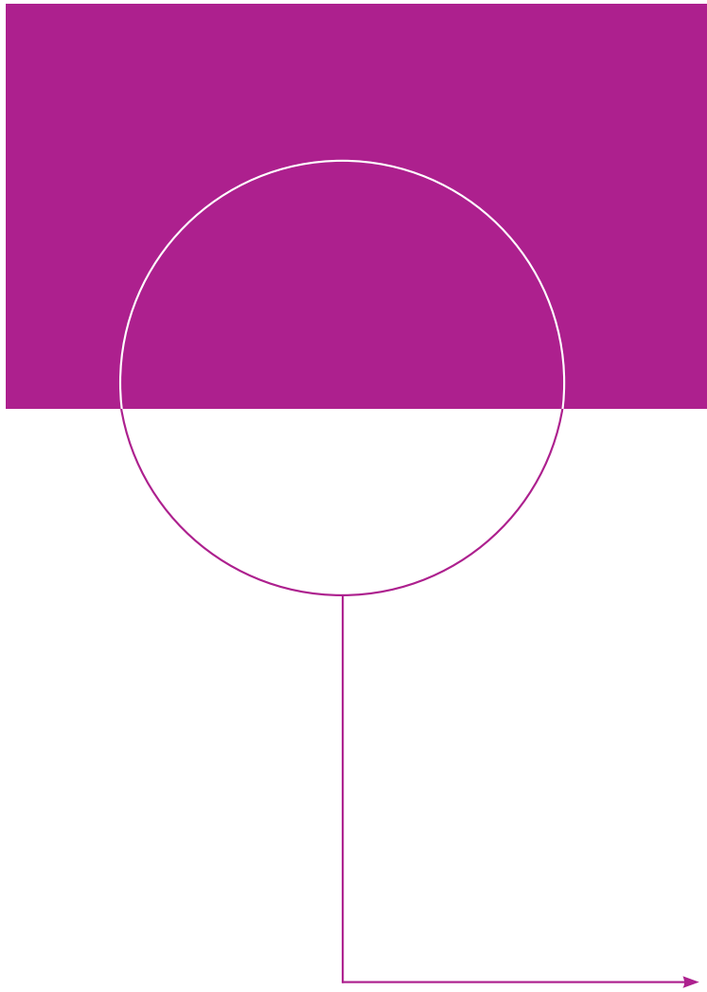


Master's thesis

2019

Lill Maria Gjerde Johannessen

**NTNU**
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Engineering Cybernetics

Lill Maria Gjerde Johannessen

# Robot Dynamics with URDF & CasADi

June 2019

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU

Norwegian University of
Science and Technology

# Robot Dynamics with URDF & CasADi

Lill Maria Gjerde Johannessen

Cybernetics & Robotics
Submission date: June 2019
Supervisor: Jan Tommy Gravdahl
Co-supervisor: Mathias Hauan Arbo

Norwegian University of Science and Technology
Department of Engineering Cybernetics

# Abstract

Fast, accurate evaluation of the dynamic parameters is a crucial ingredient for accurate control, estimation, and simulation of robots. As these are time-consuming to compute by hand, a software library for generating the rigid body dynamics symbolically can be of great use for robotics researchers. In this thesis, a library to efficiently compute and evaluate robot dynamics and its derivatives is proposed. Based on the Unified Robot Description Format (URDF), a full description of the robot's kinematic structure can be retrieved. CasADi provides a symbolic optimization framework growing in popularity among robotics researchers. By using the information provided by the URDF and the symbolic framework provided by CasADi, a library for obtaining symbolic expressions of a robot's dynamics has been implemented.

The library has gotten the name urdf2casadi and retrieval of the dynamic parameters is based on the implementation of three major rigid body dynamics algorithms. The recursive Newton-Euler algorithm is used to obtain the inverse dynamics, the Coriolis matrix, and the gravitational term. The articulated body algorithm is used to obtain the forward dynamics, and the composite rigid body algorithm is used to obtain the inertia matrix and forward dynamics. The implementation of the rigid body dynamics algorithms is inspired by the versions of the algorithms presented by Featherstone (2008). As these algorithms use spatial vector algebra for computational efficiency, a thorough presentation of spatial algebra is given. The modeling of the robot mechanism as a rigid multi-body system is also presented as well as a detailed description of the implementation of the algorithms.

To validate the numerical accuracy, the numerical evaluations of the solutions are compared against three other well-established rigid body dynamics libraries, namely RBDL, KDL, and PyBullet. We conduct a timing comparison between the libraries, and we show that the evaluation times of the symbolic expressions are at most one order of magnitude higher than the numerical evaluation times. Last, it is shown that the evaluation times of the dynamics derivatives remain of the same order as the evaluation times of the dynamics expressions, making them appropriate for use in robotics research. The work is summarized in an article accepted to the International Conference on Control, Mechatronics, and Automation (ICCMA).

# Preface

This report presents my master thesis as part of the Master of Science education in Cybernetics and Robotics at the Norwegian University of Science and Technology. The work has been done in association with the Department of Engineering Cybernetics at NTNU, and is associated with SFI Manufacturing, a center for research-based innovation focusing on high-value manufacturing in Norway with research areas on multi-material products and processes, robust and flexible automation, and innovative and sustainable organizations. This thesis is a part of the second research area.

I wish to express my sincere thanks to my supervisor Jan Tommy Gravdahl and co-supervisor Mathias Hauan Arbo for their support. Jan Tommy for being available and assistive whenever needed and Mathias for our regular meetings with discussions, constructive feedback, and guidance on the field.

The focus of this project has been to develop a software library that provides the dynamics of a robot symbolically. The work can be considered two-folded, where the first part of the work was my specialization project. The main focus of my specialization project was to study the field of rigid body dynamics algorithms and how to implement them for efficient and easy use. It was found that spatial algebra is advantageous when implementing rigid body dynamics. As I had little prior knowledge in this field, much time was spent understanding spatial algebra and how to implement the dynamics algorithms according to this use.

The second part of the work thus included completion of the implementation of urdf2casadi, and verifying the results in terms of numerical accuracy and timing efficiency, which is the work associated with my master thesis.

# Contributions

In this project, the author has made the following contributions:

- a presentation of the state-of-the-art of dynamics libraries for robotics,
- a discussion and investigation concerning optimal ways to implement algorithms in CasADi for obtaining robot dynamics,
- the development of urdf2casadi, providing symbolic expressions for a robot's dynamics,
- a numerical comparison between the solutions retrieved with urdf2casadi and other well-established numerical rigid body dynamics libraries to validate the numerical accuracy of the implementation,
- a timing comparison between the solutions retrieved with urdf2casadi and other well-established numerical rigid body dynamics libraries,
- a timing evaluation of the dynamics derivatives obtained using urdf2casadi and the CasADi framework,
- publishing an article about the work, which will, hopefully, be presented at ICCMA 2019, the 7th International, in TU Delft, Netherlands on November 6-8, 2019.

# Table of Contents

# List of Tables

# List of Figures

# CHAPTER 1

# Introduction

*This chapter is based on the Introduction chapter from the specialization project associated with this thesis. Modifications have been made according to changes during the latter part of the project.*

## 1.1 Motivation

Defining advanced feedback control techniques for robots requires the use of kinematics and dynamics. Oftentimes one requires both the forward or inverse mapping and its derivatives. These can be tedious to compute by hand, and many symbolic solver systems result in functions that have long evaluation time, making them impractical for use in feedback control.

The Robotic Operating System (ROS), presented by Quigley et al. (2009), is a software solution with a growing community among robotics programmers. Briefly explained, ROS is a middleware that implements a set of standards for communication between processes. Its usage ranges from controlling drones, unmanned boats and vehicles, to industrial robots. ROS has a Universal Robot Description Format (URDF), which is an XML file describing the robot's kinematics as a kinematic tree of frames with inertial, collision, and visual properties.

A library for using symbolic equations that is growing in popularity among robotics researchers is CasADi, presented by Andersson et al. (2018). It is a symbolic framework for algorithmic differentiation and numerical optimization. The framework provides the ability to rapidly prototype optimization algorithms and symbolic equations that are close to production ready.

This project has aimed to develop a library that exploits the information given from a URDF to generate symbolic equations in CasADi for robot kinematics and dynamics. Given that the URDF accurately describes the robot, it represents an

opportunity to automatically generate the kinematic and dynamic properties of the robot. The purpose of this library, urdf2casadi (u2c), is to use the URDF to generate the functions for forward kinematics, forward dynamics, inverse dynamics, the Coriolis and gravitational term, and the inertia matrix, symbolically in CasADi. By doing so, the life of many robotics researchers may be simplified: implementing optimal control problems will be easier to formulate, and their focus can be on solving the problem rather than extracting the kinematics and dynamics expressions.

## 1.2 Report Structure

**Theory**
The theory chapter provides the knowledge required to understand the implementation of the rigid body dynamics algorithms. This concerns spatial vector algebra for rigid bodies and how the robot's kinematic structure is modelled as a rigid multi-body system. An introduction to CasADi and why it is a desired framework when working with robot control is also provided.

**Implementation**
In the implementation chapter, the structural design of u2c is presented, and its modules are described. The implementation of rigid body dynamics algorithms is also given.

**Numerical Tests**
This chapter provides a brief overview of the numerical evaluation of the symbolic expressions returned by u2c, for four different robots. The numerical results are compared against the results of three other well-established dynamics libraries, and a discussion regarding the results is presented.

**Timing Tests**
This chapter provides a timing comparison, where the evaluation times of the dynamics expressions returned by u2c are compared against the numerical evaluation times of KDL, RBDL, and PyBullet. This is of importance, as the efficiency of the expressions is crucial in the context of closed-loop feedback control problems within robotics research.

**Derivatives Timing**
The dynamics derivatives can easily be obtained from the expressions returned by u2c. This chapter evaluates the efficiency of the derivatives expressions, which is also considered an important aspect of robotics research.

**Epilogue**
This chapter presents a discussion of the choices made in this project. A discussion regarding further work is also presented.

## 1.3 Mathematical Notation

Throughout this report, variables will be set in italic, constants, and functions in roman while vectors and matrices are set in bold italic.

Other details worth noticing are: $\mathbf{0}$ denotes a zero matrix or a zero vector and $\mathbf{1}$ denotes the identity matrix. Additionally, when writing $\boldsymbol{A}^{-T}$, the superscript denotes the transpose of the inverse, such that it is the equivalent of $(\boldsymbol{A}^{-1})^T$. Expressions on the form $\boldsymbol{a}\times$ denote a cross product operator. Last, if a symbol has a leading superscript (for instance ${}^{i}v$) then the superscript indicates a coordinate frame. Coordinate frames can also appear in subscripts when referring to transforms between coordinate frames.

