After taking this assumption into consideration, the bounds on the decision variables $\mathbf{z}_{\mathbf{lb}}, \mathbf{z}_{\mathbf{ub}}$ and the bounds on the constraints $\mathbf{g}_{\mathbf{lb}}, \mathbf{g}_{\mathbf{ub}}$ will be the following:

$$\mathbf{z}_{\mathbf{lb}} = \begin{bmatrix} \mathbf{x}_0 \\ \underline{\mathbf{x}} \\ \vdots \\ \underline{\mathbf{x}} \\ \underline{\mathbf{u}} \\ \vdots \\ \underline{\mathbf{u}} \end{bmatrix}, \;\; \mathbf{z}_{\mathbf{ub}} = \begin{bmatrix} \mathbf{x}_0 \\ \overline{\mathbf{x}} \\ \vdots \\ \overline{\mathbf{x}} \\ \overline{\mathbf{u}} \\ \vdots \\ \overline{\mathbf{u}} \end{bmatrix} \tag{3-81}$$

$$\mathbf{g}_{\mathbf{lb}} = \begin{bmatrix} \mathbf{0} \\ \vdots \\ \mathbf{0} \end{bmatrix}, \;\; \mathbf{g}_{\mathbf{ub}} = \begin{bmatrix} \mathbf{0} \\ \vdots \\ \mathbf{0} \end{bmatrix} \tag{3-82}$$

By setting the upper bounds on the decision variables that correspond to the initial state (first n elements) equal to the lower bounds on the decision variables that correspond to the initial state (first n elements), an equality constraint gets defined. However, if both of them are equal to the initial state itself $\mathbf{x}_0$, then the solver recognizes the first elements n of the vector $\mathbf{z}$ as constant and does not alter their value during the solution process.

The aforementioned NLP vectors are filled in after the code in the file *nlp_terms.cpp* is executed. The functions that should be executed to do so are

ComputeNLPDefectConstraintTrapezoidal(.)/ComputeNLPDefectConstraintHermiteSimpson(.), **ComputeNLPCostTrapezoidal(.)/ComputeNLPCostSimpson(.)**, **ComputeNLPDecisionVariablesBounds(.)** and **ComputeNLPConstraintBounds(.),** members of the custom class **NLP**. The last two are presented here:

```
void NLP::ComputeNLPDecisionVariablesBounds(
    int planning_horizon_steps, std::vector<double>& x0, VectorNx& xmax,
    VectorNx& xmin, VectorNu& umax, VectorNu& umin,
    std::vector<double>& lbz, std::vector<double>& ubz)
{
  // Initial decision variables bounds (lbz, ubz) values

  int const N = xmax.rows();
  int const M = umax.rows();

  // States

  // n = 0
  for (int i = 0; i < N; i++)
  {
    lbz.push_back(x0[i]), ubz.push_back(x0[i]);
  }

  // n = 1
  for (int n = 1; n < planning_horizon_steps; n++)
  {
    for (int i = 0; i < N; i++)
    {
      lbz.push_back(xmin(i)), ubz.push_back(xmax(i));
    }
  }

  // Controls
  for (int n = 0; n < planning_horizon_steps; n++)
  {
    for (int i = 0; i < M; i++)
    {
      lbz.push_back(umin(i)), ubz.push_back(umax(i));
    }
  }

  std::cout << "lbz :\n" << lbz << std::endl;
  std::cout << "ubz :\n" << ubz << std::endl;
}
```

```
void NLP::ComputeNLPConstraintBounds(casadi::MX& constraint,
                                     std::vector<double>& lbg,
                                     std::vector<double>& ubg)
{
  // Initial constraints bounds (lbg, ubg) values

  for (int n = 0; n < constraint.size1(); n++)
  {
    lbg.push_back(0.0), ubg.push_back(0.0);
  }

  std::cout << "lbg :\n" << lbg << std::endl;
  std::cout << "ubg :\n" << ubg << std::endl;
}
```

Then an instance of the solver must be created, and the solver must then be initialized. This initialization procedure includes the declaration of the NLP terms and the specification of the values of the solver parameters. The procedure followed to instantiate and initialize the solver is described in the code snippet below.

```
// Declaration of the NLP
MXDict nlp_prob;                // NLP declaration
nlp_prob["x"] = vertcat(X, U); // decision variables
nlp_prob["f"] = J;              // objective
nlp_prob["g"] = constraint;     // constraints

// Casadi and IPOPT (solver) settings
Dict opts_dict;
opts_dict["expand"] = 1;

opts_dict["warn_initial_bounds"] = 1;
opts_dict["eval_errors_fatal"] = 1;
opts_dict["regularity_check"] = 1;
opts_dict["inputs_check"] = 1;

opts_dict["ipopt.print_level"] = 0; //5
opts_dict["ipopt.print_user_options"] = "yes";
opts_dict["ipopt.print_timing_statistics"] = "no";

opts_dict["ipopt.mu_init"] = 1e-3;

opts_dict["ipopt.linear_solver"] = "ma57";
opts_dict["ipopt.ma57_pre_alloc"] = 1.5;
opts_dict["ipopt.tol"] = 1e-3;
// opts_dict["ipopt.acceptable_tol"] = 1e-6;
// opts_dict["ipopt.acceptable_obj_change_tol"] = 1e-6;
opts_dict["ipopt.dual_inf_tol"] = 1e10;
opts_dict["ipopt.constr_viol_tol"] = 1e-2;
opts_dict["ipopt.compl_inf_tol"] = 1e-2;
opts_dict["ipopt.jacobian_approximation"] = "exact";
opts_dict["ipopt.gradient_approximation"] = "exact";
opts_dict["ipopt.fixed_variable_treatment"] = "make_parameter_nodual";
opts_dict["ipopt.jac_d_constant"] =
    "yes"; // Indicates linear inequality constraints
// opts_dict["ipopt.hessian_approximation"] = "limited-memory";

opts_dict["ipopt.warm_start_init_point"] = "yes";
opts_dict["ipopt.warm_start_bound_push"] = 1e-6;
opts_dict["ipopt.warm_start_slack_bound_push"] = 1e-6;
opts_dict["ipopt.warm_start_mult_bound_push"] = 1e-6;
opts_dict["ipopt.warm_start_bound_frac"] = 1e-6;
opts_dict["ipopt.warm_start_slack_bound_frac"] = 1e-6;

opts_dict["ipopt.max_wall_time"] = 0.01;
opts_dict["ipopt.max_cpu_time"] = 0.01;

// Create solver instance
solver = nlpsol("solver", "ipopt", nlp_prob, opts_dict);

// Declare NLP constraint bounds
DMDict arg_nlp;
arg_nlp["x0"] = z0_std;
```

```
arg_nlp["lam_x0"] = lam_z0_std;
arg_nlp["lam_g0"] = lam_g0_std;
arg_nlp["lbx"] = lbz_std;
arg_nlp["ubx"] = ubz_std;
arg_nlp["lbg"] = lbg_std;
arg_nlp["ubg"] = ubg_std;
```

However, some NLP terms, like the bounds $\mathbf{z}_{\mathbf{lb}}, \mathbf{z}_{\mathbf{ub}}$, are dependent on the initial states $\mathbf{x}_0$. Consequently, they should be updated with the new values of the initial conditions $\mathbf{x}_0$, at every control loop iteration. Additionally, the initial guess of the decision variables (primal solution) as well as the guess for the Lagrange multipliers (dual solution) should also be updated. A vector of Lagrange multipliers corresponds for each constraint set. For this optimization problem the first vector of Lagrange multipliers corresponds to the bounds on the decision variables $\boldsymbol{\lambda}_{\mathbf{z}} \in \mathbb{R}^{(n+m)N}$ and second vector of Lagrange multipliers corresponds to the bounds on the constraints $\boldsymbol{\lambda}_{\mathbf{g}} \in \mathbb{R}^{q}$, where $q = (N-1)n$. The initial guess for the primal solution is denoted by $\mathbf{z}_0$ and the initial guess for the dual solution associated with the bounds on the decision variables and with the bounds on the constraints are denoted by $\boldsymbol{\lambda}_{\mathbf{z}_0}, \boldsymbol{\lambda}_{\mathbf{g}_0}$ respectively. Then the solver will be able to solve the problem. After solving it, the user can extract the decision variables for optimal solution $\mathbf{z}^*$, the Lagrange multipliers associated with the bounds on the decision variables at the solution $\boldsymbol{\lambda}^*_{\mathbf{z}}$, the Lagrange multipliers associated with the bounds on the constraints at the solution $\boldsymbol{\lambda}^*_{\mathbf{g}}$, the cost function for the optimal solution f^* and the constraints evaluated at the optimal solution $\mathbf{g}^*$. The actions mentioned in this paragraph are illustrated in the code snippet below:

```
// Update NLP initial conditions and decision variables bounds
arg_nlp["x0"] = z0_std;
arg_nlp["lam_x0"] = lam_z0_std;
arg_nlp["lam_g0"] = lam_g0_std;
arg_nlp["lbx"] = lbz_std;
arg_nlp["ubx"] = ubz_std;

// Solve the NLP
DMDict res_nlp = solver(arg_nlp);

// Print solution
// cout << "Optimal cost:                 \n"
//      << double(res_nlp.at("f")) << endl;
// cout << "Primal solution:              \n"
//      << vector<double>(res_nlp.at("x")) << endl;
// cout << "Constraints:                  \n"
//      << vector<double>(res_nlp.at("g")) << endl;
// cout << "Dual solution (bounds on X): \n"
//      << vector<double>(res_nlp.at("lam_x")) << endl;
// cout << "Dual solution (bounds on G): \n"
//      << vector<double>(res_nlp.at("lam_g")) << endl;

// NLP solution
NLPSolution = vector<double>(res_nlp.at("x"));

// std::cout << "Here is the NLPSolution:\n " << NLPSolution
//           << std::endl;
```

The first m elements of the vector $\mathbf{z}^*$ are equal to the control input $\mathbf{u}_0$. Then, this control input is applied to the system. Afterwards, the system responds to that control action and the current state of the system gets measured $\mathbf{x}_{\mathbf{meas}}$. In general, $\mathbf{x}_{\mathbf{meas}}$ will not be equal to the expected (predicted) state response $\mathbf{x}_1$. So, the new initial state $\mathbf{x}_0$ is set equal to $\mathbf{x}_{\mathbf{meas}}$, the NLP terms get updated, and the entire iterative procedure gets repeated at every simulation step. The initial guesses $\mathbf{z}_0$ and $\boldsymbol{\lambda}_{\mathbf{z}_0}, \boldsymbol{\lambda}_{\mathbf{g}_0}$ are warm started with the primal and dual solution respectively that were acquired in the previous control loop. This has to be done manually when using IPOPT. This entire procedure is sketched in detail in the flow chart of Figure 3-21. Also, the directory tree of this package is depicted in Figure 3-22.

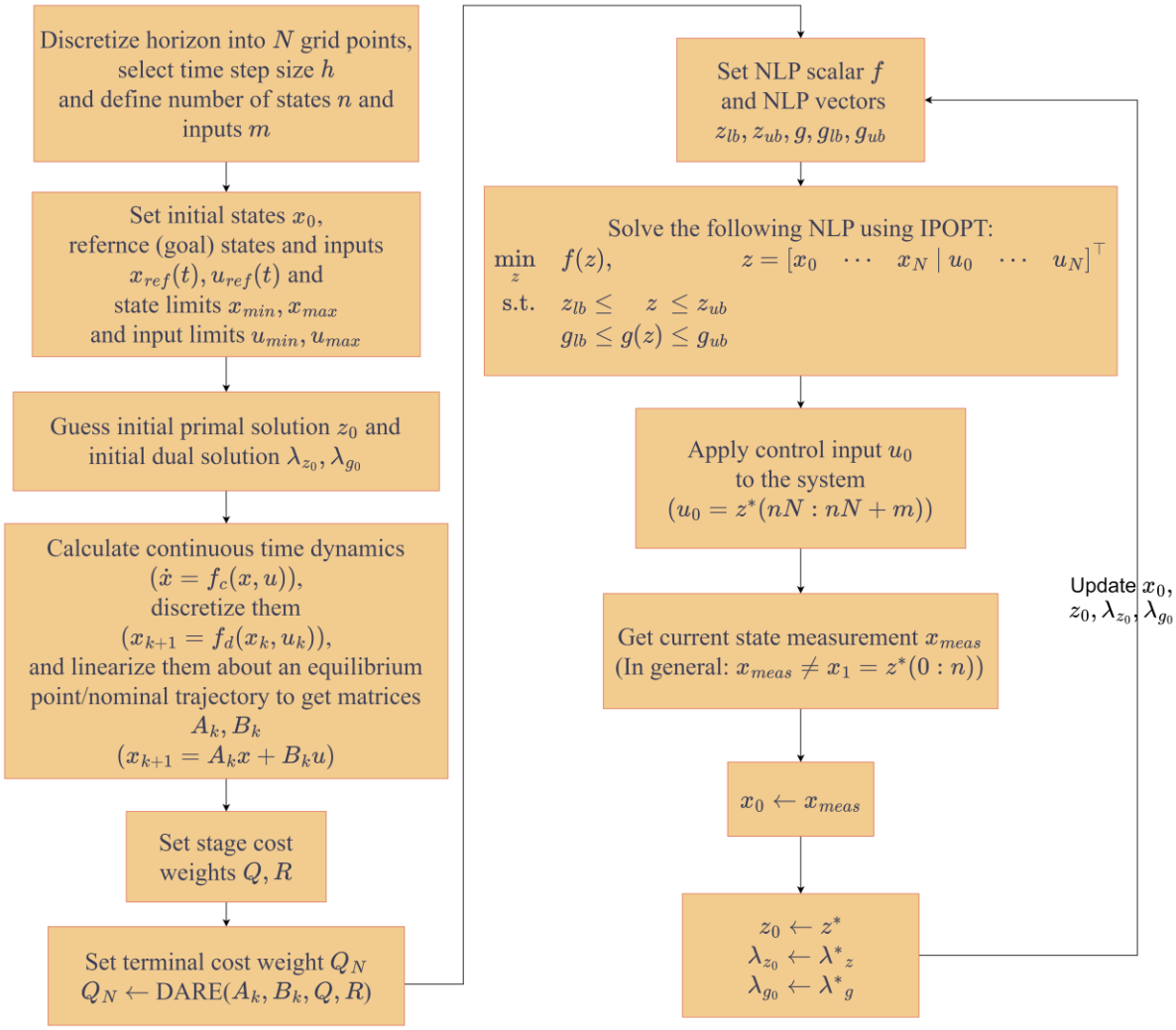


Figure 3-21. DIRTRAN/DIRCOL MPC flow chart.

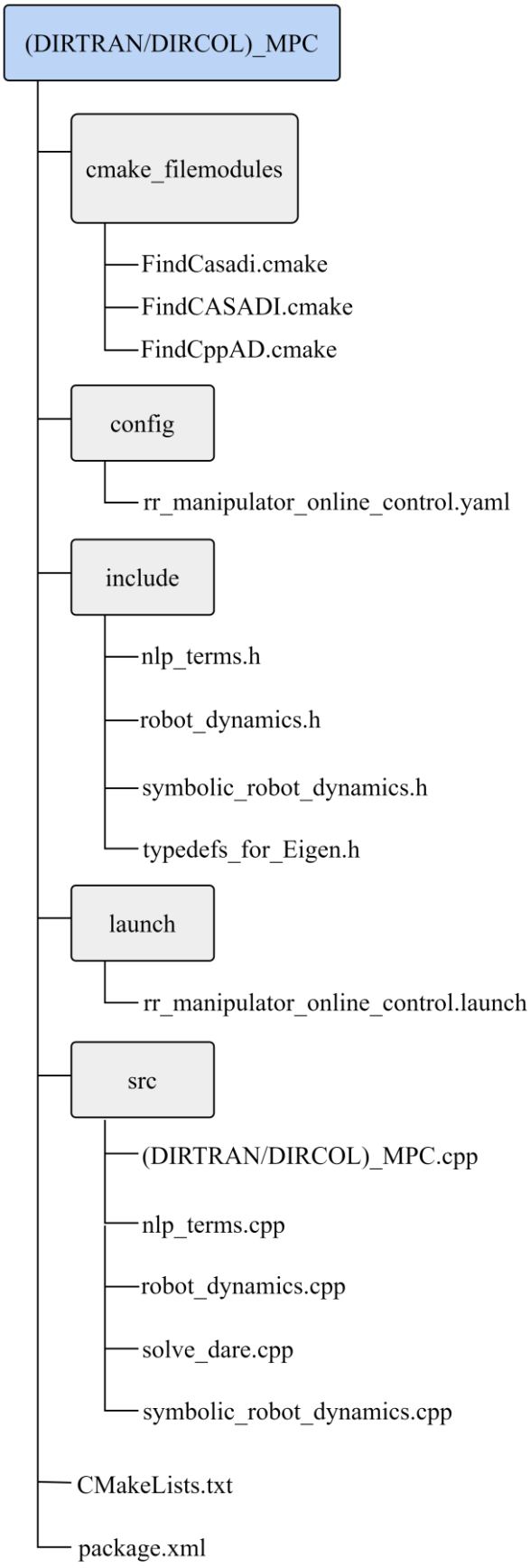


Figure 3-22. Directory tree of (DIRTRAN/DIRCOL)_MPC package.

3.6 Experiments for double pendulum in Gazebo

The numerical optimization techniques whose implementation was discussed in chapter 3.5, have been tested on a fully actuated double pendulum, also known as RR manipulator, in Gazebo simulator illustrated in Figure 3-23. Each actuator is mounted on the joints of the manipulator. The geometric and inertial parameters of the double pendulum are presented in Table 3-1. For this dynamical system, state variables $\mathbf{x}$ and the control inputs $\mathbf{u}$ are the following:

$$\mathbf{x}=\begin{bmatrix}\theta_1\\ \theta_2\\ \dot{\theta}_1\\ \dot{\theta}_2\end{bmatrix},\quad \mathbf{u}=\begin{bmatrix}\tau_1\\ \tau_2\end{bmatrix} \tag{3-83}$$

where θ_1,θ_2 are the angular positions of the two joints of the manipulator, $\dot{\theta}_1,\dot{\theta}_2$ are the angular velocities of the two joints of the manipulator and τ_1,τ_2 are the torques applied to each of the joints of the manipulator. The goal of each of the controllers tested is to bring the pendulum from an initial position to the up-ward position, at which $\theta_1=\theta_2=0\text{rad}$ with zero velocity $\dot{\theta}_1=\dot{\theta}_2=0\text{rad/s}$. This position is also an equilibrium point. Therefore, the state and control input references $\mathbf{x}_{\mathbf{ref}},\mathbf{u}_{\mathbf{ref}}$ are the following: $\mathbf{x}_{\mathbf{ref}}=\mathbf{0}\in\mathbb{R}^4$, $\mathbf{u}_{\mathbf{ref}}=\mathbf{0}\in\mathbb{R}^2$.

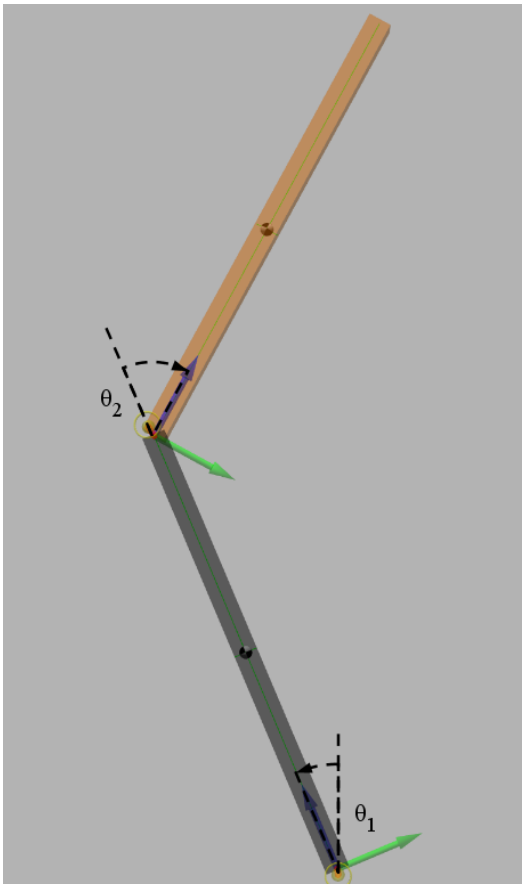


Figure 3-23. Double pendulum in Gazebo.

Table 3-1. Geometric and Inertial parameters of the double pendulum.

Parameter	Value
Link 1	

Mass	1.0kg
Length	2.0m
Distance between the CoM and the 1st joint	1.0m
Inertia Matrix	$\{I_{xx}=0.05,\ I_{yy}=0.05,\ I_{zz}=0.05\}\text{kgm}^2$
Actuator max torque	50.0Nm
Link 2	
Mass	1.0kg
Length	2.0m
Distance between the CoM and the 2nd joint	1.0m
Inertia Matrix	$\{I_{xx}=0.05,\ I_{yy}=0.05,\ I_{zz}=0.05\}\text{kgm}^2$
Actuator max torque	50.0Nm

The cost function employed for all the controllers is a quadratic that has the form described in equation (3-56). The stage cost weight matrices corresponding to the states and the control inputs $\mathbf{Q},\mathbf{R}$ are constant and have the same value for all the controllers. Their values are the following:

$$\mathbf{Q}=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix},\quad \mathbf{R}=\begin{bmatrix}0.01&0\\0&0.01\end{bmatrix} \tag{3-84}$$

3.6.1 Convex (QP-based) MPC

Firstly, experiments conducted on the Convex MPC controller are presented. Here the double pendulum dynamics are linearized around the equilibrium point that corresponds to the upward position of the double pendulum. This equilibrium point coincides with the reference state and control input $\mathbf{x}_{\mathbf{ref}},\mathbf{u}_{\mathbf{ref}}$. This linearization of the dynamics is valid from a small neighborhood around the linearizing point. Consequently, the initial state is chosen so that it lies in this neighborhood. To be more precise the initial angular positions of the joints are $\theta_{1,0}=0.4\text{rad}$, $\theta_{2,0}=0.0\text{rad}$ and the initial angular velocities of the joints are $\dot{\theta}_{1,0}=0.0\text{rad/s}$, $\dot{\theta}_{2,0}=0.0\text{rad/s}$. The size of the constant integration time step that is utilized by the RK4 explicit integrator is $h=1\text{msec}$, that is equal to the constant simulation time-step. The prediction horizon is divided into $N=2$ time steps. This means that the horizon length is equal to 2msec . Longer horizons could also be achieved but are not necessary for this task.

The time responses of the angular positions, the angular velocities and the torques of the joints of the manipulator are presented in Figure 3-24, Figure 3-25 and Figure 3-26 respectively. The responses are considerably fast since the steady state is reached in just 4sec. Also, not many oscillations are present in the responses. Finally, the actuators do not reach their maximum torque limits during the execution of the control law.

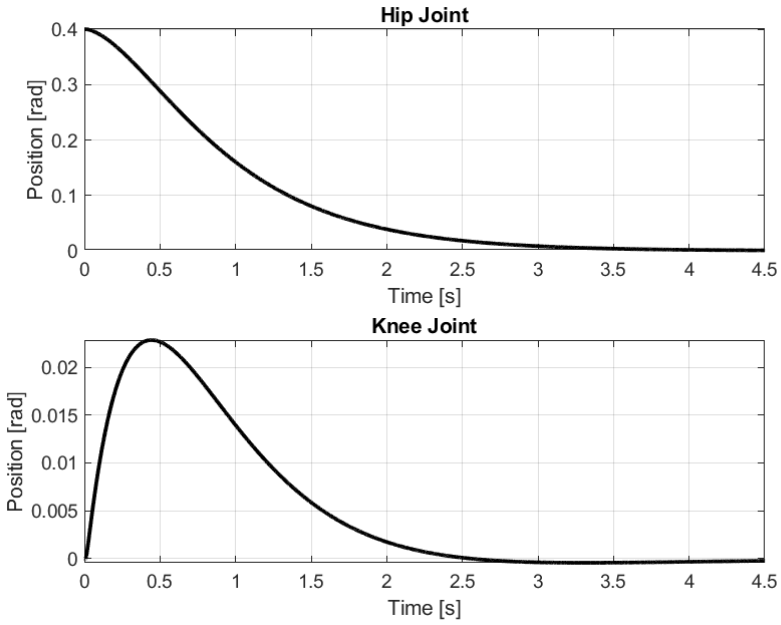


Figure 3-24. Time responses of the angular positions of the joints of the double pendulum.

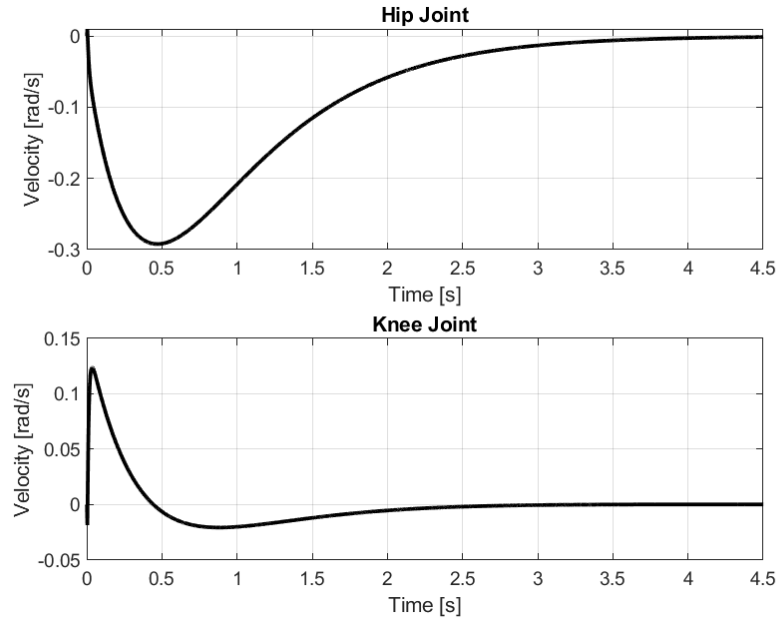


Figure 3-25. Time responses of the angular velocities of the joints of the double pendulum.

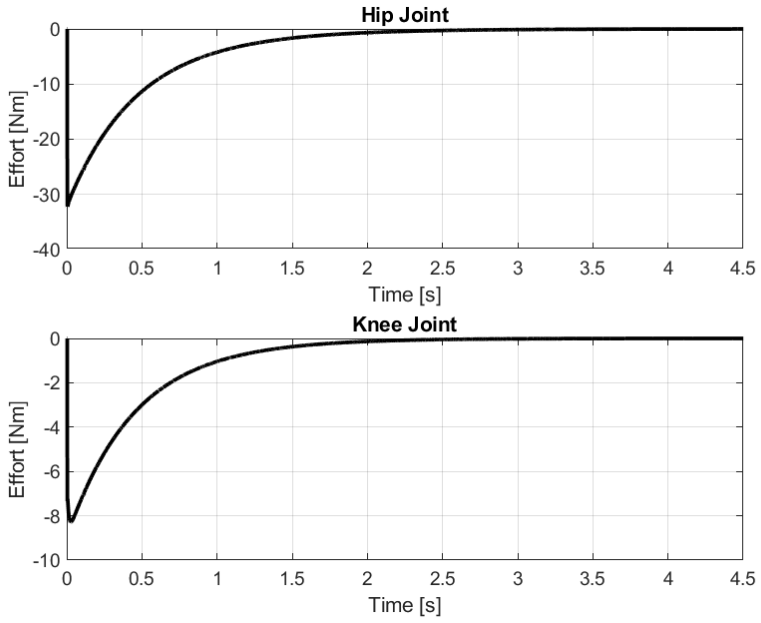


Figure 3-26. Time responses of the torques of the joints of the double pendulum.

The distribution of all Convex MPC solve time is presented in Figure 3-27. The mean value of solve time is equal to 0.3msec and the standard deviation of solve times is equal to 0.06msec. It is important to note that the vast majority of solve times of this controller exhibits small variation around the mean solve time.

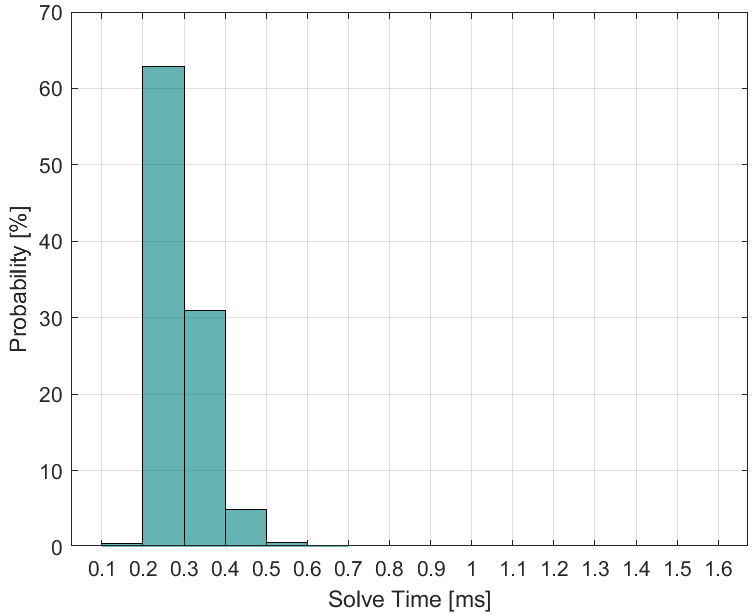


Figure 3-27. Solve time distribution of Convex MPC.

3.6.2 BOX-DDP MPC

Secondly, experiments conducted on the BOX-DDP MPC controller are presented. Here the controller makes use of the nonlinear dynamics of the manipulator. The initial state is chosen so that it corresponds to the double pendulum downward position. To be more precise the initial angular positions of the joints are $\theta_{1,0} = \pi\text{rad}$, $\theta_{2,0} = 0.0\text{rad}$ and the initial angular

velocities of the joints are $\dot{\theta}_{1,0} = 0.0\text{rad/s}$, $\dot{\theta}_{2,0} = 0.0\text{rad/s}$. The size of the constant integration time step that is utilized by the RK4 explicit integrator is $h = 100\text{msec}$. The prediction horizon is divided into $N = 2$ time steps. This means that the horizon length is equal to 200msec. The number of horizon time steps is kept the same to make a fair comparison with the Convex MPC. In general, the computational cost gets linearly increased with the number of horizon time steps. However, the time step size here was increased to have a longer horizon. The longer horizon, in comparison with the Convex MPC horizon, is necessary because the manipulator must cover a longer distance in the state space to reach the goal state. Longer horizons could be achieved but only to some extent. As was also mentioned in chapter 3.2.1, very long horizons are not feasible in DDP due to numerical ill-conditioning that is caused by error accumulation in the backward pass.

The time responses of the angular positions, the angular velocities and the torques of the joints of the manipulator are presented in Figure 3-28, Figure 3-29 and Figure 3-30. The responses are considerably fast as well since the steady state is reached in just 4sec. Also, not many oscillations are present in the responses. Finally, the hip actuator does reach the control effort saturation limits during the execution of the control law. This hip actuators control effort is saturated for approximately 0.5sec, without exceeding this limit.

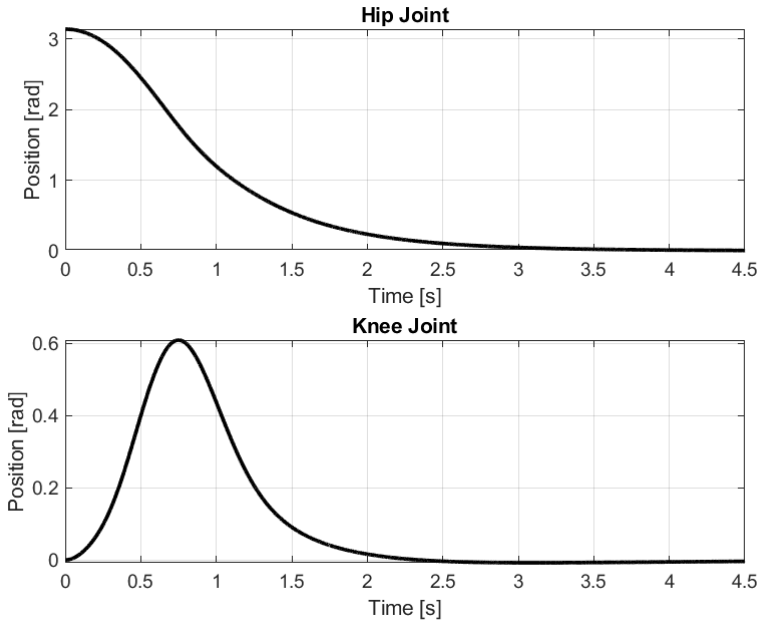


Figure 3-28. Time responses of the angular positions of the joints of the double pendulum.

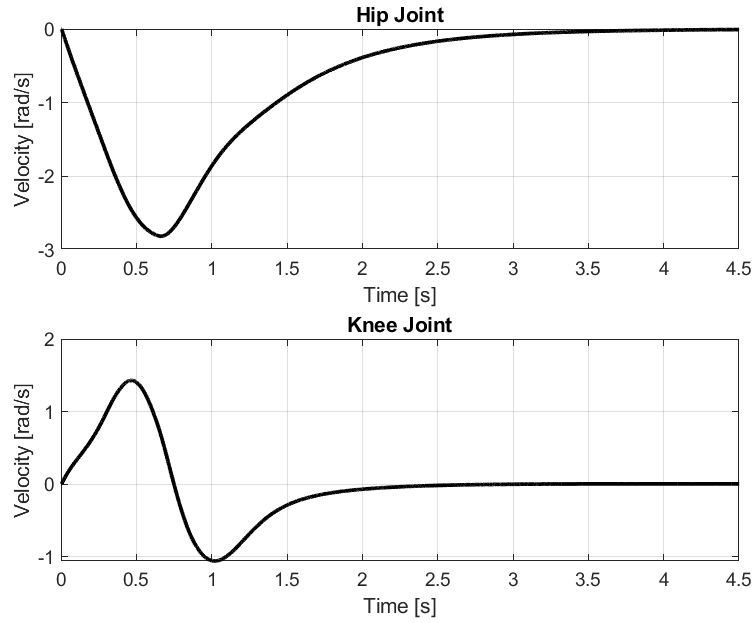


Figure 3-29. Time responses of the angular velocities of the joints of the double pendulum.

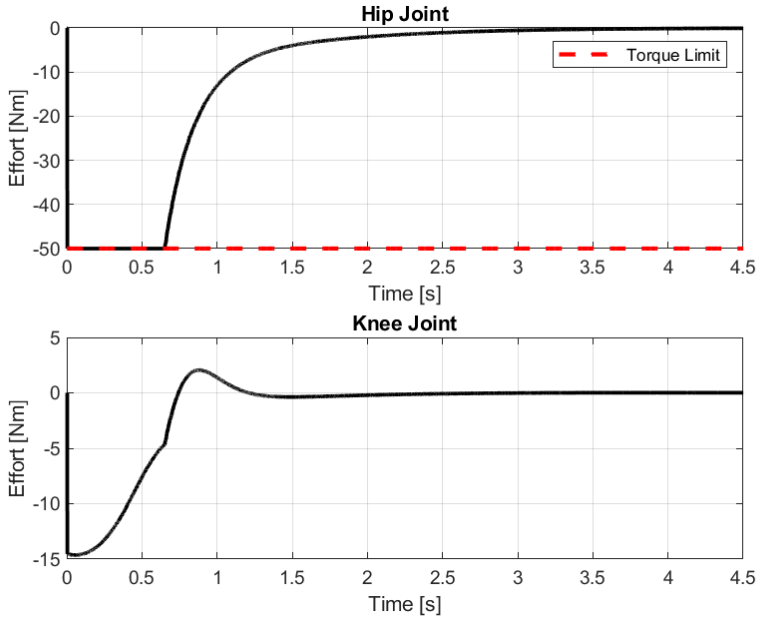


Figure 3-30. Time responses of the torques of the joints of the double pendulum.

The distribution of all BOX-DDP MPC solve time is presented in Figure 3-31. The mean value of solve time is equal to $1.0\,\mathrm{msec}$ and the standard deviation of solve times is equal to $0.52\,\mathrm{msec}$. The mean solve time is approximately three orders of magnitude higher than the one of Convex MPC. Also, the solve times are more dispersed around the mean value, when compared to Convex-MPC, which is justified by the higher standard deviation of the solve times. Consequently, the Convex-MPC controller is faster than the BOX-DDP one even though the problem that it solves has more decision variables than the BOX-DDP one. In BOX-DDP only the control inputs are decision variables, while in Convex-MPC are both the states and control inputs.

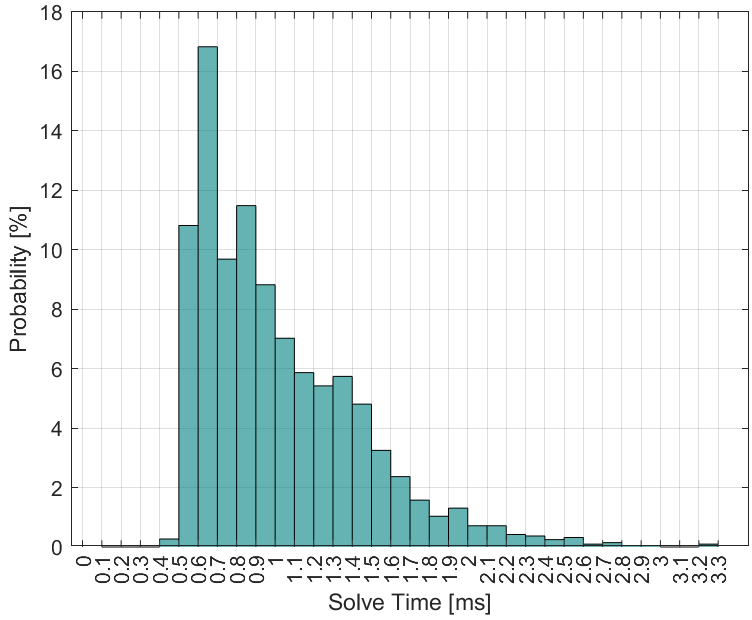


Figure 3-31. Solve time distribution of BOX-DDP MPC.

3.6.3 DIRTRAN MPC

Afterwards, experiments conducted on the DIRTRAN MPC controller are presented. Here also the controller makes use of the nonlinear dynamics of the manipulator. The initial state is chosen so that it corresponds to the double pendulum downward position. To be more precise the initial angular positions of the joints are $\theta_{1,0} = \pi \text{rad}$, $\theta_{2,0} = 0.0 \text{rad}$ and the initial angular velocities of the joints are $\dot{\theta}_{1,0} = 0.0 \text{rad/s}$, $\dot{\theta}_{2,0} = 0.0 \text{rad/s}$. The size of the constant integration time step that is utilized by the RK4 explicit integrator is $h = 100 \text{msec}$, which is equal to the one used in BOX-DDP MPC. However, the prediction horizon is divided into $N = 10$ time steps. This means that the horizon length is equal to 1sec. The number of time steps here was increased, in comparison to the BOX-DDP MPC, to achieve a more desirable response. To be more precise, when the number of time steps was smaller, the steady state error was significantly larger. In the previous two controllers, the state was integrated using RK4. This integration scheme is equivalent to utilizing piecewise cubic polynomials for the state interpolation. However, DIRTRAN utilizes piecewise linear polynomials for the state interpolation. This selection is not appropriate enough for the state trajectory of a highly nonlinear system. Therefore, the number of the horizon time steps has to be increased to compensate for the low order state interpolation scheme.

The time responses of the angular positions, the angular velocities and the torques of the joints of the manipulator are presented in Figure 3-32, Figure 3-33 and Figure 3-34. The responses are considerably fast as well since the steady state is reached in just 4sec. Also, not many oscillations are present in the responses. Finally, the hip actuator does reach the control effort saturation limits during the execution of the control law. This hip actuators control effort is saturated for approximately 0.1sec, without exceeding this limit. This duration is smaller than the one in BOX-DDP MPC.

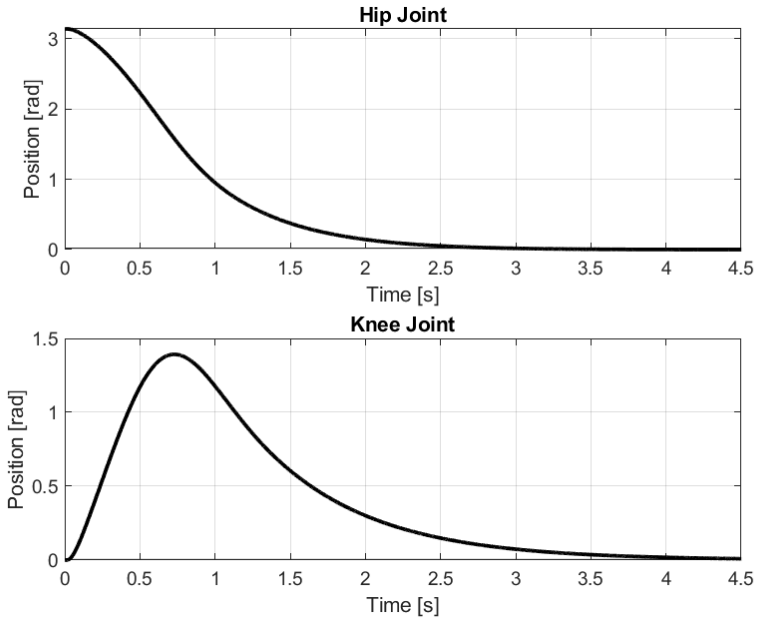


Figure 3-32. Time responses of the angular positions of the joints of the double pendulum.

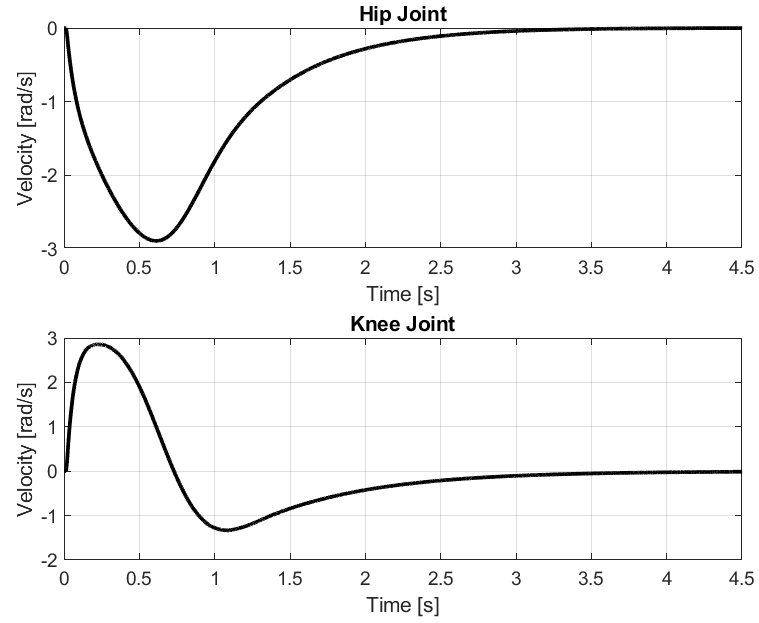


Figure 3-33. Time responses of the angular velocities of the joints of the double pendulum.

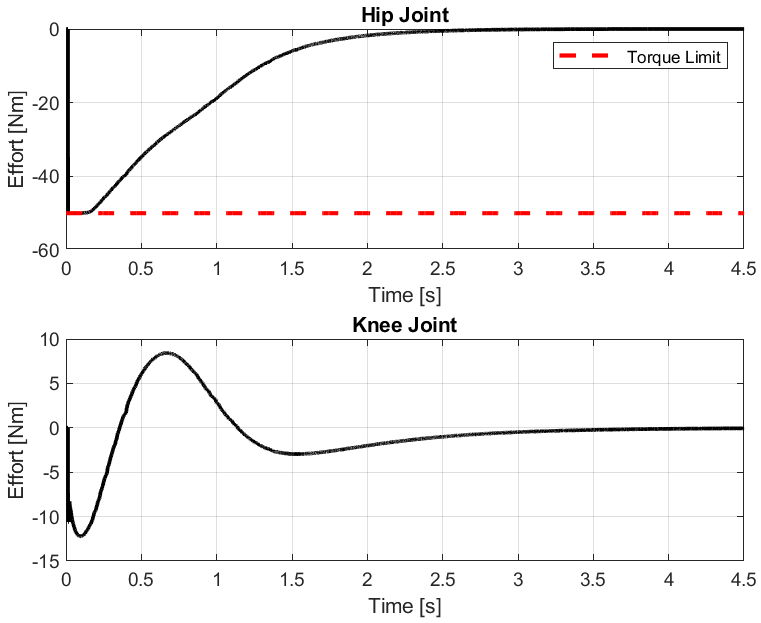


Figure 3-34. Time responses of the torques of the joints of the double pendulum.

The distribution of all DIRTRAN MPC solve time is presented in Figure 3-35. The mean value of solve time is equal to 2.3msec and the standard deviation of solve times is equal to 1.1msec. The mean solve time is approximately two orders of magnitude lower than the one of BOX-DDP MPC. Also, the solve times are more dispersed around the mean value, when compared to BOX-DPP MPC ones, which is justified by the higher standard deviation of the solve times. Consequently, the BOX-DDP MPC controller is faster than the DIRTRAN DDP one, which was expected.

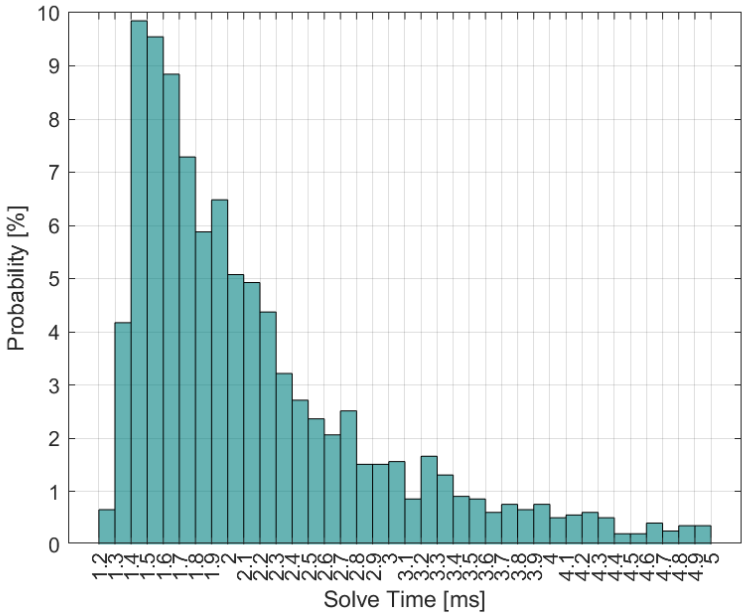


Figure 3-35. Solve time distribution of DIRTRAN MPC.

3.6.4 DIRCOL MPC

Finally, experiments conducted on the DIRCOL MPC controller are presented. Here also the controller makes use of the nonlinear dynamics of the manipulator. The initial state is chosen so that it corresponds to the double pendulum downward position. To be more precise the initial angular positions of the joints are $\theta_{1,0} = \pi \text{rad}$, $\theta_{2,0} = 0.0 \text{rad}$ and the initial angular velocities of the joints are $\dot{\theta}_{1,0} = 0.0 \text{rad/s}$, $\dot{\theta}_{2,0} = 0.0 \text{rad/s}$. The size of the constant integration time step that is utilized by the RK4 explicit integrator is $h = 100 \text{msec}$, which is equal to the one used in DIRTRAN MPC. The prediction horizon is divided into $N = 10$ time steps. This means that the horizon length is equal to 1sec. The number of horizon time steps is kept the same to make a fair comparison with the DIRTRAN MPC. Fewer time steps, for instance $N = 3$, were also tested in experiments and provided satisfying results. The mean solve time was almost equal to that of DITRAN-MPC discussed previously.

The time responses of the angular positions, the angular velocities and the torques of the joints of the manipulator are presented in Figure 3-36, Figure 3-37 and Figure 3-38. The responses are considerably fast as well since the steady state is reached in just 5sec. This settling time is slightly higher than the previous ones. Also, not many oscillations are present in the responses. Finally, the hip actuator does reach the control effort saturation limits during the execution of the control law. This hip actuators control effort is saturated for approximately 0.1sec, without exceeding this limit. The duration of saturation is similar to the one in DIRTRAN MPC.

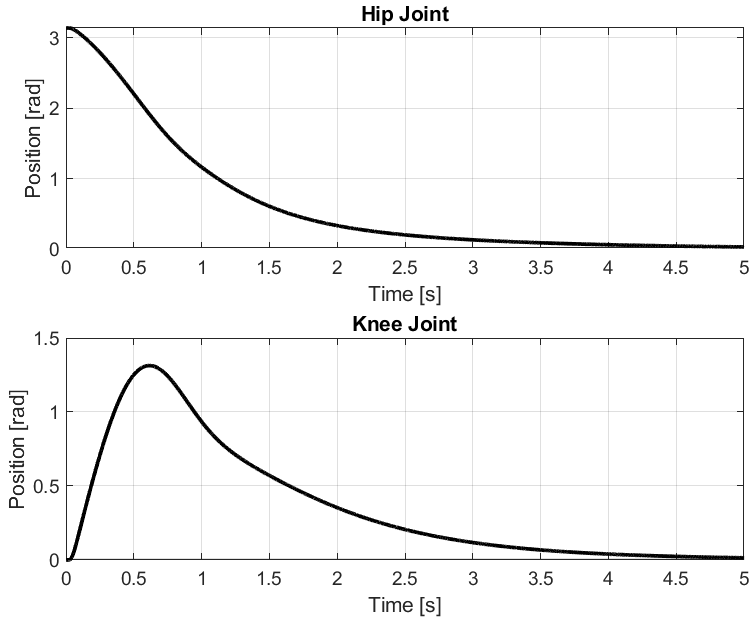


Figure 3-36. Time responses of the angular positions of the joints of the double pendulum.