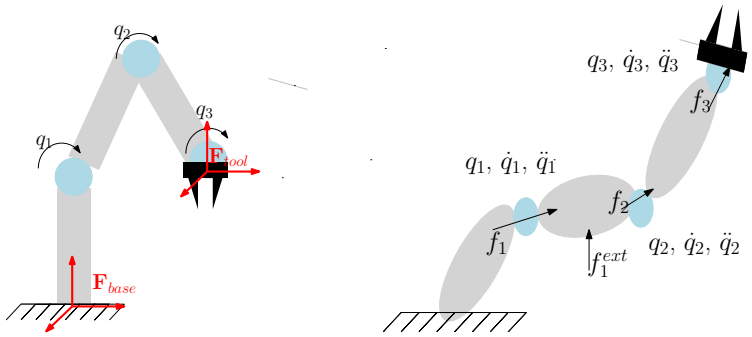
## 1.4 Robot Kinematics & Dynamics

Robot kinematics is the motion of bodies in a robotic mechanism without considering the forces and torques acting on the system. It relates the connectivity of the kinematic tree forming the robot mechanism to the position, velocity, and acceleration of each of the bodies[1] in the mechanism. Kinematics is an essential aspect of robot analysis and control, and concerns mainly two aspects:

1. **Forward kinematics** is the act of obtaining the position and orientation, i.e. the pose of the end-effector given the geometric structure and the joint positions. E.g. obtaining the position of *tool* relative to *base* given $q_1$, $q_2$, and $q_3$ in Figure 1.1a.

2. **Inverse kinematics** specifies the position of the end-effector and based on this information retrieves the associated joint positions. E.g. finding $q_1$, $q_2$, and $q_3$ given the pose of *tool* relative to *base* in Figure 1.1a.

Robot kinematics is broadly used among robotics researchers to study the motion of the mechanical robot system. To illustrate the use of robot kinematics, Song and Jung (2007) have used forward and inverse kinematics to analyze the motion of a humanoid robot and compared the motion against an industrial robot, while Dasari and Reddy (2012) have utilized forward and inverse kinematics to study the motion of a robot frog. In the latter case, the forward and inverse kinematics is used to study how any legged robot can take a perfect jump, which is practical as this is the most efficient way of traversal through uneven terrain.




**(a)** Kinematics. **(b)** Dynamics.

**Figure 1.1:** Illustration of robot kinematics & dynamics.

Robot dynamics gives the relationship between the forces acting on the robotic system and the accelerations it produces, e.g. the relationship between the gener-

[1] Also known as links.

alized joint forces[2], $f_1$, $f_2$, $f_3$ and the external force $f_1^{ext}$, and the generalized joint accelerations, $\ddot{q_1}$, $\ddot{q_2}$ and $\ddot{q_3}$, in Figure 1.1b.

The dynamics of a rigid body system are often described through the equation of motion:

$$\boldsymbol{\tau} = \boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})\boldsymbol{q} + \boldsymbol{G}(\boldsymbol{q}) - \sum_i \boldsymbol{J}_i(\boldsymbol{q})^T \boldsymbol{f}_i^{ext} \tag{1.1}$$

where $\boldsymbol{M}$ is the inertia matrix, $\boldsymbol{C}$ is the Coriolis matrix, and $\boldsymbol{G}$ is the gravitational term. $\boldsymbol{M}$, $\boldsymbol{C}$, and $\boldsymbol{G}$ are dynamic parameters describing some part of the robot's dynamics. For brevity, the dependent variables of $\boldsymbol{M}$, $\boldsymbol{C}$, and $\boldsymbol{G}$ are omitted henceforth, and the generalized joints are referred to as joints.

There are mainly five aspects of a robot's dynamics that are used in robotics research, but require major computations:

1. **Inverse dynamics** (ID) is the concern of finding the required joint forces from a specification of the joint positions, velocities, and accelerations. In Figure 1.1b, the ID problem would be the concern of finding the forces $f_1$, $f_2$ and $f_3$, from a specification of $q_i, \dot{q}_i, \ddot{q}_i$ where $i = 1, 2, 3$, and an external force $f_1^{ext}$.

2. **Forward Dynamics** (FD) is the concern of determining the produced joint accelerations given the forces applied to the robot system. In Figure 1.1b, this would be the concern of finding $\ddot{q_1}, \ddot{q_2}, \ddot{q_3}$ with specifications for $f_1, f_2, f_3$ and $f_1^{ext}$.

3. **Inertia Matrix** ($\boldsymbol{M}$) maps the joint accelerations to the joint forces. I.e. relating $f_1, f_2, f_3$ to $\ddot{q_1}, \ddot{q_2}, \ddot{q_3}$ in Figure 1.1b, where $\boldsymbol{M}_{ij}$ maps the force of joint $i$ to the acceleration of joint $j$, $i$ being the row number and $j$ being the column number of $\boldsymbol{M}$.

4. **Coriolis Matrix** ($\boldsymbol{C}$) encompasses the Coriolis forces acting on the joints, i.e. the part of $f_1, f_2, f_3$ that comes from the Coriolis effect in Figure 1.1b.

5. **Gravitational term** ($\boldsymbol{G}$) encompasses the gravitational forces acting on the joints, i.e. the part of $f_1, f_2, f_3$ that it a result of the gravitational field in Figure 1.1b.

All of these aspects have a variety of uses within robot analysis and control. Sansanayuth et al. (2012) presents teleoperative control for a PHANToM Omni haptic device using inverse dynamics, and Chandramouli and Manivannan (2018) presents minimization of torque and energy requirements for different postures of a bio-inspired reconfigurable robot using inverse dynamics, to mention some.

Forward dynamics is often used for simulation purposes and can also be used for control. Beirami and Macnab (2006) presents direct neural-adaptive control of robotic manipulators using forward dynamics.

[2]I.e. the constrained relative forces between the connected bodies.

The inertia matrix contains useful information regarding the relationship between the forces applied to the system and the acceleration it produces. Hence it has a variety of uses, ranging from neural-adaptive control of robotic manipulators using a supervisory inertia matrix, presented by Richert et al. (2000), to link mass optimization of serial robot manipulators, presented by Kucuk and Bingul (2006).

To summarize, robot kinematics and dynamics are widely used for several purposes. u2c aims to provide these aspects of a robot's kinematics[3] and dynamics as a symbolic representation of the joint state variables, for further use in robotics research. The *recursive Newton-Euler algorithm* (RNEA) is used to obtain the inverse dynamics, the Coriolis matrix, and the gravitational force. The *articulated body algorithm* (ABA) is used to obtain the forward dynamics, and the *composite rigid body algorithm* (CRBA) is used to obtain the inertia matrix. The algorithms are implemented using spatial algebra, as presented by Featherstone (2008).

Throughout this report, the joint positions, velocities, accelerations, and forces ($\boldsymbol{q}$, $\dot{\boldsymbol{q}}$, $\ddot{\boldsymbol{q}}$, $\tau$) are referred to as the joint state variables as they represent the current state of the joints in the system. They are further explained in section 2.3.3.

## 1.5 Earlier Work

When starting this project, a prototype of u2c had been developed by co-supervisor Mathias Hauan Arbo. The prototype contained functionality for deriving symbolic solutions for the forward kinematics.

The focus of this project has been to develop a library that provides symbolic solutions of the robot's dynamics, i.e. the aforementioned ID, FD, $\boldsymbol{M}$, $\boldsymbol{C}$, and $\boldsymbol{G}$. The functionality for deriving the forward kinematics is preserved, but as this work is derived independently of the original library implementation, the kinematics is not in focus.

[3] Symbolic formulations of the inverse kinematics is not prioritized at the time.

## 1.6 State-of-the-Art

Due to its importance in robotics research, several libraries for generating robot kinematics and dynamics exist. This section presents some of these libraries and discusses whether they are suited for optimal control within robotics research.

### 1.6.1 KDL

To automatically generate robot kinematics and dynamics from a URDF has already been achieved by the ROS[4] and OROCOS[5] community in the *Kinematics and Dynamics Library* (KDL), presented by Smits (2014).

OROCOS is the acronym of Open RObot COntrol Software, and the OROCOS project aims to develop an open-source framework for robot and machine control. The library contains special object types and functions so that one can take a kinematic chain and evaluate the forward kinematics, inverse kinematics, gravitational and Coriolis terms, and the inertia matrix. The routines are real-time safe, implemented in C++, and contain infrastructure for using the library with Python.

When using KDL to obtain the forward or inverse kinematics, or the dynamics, the results are numerical. There is no way to provide the kinematics or dynamics expressions symbolically, which restricts the user to the built-in functionality rather than being able to take as many partial derivatives as necessary for the users' controller formulation. Thus, in a scenario where the deriving of symbolic expressions for a robot's dynamics could be useful, for example, if one wishes to implement an MPC for robot control such as the time-optimal formulation of Verscheure et al. (2009), KDL may not provide the required functionality.

The above is in many ways the basis for this project: u2c attempts to implement a subset of the functionality of KDL for use with CasADi. Another difference of notice is that KDL do not provide forward and inverse dynamics, which u2c provides.

### 1.6.2 Rigid Body Dynamics Algorithms

Dr. Roy Featherstone is a researcher specializing in robot dynamics and related fields. He is the author of the book *Rigid Body Dynamics Algorithms* (Featherstone, 2008), whose is to present the most efficient algorithms for calculating rigid body dynamics. The dynamics algorithms presented in the book are RNEA for inverse dynamics, ABA for forward dynamics, and CRBA for obtaining the inertia matrix and forward dynamics. Featherstone's implementation of these algorithms stands out from the original versions as they are implemented using spatial vector algebra. Briefly explained, spatial algebra uses 6D vectors that contain the linear and angular characteristics of rigid body motion and forces. The fact that the linear and angular characteristics are given in one vector leads to easier implementation, more readable computer code, and more efficient algebraic computations.

[4]For further information on ROS, see `http://wiki.ros.org/`

[5]For further information on OROCOS, see `http://www.orocos.org/`

Therefore, Featherstone's versions of these algorithms are used as a basis for the development of u2c. Spatial vector algebra is further explained in section 2.2.

#### Spatial_v2

*Spatial_v2* is a set of functions, implemented by Featherstone (2012), that uses spatial vector arithmetic and the dynamics algorithms mentioned above. The library is implemented in Matlab and is based on the robot being described through a system model data structure. The data structure can describe a general fixed base kinematic tree either based on spatial 6D vector arithmetic or planar 3D vector arithmetic.