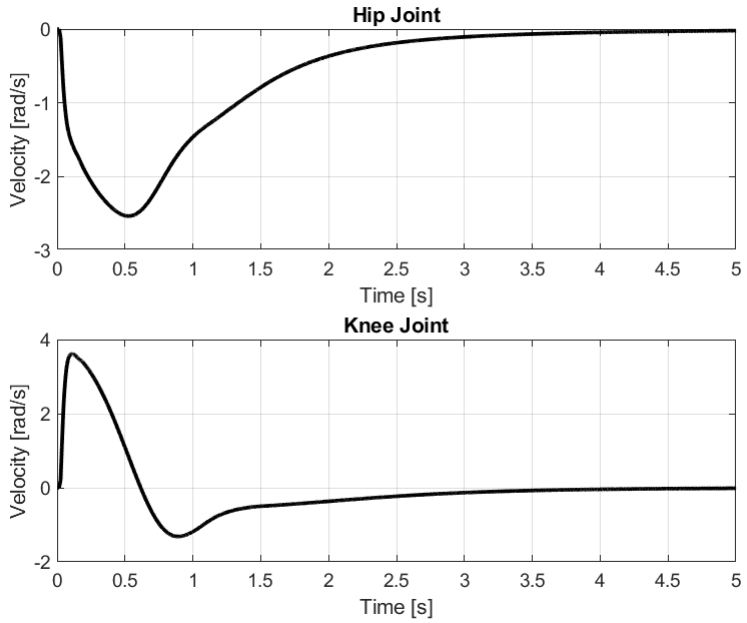


Figure 3-37. Time responses of the angular velocities of the joints of the double pendulum.

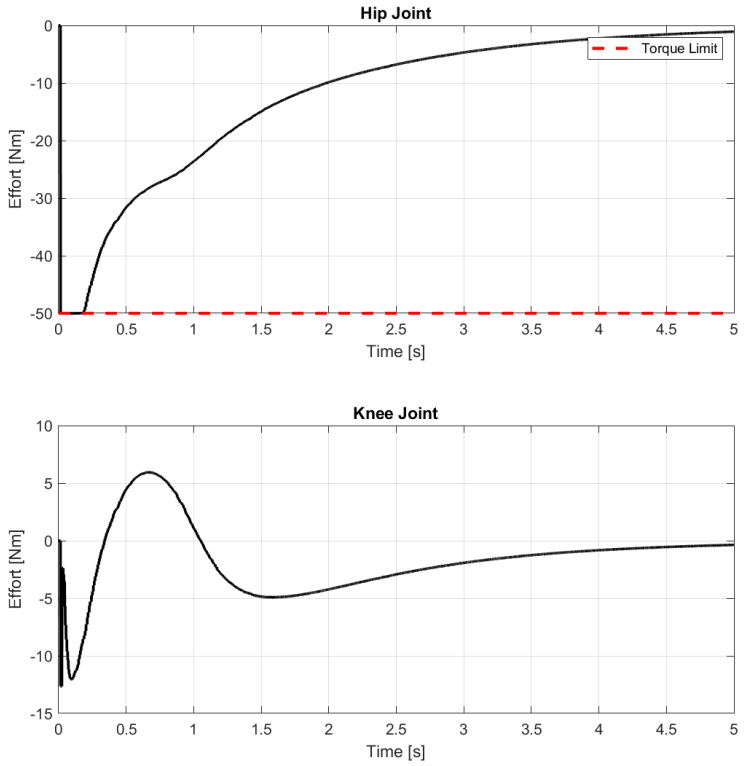


Figure 3-38. Time responses of the torques of the joints of the double pendulum.

The distribution of all DIRTRAN MPC solve time is presented in Figure 3-39. The mean value of solve time is equal to 4.3msec and the standard deviation of solve times is equal to 1.4msec. The mean solve time is approximately two orders of magnitude lower than the one of DIRTRAN MPC. Also, the solve times are almost equally dispersed around the mean value, as the DIRTRAN MPC ones. This is justified by the similar standard deviation of the solve

times that DIRTRAN and DIRCOL exhibit. Consequently, the DIRTAN MPC controller is faster than the DIRCOL DDP one, which was expected since fewer computations are performed in DIRTRAN while formulating the dynamical system constraints. Less computations are required for the computation of linear polynomials in comparison with cubic ones.

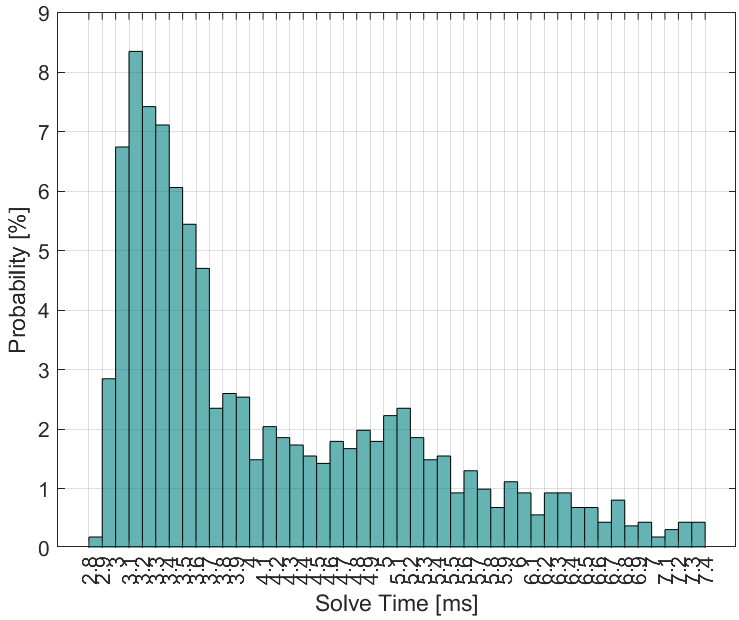


Figure 3-39. Solve time distribution of DIRCOL MPC.

3.7 Conclusion

The above results of the experiments conducted in Gazebo simulator allow to draw the following conclusions about the controllers:

- The Convex-MPC controller has the fastest convergence rate with solve times that exhibit small variation around the mean value. It also guarantees convergence to a global optimum. This attribute makes it ideal for online use on hardware. Also, it is numerically robust and can handle long horizons. Its primary downside is that it cannot handle nonlinear dynamics or other nonlinear constraints. They must be linearized to do so, in the case of QP-based MPC.
- The BOX-DDP MPC controller has the second fastest convergence rate with the second small variation of solve times. It can make use of nonlinear dynamics, even full-body dynamics, and is the only controller that produces dynamically feasible trajectories, even before convergence. However, it may converge to any arbitrary local optimum. Also, it cannot handle long horizons, due to numerical ill-conditioning. Finally, it cannot handle state constraints and input constraints that are not box constraints.
- The DIRTRAN/DIRCOL MPC controllers can handle nonlinear dynamics and any arbitrary state and control input constraints. Also, they are particularly numerically robust and can handle long horizons. However, they may converge to any arbitrary local optimum. Also, they are the slowest option with the largest variation of solve times around the mean value. Additionally, in the case of real time applications,

reduced-order dynamics are more common with such methods, due to their inherently slow convergence rate.

After taking all these conclusions into consideration, it becomes apparent that the Convex MPC controller is the most preferable choice for real time control applications and will be utilized for the control of ARGOS.

4 Optimal Control for Quadrupeds

4.1 Dynamic models in Optimal Control

The following dynamic models can be utilized in controllers of legged robots. These models are extensively analyzed in [104].

Rigid/Whole Body Dynamics (RBD/WBD)

The assumption made in this model, as shown in Figure 4-1, is the following:

- the robot is comprised of rigid bodies, that do not deform when forces/torques are applied to them.

The Rigid Body Dynamics of the robot can be expressed in the following compact matrix form, which represents the joint space dynamic model:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}}+\mathbf{c}(\mathbf{q},\dot{\mathbf{q}})+\mathbf{G}(\mathbf{q})=\mathbf{S}^T\boldsymbol{\tau}+\mathbf{J}^T(\mathbf{q})\mathbf{f} \tag{4-1}$$

where $\mathbf{q}=\begin{bmatrix}\mathbf{q_b}^T & \mathbf{q_j}^T\end{bmatrix}^T \in \mathbb{SE}(3)\times\mathbb{R}^{n_j}$ is the generalized coordinates vector, $\mathbf{q}_b \in \mathbb{SE}(3)$ is the floating base pose, $\mathbf{q_j}\in\mathbb{R}^{n_j}$ are the joints angles and n_j is the number of joints. Also, $\mathbf{M}\in\mathbb{R}^{(6+n_j)\times(6+n_j)}$ is the Joint Space Inertia matrix, $\mathbf{c}\in\mathbb{R}^{6+n_j}$ are the Coriolis and centripetal terms, $\mathbf{G}\in\mathbb{R}^{6+n_j}$ is the gravity term, $\mathbf{J}\in\mathbb{R}^{3n_j\times(6+n_j)}$ is the contact (geometric) Jacobian that that maps the external forces/torques to the generalized coordinate space, $\mathbf{f}=\begin{bmatrix}\mathbf{f}_1 & \dots & \mathbf{f}_{n_l}\end{bmatrix}^T\in\mathbb{R}^{3n_l}$ are the external forces/torques acting at the n_l end-effectors, $\mathbf{S}^T=\begin{bmatrix}\mathbf{0}_{n_j\times 6} & \mathbf{I}_{n_j\times n_j}\end{bmatrix}$ is the selection matrix that maps input forces/torques to joints and $\boldsymbol{\tau}\in\mathbb{R}^{n_j}$ are the input forces/torques.

The inputs to this system are the motor torques and the contact forces, but only motor torques are control inputs and thus decision variables of the optimization problem. This allows to consider torque limits during the optimization, but the contact forces have to be modeled as described in chapter 4.2.

However, full-body dynamic models for legged robots are very high-dimensional and complex. Also, TO problems (i.e., DDP) scale poorly (cubically) with the state dimension n, and cubically with the control input dimension m. The combined complexity of such TO problems is $O\left(N(n+m)^3\right)$ [152], where N is the optimization horizon. Thus, to make these problems more tractable, reduced-order dynamic models are used extensively in practice. In such models the state dimension is smaller. Reducing the state dimension by half would result in almost an order of magnitude reduction to the computational cost of the optimization problem. These reduced-order models are presented in this chapter.

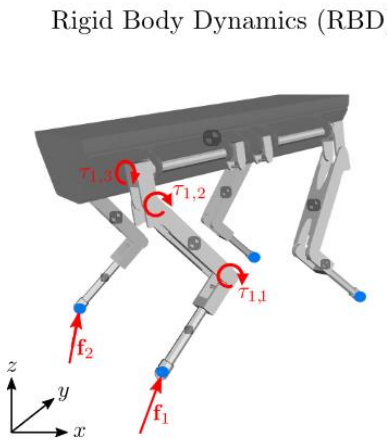


Figure 4-1. Whole Body Dynamics Model [104].

Single Rigid Body Dynamics (SRBD)

The assumptions made in this model, as shown in Figure 4-2, additionally to the one made in the RBD model, are the following:

- the momentum produced by the joint velocities is negligible.
- the full-body inertia remains similar to the one in nominal joint position.

Then, the SRBD of the robot, which are the Newton-Euler equations for a single rigid body, can be expressed in the following form:

$$\begin{aligned} m\ddot{\mathbf{p}} &= m\mathbf{g} + \sum_{i=1}^{n_l} \mathbf{f}_i \\ \frac{d}{dt}\left[\mathbf{I}(\boldsymbol{\theta})\boldsymbol{\omega}\right] &= \sum_{i=1}^{n_l} \mathbf{f}_i \times (\mathbf{p} - \mathbf{r}_i) \end{aligned} \tag{4-2}$$

where $\mathbf{p} \in \mathbb{R}^3$ is the CoM position, $\boldsymbol{\theta}$ is the base orientation, $\boldsymbol{\omega} \in \mathbb{R}^3$ is the angular velocity of the base, $\mathbf{g} \in \mathbb{R}^3$ is the acceleration of gravity, $m \in \mathbb{R}$ is the combined mass of the base and the limbs of the robot, $\mathbf{I}(\boldsymbol{\theta}) \in \mathbb{R}^{3\times3}$ is the combined inertia of the base and of all limbs of the robot in nominal joint configuration, anchored at the CoM and expressed in coordinate axis parallel to the inertial frame, $\mathbf{r}_i \in \mathbb{R}^3$ the position of the end-effector i where the ground reaction force (GRF) $\mathbf{f}_i \in \mathbb{R}^3$ acts.

The dynamics are independent of the joint angles and are expressed through Cartesian coordinates and rotations in 3-dimensional space. However, the cross-product term introduces nonlinearity in the model. The inputs to this system are vertical and tangential contact forces. This means that the variety of constraints that are imposed on the 3-dimensional forces at each foot (unilateral and friction constraints) can be directly enforced to them. This is not the case with models like the Linear Inverted Pendulum Model (LIPM) where the input to the system is the CoP position.

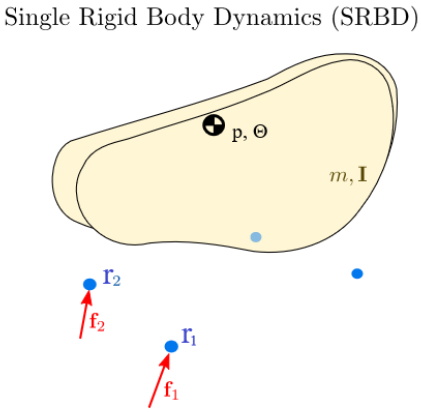


Figure 4-2. Single Rigid Body Dynamics Model [104].

Linear Inverted Pendulum Model (LIPM)

The assumptions made in this model, as shown in Figure 4-3, additionally to the ones made in the SRBD model, are the following:

- The CoM height p_z is constant.
- the base angular velocity $\boldsymbol{\omega}$ and angular acceleration $\dot{\boldsymbol{\omega}}$ are zero.
- the footholds height r_z is constant.

Then, the LIPM can be expressed in the following form:

$$\ddot{p}_x = \frac{g}{h}\left(p_x - r_{c,x}\right)$$
$$\ddot{p}_y = \frac{g}{h}\left(p_y - r_{c,y}\right) \tag{4-3}$$

where h is the walking height of the robot, g is the gravity acceleration in the direction of z axis and $r_{c,x}$, $r_{c,y}$ are the x and y position of the CoP.

This model describes how the position of the CoP linearly affects the horizontal CoM acceleration. The position of the CoP is manipulated through the vertical contact forces f_i^z and the position of the feet $r_{i,x}$. By summarizing the 3-dimensional forces into the CoP, critical information about them is lost and many characteristics of legged locomotion can't be modelled. Also, the LIPM assumptions restrict the range of motions that can be planned. More complex motions may require jumping or placing feet at different heights. Despite all that, it can be solved analytically and efficiently which makes it ideal for online motion planning.

The summary of these dynamic models, along with the corresponding state and control input variables needed to express these dynamics in state-space form, are presented in Table 4-1. It should be noted that only the position components are listed, although the state vector includes both position and velocity.

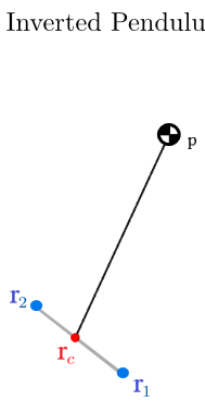


Figure 4-3. Linear Inverted Pendulum Model [104].

Table 4-1. State and input variables for state-space dynamic models for legged robots.

Dynamic Model	State variable	Input Variable
Rigid/Whole Body Dynamics (RBD/WBD)	$\mathbf{q_b}, \mathbf{q_j}$	$\boldsymbol{\tau}, \mathbf{f}_i$
Single Rigid Body Dynamics (SRBD)	$\mathbf{p}, \boldsymbol{\theta}, \mathbf{r}_i$	$\mathbf{f}_i$
Linear Inverted Pendulum Model (LIPM)	p_x, p_y	$r_{c,x}, r_{c,y}$

4.2 Contact Simulation in Robotics in Optimal Control Problems

Impacts and stiction cause velocity discontinuities, which result in infinite forces and accelerations. Thus modeling, simulation, control, and learning become more challenging tasks.

When WBD are utilized by the controller, contact forces have to be modeled explicitly. The following techniques are used to simulate contact in robotics, both for simulation and control. These models are thoroughly described in [153], and shown in Figure 4-4, Figure 4-5, Figure 4-6.

Soft/Compliant/Smooth contact model

In smooth contact models, unilateral constraints (the forces exerted by the robot to the ground, can only push into the ground, and not pull on it) are replaced by a compliant model of contact, like a nonlinear spring-damper model. This continuous force law tries to smooth out all the discontinuities that are present during impacts. A smooth contact model is implemented in MuJoCo simulator [154]. To be more precise, MuJoCo utilizes a smooth contact model that converts the contact problem to a convex optimization problem by relaxing the complementarity constraint between colliding objects (no friction cone approximation).

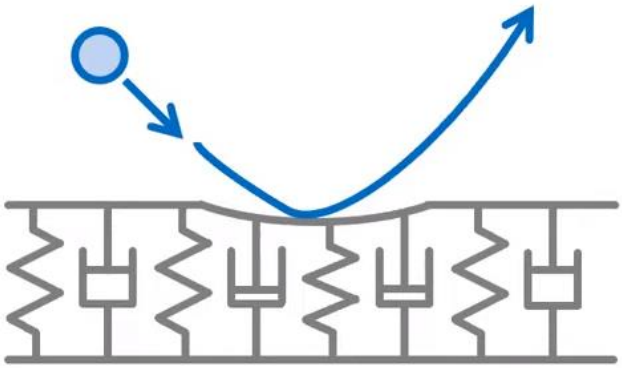


Figure 4-4. Smooth contact model [105].

In general, smooth contact models are easy models to implement and the resulting rigid body dynamics will be smooth, thus differentiable, and invertible. As a result, they are convenient for use in OCPs and RL.

The downside is that it can be hard to tune its parameters (stiffness, damping, etc). The collision outcome is highly sensitive to these values. If the model gets tuned to resemble a less compliant contact model, the ODEs become stiff, which must be avoided for real-time applications or for fast simulations. Additionally, these contact models tend to be inaccurate since they induce simulation artifacts. More specifically, non-zero contact forces may occur during non-contact phases (phantom forces), true stick-slip behavior cannot be captured due to creep and penetration always occurs between bodies and as a result only soft contacts can be modeled.

Hybrid/Event based (driven) model

In hybrid models, ODEs are integrated forward in time, using smooth integrators, while checking for contact events, then backtracking in time to find the exact moment the event occurs, by solving a root finding problem. An event finding algorithm for localizing the contact events is needed (also called guard or guard function) as well as a reset map (also called jump map) that is executed when the guard is met. The reset map is utilized to model the discontinuity. To be more precise, it re-initializes the state of the system exactly after the event (using restitution coefficient) and specifies the dynamics of the system that will be valid until the next event. Afterwards it proceeds with integrating the smooth dynamics until the next impact event occurs. This model is used extensively in controller design but not commonly used by simulators.

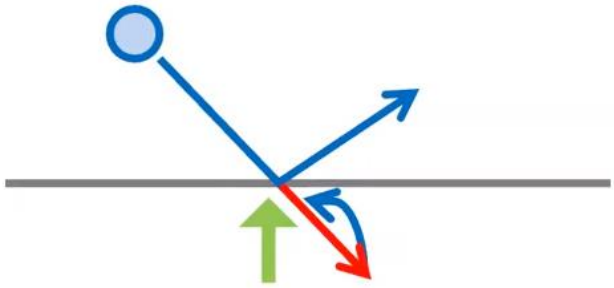


Figure 4-5. Hybrid/Event Driven methods [105].

In general, these contact models can make use of the standard, adaptive step size integrators that exist to integrate the smooth dynamics of the system that are existent between

the impact events. Moreover, they capture qualitatively correct contact dynamics since the aforementioned artifacts of compliant contact models are not present.

The downside is that they do not scale well with the number of potential contact configurations (modes) of a particular problem (combinatorial mode explosion). However, in the case of quadrupeds, where four legs with point contacts are existent, then the number of possible contact modes is finite, and this is not an issue. Consequently, these contact models are extremely popular for legged locomotion. Also, event driven methods suffer from Zeno, where an infinite number of events may occur in a finite amount of time, and consequently the integration must cease, and the reset map should be executed for every single event. Finally, these contact models are explicitly non-smooth, and thus are in general non-differentiable. However, techniques have been implemented to differentiate through these hybrid models, like the saltation matrix [52].

Time stepping model

In time stepping models [87], a time-discretization of non-smooth dynamics is used, including complementarity conditions and impact rules. Then a constrained optimization problem has to be solved at every time step. The solution to the problem is the contact forces, over the time step, that satisfy interpenetration constraints at the next time step. Called time stepping because of moving forward, in time, in discrete time steps.

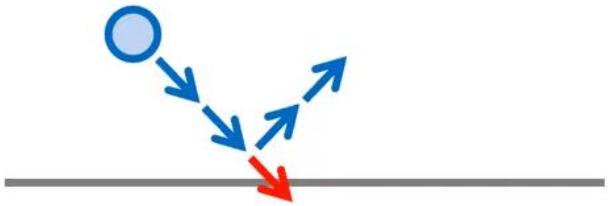


Figure 4-6. Time stepping model [105].

An advantage of these models is that they scale well (linearly) with the number of possible contact configurations of the system. They also do not suffer from issues like Zeno, since they integrate impulses over small time intervals at a time. Additionally, they capture qualitatively correct contact physics, depending on the internal approximations they make. For instance, since interpenetration constraint is enforced on the solution of the optimization, hard contacts can be modeled (objects do not sink into the ground). The approximations they make are pertinent to relaxation of constraints enforced in contact modeling, like complementarity constraints, Coulomb's friction cone constraint and maximum dissipation principle.

A downside is that first-order (Euler) integration methods are usually used in such models. Consequently, to be accurate enough, the time step has to be small (~kHz). Also, it is the most computationally expensive model. They are extensively used in many simulators like PyBullet, Dart, ODE, that solve an LCP, but not in TO. The reason is that the dynamics, that are a constraint to the TO problem, are another optimization problem by themselves which is embedded in the constraints of the TO problem (bilevel optimization [155], [156]). Frameworks that attempt to combine TO with such contact models exist like [157] and [158], but are tested on single legged hoppers and not on quadruped robots. Finally, these contact models are explicitly non-smooth, and thus are in general non-differentiable. However, techniques have been implemented to calculate derivatives.

4.3 Convex MPC for Quadrupeds

The control framework that was implemented in this research project consists of the following components: a state estimator, a gait scheduler, a swing leg controller, a stance leg controller, and a footstep planner. An overview of that control framework is presented in Figure 4-7. Each component will be thoroughly examined in the following sections.

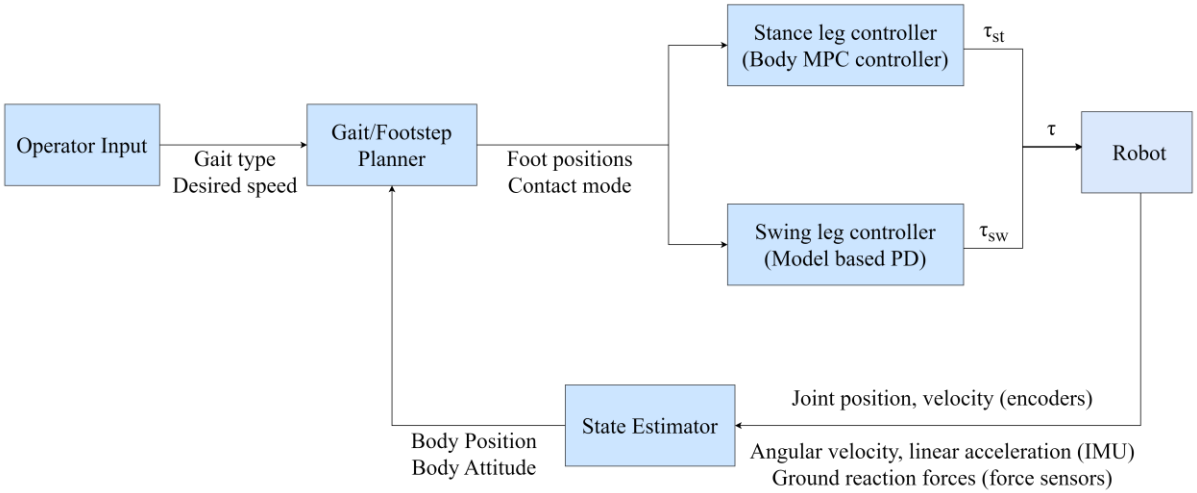


Figure 4-7. Control framework block diagram.

4.3.1 State Estimator

The state estimation for a quadruped is mainly performed using proprioceptive sensors. For instance, joint position encoders attached to each leg joint can be utilized to provide the joint angles measurement. Moreover, an estimate of the joint angular velocity can be acquired by numerically differentiating the joints angle measurements over the sensor sampling period.

Additionally, an inertia measurement unit (IMU) is usually attached to the body of the quadruped. This sensor measures the linear acceleration and the angular rate of the body of the robot. They are essential for any state estimation algorithm that can be utilized, like the Extended Kalman Filter (EKF) that is extremely common [159]. Integrating the linear CoM acceleration reading provided by the IMU provides an estimate of the linear CoM velocity. Moreover, this result can be combined with data provided from the leg velocity kinematics and thus a better estimate can be retrieved [40]. In similar fashion, an estimate of the CoM position can be obtained by fusing the estimated CoM velocity with data provided by the leg position kinematics [40]. However, instead of this fusion, the position and orientation of the robot's body can also be provided by a Motion Capture (MoCap) system. For outdoor applications MoCap is not a viable option and instead a vision-based algorithm (visual odometry) can be utilized instead for pose estimation. Finally, the robot's position and velocity can also be estimated using the proprioceptive odometry approach presented in [160]. In this approach IMU sensors are attached just above the quadruped's feet. The data provided by these IMUs as well as the IMU attached to the robot's body and the data provided by the joint encoders were fused using an EKF to estimate the robot's body and foot positions in the world frame.

Finally, the controller needs to be aware of whether the feet of the robot are in contact with the ground or not. This can be accomplished by mounting force sensors at the feet of the quadruped. One other alternative employed by Unitree robots [161] is to read the data provided by pressure sensors that are installed inside a cavity in the rubber feet of the robot.

Changes in pressure indicate contact events. Finally, the method that MIT Cheetah [162] utilizes is measuring the motor current (back-EMF voltage) and filtering it appropriately to detect contact events [163]. In the case of ARGOS in the Gazebo simulation environment, the aforementioned quantities were provided to the controller either from ROS sensors or directly from the physics engine. More information about the implementation in Gazebo will be presented in chapter 4.4.

4.3.2 Gait Scheduler

A period phase-based, open loop gait scheduler is utilized in this work similar to the one in [33]. This gait scheduler considers the gait timing to be fixed. In legged locomotion mode switches occur when a leg enters, or leaves contact with the ground. Consequently, during a locomotion task the state of a leg constantly switches between stance and swing state. The controller needs to be aware of what the state of each leg should be according to the gait executed. The main purpose of the gait scheduler is to specify this foot contact mode timing for a pre-defined gait.

This gait scheduler needs three parameters to specify a gait. These parameters are the relative phase offset of each leg $s_{offset,i}$, the stance period T_{st} and the gait cycle period T. Another important parameter that should be defined is the duty factor β that is defined as the quotient of the stance period T_{st} with the total gait period T.

$$\beta = \frac{T_{st}}{T} \tag{4-4}$$

It describes what fraction of the gait cycle is comprised of the stance. For periodic gaits, the duty factor is the same for all four legs. The phase scheduler defines a relative overall phase variable $s \in [0 \quad 1]$ that cycles over the entire gait cycle. This variable quantifies the percentile completion of the entire gait cycle. It is described by the following linear function of time:

$$s = \frac{t_{traj}}{T}, \quad s \in [0 \quad 1] \tag{4-5}$$

where

$$t_{traj} = \mathrm{mod}\left(t + s_{offset,i} \cdot T, T\right), \quad T = T_{st} + T_{sw} \tag{4-6}$$

is the current time progression of the gait cycle and T_{sw} is the swing period. By using the modulo operator, the time progression of the gait becomes invariant of the number of precedented gait cycles. If the phase variable s meets a particular threshold, then transition occurs between the leg states. This threshold is defined differently depending on the initial leg state. If a leg is in swing state at the beginning of the gait cycle and afterwards transitions in stance state, then this threshold is equal to $1-\beta$. However, if a leg is in stance state at the beginning of the gait cycle and afterwards transitions in swing state, then this threshold is equal to β. If the phase variable is smaller than this threshold then no state transition occurs, while if it is larger than the threshold a state transition occurs.

A phase offset $s_{offset,i}$ is defined for every leg at the initialization of the gait. For one of the legs the value of $s_{offset,i}$ is always equal to zero and its value for the remaining three legs is selected accordingly, depending on the gait. It must always lie in the interval $[0 \quad 1]$. The legs that are in stance/swing state simultaneously, have the same value assigned to the variable

$s_{offset,i}$. For example, in case of a trotting gait the legs that belong in the same diagonal pair will have the same value for $s_{offset,i}$. Consequently, the variable $s_{offset,i}$ needs to be included in the calculation of t_{traj} to account for the legs that have non-zero $s_{offset,i}$.

One other phase variable is also defined. This variable expresses the relative phase of a specific leg state (either stance or swing) $s_\varphi \in [0 \quad 1]$. Therefore, it reflects the leg's phase within the current swing or stance cycle. This variable cycles over the entire stance or swing period of a given leg. This variable quantifies the percentile completion of either the stance or swing state for each leg. It is described by the following linear function of time:

$$s_\varphi = \frac{t_j}{T_j}, \quad j \in \{st, sw\}, \quad s_\varphi \in [0 \quad 1] \tag{4-7}$$

where t_{st}, t_{sw} are the time durations that the leg spends in either stance or swing state if its current state is either stance or swing. The values of t_{st}, t_{sw} are computed as follows: if a leg is in stance state at the beginning of the gait cycle and afterwards transitions in swing state, then their values are the following:

$$t_{st} = t_{traj}, \quad t_{sw} = t_{traj} - T_{st} \tag{4-8}$$

On the other hand, if a leg is in swing state at the beginning of the gait cycle and afterwards transitions in stance state, then their values are the following:

$$t_{sw} = t_{traj}, \quad t_{st} = t_{traj} - T_{sw} \tag{4-9}$$

This normalized phase variable is utilized by the swing leg controller.

4.3.3 Swing Leg Controller

The swing leg controller swings the legs, that are in swing phase, from the current foothold position towards the next one. Given the desired foothold location that the leg is supposed to reach, the controller computes and tracks a swing foot trajectory. This trajectory is chosen to be an elliptical trajectory that has the following form:

$$\mathbf{r}(s_\varphi) = \mathbf{c} + \mathbf{u}\cos(\pi \cdot s_\varphi) + \mathbf{v}\sin(\pi \cdot s_\varphi), \quad s_\varphi \in [0 \quad 1] \tag{4-10}$$

where $\mathbf{c} \in \mathbb{R}^3$ is the center of the ellipse, $\mathbf{u}, \mathbf{v} \in \mathbb{R}^3$ are the vectors whose lengths are equal to the two ellipse half-axis lengths and $s_\varphi \in [0 \quad 1]$ represents a normalized phase variable. When the leg is in swing phase, this variable is equal to the quotient of the time that the leg is in swing phase t_{sw} with the swing phase period T_{sw}. The values of the vectors $\mathbf{c}, \mathbf{u}, \mathbf{v}$ can be computed given the foot's position at the beginning of the swing cycle $\mathbf{p}_{start} \in \mathbb{R}^3$, the foot's desired position at the end of the swing cycle $\mathbf{p}_{end} \in \mathbb{R}^3$ and the foot clearance on the ground at the middle of the swing phase $\delta_{\max}$ when the foot's position will be $\mathbf{p}_{mid} \in \mathbb{R}^3$.

The values of the x, y, z - coordinates of the points $\mathbf{p}_{start}, \mathbf{p}_{mid}, \mathbf{p}_{end}$ are the following:

$$\begin{aligned} \mathbf{p}_{start} &= \begin{bmatrix} \mathbf{p}_{start}^x & \mathbf{p}_{start}^y & \mathbf{p}_{start}^z \end{bmatrix}^T \\ \mathbf{p}_{end} &= \begin{bmatrix} \mathbf{p}_{end}^x & \mathbf{p}_{end}^y & \mathbf{p}_{end}^z \end{bmatrix}^T \\ \mathbf{p}_{mid} &= \begin{bmatrix} \mathbf{p}_{mid}^x & \mathbf{p}_{mid}^y & \mathbf{p}_{mid}^z \end{bmatrix}^T \end{aligned} \tag{4-11}$$

where in case of locomotion on flat ground the following will be true: $\mathbf{p}_{start}^z = \mathbf{p}_{end}^z = \mathbf{p}^z$. Consequently, the coordinates of the middle trajectory point will be the following:

$$\mathbf{p}_{mid}^{x} = \frac{1}{2}\left(\mathbf{p}_{start}^{x} + \mathbf{p}_{end}^{x}\right)$$
$$\mathbf{p}_{mid}^{y} = \frac{1}{2}\left(\mathbf{p}_{start}^{y} + \mathbf{p}_{end}^{y}\right) \tag{4-12}$$
$$\mathbf{p}_{mid}^{z} = \frac{1}{2}\left(\mathbf{p}_{start}^{z} + \mathbf{p}_{end}^{z}\right) + \delta_{\max} = \mathbf{p}^{z} + \delta_{\max}$$

Using the boundary conditions allows to extract the following expressions:

$$\begin{aligned} \mathbf{r}(0) = \mathbf{p}_{start} &\Rightarrow \mathbf{c} + \mathbf{u} = \mathbf{p}_{start} \\ \mathbf{r}(1) = \mathbf{p}_{end} &\Rightarrow \mathbf{c} - \mathbf{u} = \mathbf{p}_{end} \end{aligned} \tag{4-13}$$

The value of $\mathbf{c}$ can be acquired by combining these two equations. The result will be the following:

$$\mathbf{c} = \frac{1}{2}\left(\mathbf{p}_{start} + \mathbf{p}_{end}\right) = \begin{bmatrix} \frac{1}{2}\left(\mathbf{p}_{start}^{x} + \mathbf{p}_{end}^{x}\right) \\ \frac{1}{2}\left(\mathbf{p}_{start}^{y} + \mathbf{p}_{end}^{y}\right) \\ \mathbf{p}^{z} \end{bmatrix} \tag{4-14}$$

Given the $\mathbf{c}$ vector, the computation of $\mathbf{u}$ can be achieved using equation (4-13) which yields the following:

$$\mathbf{u} = \mathbf{p}_{start} - \mathbf{c} = \begin{bmatrix} \frac{1}{2}\left(\mathbf{p}_{start}^{x} - \mathbf{p}_{end}^{x}\right) \\ \frac{1}{2}\left(\mathbf{p}_{start}^{y} - \mathbf{p}_{end}^{y}\right) \\ 0 \end{bmatrix} \tag{4-15}$$

Computing the value of the spline at the middle point yields the following expression:

$$\mathbf{r}\left(\frac{1}{2}\right) = \mathbf{p}_{mid} \Rightarrow \mathbf{c} + \mathbf{v} = \mathbf{p}_{mid} \tag{4-16}$$

Given the $\mathbf{c}$ vector, the computation of $\mathbf{v}$ can be achieved using equation (4-16) which yields the following:

$$\mathbf{v} = \mathbf{p}_{mid} - \mathbf{c} = \begin{bmatrix} 0 \\ 0 \\ \delta_{\max} \end{bmatrix} \tag{4-17}$$

Consequently, the elliptical primitive along which the toe is supposed to move can be constructed given the vectors $\mathbf{c}, \mathbf{u}, \mathbf{v}$. This elliptical primitive should always lie inside the workspace of each leg. A point along this elliptical primitive is computed for a specified phase variable value.

However, the control action is finally implemented in the joint space. Therefore, an inverse kinematics algorithm is executed to convert the desired foot position into desired joint angles. The control law utilized by the swing leg controller to compute joint torques for the ith leg

$\boldsymbol{\tau}_{sw,i} \in \mathbb{R}^3$ consists of a feed-forward $\boldsymbol{\tau}_{sw,i}^{ff} \in \mathbb{R}^3$ and a feedback term $\boldsymbol{\tau}_{sw,i}^{fb} \in \mathbb{R}^3$ of the following form:

$$\boldsymbol{\tau}_{sw,i} = \boldsymbol{\tau}_{sw,i}^{ff} + \boldsymbol{\tau}_{sw,i}^{fb} \tag{4-18}$$

The feedforward term is based on the system dynamics and is given by the following expression:

$$\boldsymbol{\tau}_{sw,i}^{ff} = \mathbf{M}_i(\mathbf{q}_i)\mathbf{J}_i^{-1}(\mathbf{q}_i)\left[\ddot{\mathbf{r}}_i - \dot{\mathbf{J}}_i(\mathbf{q}_i)\dot{\mathbf{q}}_i\right] + \mathbf{c}_i(\mathbf{q}_i, \dot{\mathbf{q}}_i) + \mathbf{G}_i(\mathbf{q}_i) \tag{4-19}$$

where $\mathbf{q}_i, \dot{\mathbf{q}}_i \in \mathbb{R}^3$ are the joint angles and angular velocities of the ith leg, $\mathbf{M}_i \in \mathbb{R}^{3\times3}$ is the joint space inertia matrix of ith leg, $\mathbf{J}_i, \dot{\mathbf{J}}_i \in \mathbb{R}^{3\times3}$ are the ith leg Jacobian matrix and its time derivative respectively, $\mathbf{c}_i, \mathbf{G}_i \in \mathbb{R}^3$ are the Coriolis and centripetal and gravity terms of the ith leg, respectively, and $\ddot{\mathbf{r}}_i$ is the reference workspace acceleration of the foot expressed in the body frame. During the locomotion tasks that were executed, the gravity terms had the highest contribution to the resulting torque and consequently could not be neglected. Also, unlike in the case of the stance leg controller, the leg inertia was not neglected. Its contribution is taken into consideration since it is included in the Inertia matrix despite the fact that it is insignificant compared to the inertia of the robot's body. Finally, the feedback term is a PD controller described by the following expression:

$$\boldsymbol{\tau}_{sw,i}^{fb} = \mathbf{K}_\mathbf{p}\left(\mathbf{q}_i^{cmd} - \mathbf{q}_i\right) + \mathbf{K}_\mathbf{d}\left(\dot{\mathbf{q}}_i^{cmd} - \dot{\mathbf{q}}_i\right) \tag{4-20}$$

where $\mathbf{K}_\mathbf{p}, \mathbf{K}_\mathbf{d} \in \mathbb{R}^{3\times3}$ are the diagonal positive definite proportional and derivative gain matrices and $\mathbf{q}_i^{cmd}, \dot{\mathbf{q}}_i^{cmd} \in \mathbb{R}^3$ are the commanded joint angles and angular velocities that are defined by the spline presented previously. A preferable alternative to the PD controller is a PV controller that has the following form:

$$\boldsymbol{\tau}_{sw,i}^{fb} = \mathbf{K}_\mathbf{p}\left(\mathbf{q}_i^{cmd} - \mathbf{q}_i\right) - \mathbf{K}_\mathbf{d}\dot{\mathbf{q}}_i \tag{4-21}$$

The main advantage of PV control over PD is that PV control does not add any zero terms to the closed loop system, which results in transient responses with fewer oscillations.

The desired footstep locations, needed for the primitive design, are determined by the footstep planner. If an early contact event occurs for a particular leg, then it will cease to be controlled by the swing leg controller and the stance leg controller will take over.

4.3.4 Stance Leg Controller

The stance leg controller treats the entire robot as a single lumped mass rigid body and given the desired foothold positions, computes the GRFs of the feet that are in stance state. Therefore, the SRBD model is utilized in the stance leg controller design. The tradeoff between accuracy and computational cost is more favorable with this reduced-order model since it is more accurate than the LIPM and computationally less expensive than the RBD model. Another advantage of this model is that the GRFs are decision variables of the optimization problem and not the motor torques. Therefore, the GRFs will not have to be modeled explicitly using the contact models of the chapter 4.2. Additionally, the approximation of the WBD with the SRBD model is valid for a quadruped like ARGOS. It was utilized in the MIT Cheetah 3 robot [162], whose design is similar to ARGOS. To be more precise, in such robots the mass/inertia of all four legs is approximately equal to only the 10% of the total mass/inertia of the robot. Consequently, even fast leg motion does not contribute significantly to the total

momentum of the system and the leg position does not significantly alter the full-body inertia. Also, the leg actuators can move the swing legs very fast, compared to the body motion.

SRBD Linearization and Discretization

The SRBD can also be written in the following form:

$$\begin{aligned} m\ddot{\mathbf{p}} &= m\mathbf{g} + \sum_{i=1}^{n_l} \mathbf{f}_i \\ \mathbf{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times (\mathbf{I}\boldsymbol{\omega}) &= \sum_{i=1}^{n_l} \mathbf{r}_i \times \mathbf{f}_i \end{aligned} \tag{4-22}$$

where $\mathbf{I} = \mathbf{I}(\boldsymbol{\theta})$ is the inertia tensor of the robot expressed in the world frame and $\mathbf{r}_i \in \mathbb{R}^3$ is the position of the end-effector i w.r.t. the CoM of the robot. The CoM position $\mathbf{p}$, the acceleration of gravity $\mathbf{g} = \begin{bmatrix} 0 & 0 & g \end{bmatrix} \in \mathbb{R}^3$ and the GRF of the ith leg $\mathbf{f}_i \in \mathbb{R}^3$ are written w.r.t. the world frame. The body frame $\mathcal{B}$ and the inertia (or world) frame $\mathcal{I}$ are defined in Figure 4-8. Quantities expressed in the body coordinate system have a left subscript $\mathcal{B}$ while quantities expressed in the world coordinate system have no subscript.

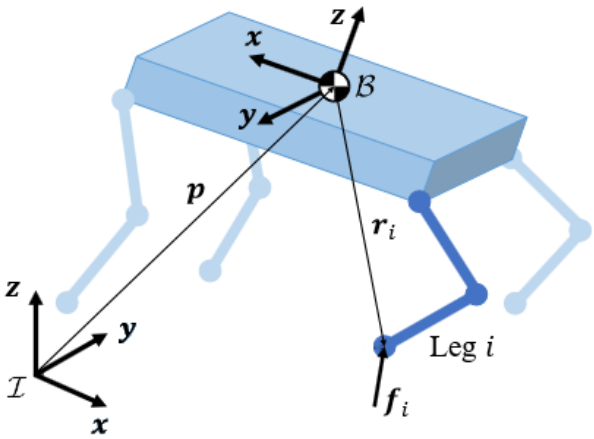


Figure 4-8. State and input vectors and coordinate systems for SRBD model [41].

However, in Convex MPC, the system dynamics must be linear time-varying (LTV) or linear time-invariant (LTI). The single lumped mass rigid body model is not linear w.r.t. the state and input variables, due to the cross-product term in the GRF moments, the quadratic cross product term in the attitude dynamics and the nonlinear rotation dynamics. Linear dynamics can be obtained if some additional approximations are made.

The orientation of the body frame $\mathcal{B}$ w.r.t. the inertia frame $\mathcal{I}$ is expressed as a vector of Z-Y-X Euler angles $\boldsymbol{\theta} = \begin{bmatrix} \varphi & \theta & \psi \end{bmatrix}^T \in \mathbb{R}^3$ where φ is the roll angle, θ is the pitch angle and ψ is the yaw angle. These angles correspond to a sequence of rotations such that the transform from the body frame to the world frame $\mathbf{R} \in \mathbb{SO}(3)$ can be expressed as:

$$\mathbf{R} = \mathbf{R}_z(\psi)\mathbf{R}_y(\theta)\mathbf{R}_x(\varphi) \tag{4-23}$$

The angular velocity in world coordinates $\boldsymbol{\omega}$ can be found from the rate of change of these angles $\dot{\boldsymbol{\theta}} = \begin{bmatrix} \dot{\varphi} & \dot{\theta} & \dot{\psi} \end{bmatrix}^T \in \mathbb{R}^3$ according to the following expression:

$$\boldsymbol{\omega} = \begin{bmatrix} \cos(\theta)\cos(\psi) & -\sin(\psi) & 0 \\ \cos(\theta)\sin(\psi) & \cos(\psi) & 0 \\ -\sin(\theta) & 0 & 1 \end{bmatrix} \dot{\boldsymbol{\theta}} \tag{4-24}$$

If the robot is not pointed vertically w.r.t the world frame, meaning that $\cos(\theta) \neq 0$, equation (4-24) can be inverted yielding the following one:

$$\dot{\boldsymbol{\theta}} = \begin{bmatrix} \cos(\psi)/\cos(\theta) & \sin(\psi)/\cos(\theta) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ \cos(\psi)\tan(\theta) & \sin(\psi)\tan(\theta) & 1 \end{bmatrix} \boldsymbol{\omega} \tag{4-25}$$

The first assumption is that the body roll and pitch angles are small. Equivalently, linearization about the roll and pitch axis is performed and not about the yaw axis. This assumption is reasonable for walking motions on relatively flat terrain. Given this assumption, equation (4-25) can be approximated as:

$$\dot{\boldsymbol{\theta}} \approx \begin{bmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \boldsymbol{\omega} = \mathbf{R}_z{}^T(\psi)\boldsymbol{\omega} \tag{4-26}$$

Also, the inertia tensor of the robot in the world frame can be computed as follows:

$$\mathbf{I} = \mathbf{R}_{\mathcal{B}}\mathbf{I}\mathbf{R}^T \tag{4-27}$$

where ${}_{\mathcal{B}}\mathbf{I}$ is the inertia tensor of the robot expressed in the body frame. Using the first assumption, equation (4-27) can be approximated as:

$$\mathbf{I} = \mathbf{R}_z(\psi)\,{}_{\mathcal{B}}\mathbf{I}\mathbf{R}_z{}^T(\psi) \tag{4-28}$$

The second assumption is that the angular velocities of the body are small. Given this assumption the following approximation can be made:

$$\frac{d}{dt}(\mathbf{I}\boldsymbol{\omega}) = \mathbf{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega}\times(\mathbf{I}\boldsymbol{\omega}) \approx \mathbf{I}\dot{\boldsymbol{\omega}} \tag{4-29}$$

where the term $\boldsymbol{\omega}\times(\mathbf{I}\boldsymbol{\omega})$ has been neglected from equation (4-29) since it is small, if the angular velocities of the body are small, and its contribution to the dynamics of the robot is insignificant. That term expresses the effect of precession and nutation of the rotating body of the robot. This approximation is even more valid if the off-diagonal terms of the inertia tensor are also small.

Given these two assumptions the SRBD of equation (4-22) can be written in the following matrix form:

$$\frac{d}{dt}\begin{bmatrix} \boldsymbol{\theta} \\ \mathbf{p} \\ \boldsymbol{\omega} \\ \dot{\mathbf{p}} \end{bmatrix} = \begin{bmatrix} \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{R}_z{}^T(\psi) & \mathbf{0}_3 \\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{I}_3 \\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 \end{bmatrix} \begin{bmatrix} \boldsymbol{\theta} \\ \mathbf{p} \\ \boldsymbol{\omega} \\ \dot{\mathbf{p}} \end{bmatrix} + \begin{bmatrix} \mathbf{0}_3 & \dots & \mathbf{0}_3 \\ \mathbf{0}_3 & \dots & \mathbf{0}_3 \\ \mathbf{I}^{-1}[\mathbf{r}_1]_\times & \dots & \mathbf{I}^{-1}[\mathbf{r}_{n_l}]_\times \\ \mathbf{I}_3/m & \dots & \mathbf{I}_3/m \end{bmatrix} \begin{bmatrix} \mathbf{f}_1 \\ \vdots \\ \mathbf{f}_{n_l} \end{bmatrix} + \begin{bmatrix} \mathbf{0}_{3\times1} \\ \mathbf{0}_{3\times1} \\ \mathbf{0}_{3\times1} \\ \mathbf{g} \end{bmatrix} \tag{4-30}$$

where $\mathbf{I}_3$ is the 3×3 identity matrix, $\mathbf{0}_3$ is the 3×3 zero matrix, $\mathbf{0}_{3\times1}$ is the 3×1 zero vector and $[\mathbf{r}_i]_\times \in \mathbb{SO}(3)$ is defined as the skew symmetric matrix such that $[\mathbf{r}_i]_\times \mathbf{b} = \mathbf{r}_i \times \mathbf{b}$, $\forall \mathbf{b} \in \mathbb{R}^3$. However, the state and input matrices of the system of this form are dependent on the yaw

angle ψ and the footstep locations $\mathbf{r}_i$. If these quantities are predetermined and computed ahead of time, then the dynamics of the equation (4-30) can be converted into LTV dynamics. Consequently, the third and final assumption necessary, to complete the linearization of SRBD, is that the actual yaw angle ψ and footstep locations $\mathbf{r}_i$ are close to their commanded values ψ^{cmd}, $\mathbf{r}_i^{cmd}$ and therefore the reference is tracked almost perfectly. Now, the state and input matrices of the system are only time varying.

Finally, the constant term of equation (4-30) $\begin{bmatrix}\mathbf{0}_{3\times1} & \mathbf{0}_{3\times1} & \mathbf{0}_{3\times1} & \mathbf{g}\end{bmatrix}^T$, has to be eliminated to achieve linear dynamics. This elimination is achieved by adding an additional gravity state to the dynamics. With the new state variable vector $\mathbf{x}(t)=\begin{bmatrix}\boldsymbol{\theta} & \mathbf{p} & \boldsymbol{\omega} & \dot{\mathbf{p}} & g\end{bmatrix}^T \in \mathbb{R}^{13}$ and the new input variable vector $\mathbf{u}(t)=\begin{bmatrix}\mathbf{f}_1 & \dots & \mathbf{f}_{n_l}\end{bmatrix}^T \in \mathbb{R}^{3n_l}$, the system continuous time dynamics can be rewritten in the following form:

$$\dot{\mathbf{x}}(t)=\mathbf{A}(t)\mathbf{x}(t)+\mathbf{B}(t)\mathbf{u}(t) \tag{4-31}$$

where

$$\mathbf{A}(t)=\begin{bmatrix}\mathbf{0}_3 & \mathbf{0}_3 & \mathbf{R}_z{}^T\left(\psi^{cmd}\right) & \mathbf{0}_3 & \mathbf{0}_{3\times1}\\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{I}_3 & \mathbf{0}_{3\times1}\\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_{3\times1}\\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \begin{pmatrix}0 & 0 & 1\end{pmatrix}^T\\ \mathbf{0}_{1\times3} & \mathbf{0}_{1\times3} & \mathbf{0}_{1\times3} & \mathbf{0}_{1\times3} & 0\end{bmatrix} \in \mathbb{R}^{13\times13} \tag{4-32}$$

and

$$\mathbf{B}(t)=\begin{bmatrix}\mathbf{0}_3 & \dots & \mathbf{0}_3\\ \mathbf{0}_3 & \dots & \mathbf{0}_3\\ \mathbf{I}^{-1}\left[\mathbf{r}_1^{cmd}\right]_\times & \dots & \mathbf{I}^{-1}\left[\mathbf{r}_{n_l}{}^{cmd}\right]_\times\\ \mathbf{I}_3/m & \dots & \mathbf{I}_3/m\\ \mathbf{0}_{1\times3} & \dots & \mathbf{0}_{1\times3}\end{bmatrix} \in \mathbb{R}^{13\times3n_l} \tag{4-33}$$

with $\mathbf{0}_{1\times3}$ being equal to the 1×3 zero vector. Ergo, for this system, the number of the state variables is $n=13$, and the number of the input variables is $m=3n_l$. Substituting $\mathbf{x}(t),\mathbf{u}(t),\mathbf{A}(t),\mathbf{B}(t)$ in (4-31) yields the following expression:

$$\frac{d}{dt}\begin{bmatrix}\boldsymbol{\theta}\\ \mathbf{p}\\ \boldsymbol{\omega}\\ \dot{\mathbf{p}}\\ g\end{bmatrix}=\begin{bmatrix}\mathbf{0}_3 & \mathbf{0}_3 & \mathbf{R}_z{}^T\left(\psi^{cmd}\right) & \mathbf{0}_3 & \mathbf{0}_{3\times1}\\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{I}_3 & \mathbf{0}_{3\times1}\\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_{3\times1}\\ \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \begin{pmatrix}0 & 0 & 1\end{pmatrix}^T\\ \mathbf{0}_{1\times3} & \mathbf{0}_{1\times3} & \mathbf{0}_{1\times3} & \mathbf{0}_{1\times3} & 0\end{bmatrix}\begin{bmatrix}\boldsymbol{\theta}\\ \mathbf{p}\\ \boldsymbol{\omega}\\ \dot{\mathbf{p}}\\ g\end{bmatrix}+\begin{bmatrix}\mathbf{0}_3 & \dots & \mathbf{0}_3\\ \mathbf{0}_3 & \dots & \mathbf{0}_3\\ \mathbf{I}^{-1}\left[\mathbf{r}_1^{cmd}\right]_\times & \dots & \mathbf{I}^{-1}\left[\mathbf{r}_{n_l}{}^{cmd}\right]_\times\\ \mathbf{I}_3/m & \dots & \mathbf{I}_3/m\\ \mathbf{0}_{1\times3} & \dots & \mathbf{0}_{1\times3}\end{bmatrix}\begin{bmatrix}\mathbf{f}_1\\ \vdots\\ \mathbf{f}_{n_l}\end{bmatrix} \tag{4-34}$$

The addition of this gravity state results in a LTV system that is no longer controllable but is still stabilizable. Finally, the continuous time dynamics of have to be discretized and written in the form $\mathbf{x}_{k+1}=\mathbf{A}_k\mathbf{x}_k+\mathbf{B}_k\mathbf{u}_k$. For simplicity, the discretization is conducted using an explicit first-order Euler integration integrator with zero-order hold on the inputs is applied such that:

$$\mathbf{A}_k=\mathbf{I}_n+\mathbf{A}(t_k)h, \quad \mathbf{B}_k=\mathbf{B}(t_k)h \tag{4-35}$$

where h is the constant time step. By choosing Euler integration the accuracy of the integration gets reduced but also the computational cost of the integration gets reduced. This loss of accuracy is not of paramount importance since the model utilized in the MPC controller is already an approximation of WBD and not a high-fidelity model [31].

Cost Function

The cost function that is used is quadratic, since Convex MPC is being implemented. This objective, in discrete time, is given by the following expression:

$$\begin{aligned} J = J(\mathbf{X},\mathbf{U}) = \sum_{k=0}^{N-1} \left(\mathbf{x}_k - \mathbf{x}_{\mathbf{ref}_k}\right)^T \mathbf{Q}_k \left(\mathbf{x}_k - \mathbf{x}_{\mathbf{ref}_k}\right) + \mathbf{u}_k{}^T \mathbf{R}_k \mathbf{u}_k \\ + \left(\mathbf{x}_N - \mathbf{x}_{\mathbf{ref}_N}\right)^T \mathbf{Q}_N \left(\mathbf{x}_N - \mathbf{x}_{\mathbf{ref}_N}\right), \quad \mathbf{Q}_k \geq 0, \quad \mathbf{R}_k > 0 \end{aligned} \tag{4-36}$$

The reference trajectory $\mathbf{x}_{\mathbf{ref}_k}$ is selected as follows: it contains only non-zero x,y-velocity, x,y,z-position, yaw angle ψ, and yaw rate $\dot{\psi}$. The robot operator selects ahead of time the desired x,y-velocity and yaw rate $\dot{\psi}$. The desired x,y-position and yaw angle ψ are determined by integrating the x,y-velocity and yaw rate $\dot{\psi}$, respectively. The remaining states, which are the desired roll angle φ, pitch angle θ, roll rate $\dot{\varphi}$, pitch rate $\dot{\theta}$ and z-velocity, are always equal to zero. This reference trajectory can also be utilized for determining the commanded yaw angle ψ_k and footstep locations $\mathbf{r}_{i,k}$, that are necessary for the computation of the matrices $\mathbf{A}_k, \mathbf{B}_k$ for every time step of the horizon.

Force Constraints

Also, constraints have to be imposed on the GRFs $\mathbf{f}_{i,k}$, for the ith leg, at the kth horizon time step, to ensure that GRFs computed by the solver are physically feasible. When the ith leg is in stance state, the normal component of the GRF acting on the leg $\mathbf{f}_{i,k}^z$, where the z axis is aligned with the normal vector of the ground, should be limited by an upper bound $\mathbf{f}_{i,\max}^z$ and a lower bound $\mathbf{f}_{i,\min}^z$, so that:

$$\mathbf{f}_{i,\min}^z \leq \mathbf{f}_{i,k}^z \leq \mathbf{f}_{i,\max}^z, \quad \mathbf{f}_{i,\min}^z \geq 0 \tag{4-37}$$

This inequality constraint is linear (box constraint) and consequently fits in the requirements of a QP. The minimum value of the normal component of GRF $\mathbf{f}_{i,\min}^z$ should be non-negative (i.e., $\mathbf{f}_{i,\min}^z \geq 0$). It is physically feasible for the legs in stance state to only create contact forces that push into the ground and not pull on it. This constraint is also known as the unilateral constraint and the solution of the QP will obey it only if the lower bound of equation (4-37) is nonnegative. The maximum value of the normal component of GRF $\mathbf{f}_{i,\max}^z$ is needed to guarantee that the commanded torque does not exceed the actuation limits of the joint actuators. While the motor torques are dependent not only on the GRFs but also on the leg configurations, a sensible $\mathbf{f}_{i,\max}^z$ value can be calculated using the nominal leg configurations and the actuator torque limits. In general, the joint angles of the legs during a locomotion task are close to the joint angles in the nominal leg configuration. Consequently, this approximation will be valid for such tasks.

Also, when the ith leg is in stance state, the tangent component of the GRF acting on the leg should lie within the Coulomb friction cone. This constraint is described by the following relationship:

$$\forall \mathbf{f}_{i,k} \in \mathbb{R}^3 \quad \mathbf{f}_{i,k}^n \geq 0, \; \left\| \mathbf{f}_{i,k}^t \right\|_2 \leq \mu \left| \mathbf{f}_{i,k}^n \right| \tag{4-38}$$

where μ is the friction coefficient, $\mathbf{f}_{i,k}^{t}$ denotes the tangential GRF component, $\mathbf{f}_{i,k}^{n}$ denotes the tangential GRF component. However, this inequality constraint is a second-order cone constraint and consequently does not fit in the requirements of a QP. It has to be linearized to meet the criteria for a QP. The linear approximation of the friction cone is a square pyramid [164].Using this approximation, the constraint of equation (4-38) can be written as follows:

$$\forall \mathbf{f}_{i,k} \in \mathbb{R}^3 \quad \left|\mathbf{f}_{i,k}^{x}\right| \le \mu \mathbf{f}_{i,k}^{z}, \quad \left|\mathbf{f}_{i,k}^{y}\right| \le \mu \mathbf{f}_{i,k}^{z}, \quad \mathbf{f}_{i,k}^{z} \ge 0 \tag{4-39}$$

where the z axis is aligned with the normal vector of the ground and x, y are two axes orthogonal to each other that lie in the tangent plane at the contact point. Combining constraints (4-37) and (4-39) yields the following constraints that are enforced on the ith leg when it is in stance state:

$$\begin{aligned} \mathbf{f}_{i,\min}^{z} \le \mathbf{f}_{i,k}^{z} &\le \mathbf{f}_{i,\max}^{z} \\ -\mu \mathbf{f}_{i,k}^{z} \le \mathbf{f}_{i,k}^{x} &\le \mu \mathbf{f}_{i,k}^{z} \\ -\mu \mathbf{f}_{i,k}^{z} \le \mathbf{f}_{i,k}^{y} &\le \mu \mathbf{f}_{i,k}^{z} \end{aligned} \tag{4-40}$$

All the inequality constraints of equation (4-40) can be encoded using a single inequality constraint. For a single leg i, at the horizon time step k, this inequality constraint has the following form:

$$\underline{\mathbf{c}}_{i,k} \le \mathbf{C}_{i,k}\mathbf{u}_{i,k} \le \overline{\mathbf{c}}_{i,k}, \quad \forall i = \{1,\ldots,n_l\}, \forall k = \{0,\ldots,N-1\} \tag{4-41}$$

where the constraint matrix $\mathbf{C}_{i,k} \in \mathbb{R}^{5\times 3}$ and constraint bounds $\underline{\mathbf{c}}_{i,k}, \overline{\mathbf{c}}_{i,k} \in \mathbb{R}^{5\times 1}$ are given by the following expressions:

$$\mathbf{C}_{i,k} = \begin{bmatrix} -1 & 0 & \mu \\ 1 & 0 & \mu \\ 0 & -1 & \mu \\ 0 & 1 & \mu \\ 0 & 0 & 1 \end{bmatrix}, \quad \underline{\mathbf{c}}_{i,k} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \mathbf{f}_{i,\min}^{z} \cdot c_{flag_{i,k}} \end{bmatrix}, \quad \overline{\mathbf{c}}_{i,k} = \begin{bmatrix} +\infty \\ +\infty \\ +\infty \\ +\infty \\ \mathbf{f}_{i,\max}^{z} \cdot c_{flag_{i,k}} \end{bmatrix} \tag{4-42}$$

where $c_{flag_{i,k}}$ is a flag that is equal to zero, if the ith leg, at the kth horizon time step is not in contact with the ground, or equal to one, if the ith leg, at the kth horizon time step is in contact with the ground. For all four legs, at the horizon step k, the inequality constraint has the following form:

$$\underline{\mathbf{c}}_{k} \le \mathbf{C}_{k}\mathbf{u}_{k} \le \overline{\mathbf{c}}_{k}, \quad \forall k = \{0,\ldots,N-1\} \tag{4-43}$$

where the constraint matrix $\mathbf{C}_{k} \in \mathbb{R}^{5n_l \times 3n_l}$ and constraint bounds $\underline{\mathbf{c}}_{k}, \overline{\mathbf{c}}_{k} \in \mathbb{R}^{5n_l \times 1}$ are given by the following expressions:

$$\mathbf{C}_{k} = \begin{bmatrix} \mathbf{C}_{1,k} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{C}_{2,k} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{C}_{n_l,k} \end{bmatrix}, \quad \underline{\mathbf{c}}_{k} = \begin{bmatrix} \underline{\mathbf{c}}_{1,k} \\ \underline{\mathbf{c}}_{2,k} \\ \vdots \\ \underline{\mathbf{c}}_{n_l,k} \end{bmatrix}, \quad \overline{\mathbf{c}}_{k} = \begin{bmatrix} \overline{\mathbf{c}}_{1,k} \\ \overline{\mathbf{c}}_{2,k} \\ \vdots \\ \overline{\mathbf{c}}_{n_l,k} \end{bmatrix} \tag{4-44}$$

When the ith leg is in swing phase, the GRF must be equal to zero since the foot of the leg does not interact with the ground. Ergo, these GRFs must obey the following relationship:

$$\mathbf{f}_{i,k} = \mathbf{0} \tag{4-45}$$

The same result can be achieved if the upper and lower bounds $\mathbf{f}_{i,\max}^{z}, \mathbf{f}_{i,\min}^{z}$ of constraint equation (4-40) are set equal to zero. These legs are handled by the swing leg controller which guides their feet towards the next foothold position. However, as it was also mentioned before, the stance leg controller assumes that the swing leg controller will track the reference almost perfectly and land the foot almost exactly at the pre-specified foothold position.

QP Formulation

After the dynamics, the cost function and the constraints have been formulated, the Convex MPC problem can be arranged and has the form:

$$\begin{aligned}
\min_{\mathbf{x}_{0:N},\mathbf{u}_{0:N-1}} \quad & J = \sum_{k=0}^{N-1} \left(\mathbf{x}_k - \mathbf{x}_{\mathbf{ref}_k}\right)^T \mathbf{Q}_k \left(\mathbf{x}_k - \mathbf{x}_{\mathbf{ref}_k}\right) + \mathbf{u}_k{}^T \mathbf{R}_k \mathbf{u}_k \\
& + \left(\mathbf{x}_N - \mathbf{x}_{\mathbf{ref}_N}\right)^T \mathbf{Q}_N \left(\mathbf{x}_N - \mathbf{x}_{\mathbf{ref}_N}\right), \quad \mathbf{Q}_k \geq 0, \quad \mathbf{R}_k > 0 \\
\text{s.t.} \quad & \mathbf{x}_{k+1} = \mathbf{A}_k \mathbf{x}_k + \mathbf{B}_k \mathbf{u}_k, \quad \mathbf{x}_{(k=0)} = \mathbf{x}_0, \qquad \forall k = \{0,\ldots,N-1\} \\
& \underline{\mathbf{c}}_k \leq \mathbf{C}_k \mathbf{u}_k \leq \overline{\mathbf{c}}_k \qquad\qquad\qquad\qquad \forall k = \{0,\ldots,N-1\}
\end{aligned} \tag{4-46}$$

The optimization problem of (4-46) can be reformulated to reduce the size of the problem. This modification is based on the condensed QP formulation presented in [165]. To be more precise, the condensed formulation reduces the size of the problem by eliminating the state trajectory from the decision variables of the optimization problem. It also removes the dynamical system constraints from the constraints of the optimization problem. Instead, both the state trajectory and the dynamical system constraints get included in the cost function of the problem. The main difference with [165] is that here the dynamics are time-varying and not time-invariant and a non-zero state reference is present. This formulation gets adopted for the controller of the quadruped since the problem size is significantly larger than that of problems similar to the double pendulum.

The state variables can be eliminated from the decision variables of the problem by expressing them as a function of the initial state and the input trajectory. This relationship becomes apparent with the following example where the states at the first three steps of the horizon $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ are being computed.

$$\begin{aligned}
\mathbf{x}_1 &= \mathbf{A}_0 \mathbf{x}_0 + \mathbf{B}_0 \mathbf{u}_0 \\
\mathbf{x}_2 &= \mathbf{A}_1 \mathbf{x}_1 + \mathbf{B}_1 \mathbf{u}_1 = \mathbf{A}_1 \left(\mathbf{A}_0 \mathbf{x}_0 + \mathbf{B}_0 \mathbf{u}_0\right) + \mathbf{B}_1 \mathbf{u}_1 = \mathbf{A}_1 \mathbf{A}_0 \mathbf{x}_0 + \mathbf{A}_1 \mathbf{B}_0 \mathbf{u}_0 + \mathbf{B}_1 \mathbf{u}_1 \\
\mathbf{x}_3 &= \mathbf{A}_2 \mathbf{x}_2 + \mathbf{B}_2 \mathbf{u}_2 = \mathbf{A}_2 \left(\mathbf{A}_1 \mathbf{A}_0 \mathbf{x}_0 + \mathbf{A}_1 \mathbf{B}_0 \mathbf{u}_0 + \mathbf{B}_1 \mathbf{u}_1\right) + \mathbf{B}_2 \mathbf{u}_2 \\
&= \mathbf{A}_2 \mathbf{A}_1 \mathbf{A}_0 \mathbf{x}_0 + \mathbf{A}_2 \mathbf{A}_1 \mathbf{B}_0 \mathbf{u}_0 + \mathbf{A}_2 \mathbf{B}_1 \mathbf{u}_1 + \mathbf{B}_2 \mathbf{u}_2
\end{aligned} \tag{4-47}$$

Consequently, the state trajectory $\mathbf{X}$ can be calculated, from the initial state $\mathbf{x}_0$ and the input trajectory $\mathbf{U}$, using the following expression:

$$\mathbf{X} = \mathbf{A}_{\mathbf{qp}} \mathbf{x}_0 + \mathbf{B}_{\mathbf{qp}} \mathbf{U} \tag{4-48}$$

where, if the system is LTV, the matrices $\mathbf{A}_{\mathbf{qp}}, \mathbf{B}_{\mathbf{qp}}$ are the following:

$$\mathbf{A_{qp}} = \begin{bmatrix} \mathbf{I}_n \\ \mathbf{A}_0 \\ \mathbf{A}_1\mathbf{A}_0 \\ \vdots \\ \mathbf{A}_{N-2}\mathbf{A}_{N-3}...\mathbf{A}_1\mathbf{A}_0 \\ \mathbf{A}_{N-1}\mathbf{A}_{N-2}...\mathbf{A}_1\mathbf{A}_0 \end{bmatrix} \in \mathbb{R}^{13N\times 13},$$

$$\mathbf{B_{qp}} = \begin{bmatrix} \mathbf{0}_{n\times m} & \mathbf{0}_{n\times m} & \cdots & \cdots & \mathbf{0}_{n\times m} \\ \mathbf{B}_0 & \mathbf{0}_{n\times m} & \cdots & \cdots & \mathbf{0}_{n\times m} \\ \mathbf{A}_1\mathbf{B}_0 & \mathbf{B}_1 & \mathbf{0}_{n\times m} & \cdots & \mathbf{0}_{n\times m} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \mathbf{A}_{N-2}\mathbf{A}_{N-3}...\mathbf{A}_2\mathbf{A}_1\mathbf{B}_0 & \mathbf{A}_{N-2}\mathbf{A}_{N-3}...\mathbf{A}_3\mathbf{A}_2\mathbf{B}_1 & \cdots & \mathbf{B}_{N-2} & \mathbf{0}_{n\times m} \\ \mathbf{A}_{N-1}\mathbf{A}_{N-2}...\mathbf{A}_2\mathbf{A}_1\mathbf{B}_0 & \mathbf{A}_{N-1}\mathbf{A}_{N-2}...\mathbf{A}_3\mathbf{A}_2\mathbf{B}_1 & \cdots & \mathbf{A}_{N-1}\mathbf{B}_{N-2} & \mathbf{B}_{N-1} \end{bmatrix} \in \mathbb{R}^{13N\times 3n_l N} \tag{4-49}$$

while, if the system is LTI, the matrices $\mathbf{A_{qp}}, \mathbf{B_{qp}}$ are the following:

$$\mathbf{A_{qp}} = \begin{bmatrix} \mathbf{I}_n \\ \mathbf{A} \\ \mathbf{A}^2 \\ \vdots \\ \mathbf{A}^{N-1} \\ \mathbf{A}^N \end{bmatrix} \in \mathbb{R}^{13N\times 13}, \quad \mathbf{B_{qp}} = \begin{bmatrix} \mathbf{0}_{n\times m} & & & & \\ \mathbf{B} & \mathbf{0}_{n\times m} & & & \\ \mathbf{AB} & \mathbf{B} & \ddots & & \\ \vdots & & \ddots & & \\ \mathbf{A}^{N-2}\mathbf{B} & & & \mathbf{B} & \mathbf{0}_{n\times m} \\ \mathbf{A}^{N-1}\mathbf{B} & \mathbf{A}^{N-2}\mathbf{B} & \cdots & \mathbf{AB} & \mathbf{B} \end{bmatrix} \in \mathbb{R}^{13N\times 3n_l N} \tag{4-50}$$

Given this expression for the state trajectory, the cost function can be rewritten as follows:

$$J = J(\mathbf{U}) = \left(\mathbf{A_{qp}}\mathbf{x}_0 + \mathbf{B_{qp}}\mathbf{U} - \mathbf{X_{ref}}\right)^T \mathbf{Q_{qp}} \left(\mathbf{A_{qp}}\mathbf{x}_0 + \mathbf{B_{qp}}\mathbf{U} - \mathbf{X_{ref}}\right) + \mathbf{U}^T\mathbf{R_{qp}}\mathbf{U} \tag{4-51}$$

where $\mathbf{X_{ref}} \triangleq \left[\mathbf{x_{ref}}_0, \ \mathbf{x_{ref}}_1 \ \ \ldots, \ \mathbf{x_{ref}}_N\right]^T \in \mathbb{R}^{nN}$ is the state trajectory reference and $\mathbf{Q_{qp}}, \mathbf{R_{qp}}$ are the following matrices:

$$\mathbf{Q_{qp}} = \begin{bmatrix} \mathbf{Q}_0 & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{Q}_1 & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{Q}_N \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{N-1} \otimes \mathbf{Q} & \mathbf{0} \\ \mathbf{0} & \mathbf{Q}_N \end{bmatrix} \in \mathbb{R}^{13N\times 13N},$$

$$\mathbf{R_{qp}} = \begin{bmatrix} \mathbf{R}_0 & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{R}_1 & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{R}_{N-1} \end{bmatrix} = \mathbf{I}_{N-1} \otimes \mathbf{R} \in \mathbb{R}^{3n_l N\times 3n_l N} \tag{4-52}$$

where $\otimes$ denotes the Kronecker product. The second equality in the expressions of $\mathbf{Q_{qp}}, \mathbf{R_{qp}}$ is valid only if $\mathbf{Q}_k = \mathbf{Q}, \ \forall k = \{0,\ldots,N-1\}$ and if $\mathbf{R}_k = \mathbf{R}, \ \forall k = \{0,\ldots,N-1\}$. With these modifications in action, the QP of (4-46) is reformulated into an inequality constrained QP and can be rewritten in the following form:

$$\begin{aligned} \min_{\mathbf{U}} \quad & \frac{1}{2}\mathbf{U}^T\mathbf{H}\mathbf{U} + \mathbf{h}^T\mathbf{U} \\ \text{s.t.} \quad & \underline{\mathbf{c}} \le \mathbf{C}\mathbf{U} \le \overline{\mathbf{c}} \end{aligned} \tag{4-53}$$

where the Hessian $\mathbf{H} \in \mathbb{R}^{3n_l N \times 3n_l N}$ and the gradient $\mathbf{h} \in \mathbb{R}^{3n_l N \times 1}$ of the cost function are the following:

$$\mathbf{H} = 2\left(\mathbf{B}_{\mathbf{qp}}^T \mathbf{Q}_{\mathbf{qp}} \mathbf{B}_{\mathbf{qp}} + \mathbf{R}_{\mathbf{qp}}\right) \tag{4-54}$$

$$\mathbf{h} = 2\mathbf{B}_{\mathbf{qp}}^T \mathbf{Q}_{\mathbf{qp}} \left(\mathbf{A}_{\mathbf{qp}} \mathbf{x}_0 - \mathbf{X}_{\mathbf{ref}}\right) \tag{4-55}$$

And the linear constraints matrix $\mathbf{C} \in \mathbb{R}^{5n_l N \times 3n_l N}$ and the constraint bound vectors $\underline{\mathbf{c}}, \overline{\mathbf{c}} \in \mathbb{R}^{5n_l N \times 1}$ are the following:

$$\mathbf{C} = \begin{bmatrix} \mathbf{C}_0 & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{C}_1 & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{C}_{N-1} \end{bmatrix}, \quad \underline{\mathbf{c}} = \begin{bmatrix} \underline{\mathbf{c}}_0 \\ \underline{\mathbf{c}}_1 \\ \vdots \\ \underline{\mathbf{c}}_{N-1} \end{bmatrix}, \quad \overline{\mathbf{c}} = \begin{bmatrix} \overline{\mathbf{c}}_0 \\ \overline{\mathbf{c}}_1 \\ \vdots \\ \overline{\mathbf{c}}_{N-1} \end{bmatrix} \tag{4-56}$$

The desired GRFs are then the first $3n_l$ elements of $\mathbf{U}$. Given these GRFs, the joint motor torques for the ith leg $\boldsymbol{\tau}_{st,i} \in \mathbb{R}^3$, are computed using the following expression:

$$\boldsymbol{\tau}_{st,i} = \mathbf{c}_i\left(\mathbf{q}_i, \dot{\mathbf{q}}_i\right) + \mathbf{G}_i\left(\mathbf{q}_i\right) - \mathbf{J}_i^T\left(\mathbf{q}_i\right)\mathbf{f}_i \tag{4-57}$$

where $\mathbf{q}_i, \dot{\mathbf{q}}_i \in \mathbb{R}^3$ are the joint angles and angular velocities of the ith leg, $\mathbf{J}_i \in \mathbb{R}^{3\times3}$ is the foot Jacobian of the ith leg and $\mathbf{c}_i, \mathbf{G}_i \in \mathbb{R}^3$ are the Coriolis and centripetal and gravity terms of the ith leg.

4.3.5 Footstep Planner

Both the swing and stance leg controllers need to be aware of the commanded foothold locations. Since foothold positions are not part of the decision variables of the MPC, they have to be pre-specified. This selection is accomplished using some standard heuristics. To be more precise, the reference foothold position of the ith leg on the ground plane $\mathbf{r}_i^{cmd}$ is computed from the corresponding hip location projected on the ground plane $\mathbf{p}_{\mathbf{h},i}$ using a linear combination of the Raibert heuristic [11], a capture-point-based velocity feedback term [26] and a centrifugal term [33], [67].

$$\mathbf{r}_i^{cmd} = \mathbf{p}_{\mathbf{h},i} + \frac{T_{st}}{2}\dot{\mathbf{p}}_{\mathbf{h},i}^{cmd} + \sqrt{\frac{z_0}{\|\mathbf{g}\|}}\left(\dot{\mathbf{p}}_{\mathbf{h},i} - \dot{\mathbf{p}}_{\mathbf{h},i}^{cmd}\right) + \frac{z_0}{g}\left(\dot{\mathbf{p}}_{\mathbf{h},i} \times \boldsymbol{\omega}^{cmd}\right) \tag{4-58}$$

where z_0 is the nominal locomotion height, T_{st} is the stance period, $\dot{\mathbf{p}}_{\mathbf{h},i}$ is the ith leg hip (linear) velocity, $\dot{\mathbf{p}}_{\mathbf{h},i}^{cmd}$ is the desired ith leg hip (linear) velocity and $\boldsymbol{\omega}^{cmd}$ is the desired angular velocity. The velocity feedback and centrifugal terms are associated with velocity and angular velocity tracking [33]. They are included in the footstep planner so that undesired moments produced during agile locomotion tasks by the GRFs are counteracted [67].

4.4 Implementation of Convex (QP-based) MPC

First of all, some important information and details about the software, routines and algorithms that were utilized by the Convex MPC controllers have to be presented. A list of all the function declarations is located in Appendix B.

First and foremost, this entire project consists of four ROS packages. The tree structure of the project is depicted in Figure 4-9. The first one is *plugins* package which is responsible for the interaction of the controllers with the simulation model and environment. The second

one is *argos_mpc_control* that contains the implementation of the Convex MPC controller. The tree structure of the *argos_mpc_control* pakage is depicted in Figure 4-10. The third one is *argos_mpc_description* that contains the URDF description of the model that is simulated. The fourth and final package is *argos_mpc_gazebo* that contains a *.launch* file that spawns the model in the simulation environment and a *.world* file that contains the description of the environment in which the model is being simulated. The entire project can be found at the CSL-Legged Team bitbucket repository[3].

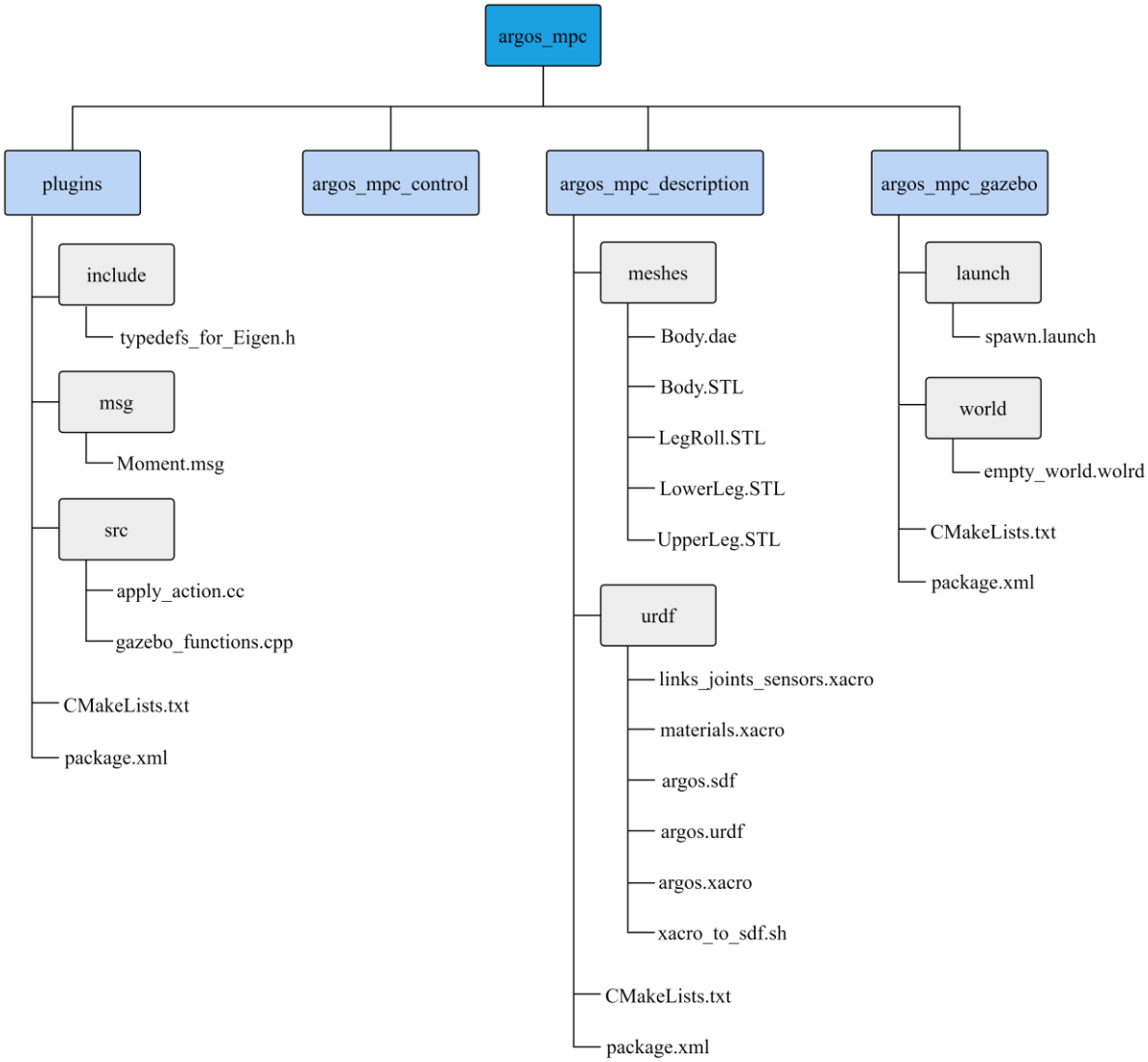


Figure 4-9. Project tree structure of argos_mpc.

[3] https://bitbucket.org/csl_legged/argos_mpc/src/main/

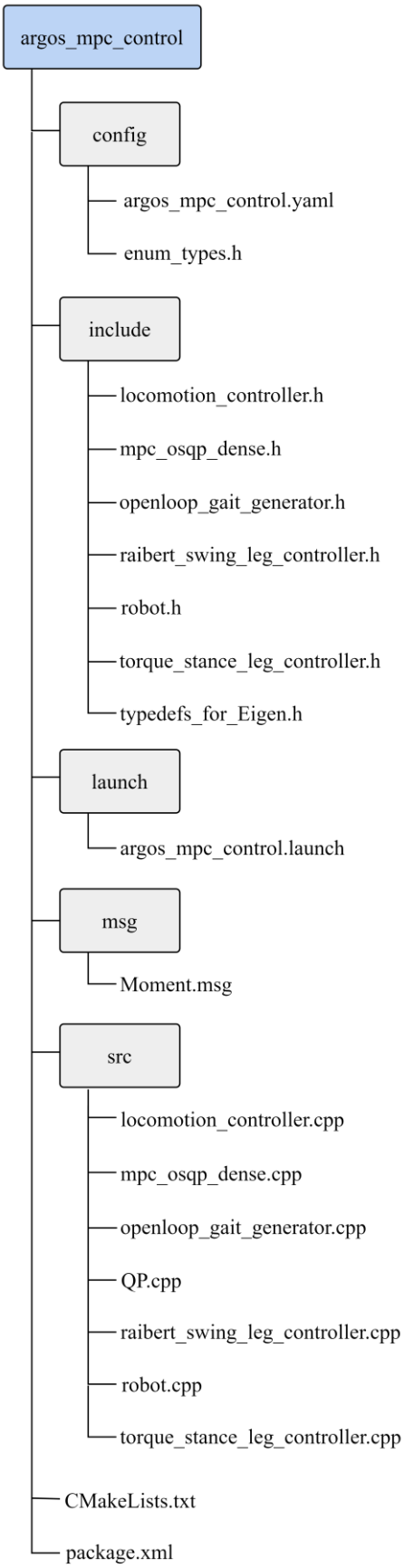


Figure 4-10. Directory tree of argos_mpc_control package.

The controller implemented in this work requires algorithms that conduct tasks such as coordinate frame transformations, forward kinematics and forward dynamics and the computation of quantities like the spatial Jacobians of specific frames of the robot and the joint space inertia matrix. These tasks and computations are performed using custom functions that belong to the custom class **Robot**. These functions rely mainly on algorithms that are provided by the Pinocchio library. The header file *robot.h* contains the definition of the class, that includes declaration of its member variables and functions. The implementation of the class is located in the *robot.cpp* file.

To do so, a Model and a Data object that correspond to the Gazebo model of ARGOS must be generated, given its *.urdf* description. This generation is conducted in Pinocchio as shown in the following code snippet, using the function **BuildPinocchioModel(.)**. Since quadruped robots are floating base systems, it is important to connect the body of the quadruped (base) to the world frame using a joint that allows the base to perform translations along the three axes of the world frame and rotations about these axes as well. This joint is a composite joint that consists of a **JointModelTranslation(.)** joint and a **JointModelSphericalZYX(.)** joint. The first one allows the translation of the base along the x, y, z axes and the second one allows the rotation of the base about these axes. The pinocchio function **addJoint(.)** allows the user to add a joint, given its $\mathbb{SE}(3)$ placement relative to its predecessor. Furthermore, this function also adds frames to the centers of the robot's toes, which are considered as its end-effectors where the GRFs act. The pinocchio function **addFrame(.)** allows the user to add a frame, given its $\mathbb{SE}(3)$ placement (relative position and orientation) w.r.t. the parent joint frame. These frames are called operational frames and are necessary for the computation of the contact Jacobians that map the GRFs to motor torques. Finally, this function also calculates the total mass of the robot and the total moment of inertia of the robot in the nominal joint position. Necessary for this calculation is the Pinocchio function **ccrba(.)**.

```
void Robot::BuildPinocchioModel(Eigen::VectorXd& q)
{
  // You should change here to set up your own URDF file or just pass it
  // as an argument of this example.
  std::string urdf_package_filename =
      ros::package::getPath("argos_mpc_description");
  std::string urdf_filename =
      urdf_package_filename + std::string("/urdf/argos.urdf");

  std::cout << "Opening file: " << urdf_filename << std::endl;

  SE3 config1 = SE3::Identity(); //(0, 0, 0);
  SE3 config2 = SE3::Identity(); //(0, 0, 0);

  JointModelComposite root_joint;
  root_joint.addJoint(pinocchio::JointModelTranslation(), config1);
  root_joint.addJoint(pinocchio::JointModelSphericalZYX(), config2);
  root_joint.setIndexes(0, 0, 0);

  std::cout << "root_joint nv: " << root_joint.nv() << std::endl;
  std::cout << "root_joint nq: " << root_joint.nq() << std::endl;
  std::cout << "root_joint njoints: " << root_joint.njoints << std::endl;

  pinocchio::urdf::buildModel(urdf_filename, root_joint, model);
  std::cout << "model name: " << model.name << std::endl;
```

```
std::cout << "model nv: " << model.nv << std::endl;
std::cout << "model nq: " << model.nq << std::endl;

// Set model gravity
model.gravity.linear() << 0.0, 0.0, -kGravity;

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

// Add frame at leg toes

Eigen::Vector3d trans;
trans(0) = 0.0;
trans(1) = -0.107521;
trans(2) = -0.578551;
Eigen::Matrix3d rot;
rot = Eigen::Matrix3d::Identity();

SE3 framePlacement;
framePlacement.translation_impl(trans);
framePlacement.rotation_impl(rot);

std::cout << "Here is framePlacement:\n"
          << framePlacement << std::endl;

////////////////////////////////////////////////////////////////////////

Model::Index parent_idx = model.existJointName("FL_joint_knee") ?
                              model.getJointId("FL_joint_knee") :
                              (Model::Index)(model.njoints - 1);
std::cout << "Here is parent_idx:\n" << parent_idx << std::endl;

Model::Index previous_frame_idx = model.getFrameId("FL_joint_knee");
std::cout << "Here is previous_frame_idx :\n"
          << previous_frame_idx << std::endl;

const std::string& frame_name_FL =
    std::string(model.names[parent_idx] + "_foot_frame");
std::cout << "Here is frame_name:\n" << frame_name_FL << std::endl;

model.addFrame(Frame(frame_name_FL, parent_idx, previous_frame_idx,
                     framePlacement, OP_FRAME));

Model::Index frame_idx = model.getFrameId(frame_name_FL);
std::cout << "Here is frame_idx :\n" << frame_idx << std::endl;

////////////////////////////////////////////////////////////////////////

parent_idx = model.existJointName("FR_joint_knee") ?
                 model.getJointId("FR_joint_knee") :
                 (Model::Index)(model.njoints - 1);
previous_frame_idx = model.getFrameId("FR_joint_knee");

const std::string& frame_name_FR =
    std::string(model.names[parent_idx] + "_foot_frame");
std::cout << "Here is frame_name:\n" << frame_name_FR << std::endl;

model.addFrame(Frame(frame_name_FR, parent_idx, previous_frame_idx,
                     framePlacement, OP_FRAME));
```

```
  ////////////////////////////////////////////////////////////////////////

  parent_idx = model.existJointName("HL_joint_knee") ?
                   model.getJointId("HL_joint_knee") :
                   (Model::Index)(model.njoints - 1);
  previous_frame_idx = model.getFrameId("HL_joint_knee");

  const std::string& frame_name_HL =
      std::string(model.names[parent_idx] + "_foot_frame");
  std::cout << "Here is frame_name:\n" << frame_name_HL << std::endl;

  model.addFrame(Frame(frame_name_HL, parent_idx, previous_frame_idx,
                       framePlacement, OP_FRAME));

  ////////////////////////////////////////////////////////////////////////

  parent_idx = model.existJointName("HR_joint_knee") ?
                   model.getJointId("HR_joint_knee") :
                   (Model::Index)(model.njoints - 1);
  previous_frame_idx = model.getFrameId("HR_joint_knee");

  const std::string& frame_name_HR =
      std::string(model.names[parent_idx] + "_foot_frame");
  std::cout << "Here is frame_name:\n" << frame_name_HR << std::endl;

  model.addFrame(Frame(frame_name_HR, parent_idx, previous_frame_idx,
                       framePlacement, OP_FRAME));

  ////////////////////////////////////////////////////////////////////////

  data = Data(model);
  Eigen::VectorXd v = Eigen::VectorXd::Zero(model.nv);

  pinocchio::ccrba(model, data, q, v);

  robot_mass = data.Ig.mass();
  robot_inertia = data.Ig.inertia();

  std::cout << "Here is mass :\n" << robot_mass << std::endl;
  std::cout << "Here is inertia:\n" << robot_inertia << std::endl;
}
```

Additionally, the positions of the abduction and hip joints of the robot relative to the body frame, w.r.t the body frame, must also be computed. These computations are performed using the functions **HipPositionsInBaseFrame()** and **AbdPositionsInBaseFrame()** that are illustrated in the following two code snippets. The joint frame $\mathbb{SE}(3)$ placement w.r.t. the world frame is equal to **data.oMi[_index]**, where **_index** is the index of that particular joint, while the $\mathbb{SE}(3)$ placement of an operational frame w.r.t. the world frame is equal to **data.oMf[_index]**, where **_index** is the index of that particular frame. The index of a particular joint can be extracted by calling the Pinocchio function **getJointId(.)** while the index of a particular frame can be extracted by calling the Pinocchio function **getFrameId(.)**.

```
void Robot::AbdPositionsInBaseFrame()
{
  Eigen::VectorXd q = Eigen::VectorXd::Zero(com_dof + num_motors);
  forwardKinematics(model, data, q);
```

```
  Model::Index parent_idx = model.existJointName("root_joint") ?
                                model.getJointId("root_joint") :
                              (Model::Index)(model.njoints - 1);
  Model::Index parent_idx_FL = model.existJointName("FL_joint_abd") ?
                                   model.getJointId("FL_joint_abd") :
                                  (Model::Index)(model.njoints - 1);

  Eigen::Vector3d trans_B = data.oMi[parent_idx].translation();
  Eigen::Vector3d trans_FL = data.oMi[parent_idx_FL].translation();

  abd_positions_base_frame.col(0) =
      data.oMi[parent_idx].rotation().transpose() * (trans_FL - trans_B);

  ////////////////////////////////////////////////////////////////////////

  Model::Index parent_idx_FR = model.existJointName("FR_joint_abd") ?
                                   model.getJointId("FR_joint_abd") :
                                  (Model::Index)(model.njoints - 1);

  Eigen::Vector3d trans_FR = data.oMi[parent_idx_FR].translation();

  abd_positions_base_frame.col(1) =
      data.oMi[parent_idx].rotation().transpose() * (trans_FR - trans_B);

  ////////////////////////////////////////////////////////////////////////

  Model::Index parent_idx_HL = model.existJointName("HL_joint_abd") ?
                                   model.getJointId("HL_joint_abd") :
                                  (Model::Index)(model.njoints - 1);

  Eigen::Vector3d trans_HL = data.oMi[parent_idx_HL].translation();

  abd_positions_base_frame.col(2) =
      data.oMi[parent_idx].rotation().transpose() * (trans_HL - trans_B);

  ////////////////////////////////////////////////////////////////////////

  Model::Index parent_idx_HR = model.existJointName("HR_joint_abd") ?
                                   model.getJointId("HR_joint_abd") :
                                  (Model::Index)(model.njoints - 1);

  Eigen::Vector3d trans_HR = data.oMi[parent_idx_HR].translation();

  abd_positions_base_frame.col(3) =
      data.oMi[parent_idx].rotation().transpose() * (trans_HR - trans_B);

  std::cout << "Here is abd_positions_base_frame:\n"
            << abd_positions_base_frame << std::endl;
}
```

```
void Robot::HipPositionsInBaseFrame()
{
  Eigen::VectorXd q = Eigen::VectorXd::Zero(com_dof + num_motors);
  forwardKinematics(model, data, q);

  Model::Index parent_idx = model.existJointName("root_joint") ?
                                model.getJointId("root_joint") :
                              (Model::Index)(model.njoints - 1);
  Model::Index parent_idx_FL = model.existJointName("FL_joint_hip") ?
                                   model.getJointId("FL_joint_hip") :
                                  (Model::Index)(model.njoints - 1);
```

```
  Eigen::Vector3d trans_B = data.oMi[parent_idx].translation();
  Eigen::Vector3d trans_FL = data.oMi[parent_idx_FL].translation();

  hip_positions_base_frame.col(0) =
      data.oMi[parent_idx].rotation().transpose() * (trans_FL - trans_B);

  ////////////////////////////////////////////////////////////////////////

  Model::Index parent_idx_FR = model.existJointName("FR_joint_hip") ?
                                    model.getJointId("FR_joint_hip") :
                                   (Model::Index)(model.njoints - 1);

  Eigen::Vector3d trans_FR = data.oMi[parent_idx_FR].translation();

  hip_positions_base_frame.col(1) =
      data.oMi[parent_idx].rotation().transpose() * (trans_FR - trans_B);

  ////////////////////////////////////////////////////////////////////////

  Model::Index parent_idx_HL = model.existJointName("HL_joint_hip") ?
                                    model.getJointId("HL_joint_hip") :
                                   (Model::Index)(model.njoints - 1);

  Eigen::Vector3d trans_HL = data.oMi[parent_idx_HL].translation();

  hip_positions_base_frame.col(2) =
      data.oMi[parent_idx].rotation().transpose() * (trans_HL - trans_B);

  ////////////////////////////////////////////////////////////////////////

  Model::Index parent_idx_HR = model.existJointName("HR_joint_hip") ?
                                    model.getJointId("HR_joint_hip") :
                                   (Model::Index)(model.njoints - 1);

  Eigen::Vector3d trans_HR = data.oMi[parent_idx_HR].translation();

  hip_positions_base_frame.col(3) =
      data.oMi[parent_idx].rotation().transpose() * (trans_HR - trans_B);

  std::cout << "Here is hip_positions_base_frame:\n"
            << hip_positions_base_frame << std::endl;
}
```

Additionally, the Joint Space Inertia matrix of each leg of the quadruped is required by the swing leg controller for the commanded torque computation. It is computed using a rigid body dynamics algorithm named Composite Rigid Body Algorithm (CRBA) [143]. In Pinocchio the upper triangular part of the joint space inertia matrix is computed by calling the function **crba(.)** and the result is accessible through **data.M**. This computation is executed by calling the **ComputeInertiaMatrix(.)** function that is implemented as shown in the following code snippet:

```
void Robot::ComputeInertiaMatrix(Eigen::VectorXd& q,
                                 Eigen::MatrixXd& mass_matrix)
{
  // Joint Space Inertia Matrix
  crba(model, data, q);

  // data.M.triangularView<Eigen::StrictlyLower>() =
  //     data.M.transpose().triangularView<Eigen::StrictlyLower>();
```

```
  // MatrixXd M_ = data.M;

  mass_matrix = data.M;
}
```

Additionally, the sum of the Coriolis, centrifugal and gravitational terms of each leg of the quadruped is required by the stance and swing leg controller for the commanded torque computation. These terms are also called non-linear effects or bias terms. In Pinocchio the non-linear effects are computed using the function **nonLinearEffects(.)**. This computation is executed by calling the **ComputeBiasTerms(.)** function that is implemented as shown in the following code snippet:

```
void Robot::ComputeBiasTerms(Eigen::VectorXd& q, Eigen::VectorXd& q_dot,
                             Eigen::VectorXd& bias_terms)
{
  // bias_terms = computeGeneralizedGravity(model, data, q);
  bias_terms = nonLinearEffects(model, data, q, q_dot);
}
```