As opposed to u2c and KDL, Spatial_v2 cannot generate the dynamics based on a URDF and does not contain any functionality for deriving robot kinematics. As with KDL, Spatial_v2 provides numerical solutions to the dynamics problems. Additionally, the library is implemented in Matlab, which reduces its usefulness as Matlab requires a license. Despite this, Spatial_v2 has been an inspiration in this thesis for discovering ideas of how to implement the dynamics algorithms efficiently.

#### Rigid Body Dynamics Library

Felis (2016) presents the *Rigid Body Dynamics Library* (RBDL), which is a C++ library inspired by the pseudocode of Featherstone's algorithms in Rigid Body Dynamics Algorithms. The library contains the same essential rigid body algorithms: RNEA for inverse dynamics, ABA for forward dynamics and CRBA for computing the inertia matrix. Further, forward and inverse kinematics are provided, as well as functionality for the handling of external constraints such as contacts and collisions. Models of closed kinematic chains, such as Stewart platforms, are also provided.

RBDL is a very efficient library for robot kinematics and dynamics and even contains support for loading robot models from URDFs. Yet, as with the other libraries mentioned, it gives numerical solutions, and if one wishes to obtain the robot kinematics or dynamics expressions symbolically, it is of little use.

### 1.6.3 SymPyBotics

Sousa (2014) presents *SymPyBotics*, a symbolic framework for modeling and identification of robot dynamics. It is implemented in Python and makes use of SymPy for symbolic expressions and NumPy as the underlying numerical matrix library. SymPy is a Python library for symbolic mathematics, and SymPyBotics thus provides symbolic expressions for robot kinematics and dynamics.

To the author's knowledge, this is the only already existing library that provides symbolic solutions of robot kinematics and dynamics. However, there are a few things of important notice. Firstly, these symbols are given as SymPy data types which are naturally not compatible with CasADi. Secondly, an essential difference between SymPy and CasADi is that while SymPy uses symbolic differentiation, CasADi uses algorithmic differentiation. Briefly explained, symbolic

differentiation manipulates the total expression using rules of differentiation while algorithmic differentiation decomposes the expression into atomic operations and differentiates these. The result is that symbolic differentiation has a tendency to lead to inefficient computer code and faces the difficulty of converting an algorithm such as 4th order Runge-Kutte into a single expression. With algorithmic differentiation, accuracy is guaranteed, and the complexity does not get worse than the original function. Hence, when computing the Jacobian or Hessian, the symbolic framework provided by CasADi is significantly more efficient than SymPy. Also, CasADi exploits sparsity, which improves memory usage and computational efficiency. This is further discussed in section 2.1.2.

Based on the above one can conclude that SymPyBotics is not ideal in situations where u2c is ideal. Mainly because the data types of SymPyBotics are not compatible with CasADi, but also because CasADi provides advantages and functionality that SymPy is not able to provide. Thus, u2c has the potential to generate symbolic expressions that are more efficient than those of SymPyBotics. Also, SymPyBotics does not provide URDF loadings[6].

### 1.6.4 PyBullet

*PyBullet*, presented by Coumans and Bai (2016–2018), is an open-source collision detection and rigid body dynamics library and has support for forward and inverse kinematics and dynamics. The library supports URDF loading, and the solutions are given numerically.

PyBullet stands out from the other dynamics libraries as it uses maximal coordinate algorithms. In general, there are two classes of dynamics algorithms where one of the classes is the maximal coordinate algorithms, and the other class is the reduced coordinate algorithms. Maximal coordinate algorithms use joint constraint equations to restrict the relative motions of the bodies and model each body individually, in contrast to modeling the robot as a rigid multi-body system. The advantage of these methods is that they make the handling of certain closed kinematic chains simpler. The disadvantage is that the constraint equations, including joint constraints, are not necessarily fulfilled at all times, which may lead to separation of connected bodies.

In contrast to PyBullet, u2c uses reduced coordinate algorithms to obtain the dynamics. These algorithms assume that the connections of rigid multi-body systems can be described as a tree. The main advantage of these algorithms is that they operate on the actual degrees of freedom of the system that fulfill the joint constraints at all times. However, they are more complicated to implement than maximal coordinate algorithms and require special treatment in cases where the robot model is not given as a tree, for instance when working with closed kinematic chains such as Stewart platforms or arms in contact situations. It should be mentioned that one of the shortcomings of using a URDF to describe the robot is that URDF does not support closed kinematic chains. Therefore, the handling of closed kinematic chains in the reduced coordinate algorithms is not a concern at

[6]For that reason, it is not conducted a timing comparison between SymPyBotics and u2c.

the time for u2c.

### 1.6.5 Conclusion

**Table 1.1:** Kinematics & Dynamics Libraries.

| | Symbolic | URDF | Closed chains | Code compilation |
|---|---|---|---|---|
| KDL | | ✓ | | ✓ |
| spatial_v2 | | | | |
| RGBL | | ✓ | ✓ | ✓ |
| SymPyBotics | ✓ | | ✓ | ✓ |
| PyBullet | | ✓ | ✓ | ✓ |
| urdf2casadi | ✓ | ✓ | | ✓ |

In Table 1.1, a summary of the results obtained by investigating the state-of-the-art is given. By evaluating the table, it is clear that few libraries provide symbolic expressions of robot kinematics and dynamics, even though they are widely used among robotics researchers. To the author's knowledge, only one existing software library provides this. Although SymPyBotics provide symbolic expressions for kinematics and dynamics, it has other limitations such as not supporting URDF loading, the use of symbolic differentiation, and the fact that it does not exploit sparsity, making it less suitable for use in time-critical control.

Thus, u2c can provide robotics researchers with the opportunity to obtain and evaluate symbolic expressions for robot kinematics and dynamics fast and efficient, which can further be used for optimal control, trajectory optimization, and other tasks where the dynamics parameters and their derivatives are needed.

# CHAPTER 2

# Theory

*This chapter is based on the Theory chapter from the specialization project associated with this thesis. Modifications have been made according to changes during the latter part of the project.*

## 2.1 CasADi

As mentioned in the introduction, CasADi[1] is the go-to framework when working with numerical optimization and optimal control in particular. CasADi is developed by Joel Andersson, Joris Gillis, and Greg Horn at the Optimization in Engineering Center (OPTEC) of KU Leuven, under supervision of Moritz Diehl. It is an open-source framework available for Matlab, Python, and C++, and started as a tool for algorithmic differentiation (AD) that used the syntax of a Computer Algebra System (CAS), from which it has gotten its name.

In this section, an introduction to CasADi and its functionality will be presented. For a more in-depth description of CasADi, the reader is referred to Andersson et al. (2018).

### 2.1.1 Motivation for using CasADi

Derivatives play a central role when working with optimal control problems. There are several ways to compute these derivatives. One could find the derivatives by hand, which is a time-consuming and error prone activity. Therefore, differentiation performed by software is generally desirable. The two most common ways are symbolic and algorithmic differentiation. Most symbolic frameworks nowadays,

[1] Available at `https://web.casadi.org/`

such as the Symbolic Toolbox for Matlab[2] and SymPy, presented by Meurer et al. (2017), use symbolic differentiation. This is the most straightforward way of differentiation. The system knows the derivatives for most mathematical expressions and uses various rules, such as the product rule, to calculate the resulting derivative. In the end, the expression is simplified to obtain the resulting expression. Although this is easy to use, the resulting code is often long and computationally expensive to evaluate. Algorithmic differentiation, on the other hand, decomposes expressions into atomic operations[3] with known differentiation rules and differentiates these. Thus, CasADi can evaluate derivatives fast and accurately.

Further, the CasADi framework approaches optimal control problems in a way not seen in other optimization frameworks earlier. Rather than providing the programmer with a black box optimal control problem solver, CasADi provides the user with a suite of building blocks that can be used to implement general-purpose or specific-purpose solvers.

Another advantage of CasADi is that sparsity patterns are exploited by storing matrices using the Compressed Column Storage (CCS) format. In addition to reduced memory usage, CCS provides linear algebra operations to be performed efficiently. This is especially advantageous when computing rigid body dynamics, as it requires a large number of operations performed on sparse matrices.

Based on the above, generating symbolic expressions for robot kinematics and dynamics can be done efficiently when working with CasADi. These expressions can further be used to calculate Jacobians and Hessians, and solve optimal control problems using the optimization building blocks provided by CasADi.

It should also be mentioned that CasADi provides C generation of code. For u2c, this means that C code can be generated from the symbolic expressions. This has several advantages, the main two being faster execution times and that the user is not dependent on CasADi for using these expressions. The latter implies that the user can find the Jacobians and Hessians in CasADi for use in one's own libraries, or for porting the resulting controller formulations to embedded platforms.

### 2.1.2 Symbolic Framework

An AD implementation can be implemented to support both matrix-valued and scalar-valued atomic operations. However, when used on an expression only containing scalar atomic operations, this could lead to more overhead in terms of memory and computation. To avoid this, CasADi uses a structure where two expression graphs are defined to represent the symbolic expressions: one for scalar-valued atomic operations and another for sparse-matrix atomic operations.

Both expression graphs represent an atomic operation as a node in the graph. The graphs are topologically sorted so that they can be evaluated in two different virtual machines, one for each expression graph.

[2]For more information, see `https://www.mathworks.com/products/symbolic.html`
[3]I.e. expression that cannot be decomposed further.

### SX - Scalar Expression Graph

The SX data type is used to express matrices where elements are expressions made of unary and binary operations, thus representing the scalar expression type.

In the code snipped below it is shown how to declare SX symbolics in CasADi using Python[4]. Two symbolics, the $3 \times 1$ matrix q and the $3 \times 3$ matrix M are created and used to make the symbolic expression f = Mcos(q).

```
import casadi as cs
q = cs.SX.sym('q', 3)
M = cs.SX.sym('M', 3, 3)
f = cs.mtimes(M, cs.cos(q))
print (f)
```

Output:

```
@1=cos(q_0), @2=cos(q_1), @3=cos(q_2),
[(((M_0*@1)+(M_3*@2))+(M_6*@3)),
(((M_1*@1)+(M_4*@2))+(M_7*@3)),
(((M_2*@1)+(M_5*@2))+(M_8*@3))]
```

As can be observed, there are three shared subexpressions @1 = cos(q_0), @2 = cos(q_1), and @3 = cos(q_2) which defines the resulting $3 \times 1$ matrix f. The fact that one can have shared subexpressions in CasADi is very powerful as they are used to build up longer expressions. By having an expression graph for scalar-valued operations and at the same time exploiting reusable subexpressions, the result is that expressions with millions of operations can be represented and numerically evaluated with minimal overhead.

### MX - Matrix Expression Graph

The MX data type is the matrix expression type. When operating with MX symbols, each atomic operation in the expression graph is a matrix operation. Hence, expressions of the form A + B, where A and B are $n \times m$ matrices, results in one single addition operation, in contrast to when using SX. The MX usage is illustrated in the code snippet below where M and q are used to obtain f = Mq.

```
q = cs.MX.sym('q', 3)
M = cs.MX.sym('M', 3, 3)
f = cs.mtimes(M, q)
print (e)
```

Output:

```
mac(M,q,zeros(3x1))
```

mac represents a matrix multiply accumulate operation, which is an operation where two matrix operands are multiplied.

The MX expression graph only supports a small set of atomic operations. These are selected carefully so that derivatives calculated using AD can be expressed efficiently using this set of atomic operations.

[4]Only Python syntax is provided as this is what is used in u2c.

**Function Objects and Virtual Machines**

CasADi provides a class for function objects. Function objects represent a way of constructing symbolical expressions for numerical and symbolic evaluation, thus defining a way for calculating directional derivatives and Jacobians. The function objects are created by giving them a display name and inputs and outputs for the expression:

```
F = cs.Function('F', [q,M], [f])
```

The above example defines a function object with name `F`, two inputs `q` and `M`, and one output `f`. It is worth knowing that `MX` variables can be used as input to `SX` functions.

The construction of a function object works by sorting the expression graph into an algorithm that is evaluated when performing AD. After sorting of the operations of a function object, CasADi implements two virtual machines, one for the `SX` expression graphs and another for the `MX` expression graphs. Based on the operation, one of the virtual machines is assigned from a work vector. The work vector represents live variables, and makes it possible to reuse memory locations for variables going out of scope, which again decreases the overall memory usage. It also makes it possible for CasADi to identify in-place binary operations such as `x = x+y` $\Leftrightarrow$ `x+=y` and `x = x·y` $\Leftrightarrow$ `x·=y`.

Although the decrease in overhead by using such in-place operators is small, CasADi is able to combine this with other elementary operations such that `x += y*z` becomes in-place multiplication and addition. By doing so, Andersson et al. (2012) has shown that atomic operations are decreased by a factor of five, reducing the computational time sufficiently. A particular important consequence of this is that when expressing the Jacobian or Hessian, this feature will reduce the computational cost remarkably as these matrices consist of many atomic expressions such as addition and multiplication.

### 2.1.3 Jacobian Sparsity and Symmetry Exploiting

A strong feature of CasADi is that it exploits sparsity and symmetry patterns when calculating Jacobians. The algorithm for generating Jacobians in CasADi is very complex, but essentially consists of four steps:

1. Detect sparsity patterns.
2. Find derivatives needed to construct the complete Jacobian by using graph coloring techniques.
3. Calculate the directional derivatives numerically or symbolically.
4. Construct complete Jacobian.

Hessians are calculated by calculating the gradient and then obtaining the Jacobian of the gradient as explained above and exploiting symmetry by using star coloring algorithms. For a more detailed description of the algorithms, the reader

is referred to Andersson et al. (2018). Obtaining the Jacobian is done easily in CasADi using the following syntax:

```
J = cs.jacobian(mtimes(M,q),q)
```

One can also find the Jacobian-times-vector product using `cs.jtimes`. This is an efficient way of calculating the time derivatives of functions, and reduces the number of operations compared to first obtaining the full Jacobian with `cs.jacobian`, and then multiply the Jacobian with a vector. A more detailed description of `cs.jacobian` and `cs.jtimes` is given in Chapter 6.

### 2.1.4 C-code Generation

As mentioned above, one of the advantages of CasADi is that it support generation of C-code for a large number of function objects. C-code generation has several advantages such as:

- C-code generation of CasADi functions provides a way of running the code on a platform where CasADi is not installed: all that is needed is a C compiler.
- It is also a way to speed up the evaluation time. A rule of thumb is that the numerical evaluation of the generated C code can be 4 to 10 times faster than if the code was executed using CasADi's virtual machines, especially when using optimization flags with the compiler.
- C generation provides a way to debug the code and detect potentially suboptimal code. This is because the generated code reflects the evaluation that happens in the virtual machines. Thus, if an operation is slow, it is likely to show up when analyzing the generated code. If the code takes a long time to compile, this indicates that the code contains suboptimal solutions and should be broken into smaller nested functions.

C-code generation of function objects is done very easily in CasADi. For instance, if one was to C-code generate the function `F`, this can be done by the following syntax:

```
F.generate('gen.c')
```

This generates a C file `gen.c` that contains the function `F` with all its dependencies and helper functions. It is also possible to generate a C file containing several CasADi functions by using the `CodeGenerator` class:

```
C = CodeGenerator('gen.c')
C.add(F1)
C.add(F2)
C.add(F3)
C.generate()
```

Explicitly generating the code for function objects is not always necessary, and one can use just-in-time (jit) compilation of the function objects to achieve function objects that evaluate close to the speed of the C generated code, while being callable from Python. This is done easily when declaring the CasADi function. Using the aforementioned function `F`, jit compilation can be achieved by:

```
F = cs.Function('F', [q,M], [f],
{"jit": True, "jit_options":{"flags":"-Ofast"}})
```

where the -OFast compiler flag is used for maximum efficiency.

### 2.1.5 Conclusion

To conclude, CasADi is a framework that provides a transparent and efficient approach to algorithmic differentiation and optimal control, and is thus useful in the context of optimal control of robot systems. Its efficient approach is based on having two expression graph representations: one for minimal overhead, the `SX` graph, and another for maximum generality, the `MX` graph. Further, CasADi uses advanced algorithms to exploit sparsity and symmetry for the calculation of Jacobians and Hessians, making it an adventageous framework for obtaining derivatives of functions, while C-generation of code provides fast execution time and flexible usage.

## 2.2 Spatial Vector Algebra

Spatial vectors are 6D vectors that contain the linear and angular characteristics of rigid body motion and forces. The reason for using spatial vector algebra in the algorithms implemented in u2c is that they provide a compact notation to study the dynamics of a rigid multi-body system. A spatial vector can perform the work of two 3D vectors and thus replace two or more 3D equations. Hence, dynamics algorithms can be derived quickly and expressed in a compact form leading to efficient computer code.

This section aims to give the reader a proper understanding of spatial vector algebra and the advantages of using this notation related to rigid body dynamics. Spatial algebra is essential to understand the algorithms implemented in u2c, presented in Chapter 3. The section is based on the spatial algebra presented by Featherstone (2008) and Siciliano and Oussama (2008).

### 2.2.1 Preliminaries

First, some mathematical concepts and notations used to explain spatial vector algebra are presented.

**Vector Spaces:** In linear algebra, a vector is defined to be an element of a vector space. Four vector spaces that frequently occur in spatial algebra are:

$\mathtt{R}^n$ - coordinate vectors,

$\mathtt{E}^n$ - Euclidean vectors,

$\mathtt{M}^n$ - spatial motion vectors,

$\mathtt{F}^n$ - spatial force vectors.

The superscript indicates the dimension. When 3D vectors are used to describe rigid body dynamics, they are Euclidean vectors, and thus an inner product is defined on them. Spatial vectors are instead either motion vectors or force vectors. Spatial motion vectors describe rigid body velocity or acceleration, while spatial force vectors describe force and momentum.

**Hats and Underlines:** If spatial vectors and 3D vectors occur together, spatial vectors are represented with hats, $\hat{\boldsymbol{p}}$, to avoid name clashes with 3D vectors of the same name, $\boldsymbol{p}$. Further, if a distinction is required between the coordinate vectors and the abstract vectors they represent, coordinate vectors are underlined, $\underline{\boldsymbol{p}}$. These distinctions are only used to explain the concept of spatial algebra and are not used outside this chapter.

### 2.2.2 Spatial Vectors

To illustrate the advantage of spatial vectors explained above, one can evaluate the equation of motion for a rigid body. Usually, rigid body dynamics are expressed

using 3D vectors, although a rigid body in 3D has six degrees of freedom. As a result, one needs two vector equations to express the equation of motion for a rigid body:

$$\boldsymbol{f} = m\boldsymbol{a}_C \tag{2.1}$$

$$\boldsymbol{n}_C = \boldsymbol{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times \boldsymbol{I}\boldsymbol{\omega} \tag{2.2}$$

(2.1) expresses the relationship between the force applied to the body and the linear acceleration of its center of mass, while (2.2) expresses the relationship between the moment applied to the body and its angular acceleration.

In spatial vector notation 6D vectors are used instead of 3D vectors, where the 6D vectors combine linear and angular aspects of rigid body motion. Thus, the equation of motion reduces to one equation:

$$\hat{\boldsymbol{f}} = \hat{\boldsymbol{I}}\hat{\boldsymbol{a}} + \hat{\boldsymbol{v}} \times^* \hat{\boldsymbol{I}}\hat{\boldsymbol{v}} \tag{2.3}$$

In (2.3) $\hat{\boldsymbol{f}}$ is the spatial force acting on the body, $\hat{\boldsymbol{v}}$ and $\hat{\boldsymbol{a}}$ denotes its spatial velocity and acceleration, respectively, and $\hat{\boldsymbol{I}}$ is its spatial inertia tensor. $\times^*$ denotes a spatial force cross product. Further, spatial vector notation provides more simplifications than this where the overall effect is that spatial notation often cuts algebraic computation by at least a factor of 4 compared to standard 3D vector notation. This advantage of spatial vectors is especially useful when it comes to developing fast and efficient rigid body algorithms: the code becomes shorter and easier to read, write, and debug. In the following sections, the spatial concepts illustrated here are explained in further detail.

### 2.2.3 Plücker Coordinates

Plücker coordinates are the coordinates of choice for 6D vectors as they are easy to use as well as tending to be efficient for computer implementation. Thus, when implementing rigid body dynamics algorithms, it is useful to express motion and forces in Plücker coordinates. If one has a moving body where $\boldsymbol{\omega}$ is the angular velocity of the body, and $\boldsymbol{v}_O$ is the linear velocity around the fixed point $O$ the Plücker coordinates becomes the Cartesian coordinates of $\boldsymbol{\omega}$ and $\boldsymbol{v}_O$. This is further explained in section 2.2.4.

### 2.2.4 Spatial Velocity and Force

In spatial vector algebra, two vector spaces are used to describe the spatial vectors. Motion vectors are vectors describing the motion of a rigid body such as velocity vectors and acceleration vectors. These belong to the motion vector space $\mathtt{M}^6$. Force vectors describe force and momentum and belong to the vector space $\mathtt{F}^6$.