Additionally, the Jacobian matrix of the operational frame of each leg's toe is required by the stance leg controller for mapping the computed GRFs to motor torques. This computation is performed using the function **ComputeFootJacobian(.)** that is illustrated in the following code snippet. These Jacobians are computed w.r.t. the robot's body frame. Firstly, the forward kinematics of the model must be computed given the current configuration of the model, using the Pinocchio function **forwardkinematics(.)**. In this way the joint placements get updated according to the current joint configuration. Then, the placement of each frame contained in the model should also be updated, according to the current joint configuration. This is achieved by calling the Pinocchio function **updateFramePlacements(.)**. Afterwards the Jacobians of the frames of the model can be computed by calling the Pinocchio function **computeJointJacobians(.)**. Then, the Jacobian of a specific end-effector can be extracted by calling the function **getFrameJacobian(.)**, where the index of the frame has to be specified as well as the reference frame w.r.t. which the Jacobian is computed. In this function, it is calculated w.r.t. the world frame. This choice is signified by setting the reference frame argument to LOCAL_WORLD_ALIGNED. Then it gets transformed so that it is expressed w.r.t. the robot's body frame.

```
void Robot::ComputeFootJacobian(Eigen::VectorXd& q, int& leg_id,
                                Eigen::MatrixXd& Jac)
{
  // Computes the Jacobian matrix for the given leg

  forwardKinematics(model, data, q);
  updateFramePlacements(model, data);

  Matrix3d R = data.oMi[1].rotation();
  // R = Eigen::Matrix3d::Identity();

  pinocchio::computeJointJacobians(model, data, q);

  ////////////////////////////////////////////////////////////////////////

  if (leg_id == FL)
  {
```

```
    Model::Index frame_idx =
    model.getFrameId("FL_joint_knee_foot_frame");
    // std::cout << "Here is frame_idx :\n" << frame_idx << std::endl;

    MatrixXd Jac4_loc(6, model.nv);
    Jac4_loc.fill(0);
    getFrameJacobian(model, data, frame_idx, LOCAL_WORLD_ALIGNED,
                     Jac4_loc);

    VectorXd Jcol3(6);
    VectorXd Jcol5(6);

    Jcol3 = Jac4_loc.col(3);
    Jcol5 = Jac4_loc.col(5);

    Jac4_loc.col(3) = Jcol5;
    Jac4_loc.col(5) = Jcol3;

    // std::cout << "Here is Jac4_loc:\n" << Jac4_loc << std::endl;

    MatrixXd Jac4(6, model.nv);
    Jac4.fill(0);

    Jac4.block(0, 6, 3, 3) = R.transpose() * Jac4_loc.block(0, 6, 3, 3);

    // std::cout << "Here is Jac4:\n" << Jac4 << std::endl;

    Jac = Jac4.block(0, 0, 3, Jac4.cols());

    // std::cout << "Here is Jac:\n" << Jac << std::endl;
  }

  ////////////////////////////////////////////////////////////////////////

  else if (leg_id == FR)
  {
    Model::Index frame_idx =
    model.getFrameId("FR_joint_knee_foot_frame");
    // std::cout << "Here is frame_idx :\n" << frame_idx << std::endl;

    MatrixXd Jac7_loc(6, model.nv);
    Jac7_loc.fill(0);
    getFrameJacobian(model, data, frame_idx, LOCAL_WORLD_ALIGNED,
                     Jac7_loc);

    VectorXd Jcol3(6);
    VectorXd Jcol5(6);

    Jcol3 = Jac7_loc.col(3);
    Jcol5 = Jac7_loc.col(5);

    Jac7_loc.col(3) = Jcol5;
    Jac7_loc.col(5) = Jcol3;

    // std::cout << "Here is Jac7_loc:\n" << Jac7_loc << std::endl;

    MatrixXd Jac7(6, model.nv);
    Jac7.fill(0);

    Jac7.block(0, 9, 3, 3) = R.transpose() * Jac7_loc.block(0, 9, 3, 3);
```

```
    // std::cout << "Here is Jac7:\n" << Jac7 << std::endl;

    Jac = Jac7.block(0, 0, 3, Jac7.cols());

    // std::cout << "Here is Jac:\n" << Jac << std::endl;
  }

  ////////////////////////////////////////////////////////////////////////

  else if (leg_id == HL)
  {
    Model::Index frame_idx =
    model.getFrameId("HL_joint_knee_foot_frame");
    // std::cout << "Here is frame_idx :\n" << frame_idx << std::endl;

    MatrixXd Jac10_loc(6, model.nv);
    Jac10_loc.fill(0);
    getFrameJacobian(model, data, frame_idx, LOCAL_WORLD_ALIGNED,
                     Jac10_loc);

    VectorXd Jcol3(6);
    VectorXd Jcol5(6);

    Jcol3 = Jac10_loc.col(3);
    Jcol5 = Jac10_loc.col(5);

    Jac10_loc.col(3) = Jcol5;
    Jac10_loc.col(5) = Jcol3;

    // std::cout << "Here is Jac10_loc:\n" << Jac10_loc << std::endl;

    MatrixXd Jac10(6, model.nv);
    Jac10.fill(0);

    Jac10.block(0, 12, 3, 3) =
        R.transpose() * Jac10_loc.block(0, 12, 3, 3);

    // std::cout << "Here is Jac10:\n" << Jac10 << std::endl;

    Jac = Jac10.block(0, 0, 3, Jac10.cols());

    // std::cout << "Here is Jac:\n" << Jac << std::endl;
  }

  ////////////////////////////////////////////////////////////////////////

  else if (leg_id == HR)
  {
    Model::Index frame_idx =
    model.getFrameId("HR_joint_knee_foot_frame");
    // std::cout << "Here is frame_idx :\n" << frame_idx << std::endl;

    MatrixXd Jac13_loc(6, model.nv);
    Jac13_loc.fill(0);
    getFrameJacobian(model, data, frame_idx, LOCAL_WORLD_ALIGNED,
                     Jac13_loc);

    VectorXd Jcol3(6);
    VectorXd Jcol5(6);

    Jcol3 = Jac13_loc.col(3);
```

```
    Jcol5 = Jac13_loc.col(5);

    Jac13_loc.col(3) = Jcol5;
    Jac13_loc.col(5) = Jcol3;

    // std::cout << "Here is Jac13_loc:\n" << Jac13_loc << std::endl;

    MatrixXd Jac13(6, model.nv);
    Jac13.fill(0);

    Jac13.block(0, 15, 3, 3) =
        R.transpose() * Jac13_loc.block(0, 15, 3, 3);

    // std::cout << "Here is Jac13:\n" << Jac13 << std::endl;

    Jac = Jac13.block(0, 0, 3, Jac13.cols());

    // std::cout << "Here is Jac:\n" << Jac << std::endl;
  }
}
```

Additionally, the time derivative of the Jacobian matrix of the operational frame of each leg's toe is required by the swing leg controller for the commanded torque computation. This computation is performed using the function **ComputeFootJacobianTimeVariation(.)** that is illustrated in the following code snippet. The same operations that are performed in the function **ComputeFootJacobian(.)** are also performed here but instead of the Pinocchio functions **computeJointJacobians(.)** and **getFrameJacobian(.)** the functions **computeJointJacobiansTimeVariation(.)** and **getFrameJacobianTimeVariation(.)** are utilized. These functions compute instead the respective Jacobian variations with respect to time.

```
void Robot::ComputeFootJacobianTimeVariation(Eigen::VectorXd& q,
                                             Eigen::VectorXd& q_dot,
                                             int& leg_id,
                                             Eigen::MatrixXd& Jac_dot)
{
  // Computes the Jacobian matrix for the given leg

  forwardKinematics(model, data, q);
  updateFramePlacements(model, data);

  Matrix3d R = data.oMi[1].rotation();
  // R = Eigen::Matrix3d::Identity();

  pinocchio::computeJointJacobiansTimeVariation(model, data, q, q_dot);

  ////////////////////////////////////////////////////////////////////////

  if (leg_id == FL)
  {
    Model::Index frame_idx =
        model.getFrameId("FL_joint_knee_foot_frame");
    // std::cout << "Here is frame_idx :\n" << frame_idx << std::endl;

    MatrixXd Jac4_loc_dot(6, model.nv);
    Jac4_loc_dot.fill(0);
    getFrameJacobianTimeVariation(model, data, frame_idx,
                                  LOCAL_WORLD_ALIGNED, Jac4_loc_dot);
```

```
  VectorXd Jcol3(6);
  VectorXd Jcol5(6);

  Jcol3 = Jac4_loc_dot.col(3);
  Jcol5 = Jac4_loc_dot.col(5);

  Jac4_loc_dot.col(3) = Jcol5;
  Jac4_loc_dot.col(5) = Jcol3;

  // std::cout << "Here is Jac4_loc_dot:\n" << Jac4_loc_dot <<
  // std::endl;

  MatrixXd Jac4_dot(6, model.nv);
  Jac4_dot.fill(0);

  Jac4_dot.block(0, 6, 3, 3) =
      R.transpose() * Jac4_loc_dot.block(0, 6, 3, 3);

  // std::cout << "Here is Jac4_dot:\n" << Jac4_dot << std::endl;

  Jac_dot = Jac4_dot.block(0, 0, 3, Jac4_dot.cols());

  // std::cout << "Here is Jac_dot:\n" << Jac_dot << std::endl;
}

////////////////////////////////////////////////////////////////////////

else if (leg_id == FR)
{
  Model::Index frame_idx =
      model.getFrameId("FR_joint_knee_foot_frame");
  // std::cout << "Here is frame_idx :\n" << frame_idx << std::endl;

  MatrixXd Jac7_loc_dot(6, model.nv);
  Jac7_loc_dot.fill(0);
  getFrameJacobianTimeVariation(model, data, frame_idx,
                                LOCAL_WORLD_ALIGNED, Jac7_loc_dot);

  VectorXd Jcol3(6);
  VectorXd Jcol5(6);

  Jcol3 = Jac7_loc_dot.col(3);
  Jcol5 = Jac7_loc_dot.col(5);

  Jac7_loc_dot.col(3) = Jcol5;
  Jac7_loc_dot.col(5) = Jcol3;

  // std::cout << "Here is Jac7_loc_dot:\n" << Jac7_loc_dot <<
  // std::endl;

  MatrixXd Jac7_dot(6, model.nv);
  Jac7_dot.fill(0);

  Jac7_dot.block(0, 9, 3, 3) =
      R.transpose() * Jac7_loc_dot.block(0, 9, 3, 3);

  // std::cout << "Here is Jac7_dot:\n" << Jac7_dot << std::endl;

  Jac_dot = Jac7_dot.block(0, 0, 3, Jac7_dot.cols());
```

```
    // std::cout << "Here is Jac_dot:\n" << Jac_dot << std::endl;
  }

  ////////////////////////////////////////////////////////////////////////

  else if (leg_id == HL)
  {
    Model::Index frame_idx =
        model.getFrameId("HL_joint_knee_foot_frame");
    // std::cout << "Here is frame_idx :\n" << frame_idx << std::endl;

    MatrixXd Jac10_loc_dot(6, model.nv);
    Jac10_loc_dot.fill(0);
    getFrameJacobianTimeVariation(model, data, frame_idx,
                                  LOCAL_WORLD_ALIGNED, Jac10_loc_dot);

    VectorXd Jcol3(6);
    VectorXd Jcol5(6);

    Jcol3 = Jac10_loc_dot.col(3);
    Jcol5 = Jac10_loc_dot.col(5);

    Jac10_loc_dot.col(3) = Jcol5;
    Jac10_loc_dot.col(5) = Jcol3;

    // std::cout << "Here is Jac10_loc:\n" << Jac10_loc << std::endl;

    MatrixXd Jac10_dot(6, model.nv);
    Jac10_dot.fill(0);

    Jac10_dot.block(0, 12, 3, 3) =
         R.transpose() * Jac10_loc_dot.block(0, 12, 3, 3);

    // std::cout << "Here is Jac10_dot:\n" << Jac10_dot << std::endl;

    Jac_dot = Jac10_dot.block(0, 0, 3, Jac10_dot.cols());

    // std::cout << "Here is Jac_dot:\n" << Jac_dot << std::endl;
  }

  ////////////////////////////////////////////////////////////////////////

  else if (leg_id == HR)
  {
    Model::Index frame_idx =
        model.getFrameId("HR_joint_knee_foot_frame");
    // std::cout << "Here is frame_idx :\n" << frame_idx << std::endl;

    MatrixXd Jac13_loc_dot(6, model.nv);
    Jac13_loc_dot.fill(0);
    getFrameJacobianTimeVariation(model, data, frame_idx,
                                  LOCAL_WORLD_ALIGNED, Jac13_loc_dot);

    VectorXd Jcol3(6);
    VectorXd Jcol5(6);

    Jcol3 = Jac13_loc_dot.col(3);
    Jcol5 = Jac13_loc_dot.col(5);

    Jac13_loc_dot.col(3) = Jcol5;
    Jac13_loc_dot.col(5) = Jcol3;
```

```
    // std::cout << "Here is Jac13_loc_dot:\n" << Jac13_loc_dot <<
    // std::endl;

    MatrixXd Jac13_dot(6, model.nv);
    Jac13_dot.fill(0);

    Jac13_dot.block(0, 15, 3, 3) =
         R.transpose() * Jac13_loc_dot.block(0, 15, 3, 3);

    // std::cout << "Here is Jac13_dot:\n" << Jac13_dot << std::endl;

    Jac_dot = Jac13_dot.block(0, 0, 3, Jac13_dot.cols());

    // std::cout << "Here is Jac_dot:\n" << Jac_dot << std::endl;
  }
}
```

Additionally, the GRFs computed by the stance leg controller have to be mapped to joint motor torque, as it is described by equation (4-57). This mapping is executed by calling the function **MapContactForceToJointTorques(.)** and is presented in the following code snippet. The Jacobian and the bias terms needed for the computation are computed by calling the functions **ComputeFootJacobian(.)**, **ComputeBiasTerms(.)** previously described.

```
void Robot::MapContactForceToJointTorques(int& leg_id,
                                          Eigen::VectorXd& q,
                                          Eigen::Vector3d& contact_force,
                                          Eigen::VectorXd& nle,
                                          VectorNm& motor_torques)
{
  // Maps the foot contact force to the leg joint torques

  Eigen::MatrixXd Jv;
  ComputeFootJacobian(q, leg_id, Jv);

  Eigen::VectorXd all_motor_torques;
  all_motor_torques = Jv.transpose() * contact_force;

  // std::cout << "Here is all_motor_torques:\n" << all_motor_torques <<
  // std::endl;

  all_motor_torques = nle - all_motor_torques;

  for (int joint_id = leg_id * motors_per_leg;
       joint_id < (leg_id + 1) * motors_per_leg; joint_id++)
  {
    motor_torques(joint_id) = all_motor_torques(com_dof + joint_id);
  }

  // std::cout << "Here is motor_torques:\n" << motor_torques <<
  // std::endl;
}
```

Also, the angular velocity of the robot's body provided by the ROS topic must be converted to the rate of change of the Z-Y-X Euler angles. This conversion is achieved by calling the function **BaseRollPitchYawRate(.)** that can be viewed in the following code snippet.

```
void Robot::BaseRollPitchYawRate(Vector3d& orientation_rpy,
                                 Vector3d& angular_velocity,
                                 Eigen::Vector3d& rpy_rate)
{
  // Get the rate of orientation change of the argos's base in euler
  // angle ( rate of (roll, pitch, yaw) change )

  Eigen::Matrix3d Rot = rpy::rpyToMatrix(
      orientation_rpy(0), orientation_rpy(1), orientation_rpy(2));
  Eigen::Matrix3d Rot_inv = Rot.transpose();

  rpy_rate = Rot_inv * angular_velocity;
}
```

Also, the linear velocity of the CoM of the robot's body provided by the ROS topic in the world frame must be expressed w.r.t. the robot's body frame. This conversion is achieved by calling the function **BaseVelocityInBodyFrame(.)** that can be viewed in the following code snippet.

```
void Robot::BaseVelocityInBodyFrame(
    Vector3d& orientation_rpy, Vector3d& linear_vel_b,
    Eigen::Vector3d& relative_linear_vel_b)
{
  // Get the linear velocity of argos's base (body) in the base frame

  Eigen::Matrix3d Rot = rpy::rpyToMatrix(
      orientation_rpy(0), orientation_rpy(1), orientation_rpy(2));
  Eigen::Matrix3d Rot_inv = Rot.transpose();

  relative_linear_vel_b = Rot_inv * linear_vel_b;
}
```

In addition to that, an inverse kinematics algorithm needs to be executed to convert the desired foot position into desired joint angles, for the joint space control implemented by the swing leg controller. This conversion is achieved by calling the function **ComputeJointAnglesFromFootLocalPosition(.)**. This function is presented in the code snippet below. This function accepts as an argument the foot positions expressed in the robot's body frame. Initially it performs a transformation to express the foot positions in a frame whose origin coincides with the origin of the abduction joint frame of each leg and is aligned with the robot's body frame. This conversion is necessary since the inverse kinematics algorithm utilized accepts foot positions that are expressed in that specific frame. Coordinate transformations in Pinocchio can be performed as follows: an $\mathbb{SE}(3)$ variable is created given the relative position and orientation of the one frame relative to the other one. Thus, the coordinate transformation is defined. It can be then applied to a vector, using the function **act(.)**.

```
void Robot::ComputeJointAnglesFromFootLocalPosition(
    int& leg_id, Eigen::VectorXd& q,
    Vector3d& foot_local_position_base_frame, Vector3d& joint_ids_leg_id,
    Vector3d& joint_angles_leg_id)
{
  forwardKinematics(model, data, q);
  updateFramePlacements(model, data);
```

```
  Eigen::Vector3d trans_B_0 = abd_positions_base_frame.col(leg_id);
  Eigen::Matrix3d rot_B_0 = Eigen::MatrixXd::Identity(3, 3);

  SE3 transformation_B_0(rot_B_0, trans_B_0);
  SE3 transformation_0_B = transformation_B_0.inverse();

  Vector3d foot_local_position_abd_frame =
      transformation_0_B.act(foot_local_position_base_frame);

  InverseKinematics(foot_local_position_abd_frame, leg_id,
                    joint_angles_leg_id);

  joint_ids_leg_id =
      Vector3d(motors_per_leg * leg_id, motors_per_leg * leg_id + 1,
               motors_per_leg * leg_id + 2);
}
```

Then, the necessary inverse kinematics algorithm is executed by calling the function **InverseKinematics(.)** which is illustrated in the following code snippet:

```
void Robot::InverseKinematics(Vector3d& foot_local_position_abd_frame,
                              int& leg_id, Vector3d& joint_angles_leg_id)
{
  double xE = foot_local_position_hip_frame(0);
  double yE = foot_local_position_hip_frame(1);
  double zE = foot_local_position_hip_frame(2);

  double s1 = sqrt(pow(xE, 2.0) + pow(zE, 2.0) - pow(l0, 2.0));

  double cos_th2 =
      ((pow(s1, 2.0) + pow(yE, 2.0) - pow(l1, 2.0) - pow(l2, 2.0)) /
       (2 * l1 * l2));
  double sin_th2_plus = sqrt(1 - cos_th2 * cos_th2);
  // double sin_th2_minus = -sqrt(1 - cos_th2 * cos_th2);
  double th2 = atan2(sin_th2_plus, cos_th2) + gamma;
  double th1 =
      atan2(yE, s1) - atan2(l2 * sin_th2_plus, l1 + l2 * cos_th2);

  double th0;

  if (leg_id == FR || leg_id == HR)
  {
    th0 = atan2(-zE, xE) - atan2(s1, l0);
  }
  else if (leg_id == FL || leg_id == HL)
  {
    th0 = -(atan2(-zE, -xE) - atan2(s1, l0));
  }

  joint_angles_leg_id = Vector3d(th0, th1, th2);
  //  std::cout << "Here is joint_angles_leg_id:\n"
  //            << leg_id << ":\n"
  //            << joint_angles_leg_id << std::endl;
}
```

Moreover, the position of the feet of the robot (end-effectors) relative to the robot's body frame must be available, for a given model configuration. This computation is executed by

calling the function **FootPositionsInBaseFrame(.)**. This function is presented in the code snippet below.

```
void Robot::FootPositionsInBaseFrame(
    Eigen::VectorXd& q, Matrix3x4& foot_positions_in_base_frame)
{
  forwardKinematics(model, data, q);
  updateFramePlacements(model, data);

  Model::Index parent_idx = model.existJointName("root_joint") ?
                                model.getJointId("root_joint") :
                                (Model::Index)(model.njoints - 1);
  Eigen::Vector3d trans_B = data.oMi[parent_idx].translation();
  Eigen::Matrix3d rot_B = data.oMi[parent_idx].rotation();

  SE3 transformation_I_B(rot_B, trans_B);
  SE3 transformation_B_I = transformation_I_B.inverse();

  Eigen::Vector3d pos_B_FL_fc;
  pos_B_FL_fc.setZero();
  Eigen::Vector3d pos_B_FR_fc;
  pos_B_FR_fc.setZero();
  Eigen::Vector3d pos_B_HL_fc;
  pos_B_HL_fc.setZero();
  Eigen::Vector3d pos_B_HR_fc;
  pos_B_HR_fc.setZero();

  Model::Index frame_idx;
  Eigen::Vector3d pos_I_fc;
  pos_I_fc.setZero();

  ////////////////////////////////////////////////////////////////////////

  frame_idx = model.getFrameId("FL_joint_knee_foot_frame");

  pos_I_fc = data.oMf[frame_idx].translation();

  // std::cout << "Here is pos_I_FL_fc:\n" << pos_I_fc << std::endl;

  pos_B_FL_fc = transformation_B_I.act(pos_I_fc);

  // std::cout << "Here is pos_B_FL_fc:\n" << pos_B_FL_fc << std::endl;

  ////////////////////////////////////////////////////////////////////////

  frame_idx = model.getFrameId("FR_joint_knee_foot_frame");

  pos_I_fc = data.oMf[frame_idx].translation();

  // std::cout << "Here is pos_I_FR_fc:\n" << pos_I_fc << std::endl;

  pos_B_FR_fc = transformation_B_I.act(pos_I_fc);

  // std::cout << "Here is pos_B_FR_fc:\n" << pos_B_FR_fc << std::endl;

  ////////////////////////////////////////////////////////////////////////

  frame_idx = model.getFrameId("HL_joint_knee_foot_frame");

  pos_I_fc = data.oMf[frame_idx].translation();
```

```
  // std::cout << "Here is pos_I_HL_fc:\n" << pos_I_fc << std::endl;

  pos_B_HL_fc = transformation_B_I.act(pos_I_fc);

  // std::cout << "Here is pos_B_HL_fc:\n" << pos_B_HL_fc << std::endl;

  ////////////////////////////////////////////////////////////////////////

  frame_idx = model.getFrameId("HR_joint_knee_foot_frame");

  pos_I_fc = data.oMf[frame_idx].translation();

  // std::cout << "Here is pos_I_HR_fc:\n" << pos_I_fc << std::endl;

  pos_B_HR_fc = transformation_B_I.act(pos_I_fc);

  // std::cout << "Here is pos_B_HR_fc:\n" << pos_B_HR_fc << std::endl;

  ////////////////////////////////////////////////////////////////////////

  foot_positions_in_base_frame.col(0) = pos_B_FL_fc;
  foot_positions_in_base_frame.col(1) = pos_B_FR_fc;
  foot_positions_in_base_frame.col(2) = pos_B_HL_fc;
  foot_positions_in_base_frame.col(3) = pos_B_HR_fc;
}
```

However, it is possible that only the position of a specific foot, relative to the robot's body frame for a given model configuration, will be needed and not the position of all four feet. This computation is executed by calling the function **FootPositionsLegidInBaseFrame(.)**. It follows the same procedure that the function **FootPositionsInBaseFrame(.)** follows, but it requires to specify the index of the leg whose foot position must be calculated. This function is presented in the code snippet below.

```
void Robot::FootPositionsLegidInBaseFrame(
    int& leg_id, Eigen::VectorXd& q,
    Vector3d& foot_positions_in_base_frame)
{
  forwardKinematics(model, data, q);
  updateFramePlacements(model, data);

  Model::Index parent_idx = model.existJointName("root_joint") ?
                                model.getJointId("root_joint") :
                                (Model::Index)(model.njoints - 1);
  Eigen::Vector3d trans_B = data.oMi[parent_idx].translation();
  Eigen::Matrix3d rot_B = data.oMi[parent_idx].rotation();

  SE3 transformation_I_B(rot_B, trans_B);
  SE3 transformation_B_I = transformation_I_B.inverse();

  Eigen::Vector3d pos_B_fc;

  if (leg_id == FL)
  {
    Model::Index frame_idx =
        model.getFrameId("FL_joint_knee_foot_frame");

    Eigen::Vector3d pos_I_fc = data.oMf[frame_idx].translation();
```

```
    pos_B_fc = transformation_B_I.act(pos_I_fc);

    std::cout << "Here is pos_B_FL_fc:\n" << pos_B_fc << std::endl;
  }

  ////////////////////////////////////////////////////////////////////////

  else if (leg_id == FR)
  {
    Model::Index frame_idx =
        model.getFrameId("FR_joint_knee_foot_frame");

    Eigen::Vector3d pos_I_fc = data.oMf[frame_idx].translation();

    pos_B_fc = transformation_B_I.act(pos_I_fc);

    std::cout << "Here is pos_B_FL_fc:\n" << pos_B_fc << std::endl;
  }

  ////////////////////////////////////////////////////////////////////////

  else if (leg_id == HL)
  {
    Model::Index frame_idx =
        model.getFrameId("HL_joint_knee_foot_frame");

    Eigen::Vector3d pos_I_fc = data.oMf[frame_idx].translation();

    pos_B_fc = transformation_B_I.act(pos_I_fc);

    std::cout << "Here is pos_B_FL_fc:\n" << pos_B_fc << std::endl;
  }

  ////////////////////////////////////////////////////////////////////////

  else if (leg_id == HR)
  {
    Model::Index frame_idx =
        model.getFrameId("HR_joint_knee_foot_frame");

    Eigen::Vector3d pos_I_fc = data.oMf[frame_idx].translation();

    pos_B_fc = transformation_B_I.act(pos_I_fc);

    std::cout << "Here is pos_B_FL_fc:\n" << pos_B_fc << std::endl;
  }

  ////////////////////////////////////////////////////////////////////////

  foot_positions_in_base_frame = pos_B_fc;
}
```

Moreover, the position of the hip of a particular leg relative to the world frame must be available, for a given model configuration. This computation is executed by calling the function **HipPositionsLegidInWorldFrame(.)**. It requires specifying the index of the leg whose hip position must be calculated. This function is presented in the code snippet below.

```
void Robot::HipPositionsLegidInWorldFrame(
    int& leg_id, Eigen::VectorXd& q,
```

```
    Vector3d& hip_positions_in_world_frame)
{
  forwardKinematics(model, data, q);

  Eigen::Vector3d pos_I_hip;

  if (leg_id == FL)
  {
    Model::Index joint_idx = model.getJointId("FL_joint_hip");

    pos_I_hip = data.oMi[joint_idx].translation();

    // std::cout << "Here is pos_I_FL_hip:\n" << pos_I_hip << std::endl;
  }

  ////////////////////////////////////////////////////////////////////////

  else if (leg_id == FR)
  {
    Model::Index joint_idx = model.getJointId("FR_joint_hip");

    pos_I_hip = data.oMi[joint_idx].translation();

    // std::cout << "Here is pos_I_FR_fc:\n" << pos_I_hip << std::endl;
  }

  ////////////////////////////////////////////////////////////////////////

  else if (leg_id == HL)
  {
    Model::Index joint_idx = model.getJointId("HL_joint_hip");

    pos_I_hip = data.oMi[joint_idx].translation();

    // std::cout << "Here is pos_I_HL_fc:\n" << pos_I_hip << std::endl;
  }

  ////////////////////////////////////////////////////////////////////////

  else if (leg_id == HR)
  {
    Model::Index joint_idx = model.getJointId("HR_joint_hip");

    pos_I_hip = data.oMi[joint_idx].translation();

    // std::cout << "Here is pos_I_HR_fc:\n" << pos_I_hip << std::endl;
  }

  ////////////////////////////////////////////////////////////////////////

  hip_positions_in_world_frame = pos_I_hip;
}
```

Also, the position of the various frames of the robot w.r.t the world frame must be converted so that it is expressed w.r.t. the robot's body frame. This conversion is achieved by calling the function **ConvertPositionWorldToBaseFrame(.)** that can be viewed in the following code snippet.

```
void Robot::ConvertPositionWorldToBaseFrame(Eigen::VectorXd& q,
```

```
                                          Vector3d& pos_I,
                                          Vector3d& pos_B)
{
  forwardKinematics(model, data, q);
  updateFramePlacements(model, data);

  Model::Index parent_idx = model.existJointName("root_joint") ?
                            model.getJointId("root_joint") :
                            (Model::Index)(model.njoints - 1);
  Eigen::Vector3d trans_B = data.oMi[parent_idx].translation();
  Eigen::Matrix3d rot_B = data.oMi[parent_idx].rotation();

  SE3 transformation_I_B(rot_B, trans_B);
  SE3 transformation_B_I = transformation_I_B.inverse();

  pos_B = transformation_B_I.act(pos_I);
}
```

Finally, it is important to define the desired speed profile that the robot should track during the locomotion task. This speed profile determines the desired x,y - linear velocity and the current desired z - angular velocity as functions of time. This profile is designed as follows: some characteristic pairs of points along the profile are specified initially, and then linear interpolation is performed to decide the desired x,y - linear velocity and the current desired z - angular velocity at a given time instance. This linear interpolation is performed using a function of the linear algebra library Armadillo [166], that is called **interp1(.)**. If the current time surpasses the maximum time used to define the profile, then linear extrapolation is utilized. This line is defined using the last two pairs of points used to define the profile. The desired speed profile is created using the function **GenerateSpeedProfile(.)** that can be viewed in the following code snippet.

```
void Robot::GenerateSpeedProfile(double& time, Vector2d& desired_speed,
                                 double& desired_twisting_speed)
{
  double vx = nominal_speed_(0);
  double vy = nominal_speed_(1);
  double wz = nominal_twisting_speed_;

  arma::vec time_points = { 0, 5, 10, 15, 20, 25, 30 };
  arma::mat speed_points = { { 0, 0, 0, 0 },    { vx, vy, 0, wz },
                             { vx, vy, 0, wz }, { vx, vy, 0, wz },
                             { vx, vy, 0, wz }, { vx, vy, 0, wz },
                             { vx, vy, 0, wz } };

  arma::vec t = { time };
  arma::vec vx_t;
  arma::vec vy_t;
  arma::vec wz_t;

  interp1(time_points, speed_points.col(0), t, vx_t, "*linear", vx);
  interp1(time_points, speed_points.col(1), t, vy_t, "*linear", vy);
  interp1(time_points, speed_points.col(3), t, wz_t, "*linear", wz);

  std::cout << "Here is vx_t:\n" << vx_t << std::endl;
  std::cout << "Here is vy_t:\n" << vy_t << std::endl;
  std::cout << "Here is wz_t:\n" << wz_t << std::endl;

  desired_speed = Vector2d(vx_t(0), vy_t(0));
```

```
  desired_twisting_speed = wz_t(0);
}
```

Also, it is necessary for the swing leg controller to invert the foot Jacobian matrix of the swinging leg, as dictated by equation (4-19). This inversion of the Jacobian is accomplished by calling the function **sdlsInv(.)**, by performing singular value decomposition on the Jacobian matrix. In this way the selective damping least square inverse matrix can be computed. This function is presented in the code snippet below.

```
Eigen::MatrixXd Robot::sdlsInv(const Eigen::MatrixXd& jacobian)
{
  Eigen::JacobiSVD<Eigen::MatrixXd> svd(
      jacobian, Eigen::ComputeThinU | Eigen::ComputeThinV);

  Eigen::VectorXd sigval = svd.singularValues();
  Eigen::VectorXd sigval_inv = sigval;

  for (size_t i = 0; i < sigval.size(); i++)
  {
    sigval_inv(i) =
        std::min(1 / sigval(i), 1 / (1e-1 * sigval.maxCoeff()));
  }

  Eigen::MatrixXd jacobian_inv = svd.matrixV() *
                                 sigval_inv.asDiagonal() *
                                 svd.matrixU().transpose();

  return jacobian_inv;
}
```

4.4.1 State Estimator

For the control framework presented in this work no state estimator was utilized. Instead, physical quantities that are required by the controller to calculate the optimal control action were provided to it by either ROS sensors or by the physics engine itself. More specifically, the Gazebo model plugin, named *apply_action.cc*, interacts directly with the physics engine and has access to such physical quantities, with the help of some functions, and publishes them to certain topics. Then, the MPC controller that is implemented using a ROS node, subscribes to these topics and gets access to these quantities. These topics as well as the aforementioned Gazebo functions are described in this section. After each control loop iteration, the calculated optimal action gets published to the ROS topic */joint_commands* to get it applied to the model.

To be more precise, the controller needs the pose of the robot's body frame w.r.t. the world frame. Gazebo offers the function **WorldCoGPose()** that provides the x, y, z -position of the CoM of a certain rigid body w.r.t. the world frame, as **WorldCoGPose().X()**, **WorldCoGPose().Y()**, **WorldCoGPose().Z()**, and the orientation of the robot's body frame w.r.t. the world frame expressed using roll-pitch-yaw angles, as **WorldCoGPose().Roll()**, **WorldCoGPose().Pitch()**, **WorldCoGPose().Yaw()**. The body's pose is then published to the topic */body_pose*. Furthermore, the controller requires the linear velocity of the robot's body CoM w.r.t. the world frame. Gazebo offers the function **WorldCoGLinearVel()** that provides the x, y, z -linear velocity of the CoM of a certain rigid body w.r.t. the world frame. Additionally,

the controller requires the angular velocity of the robot's body w.r.t. the world frame. Gazebo offers the function **WorldAngularVel()** that provides the x, y, z -angular velocity of a certain rigid body w.r.t. the world frame. The body's velocity is then published to the topic */body_twist*. Moreover, the joint positions and velocities are needed by the controller. Gazebo offers the functions **Position()** and **GetVelocity(_index)** that provide the joint position and velocity respectively, where **_index** represents index that corresponds to the joint axis, that is equal to zero for 1-DoF joints. These values are then published to the topic */joint_states.* The controller needs to be aware of whether the feet of the robot are in contact with the ground or not. The ROS package provides contact force sensors that are attached to the four feet of the robot. These sensors measure the GRFs and publish their values to the topics */FL_Lower_Leg_state*, */FR_Lower_Leg_state*, */HL_Lower_Leg_state* and */HR_Lower_Leg_state*. Moreover, this plugin applies the control inputs (actions), which are joint torques calculated by the MPC controller, to the joints of the pendulum. It subscribes to the topic */joint_commands* to have access to them. These actions then are applied using the Gazebo function **SetForce(_index, _effort)** where **_effort** represents the torque that should be applied.

4.4.2 Gait Scheduler

The gait scheduler consists of the class **OpenLoopGaitGenerator**. The functions that belong to this class are responsible for deciding what are the desired states of the legs at each time instance, according to the selected gait and for the normalized phase of the entire gait cycle and of the stance or swing phase of each leg, for the current time instance. The header file *openloop_gait_generator.h* contains the definition of the class, that includes declaration of its member variables and functions. The implementation of the class is located in the *openloop_gait_generator.cpp* file.

First of all, the class **OpenLoopGaitGenerator** should be initialized. The parameters necessary to initialize this class and thus define the desired gait are the following: the stance duration, the duty factor, the initial leg state, the initial leg phase, and the contact detection phase threshold. The first four parameters were explained in the chapter 4.3.2. The contact detection phase threshold parameter serves the following purpose: the state of each leg gets updated after taking into account the measurements provided by the contact sensors when the current normalized phase is greater than this threshold. This is essential to remove false positives in contact detection when phase switches. For example, a swing foot at the beginning of the gait cycle might be still on the ground. The function that performs this initialization is shown in the following code snippet.

```
OpenLoopGaitGenerator::OpenLoopGaitGenerator(
    Robot* robot, Vector4d stance_duration, Vector4d duty_factor,
    Vector4d initial_leg_state, Vector4d initial_leg_phase,
    double contact_detection_phase_threshold)
{
  this->robot = robot;
  this->stance_duration = stance_duration;
  this->duty_factor = duty_factor;
  this->initial_leg_state = initial_leg_state;
  this->initial_leg_phase = initial_leg_phase;
  this->contact_detection_phase_threshold =
      contact_detection_phase_threshold;
```

```
  // The ratio in cycle is duty factor if initial state of the leg is
  // STANCE, and 1 - duty_factory if the initial state of the leg is
  // SWING.

  for (int leg_id = 0; leg_id < initial_leg_state.size(); leg_id++)
  {
    if (duty_factor(leg_id) == 0.0)
    {
      this->swing_duration(leg_id) = 1e3;
      initial_leg_state(leg_id) = LegState::USERDEFINED_SWING;
      next_leg_state(leg_id) = LegState::USERDEFINED_SWING;
      printf("Leg [%i] is userdefined leg!\n", leg_id);
    }
    else
    {
      this->swing_duration(leg_id) =
          stance_duration(leg_id) / duty_factor(leg_id) -
          stance_duration(leg_id);

      if (initial_leg_state(leg_id) == LegState::SWING)
      {
        initial_state_ratio_in_cycle(leg_id) = 1 - duty_factor(leg_id);
        next_leg_state(leg_id) = LegState::STANCE;
      }
      else
      {
        initial_state_ratio_in_cycle(leg_id) = duty_factor(leg_id);
        next_leg_state(leg_id) = LegState::SWING;
      }
    }
  }

  this->Reset(0);
}
```

Secondly, the function **Reset(.)** is also a member of this class. It needs to be called to reset the gait parameters. This function is shown in the following code snippet.

```
void OpenLoopGaitGenerator::Reset(double current_time)
{
  normalized_phase = Eigen ::VectorXd ::Zero(num_legs);
  leg_state = initial_leg_state;
  desired_leg_state = initial_leg_state;
}
```

Finally, the function **Update(.)** needs to be called at every control loop iteration so that the various gait parameters get updated. This function is shown in the following code snippet. It is important to note that apart from the swing and stance state, two more possible leg states are defined. These are early contact and lose contact state. The actual leg state can be swing, stance, early contact and lose contact while the desired leg state can be only swing or stance. If the desired state of a leg is swing but the force sensors measurements indicate that the foot is in contact with the ground, then the state of that leg will be early contact. However, if the desired state of a leg in stance but the force sensors measurements indicate that the foot is not in contact with the ground, then the state of that leg will be lose contact. This distinction is taken into consideration by the function below.

```
void OpenLoopGaitGenerator::Update(double current_time,
                                   Vector4d& contact_state,
                                   VectorNm& contact_force)
{
  std::cout << "gait_generator_update:\n" << std::endl;

  // Called at each control step

  // The phase within the full swing/stance cycle is used to determine
  // if a swing/stance switch occurs for a leg. The threshold value is
  // the "initial_state_ratio_in_cycle" If the current phase is less than
  // the initial state ratio, then the leg is either in the initial state
  // or has switched back after one or more full cycles.

  for (int leg_id = 0; leg_id < num_legs; leg_id++)
  {
    if (initial_leg_state(leg_id) == LegState::USERDEFINED_SWING)
    {
      desired_leg_state(leg_id) = initial_leg_state(leg_id);
      leg_state(leg_id) = desired_leg_state(leg_id);
      continue;
    }

    double full_cycle_period =
        stance_duration(leg_id) / duty_factor(leg_id);

    // To account for the non-zero initial phase, we offset the time
    // duration with the effect time contribution from the initial leg
    // phase.

    double augmented_time =
        current_time + initial_leg_phase(leg_id) * full_cycle_period;
    double phase_in_full_cycle =
        fmod(augmented_time, full_cycle_period) / full_cycle_period;

    double ratio = initial_state_ratio_in_cycle(leg_id);
    if (phase_in_full_cycle < ratio)
    {
      desired_leg_state(leg_id) = initial_leg_state(leg_id);
      normalized_phase(leg_id) = phase_in_full_cycle / ratio;
    }
    else
    {
      // A phase switch happens for this leg.
      desired_leg_state(leg_id) = next_leg_state(leg_id);
      normalized_phase(leg_id) =
          (phase_in_full_cycle - ratio) / (1 - ratio);
    }

    leg_state(leg_id) = desired_leg_state(leg_id);

    // No contact detection at the beginning of each SWING/STANCE phase.
    if (normalized_phase(leg_id) < contact_detection_phase_threshold)
    {
      continue;
    }

    if (leg_state(leg_id) == LegState::SWING &&
        contact_state(leg_id) == 1.0 &&
        contact_force(3 * leg_id + 2) > robot->robot_mass * kGravity / 4)
    {
```

```
      printf("Early touch down detected \n");
      leg_state(leg_id) = LegState::EARLY_CONTACT;
    }
    if (leg_state(leg_id) == LegState::STANCE &&
        contact_state(leg_id) == 0.0 &&
        contact_force(3 * leg_id + 2) < robot->robot_mass * kGravity / 4)
    {
      printf("Lost contact detected \n");
      leg_state(leg_id) = LegState::LOSE_CONTACT;
    }
  }

  std::cout << "Here is leg_state :\n"
            << leg_state.transpose() << std::endl;
}
```

4.4.3 Swing Leg Controller

The swing leg controller consists of the class **RaibertSwingLegController**. The functions that belong to this class are responsible for selecting the desired foothold positions, according to the footstep planner of equation (4-58), and for calculating the desired configuration of the swinging legs, for the current phase of the gait cycle. The header file *raibert_swing_leg_controller.h* contains the definition of the class, that includes declaration of its member variables and functions. The implementation of the class is located in the *raibert_swing_leg_controller.cpp* file.

Firstly, the function **GenSwingFootTrajectory(.)** is defined as a member function of the class. It generates the trajectory of the swing leg using an ellipse. To be more precise, it returns desired foot position and acceleration at the current phase, in the cartesian space, given the foot's position at the beginning of the swing cycle, the foot's position at the middle of the swing cycle, that is determined by the maximum clearance that the foot should have w.r.t the ground, and the foot's desired position at the end of the swing cycle. This function is presented in the following code snippet.

```
void RaibertSwingLegController::GenSwingFootTrajectory(
    double& swing_period, double& max_clearance, double& input_phase,
    Vector3d& start_pos, Vector3d& end_pos, Vector3d& desired_pos,
    Vector3d& desired_acc)
{
  // Generates the swing trajectory using an ellipse.

  double phase = input_phase;

  double mid_pos_z;

  mid_pos_z = 0.5 * (start_pos(2) + end_pos(2)) + max_clearance;
  // mid_pos_z = start_pos(2) + max_clearance;

  GenEllipse(swing_period, phase, start_pos, mid_pos_z, end_pos,
             desired_pos, desired_acc);

  // std::cout << "Here is desired_acc:\n" << desired_acc << std::endl;
}
```

The points along the elliptical trajectory are generated by the function **GenEllipse(.)**. This function computes the coefficients $\mathbf{c},\mathbf{u},\mathbf{v}$ of the ellipse using the equations (4-14), (4-15) and given the foot's position at the beginning of the swing cycle, the maximum clearance and the foot's desired position at the end of the swing cycle. Then, the function computes the desired foot position and the desired foot acceleration for the given phase in the swing cycle. The function that performs this initialization is presented in the following code snippet.

```
void RaibertSwingLegController::GenEllipse(
    double& swing_period, double& phase, Vector3d& start, double& mid,
    Vector3d& end, Vector3d& pos_des, Vector3d& pos_des_ddot)
{
  // Gets a point on an ellipse.

  Vector3d coef_c, coef_u, coef_v;
  Vector3d middle = 0.5 * (start + end);
  middle(2) = mid;

  coef_c = 0.5 * (start + end);
  // coef_c(2) = start(2);
  // coef_c(2) = end(2);

  coef_u = start - coef_c;
  // coef_u = - end_pos + coef_c;

  coef_v = middle - coef_c;

  pos_des =
      coef_c + coef_u * cos(M_PI * phase) + coef_v * sin(M_PI * phase);

  // std::cout << "Here is pos_des :\n" << pos_des << std::endl;

  pos_des_ddot = -coef_u *
                     ((pow(M_PI, 2.0)) / (pow(swing_period, 2.0))) *
                     cos(M_PI * phase) -
                 coef_v * ((pow(M_PI, 2.0)) / (pow(swing_period, 2.0))) *
                     sin(M_PI * phase);

  // std::cout << "Here is pos_des_ddot :\n" << pos_des_ddot <<
  // std::endl;
}
```

First of all, the class **RaibertSwingLegController** should be initialized. The parameters necessary to initialize this class are the following: an instance of the **Robot** class, an instance of the class **OpenLoopGaitGenerator**, the initial desired x,y - linear velocity of the robot, the initial desired z - angular velocity of the robot, the desired locomotion height and the maximum clearance. This function is shown in the following code snippet.

```
RaibertSwingLegController::RaibertSwingLegController(
    Robot* robot, OpenLoopGaitGenerator* gaitGenerator,
    Vector2d desired_speed, double desired_twisting_speed,
    double desired_height, double max_clearance)
{
  this->robot = robot;
  this->gaitGenerator = gaitGenerator;
  this->desired_speed = Vector3d(desired_speed(0), desired_speed(1), 0);
  this->desired_twisting_speed = desired_twisting_speed;
```

```
  this->desired_height = Vector3d(0.0, 0.0, desired_height);
  this->max_clearance = max_clearance;

  last_leg_state = gaitGenerator->desired_leg_state;

  foot_position_ddot_des.setZero();

  joint_angles.resize(num_motors, 2);
  joint_angles.setZero();
  joint_angles.col(1) << LegId::FL, LegId::FL, LegId::FL, LegId::FR,
      LegId::FR, LegId::FR, LegId::HL, LegId::HL, LegId::HL, LegId::HR,
      LegId::HR, LegId::HR;
}
```

Secondly, the function **Reset(.)** is also a member of this class. It needs to be called to reset the swing leg controller parameters. This function is shown in the following code snippet.

```
void RaibertSwingLegController::Reset(double current_time,
                                      Eigen::VectorXd& q)
{
  std::cout << "swing_leg_controller_reset:\n" << std::endl;

  last_leg_state = gaitGenerator->desired_leg_state;

  Matrix3x4 foot_positions_in_base_frame;
  foot_positions_in_base_frame.setZero();
  robot->FootPositionsInBaseFrame(q, foot_positions_in_base_frame);

  phase_switch_foot_local_position = foot_positions_in_base_frame;
}
```

Moreover, the function **Update(.)** needs to be called at every control loop iteration so that the swing leg controller parameters get updated. To be more specific, it detects if a phase switch has occurred for each leg, facilitating the controller in remembering the feet positions at the beginning of the swing phase. These positions are stored in the phase switch foot local position variable. This function is shown in the following code snippet.

```
void RaibertSwingLegController::Update(double current_time,
                                       Eigen::VectorXd& q)
{
  std::cout << "swing_leg_controller_update:\n" << std::endl;

  // Called at each control step

  Vector4d new_leg_state = gaitGenerator->desired_leg_state;

  // Detects phase switch for each leg so we can remember the feet
  // position at the beginning of the swing phase.

  Matrix3x4 foot_positions_in_base_frame;
  Vector3d foot_positions_in_base_frame_leg_id;

  for (int leg_id = 0; leg_id < new_leg_state.size(); leg_id++)
  {
    if (new_leg_state(leg_id) == LegState::SWING &&
        new_leg_state(leg_id) != last_leg_state(leg_id))
    {
    robot->FootPositionsLegidInBaseFrame(
```

```
        leg_id, q, foot_positions_in_base_frame_leg_id);
    // std::cout << "Here is foot_positions_in_base_frame_leg_id :\n"
    //           << foot_positions_in_base_frame_leg_id << std::endl;

    phase_switch_foot_local_position.col(leg_id) =
        foot_positions_in_base_frame_leg_id;
    }
  }

  // std::cout << "Here is phase_switch_foot_local_position :\n"
  //           << phase_switch_foot_local_position << std::endl;
  last_leg_state = new_leg_state;
}
```

Also, the function **UpdateControlParameters(.)** needs to be called at every control loop iteration so that the current desired x, y - linear velocity of the robot, the current desired z - angular velocity of the robot get updated, according to the desired speed profile. This function is shown in the following code snippet.

```
void RaibertSwingLegController::UpdateControlParameters(
    Vector2d& linSpeed, double& angSpeed)
{
  desired_speed = Vector3d(linSpeed(0), linSpeed(1), 0);
  desired_twisting_speed = angSpeed;
}
```

Finally, the function **GetAction(.)** needs to be called at every control loop iteration to compute the actions corresponding to the swing legs. This function is shown in the following code snippet. The computation proceeds only if the leg that is supposed to be in swing phase is not in contact with the ground. Therefore, checks are performed to verify if the desired leg state is not stance, or if the leg state is not early contact, for a specific leg. If they are not, then the computation proceeds. In that case, the hip x, y - linear velocity and the desired hip x, y - linear velocity get calculated. Afterwards, the target foothold position gets computed using the footstep planner of equation (4-58). Necessary for this computation are the hip linear velocities $\dot{\mathbf{p}}_{\mathbf{h},i}, \dot{\mathbf{p}}_{\mathbf{h},i}^{cmd}$ that are calculated beforehand. Then, the desired foot position for the given swing cycle phase is computed by calling the function **GenSwingFootTrajectory(.)**. Given the desired foot position, the leg's joint angles that correspond to that position can be computed by calling the function **ComputeJointAnglesFromFootLocalPosition(.)**. This process is repeated for every swinging leg. Finally, these desired joint angles are stored in a matrix alongside the gains of the PD controller that correspond to that joint, for every swinging leg.

```
void RaibertSwingLegController::GetAction(
    double current_time, Eigen::VectorXd& q, Vector3d& com_position,
    Vector3d& body_orientation_rpy, Vector3d& com_velocity,
    Vector3d& body_angular_velocity)
{
  std::cout << "swing_leg_controller_get_action:\n" << std::endl;

  // std::cout << "Here is normalized_phase :\n"
  //           << gaitGenerator->normalized_phase << std::endl;

  Vector3d com_velocity_body_frame;
  robot->BaseVelocityInBodyFrame(body_orientation_rpy, com_velocity,
```

```
                                com_velocity_body_frame);
com_velocity_body_frame(2) = 0.0;

Vector3d desired_speed_body_frame;
robot->BaseVelocityInBodyFrame(body_orientation_rpy, desired_speed,
                                desired_speed_body_frame);
desired_speed_body_frame(2) = 0.0;

Vector3d base_rpy_rate;
robot->BaseRollPitchYawRate(body_orientation_rpy,
                            body_angular_velocity, base_rpy_rate);

double roll_dot = base_rpy_rate(0);
double pitch_dot = base_rpy_rate(1);
double yaw_dot = base_rpy_rate(2);

joint_angles.col(0).setZero();
foot_position_ddot_des.setZero();
action.setZero();

for (int leg_id = 0; leg_id < gaitGenerator->leg_state.size();
     leg_id++)
{
  // std::cout << "Now doing leg " << leg_id << std::endl;

  double c_flag =
      gaitGenerator->desired_leg_state(leg_id) == LegState::STANCE ||
      gaitGenerator->leg_state(leg_id) == LegState::EARLY_CONTACT;

  if (c_flag == 1)
  {
    // std::cout << "The leg " << leg_id
    //           << " is in STANCE or EARLY_CONTACT or "
    //             "USERDEFINED_SWING \n"
    //           << std::endl;
    continue;
  }

  // All calculation is in the body frame.
  Vector3d hip_offset = robot->hip_positions_base_frame.col(leg_id);

  Vector3d r_cross_omega = skew(hip_offset) * base_rpy_rate;
  r_cross_omega(2) = 0.0;

  Vector3d hip_horizontal_velocity =
      com_velocity_body_frame + r_cross_omega;
  Vector3d target_hip_horizontal_velocity =
      desired_speed +
      skew(hip_offset) * Vector3d(0.0, 0.0, desired_twisting_speed);

  // std::cout << "Here is hip_horizontal_velocity :\n"
  //           << hip_horizontal_velocity << std::endl;

  Vector3d hip_positions_world;
  robot->HipPositionsLegidInWorldFrame(leg_id, q, hip_positions_world);
  hip_positions_world(2) = 0.0;
  Vector3d hip_positions_base;
  robot->ConvertPositionWorldtoBaseFrame(q, hip_positions_world,
                                         hip_positions_base);

  std::cout << "Here is leg_id :\n" << leg_id << std::endl;
```

```
    Vector3d foot_target_position =
        (0.5 * gaitGenerator->stance_duration(leg_id) *
             target_hip_horizontal_velocity -
         sqrt(desired_height(2) / kGravity) *
             (target_hip_horizontal_velocity -
              hip_horizontal_velocity)) +
        (desired_height(2) / kGravity) * skew(hip_horizontal_velocity) *
            (Vector3d(0.0, 0.0, desired_twisting_speed)) +
        hip_positions_base;
    //- Vector3d( 0.0, 0.0, com_position(2) ) + Vector3d( hip_offset(0),
    // hip_offset(1), 0.0 );

    // std::cout << "Here is foot_target_position :\n"
    //           << foot_target_position << std::endl;

    Vector3d phase_switch_foot_local_position_leg_id =
        phase_switch_foot_local_position.col(leg_id);

    Vector3d foot_position;
    double normalized_phase_leg_id =
        gaitGenerator->normalized_phase(leg_id);

    // Elliptic Curve
    GenSwingFootTrajectory(gaitGenerator->swing_duration(leg_id),
                           max_clearance, normalized_phase_leg_id,
                           phase_switch_foot_local_position_leg_id,
                           foot_target_position, foot_position,
                           foot_position_ddot_des_leg_id);
    foot_position_ddot_des.col(leg_id) = foot_position_ddot_des_leg_id;

    // std::cout << "Here is foot_position :\n"
    //           << foot_position << std::endl;

    Vector3d joint_angles_leg_id;
    Vector3d joint_ids_leg_id;

    // compute joint position
    robot->ComputeJointAnglesFromFootLocalPosition(
        leg_id, q, foot_position, joint_ids_leg_id, joint_angles_leg_id);

    // Update the stored joint angles as needed.
    for (int i = 0; i < joint_ids_leg_id.size(); i++)
    {
      double joint_id = joint_ids_leg_id(i);
      double joint_angle = joint_angles_leg_id(i);
      joint_angles.row(joint_id) << joint_angle, leg_id;
    }
  }

  for (int joint_id = 0; joint_id < joint_angles.rows(); joint_id++)
  {
    double joint_angle_leg_id = joint_angles(joint_id, 0);
    int leg_id = joint_angles(joint_id, 1);

    double c_flag =
        gaitGenerator->desired_leg_state(leg_id) == LegState::STANCE ||
        gaitGenerator->leg_state(leg_id) == LegState::EARLY_CONTACT;
    if (c_flag == 0)
    {
      // This is a hybrid action for PD control.
      action.row(joint_id) << joint_angle_leg_id, robot->Kps(joint_id),
```

```
            0.0, robot->Kds(joint_id), 0.0;
        // std::cout << "Here is swing action :\n" << action <<
        // std::endl;
      }
    }

    std::cout << "Here is swing action :\n" << action << std::endl;
}
```

4.4.4 Stance Leg Controller

The stance leg controller consists of the classes **TorqueStanceLegController** and **ConvexMpcDense**. The functions that belong to the first class are responsible for providing all the necessary high level parameters and variables needed to set up the QP of equation (4-53). They also map the predicted GRFs that constitute the QP solution into the desired motor torques. The functions that belong to the second class are responsible for setting up and solving that QP, to compute the predicted GRFs that track the desired position and velocity of the robot. The header file *torque_stance_leg_controller.h* contains the definition of the class, that includes declaration of its member variables and functions. The implementation of the class is located in the *torque_stance_leg_controller.cpp* file. Also, the header file *mpc_osqp_dense.h* contains the definition of the class, that includes declaration of its member variables and functions. The implementation of the class is located in the *mpc_osqp_dense.cpp* file.