#### Basis Vectors

The notation currently used for rigid body dynamics use 3D vectors to express velocity. Here $\underline{\boldsymbol{v}} = (v_{Ox}, v_{Oy}, v_{Oz})^T$ is the 3D Cartesian velocity vector that represents $\boldsymbol{v}$ in the orthonomal basis $\{\boldsymbol{i},\boldsymbol{j},\boldsymbol{k}\}$, and the relationship between the rigid

body 3D velocity $\boldsymbol{v}$ and $\underline{\boldsymbol{v}}$ becomes:

$$\boldsymbol{v}_O = v_{Ox}\boldsymbol{i} + v_{Oy}\boldsymbol{j} + v_{Oz}\boldsymbol{k}$$

The same principal applies for spatial velocity, except that Plücker coordinates and a Plücker basis are used. A Plücker basis consists of 12 basis vectors, 6 basis vectors to represent motion and 6 basis vectors to represent force. Given a point $O$ and a Cartesian reference frame, the Plücker basis is given as three unit rotations about the lines $O_x$, $O_y$, and $O_z$, three unit translations in $x$, $y$, and $z$ directions, three unit couples about $x$, $y$, and $z$ directions, and three unit forces along the lines $O_x$, $O_y$, and $O_z$. The unit rotations are denoted $\{\boldsymbol{d}_{Ox}, \boldsymbol{d}_{Oy}, \boldsymbol{d}_{Oz}\}$, the unit translations are denoted $\{\boldsymbol{d}_x, \boldsymbol{d}_y, \boldsymbol{d}_z\}$, the unit couples are denoted $\{\boldsymbol{e}_x, \boldsymbol{e}_y, \boldsymbol{e}_z\}$, and the unit forces are denoted $\{\boldsymbol{e}_{Ox}, \boldsymbol{e}_{Oy}, \boldsymbol{e}_{Oz}\}$. These are illustrated in Figure 2.1.

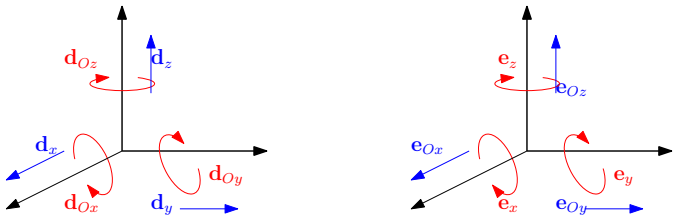



**Figure 2.1:** Basis vectors for Plücker coordinates.

Thus, the basises used for spatial motion and force vectors becomes, respectively:

$$\mathcal{D}_O = \{\boldsymbol{d}_{Ox}, \boldsymbol{d}_{Oy}, \boldsymbol{d}_{Oz}, \boldsymbol{d}_x, \boldsymbol{d}_y, \boldsymbol{d}_z\} \subset \mathtt{M}^6 \tag{2.4}$$

$$\mathcal{E}_O = \{\boldsymbol{e}_x, \boldsymbol{e}_y, \boldsymbol{e}_z, \boldsymbol{e}_{Ox}, \boldsymbol{e}_{Oy}, \boldsymbol{e}_{Oz}\} \subset \mathtt{F}^6 \tag{2.5}$$

### Velocity and Force

The velocity of a rigid body around a fixed point $O$ can be represented by two 3D vectors: $\boldsymbol{\omega}$ - the angular velocity and $\boldsymbol{v}_O$ - the linear velocity. The same velocity can be described using one single spatial motion vector, $\hat{\boldsymbol{v}} \in \mathtt{M}^6$, obtained from the 3D vectors. Given the Plücker basis $\mathcal{D}_O$ and a Cartesian coordinate frame $O_{xyz}$, it can be shown that the spatial velocity becomes:

$$\hat{\boldsymbol{v}} = \boldsymbol{d}_{Ox}\omega_x + \boldsymbol{d}_{Oy}\omega_y + \boldsymbol{d}_{Oz}\omega_z + \boldsymbol{d}_x v_{Ox} + \boldsymbol{d}_y v_{Oy} + \boldsymbol{d}_z v_{Oz} \tag{2.6}$$

In (2.6) $\omega_x, ..., v_{Oz}$ are the Cartesian coordinates of $\boldsymbol{\omega}$ and $\boldsymbol{v}_O$ and represent the Plücker coordinates of the spatial velocity vector $\hat{\boldsymbol{v}}$. The Plücker coordinate vector can thus be written:

$$\underline{\hat{\boldsymbol{v}}}_O = \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \\ v_{Ox} \\ v_{Oy} \\ v_{Oz} \end{pmatrix} = \begin{pmatrix} \underline{\boldsymbol{\omega}} \\ \underline{\boldsymbol{v}}_O \end{pmatrix} \tag{2.7}$$

The same principle applies to spatial force. Given a system of forces that act on a rigid body and a point $O$, the spatial force can be represented by the force $\boldsymbol{f}$ acting on a line passing through $O$ and the couple $\boldsymbol{n}_O$, which is the moment of the force about $O$. Hence, the force can be represented in the same way as the velocity using these two 3D vectors. Using the same coordinate frame as explained above, it can be shown that:

$$\hat{\boldsymbol{f}} = \boldsymbol{e}_x n_{Ox} + \boldsymbol{e}_y n_{Oy} + \boldsymbol{e}_z n_{Oz} + \boldsymbol{e}_{Ox} f_x + \boldsymbol{e}_y f_{Oy} + \boldsymbol{e}_z f_{Oz} \tag{2.8}$$

In (2.8) $n_{Ox}, ..., f_z$ are the Cartesian coordinates of $\boldsymbol{f}$ and $\boldsymbol{n}_O$ and represent the Plücker coordinates of the spatial force vector $\hat{\boldsymbol{f}}$. The Plücker coordinate vector thus becomes:

$$\underline{\hat{\boldsymbol{f}}}_O = \begin{pmatrix} n_{Ox} \\ n_{Oy} \\ n_{Oz} \\ f_x \\ f_y \\ f_z \end{pmatrix} = \begin{pmatrix} \underline{\boldsymbol{e}} \\ \underline{\boldsymbol{n}}_O \end{pmatrix} \tag{2.9}$$

### 2.2.5 Spatial Scalar Product

A scalar product is only defined on spatial vectors between a motion vector and a force vector. That is, given $\hat{\boldsymbol{m}} \in \mathtt{M}^6$ and $\hat{\boldsymbol{f}} \in \mathtt{F}^6$ the expressions $\hat{\boldsymbol{m}} \cdot \hat{\boldsymbol{f}}$ and $\hat{\boldsymbol{f}} \cdot \hat{\boldsymbol{m}}$ are defined and both express the power delivered by $\hat{\boldsymbol{f}}$ on the rigid body. $\hat{\boldsymbol{m}} \cdot \hat{\boldsymbol{m}}$ and $\hat{\boldsymbol{f}} \cdot \hat{\boldsymbol{f}}$ are not defined as there is no inner product on $\mathtt{M}^6$ or $\mathtt{F}^6$.

The scalar product creates a duality relationship which means that each vector space is the dual of the other. Given the vector space $\mathtt{M}^6$, it's dual, denoted $\mathtt{M}^{6*}$, is a vector space having the same dimension as $\mathtt{M}^6$ and having the property that a scalar product is defined between it and $\mathtt{M}^6$. Hence, given the above, $\mathtt{F}^6$ is the dual of $\mathtt{M}^6$ and $\mathtt{M}^6$ is the dual of $\mathtt{F}^6$.

The duality relationship between $\mathtt{M}^6$ and $\mathtt{F}^6$ results in a dual basis formed by a basis of $\mathtt{M}^6$, $\mathcal{D}_O = \{\boldsymbol{d}_1, \dots \boldsymbol{d}_6\} \subset \mathtt{M}^6$ as defined in (2.4), and a basis of $\mathtt{F}^6$, $\mathcal{E}_O = \{\boldsymbol{e}_1, \dots \boldsymbol{e}_6\} \subset \mathtt{F}^6$ as defined in (2.5). The dual basis on $\mathtt{M}^6$ and $\mathtt{F}^6$ satisfy the condition:

$$\boldsymbol{d}_i \cdot \boldsymbol{e}_j = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases} \tag{2.10}$$

A dual basis further defines a dual coordinate system. Hence, dual coordinate systems are used for representing spatial vectors. An important property of dual coordinates is:

$$\hat{\boldsymbol{m}} \cdot \hat{\boldsymbol{f}} = \underline{\hat{\boldsymbol{m}}}^T \underline{\hat{\boldsymbol{f}}} \tag{2.11}$$

In (2.11), $\underline{\hat{\boldsymbol{m}}}$ and $\underline{\hat{\boldsymbol{f}}}$ are the Plücker coordinate vectors that represent $\hat{\boldsymbol{m}} \in \mathtt{M}^6$ and $\hat{\boldsymbol{f}} \in \mathtt{F}^6$ and the individual elements in $\underline{\hat{\boldsymbol{m}}}$ and $\underline{\hat{\boldsymbol{f}}}$ are given by $\hat{m}_i = \boldsymbol{d}_i \hat{\boldsymbol{m}}$ and $\hat{f}_i = \boldsymbol{e}_i \hat{\boldsymbol{f}}$, respectively.

The use of dual coordinate systems results in the need of different transformation matrices: one to operate on the motion coordinate vector $\underline{\hat{\boldsymbol{m}}}$ and another to operate on the force coordinate vector $\underline{\hat{\boldsymbol{f}}}$. A motion transform is throughout the report denoted $\boldsymbol{X}$ and a force transform is denoted $\boldsymbol{X}^*$. Their relationship can be found from (2.11): $\underline{\hat{\boldsymbol{m}}}^T\underline{\hat{\boldsymbol{f}}} = (\boldsymbol{X}\underline{\hat{\boldsymbol{m}}})^T(\boldsymbol{X}^*\underline{\hat{\boldsymbol{f}}})$, which leads to the following relationship between the motion and force transforms:

$$\boldsymbol{X}^* = \boldsymbol{X}^{-T} \tag{2.12}$$

The duality relationship also results in different spatial cross product operators for motion and force vectors. This is further explained in section 2.2.7.

### 2.2.6 Spatial Coordinate Transforms

The Plücker coordinate transforms from $A$ to $B$ for motion and forces obey different transformation rules as a result of the spatial scalar product. The transform for a motion vector from $A$ to $B$ is denoted ${}^B\boldsymbol{X}_A$ and the transform for a force vector from $A$ to $B$ is denoted ${}^B\boldsymbol{X}_A^*$. The two transforms are related by the force transform being the inverse transpose of the motion transform. Their relationship is given by the equation ${}^B\boldsymbol{X}_A^* = {}^B\boldsymbol{X}_A^{-T}$ as explained earlier. This section focuses on explaining these transforms between Plücker coordinate systems.

Plücker coordinate systems denoted $A$ and $B$, are defined by the position and orientation of a Cartesian frame. Therefore each Plücker coordinate system also represents a Cartesian coordinate frame. The transformation from $A$ to $B$ depends only on the position and orientation of frame $B$ relative to frame $A$ and can hence be expressed as a product of the rotation transform and the translation transform.

#### Spatial Rotation Transform

Let $A$ and $B$ represent two Cartesian frames with common origin $O$. $\hat{\boldsymbol{m}}$ is any given spatial motion vector which can be expressed by two 3D vectors, $\boldsymbol{m}$ and $\boldsymbol{m}_O$, as explained in section 2.1.3. The rotational transform between frame $A$ and $B$ is then given by:

$${}^B\hat{\boldsymbol{m}} = \begin{pmatrix} {}^B\boldsymbol{m} \\ {}^B\boldsymbol{m}_O \end{pmatrix} = \begin{pmatrix} \boldsymbol{R}^A\boldsymbol{m} \\ \boldsymbol{R}^A\boldsymbol{m}_O \end{pmatrix} = \begin{pmatrix} {}^B\boldsymbol{R}_A & \boldsymbol{0} \\ \boldsymbol{0} & {}^B\boldsymbol{R}_A \end{pmatrix} {}^A\hat{\boldsymbol{m}} \tag{2.13}$$

In (2.13) ${}^A\hat{\boldsymbol{m}}$, ${}^A\boldsymbol{m}$, ${}^A\boldsymbol{m}_O$, ${}^B\hat{\boldsymbol{m}}$, ${}^B\boldsymbol{m}$ and ${}^B\boldsymbol{m}_O$ are the coordinate vectors[5] given in frame $A$ and $B$ respectively and $\boldsymbol{R}$ is the $3 \times 3$ rotation matrix that transforms coordinate frames from frame $A$ to $B$. The $6 \times 6$ rotational transform becomes:

$${}^B\boldsymbol{X}_A = \begin{pmatrix} {}^B\boldsymbol{R}_A & \boldsymbol{0} \\ \boldsymbol{0} & {}^B\boldsymbol{R}_A \end{pmatrix} \tag{2.14}$$

The upper left rotation matrix transforms the angular part of the motion vector and the lower left rotation matrix transforms the linear part of the motion vector.

[5] The coordinate vectors are not underlined here as no distinction is necessary.

By using the correlation between motion and force transforms given by (2.12) the following applies:

$$ {}^{B}\boldsymbol{X}_{A}^{*} = {}^{B}\boldsymbol{X}_{A}^{-T} = \begin{pmatrix} {}^{B}\boldsymbol{R}_{A} & \mathbf{0} \\ \mathbf{0} & {}^{B}\boldsymbol{R}_{A} \end{pmatrix} \tag{2.15} $$

An important detail of notice is that when a spatial transform is purely rotational ${}^{B}\boldsymbol{X}_{A}^{*} = {}^{B}\boldsymbol{X}_{A}$.

### Spatial Translation Transform

While the spatial rotation transforms accounts for the orientation of frame B, the translation transform accounts for the position of the frame. Let $O$ and $P$ be two points where one Cartesian frame is located at each point with the same orientation. Again, there are motion vectors that can be expressed as two 3D vectors: $\boldsymbol{m}$ and $\boldsymbol{m}_O$ at $O$, and $\boldsymbol{m}$ and $\boldsymbol{m}_P$ at $P$. By definition $\boldsymbol{m}_P = \boldsymbol{m}_O - \vec{OP} \times \boldsymbol{m}$. For both cases the Plücker coordinates are the Cartesian coordinates of both vectors and the following is obtained:

$$ \hat{\boldsymbol{m}}_P = \begin{pmatrix} \boldsymbol{m} \\ \boldsymbol{m}_P \end{pmatrix} = \begin{pmatrix} \boldsymbol{m} \\ \boldsymbol{m}_O - \boldsymbol{r} \times \boldsymbol{m} \end{pmatrix} = \begin{pmatrix} \mathbf{1} & \mathbf{0} \\ -\boldsymbol{r}\times & \mathbf{1} \end{pmatrix} \hat{\boldsymbol{m}}_{\boldsymbol{O}} \tag{2.16} $$

where $\boldsymbol{r} = \vec{OP}$. An expression on the form $\boldsymbol{r}\times$, where $\boldsymbol{r} = (x\ y\ z)^T$, is the skew-symmetric matrix that satisfies $\boldsymbol{r} \times \boldsymbol{a}$ for any 3D vector $\boldsymbol{a}$. It is defined by the equation:

$$ \begin{pmatrix} x \\ y \\ z \end{pmatrix} \times = \begin{pmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{pmatrix} \tag{2.17} $$

Hence, the spatial translation motion transform from a frame $A$ with origin $O$ to a frame $B$ with origin $P$, given the same orientation, becomes:

$$ {}^{B}\boldsymbol{X}_{A} = \begin{pmatrix} \mathbf{1} & \mathbf{0} \\ -\boldsymbol{r}\times & \mathbf{1} \end{pmatrix} \tag{2.18} $$

And the translation transform for force vectors is given by:

$$ {}^{B}\boldsymbol{X}_{A}^{*} = {}^{B}\boldsymbol{X}_{A}^{-T} = \begin{pmatrix} \mathbf{1} & -\boldsymbol{r}\times \\ \mathbf{0} & \mathbf{1} \end{pmatrix} \tag{2.19} $$

as $\boldsymbol{r}\times$ is skew symmetric.

### General Spatial Transform

The total spatial transform from frame $A$ to frame $B$ is given by the change of orientation and the change of position from frame $A$ to frame $B$. Hence, the general spatial transform becomes the product of the rotational and translational spatial transform. For motion vectors, this gives:

$$ {}^{B}\boldsymbol{X}_{A} = \begin{pmatrix} {}^{B}\boldsymbol{R}_{A} & \mathbf{0} \\ \mathbf{0} & {}^{B}\boldsymbol{R}_{A} \end{pmatrix} \begin{pmatrix} \mathbf{1} & \mathbf{0} \\ -\boldsymbol{r}\times & \mathbf{1} \end{pmatrix} = \begin{pmatrix} {}^{B}\boldsymbol{R}_{A} & \mathbf{0} \\ -{}^{B}\boldsymbol{R}_{A}\boldsymbol{r}\times & {}^{B}\boldsymbol{R}_{A} \end{pmatrix} \tag{2.20} $$

While for force vectors, the general transform becomes:

$$ {}^{B}\boldsymbol{X}_{A}^{*} = \begin{pmatrix} {}^{B}\boldsymbol{R}_{A} & \mathbf{0} \\ \mathbf{0} & {}^{B}\boldsymbol{R}_{A} \end{pmatrix} \begin{pmatrix} \mathbf{1} & -\boldsymbol{r}\times \\ \mathbf{0} & \mathbf{1} \end{pmatrix} = \begin{pmatrix} {}^{B}\boldsymbol{R}_{A} & -{}^{B}\boldsymbol{R}_{A}\boldsymbol{r}\times \\ \mathbf{0} & {}^{B}\boldsymbol{R}_{A} \end{pmatrix} \tag{2.21} $$

The inverse of these transforms, i.e. the transform from $B$ to $A$, can be easily obtained from (2.20) and (2.21):

$$ {}^{A}\boldsymbol{X}_{B} = \begin{pmatrix} \mathbf{1} & \mathbf{0} \\ \boldsymbol{r}\times & \mathbf{1} \end{pmatrix} \begin{pmatrix} {}^{B}\boldsymbol{R}_{A}^{T} & \mathbf{0} \\ \mathbf{0} & {}^{B}\boldsymbol{R}_{A}^{T} \end{pmatrix} = \begin{pmatrix} {}^{B}\boldsymbol{R}_{A}^{T} & \mathbf{0} \\ \boldsymbol{r} \times {}^{B}\boldsymbol{R}_{A}^{T} & {}^{B}\boldsymbol{R}_{A}^{T} \end{pmatrix} \tag{2.22} $$

$$ {}^{A}\boldsymbol{X}_{B}^{*} = \begin{pmatrix} \mathbf{1} & \boldsymbol{r}\times \\ \mathbf{0} & \mathbf{1} \end{pmatrix} \begin{pmatrix} {}^{B}\boldsymbol{R}_{A}^{T} & \mathbf{0} \\ \mathbf{0} & {}^{B}\boldsymbol{R}_{A}^{T} \end{pmatrix} = \begin{pmatrix} {}^{B}\boldsymbol{R}_{A}^{T} & \boldsymbol{r} \times {}^{B}\boldsymbol{R}_{A}^{T} \\ \mathbf{0} & {}^{B}\boldsymbol{R}_{A}^{T} \end{pmatrix} \tag{2.23} $$

To compare, the $4 \times 4$ homogeneous transformation matrix is given by:

$$ {}^{B}\boldsymbol{T}_{A} = \begin{pmatrix} {}^{B}\boldsymbol{R}_{A} & -{}^{B}\boldsymbol{R}_{A}\boldsymbol{r} \\ \mathbf{0} & 1 \end{pmatrix} \tag{2.24} $$

### 2.2.7 Spatial Cross Products

If $\boldsymbol{r} \in \mathtt{E}^3$ is a Euclidean vector with angular velocity $\boldsymbol{\omega}$ and otherwise not changing, then the derivative of $\boldsymbol{r}$ is given by $\dot{\boldsymbol{r}} = \boldsymbol{\omega}\times\boldsymbol{r}$. $\boldsymbol{\omega}\times$ is thus working as a differentiation operator as it maps $\boldsymbol{r}$ to $\dot{\boldsymbol{r}}$. This idea is extended to spatial vectors, except that here it is in need of two operators: one for the vectors of $\mathtt{M}^6$ and another for the vectors of $\mathtt{F}^6$. Hence, if $\hat{\boldsymbol{m}} \in \mathtt{M}^6$ and $\hat{\boldsymbol{f}} \in \mathtt{F}^6$ are two spatial vectors moving with the spatial velocity $\hat{\boldsymbol{v}} \in \mathtt{M}^6$ and otherwise not changing, one can define two cross operations that must satisfy:

$$ \dot{\hat{\boldsymbol{m}}} = \hat{\boldsymbol{v}} \times \hat{\boldsymbol{m}} \tag{2.25} $$

$$ \dot{\hat{\boldsymbol{f}}} = \hat{\boldsymbol{v}} \times^{*} \hat{\boldsymbol{f}} \tag{2.26} $$