First of all, the class **TorqueStanceLegController** should be initialized. The parameters necessary to initialize this class are the following: an instance of the **Robot** class, an instance of the class **OpenLoopGaitGenerator**, an instance of the class **RaibertSwingLegController**, the reference robot's body yaw angle, the foot friction coefficients, the number of the timesteps into which the horizon was divided, the timestep size and the minimum and maximum normal contact force limits. The function that performs this initialization is shown in the following code snippet. Moreover, this function also calculates the values of the matrices $\mathbf{Q}_{qp}, \mathbf{R}_{qp}$ and initializes the class **ConvexMpcDense.**

```
TorqueStanceLegController::TorqueStanceLegController(
    Robot* robot, OpenLoopGaitGenerator* gaitGenerator,
    RaibertSwingLegController* swingLegController, double body_yaw_ref,
    Vector4d friction_coeffs, int planning_horizon_steps,
    double planning_timestep, VectorNx state_weights,
    VectorNu input_weights, double fz_max, double fz_min)
{
  this->robot = robot;
  this->gaitGenerator = gaitGenerator;
  this->swingLegController = swingLegController;
  this->body_yaw_ref = body_yaw_ref;
  this->foot_friction_coeffs = friction_coeffs;
  this->planning_horizon_steps = planning_horizon_steps;
  this->planning_timestep = planning_timestep;
  this->fz_max = fz_max;
  this->fz_min = fz_min;

  desired_speed = swingLegController->desired_speed;
  desired_twisting_speed = swingLegController->desired_twisting_speed;
  desired_height = swingLegController->desired_height;
```

```
  Q_qp.setZero();
  R_qp.setZero();

  Eigen::Matrix<double, Nx * Nh, 1> q_weights_mpc;
  Eigen::Matrix<double, Nu * Nh, 1> r_weights_mpc;

  for (int i = 0; i < Nh; ++i)
  {
    q_weights_mpc.segment(i * Nx, Nx) = state_weights;
  }
  Q_qp.diagonal() = 2 * q_weights_mpc;

  for (int i = 0; i < Nh; ++i)
  {
    r_weights_mpc.segment(i * Nu, Nu) = input_weights;
  }
  R_qp.diagonal() = 2 * r_weights_mpc;

  convexMPC = new ConvexMPCDense(
      robot, body_yaw_ref, friction_coeffs, planning_horizon_steps,
      planning_timestep, Q_qp, R_qp, fz_max, fz_min);
}
```

Also, the function **UpdateControlParameters(.)** needs to be called at every control loop iteration so that the current desired x,y - linear velocity of the robot, the current desired z - angular velocity of the robot get updated, according to the desired speed profile. This function is shown in the following code snippet.

```
void TorqueStanceLegController::UpdateControlParameters(
    Vector2d& linSpeed, double& angSpeed)
{
  desired_speed = Vector3d(linSpeed(0), linSpeed(1), 0);
  desired_twisting_speed = angSpeed;
}
```

Finally, the function **GetAction(.)** needs to be called at every control loop iteration to compute the actions corresponding to the stance legs. This function is shown in the following code snippet. The computation proceeds only if the leg that is supposed to be in stance phase is in contact with the ground. Therefore, checks are performed to verify if the desired leg state is stance, or if the leg state is lose contact, for a specific leg. If they are, then the computation proceeds. This information is of paramount importance for the QP setup, since the solver must be aware of which feet are in contact with the ground or not. The QP setup also requires the foot positions w.r.t the robot's body frame. Additionally, for the MPC computations, a body yaw aligned world frame is utilized. To be more precise, this frame's origin coincides with the world frame's origin, its z - axis is aligned with the world frames z - axis and its x,y - axes are aligned with the robot's body frame x,y - axes, respectively. Thus, the state vector is being expressed in this frame. In case were the desired z - angular velocity of the robot is zero, then robot's body yaw angle, w.r.t. the world frame, must be kept constant and thus this value is the reference yaw angle. In case were the desired z - angular velocity of the robot is non zero, then the yaw angle from the previous control loop iteration is utilized to formulate the initial robot state. Given these higher-level parameters, the QP can be solved by calling the member function of the class **ConvexMPCDense**, **ComputeContactForces(.)**. Then the predicted GRFs can be mapped to motor torques using the relationship of equation (4-57).

```
void TorqueStanceLegController::GetAction(
    double current_time, Eigen::VectorXd& q, Eigen::VectorXd& q_dot,
    Vector3d& com_position, Vector3d& body_orientation_rpy,
    Vector3d& com_velocity, Vector3d& body_angular_velocity)
{
  std::cout << "stance_leg_controller_get_action:\n" << std::endl;

  // Computes the torque for stance legs

  Vector3d desired_com_position = Vector3d(0.0, 0.0, desired_height(2));
  Vector3d desired_com_velocity =
      Vector3d(desired_speed(0), desired_speed(1), 0.0);
  Vector3d desired_com_roll_pitch_yaw = Vector3d(0.0, 0.0, 0.0);
  Vector3d desired_com_angular_velocity =
      Vector3d(0.0, 0.0, desired_twisting_speed);

  VectorXd foot_contact_state(gaitGenerator->desired_leg_state.size());

  for (int i = 0; i < gaitGenerator->desired_leg_state.size(); ++i)
  {
    foot_contact_state(i) =
        (gaitGenerator->desired_leg_state(i) == LegState::STANCE ||
         gaitGenerator->leg_state(i) == LegState::EARLY_CONTACT);
  }

  // We use the body yaw aligned world frame for MPC computation.

  Vector3d com_velocity_body_frame;
  robot->BaseVelocityInBodyFrame(body_orientation_rpy, com_velocity,
                                 com_velocity_body_frame);

  // Angular velocity in the yaw aligned world frame is actually
  // different from rpy rate. We use it here as a simple approximation.
  Vector3d base_rpy_rate;
  robot->BaseRollPitchYawRate(body_orientation_rpy,
                              body_angular_velocity, base_rpy_rate);

  Matrix3d rotation =
      rpy::rpyToMatrix(0.0, 0.0,
                       ((desired_com_angular_velocity(2) == 0.0) ?
                            (body_yaw_ref) :
                            (convexMPC->body_previous_yaw)));
  Vector3d com_position_by = rotation_.transpose() * com_position;

  Vector3d body_rpy = body_orientation_rpy;
  double body_yaw = body_orientation_rpy(2);
  body_rpy(2) = 0.0;

  // std::cout << "Here is body_orientation_rpy:\n"
  //           << body_orientation_rpy << std::endl;

  Matrix3x4 foot_positions_base_frame;
  foot_positions_base_frame.setZero();
  robot->FootPositionsInBaseFrame(q, foot_positions_base_frame);

  convexMPC->ComputeContactForces(
      current_time, body_yaw, body_yaw_ref, foot_contact_state,
      foot_positions_base_frame, com_position_by,
      com_velocity_body_frame, body_rpy, base_rpy_rate,
      desired_com_position, desired_com_velocity,
```

```
      desired_com_roll_pitch_yaw, desired_com_angular_velocity);

  Matrix3x4 contact_forces;
  for (int i = 0; i < num_legs; i++)
  {
    contact_forces.col(i) =
        convexMPC->predicted_contact_forces.segment<k3Dim>(i * k3Dim);
  }

  action.setZero();
  VectorNm motor_torques;
  motor_torques.setZero();

  Eigen::VectorXd nle;
  robot->ComputeBiasTerms(q, q_dot, nle);

  for (int leg_id = 0; leg_id < num_legs; leg_id++)
  {
    Vector3d force = contact_forces.col(leg_id);

    // While "LOSE CONTACT" is useful in simulation, in real environment
    // it's susceptible to sensor noise.
    if (gaitGenerator->leg_state(leg_id) == LegState::LOSE_CONTACT)
    {
      std::cout << "The leg " << leg_id << " is in LOSE_CONTACT \n"
                << std::endl;
      force << 0.0, 0.0, 0.0;
    }

    // std::cout << "Here is leg_id:\n" << leg_id << std::endl;

    robot->MapContactForceToJointTorques(leg_id, q, force, nle,
                                         motor_torques);

    // std::cout << "Here is motor_torques:\n" << motor_torques <<
    // std::endl;

    Vector3i joint_ids;
    joint_ids << motors_per_leg * leg_id, motors_per_leg * leg_id + 1,
        motors_per_leg * leg_id + 2;

    for (int i = 0; i < joint_ids.size(); i++)
    {
      int joint_id = joint_ids(i);
      double torque = motor_torques(joint_id);

      action.row(joint_id) << 0.0, 0.0, 0.0, 0.0, torque;
    }
  }

  std::cout << "Here is stance action:\n" << action << std::endl;
}
```

Now follows the description of the class **ConvexMPCDense**. First of all, the class **ConvexMPCDense** should be initialized. The parameters necessary to initialize this class are the following: an instance of the **Robot** class, the reference robot's body yaw angle in the previous control loop iteration, the foot friction coefficients, the number of the timesteps into which the horizon was divided, the timestep size, the matrices $\mathbf{Q}_{qp}, \mathbf{R}_{qp}$ and the minimum and

maximum normal contact force limits. The function that performs this initialization is shown in the following code snippet.

```
ConvexMPCDense::ConvexMPCDense(Robot* robot, double body_previous_yaw,
                               Vector4d friction_coeffs,
                               int planning_horizon_steps,
                               double planning_timestep,
                               MatrixNxNhNxNh Q_qp, MatrixNuNhNuNh R_qp,
                               double fz_max, double fz_min)
{
  this->robot = robot;
  this->body_previous_yaw = body_previous_yaw;
  this->foot_friction_coeffs = friction_coeffs;
  this->planning_horizon_steps = planning_horizon_steps;
  this->planning_timestep = planning_timestep;
  this->Q_qp = Q_qp;
  this->R_qp = R_qp;
  this->fz_max = fz_max;
  this->fz_min = fz_min;

  inv_robot_mass = 1 / robot->robot_mass;
  inv_robot_inertia = robot->robot_inertia.inverse();

  H.resize(Nu * Nh, Nu * Nh);
  Ac.resize(Nc * Nh, Nu * Nh);
}
```

The linear dynamics matrices $\mathbf{A}$,$\mathbf{B}$ defined by the equations (4-32), (4-33) are computed using the functions **CalculateAMatrix(.)** and **CalculateBMatrix(.)** respectively that are implemented in the following two code snippets.

```
void ConvexMPCDense::CalculateAMatrix(Vector3d& body_rpy,
                                      MatrixNxNx& A_mat)
{
  // The CoM dynamics can be written as:
  // x_dot = A x + B u
  // where x is the 13-dimensional state vector (r, p, y, x, y, z, r_dot,
  // p_dot, y_dot, vx, vy, vz, -g) constructed from the CoM
  // roll/pitch/yaw/position, and their first order derivatives. 'g' is
  // the gravity constant. u is the 3*num_legs -dimensional input vector
  // ( (fx, fy, fz) for each leg )

  // Construct A matrix (13x13) in the linearized CoM dynamics equation

  // We assume that the input rotation is in X->Y->Z order in the
  // extrinsic/fixed frame, or z->y'->x'' order in the intrinsic frame.

  double cos_yaw = cos(body_rpy(2));
  double sin_yaw = sin(body_rpy(2));
  double cos_pitch = cos(body_rpy(1));
  double tan_pitch = tan(body_rpy(1));

  Matrix3d angular_velocity_to_rpy_rate;
  angular_velocity_to_rpy_rate << cos_yaw / cos_pitch,
      sin_yaw / cos_pitch, 0, -sin_yaw, cos_yaw, 0, cos_yaw * tan_pitch,
      sin_yaw * tan_pitch, 1;

  // angular_velocity_to_rpy_rate << cos_yaw, sin_yaw, 0,
```

```
  //                                  -sin_yaw, cos_yaw, 0,
  //                                         0,       0, 1;

  A_mat.setZero();

  A_mat.block<3, 3>(0, 6) = angular_velocity_to_rpy_rate;
  A_mat.block<3, 3>(3, 9) = Eigen::Matrix3d::Identity();
  A_mat(11, 12) = 1;
}
```

```
void ConvexMPCDense::CalculateBMatrix(double& inv_mass,
                                      Matrix3d& inv_inertia,
                                      Matrix3x4& foot_positions,
                                      MatrixNxNu& B_mat)
{
  // Construct B matrix in the linearized CoM dynamics equation

  // B (13x(num_legs*3)) contains non_zero elements only in row 6:12.

  B_mat.setZero();

  for (int i = 0; i < num_legs; ++i)
  {
    B_mat.block<k3Dim, k3Dim>(6, i * k3Dim) =
        inv_inertia * skew(foot_positions.col(i)); // r x f torque
    B_mat.block<k3Dim, k3Dim>(9, i * k3Dim) =
        inv_mass * Eigen::Matrix3d::Identity(); // f = ma
  }
}
```

Also, the discretized linear dynamics matrices $\mathbf{A}_k$,$\mathbf{B}_k$ defined by equation (4-35) are computed using the function **CalculateDiscreteABMatrices(.)** that is implemented in the following code snippet.

```
void ConvexMPCDense::CalculateDiscreteABMatrices(
    MatrixNxNx& A_mat, MatrixNxNu& B_mat, double& planning_timestep,
    MatrixNxNx& Ad_mat, MatrixNxNu& Bd_mat)
{
  // Calculates the discretized space-time dynamics. Given the dynamics
  // equation:
  //   xdot = A x + B u
  // and a timestep dt, we can estimate the snapshot of the state at t +
  // dt by:
  //   x(t + dt) = = Ad x + Bd u

  // Using explicit 1st order Euler integration with zero-order hold on u

  const int state_dim = kStateDim;

  Ad_mat.setZero();
  Bd_mat.setZero();

  Ad_mat = MatrixXd::Identity(state_dim, state_dim) +
           A_mat * planning_timestep;
  Bd_mat = B_mat * planning_timestep;
}
```

The solver OSQP is also utilized here to solve the QP. The decision variables vector $\mathbf{z}$ will be equal to: $\mathbf{z}=\begin{bmatrix}\mathbf{u}_0^T & \mathbf{u}_1^T & \dots & \mathbf{u}_N^T\end{bmatrix}^T=\mathbf{U}$. According to this arrangement of the decision variables, and to the form of the QP formulation of equation (4-53), will the rest of the QP matrices and vectors be selected. The Hessian $\mathbf{P}$ and the gradient $\mathbf{q}$ in the cost function are the following:

$$\mathbf{P}=\mathbf{H},\quad \mathbf{q}=\mathbf{h} \tag{4-59}$$

The linear constraints matrix $\mathbf{A}_c$ and the constraint bound vectors $\mathbf{l}_b,\mathbf{u}_b$ are the following:

$$\mathbf{A}_c=\mathbf{C},\ \mathbf{l}_b=\underline{\mathbf{c}},\ \mathbf{u}_b=\bar{\mathbf{c}} \tag{4-60}$$

The aforementioned QP matrices and vectors are filled in by calling the functions **CalculateQPMatrices(.)**, **CalculateConstraintMatrix(.)** and **CalculateConstraintBounds(.)**. The first function builds the matrix and vector $\mathbf{P},\mathbf{q}$, the second function builds the matrix $\mathbf{A}_c$ and the third function builds the vectors $\mathbf{l}_b,\mathbf{u}_b$. These functions are presented in the three code snippets below:

```
// Calculate Hessian matrix H and gradient vector q for QP
void ConvexMPCDense::CalculateQPMatrices(
    int& planning_horizon_steps, VectorNx& x0, VectorNxNh& state_ref,
    VectorNuNh& input_ref, MatrixNxNhNxNh& Q, MatrixNuNhNuNh& R,
    MatrixNxNx& Ad_mat, MatrixNxNu& Bd_mat, Sparse_Matrix& H_sparse,
    VectorNuNh& q_dense)
{
  const int state_dim = Ad.cols();
  const int action_dim = Bd.cols();

  MatrixNxNhNx A_qp;
  A_qp.setZero();
  Eigen::MatrixXd B_qp(state_dim * planning_horizon_steps,
                       action_dim * planning_horizon_steps);
  B_qp.setZero();

  Eigen::MatrixXd H_dense(action_dim * planning_horizon_steps,
                          action_dim * planning_horizon_steps);
  H_dense.setZero();
  q_dense.setZero();

  ////////////////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////////////////

  // calculate A_qp and B_qp

  // A_qp = [A,
  // A^2,
  // A^3,
  // ...
  // A^k]'

  // B_qp = [A^0*B(0),
  // A^1*B(0), B(1),
  // A^2*B(0), A*B(1), B(2),
  // ...
  // A^(k-1)*B(0), A^(k-2)*B(1), A^(k-3)*B(2), ... B(k-1)]

  for (int i = 0; i < planning_horizon_steps; ++i)
  {
```

```
    if (i == 0)
    {
      A_qp.block(state_dim * i, 0, state_dim, state_dim) = Ad;
    }
    else
    {
      A_qp.block(state_dim * i, 0, state_dim, state_dim) =
          A_qp.block(state_dim * (i - 1), 0, state_dim, state_dim) * Ad;
    }

    for (int j = 0; j < i + 1; ++j)
    {
      if (i - j == 0)
      {
        B_qp.block(state_dim * i, action_dim * j, state_dim,
                   action_dim) =
            Bd; // Bd.block(j * state_dim, 0, state_dim, action_dim);
      }
      else
      {
        B_qp.block(state_dim * i, action_dim * j, state_dim,
                   action_dim) =
            A_qp.block(state_dim * (i - j - 1), 0, state_dim,
                       state_dim) *
            Bd; // Bd.block(j * state_dim, 0, state_dim, action_dim);
      }
    }
  }

  ////////////////////////////////////////////////////////////////////////

  // calculate hessian
  H_dense = (B_qp.transpose() * Q * B_qp + R);
  H_sparse = H_dense.sparseView();

  // calculate gradient
  q_dense = B_qp.transpose() * Q * (A_qp * x0 - state_ref);
}
```

```
// Calculate Linear Constraints Matrix Ac for QP
void ConvexMPCDense::CalculateConstraintMatrix(
    int& planning_horizon_steps, MatrixNhNl& contact_states,
    Vector4d& friction_coeff, Sparse_Matrix& Ac_sparse)
{
  Eigen::MatrixXd Ac_dense(kConstraintDim * num_legs *
                               planning_horizon_steps,
                           kInputDim * planning_horizon_steps);
  Ac_dense.setZero();

  for (int i = 0; i < planning_horizon_steps * num_legs; ++i)
  {
    Ac_dense.block<kConstraintDim, k3Dim>(i * kConstraintDim, i * k3Dim)
    << -1,
    0, friction_coeff(0),     // -fx + mu * fz
    1, 0, friction_coeff(1),  //  fx + mu * fz
    0, -1, friction_coeff(2), // -fy + mu * fz
    0, 1, friction_coeff(3),  //  fy + mu * fz
    0, 0, 1;                  //  fz
  }

  Ac_sparse = Ac_dense.sparseView();
```

```
}
```

```
// Build lower and upper bound vectors lb, ub for QP
void ConvexMPCDense::CalculateConstraintBounds(
    int& planning_horizon_steps, MatrixNhNl& contact_state,
    double& fz_max, double& fz_min, double& friction_coeff,
    VectorNcNh& l, VectorNcNh& u)
{
  VectorXd constraint_lb_C(kConstraintDim * int(num_legs) *
                           planning_horizon_steps);
  VectorXd constraint_ub_C(kConstraintDim * int(num_legs) *
                           planning_horizon_steps);

  for (int i = 0; i < planning_horizon_steps; ++i)
  {
    for (int j = 0; j < num_legs; ++j)
    {
      const int row = (i * num_legs + j) * kConstraintDim;

      constraint_lb_C(row) = 0.0;
      constraint_lb_C(row + 1) = 0.0;
      constraint_lb_C(row + 2) = 0.0;
      constraint_lb_C(row + 3) = 0.0;
      constraint_lb_C(row + 4) = fz_min * contact_state(i, j);

      const double friction_ub = OsqpEigen::INFTY * contact_state(i, j);

      constraint_ub_C(row) = friction_ub;
      constraint_ub_C(row + 1) = friction_ub;
      constraint_ub_C(row + 2) = friction_ub;
      constraint_ub_C(row + 3) = friction_ub;
      constraint_ub_C(row + 4) = fz_max * contact_state(i, j);
    }
  }

  l = constraint_lb_C;
  u = constraint_ub_C;
}
```

Finally, the function **ComputeContactForces(.)** needs to be called at every control loop iteration to solve the QP and compute the GRFs that track the desired position and velocity of the robot. This function is shown in the following code snippet. The initial state vector and the desired (reference) state vector of the linearized SRBD have to be expressed in the body yaw aligned world frame. The desired x,y -position and yaw angle ψ are determined by integrating the x,y -velocity and z - angular velocity respectively. In case were the desired z - angular velocity of the robot is nonzero, then the yaw angle from the previous control loop iteration is used as the initial condition for the integration. Then the QP matrices and vectors can be calculated by calling the functions **CalculateQPMatrices(.)**, **CalculateConstraintMatrix(.)** and **CalculateConstraintBounds(.)**. Afterwards, the QP solver gets either initialized or updated and solves the problem. The predicted GRFs are then expressed in the robot's body frame since the foot Jacobians matrices computed by calling the function **ComputeFootJacobian(.)** are also expressed in this frame.

```
void ConvexMPCDense::ComputeContactForces(
    double current_time, double& yaw, double& yaw_ref,
    VectorXd& foot_contact_states, Matrix3x4& foot_positions_body_frame,
```

```
    Vector3d& com_position, Vector3d& com_velocity,
    Vector3d& body_orientation_rpy, Vector3d& body_angular_velocity,
    Vector3d& desired_com_position, Vector3d& desired_com_velocity,
    Vector3d& desired_body_orientation_rpy,
    Vector3d& desired_body_angular_velocity)
{
  std::cout << "osqp_compute_contact_forces_DENSE:\n" << std::endl;

  // std::cout << "Here is robot_mass:\n"
  //           << 1 / inv_robot_mass << std::endl;

  // First we compute the foot positions in the world frame.
  DCHECK_EQ(body_orientation_rpy.size(), k3Dim);
  const Quaterniond com_rotation =
      AngleAxisd(body_orientation_rpy(0), Vector3d::UnitX()) *
      AngleAxisd(body_orientation_rpy(1), Vector3d::UnitY()) *
      AngleAxisd(body_orientation_rpy(2), Vector3d::UnitZ());

  DCHECK_EQ(foot_positions_body_frame.size(), k3Dim * num_legs);
  foot_positions_base = foot_positions_body_frame;

  for (int i = 0; i < num_legs; ++i)
  {
    foot_positions_world.col(i) =
        com_rotation * foot_positions_base.col(i);
  }

  // In MPC planning we don't care about absolute position in the
  // horizontal plane.
  double com_x = com_position(0);
  double com_y = com_position(1);
  double com_z = com_position(2);

  VectorNx state;
  state << body_orientation_rpy(0), body_orientation_rpy(1),
      ((desired_body_angular_velocity(2) == 0.0) ?
           (yaw - yaw_ref) :
           (yaw - body_previous_yaw)),
      com_x, com_y, com_z, body_angular_velocity(0),
      body_angular_velocity(1), body_angular_velocity(2),
      com_velocity(0), com_velocity(1), com_velocity(2), -kGravity;

  // Prepare the current and desired state vectors of length kStateDim *
  // planning_horizon.

  desired_states(0) = desired_body_orientation_rpy(0);
  desired_states(1) = desired_body_orientation_rpy(1);
  desired_states(2) =
      ((desired_body_angular_velocity(2) == 0.0) ?
           (0.0) :
           (yaw - body_previous_yaw)) +
      planning_timestep * desired_body_angular_velocity(2);

  desired_states(3) = ((desired_com_velocity(0) == 0.0) ? 0.0 : com_x) +
                      planning_timestep * desired_com_velocity(0);
  desired_states(4) = ((desired_com_velocity(1) == 0.0) ? 0.0 : com_y) +
                      planning_timestep * desired_com_velocity(1);
  desired_states(5) = desired_com_position(2);

  // Prefer to stabilize roll and pitch.
  desired_states(6) = desired_body_angular_velocity(0);
```

```
  desired_states(7) = desired_body_angular_velocity(1);
  desired_states(8) = desired_body_angular_velocity(2);

  desired_states(9) = desired_com_velocity(0);
  desired_states(10) = desired_com_velocity(1);

  // Prefer to stabilize the body height.
  desired_states(11) = 0.0;

  desired_states(12) = -kGravity;

  for (int i = 1; i < planning_horizon_steps; ++i)
  {
    desired_states(i * kStateDim + 0) = desired_body_orientation_rpy(0);
    desired_states(i * kStateDim + 1) = desired_body_orientation_rpy(1);
    desired_states(i * kStateDim + 2) =
        desired_states((i - 1) * kStateDim + 2) +
        planning_timestep * desired_body_angular_velocity(2);

    desired_states(i * kStateDim + 3) =
        desired_states((i - 1) * kStateDim + 3) +
        planning_timestep * desired_com_velocity(0);
    desired_states(i * kStateDim + 4) =
        desired_states((i - 1) * kStateDim + 4) +
        planning_timestep * desired_com_velocity(1);
    desired_states(i * kStateDim + 5) = desired_com_position(2);

    // Prefer to stabilize roll and pitch.
    desired_states(i * kStateDim + 6) =
        desired_body_angular_velocity(0);
    desired_states(i * kStateDim + 7) =
        desired_body_angular_velocity(1);
    desired_states(i * kStateDim + 8) =
        desired_body_angular_velocity(2);

    desired_states(i * kStateDim + 9) = desired_com_velocity(0);
    desired_states(i * kStateDim + 10) = desired_com_velocity(1);

    // Prefer to stabilize the body height.
    desired_states(i * kStateDim + 11) = 0.0;

    desired_states(i * kStateDim + 12) = -kGravity;
  }

  // Prepare the current and desired input vectors of length kInputDim *
  // planning_horizon_steps.
  desired_inputs = VectorXd::Zero(kInputDim * planning_horizon_steps);

  Vector3d rpy(body_orientation_rpy(0), body_orientation_rpy(1),
               body_orientation_rpy(2));

  // std::cout << "Here is the yaw : \n" << yaw << std::endl;
  // std::cout << "Here is the yaw_ref : \n" << yaw_ref <<
  // std::endl;

  CalculateAMatrix(rpy, A);

  Matrix3d rotation = rpy::rpyToMatrix(rpy(0), rpy(1), rpy(2));
  Matrix3d inv_inertia_world =
      rotation * inv_robot_inertia * rotation.transpose();
```

```
  CalculateBMatrix(inv_robot_mass, inv_inertia_world,
                   foot_positions_world, B);

  CalculateDiscreteABMatrices(A, B, planning_timestep, Ad, Bd);

  CalculateQPMatrices(planning_horizon_steps, state, desired_states,
                      desired_inputs, Q_qp, R_qp, Ad, Bd, H, q);

  const VectorXd one_vec =
      VectorXd::Constant(planning_horizon_steps, 1.0);
  const VectorXd zero_vec = VectorXd::Zero(planning_horizon_steps);

  for (int j = 0; j < foot_contact_states.size(); ++j)
  {
    if (foot_contact_states(j))
    {
      contact_states.col(j) = one_vec;
    }
    else
    {
      contact_states.col(j) = zero_vec;
    }
  }

  std::cout << "Here is the matrix contact_states:\n"
            << contact_states << std::endl;

  double mu = foot_friction_coeffs(0);
  CalculateConstraintBounds(planning_horizon_steps, contact_states,
                            fz_max, fz_min, mu, lb, ub);

  CalculateConstraintMatrix(planning_horizon_steps, contact_states,
                            foot_friction_coeffs, Ac);

  if (!solver.isInitialized())
  {
    std::cout << "It IS NOT initialized \n" << std::endl;

    // settings
    solver.settings()->setVerbosity(false);
    solver.settings()->setWarmStart(true);
    solver.settings()->setPolish(true);
    // solver.settings()->setAdaptiveRhoInterval(25) ;

    solver.settings()->setAbsoluteTolerance(1e-03);
    solver.settings()->setRelativeTolerance(1e-03);
    solver.settings()->setCheckTermination(1);
    // solver.settings()->setScaledTerimination(1) ;

    // set the initial data of the QP solver
    solver.data()->setNumberOfVariables(Ac.cols());
    solver.data()->setNumberOfConstraints(Ac.rows());

    solver.data()->setHessianMatrix(H);
    solver.data()->setGradient(q);
    solver.data()->setLinearConstraintsMatrix(Ac);
    solver.data()->setLowerBound(lb);
    solver.data()->setUpperBound(ub);

    // instantiate the solver
    solver.initSolver();
```

```
    VectorNuNh primal_variable_init;
    VectorNu primal_variable_vec;
    primal_variable_vec << 0.0, 0.0,
        (robot->robot_mass * kGravity) / 4.0, 0.0, 0.0,
        (robot->robot_mass * kGravity) / 4.0, 0.0, 0.0,
        (robot->robot_mass * kGravity) / 4.0, 0.0, 0.0,
        (robot->robot_mass * kGravity) / 4.0;
    primal_variable_init =
        primal_variable_vec.replicate(planning_horizon_steps, 1);

    solver.setPrimalVariable(primal_variable_init);
  }
  else
  {
    std::cout << "It IS initialized \n" << std::endl;

    // update the QP matrices and vectors
    solver.updateHessianMatrix(H);
    solver.updateGradient(q);
    // solver.updateLinearConstraintsMatrix( Ac );
    // solver.updateLowerBound( lb ) ;
    // solver.updateUpperBound( ub ) ;
    solver.updateBounds(lb, ub);
  }

  // solve the QP problem
  //(!solver->solve());
  solver.solveProblem() != OsqpEigen::ErrorExitFlag::NoError;

  VectorXd qp_solution = solver.getSolution();

  VectorNm contact_forces_world_yaw_aligned =
      qp_solution.head<kInputDim>();

  for (int i = 0; i < num_legs; i++)
  {
    predicted_contact_forces.segment(i * 3, 3) =
        rotation * contact_forces_world_yaw_aligned.segment(i * 3, 3);
  }

  body_previous_yaw =
      ((desired_body_angular_velocity(2) == 0.0) ? (yaw_ref) : (yaw));

  std::cout << "Here is predicted_contact_forces:\n"
            << predicted_contact_forces << std::endl;
}
```

4.4.5 Locomotion Controller

The locomotion controller consists of the class **LocomotionController**. The functions that belong to this class are responsible for deciding the motor torque (action) of each leg joint depending on whether the leg is in stance or swing state. In the case of the swinging legs, it computes the actions that should be applied to the leg joints using equation (4-18). The header file *locomotion_controller.h* contains the definition of the class, that includes declaration of its member variables and functions. The implementation of the class is located in the *locomotion_controller.cpp* file.

First of all, the class **LocomotionController** should be initialized. The parameters necessary to initialize this class are the following: an instance of the **Robot** class, an instance of the class **OpenLoopGaitGenerator**, an instance of the class **RaibertSwingLegController** and an instance of the class **TorqueStanceLegController**. The function that performs this initialization is shown in the following code snippet.

```
LocomotionController::LocomotionController(
    Robot* robot, OpenLoopGaitGenerator* gaitGenerator,
    RaibertSwingLegController* swingLegController,
    TorqueStanceLegController* stanceLegController)
{
  this->robot = robot;
  this->gaitGenerator = gaitGenerator;
  this->swingLegController = swingLegController;
  this->stanceLegController = stanceLegController;
}
```

Secondly, the function **Reset(.)** is also a member of this class. It needs to be called to reset the parameters of the planners and the controllers. This function is shown in the following code snippet.

```
void LocomotionController::Reset(double current_time, Eigen::VectorXd& q)
{
  // Resets the controller's internal state

  gaitGenerator->Reset(current_time);
  swingLegController->Reset(current_time, q);
}
```

Furthermore, the function **Update(.)** needs to be called at every control loop iteration so that the various parameters of the planners and the controllers get updated. This function is shown in the following code snippet.

```
void LocomotionController::Update(double current_time,
                                  Vector4d& contact_state,
                                  VectorNm& contact_force,
                                  Eigen::VectorXd& q, Vector2d& linSpeed,
                                  double& angSpeed)
{
  // Updates the controller's internal state

  gaitGenerator->Update(current_time, contact_state, contact_force);
  swingLegController->UpdateControlParameters(linSpeed, angSpeed);
  swingLegController->Update(current_time, q);
  stanceLegController->UpdateControlParameters(linSpeed, angSpeed);
}
```

Finally, the function **GetAction(.)** needs to be called at every control loop iteration to compute the motor torques via the sub controllers. This function is shown in the following code snippet. The motor torques that correspond to the stance legs are computed directly by the member function of the class **TorqueStanceLegController**, **GetAction(.)**. The motor torques that correspond to the swing legs are computed in this function, using equation (4-18). The

member function of the class **RaibertSwingLegController**, **GetAction(.)**, merely computes the desired joint angles and the PD controller gains corresponding to the swing legs.

```
void LocomotionController::GetAction(
    double current_time, Eigen::VectorXd& q, Eigen::VectorXd& q_dot,
    VectorNm& qj_dot, Vector3d& com_position,
    Vector3d& body_orientation_rpy, Vector3d& com_velocity,
    Vector3d& body_angular_velocity)
{
  // Returns the control outputs (torques) for all motors

  std::cout << "locomotion_controller_get_action:\n" << std::endl;

  swingLegController->GetAction(current_time, q, com_position,
                                body_orientation_rpy, com_velocity,
                                body_angular_velocity);
  MatrixNm5 swing_action = swingLegController->action;

  stanceLegController->GetAction(current_time, q, q_dot, com_position,
                                 body_orientation_rpy, com_velocity,
                                 body_angular_velocity);
  MatrixNm5 stance_action = stanceLegController->action;

  action.setZero();
  VectorNm qj = q.segment(com_dof, num_motors);

  Eigen::MatrixXd M;
  Eigen::MatrixXd Jv;
  Eigen::MatrixXd Jv_dot;

  M.setZero();

  if (!swing_action.isZero())
  {
    robot->ComputeInertiaMatrix(q, M);
  }

  for (int leg_id = 0; leg_id < num_legs; leg_id++)
  {
    if (swing_action(motors_per_leg * leg_id, 0) != 0.0)
    {
      robot->ComputeFootJacobian(q, leg_id, Jv);
      robot->ComputeFootJacobianTimeVariation(q, q_dot, leg_id, Jv_dot);

      Eigen::MatrixXd Jv_inv = robot->sdlsInv(
          Jv.block(0, com_dof + motors_per_leg * leg_id, 3, 3));

      action.segment(motors_per_leg * leg_id, 3) =
          M.block(com_dof + motors_per_leg * leg_id,
                  com_dof + motors_per_leg * leg_id, 3, 3) *
              Jv_inv *
              (swingLegController->foot_position_ddot_des.col(leg_id) -
               Jv_dot.block(0, com_dof + motors_per_leg * leg_id, 3, 3) *
                   q_dot.segment(com_dof + motors_per_leg * leg_id, 3)) +
          (-swing_action.block(motors_per_leg * leg_id, 1, 3, 1)
                .cwiseProduct(qj.segment(motors_per_leg * leg_id, 3) -
                              swing_action.block(motors_per_leg * leg_id,
                                                 0, 3, 1)) -
           swing_action.block(motors_per_leg * leg_id, 3, 3, 1)
               .cwiseProduct(qj_dot.segment(motors_per_leg * leg_id, 3) -
```

```
                              swing_action.block(motors_per_leg * leg_id,
                                                 2, 3, 1)));

      // Bias terms compensation
      action.segment(motors_per_leg * leg_id, 3) +=
          stance_action.block(motors_per_leg * leg_id, 4, 3, 1);
    }
    else
    {
      action.segment(motors_per_leg * leg_id, 3) +=
          stance_action.block(motors_per_leg * leg_id, 4, 3, 1);
    }

    VectorNm lower;
    lower = -torque_limit * Eigen ::MatrixXd ::Ones(num_motors, 1);
    VectorNm upper;
    upper = torque_limit * Eigen ::MatrixXd ::Ones(num_motors, 1);
    action = lower.cwiseMax(upper.cwiseMin(action));
  }

  std::cout << "Here is action:\n" << action << std::endl;
}
```

5 Experiments for ARGOS in Gazebo

The locomotion MPC controller whose implementation was discussed in chapter 4.4, is tested on the quadruped robot ARGOS, in Gazebo simulator. Two different gaits were performed by ARGOS in these simulations: walking and trotting gait. For each of these gaits, both nonzero desired x, y - linear velocity with zero desired z - angular velocity and zero desired x, y - linear velocity with nonzero desired z - angular velocity were tested in the simulations. The numerical values of the parameters for the MPC controller got tabulated and are presented in Table 5-1. Different gait patterns are selected depending on the locomotion speed that should be achieved. With gait patterns like trot, higher locomotion speeds can be achieved in comparison to gait patterns like walk. Thus, different gait patterns were tested. Finally, the controller should maintain the locomotion height constant and equal to the reference during the execution of the commanded tasks. Moreover, the horizon was set equal to the stance duration of each gait and did not cover the swing state. In this way, the computational cost remains relatively small while achieving adequately accurate tracking of the desired body motion.

Table 5-1. Numerical values of the MPC parameters implemented on ARGOS in Gazebo simulator.

Parameter Description	Symbol	Numerical Value
State Stage Cost	$\mathbf{Q}$	$diag\{1e^3, 1e^3, 1e^4, 1e^3, 1e^3, 5e^4, 1e^3, 1e^3, 1e^4, 1, 1, 1, 0\}$
Input Stage Cost	$\mathbf{R}$	$diag\{1e^{-4}, 1e^{-4}, 1e^{-4}, 1e^{-4}, 1e^{-4}, 1e^{-4}, 1e^{-4}, 1e^{-4}, 1e^{-4}, 1e^{-4}, 1e^{-4}, 1e^{-4}\}$
Terminal State Cost	$\mathbf{Q}_N$	$\mathbf{Q}$
Coefficient of friction	μ	0.3
Desired Locomotion Height	z_0	0.94m
Maximum foot clearance	$\delta_{\max}$	0.1m
Proportional Gain of the Abduction joint	$K_{p,abd}$	100Nm/rad
Derivative Gain of the Abduction joint	$K_{d,abd}$	3Nm·s/rad
Proportional Gain of the Hip joint	$K_{p,hip}$	100Nm/rad
Derivative Gain of the Hip joint	$K_{d,hip}$	3Nm·s/rad
Proportional Gain of the Knee joint	$K_{p,knee}$	100Nm/rad
Derivative Gain of the Knee joint	$K_{d,knee}$	3Nm·s/rad

5.1 Walking Gait

The first gait that was executed in the simulation is walk. The gait plot that corresponds to the walking gait tested is presented in Figure 5-1. This is a timing diagram that specifies the desired state of each leg, which is either stance or swing, throughout the entire gait cycle. The horizontal axis represents the values of phase variable s that cycles over the entire gait cycle. The grey colored areas indicate stance while the white colored areas indicate swing. Also, the parameters that specify that gait and other gait-dependent parameters for the MPC controller got tabulated and are presented in Table 5-2.

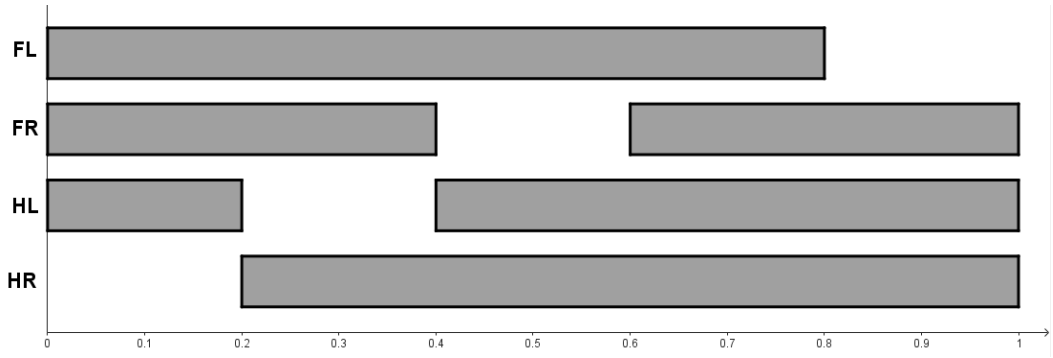


Figure 5-1. Walking gait graph. The grey colored areas indicate stance while the white colored areas indicate swing.

Table 5-2. Numerical values of the gait specific MPC parameters implemented on ARGOS in Gazebo simulator, for the walking gait.

Parameter Description	Symbol	Numerical Value
Stance Duration	T_{st}	$[0.8 \quad 0.8 \quad 0.8 \quad 0.8]\text{sec}$
Duty Factor	β	$[0.8 \quad 0.8 \quad 0.8 \quad 0.8]$
Initial Leg State		$[1 \quad 1 \quad 1 \quad 1]$ Stance: 1, Swing: 0
Initial Leg Phase		$[0 \quad 0.25 \quad 0.5 \quad 0.75]$
Nominal Contact Detection Phase Threshold		0.1
Desired Linear Velocity		0.3m/s
Desired Angular Velocity		0.3rad/s
Maximum Vertical Contact Force (translation)	$\mathbf{f}^z_{\max}$	$mg \cdot 1.0 = 343.35\text{N}$
Minimum Vertical Contact Force (translation)	$\mathbf{f}^z_{\min}$	$mg \cdot 0.1 = 34.34\text{N}$
Maximum Vertical Contact Force (rotation)	$\mathbf{f}^z_{\max}$	$mg \cdot 0.5 = 171.68\text{N}$
Minimum Vertical Contact Force (rotation)	$\mathbf{f}^z_{\min}$	$mg \cdot 0.1 = 34.34\text{N}$
Planning Horizon Steps	N	5
Planning Timestep	h	0.16sec

5.1.1 Linear Velocity Command

In this set of experiments, the reference linear velocity profile requires the robot to reach a linear velocity of 0.3m/s along the x-axis of the world frame, within 5sec, by accelerating linearly over time. The time responses of the x, y, z-position of the CoM of the robot's body

frame are presented in Figure 5-2. The reference x-position is derived by integrating the reference x-linear velocity of the CoM of the robot's body frame. The actual response of the x-position and the reference position diverge from each other after the first 5sec of simulation time have passed, but not significantly. This occurs since it cannot track perfectly the desired x-linear velocity command. This effect will become apparent when the time response of the x-linear velocity of the CoM of the robot's body will be examined. Moreover, the z-position of the CoM is closer to the reference in comparison to the y-position and exhibits smaller periodic state error. To be more precise, the mean value of the y,z-position, after the first 5sec of simulation time have passed, is equal to 0.003m and 0.9423m respectively, while amplitude of the time responses of y,z-position is approximately equal to $10^{-2}\,\text{m}$ and $10^{-4}\,\text{m}$ respectively. Therefore, the controller can successfully stabilize the robot's locomotion height and force the robot to move only along the x-axis of the world frame, while altering insignificantly the y-position of the robot's body CoM in the process.