$\times^*$ can be viewed as the dual of $\times$, and their relationship is similar to the relationship between $\boldsymbol{X}$ and $\boldsymbol{X}^*$. When using the Plücker basises $\mathtt{M}^6$ and $\mathtt{F}^6$ and their duality relationship explained in section 2.2.5, the $6 \times 6$ matrices representing $\hat{\boldsymbol{v}}\times$ and $\hat{\boldsymbol{v}}\times^*$ in Plücker coordinates can be deduced. The results are:

$$ \hat{\boldsymbol{v}}_{O}\times = \begin{pmatrix} \boldsymbol{\omega} \\ \boldsymbol{v}_{O} \end{pmatrix} \times = \begin{pmatrix} \boldsymbol{\omega}\times & \mathbf{0} \\ \boldsymbol{v}_{O}\times & \boldsymbol{\omega}\times \end{pmatrix} \tag{2.27} $$

$$ \hat{\boldsymbol{v}}_{O}\times^{*} = \begin{pmatrix} \boldsymbol{\omega} \\ \boldsymbol{v}_{O} \end{pmatrix} \times^{*} = \begin{pmatrix} \boldsymbol{\omega}\times & \boldsymbol{v}_{O}\times \\ \mathbf{0} & \boldsymbol{\omega}\times \end{pmatrix} = -(\hat{\boldsymbol{v}}_{O}\times)^{T} \tag{2.28} $$

The spatial cross products of two motion vectors becomes:

$$ \begin{pmatrix} \boldsymbol{\omega} \\ \boldsymbol{v}_{O} \end{pmatrix} \times \begin{pmatrix} \boldsymbol{m} \\ \boldsymbol{m}_{O} \end{pmatrix} = \begin{pmatrix} \boldsymbol{\omega} \times \boldsymbol{m} \\ \boldsymbol{\omega} \times \boldsymbol{m}_{O} + \boldsymbol{v}_{O} \times \boldsymbol{m} \end{pmatrix} \tag{2.29} $$

And the spatial cross product for two force vectors is given by:

$$ \begin{pmatrix} \boldsymbol{\omega} \\ \boldsymbol{v}_{O} \end{pmatrix} \times^{*} \begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{f}_{O} \end{pmatrix} = \begin{pmatrix} \boldsymbol{\omega} \times \boldsymbol{f}_{O} + \boldsymbol{v}_{O} \times \boldsymbol{f} \\ \boldsymbol{\omega} \times \boldsymbol{f} \end{pmatrix} \tag{2.30} $$

### 2.2.8 Spatial Acceleration

The spatial acceleration of a rigid body is defined as the rate of change of spatial velocity. Although this seems obvious, the spatial acceleration differs from the classical rigid body acceleration, which is here referred to as *classical acceleration* and will be denoted $\boldsymbol{a}'$. Spatial acceleration is given as the time derivative of spatial velocity:

$$\hat{\boldsymbol{a}}_O = \frac{d}{dt}\begin{pmatrix}\boldsymbol{\omega}\\ \boldsymbol{v}_O\end{pmatrix} = \begin{pmatrix}\dot{\boldsymbol{\omega}}\\ \dot{\boldsymbol{v}_O}\end{pmatrix} \tag{2.31}$$

The classical acceleration is defined as the following 6D vector:

$$\hat{\boldsymbol{a}}'_O = \begin{pmatrix}\dot{\boldsymbol{\omega}}\\ \dot{\boldsymbol{v}}'_O\end{pmatrix} \tag{2.32}$$

Given $\boldsymbol{r}$ as the position vector related to the position of the fixed point $O$ relative to another fixed point, the following relations are given:

$$\begin{aligned}\boldsymbol{v}_O &= \dot{\boldsymbol{r}}\\ \dot{\boldsymbol{v}}'_O &= \ddot{\boldsymbol{r}}\\ \dot{\boldsymbol{v}}_O &= \ddot{\boldsymbol{r}} - \boldsymbol{\omega}\times\boldsymbol{v}_O\end{aligned} \tag{2.33}$$

The difference between the spatial acceleration, given by (2.31), and the classical acceleration, given by (2.32), lies in spatial acceleration taking the derivative of the linear velocity $\boldsymbol{v}_O$ where $O$ is fixed in space, while for classical acceleration the derivative of the linear acceleration is derived as if $O$ is fixed on the rigid body. The two accelerations are related by the equation:

$$\hat{\boldsymbol{a}}'_O = \hat{\boldsymbol{a}}_O + \begin{pmatrix}\boldsymbol{0}\\ \boldsymbol{\omega}\times\boldsymbol{v}_O\end{pmatrix} \tag{2.34}$$

The great advantage of spatial acceleration is that it obeys the same combination of coordinate transformation rules as velocity. To demonstrate, if two bodies $\boldsymbol{B}_1$ and $\boldsymbol{B}_2$ have the spatial velocities $\hat{\boldsymbol{v}}_1$ and $\hat{\boldsymbol{v}}_2$, respectively, then it is already known that the velocity of $\boldsymbol{B}_2$ relative to $\boldsymbol{B}_1$ is given by the relative velocity:

$$\hat{\boldsymbol{v}}_{rel} = \hat{\boldsymbol{v}}_2 - \hat{\boldsymbol{v}}_1$$

The relationship between spatial acceleration is then obtained by differentiating the velocity formula:

$$\frac{d}{dt}\hat{\boldsymbol{v}}_{rel} = \frac{d}{dt}\hat{\boldsymbol{v}}_2 - \frac{d}{dt}\hat{\boldsymbol{v}}_1 \Rightarrow \hat{\boldsymbol{a}}_{rel} = \hat{\boldsymbol{a}}_2 - \hat{\boldsymbol{a}}_1 \tag{2.35}$$

As one can see from (2.35), spatial accelerations are composed simply by addition, just as velocities. Hence, spatial accelerations are easier to use than classical acceleration, where there are Coriolis and centrifugal terms to take into consideration. This is an important advantage of spatial vector algebra as it simplifies the implementation of rigid body algorithms relative to using traditional 3D vectors.

### 2.2.9 Spatial Momentum

Suppose there is a moving rigid body with mass $m$, center of mass $C$ and a rotational inertia $\bar{\boldsymbol{I}}_C$, that is moving with a spatial velocity $\hat{\boldsymbol{v}} = (\boldsymbol{\omega}, \boldsymbol{v}_C)^T$. Then the momentum of the body is described as two 3D vectors $\boldsymbol{h} = m\boldsymbol{v}_C$ and $\boldsymbol{h}_C = \bar{\boldsymbol{I}}_C\boldsymbol{\omega}$. $\boldsymbol{h}$ is the linear momentum of the body and is a line vector, such as linear force, where its line of action passes through the body's centre of mass. $\boldsymbol{h}_C$ is the body's angular momentum. The total moment of momentum about a given point $O$ is the sum of these two:

$$\boldsymbol{h}_O = \boldsymbol{h}_C + \vec{OC} \times \boldsymbol{h} \tag{2.36}$$

These definitions can be assembled into the definitions of spatial momentum. $\hat{\boldsymbol{h}}$ denotes the spatial momentum and $\hat{\boldsymbol{h}}_C$ and $\hat{\boldsymbol{h}}_O$ are the coordinate vectors that represent the Plücker coordinate system at $C$ and $O$, respectively. They are given by:

$$\hat{\boldsymbol{h}}_C = \begin{pmatrix} \boldsymbol{h}_C \\ \boldsymbol{h} \end{pmatrix} = \begin{pmatrix} \bar{\boldsymbol{I}}_C\boldsymbol{\omega} \\ m\boldsymbol{v}_C \end{pmatrix} \tag{2.37}$$

$$\hat{\boldsymbol{h}}_O = \begin{pmatrix} \boldsymbol{h}_O \\ \boldsymbol{h} \end{pmatrix} = \begin{pmatrix} \bar{\boldsymbol{I}}_C\boldsymbol{\omega} + \vec{OC} \times m\boldsymbol{v}_C \\ m\boldsymbol{v}_C \end{pmatrix} = \begin{pmatrix} \mathbf{1} & \vec{OC}\times \\ \mathbf{0} & \mathbf{1} \end{pmatrix} \hat{\boldsymbol{h}}_C \tag{2.38}$$

Spatial momentum obey the same properties as spatial force as they are elements of $\mathtt{F}^6$. Thus, they transform accordingly: ${}^B\hat{\boldsymbol{h}} = {}^B\boldsymbol{X}^*_A\,{}^A\hat{\boldsymbol{h}}$.

### 2.2.10 Spatial Inertia

The spatial inertia of a rigid body defines the relationship between its spatial velocity and momentum, and is therefore the mapping from the motion basis $\mathtt{M}^6$ to the force basis $\mathtt{F}^6$:

$$\hat{\boldsymbol{h}} = \hat{\boldsymbol{I}}\hat{\boldsymbol{v}} \tag{2.39}$$

As can be seen from (2.39), the spatial momentum of a rigid body is the product of the spatial inertia and velocity. By combining (2.37) and (2.39) the following is obtained:

$$\hat{\boldsymbol{h}}_C = \hat{\boldsymbol{I}}_C\hat{\boldsymbol{v}}_C = \begin{pmatrix} \bar{\boldsymbol{I}}_C & \mathbf{0} \\ \mathbf{0} & m\mathbf{1} \end{pmatrix} \hat{\boldsymbol{v}}_C \tag{2.40}$$

where $\mathbf{1}$ denotes the $3 \times 3$ identity matrix. One can observe that the spatial $6 \times 6$ inertia tensor at the body's center of mass becomes:

$$\hat{\boldsymbol{I}}_C = \begin{pmatrix} \bar{\boldsymbol{I}}_C & \mathbf{0} \\ \mathbf{0} & m\mathbf{1} \end{pmatrix} \tag{2.41}$$

(2.41) is the general formula for spatial inertia expressed at its center of mass and takes this form whenever the origin coincides with the center of mass. To express the spatial inertia at an origin $O$ that does not coincide with the center of

mass, an expression for $\hat{\boldsymbol{I}}_O$ is extracted from (2.38) and (2.39). Using $\boldsymbol{c} = \vec{OC}$ the following is obtained:

$$\begin{aligned}\hat{\boldsymbol{h}}_O &= \begin{pmatrix}\mathbf{1} & \boldsymbol{c}\times \\ \mathbf{0} & \mathbf{1}\end{pmatrix} \hat{\boldsymbol{I}}_C \hat{\boldsymbol{v}}_C \\ &= \begin{pmatrix}\mathbf{1} & \boldsymbol{c}\times \\ \mathbf{0} & \mathbf{1}\end{pmatrix} \hat{\boldsymbol{I}}_C \begin{pmatrix}\mathbf{1} & \mathbf{0} \\ \boldsymbol{c}\times^T & \mathbf{1}\end{pmatrix} \hat{\boldsymbol{I}}_C \hat{\boldsymbol{v}}_O\end{aligned} \tag{2.42}$$

From (2.42) one can extract the spatial inertia tensor at an origin $O$ using the results obtained in (2.41):

$$\hat{\boldsymbol{I}}_O = \begin{pmatrix}\bar{\boldsymbol{I}}_C + m\boldsymbol{c} \times \boldsymbol{c}\times^T & m\boldsymbol{c}\times \\ m\boldsymbol{c}\times^T & m\mathbf{1}\end{pmatrix} \tag{2.43}$$

(2.43) is the general form of the spatial inertia in Plücker coordinates. The upper left of the inertia matrix ($\bar{\boldsymbol{I}}_C + m\boldsymbol{c} \times \boldsymbol{c}\times^T$) is the rotational inertia of the body around $O$. Spatial inertia matrices are symmetric and positive definite.