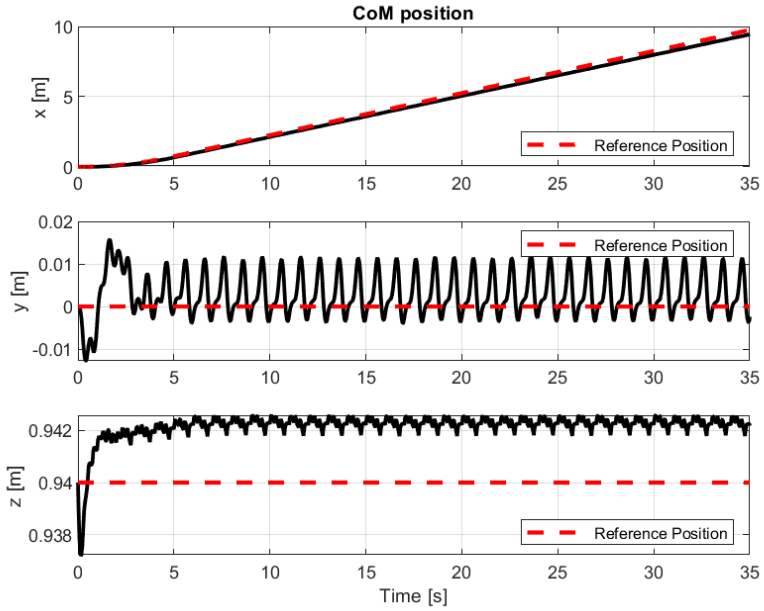


Figure 5-2. The time responses of the x,y,z-position of the CoM of the robot's body frame, for the walking gait (translation).

The time responses of the orientation of the robot's body frame, expressed using a Z-Y-X Euler angles parameterization, are presented in Figure 5-3. The amplitude of the time responses of the body roll and pitch angles is small since its order of magnitude is approximately $10^{-3}\,\text{rad}$. Also, the mean value of these two responses is almost equal to zero. Thus, the assumption made when deriving the linearized SRBD is valid. Finally, the response of the yaw angle of the body exhibits a mean value equal to -1.5709rad that does not differ significantly with the reference value of $-\pi/2\,\text{rad}$, and a small variation around the mean value that is equal to 0.004rad. Consequently, the controller is able to successfully stabilize the body roll, pitch and yaw angles.

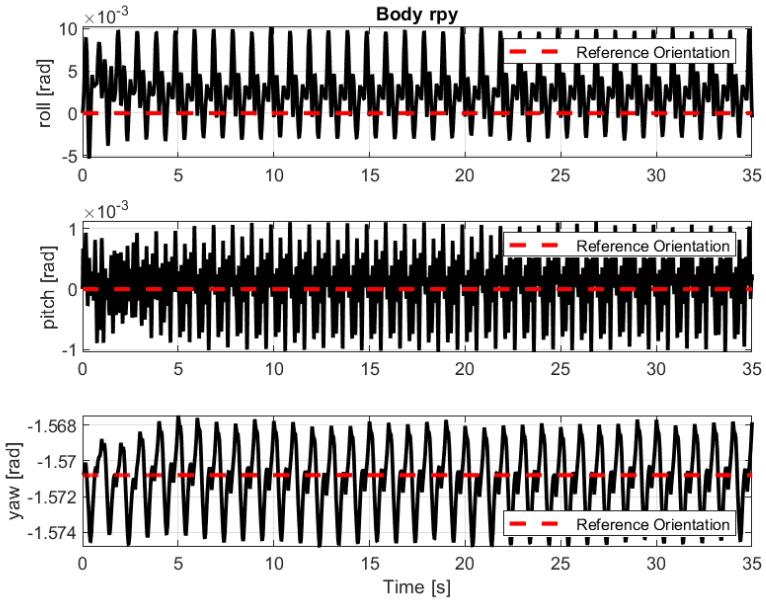


Figure 5-3. The time responses of the roll, pitch, yaw orientation of the robot's body frame, for the walking gait (translation).

The time responses of the x, y, z-linear velocity of the CoM of the robot's body frame are presented in Figure 5-4. The mean value of the x-linear velocity of the CoM after the first 5sec of simulation time is equal to 0.29m/s which is close to the desired value. This periodic state error is responsible for the divergence that was observed between the reference and the actual x-position of the robot's body CoM. Also, the amplitude of the time response of the x-linear velocity is relatively small but not insignificant since it is approximately equal to 0.05m/s. Despite that quantitative inaccuracy, the controller is able to track qualitatively correctly the desired speed profile. The mean value of the z-linear velocity of the CoM is close to the reference value with insignificant variations around that reference. The amplitude of the time response of the y-linear velocity though is higher than the one of the z-linear velocity. This difference can also be observed from the time responses of the x, y, z-position of the CoM, shown in Figure 5-2. Nevertheless, that amplitude is relatively small but not insignificant, since it is equal to 0.1m/s.

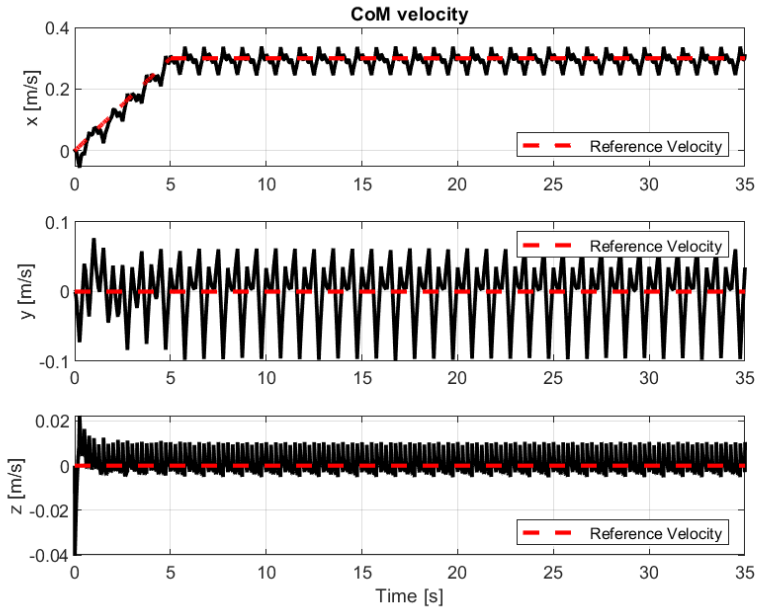


Figure 5-4. The time responses of the x,y,z-linear velocity of the CoM of the robot's body frame, for the walking gait (translation).

The time responses of the x, y, z-angular velocity of the robot's body frame are presented in Figure 5-5. The mean value of all the time responses is almost equal to zero, which is expected since the controller must keep the angular velocities of the robot's body as close to zero as possible. The amplitude of the time responses of the x, z-angular velocity is small since it is approximately equal to 0.05rad/s and 0.02rad/s respectively. However, the amplitude of the time response of the y-angular velocity is higher than the other two. This difference is justified by the fact that only the x-linear velocity of the robot's body CoM is nonzero, and thus this pitching motion is expected. Nevertheless, the amplitude is relatively small but not insignificant, since it is equal to 0.1rad/s.

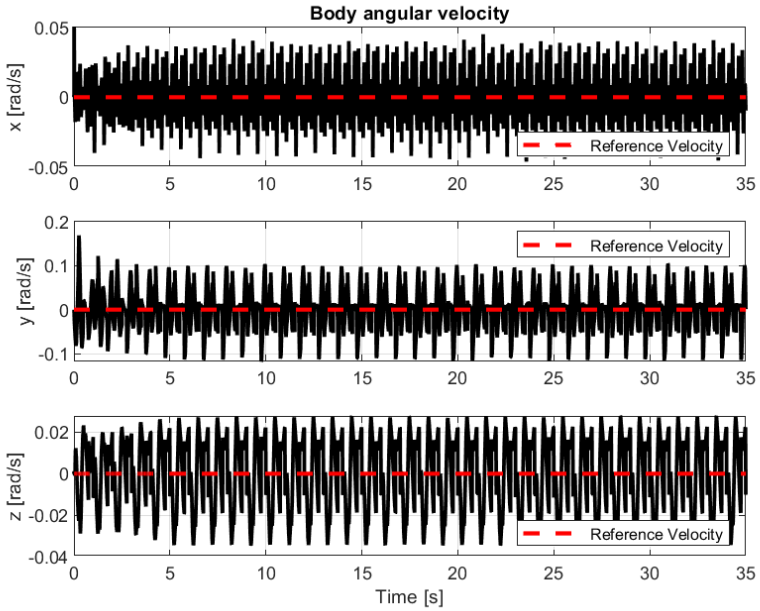


Figure 5-5. The time responses of the x,y,z-angular velocity of the robot's body frame, for the walking gait (translation).

The time responses of the angular positions, the angular velocities and the torques of the joints of the FL leg of the quadruped are presented in Figure 5-6, Figure 5-7 and Figure 5-8 respectively. The motors utilized in legged robots can usually achieve maximum angular rates that do not exceed 8rad/s. Thus, the angular velocities of all the joints must not exceed this bound. Also, such actuators can achieve maximum torques that do not exceed 50Nm. The torques of the joint actuators do not exceed this bound. The maximum vertical contact force limit was selected so that this constraint can be satisfied.

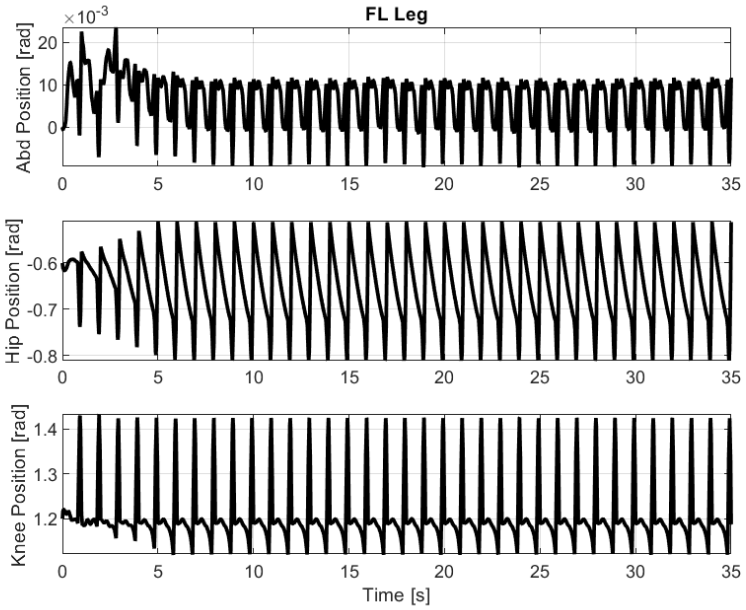


Figure 5-6. Time responses of the angular positions of the joints of the FL leg, for the walking gait (translation).

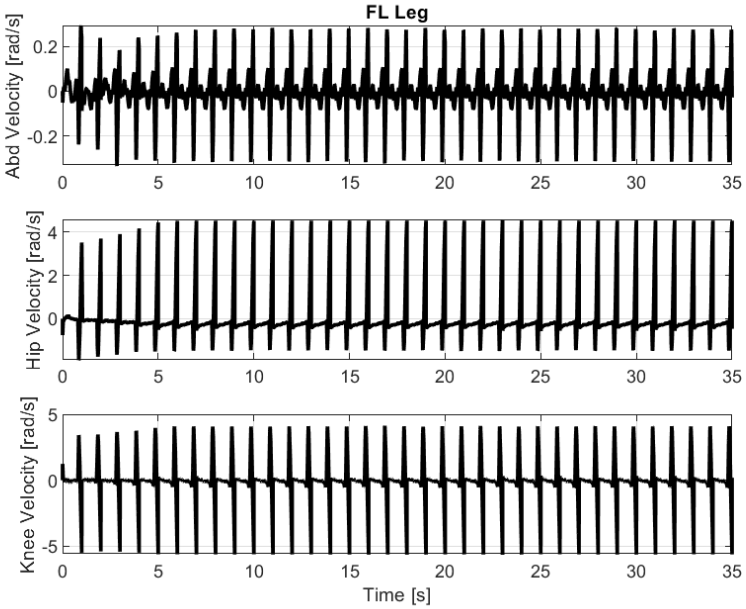


Figure 5-7. Time responses of the angular velocities of the joints of the FL leg, for the walking gait (translation).

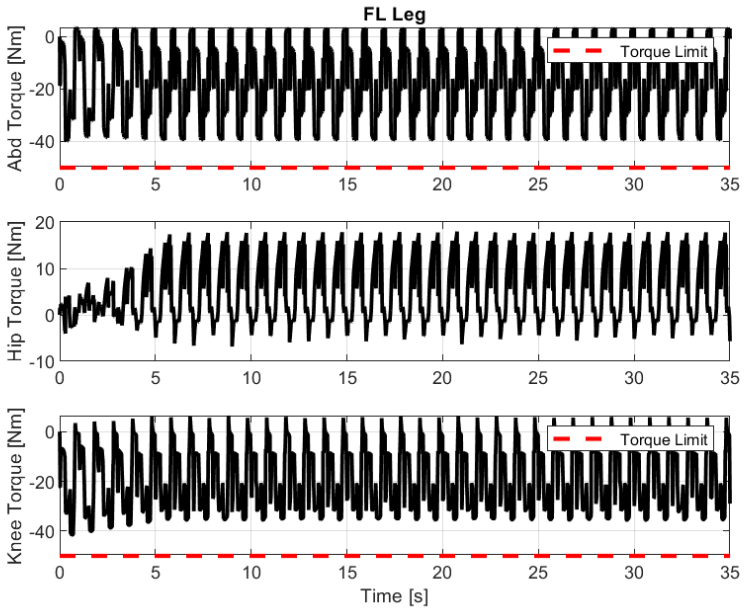


Figure 5-8. Time responses of the torques of the joints of the FL leg, for the walking gait (translation).

Finally, the distribution of the solve times of every control loop is presented in the histogram of Figure 5-9. The majority of the solve times (70%) are between 1.0ms and 1.5ms. Also, the mean value of the solve times is equal to 1.4ms and the standard deviation of the solve times is equal to 0.31ms. Thus, the control inputs can be computed very fast, since the value of the mean solve time is small, and at guaranteed rates, since the distribution around that mean value is also small.

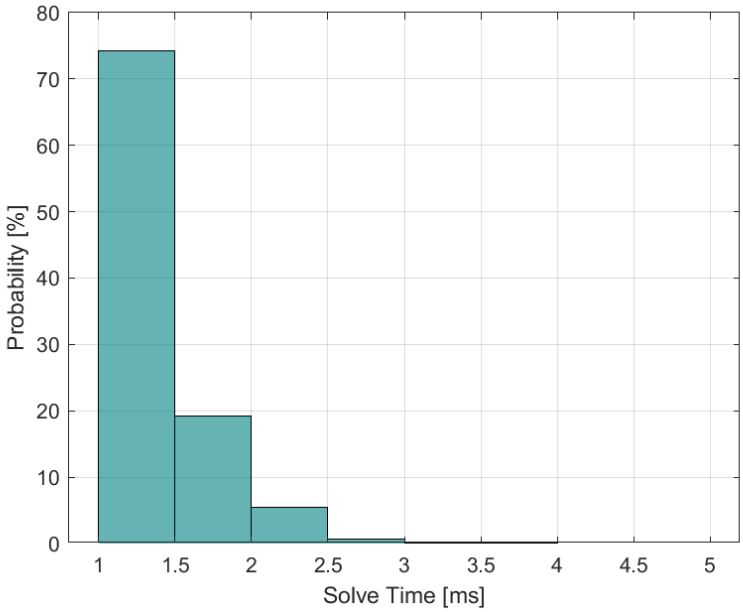


Figure 5-9. Solve time distribution of Convex MPC, for the walking gait (translation).

5.1.2 Angular Velocity Command

In this set of experiments, the reference angular velocity profile requires the robot to reach an angular velocity of 0.3rad/s along the z-axis of the world frame, within 5sec, by accelerating linearly over time. The time responses of the x, y, z-position of the CoM of the robot's body frame are presented in Figure 5-10. The amplitude of the time responses of x, y-position of the robot's CoM is relatively small but not insignificant since it is approximately equal to 0.02m. Also, the mean value of these two responses is almost equal to zero since it is approximately equal to -0.003m. Finally, the z-position of the CoM is also close to the reference and exhibits small periodic state error. To be more precise, its mean value after the first 5sec of simulation time have passed, is equal to 0.9417m, and its variation around the mean value is small in comparison to one present in time responses of the x, y-position, since its order of magnitude is approximately $10^{-4}\,\text{m}$. Therefore, the controller can successfully stabilize the robot's locomotion height and force the robot to rotate about the z-axis of the world frame, while altering insignificantly the x, y-position of the robot's body CoM in the process.

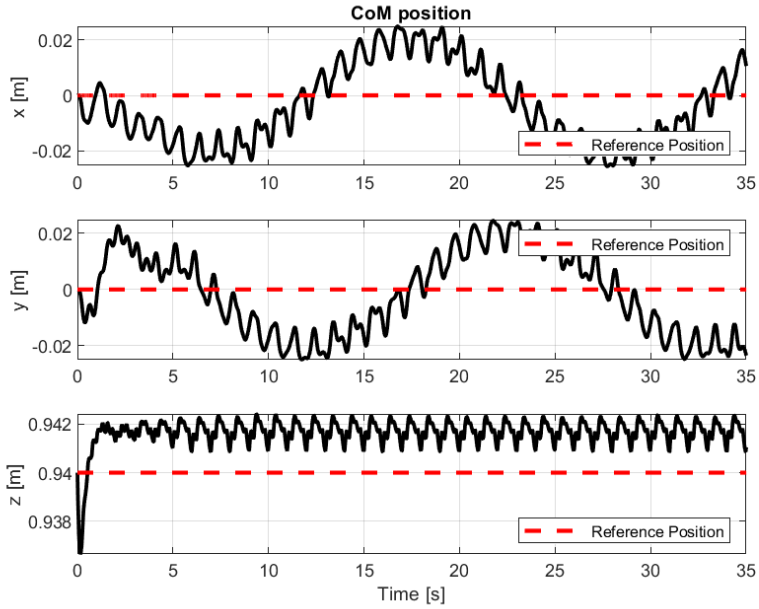


Figure 5-10. The time responses of the x,y,z-position of the CoM of the robot's body frame, for the walking gait (rotation).

The time responses of the orientation of the robot's body frame, expressed using a Z-Y-X Euler angles parameterization, are presented in Figure 5-11. The reference body yaw angle is derived by integrating the reference z-angular velocity of the robot's body frame. The actual response of the yaw angle and the reference angular position diverge from each other after the first 5sec of simulation time have passed, but not significantly. This occurs since it can track almost perfectly the desired z-angular velocity command. This effect will become apparent when the time response of the z-angular velocity of the robot's body will be examined. The amplitude of the time responses of the body roll and pitch angles is small since its order of magnitude is approximately $10^{-2}\,\text{rad}$ and $10^{-3}\,\text{rad}$ respectively. Also, the mean value of these two responses is almost equal to zero since the mean value of the roll and pitch angle of the robot's body is almost equal to $10^{-3}\,\text{rad}$ and $10^{-5}\,\text{rad}$ respectively. Thus, the assumption made when deriving the linearized SRBD is valid. Consequently, the controller is

able to successfully stabilize the body roll and pitch angles while rotating about the z-axis of the world frame.

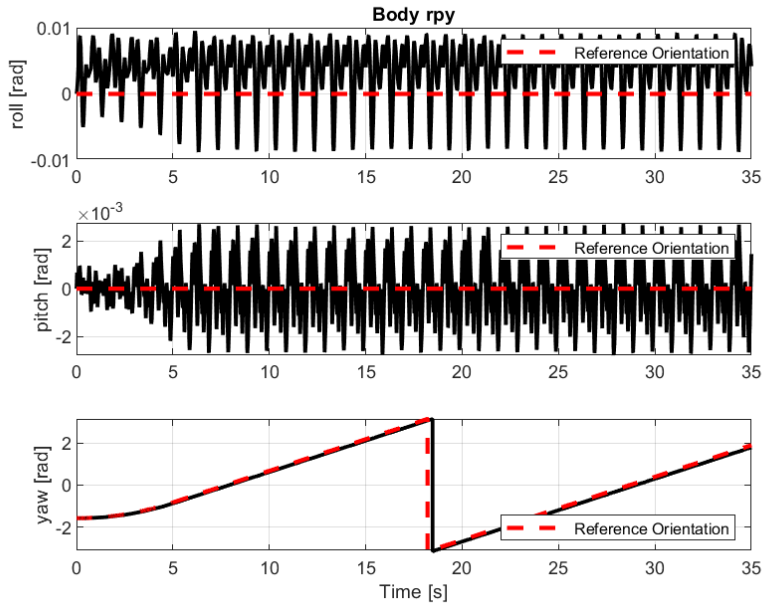


Figure 5-11. The time responses of the roll, pitch, yaw orientation of the robot's body frame, for the walking gait (rotation).

The time responses of the x, y, z-linear velocity of the CoM of the robot's body frame are presented in Figure 5-12. The mean value of all the time responses is almost equal to zero, which is expected since the controller must keep the linear velocities of the robot's body CoM as close to zero as possible. The amplitude of the time response of the z-linear velocity is relatively small since it is approximately equal to 0.01m/s. However, the amplitude of the time responses of the x, y-linear velocity is noticeably higher and almost equal to 0.1m/s. This difference can also be observed from the time responses of the x, y, z-position of the CoM, shown in Figure 5-10. This difference is justified by the fact that the robot is moving on the xy-plane of the world frame, while attempting to perform the desired rotation. Nevertheless, that amplitude is also relatively small, but not insignificant.

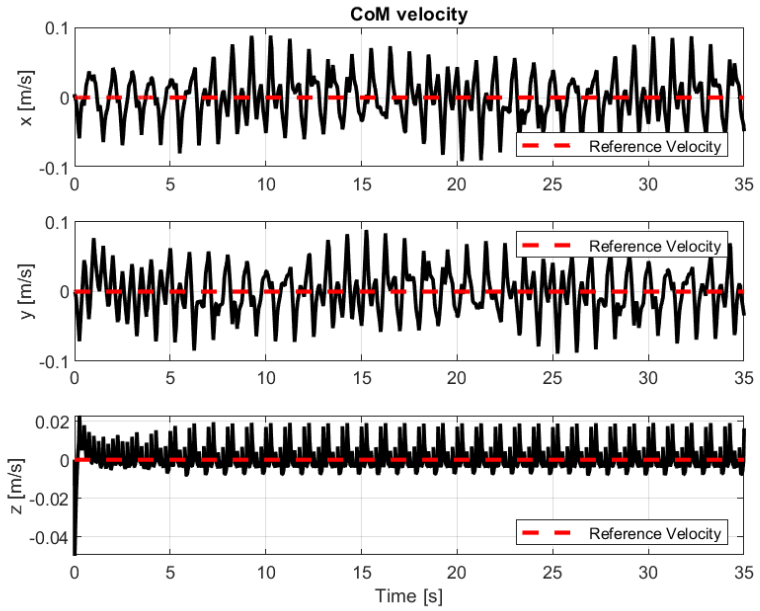


Figure 5-12. The time responses of the x,y,z-linear velocity of the CoM of the robot's body frame, for the walking gait (rotation).

The time responses of the x, y, z-angular velocity of the robot's body frame are presented in Figure 5-13. The mean value of the z-angular velocity of the robot's body frame after the first 5sec of simulation time is almost equal 0.3rad/s which is equal to the reference. Also, the amplitude of the time response of the z-angular velocity is relatively small since it is approximately equal to 0.04rad/s. Despite that quantitative inaccuracy, the controller is able to track qualitatively correctly the desired speed profile. The mean value of the y, z-angular velocity of the robot's body is close to the reference value and the amplitude of these responses is relatively small but not insignificant since it is equal to 0.2rad/s

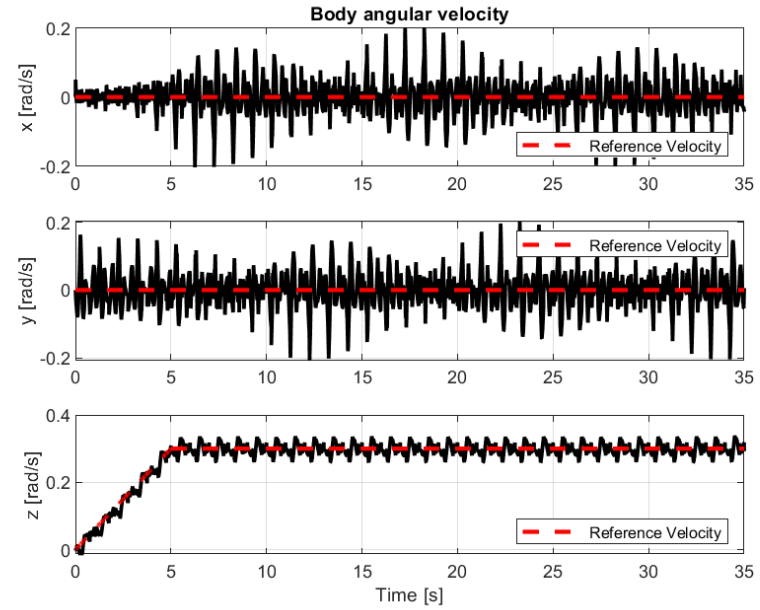


Figure 5-13. The time responses of the x,y,z-angular velocity of the robot's body frame, for the walking gait (rotation).

The time responses of the angular positions, the angular velocities and the torques of the joints of the FL leg of the quadruped are presented in Figure 5-14, Figure 5-15 and Figure 5-16 respectively. The motors utilized in legged robots can usually achieve maximum angular rates that do not exceed 8rad/s. Thus, the angular velocities of all the joints must not exceed this bound. Also, such actuators can achieve maximum torques that do not exceed 50Nm. The torques of the joint actuators do not exceed this bound. The maximum vertical contact force limit was selected so that this constraint can be satisfied.

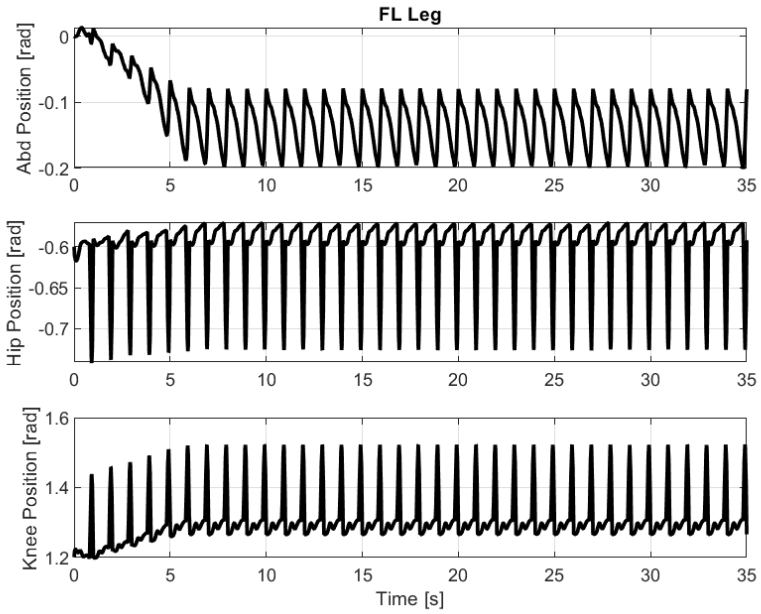


Figure 5-14. Time responses of the angular positions of the joints of the FL leg, for the walking gait (rotation).

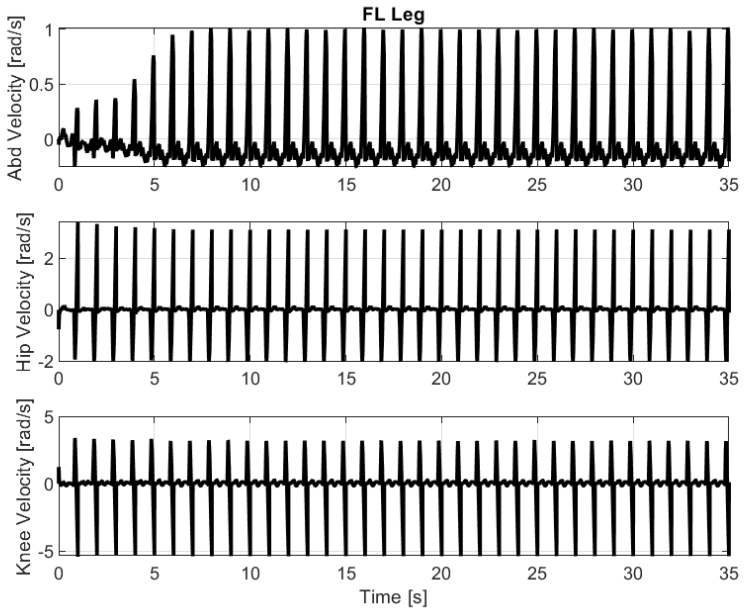


Figure 5-15. Time responses of the angular velocities of the joints of the FL leg, for the walking gait (rotation).

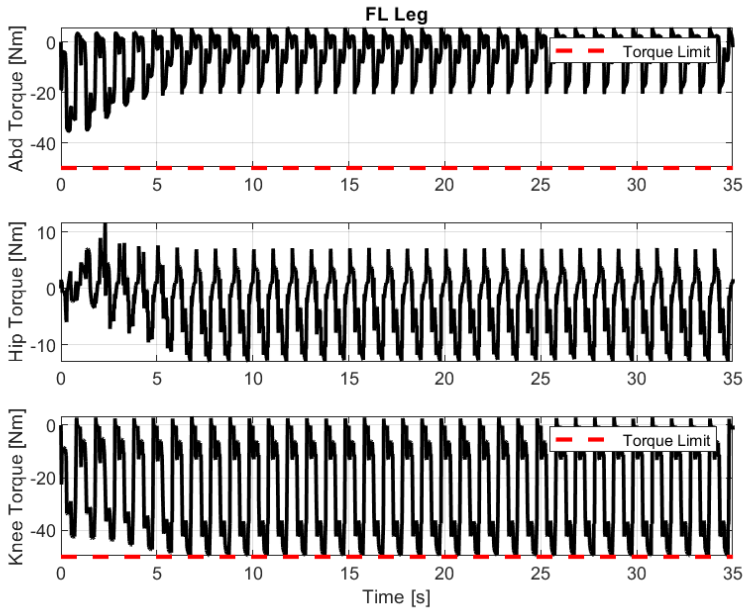


Figure 5-16. Time responses of the torques of the joints of the FL leg, for the walking gait (rotation).

Finally, the distribution of the solve times of every control loop is presented in the histogram of Figure 5-17. The majority of the solve times (70%) are between 1.0ms and 1.5ms. Also, the mean value of the solve times is equal to 1.34ms and the standard deviation of the solve times is equal to 0.32ms. Thus, the control inputs can be computed very fast, since the value of the mean solve time is small, and at guaranteed rates, since the distribution around that mean value is also small.

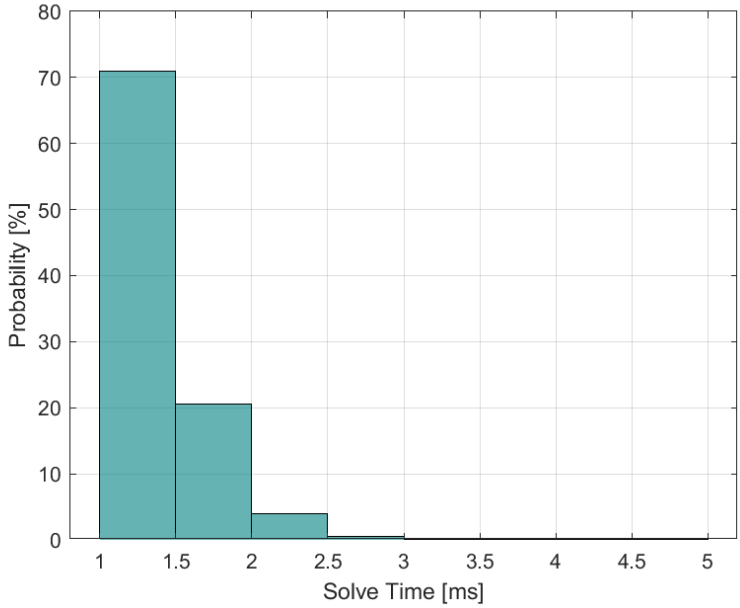


Figure 5-17. Solve time distribution of Convex MPC, for the walking gait (rotation).

5.2 Trotting Gait

The second gait that was executed in the simulation is trot. The gait plot that corresponds to the trotting gait tested is presented in Figure 5-18. This gait pattern allows the robot to achieve higher locomotion speeds, while avoiding saturation of its actuators. Also, the parameters that specify that gait and other gait-dependent parameters for the MPC controller got tabulated and are presented in Table 5-3.

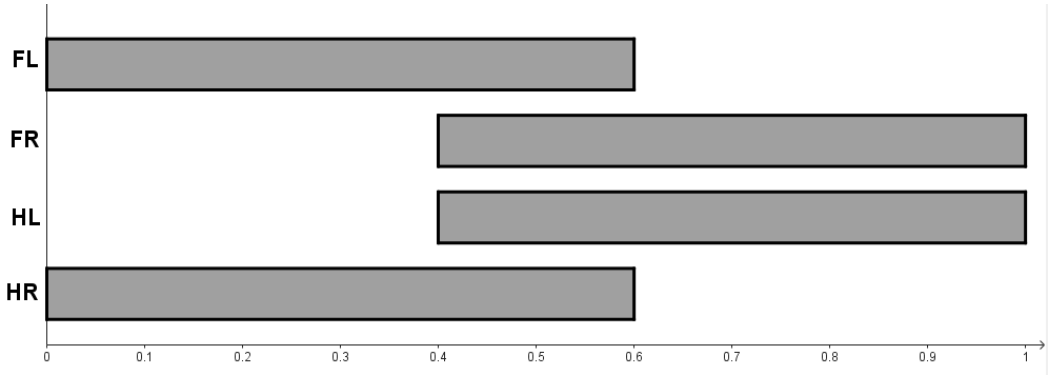


Figure 5-18. Trotting gait graph. The grey colored areas indicate stance while the white colored areas indicate swing.

Table 5-3. Numerical values of the gait specific MPC parameters implemented on ARGOS in Gazebo simulator, for the trotting gait.

Parameter Description	Symbol	Numerical Value
Stance Duration	T_{st}	$[0.3 \quad 0.3 \quad 0.3 \quad 0.3]\text{sec}$
Duty Factor	β	$[0.6 \quad 0.6 \quad 0.6 \quad 0.6]$
Initial Leg State		$[0 \quad 1 \quad 1 \quad 0]$ Stance: 1, Swing: 0
Initial Leg Phase		$[0.9 \quad 0 \quad 0 \quad 0.9]$
Nominal Contact Detection Phase Threshold		0.1
Desired Linear Velocity		0.5m/s
Desired Angular Velocity		0.5rad/s
Maximum Vertical Contact Force (translation)	$\mathbf{f}^z_{\max}$	$mg \cdot 0.6 = 206.01\text{N}$
Minimum Vertical Contact Force (translation)	$\mathbf{f}^z_{\min}$	$mg \cdot 0.1 = 34.34\text{N}$
Maximum Vertical Contact Force (rotation)	$\mathbf{f}^z_{\max}$	$mg \cdot 0.5 = 171.68\text{N}$
Minimum Vertical Contact Force (rotation)	$\mathbf{f}^z_{\min}$	$mg \cdot 0.1 = 34.34\text{N}$
Planning Horizon Steps	N	5
Planning Timestep	h	0.06sec

5.2.1 Linear Velocity Command

In this set of experiments, the reference linear velocity profile requires the robot to reach a linear velocity of 0.5m/s along the x-axis of the world frame, within 5sec, by accelerating linearly over time. The time responses of the x, y, z-position of the CoM of the robot's body frame are presented in Figure 5-19. The reference x-position is derived by integrating the reference x-linear velocity of the CoM of the robot's body frame. The actual response of the

x-position and the reference position coincide almost perfectly. Therefore, the controller can track almost perfectly the desired x-linear velocity command. The tracking accuracy will become more apparent when the time response of the x-linear velocity of the CoM of the robot's body will be examined. Moreover, the z-position of the CoM is closer to the reference in comparison to the y-position and exhibits smaller periodic state error. To be more precise, the mean value of the y,z-position, after the first 5sec of simulation time have passed, is approximately equal to $10^{-5}\,\text{m}$ and $0.9407\,\text{m}$ respectively, while amplitude of the time responses of y,z-position is approximately equal to $10^{-3}\,\text{m}$ and $10^{-4}\,\text{m}$ respectively. Therefore, the controller can successfully stabilize the robot's locomotion height and force the robot to move only along the x-axis of the world frame, without altering the y-position of the robot's body CoM in the process.

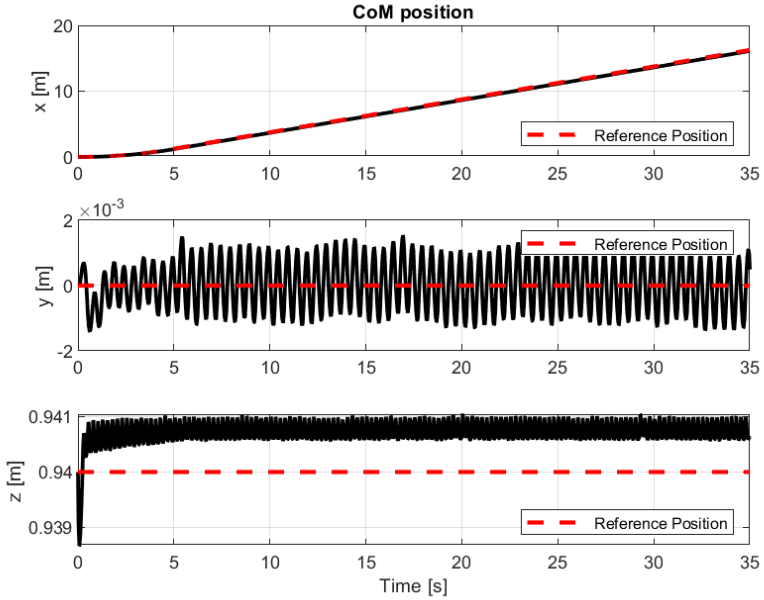


Figure 5-19. The time responses of the x,y,z-position of the CoM of the robot's body frame, for the trotting gait (translation).

The time responses of the orientation of the robot's body frame, expressed using a Z-Y-X Euler angles parameterization, are presented in Figure 5-20. The amplitude of the time responses of the body roll and pitch angles is small since its order of magnitude is approximately $10^{-3}\,\text{rad}$ and $10^{-4}\,\text{rad}$ respectively. Also, the mean value of these two responses is almost equal to zero. Thus, the assumption made when deriving the linearized SRBD is valid. Finally, the response of the yaw angle of the body exhibits a mean value equal to -1.5708rad that does not differ significantly with the reference value of $-\pi/2\,\text{rad}$, and a small variation around the mean value that is equal to 0.001rad. Consequently, the controller is able to successfully stabilize the body roll, pitch and yaw angles.

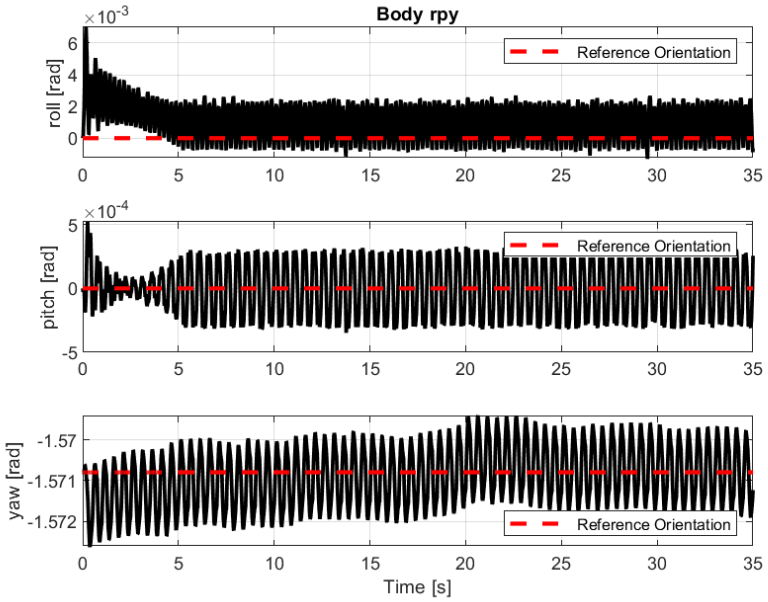


Figure 5-20. The time responses of the roll, pitch, yaw orientation of the robot's body frame, for the trotting gait (translation).

The time responses of the x, y, z-linear velocity of the CoM of the robot's body frame are presented in Figure 5-21. The mean value of the x-linear velocity of the CoM after the first 5sec of simulation time is equal to 0.497m/s which is almost equal to the desired value. Also, the amplitude of the time response of the x-linear velocity in insignificant since it is approximately equal to 0.013m/s. Therefore, the controller is able to track qualitatively and quantitatively correctly the desired speed profile. The mean value of the z-linear velocity of the CoM is close to the reference value with insignificant variations around that reference. The amplitude of the time response of the y-linear velocity though is higher than the one of the z-linear velocity. This difference can also be observed from the time responses of the x, y, z-position of the CoM, shown in Figure 5-19. Nevertheless, that amplitude is insignificant since it is equal to 0.02m/s.

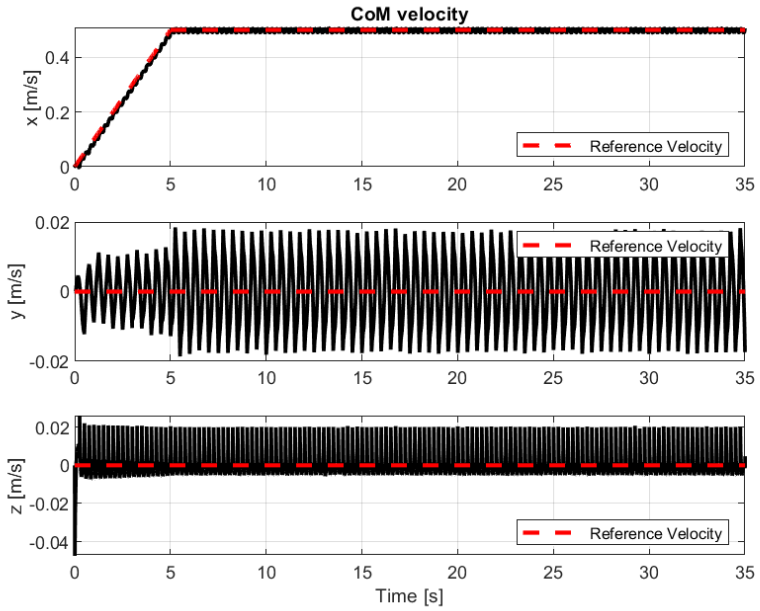


Figure 5-21. The time responses of the x,y,z-linear velocity of the CoM of the robot's body frame, for the trotting gait. (translation).

The time responses of the x, y, z-angular velocity of the robot's body frame are presented in Figure 5-22. The mean value of all the time responses is almost equal to zero, which is expected since the controller must keep the angular velocities of the robot's body as close to zero as possible. The amplitude of the time responses of the x, z-angular velocity is small since it is approximately equal to 0.01rad/s and 0.02rad/s respectively. However, the amplitude of the time response of the y-angular velocity is higher than the other two. This difference is justified by the fact that only the x-linear velocity of the robot's body CoM is nonzero, and thus this pitching motion is expected. Nevertheless, the amplitude is relatively small but not insignificant, since it is equal to 0.1rad/s.

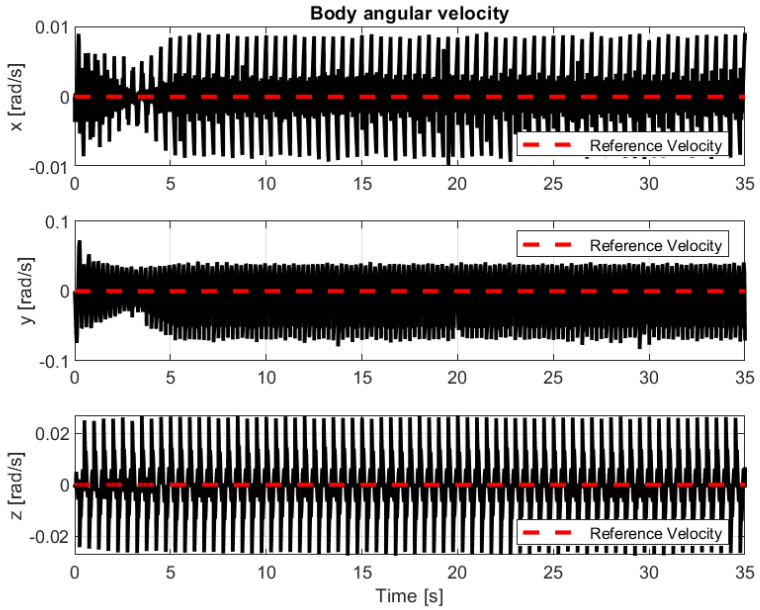


Figure 5-22. The time responses of the x,y,z-angular velocity of the robot's body frame, for the trotting gait (translation).

The time responses of the angular positions, the angular velocities and the torques of the joints of the FL leg of the quadruped are presented in Figure 5-23, Figure 5-24 and Figure 5-25 respectively. The motors utilized in legged robots can usually achieve maximum angular rates that do not exceed 8rad/s. Thus, the angular velocities of all the joints must not exceed this bound. Also, such actuators can achieve maximum torques that do not exceed 50Nm. The torques of the joint actuators do not exceed this bound. The maximum vertical contact force limit was selected so that this constraint can be satisfied.

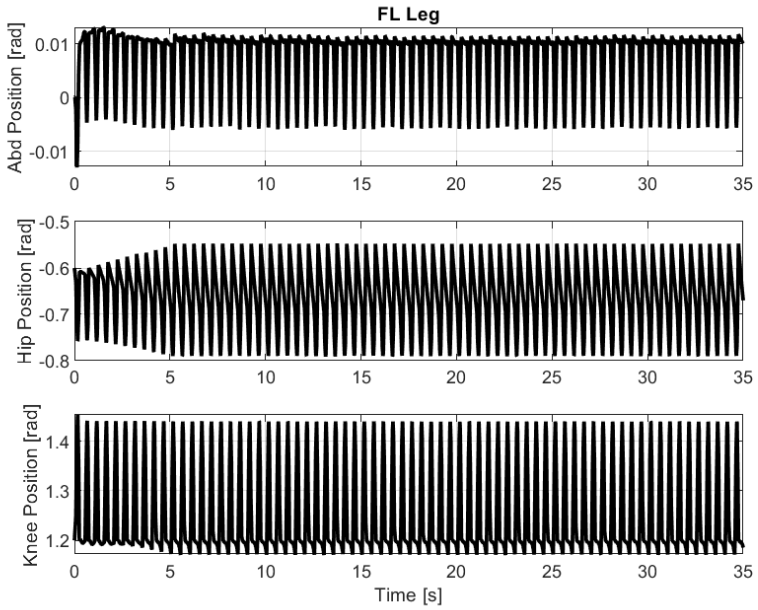


Figure 5-23. Time responses of the angular positions of the joints of the FL leg, for the trotting gait (translation).

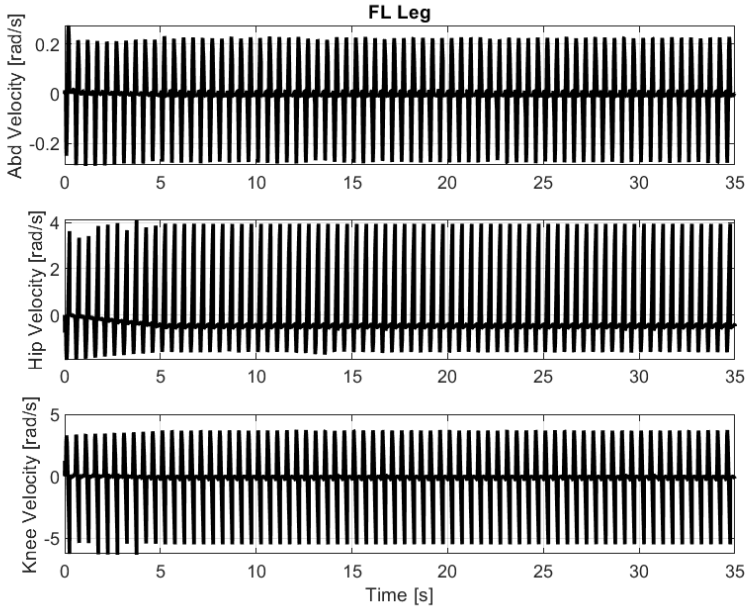


Figure 5-24. Time responses of the angular velocities of the joints of the FL leg, for the trotting gait (translation).

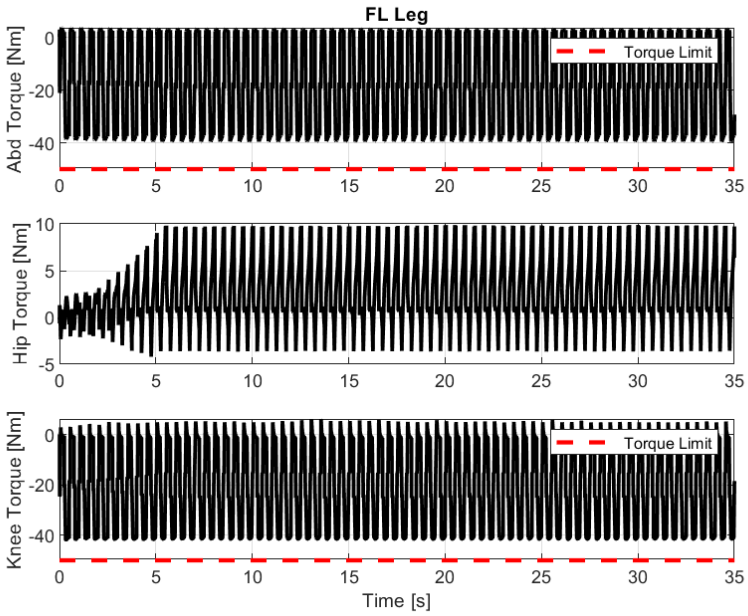


Figure 5-25. Time responses of the torques of the joints of the FL leg, for the trotting gait (translation).

Finally, the distribution of the solve times of every control loop is presented in the histogram of Figure 5-26. The majority of the solve times (70%) are between 1.0ms and 1.5ms. Also, the mean value of the solve times is equal to 1.5ms and the standard deviation of the solve times is equal to 0.33ms. Thus, the control inputs can be computed very fast, since the value of the mean solve time is small, and at guaranteed rates, since the distribution around that mean value is also small.

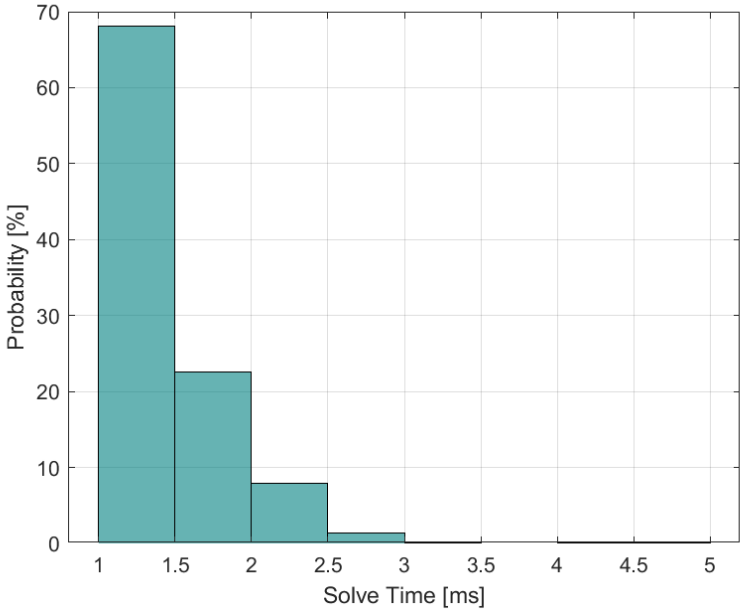


Figure 5-26. Solve time distribution of Convex MPC, for the trotting gait. (translation).

5.2.2 Angular Velocity Command

In this set of experiments, the reference angular velocity profile requires the robot to reach a linear velocity of 0.5rad/s along the z-axis of the world frame, within 5sec, by accelerating linearly over time. The time responses of the x, y, z-position of the CoM of the robot's body frame are presented in Figure 5-27. The amplitude of the time responses of x, y-position of the robot's CoM is insignificant since its order of magnitude is approximately $10^{-3}\,\text{m}$. Also, the mean value of these two responses is almost equal to zero since it is approximately equal to $10^{-4}\,\text{m}$. Finally, the z-position of the CoM is also close to the reference and exhibits small periodic state error. To be more precise, its mean value after the first of simulation time have passed, is equal to 0.9402m, and its variation around the mean value is also small but larger than the one present in time responses of the x, y-position, since its order of magnitude is approximately $10^{-3}\,\text{m}$. Therefore, the controller can successfully stabilize the robot's locomotion height and force the robot to rotate about the z-axis of the world frame, without altering the x, y-position of the robot's body CoM in the process.

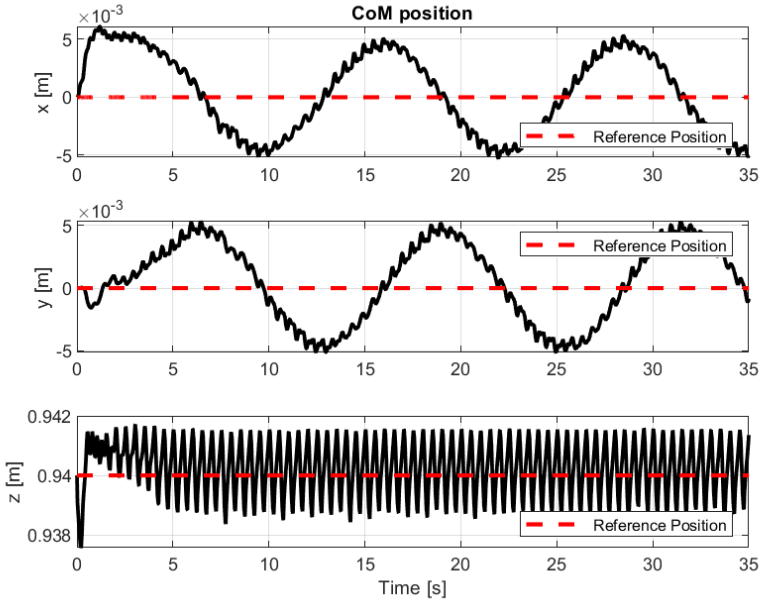


Figure 5-27. The time responses of the x,y,z-position of the CoM of the robot's body frame, for the trotting gait (rotation).

The time responses of the orientation of the robot's body frame, expressed using a Z-Y-X Euler angles parameterization, are presented in Figure 5-28. The reference body yaw angle is derived by integrating the reference z-angular velocity of the robot's body frame. The actual response of the yaw angle and the reference angular position coincide almost perfectly. Therefore, the controller can track almost perfectly the desired x-linear velocity command. This effect will become apparent when the time response of the z-angular velocity of the robot's body will be examined. The amplitude of the time responses of the body roll and pitch angles is small since its order of magnitude is approximately $10^{-3}\,\text{rad}$. Also, the mean value of these two responses is almost equal to zero since the mean value of the roll and pitch angle of the robot's body is almost equal to $10^{-3}\,\text{rad}$ and $10^{-4}\,\text{rad}$ respectively. Thus, the assumption made when deriving the linearized SRBD is valid. Consequently, the controller is able to successfully stabilize the body roll and pitch angles while rotating about the z-axis of the world frame.

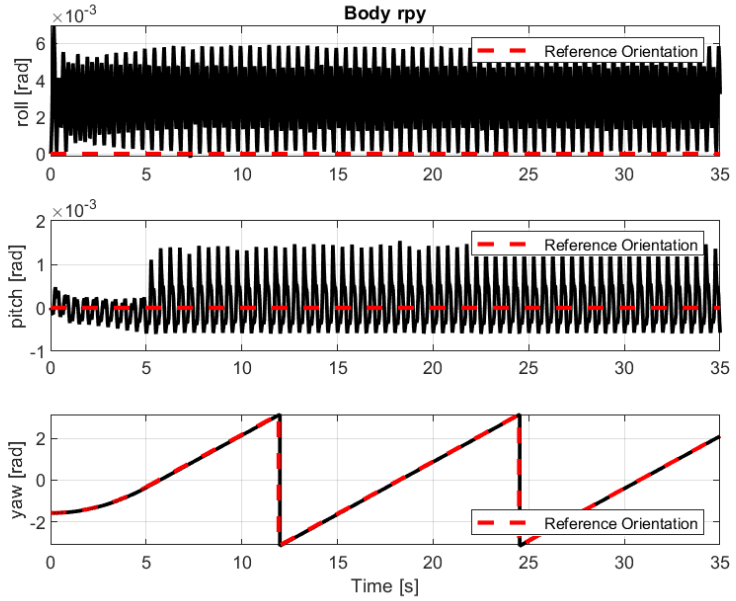


Figure 5-28. The time responses of the roll, pitch, yaw orientation of the robot's body frame, for the trotting gait (rotation).

The time responses of the x, y, z-linear velocity of the CoM of the robot's body frame are presented in Figure 5-29. The mean value of all the time responses is almost equal to zero, which is expected since the controller must keep the linear velocities of the robot's body CoM as close to zero as possible. The amplitude of all the time responses is relatively small since it is approximately equal to $10^{-2}\,\mathrm{m/s}$. Therefore, the variations of the linear velocities around the mean value are insignificant.

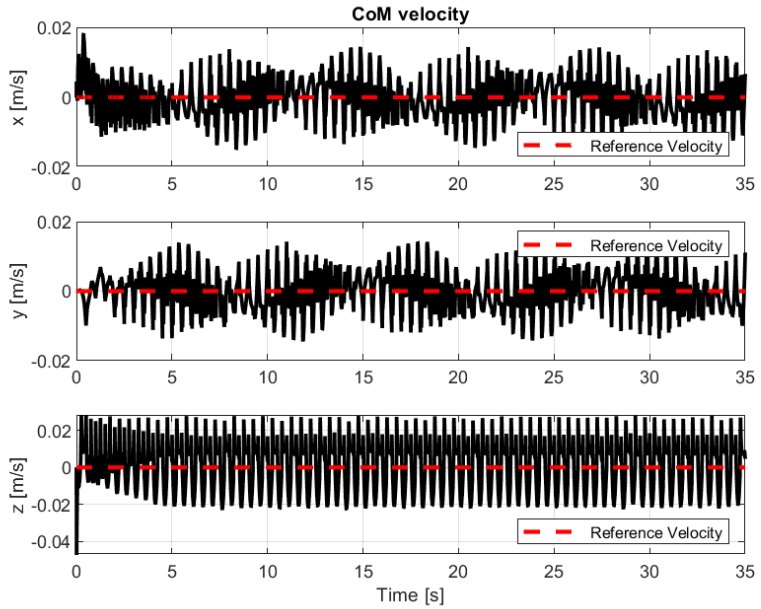


Figure 5-29. The time responses of the x,y,z-linear velocity of the CoM of the robot's body frame, for the trotting gait (rotation).

The time responses of the x, y, z-angular velocity of the robot's body frame are presented in Figure 5-30. The mean value of the z-angular velocity of the robot's body frame after the first

5sec of simulation time is almost equal 0.5rad/s which is equal to the reference. Also, the amplitude of the time response of the z-angular velocity is relatively small since it is approximately equal to 0.03rad/s. Despite that quantitative inaccuracy, the controller is able to track qualitatively correctly the desired speed profile. The mean value of the y,z-angular velocity of the robot's body is close to the reference value and the amplitude of these responses is relatively small but not insignificant since it is equal to 0.1rad/s

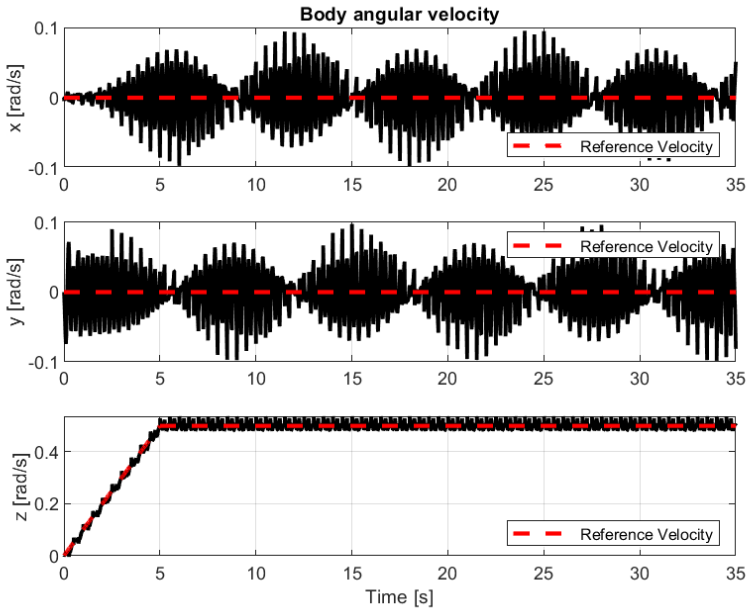


Figure 5-30. The time responses of the x,y,z-angular velocity of the robot's body frame, for the trotting gait (rotation).

The time responses of the angular positions, the angular velocities and the torques of the joints of the FL leg of the quadruped are presented in Figure 5-31, Figure 5-32 and Figure 5-33 respectively. The motors utilized in legged robots can usually achieve maximum angular rates that do not exceed 8rad/s. Thus, the angular velocities of all the joints must not exceed this bound. Also, such actuators can achieve maximum torques that do not exceed 50Nm. The torques of the joint actuators do not exceed this bound. The maximum vertical contact force limit was selected so that this constraint can be satisfied.

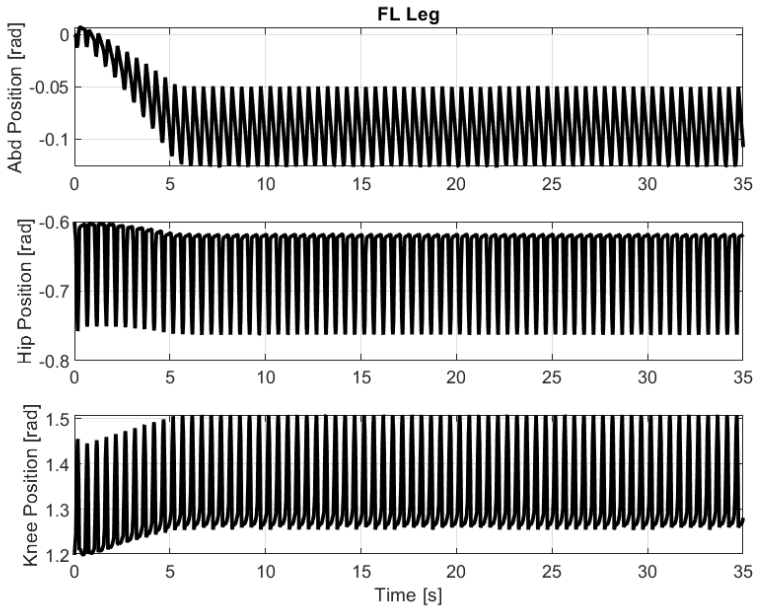


Figure 5-31. Time responses of the angular positions of the joints of the FL leg, for the trotting gait (rotation).

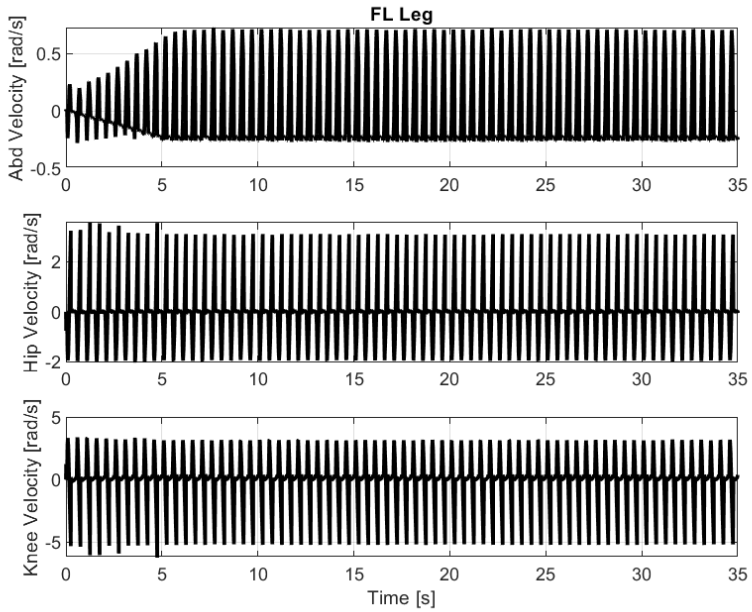


Figure 5-32. Time responses of the angular velocities of the joints of the FL leg, for the trotting gait (rotation).

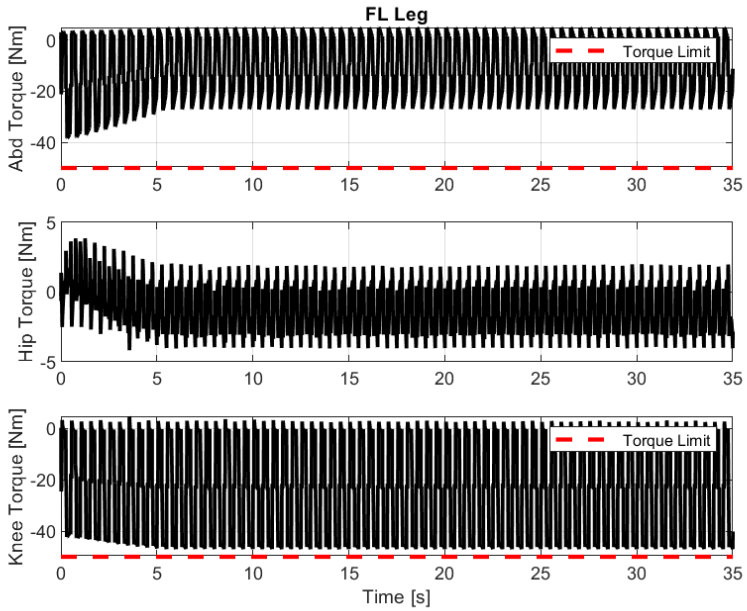


Figure 5-33. Time responses of the torques of the joints of the FL leg, for the trotting gait (rotation).

Finally, the distribution of the solve times of every control loop is presented in the histogram of Figure 5-34. The majority of the solve times (70%) are between 1.0ms and 1.5ms. Also, the mean value of the solve times is equal to 1.4ms and the standard deviation of the solve times is equal to 0.33ms. Thus, the control inputs can be computed very fast, since the value of the mean solve time is small, and at guaranteed rates, since the distribution around that mean value is also small.

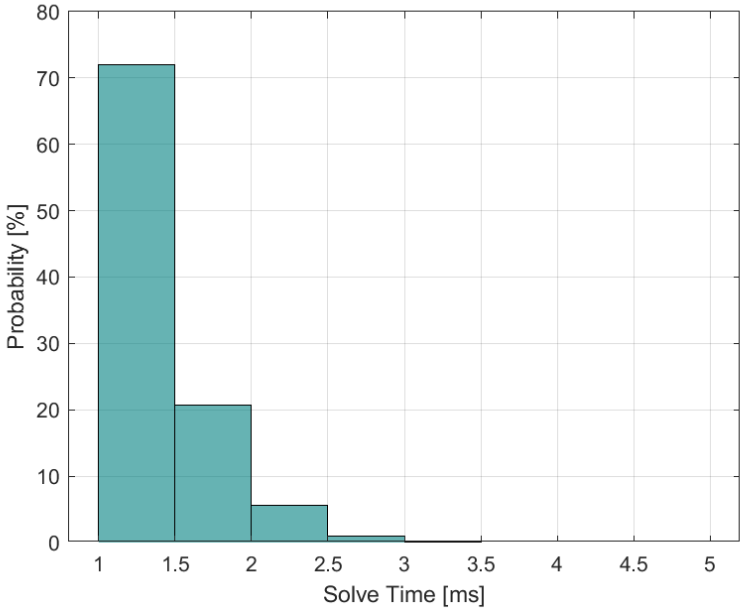


Figure 5-34. Solve time distribution of Convex MPC, for the trotting gait (rotation).

6 Conclusions and Future Work

6.1 Conclusions

In this thesis, an effort was made to build a simulation framework for ARGOS in Gazebo. Throughout the second chapter, the various aspects and features of modelling and simulation of robots in Gazebo were identified and documented in detail. Moreover, the results of the various trials and tests conducted showcased many artifacts that emerged in the simulation. Many of these artifacts were the result of poor contact handling and inaccurate numerical solutions. Without the appropriate parameter selection and tuning, these artifacts emerged, and the yielded simulation results are inaccurate or even physically infeasible. Such selection regarded the collision model selection for the quadruped's feet and the LCP solver selection and tuning regarded mainly the values of the ERP, CFM parameters corresponding to the robot's joints. These modifications resulted in simulation results that were qualitatively correct. The comparisons with the analytical EoM proved that the Gazebo simulation results are also quantitatively correct since the errors were insignificant.

In the third chapter, the various TO methods were presented in detail and were compared with each other by conducting experiments on a simulated double pendulum. These comparisons showcased that each method has its own pros and cons, and that no method inherently the best choice among the rest of them. The selection of the method is dependent upon the application itself and on its specific characteristics. These characteristics include the convergence to the global optimum or to a local one, the form of the cost function, whether the system's dynamics are linear or not, or if they are reduced or full order, whether the optimization problem's constraints are linear or not, the average solution time, the variation of solve times, the horizon length and numerical robustness of the method. The criterion that determined the final section of the TO method was pertinent to how fast and reliably can a TO method solve the optimization problem. The experiments showcased that Convex Optimization exhibits the smallest mean solution time and the smallest variation of the solve times around that mean value. Therefore, a Convex MPC controller was implemented.

However, since legged robots' dynamics are high-dimensional and nonlinear, such TO problems are high-dimensional and non-convex. The high dimension is especially concerning in the context of online TO, where fast computation times are mandatory. The non-convexity is also concerning, since it renders the solver highly sensitive to the initial guess utilized. Thus, in the fourth chapter, various assumptions are made to create a convex optimization problem. These assumptions include using linearized SRBD and linearizing the optimization problems constraints. Such assumptions are valid according to the quadruped design and the specifications of the locomotion tasks that ARGOS will be executing. Also, the QP condensed formulation was utilized since it reduces the size of the problem by eliminating the state trajectory from the decision variables of the optimization problem. The resulting MPC can solve the OCP fast but since the SRBD are utilized the swing foot trajectories need to be planned since the swing leg motion is not computed as part of the decision variables of the optimization problem. As a result, a swing leg controller was also implemented as part of the framework. Also, the contact mode sequence, timings and the footstep locations problem needed to be planned since they were not computed as part of the decision variables of the optimization problem. As a result, a gait scheduler and a footstep planner were also implemented as part of the framework. Finally, the sensor measurements needed for closing

the loop were not estimated but were provided by either ROS sensors or directly by the physics engine itself.

Finally, in the fifth chapter, to evaluate the Convex MPC controller, it was tested on two different gaits, walk and trot, while achieving linear speeds up to $0.5\mathrm{m/s}$ and angular speeds up to $0.5\mathrm{rad/s}$. This approach is general enough since it can execute all these behaviors using the same set of cost function weights and PD controllers' gains. The parameters that differed were the upper and lower bounds on the value of the normal on the ground surface component of the GRF. Trotting gait allows the robot to achieve higher locomotion speeds than with walking gait, while avoiding saturation of its actuators in the process. The MPC was able to successfully stabilize these gaits despite relatively short planning horizon was utilized. The horizon was divided into five steps and its duration was equal to the stance duration of each gait. Even though the actuator torques were bounded implicitly by bounding the normal components of the GRFs, the value of the motor torques obeyed the constraints and did not exceed the limits of $50\mathrm{Nm}$. However, with that constraint the resulting torques are on the conservative side and thus the capabilities of the actuators are not fully harnessed. While the aforementioned locomotion tasks were executed, the controller was able to successfully stabilize the robot's body roll and pitch angles, keep the desired locomotion height constant and track the commanded velocity profile accurately enough. Moreover, it was observed that the controller could track the desired speed profile more accurately when the robot followed the trotting gait rather than when it followed the walking gait. The tracking was more accurate since the mean value of the velocity was closer to the desired one and the variation around that mean value was smaller. Finally, the resultant controller was designed to be portable so that it can be executed on other ROS-compatible robotic simulators apart from Gazebo and on other quadrupeds apart from ARGOS given their urdf description format. This was achieved by using Pinocchio to compute the systems kinematics and dynamics given the *.urdf* file that describes the structure of the quadruped. Therefore, they were not computed using analytical expressions tailor-made for ARGOS.

6.2 Future Work

Although the current implementations of the simulation and control framework have been tested and have yielded satisfying results, there is always room for improvement. The simulation framework for ARGOS could also be built using other real-time robotics simulators like MuJoCo and RaiSim [167]. The frameworks built with MuJoCo and RaiSim can be used to validate furthermore the accuracy of the Gazebo framework by comparing their results with Gazebo's. The MuJoCo simulator is selected since it is a simulator that is particularly popular within the learning community and is commonly used for RL applications on legged robots. A simulation framework in MuJoCo will make it possible to test learning-based strategies on quadruped robots. The RaiSim simulator is selected due to its more accurate contact modelling and its faster computations when compared to the LCP solvers utilized by Gazebo. Most of the simulators make some approximations regarding impact and friction modelling, like friction cone approximation (Gazebo) or relaxation of the complementarity constraint (MuJoCo). Neither of these approximations are made in RaiSim and therefore tends to be more accurate than the other two. More accurate contact simulation will contribute to bridging the gap between simulation and reality.

Another step could be to utilize ROS sensors and a state estimation algorithm to estimate the physical quantities that are not directly provided by the sensors, instead of acquiring them through the physics engine itself. ROS provides models for sensors, like IMUs and encoders, that can be attached to the quadruped, and fuse the data that they provide to estimate physical quantities like the position and the velocity of the robot's body. Additionally, instead of proprioceptive odometry, a visual odometry approach could also be adopted by the proposed framework. Some available state estimation techniques for legged robots were sketched briefly in chapter 4.3.1. Finally, ROS provides noise models for its sensors which can also be utilized. In the real world, sensors exhibit noise, and thus they do not observe the world perfectly. Including such models will also help bridge the gap between simulation and reality significantly. More information about Gazebo's sensor noise models can be found in [168].

The MPC controller for ARGOS can also be implemented using a different TO method like multiple shooting or collocation. These problems can be solved by using state-of-the-art solvers like the FDDP or IPOPT solvers. Such nonlinear optimization techniques will allow the utilization of higher order and nonlinear dynamics or even nonlinear constraints. Such higher order models may include centroidal dynamics plus the full kinematics [55] or even whole-body dynamics. Comparisons between the different MPC formulations will assess whether the loss of physical accuracy, caused by the use of reduced-order models to reduce the computational cost of the optimization, is a trade-off worth accepting. The nonlinear constraints may include imposing leg kinematics limits as a constraint to ensure that the foothold position computed by the optimization problem solver lies inside the leg's workspace or enforcing stricter motor torque limits by using the foot Jacobian while formulating the pertinent constraint. Finally, these nonlinear optimization techniques will allow performing gait discovery where contact sequences, timings and footstep locations are not pre-specified but are computed real-time by the optimization algorithm.

Moreover, the Convex MPC controller can be utilized in conjunction with learning-based approaches. Complementing and improving the performance of model-based approaches with learning-based ones is an active research area. For instance, nonlinear optimization problems can be "warm started" with an initial guess learned from data thus improving the convergence rate of OCP solvers or helping avoid poor local minima (suboptimal solutions). Also, an approximation of the value function, which can function as a terminal cost to enable online TO over shorter horizons while avoiding myopic behavior, can be learned from data. Finally, in the Convex MPC case, learning can be used as a high-level footstep planner. The desired foothold positions can be learned, as in [169], and not be specified by heuristics as was done in this work.

Last but not least, the proposed control framework needs to be tested extensively on in-house hardware. Only through hardware testing one can be sure about the performance of a controller no matter how extensive testing was performed in simulations. Deploying the controller on hardware will be facilitated since it is built using ROS, which is extensively used for real-time applications on real-world robots and is establishing itself as the primary standard in the field of robotics research.

7 References

[1] “Gazebo.” Accessed: Mar. 22, 2022. [Online]. Available: http://gazebosim.org/

[2] “ROS: Home.” Accessed: Mar. 22, 2022. [Online]. Available: https://www.ros.org/

[3] “Atlas,” Boston Dynamics. Accessed: Feb. 29, 2024. [Online]. Available: https://bostondynamics.com/atlas/

[4] “Spot | Boston Dynamics.” Accessed: Feb. 29, 2024. [Online]. Available: https://bostondynamics.com/products/spot/

[5] “Legacy Robots,” Boston Dynamics. Accessed: Feb. 29, 2024. [Online]. Available: https://bostondynamics.com/legacy/

[6] “ANYmal - Autonomous Robotic Inspection Solution - ANYbotic.” Accessed: Feb. 29, 2024. [Online]. Available: https://www.anybotics.com/robotics/anymal/

[7] “MIT Biomimetic Robotics Lab.” Accessed: Feb. 29, 2024. [Online]. Available: https://biomimetics.mit.edu/

[8] “Agility Robotics,” Agility Robotics. Accessed: Feb. 29, 2024. [Online]. Available: https://agilityrobotics.com/digit

[9] J. T. Betts, *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*, Second. Society for Industrial and Applied Mathematics, 2010. doi: 10.1137/1.9780898718577.

[10] D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. M. Scokaert, “Constrained model predictive control: Stability and optimality,” *Automatica*, vol. 36, no. 6, pp. 789–814, Jun. 2000, doi: 10.1016/S0005-1098(99)00214-9.

[11] M. H. Raibert, *Legged Robots that Balance*. MIT Press, 1986.

[12] L. Righetti, J. Buchli, M. Mistry, M. Kalakrishnan, and S. Schaal, “Optimal distribution of contact forces with inverse-dynamics control,” *The International Journal of Robotics Research*, vol. 32, no. 3, pp. 280–298, Mar. 2013, doi: 10.1177/0278364912469821.

[13] M. Hutter, M. A. Hoepflinger, C. Gehring, M. Bloesch, C. D. Remy, and R. Siegwart, “Hybrid Operational Space Control for Compliant Legged Systems,” p. 8 p., 2012, doi: 10.3929/ETHZ-A-010184796.

[14] C. D. Bellicoso, F. Jenelten, P. Fankhauser, C. Gehring, J. Hwangbo, and M. Hutter, “Dynamic locomotion and whole-body control for quadrupedal robots,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2017, pp. 3359–3365.

[15] C. D. Bellicoso, F. Jenelten, C. Gehring, and M. Hutter, “Dynamic locomotion through online nonlinear motion optimization for quadrupedal robots,” *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 2261–2268, 2018.

[16] M. L. Felis, “RBDL: an efficient rigid-body dynamics library using recursive algorithms,” *Auton Robot*, vol. 41, no. 2, pp. 495–511, Feb. 2017, doi: 10.1007/s10514-016-9574-0.

[17] D. Goldfarb and A. Idnani, “A numerically stable dual method for solving strictly convex quadratic programs,” *Mathematical Programming*, vol. 27, no. 1, pp. 1–33, Sep. 1983, doi: 10.1007/BF02591962.

[18] C. Mastalli, I. Havoutis, M. Focchi, D. G. Caldwell, and C. Semini, “Hierarchical planning of dynamic movements without scheduled contact sequences,” in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, Stockholm, Sweden: IEEE, May 2016, pp. 4636–4641. doi: 10.1109/ICRA.2016.7487664.

[19] D. Zimmermann, S. Coros, Y. Ye, R. W. Sumner, and M. Gross, “Hierarchical planning and control for complex motor tasks,” in *Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, Los Angeles California: ACM, Aug. 2015, pp. 73–81. doi: 10.1145/2786784.2786795.

[20] J. Carpentier, S. Tonneau, M. Naveau, O. Stasse, and N. Mansard, “A Versatile and Efficient Pattern Generator for Generalized Legged Locomotion,” in *IEEE International Conference on Robotics and Automation (ICRA), May 2016, Stockholm, Sweden*, Stockholm, Sweden, Feb. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01203507

[21] A. Herzog, N. Rotella, S. Mason, F. Grimminger, S. Schaal, and L. Righetti, “Momentum Control with Hierarchical Inverse Dynamics on a Torque-Controlled Humanoid,” *Auton Robot*, vol. 40, no. 3, pp. 473–491, Mar. 2016, doi: 10.1007/s10514-015-9476-6.
[22] A. W. Winkler, F. Farshidian, M. Neunert, D. Pardo, and J. Buchli, “Online walking motion and foothold optimization for quadruped locomotion,” in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2017, pp. 5308–5313.
[23] A. W. Winkler, F. Farshidian, D. Pardo, M. Neunert, and J. Buchli, “Fast trajectory optimization for legged robots using vertex-based zmp constraints,” *IEEE Robotics and Automation Letters*, vol. 2, no. 4, pp. 2201–2208, 2017.
[24] A. Wächter and L. T. Biegler, “On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming,” *Math. Program.*, vol. 106, no. 1, pp. 25–57, Mar. 2006, doi: 10.1007/s10107-004-0559-y.
[25] “HyQ - Dynamic Legged Systems - IIT.” Accessed: Feb. 29, 2024. [Online]. Available: https://dls.iit.it/hyq
[26] J. Pratt, J. Carff, S. Drakunov, and A. Goswami, “Capture Point: A Step toward Humanoid Push Recovery,” in *2006 6th IEEE-RAS International Conference on Humanoid Robots*, Sep. 2006, pp. 200–207. doi: 10.1109/ICHR.2006.321385.
[27] P. E. Gill, W. Murray, and M. A. Saunders, “SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization,” *SIAM Rev.*, vol. 47, no. 1, pp. 99–131, Jan. 2005, doi: 10.1137/S0036144504446096.
[28] A. W. Winkler, C. D. Bellicoso, M. Hutter, and J. Buchli, “Gait and trajectory optimization for legged systems through phase-based end-effector parameterization,” *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 1560–1567, 2018.
[29] F. Farshidian, E. Jelavic, A. Satapathy, M. Giftthaler, and J. Buchli, “Real-time motion planning of legged robots: A model predictive control approach,” in *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, Birmingham: IEEE, Nov. 2017, pp. 577–584. doi: 10.1109/HUMANOIDS.2017.8246930.
[30] M. Neunert *et al.*, “Whole-Body Nonlinear Model Predictive Control Through Contacts for Quadrupeds,” *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 1458–1465, Jul. 2018, doi: 10.1109/LRA.2018.2800124.
[31] G. Bledt and S. Kim, “Implementing Regularized Predictive Control for Simultaneous Real-Time Footstep and Ground Reaction Force Optimization,” in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Macau, China: IEEE, Nov. 2019, pp. 6316–6323. doi: 10.1109/IROS40897.2019.8968031.
[32] J. Di Carlo, P. M. Wensing, B. Katz, G. Bledt, and S. Kim, “Dynamic Locomotion in the MIT Cheetah 3 Through Convex Model-Predictive Control,” in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Madrid: IEEE, Oct. 2018, pp. 1–9. doi: 10.1109/IROS.2018.8594448.
[33] D. Kim, J. Di Carlo, B. Katz, G. Bledt, and S. Kim, “Highly Dynamic Quadruped Locomotion via Whole-Body Impulse Control and Model Predictive Control.” arXiv, Sep. 14, 2019. Accessed: Oct. 05, 2022. [Online]. Available: http://arxiv.org/abs/1909.06586
[34] Y. Ding, A. Pandala, and H.-W. Park, “Real-time Model Predictive Control for Versatile Dynamic Motions in Quadrupedal Robots,” in *2019 International Conference on Robotics and Automation (ICRA)*, Montreal, QC, Canada: IEEE, May 2019, pp. 8484–8490. doi: 10.1109/ICRA.2019.8793669.
[35] S. P. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge, UK ; New York: Cambridge University Press, 2004.
[36] Y. Wang and S. Boyd, “Fast Model Predictive Control Using Online Optimization,” *IEEE Trans. Contr. Syst. Technol.*, vol. 18, no. 2, pp. 267–278, Mar. 2010, doi: 10.1109/TCST.2009.2017934.
[37] H. J. Ferreau, C. Kirches, A. Potschka, H. G. Bock, and M. Diehl, “qpOASES: a parametric active-set algorithm for quadratic programming,” *Math. Prog. Comp.*, vol. 6, no. 4, pp. 327–363, Dec. 2014, doi: 10.1007/s12532-014-0071-1.

[38] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, “OSQP: an operator splitting solver for quadratic programs,” *Math. Prog. Comp.*, vol. 12, no. 4, pp. 637–672, Dec. 2020, doi: 10.1007/s12532-020-00179-2.
[39] E. M. Gertz and S. J. Wright, “Object-oriented software for quadratic programming,” *ACM Trans. Math. Softw.*, vol. 29, no. 1, pp. 58–81, Mar. 2003, doi: 10.1145/641876.641880.
[40] Y. Ding, A. Pandala, C. Li, Y.-H. Shin, and H.-W. Park, “Representation-Free Model Predictive Control for Dynamic Motions in Quadrupeds,” *IEEE Trans. Robot.*, vol. 37, no. 4, pp. 1154–1171, Aug. 2021, doi: 10.1109/TRO.2020.3046415.
[41] J. Shen and D. Hong, “Convex Model Predictive Control of Single Rigid Body Model on SO(3) for Versatile Dynamic Legged Motions,” in *2022 International Conference on Robotics and Automation (ICRA)*, Philadelphia, PA, USA: IEEE, May 2022, pp. 6586–6592. doi: 10.1109/ICRA46639.2022.9811926.
[42] W. Chi, X. Jiang, and Y. Zheng, “A Linearization of Centroidal Dynamics for the Model-Predictive Control of Quadruped Robots,” in *2022 International Conference on Robotics and Automation (ICRA)*, May 2022, pp. 4656–4663. doi: 10.1109/ICRA46639.2022.9812433.
[43] A. G. Pandala, Y. Ding, and H.-W. Park, “qpSWIFT: A Real-Time Sparse Quadratic Program Solver for Robotic Applications,” *IEEE Robot. Autom. Lett.*, vol. 4, no. 4, pp. 3355–3362, Oct. 2019, doi: 10.1109/LRA.2019.2926664.
[44] D. Mayne, “A Second-order Gradient Method for Determining Optimal Trajectories of Non-linear Discrete-time Systems,” *International Journal of Control*, vol. 3, no. 1, pp. 85–95, Jan. 1966, doi: 10.1080/00207176608921369.
[45] E. Todorov and Weiwei Li, “A generalized iterative LQG method for locally-optimal feedback control of constrained nonlinear stochastic systems,” in *Proceedings of the 2005, American Control Conference, 2005.*, Portland, OR, USA: IEEE, 2005, pp. 300–306. doi: 10.1109/ACC.2005.1469949.
[46] W. Li and E. Todorov, “Iterative linear quadratic regulator design for nonlinear biological movement systems.,” in *ICINCO (1)*, Citeseer, 2004, pp. 222–229.
[47] A. Sideris and J. E. Bobrow, “An Efficient Sequential Linear Quadratic Algorithm for Solving Nonlinear Optimal Control Problems,” p. 6.
[48] Y. Tassa, T. Erez, and E. Todorov, “Synthesis and stabilization of complex behaviors through online trajectory optimization,” in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vilamoura-Algarve, Portugal: IEEE, Oct. 2012, pp. 4906–4913. doi: 10.1109/IROS.2012.6386025.
[49] Y. Tassa, N. Mansard, and E. Todorov, “Control-limited differential dynamic programming,” in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, Hong Kong, China: IEEE, May 2014, pp. 1168–1175. doi: 10.1109/ICRA.2014.6907001.
[50] J. Koenemann *et al.*, “Whole-body model-predictive control applied to the HRP-2 humanoid,” in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Hamburg, Germany: IEEE, Sep. 2015, pp. 3346–3351. doi: 10.1109/IROS.2015.7353843.
[51] M. Neunert, F. Farshidian, A. W. Winkler, and J. Buchli, “Trajectory optimization through contacts and automatic gait discovery for quadrupeds,” *IEEE Robotics and Automation Letters*, vol. 2, no. 3, pp. 1502–1509, 2017.
[52] N. J. Kong, C. Li, and A. M. Johnson, “Hybrid iLQR Model Predictive Control for Contact Implicit Stabilization on Legged Robots.” arXiv, Jul. 10, 2022. Accessed: Oct. 05, 2022. [Online]. Available: http://arxiv.org/abs/2207.04591
[53] C. Mastalli *et al.*, “Agile Maneuvers in Legged Robots: a Predictive Control Approach.” arXiv, Jul. 18, 2022. Accessed: Oct. 10, 2022. [Online]. Available: http://arxiv.org/abs/2203.07554
[54] F. Farshidian, M. Neunert, A. W. Winkler, G. Rey, and J. Buchli, “An efficient optimal planning and control framework for quadrupedal locomotion,” in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2017, pp. 93–100.

[55] H. Dai, A. Valenzuela, and R. Tedrake, “Whole-body motion planning with centroidal dynamics and full kinematics,” in *2014 IEEE-RAS International Conference on Humanoid Robots*, Madrid: IEEE, Nov. 2014, pp. 295–302. doi: 10.1109/HUMANOIDS.2014.7041375.
[56] R. Grandia, F. Farshidian, R. Ranftl, and M. Hutter, “Feedback MPC for Torque-Controlled Legged Robots.” arXiv, Aug. 09, 2019. Accessed: Oct. 05, 2022. [Online]. Available: http://arxiv.org/abs/1905.06144
[57] M. Gaertner, M. Bjelonic, F. Farshidian, and M. Hutter, “Collision-Free MPC for Legged Robots in Static and Dynamic Scenes.” arXiv, Mar. 25, 2021. Accessed: Oct. 05, 2022. [Online]. Available: http://arxiv.org/abs/2103.13987
[58] R. Grandia, A. J. Taylor, A. D. Ames, and M. Hutter, “Multi-Layered Safety for Legged Robots via Control Barrier Functions and Model Predictive Control.” arXiv, Jun. 03, 2021. Accessed: Oct. 10, 2022. [Online]. Available: http://arxiv.org/abs/2011.00032
[59] M. Giftthaler, M. Neunert, M. Stäuble, J. Buchli, and M. Diehl, “A Family of Iterative Gauss-Newton Shooting Methods for Nonlinear Optimal Control.” arXiv, Dec. 11, 2017. Accessed: Oct. 11, 2022. [Online]. Available: http://arxiv.org/abs/1711.11006
[60] M. Frigerio, J. Buchli, and D. G. Caldwell, “Code generation of algebraic quantities for robot controllers,” in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vilamoura-Algarve, Portugal: IEEE, Oct. 2012, pp. 2346–2351. doi: 10.1109/IROS.2012.6385694.
[61] M. Giftthaler, M. Neunert, M. Stäuble, and J. Buchli, “The control toolbox — An open-source C++ library for robotics, optimal and model predictive control,” in *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, May 2018, pp. 123–129. doi: 10.1109/SIMPAR.2018.8376281.
[62] C. Mastalli *et al.*, “Crocoddyl: An Efficient and Versatile Framework for Multi-Contact Optimal Control.” arXiv, Mar. 11, 2020. Accessed: Oct. 11, 2022. [Online]. Available: http://arxiv.org/abs/1909.04947
[63] J. Carpentier *et al.*, “The Pinocchio C++ library : A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives,” in *2019 IEEE/SICE International Symposium on System Integration (SII)*, Paris, France: IEEE, Jan. 2019, pp. 614–619. doi: 10.1109/SII.2019.8700380.
[64] G. Bledt, P. M. Wensing, and S. Kim, “Policy-regularized model predictive control to stabilize diverse quadrupedal gaits for the MIT cheetah,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Vancouver, BC: IEEE, Sep. 2017, pp. 4102–4109. doi: 10.1109/IROS.2017.8206268.
[65] G. Bledt and S. Kim, *Extracting Legged Locomotion Heuristics with Regularized Predictive Control*. 2020. doi: 10.1109/ICRA40945.2020.9197488.
[66] S. H. Jeon, S. Kim, and D. Kim, “Real-time Optimal Landing Control of the MIT Mini Cheetah.” arXiv, Oct. 06, 2021. Accessed: Nov. 12, 2022. [Online]. Available: http://arxiv.org/abs/2110.02799
[67] J. Norby *et al.*, “Quad-SDK: Full Stack Software Framework for Agile Quadrupedal Locomotion”.
[68] D. Pardo, L. Moller, M. Neunert, A. W. Winkler, and J. Buchli, “Evaluating Direct Transcription and Nonlinear Optimization Methods for Robot Motion Planning,” *IEEE Robot. Autom. Lett.*, vol. 1, no. 2, pp. 946–953, Jul. 2016, doi: 10.1109/LRA.2016.2527062.
[69] D. Pardo, M. Neunert, A. W. Winkler, R. Grandia, and J. Buchli, “Hybrid direct collocation and control in the constraint-consistent subspace for dynamic legged robot locomotion.,” in *Robotics: Science and Systems*, 2017.
[70] M. Posa, C. Cantu, and R. Tedrake, “A direct method for trajectory optimization of rigid bodies through contact,” *The International Journal of Robotics Research*, vol. 33, no. 1, pp. 69–81, Jan. 2014, doi: 10.1177/0278364913506757.
[71] R. H. Byrd, J. Nocedal, and R. A. Waltz, “Knitro: An Integrated Package for Nonlinear Optimization,” in *Large-Scale Nonlinear Optimization*, G. Di Pillo and M. Roma, Eds., in

Nonconvex Optimization and Its Applications. , Boston, MA: Springer US, 2006, pp. 35–59. doi: 10.1007/0-387-30065-1_4.
[72] J. Collins, S. Chand, A. Vanderkop, and D. Howard, “A Review of Physics Simulators for Robotic Applications,” *IEEE Access*, vol. 9, pp. 51416–51431, 2021, doi: 10.1109/ACCESS.2021.3068769.
[73] R. C. Kooijman, “Evaluation Of Open Dynamics Engine Software,” p. 69.
[74] “xacro - ROS Wiki.” Accessed: Mar. 22, 2022. [Online]. Available: https://wiki.ros.org/xacro
[75] “Gazebo : Tutorial : Physics Parameters.” Accessed: Mar. 22, 2022. [Online]. Available: http://gazebosim.org/tutorials?tut=physics_params&cat=physics
[76] “Open Dynamics Engine.” Accessed: Mar. 22, 2022. [Online]. Available: https://ode.org/ode-latest-userguide.html
[77] “SDFormat Specification.” Accessed: Apr. 24, 2022. [Online]. Available: http://sdformat.org/spec
[78] R. W. Cottle and G. B. Dantzig, “Complementary pivot theory of mathematical programming,” *Linear Algebra and its Applications*, vol. 1, no. 1, pp. 103–125, Jan. 1968, doi: 10.1016/0024-3795(68)90052-9.
[79] J. M. Hsu and S. C. Peters, “Extending Open Dynamics Engine for the DARPA Virtual Robotics Challenge,” in *Simulation, Modeling, and Programming for Autonomous Robots*, vol. 8810, D. Brugali, J. F. Broenink, T. Kroeger, and B. A. MacDonald, Eds., in Lecture Notes in Computer Science, vol. 8810. , Cham: Springer International Publishing, 2014, pp. 37–48. doi: 10.1007/978-3-319-11900-7_4.
[80] R. W. Cottle, J.-S. Pang, and R. E. Stone, *The Linear Complementarity Problem*. Society for Industrial and Applied Mathematics, 2009. doi: 10.1137/1.9780898719000.
[81] G. Arechavaleta, E. López-Damian, and J. L. Morales, “On the Use of Iterative LCP Solvers for Dry Frictional Contacts in Grasping,” in *International Conference on Advanced Robotics (ICAR 2009)*, Munich, Germany: IEEE, Jun. 2009. Accessed: Apr. 19, 2022. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01309171
[82] M. Anitescu and A. Tasora, “An iterative approach for cone complementarity problems for nonsmooth dynamics,” *Comput Optim Appl*, vol. 47, no. 2, pp. 207–235, Oct. 2010, doi: 10.1007/s10589-008-9223-4.
[83] R. Smith, “How to make new joints in ODE.” Feb. 24, 2002.
[84] E. Drumwright, J. Hsu, N. Koenig, and D. Shell, “Extending Open Dynamics Engine for Robotics Simulation,” in *Simulation, Modeling, and Programming for Autonomous Robots*, vol. 6472, N. Ando, S. Balakirsky, T. Hemker, M. Reggiani, and O. von Stryk, Eds., in Lecture Notes in Computer Science, vol. 6472. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 38–50. doi: 10.1007/978-3-642-17319-6_7.
[85] J. Baumgarte, “Stabilization of constraints and integrals of motion in dynamical systems,” *Computer Methods in Applied Mechanics and Engineering*, vol. 1, no. 1, pp. 1–16, Jun. 1972, doi: 10.1016/0045-7825(72)90018-7.
[86] “Manual - ODE.” Accessed: Feb. 25, 2024. [Online]. Available: https://ode.org/wiki/index.php/Manual
[87] D. E. Stewart, “Rigid-Body Dynamics with Friction and Impact,” *SIAM Rev.*, vol. 42, no. 1, pp. 3–39, Jan. 2000, doi: 10.1137/S0036144599360110.
[88] “Gazebo : Tutorial : Torsional friction.” Accessed: Apr. 24, 2022. [Online]. Available: https://classic.gazebosim.org/tutorials?tut=torsional_friction&cat=physics
[89] V. L. Popov, *Contact Mechanics and Friction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. doi: 10.1007/978-3-642-10803-7.
[90] E. Catto, “Soft Constraints reinventing the spring,” presented at the Game Developer Conference, 2011.
[91] “GIMPACT.” Accessed: May 07, 2022. [Online]. Available: http://gimpact.sourceforge.net/
[92] “Gazebo ODE flickering contact points · Issue #6 · osrf/collision-benchmark,” GitHub. Accessed: Mar. 22, 2022. [Online]. Available: https://github.com/osrf/collision-benchmark/issues/6

[93] K. Hauser, “Robust Contact Generation for Robot Simulation with Unstructured Meshes,” in *Robotics Research*, vol. 114, M. Inaba and P. Corke, Eds., in Springer Tracts in Advanced Robotics, vol. 114. , Cham: Springer International Publishing, 2016, pp. 357–373. doi: 10.1007/978-3-319-28872-7_21.
[94] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics*. in Advanced Textbooks in Control and Signal Processing. London: Springer London, 2009. doi: 10.1007/978-1-84628-642-1.
[95] J. B. Rawlings, D. Q. Mayne, and M. Diehl, *Model predictive control: theory, computation, and design*, 2nd edition. Madison, Wisconsin: Nob Hill Publishing, 2017.
[96] S. Katayama, M. Murooka, and Y. Tazaki, “Model predictive control of legged and humanoid robots: models and algorithms,” *Advanced Robotics*, pp. 1–18, Feb. 2023, doi: 10.1080/01691864.2023.2168134.
[97] R. Bellman, “The theory of dynamic programming,” *Bull. Amer. Math. Soc.*, vol. 60, no. 6, pp. 503–515, 1954, doi: 10.1090/S0002-9904-1954-09848-8.
[98] M. Kelly, “An Introduction to Trajectory Optimization: How to Do Your Own Direct Collocation,” *SIAM Rev.*, vol. 59, no. 4, pp. 849–904, Jan. 2017, doi: 10.1137/16M1062569.
[99] D. Liberzon, *Calculus of Variations and Optimal Control Theory: A Concise Introduction*. USA: Princeton University Press, 2011.
[100] J. T. Betts, “Survey of Numerical Methods for Trajectory Optimization,” *Journal of Guidance, Control, and Dynamics*, vol. 21, no. 2, pp. 193–207, Mar. 1998, doi: 10.2514/2.4231.
[101] M. P. Kelly, “Transcription Methods for Trajectory Optimization: a beginners tutorial.” arXiv, Jul. 02, 2017. Accessed: Oct. 10, 2022. [Online]. Available: http://arxiv.org/abs/1707.00284
[102] P. M. Wensing, M. Posa, Y. Hu, A. Escande, N. Mansard, and A. D. Prete, “Optimization-Based Control for Dynamic Legged Robots,” *IEEE Trans. Robot.*, pp. 1–20, 2023, doi: 10.1109/TRO.2023.3324580.
[103] M. Diehl, H. G. Bock, H. Diedam, and P.-B. Wieber, “Fast Direct Multiple Shooting Algorithms for Optimal Robot Control,” in *Fast Motions in Biomechanics and Robotics*, vol. 340, M. Diehl and K. Mombaur, Eds., in Lecture Notes in Control and Information Sciences, vol. 340. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 65–93. doi: 10.1007/978-3-540-36119-0_4.
[104] A. W. Winkler, “Optimization-based motion planning for legged robots,” ETH Zurich, 2018. doi: 10.3929/ETHZ-B-000272432.
[105] A. M. Johnson, “Trajectory Optimization with Changing Contact Conditions,” presented at the Software Tools for Real-Time Optimal Control Workshop at RSS 2021, 2021. Accessed: Feb. 26, 2024. [Online]. Available: https://www.youtube.com/watch?v=zkgB2aVYraQ
[106] C. Mastalli *et al.*, “A Feasibility-Driven Approach to Control-Limited DDP.” arXiv, Aug. 15, 2022. Accessed: Oct. 10, 2022. [Online]. Available: http://arxiv.org/abs/2010.00411
[107] R. Tedrake, *Underactuated Robotics: Algorithms for Walking, Running, Swimming, Flying, and Manipulation*. 2023. [Online]. Available: https://underactuated.csail.mit.edu
[108] H. G. Bock and K. J. Plitt, “A Multiple Shooting Algorithm for Direct Solution of Optimal Control Problems*,” *IFAC Proceedings Volumes*, vol. 17, no. 2, pp. 1603–1608, Jul. 1984, doi: 10.1016/S1474-6670(17)61205-9.
[109] A. Edelman, “Applied Parallel Computing.” Mathematics 18.337, Computer Science 6.338, SMA 5505, Spring 2004, 2004. Accessed: Feb. 27, 2024. [Online]. Available: https://dspace.mit.edu/bitstream/handle/1721.1/77902/18-337j-spring-2005/contents/lecture-notes/lecture_notes_04.pdf
[110] “HSL - Home Page.” Accessed: Jul. 12, 2023. [Online]. Available: https://www.hsl.rl.ac.uk/
[111] C. R. Hargraves and S. Paris, “Direct Trajectory Optimization Using Nonlinear Programming and Collocation,” *AIAA J. Guidance*, vol. 10, pp. 338–342, Jul. 1987, doi: 10.2514/3.20223.

[112] A. Rao, “A Survey of Numerical Methods for Optimal Control,” *Advances in the Astronautical Sciences*, vol. 135, Jan. 2010.
[113] F. Topputo and C. Zhang, “Survey of Direct Transcription for Low-Thrust Space Trajectory Optimization with Applications,” *Abstract and Applied Analysis*, vol. 2014, pp. 1–15, 2014, doi: 10.1155/2014/851720.
[114] O. von Stryk, “Numerical Solution of Optimal Control Problems by Direct Collocation,” in *Optimal Control*, R. Bulirsch, A. Miele, J. Stoer, and K. Well, Eds., Basel: Birkhäuser Basel, 1993, pp. 129–143. doi: 10.1007/978-3-0348-7539-4_10.
[115] I. M. Ross and F. Fahroo, “Pseudospectral knotting methods for solving nonsmooth optimal control problems,” *Journal of Guidance, Control, and Dynamics*, vol. 27, no. 3, pp. 397–405, 2004.
[116] T. A. Howell, B. E. Jackson, and Z. Manchester, “ALTRO: A Fast Solver for Constrained Trajectory Optimization,” in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Macau, China: IEEE, Nov. 2019, pp. 7674–7679. doi: 10.1109/IROS40897.2019.8967788.
[117] E. Pellegrini and R. P. Russell, “A multiple-shooting differential dynamic programming algorithm. Part 1: Theory,” *Acta Astronautica*, vol. 170, pp. 686–700, May 2020, doi: 10.1016/j.actaastro.2019.12.037.
[118] B. Plancher and S. Kuindersma, “A Performance Analysis of Parallel Differential Dynamic Programming on a GPU,” 2020, pp. 656–672. doi: 10.1007/978-3-030-44051-0_38.
[119] E. Todorov, “Optimal Control Theory (To appear in Bayesian Brain, Doya, K. (ed), MIT Press (2006)).” University of California San Diego.
[120] D. E. Kirk, *Optimal Control Theory: An Introduction*. Prentice-Hall, 1970.
[121] B. E. Jackson, “AL-iLQR Tutorial,” 2019. [Online]. Available: https://bjack205.github.io/papers/AL_iLQR_Tutorial.pdf
[122] M. Sewak, *Deep Reinforcement Learning: Frontiers of Artificial Intelligence*. Singapore: Springer, 2019. doi: 10.1007/978-981-13-8285-7.
[123] L. Liao and C. A. Shoemaker, “Advantages of Differential Dynamic Programming Over Newton’s Method for Discrete-time Optimal Control Problems,” Jul. 1992, Accessed: Jul. 18, 2023. [Online]. Available: https://ecommons.cornell.edu/handle/1813/5474
[124] J. N. Nganga and P. M. Wensing, “Accelerating Second-Order Differential Dynamic Programming for Rigid-Body Systems,” *IEEE Robot. Autom. Lett.*, vol. 6, no. 4, pp. 7659–7666, Oct. 2021, doi: 10.1109/LRA.2021.3098928.
[125] S. Singh, R. P. Russell, and P. M. Wensing, “Analytical Second-Order Partial Derivatives of Rigid-Body Inverse Dynamics.” arXiv, Aug. 14, 2022. Accessed: Nov. 09, 2023. [Online]. Available: http://arxiv.org/abs/2203.01497
[126] L.-Z. Liao and C. A. Shoemaker, “Convergence in unconstrained discrete-time differential dynamic programming,” *IEEE Transactions on Automatic Control*, vol. 36, no. 6, pp. 692–706, Jun. 1991, doi: 10.1109/9.86943.
[127] B. Jackson and T. Howell, “Robotic Exploration Lab, Stanford University”.
[128] J. Marti-Saumell, J. Solà, C. Mastalli, and A. Santamaria-Navarro, “Squash-Box Feasibility Driven Differential Dynamic Programming,” in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Jul. 2020, pp. 7637–7644. doi: 10.1109/IROS45743.2020.9340883.
[129] D. Bertsekas, “Projected Newton Methods for Optimization Problems with Simple Constraints,” presented at the SIAM Journal on Control and Optimization, Mar. 1982, pp. 762–767. doi: 10.1137/0320018.
[130] Yuval, “iLQG/DDP trajectory optimization.” MATLAB Central File Exchange, 2023. Accessed: Aug. 29, 2023. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/52069-ilqg-ddp-trajectory-optimization
[131] J. Nocedal and S. J. Wright, *Numerical optimization*, 2nd ed. in Springer series in operations research. New York: Springer, 2006.
[132] F. Farshidian and others, “OCS2: An open source library for Optimal Control of Switched Systems.”

[133] M. Diehl, “Lecture Notes on Numerical Optimization (Preliminary Draft).” Department of Microsystems Engineering and Department of Mathematics, University of Freiburg, Germany, 2016. Accessed: Feb. 27, 2024. [Online]. Available: https://www.syscop.de/files/2015ws/numopt/numopt_0.pdf
[134] A. A. Ahmadi, “Convex and Conic Optimization, Lecture 9.” Princeton University.
[135] M. S. Lobo, L. Vandenberghe, S. Boyd, and H. Lebret, “Applications of second-order cone programming,” *Linear Algebra and its Applications*, vol. 284, no. 1, pp. 193–228, Nov. 1998, doi: 10.1016/S0024-3795(98)10032-0.
[136] B. Acikmese, J. M. Carson, and L. Blackmore, “Lossless Convexification of Nonconvex Control Bound and Pointing Constraints of the Soft Landing Optimal Control Problem,” *IEEE Trans. Contr. Syst. Technol.*, vol. 21, no. 6, pp. 2104–2113, Nov. 2013, doi: 10.1109/TCST.2012.2237346.
[137] X. Liu, “Fuel-Optimal Rocket Landing with Aerodynamic Controls,” *Journal of Guidance, Control, and Dynamics*, vol. 42, no. 1, pp. 65–77, Jan. 2019, doi: 10.2514/1.G003537.
[138] J. V. Frasch, S. Sager, and M. Diehl, “A parallel quadratic programming method for dynamic optimization problems,” *Math. Prog. Comp.*, vol. 7, no. 3, pp. 289–329, Sep. 2015, doi: 10.1007/s12532-015-0081-7.
[139] J. Mattingley and S. Boyd, “CVXGEN: a code generator for embedded convex optimization,” *Optim Eng*, vol. 13, no. 1, pp. 1–27, Mar. 2012, doi: 10.1007/s11081-011-9176-9.
[140] G. Frison and M. Diehl, “HPIPM: a high-performance quadratic programming framework for model predictive control,” *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 6563–6569, 2020, doi: 10.1016/j.ifacol.2020.12.073.
[141] G. Guennebaud, B. Jacob, and others, “Eigen v3.” 2010. [Online]. Available: http://eigen.tuxfamily.org
[142] R. Featherstone, “The Calculation of Robot Dynamics Using Articulated-Body Inertias,” *The International Journal of Robotics Research*, vol. 2, no. 1, pp. 13–30, Mar. 1983, doi: 10.1177/027836498300200102.
[143] R. Featherstone, *Rigid Body Dynamics Algorithms*. Boston, MA: Springer US, 2008. doi: 10.1007/978-1-4899-7560-7.
[144] M. Neunert *et al.*, “Fast nonlinear Model Predictive Control for unified trajectory optimization and tracking,” in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, Stockholm: IEEE, May 2016, pp. 1398–1404. doi: 10.1109/ICRA.2016.7487274.
[145] S. Teng, D. Chen, W. Clark, and M. Ghaffari, “An Error-State Model Predictive Control on Connected Matrix Lie Groups for Legged Robot Control.” arXiv, Mar. 16, 2022. Accessed: Oct. 24, 2022. [Online]. Available: http://arxiv.org/abs/2203.08728
[146] K. Lowrey, A. Rajeswaran, S. Kakade, E. Todorov, and I. Mordatch, “Plan Online, Learn Offline: Efficient Learning and Exploration via Model-Based Control,” in *International Conference on Learning Representations (ICLR)*, 2019.
[147] E. K.-W. Chu, H.-Y. Fan, W.-W. Lin, and C.-S. Wang, “Structure-Preserving Algorithms for Periodic Discrete-Time Algebraic Riccati Equations,” *International Journal of Control*, vol. 77, no. 8, pp. 767–788, May 2004, doi: 10.1080/00207170410001714988.
[148] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, “Automatic differentiation in machine learning: a survey.” arXiv, Feb. 05, 2018. Accessed: Aug. 09, 2023. [Online]. Available: http://arxiv.org/abs/1502.05767
[149] “osqp-eigen.” Robotology, Aug. 04, 2023. Accessed: Aug. 06, 2023. [Online]. Available: https://github.com/robotology/osqp-eigen
[150] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, “CasADi – A software framework for nonlinear optimization and optimal control,” *Mathematical Programming Computation*, vol. 11, no. 1, pp. 1–36, 2019, doi: 10.1007/s12532-018-0139-4.
[151] A. Wächter and L. T. Biegler, “Line Search Filter Methods for Nonlinear Programming: Local Convergence,” *SIAM Journal on Optimization*, vol. 16, no. 1, pp. 32–48, 2005, doi: 10.1137/S1052623403426544.

[152] G. Frison and J. B. Jørgensen, “Efficient implementation of the Riccati recursion for solving linear-quadratic control problems,” in *2013 IEEE International Conference on Control Applications (CCA)*, 2013, pp. 1117–1122. doi: 10.1109/CCA.2013.6662901.
[153] B. Brogliato, A. ten Dam, L. Paoli, F. Ge´not, and M. Abadie, “Numerical simulation of finite dimensional multibody nonsmooth mechanical systems,” *Applied Mechanics Reviews*, vol. 55, no. 2, pp. 107–150, Mar. 2002, doi: 10.1115/1.1454112.
[154] E. Todorov, T. Erez, and Y. Tassa, “MuJoCo: A physics engine for model-based control,” in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vilamoura-Algarve, Portugal: IEEE, Oct. 2012, pp. 5026–5033. doi: 10.1109/IROS.2012.6386109.
[155] B. Colson, P. Marcotte, and G. Savard, “An overview of bilevel optimization,” *Annals of Operations Research*, vol. 153, no. 1, pp. 235–256, 2007.
[156] B. Landry, J. Lorenzetti, Z. Manchester, and M. Pavone, “Bilevel Optimization for Planning through Contact: A Semidirect Method.” arXiv, Sep. 05, 2019. Accessed: Jul. 26, 2023. [Online]. Available: http://arxiv.org/abs/1906.04292
[157] T. A. Howell, S. Le Cleac’h, S. Singh, P. Florence, Z. Manchester, and V. Sindhwani, “Trajectory Optimization with Optimization-Based Dynamics,” *IEEE Robot. Autom. Lett.*, vol. 7, no. 3, pp. 6750–6757, Jul. 2022, doi: 10.1109/LRA.2022.3152696.
[158] J. Carius, R. Ranftl, V. Koltun, and M. Hutter, “Trajectory Optimization With Implicit Hard Contacts,” *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3316–3323, Oct. 2018, doi: 10.1109/LRA.2018.2852785.
[159] P. Agarwal *et al.*, “State Estimation for Legged Robots: Consistent Fusion of Leg Kinematics and IMU,” in *Robotics: Science and Systems VIII*, 2013, pp. 17–24.
[160] S. Yang, Z. Zhang, B. Bokser, and Z. Manchester, “Multi-IMU Proprioceptive Odometry for Legged Robots,” in *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Detroit, MI, USA: IEEE, Oct. 2023, pp. 774–779. doi: 10.1109/IROS55552.2023.10342061.
[161] “UnitreeRobotics-More than Wisdom, Be with You,” UnitreeRobotics. Accessed: Feb. 29, 2024. [Online]. Available: https://shop.unitree.com/
[162] G. Bledt, M. J. Powell, B. Katz, J. Di Carlo, P. M. Wensing, and S. Kim, “MIT Cheetah 3: Design and Control of a Robust, Dynamic Quadruped Robot,” in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Madrid: IEEE, Oct. 2018, pp. 2245–2252. doi: 10.1109/IROS.2018.8593885.
[163] G. Bledt, P. M. Wensing, S. Ingersoll, and S. Kim, “Contact Model Fusion for Event-Based Locomotion in Unstructured Terrains,” in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, Brisbane, QLD: IEEE, May 2018, pp. 4399–4406. doi: 10.1109/ICRA.2018.8460904.
[164] J. C. Trinkle, J.-S. Pang, S. Sudarsky, and G. Lo, “On Dynamic Multi-Rigid-Body Contact Problems with Coulomb Friction,” *Z. angew. Math. Mech.*, vol. 77, no. 4, pp. 267–279, 1997, doi: 10.1002/zamm.19970770411.
[165] J. L. Jerez, E. C. Kerrigan, and G. A. Constantinides, “A condensed and sparse QP formulation for predictive control,” in *IEEE Conference on Decision and Control and European Control Conference*, Orlando, FL, USA: IEEE, Dec. 2011, pp. 5217–5222. doi: 10.1109/CDC.2011.6160293.
[166] C. Sanderson and R. Curtin, “Armadillo: a template-based C++ library for linear algebra,” *JOSS*, vol. 1, no. 2, p. 26, Jun. 2016, doi: 10.21105/joss.00026.
[167] J. Hwangbo, J. Lee, and M. Hutter, “Per-Contact Iteration Method for Solving Contact Dynamics,” *IEEE Robot. Autom. Lett.*, vol. 3, no. 2, pp. 895–902, Apr. 2018, doi: 10.1109/LRA.2018.2792536.
[168] “Gazebo : Tutorial : Sensor Noise Model.” Accessed: Dec. 22, 2023. [Online]. Available: https://classic.gazebosim.org/tutorials?tut=sensor_noise
[169] S. Gangapurwala, M. Geisert, R. Orsolino, M. Fallon, and I. Havoutis, “RLOC: Terrain-Aware Legged Locomotion Using Reinforcement Learning and Optimal Control,” *IEEE Trans. Robot.*, vol. 38, no. 5, pp. 2908–2927, Oct. 2022, doi: 10.1109/TRO.2022.3172469.

Appendix A

The Joint Space Inertia matrix $\mathbf{M} \in \mathbb{R}^{2\times 2}$ is a symmetric positive definite matrix that has the following form:

$$\mathbf{M} = \begin{bmatrix} I_1 + m_1 d_1^{\,2} + m_2 l_1^{\,2} & m_2 l_1 d_2 \cos\left(\theta_1 - \theta_2 + \beta\right) \\ m_2 l_1 d_2 \cos\left(\theta_1 - \theta_2 + \beta\right) & I_2 + m_2 d_2^{\,2} \end{bmatrix} \tag{A-1}$$

The Coriolis and centripetal terms vector $\mathbf{c} \in \mathbb{R}^{2\times 1}$ has the following form:

$$\mathbf{c} = \begin{bmatrix} m_2 l_1 d_2 \dot{\theta}_2^{\,2} \sin\left(\theta_1 - \theta_2 + \beta\right) \\ -m_2 l_1 d_2 \dot{\theta}_1^{\,2} \sin\left(\theta_1 - \theta_2 + \beta\right) \end{bmatrix} \tag{A-2}$$

The gravity terms vector $\mathbf{G} \in \mathbb{R}^{2\times 1}$ has the following form:

$$\mathbf{G} = \begin{bmatrix} \left(m_1 d_1 + m_2 l_1\right)\sin\left(\theta_1\right) \\ -m_2 d_2 \sin\left(\beta - \theta_2\right) \end{bmatrix} \tag{A-3}$$

The selection matrix $\mathbf{S} \in \mathbb{R}^{2\times 2}$ has the following form:

$$\mathbf{S} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \tag{A-4}$$

The contact Jacobian matrix $\mathbf{J} \in \mathbb{R}^{2\times 2}$ has the following form:

$$\begin{aligned} J_{11} &= l_1 \cos\left(\theta_1\right) \\ J_{21} &= l_1 \sin\left(\theta_1\right) \end{aligned} \tag{A-5}$$

$$\begin{aligned} J_{12} &= \cos\left(\gamma - \theta_2\right)\sqrt{\frac{\frac{r^2\cos\left(2\gamma - 2\theta_2\right)}{2} + l_2^{\,2} + \frac{r^2}{2} + 2 l_2 r \cos\left(\gamma - \theta_2\right)}{l_2^{\,2} + 2\cos\left(\gamma - \theta_2\right) l_2 r + r^2}} \cdot \\ &\quad \sqrt{l_2^{\,2} + 2\cos\left(\gamma - \theta_2\right) l_2 r + r^2} + \frac{r}{2} - \frac{r\cos\left(2\gamma - 2\theta_2\right)}{2} \\ J_{22} &= -\sin\left(\gamma - \theta_2\right)\sqrt{\frac{\frac{r^2\cos\left(2\gamma - 2\theta_2\right)}{2} + l_2^{\,2} + \frac{r^2}{2} + 2 l_2 r \cos\left(\gamma - \theta_2\right)}{l_2^{\,2} + 2\cos\left(\gamma - \theta_2\right) l_2 r + r^2}} \cdot \\ &\quad \sqrt{l_2^{\,2} + 2\cos\left(\gamma - \theta_2\right) l_2 r + r^2} + \frac{r\sin\left(2\gamma - 2\theta_2\right)}{2} \end{aligned} \tag{A-6}$$

Appendix B

Gazebo Functions

GetVelocity

```
virtual double GetVelocity(unsigned int _index);
```

Get the rotation rate of an axis of a joint (index).

Position

```
virtual double Position(const unsigned int _index = 0);
```

Get the position of an axis of a joint according to its index.

SetForce

```
virtual void SetForce(unsigned int _index, double _effort);
```

Set the force applied to this joint. The force applied is additive, meaning that multiple calls to SetForce to the same joint in the same time step will accumulate forces on that joint.

WorldAngularVel

```
virtual ignition::math::Vector3d WorldAngularVel()const
```

Get the angular velocity of the entity in the world frame.

WorldCoGLinearVel

```
virtual ignition::math::Vector3d WorldCoGLinearVel()const
```

Get the linear velocity at the body's center of gravity in the world frame.

WorldCoGPose

```
ignition::math::Pose3d WorldCoGPose()const
```

Get the pose of the body's center of gravity in the world frame.

Pinocchio Functions

aba

```
const DataTpl<Scalar, Options, JointCollectionTpl>::TangentVectorType&
aba(const ModelTpl<Scalar, Options, JointCollectionTpl>& model,
    DataTpl<Scalar, Options, JointCollectionTpl>& data,
    const Eigen::MatrixBase<ConfigVectorType>& q,
    const Eigen::MatrixBase<TangentVectorType1>& v,
    const Eigen::MatrixBase<TangentVectorType2>& tau)
```

Computes the joint accelerations given the current state and actuation of the model using the ABA.

addFrame

```
FrameIndex addFrame(const Frame& frame, const bool append_inertia = true)
```

Add a frame to the robot’s kinematic tree.

addJoint

```
JointModelDerived& addJoint(const JointModelBase<JointModel>& jmodel,
                            const SE3& placement = SE3::Identity())
```

Add a joint to the robot’s model at a given placement.

buildModel [1/2]

```
ModelTpl<Scalar, Options, JointCollectionTpl>&
pinocchio::urdf::buildModel(
    const std::string& filename,
    ModelTpl<Scalar, Options, JointCollectionTpl>& model,
    const bool verbose = false)
```

Build the dynamic model of the robot from a URDF file with a fixed joint as root of the model tree.

buildModel [2/2]

```
ModelTpl<Scalar, Options, JointCollectionTpl>&
pinocchio::urdf::buildModel(
    const std::string& filename,
    const typename ModelTpl<Scalar, Options,
                            JointCollectionTpl>::JointModel& rootJoint,
    ModelTpl<Scalar, Options, JointCollectionTpl>& model,
    const bool verbose = false)
```

Build the dynamic model of the robot from a URDF file with a particular joint as root of the model tree given as a reference argument.

ccrba

```
const DataTpl<Scalar, Options, JointCollectionTpl>::Matrix6x&
pinocchio::ccrba(
    const ModelTpl<Scalar, Options, JointCollectionTpl>& model,
    DataTpl<Scalar, Options, JointCollectionTpl>& data,
    const Eigen::MatrixBase<ConfigVectorType>& q,
    const Eigen::MatrixBase<TangentVectorType>& v)
```

Computes the Centroidal Momentum Matrix, the Composite Rigid Body Inertia as well as the centroidal momenta according to the current joint configuration and velocity.

computeJointJacobians

```
const DataTpl<Scalar, Options, JointCollectionTpl>::Matrix6x&
pinocchio::computeJointJacobians(
    const ModelTpl<Scalar, Options, JointCollectionTpl>& model,
    DataTpl<Scalar, Options, JointCollectionTpl>& data,
    const Eigen::MatrixBase<ConfigVectorType>& q)
```

Computes the full model Jacobian expressed in the world frame. This Jacobian does not correspond to any specific joint frame Jacobian. From this Jacobian, it is then possible to easily extract the Jacobian of a specific joint frame.

computeJointJacobiansTimeVariation

```
const DataTpl<Scalar, Options, JointCollectionTpl>::Matrix6x&
pinocchio::computeJointJacobiansTimeVariation(
    const ModelTpl<Scalar, Options, JointCollectionTpl>& model,
    DataTpl<Scalar, Options, JointCollectionTpl>& data,
    const Eigen::MatrixBase<ConfigVectorType>& q,
    const Eigen::MatrixBase<TangentVectorType>& v)
```

Computes the full model Jacobian variations with respect to time.

crba

```
const DataTpl<Scalar, Options, JointCollectionTpl>::MatrixXs&
pinocchio::crba(
    const ModelTpl<Scalar, Options, JointCollectionTpl>& model,
    DataTpl<Scalar, Options, JointCollectionTpl>& data,
```

```
    const Eigen::MatrixBase<ConfigVectorType>& q)
```
Computes the upper triangular part of the joint space inertia matrix by using the CRBA.

forwardKinematics
```
void pinocchio::forwardKinematics(
    const ModelTpl<Scalar, Options, JointCollectionTpl>& model,
    DataTpl<Scalar, Options, JointCollectionTpl>& data,
    const Eigen::MatrixBase<ConfigVectorType>& q)
```
Update the joint placements according to the current joint configuration.

getFrameJacobian
```
void pinocchio::getFrameJacobian(
    const ModelTpl<Scalar, Options, JointCollectionTpl>& model,
    DataTpl<Scalar, Options, JointCollectionTpl>& data,
    const FrameIndex frame_id, const ReferenceFrame rf,
    const Eigen::MatrixBase<Matrix6xLike>& J)
```
Returns the Jacobian of the frame expressed in the coordinate system specified by the value of value of rf.

getFrameJacobianTimeVariation
```
void pinocchio::getFrameJacobianTimeVariation(
    const ModelTpl<Scalar, Options, JointCollectionTpl>& model,
    DataTpl<Scalar, Options, JointCollectionTpl>& data,
    const FrameIndex frame_id, const ReferenceFrame rf,
    const Eigen::MatrixBase<Matrix6xLike>& dJ)
```
Returns the Jacobian time variation of the frame expressed in the coordinate system specified by the value of value of rf.

getFrameId
```
FrameIndex getFrameId(
    const std::string& name,
    const FrameType& type = (FrameType)(JOINT | FIXED_JOINT | BODY |
                                        OP_FRAME | SENSOR)) const
```
Get the index of a frame given by its name.

getJointId
```
JointIndex getJointId(const std::string& name) const
```
Get the index of a joint given by its name.

nonLinearEffects
```
const DataTpl<Scalar, Options, JointCollectionTpl>::TangentVectorType&
pinocchio::nonLinearEffects(
    const ModelTpl<Scalar, Options, JointCollectionTpl>& model,
    DataTpl<Scalar, Options, JointCollectionTpl>& data,
    const Eigen::MatrixBase<ConfigVectorType>& q,
    const Eigen::MatrixBase<TangentVectorType>& v)
```
Computes the non-linear effects (Coriolis, centrifugal and gravitational effects), also called the bias terms of the Lagrangian dynamics.

updateFramePlacements
```
void pinocchio::updateFramePlacements(
    const ModelTpl<Scalar, Options, JointCollectionTpl>& model,
    DataTpl<Scalar, Options, JointCollectionTpl>& data)
```

Updates the position of each frame contained in the model.

RBDL Functions

ForwardDynamics

```
void ForwardDynamics(Model& model, const Math::VectorNd& Q,
                     const Math::VectorNd& QDot,
                     const Math::VectorNd& Tau, Math::VectorNd& QDDot,
                     std::vector<Math::SpatialVector>* f_ext = NULL);
```

Computes the joint accelerations given the current state and actuation of the model using the ABA.

URDFReadFromFile

```
bool URDFReadFromFile(const char* filename, Model* model,
                      bool floating_base, bool verbose = false);
```

This function loads a URDF model from a file and assigns it to the referenced model.

RobotDynamics Class

BuildPinocchioModel

```
void RobotDynamics::BuildPinocchioModel()
```

Builds a Pinocchio Model and Data object, that are also compatible with the automatic differentiation framework of CppAD, using the URDF model of the robot.

ContinuousDynamics

```
void RobotDynamics::ContinuousDynamics(ADConfigVectorType& ad_x,
                                       ADTangentVectorType& ad_u,
                                       ADTangentVectorType& ad_xdot)
```

Computes the continuous dynamics, more specifically the derivative of the states, of a given model and for a particular state and input using Pinocchio functions.

DiscreteDynamicsRK4

```
void RobotDynamics::DiscreteDynamicsRK4(double& planning_timestep,
                                        VectorNx& x_k, VectorNu& u_k,
                                        ADTangentVectorType& ad_f_k,
                                        VectorNx& f_k)
```

Discretizes the continuous time dynamics, computed at a particular state and input, using the explicit RK4 integrator.

LinearizedDiscreteDynamics

```
void RobotDynamics::LinearizedDiscreteDynamics(double& planning_timestep,
                                               VectorNx& x_k,
                                               VectorNu& u_k,
                                               MatrixNxNx& A_k,
                                               MatrixNxNu& B_k)
```

Linearizes the discretized dynamics about a given reference state and input vector.

QP Class

ComputeQPConstraintMatrix

```
void QP::ComputeQPConstraintMatrix(int planning_horizon_steps,
```

```
                                        MatrixNxNx& A, MatrixNxNu& B,
                                        SpMat& Ac)
```

Calculates the linear constraints matrix of the QP.

ComputeQPConstraintVectors

```
void QP::ComputeQPConstraintVectors(int planning_horizon_steps,
                                    VectorNx& xmin, VectorNx& xmax,
                                    VectorNu& umin, VectorNu& umax,
                                    Vector_constr& lb, Vector_constr& ub)
```

Calculates the bounds of the linear constraints of the QP.

ComputeQPGradient

```
void QP::ComputeQPGradient(int planning_horizon_steps, MatrixNxNx& Q,
                           MatrixNuNu& R, MatrixNxNx& P, VectorNx& x_ref,
                           VectorNu& u_ref, Vector_sol& q)
```

Calculates the gradient vector of the cost function of the QP.

ComputeQPHessian

```
void QP::ComputeQPHessian(int planning_horizon_steps, MatrixNxNx& Q,
                          MatrixNuNu& R, MatrixNxNx& P, SpMat& H)
```

Calculates the Hessian matrix of the cost function of the QP.

BoxDDP Class

BackwardPass

```
void BoxDDP::BackwardPass(MatrixNxNh& X_traj, MatrixNuNh_1& U_traj,
                          MatrixNuNh_1& d, MatrixNuNxNh_1& K)
```

This function performs the backward pass needed by the BOX-DDP algorithm.

BoxDDPSolver

```
void BoxDDP::BoxDDPSolver(MatrixNxNh& X_old, MatrixNuNh_1& U_old,
                          EigenDouble& J_old, MatrixNxNh& X_new,
                          MatrixNuNh_1& U_new, EigenDouble& J_new)
```

This function contains the main body of the BOX-DDP algorithm.

ComputeCost

```
void BoxDDP::ComputeCost(MatrixNxNh& X_traj, MatrixNuNh_1& U_traj,
                         EigenDouble& J)
```

Computes the discretized cost function that is necessary for the BOX-DDP algorithm.

ForwardPass

```
void BoxDDP::ForwardPass(MatrixNxNh& X_old, MatrixNuNh_1& U_old,
                         EigenDouble& J_old, MatrixNuNh_1& d,
                         MatrixNuNxNh_1& K, MatrixNxNh& X_new,
                         MatrixNuNh_1& U_new, EigenDouble& J_new)
```

This function performs the forward pass needed by the BOX-DDP algorithm.

SymbolicRobotDynamics Class

BuildRBDLModel

```
void SymbolicRobotDynamics::BuildRBDLModel()
```

This function builds a RBDL model object using the URDF model of the robot.

ContinousDynamicsRBDL

```
void SymbolicRobotDynamics::ContinuousDynamicsRBDL(VectorNd& q,
                                                   VectorNd& q_dot,
                                                   VectorNd& u,
                                                   casadi::MX& x_dot)
```

Computes the continuous dynamics, more specifically the derivative of the states, of a given model and for a particular state and input using RBDL functions. It creates a CasADi function that represents the dynamics of the system and can handle symbolic variables as arguments and outputs. Here the input state is a CasADi symbolic variable, and the output derivative of the state is also a CasADi symbolic variable.

fd

```
casadi::MX SymbolicRobotDynamics::fd(const VectorNd& Q,
                                     const VectorNd& Qdot,
                                     const VectorNd& Tau)
```

Computes the continuous dynamics, more specifically the derivative of the states, of a given model and for a particular state and input, using RBDL functions. Here the input state is a double vector. The output derivative of the state is also a double vector.

NLP Class

ComputeNLPConstraintBounds

```
void NLP::ComputeNLPConstraintBounds(casadi::MX& constraint,
                                     std::vector<double>& lbg,
                                     std::vector<double>& ubg)
```

Computes the bounds of the constraints of the NLP.

ComputeNLPCostSimpson

```
void NLP::ComputeNLPSimpson(int planning_horizon_steps,
                            double& planning_timestep, casadi::MX& X,
                            casadi::MX& U, casadi::MX& x_goal,
                            casadi::MX& Q, casadi::MX& R,
                            casadi::MX& Qn, casadi::MX& J)
```

This function integrates the cost function using the Simpson integration rule and is utilized by the DIRCOL algorithm.

ComputeNLPCostTrapezoidal

```
void NLP::ComputeNLPCostTrapezoidal(int planning_horizon_steps,
                                    double& planning_timestep,
                                    casadi::MX& X, casadi::MX& U,
                                    casadi::MX& x_goal, casadi::MX& Q,
                                    casadi::MX& R, casadi::MX& Qn,
                                    casadi::MX& J)
```

This function integrates the cost function using the trapezoid integration rule and is utilized by the DIRTRAN algorithm.

ComputeNLPDecisionVariablesBounds

```
void NLP::ComputeNLPDecisionVariablesBounds(
    int planning_horizon_steps, std::vector<double>& x0, VectorNx& xmax,
    VectorNx& xmin, VectorNu& umax, VectorNu& umin,
```

```
    std::vector<double>& lbz, std::vector<double>& ubz)
```
Computes the bounds of the decision variables of the NLP.

ComputeNLPDefectConstraintHermiteSimpson

```
void NLP::ComputeNLPDefectConstraintHermiteSimpson(
    int planning_horizon_steps, double& planning_timestep, casadi::MX& X,
    casadi::MX& U, casadi::MX& constraint)
```
Computes the dynamical system constraint using Hermite-Simpson collocation, in defect form, needed by the DIRCOL algorithm.

ComputeNLPDefectConstraintTrapezoidal

```
void NLP::ComputeNLPDefectConstraintTrapezoidal(
    int planning_horizon_steps, double& planning_timestep, casadi::MX& X,
    casadi::MX& U, casadi::MX& constraint)
```
Computes the dynamical system constraint using the trapezoid integration rule, in defect form, needed by the DIRTRAN algorithm.

Robot Class

AbdPositionsInBaseFrame

```
void Robot::AbdPositionsInBaseFrame()
```
This function computes the position of the abduction joints of the robot's legs expressed in the robot's body frame.

BaseRollPitchYawRate

```
void Robot::BaseRollPitchYawRate(Vector3d& orientation_rpy,
                                 Vector3d& angular_velocity,
                                 Eigen::Vector3d& rpy_rate)
```
This function converts the angular velocity of the robot's base to rate of orientation change of the robot's base in Euler angles.

BaseVelocityInBodyFrame

```
void Robot::BaseVelocityInBodyFrame(
    Vector3d& orientation_rpy, Vector3d& linear_vel_b,
    Eigen::Vector3d& relative_linear_vel_b)
```
This function converts the linear velocity of the robot's body CoM expressed in the world frame to the base frame of the robot.

BuildPinocchioModel

```
void Robot::BuildPinocchioModel(Eigen::VectorXd& q)
```
This function builds a Pinocchio Model and Data object using the URDF model of the robot.

ComputeInertiaMatrix

```
void Robot::ComputeInertiaMatrix(Eigen::VectorXd& q,
                                 Eigen::MatrixXd& mass_matrix)
```
Computes the joint space inertia matrix of the robot using the CRBA.

ComputeBiasTerms

```
void Robot::ComputeBiasTerms(Eigen::VectorXd& q, Eigen::VectorXd& q_dot,
                             Eigen::VectorXd& bias_terms)
```

Computes the bias terms of the Lagrangian dynamics (Coriolis, centrifugal and gravitational effects) of the robot.

ComputeFootJacobian

```
void Robot::ComputeFootJacobian(Eigen::VectorXd& q, int& leg_id,
                                Eigen::MatrixXd& Jac)
```

Computes current foot Jacobian of leg leg_id.

ComputeFootJacobianTimeVariation

```
void Robot::ComputeFootJacobianTimeVariation(Eigen::VectorXd& q,
                                             Eigen::VectorXd& q_dot,
                                             int& leg_id,
                                             Eigen::MatrixXd& Jac_dot)
```

Computes current time variation of the foot Jacobian of leg leg_id.

ComputeJointAnglesFromFootLocalPosition

```
void Robot::ComputeJointAnglesFromFootLocalPosition(
    int& leg_id, Eigen::VectorXd& q,
    Vector3d& foot_local_position_base_frame, Vector3d& joint_ids_leg_id,
    Vector3d& joint_angles_leg_id)
```

This function uses the inverse kinematics to compute the joint angles, given the foot's local position.

ConvertPositionWorldToBaseFrame

```
void Robot::ConvertPositionWorldToBaseFrame(Eigen::VectorXd& q,
                                            Vector3d& pos_I,
                                            Vector3d& pos_B)
```

Converts a position vector expressed w.r.t. the world frame to a vector expressed w.r.t. the base frame of the robot.

FootPositionsInBaseFrame

```
void Robot::FootPositionsInBaseFrame(
    Eigen::VectorXd& q, Matrix3x4& foot_positions_in_base_frame)
```

This function calculates the robot's feet positions w.r.t the base frame of the robot.

FootPositionsLegidInBaseFrame

```
void Robot::FootPositionsLegidInBaseFrame(
    int& leg_id, Eigen::VectorXd& q,
    Vector3d& foot_positions_in_base_frame)
```

This function calculates the foot positions of the robot's leg leg_id w.r.t the base frame of the robot.

GenerateSpeedProfile

```
void Robot::GenerateSpeedProfile(double& time, Vector2d& desired_speed,
                                 double& desired_twisting_speed)
```

Gets the desired linear and angular velocity from the speed profile for the given time.

HipPositionsInBaseFrame

```
void Robot::HipPositionsInBaseFrame()
```

This function computes the position of the hip joints of the robot's legs expressed in the robot's body frame.

HipPositionsLegidInWorldFrame

```
void Robot::HipPositionsLegidInWorldFrame(
    int& leg_id, Eigen::VectorXd& q,
    Vector3d& hip_positions_in_world_frame)
```

This function computes the position of the hip joint of the robot's leg leg_id expressed in the world frame.

InverseKinematics

```
void Robot::InverseKinematics(Vector3d& foot_local_position_abd_frame,
                              int& leg_id, Vector3d& joint_angles_leg_id)
```

This function implements an inverse kinematics algorithm to compute the joint positions from the end effector position of leg leg_id.

MapContactForceToJointTorques

```
void Robot::MapContactForceToJointTorques(int& leg_id,
                                          Eigen::VectorXd& q,
                                          Eigen::Vector3d& contact_force,
                                          Eigen::VectorXd& nle,
                                          VectorNm& motor_torques)
```

Maps the foot contact force to the leg leg_id joint torques.

sdlsInv

```
Eigen::MatrixXd Robot::sdlsInv(const Eigen::MatrixXd& jacobian)
```

This function computes the selective damping least square inverse matrix.

OpenLoopGaitGenerator Class

OpenLoopGaitGenerator

```
OpenLoopGaitGenerator::OpenLoopGaitGenerator(
    Robot* robot, Vector4d stance_duration, Vector4d duty_factor,
    Vector4d initial_leg_state, Vector4d initial_leg_phase,
    double contact_detection_phase_threshold)
```

Construct an OpenLoopGaitGenerator object using the given parameters.

Reset

```
void OpenLoopGaitGenerator::Reset(double current_time)
```

Reset the gait parameters.

Update

```
void OpenLoopGaitGenerator::Update(double current_time,
                                   Vector4d& contact_state,
                                   VectorNm& contact_force)
```

Update the gait parameters.

RaibertSwingLegController Class

GenEllipse

```
void RaibertSwingLegController::GenEllipse(
    double& swing_period, double& phase, Vector3d& start, double& mid,
    Vector3d& end, Vector3d& pos_des, Vector3d& pos_des_ddot)
```

Elliptic interpolation function used to generate the desired polygon curve.

GenSwingFootTrajectory

```
void RaibertSwingLegController::GenSwingFootTrajectory(
    double& swing_period, double& max_clearance, double& input_phase,
    Vector3d& start_pos, Vector3d& end_pos, Vector3d& desired_pos,
    Vector3d& desired_acc)
```

Generating the trajectory of the swing leg.

RaibertSwingLegController

```
RaibertSwingLegController::RaibertSwingLegController(
    Robot* robot, OpenLoopGaitGenerator* gaitGenerator,
    Vector2d desired_speed, double desired_twisting_speed,
    double desired_height, double max_clearance)
```

Construct a RaibertSwingLegController object using the given parameters.

Reset

```
void RaibertSwingLegController::Reset(double current_time,
                                      Eigen::VectorXd& q)
```

Reset the parameters of the RaibertSwingLegController.

Update

```
void RaibertSwingLegController::Update(double current_time,
                                       Eigen::VectorXd& q)
```

Update the parameters of the RaibertSwingLegController.

UpdateControlParameters

```
void RaibertSwingLegController::UpdateControlParameters(
    Vector2d& linSpeed, double& angSpeed)
```

Update the linear velocity and angular velocity of the controller.

GetAction

```
void RaibertSwingLegController::GetAction(
    double current_time, Eigen::VectorXd& q, Vector3d& com_position,
    Vector3d& body_orientation_rpy, Vector3d& com_velocity,
    Vector3d& body_angular_velocity)
```

Compute all motors' position commands using this controller.

TorqueStanceLegController Class

TorqueStanceLegController

```
TorqueStanceLegController::TorqueStanceLegController(
    Robot* robot, OpenLoopGaitGenerator* gaitGenerator,
    RaibertSwingLegController* swingLegController, double body_yaw_ref,
    Vector4d friction_coeffs, int planning_horizon_steps,
    double planning_timestep, VectorNx state_weights,
    VectorNu input_weights, double fz_max, double fz_min)
```

Construct a TorqueStanceLegController object using the given parameters.

UpdateControlParameters

```
void TorqueStanceLegController::UpdateControlParameters(
    Vector2d& linSpeed, double& angSpeed)
```

Update the linear velocity and angular velocity of the controller.

GetAction

```
void TorqueStanceLegController::GetAction(
    double current_time, Eigen::VectorXd& q, Eigen::VectorXd& q_dot,
    Vector3d& com_position, Vector3d& body_orientation_rpy,
    Vector3d& com_velocity, Vector3d& body_angular_velocity)
```

Compute all motors' torque commands using this controller.

ConvexMPCDense Class

ConvexMPCDense

```
ConvexMPCDense::ConvexMPCDense(Robot* robot, double body_previous_yaw,
                               Vector4d friction_coeffs,
                               int planning_horizon_steps,
                               double planning_timestep,
                               MatrixNxNhNxNh Q_qp, MatrixNuNhNuNh R_qp,
                               double fz_max, double fz_min)
```

Construct a ConvexMPCDense object using the given parameters.

CalculateAMatrix

```
void ConvexMPCDense::CalculateAMatrix(Vector3d& body_rpy,
                                      MatrixNxNx& A_mat)
```

Calculates the A matrix of the linearized state-space SRBD equation.

CalculateBMatrix

```
void ConvexMPCDense::CalculateBMatrix(double& inv_mass,
                                      Matrix3d& inv_inertia,
                                      Matrix3x4& foot_positions,
                                      MatrixNxNu& B_mat)
```

Calculates the B matrix of the linearized state-space SRBD equation.

CalculateDiscreteABMatrices

```
void ConvexMPCDense::CalculateDiscreteABMatrices(
    MatrixNxNx& A_mat, MatrixNxNu& B_mat, double& planning_timestep,
    MatrixNxNx& Ad_mat, MatrixNxNu& Bd_mat)
```

Calculates the A, B matrices of the discretized, linearized state-space SRBD equation.

CalculateQPMatrices

```
void ConvexMPCDense::CalculateQPMatrices(
    int& planning_horizon_steps, VectorNx& x0, VectorNxNh& state_ref,
    VectorNuNh& input_ref, MatrixNxNhNxNh& Q, MatrixNuNhNuNh& R,
    MatrixNxNx& Ad_mat, MatrixNxNu& Bd_mat, Sparse_Matrix& H_sparse,
    VectorNuNh& q_dense)
```

This function calculates the Hessian matrix and the gradient vector of the cost function of the condensed QP.

CalculateConstraintMatrix

```
void ConvexMPCDense::CalculateConstraintMatrix(
    int& planning_horizon_steps, MatrixNhNl& contact_states,
    Vector4d& friction_coeff, Sparse_Matrix& Ac_sparse)
```

This function calculates the linear constraints matrix of the condensed QP.

CalculateConstraintBounds

```
void ConvexMPCDense::CalculateConstraintBounds(
    int& planning_horizon_steps, MatrixNhNl& contact_state,
    double& fz_max, double& fz_min, double& friction_coeff,
    VectorNcNh& l, VectorNcNh& u)
```

This function calculates the bounds of the linear constraints of the condensed QP.

ComputeContactForces

```
void ConvexMPCDense::ComputeContactForces(
    double current_time, double& yaw, double& yaw_ref,
    VectorXd& foot_contact_states, Matrix3x4& foot_positions_body_frame,
    Vector3d& com_position, Vector3d& com_velocity,
    Vector3d& body_orientation_rpy, Vector3d& body_angular_velocity,
    Vector3d& desired_com_position, Vector3d& desired_com_velocity,
    Vector3d& desired_body_orientation_rpy,
    Vector3d& desired_body_angular_velocity)
```

Compute four leg's contact forces by solving the condensed QP.

LocomotionController Class

LocomotionController

```
LocomotionController::LocomotionController(
    Robot* robot, OpenLoopGaitGenerator* gaitGenerator,
    RaibertSwingLegController* swingLegController,
    TorqueStanceLegController* stanceLegController)
```

Construct a LocomotionController object using the other planners and controllers.

Reset

```
void LocomotionController::Reset(double current_time, Eigen::VectorXd& q)
```

Reset the parameters of the planners and controllers.

Update

```
void LocomotionController::Update(double current_time,
                                  Vector4d& contact_state,
                                  VectorNm& contact_force,
                                  Eigen::VectorXd& q, Vector2d& linSpeed,
                                  double& angSpeed)
```

Update the parameters of the planners and controllers.

GetAction

```
void LocomotionController::GetAction(
    double current_time, Eigen::VectorXd& q, Eigen::VectorXd& q_dot,
    VectorNm& qj_dot, Vector3d& com_position,
    Vector3d& body_orientation_rpy, Vector3d& com_velocity,
    Vector3d& body_angular_velocity)
```

Compute all motors' torque commands using the subcontrollers.

Other Functions

DARE

```
void DARE(MatrixNxNx& A, MatrixNxNu& B, MatrixNxNx& Q, MatrixNuNu& R,
          MatrixNxNx& Pn)
```

Solves the DARE.