The transformation of spatial inertia between two different reference frames $A$ and $B$ is given by:

$$\hat{\boldsymbol{I}}_B = {}^B\boldsymbol{X}_A^* \hat{\boldsymbol{I}}_A {}^A\boldsymbol{X}_B \tag{2.44}$$

One can observe from (2.44) that both a force transform and a motion transform are needed to transform the spatial inertia from one frame to another. This is a result of spatial inertia being the mapping between $\mathtt{M}^6$ to $\mathtt{F}^6$, and thus, one transform for each basis is needed.

An important advantage of spatial inertia is that if one is to find the total inertia of a composite rigid body, the total inertia becomes the sum of all rigid body inertias. To illustrate, if two bodies, having inertia $\boldsymbol{I}_1$ and $\boldsymbol{I}_2$, are rigidly connected and form a composite body, then the inertia of the total composite body becomes:

$$\hat{\boldsymbol{I}}_{tot} = \hat{\boldsymbol{I}}_1 + \hat{\boldsymbol{I}}_2 \tag{2.45}$$

If one were using the traditional 3D vector approach, (2.45) would take the place of three equations: one to compute the composite mass, another to compute the center of mass, and a last equation to compute the composite rotational inertia.

## 2.3 Modeling Rigid Multi-Body Systems

Implementing rigid body dynamics algorithms requires to model the robot as a rigid multi-body system, where the links of the robot represent the bodies in the multi-body system.

A rigid multi-body system consists of several bodies and joints that connect them. A joint is used to connect two bodies, and the joints thus affect the relative motion of the bodies. In Table 2.1, the variables used to describe a rigid multi-body system are presented, where some of them are shown in context to a rigid multi-body system in Figure 2.2. In the following sections the formulation of a rigid multi-body system follows.

**Table 2.1:** Variables of a rigid multi-body system.

| Symbol | Representation |
|---|---|
| $\lambda_i$ | Index of the parent body for joint $i$ |
| | where joint $i$ connects body $i$ with body $\lambda_i$ |
| $\kappa(i)$ | The set of joints that influence (i.e. support) body $i$ |
| $\mu(i)$ | The set of children of joint $i$ |
| $\nu(i)$ | The subtree that starts at joint $i$ |
| $\boldsymbol{S}_i$ | Joint space matrix for joint $i$ |
| $\boldsymbol{v}_i$ | Spatial velocity of body $i$ |
| $\boldsymbol{a}_i$ | Spatial acceleration of body $i$ |
| $\boldsymbol{f}_i$ | Spatial force acting on body $i$ from |
| | parent body $\lambda_i$ through the connecting joint $i$ |
| $\boldsymbol{I}$ | Spatial inertia of body $i$ |
| $\boldsymbol{I}^c$ | Composite body inertia of body $i$ |
| $\boldsymbol{I}^A$ | Articulated body inertia of body $i$ |
| $\boldsymbol{p}^A$ | Articulated bias force of body $i$ |

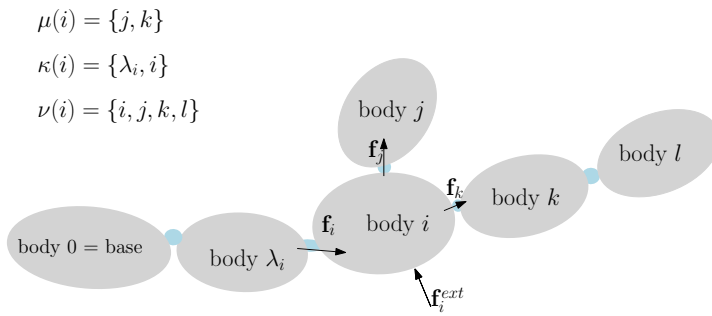



**Figure 2.2:** Branched kinematic tree modeled as a rigid multi-body system.

### 2.3.1 Transforms and Coordinate Systems

Obtaining the dynamics of a rigid multi-body system involves knowing the relative position and orientation of the bodies. For this purpose, several coordinate frames and their transforms are used. These are explained in this section, and a summary is given in Table 2.2 together with an illustrative figure of the frames and transforms in Figure 2.20.

The first frame to consider is the *global reference frame*. This coordinate system is fixed and therefore, does not move. The root[6] body $\boldsymbol{B}_0$ is attached to this reference frame. Further, each body in the system has a frame attached to it, which is known as the *body (local) reference frame*. In the case of body $\boldsymbol{B}_0$, the global and the local reference frame are the same.

The bodies are attached through joints, where one joint connects two bodies as shown in Figure 2.2. Since the joints are not usually located at the origin of the parent body, the *joint location frame* is needed. This frame describes the position and orientation relative to the coordinate frame of the parent body. The transformation from the parent body to the joint location frame is denoted $\boldsymbol{X_T}$. This frame is fixed and specified in the model by the parent's body reference frame.

A joint that is moving, e.g. a prismatic joint that is translating or a revolute joint that is rotating, requires an additional frame to represent this change. This frame is called the *joint motion frame* and changes with the joint as it moves. It is denoted $\boldsymbol{X_J}$ and coincides with the child body reference frame.

**Table 2.2:** Coordinate frames and transforms of a rigid multi-body system.

| Name | Representation |
|---:|:---:|
| *Global reference frame* | Fixed reference frame on root body |
| *Body (local) reference frame* | Frame attached on body |
| *Joint location frame* | Frame attached on joint relative to parent body |
| *Joint motion frame* | Frame attached on joint such that it follows its movement |
| $\boldsymbol{X_{T}}_i$ | Transformation from body $\lambda_i$ reference frame to joint $i$ location frame |
| $\boldsymbol{X_{J}}_i$ | Transformation from joint location frame to joint motion frame of joint $i$ |
| ${}^i\boldsymbol{X}_{\lambda_i}$ | Transformation from parent body reference frame to body $i$ reference frame |
| ${}^i\boldsymbol{X}_0$ | Transformation from global reference frame to local frame of body $i$ |

[6] Also known as the base.

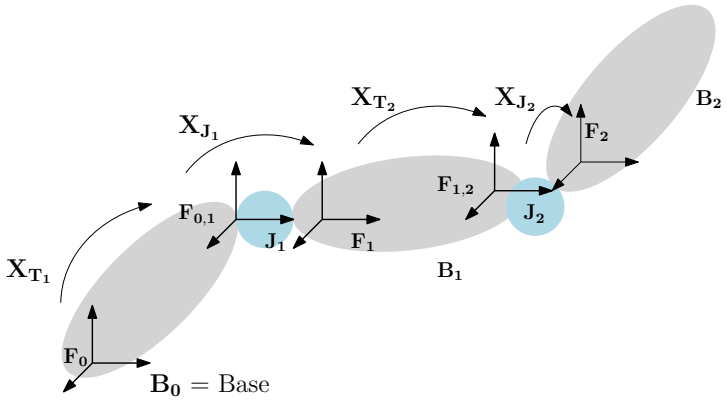



**Figure 2.3:** Coordinate transforms of a rigid multi-body system.

In Figure 2.3 $\boldsymbol{F}_0$ is the global reference frame, which is also the reference frame of body 0. $\boldsymbol{F}_{0,1}$ is the fixed joint location frame of joint 1 and the transformation from $\boldsymbol{F}_0$ to $\boldsymbol{F}_{0,1}$ is given by $\boldsymbol{X}_{\boldsymbol{T}1}$. $\boldsymbol{F}_1$ is the joint motion frame, also being the body reference frame of body 1, which is dependent on the motion of joint 1. The transformation from the joint reference frame to the joint motion frame is given by $\boldsymbol{X}_{\boldsymbol{J}1}$. The transformation from the the parent body 0 reference frame to child body 1 reference frame thus becomes: ${}^1\boldsymbol{X}_0 = \boldsymbol{X}_{\boldsymbol{T}1}\boldsymbol{X}_{\boldsymbol{J}1}$. The general transformation from a parent body to a child body is given by:

$$ {}^i\boldsymbol{X}_{\lambda_i} = \boldsymbol{X}_{\boldsymbol{T}i}\boldsymbol{X}_{\boldsymbol{J}i} \tag{2.46} $$

And the transformation from the global fixed reference frame to a body $i$ is given recursively as:

$$ {}^i\boldsymbol{X}_0 = {}^i\boldsymbol{X}_{\lambda_i}\,{}^{\lambda_i}\boldsymbol{X}_0 \tag{2.47} $$

where the starting point is given by $i = 1 \Rightarrow {}^i\boldsymbol{X}_{\lambda_i} = {}^i\boldsymbol{X}_0$.

### 2.3.2 Bodies

In a rigid multi-body system each body is described by its mass $m$, the location of the center of mass $\boldsymbol{c} \in \mathbb{R}^3$ and the rotational inertia $\boldsymbol{I} \in \mathbb{R}^{3\times3}$. Based on this information, the spatial inertia tensor $\hat{\boldsymbol{I}} \in \mathbb{R}^{6\times6}$ can be composed for each body using (2.43).

### 2.3.3 Joints

A joint is used to connect two bodies and hence restricts their relative motion. A joint can allow between 0 and 6 degrees of freedom (DOF), where a joint that allows 0 DOF is fixed, i.e. the two bodies are rigidly connected to each other, and a joint that allows 6 DOF does not constrain the motion at all.

There is a specific subset of state variables associated with each joint, and these variables represent the joint state. For a joint $i$ with $n$ DOF the values of

the variable $\boldsymbol{q}_i \in \mathbb{R}^n$ are the associated joint positions of joint $i$. Similarly $\dot{\boldsymbol{q}}_i$, $\ddot{\boldsymbol{q}}_i$, $\boldsymbol{\tau}_i \in \mathbb{R}^n$ represent the associated joint velocities, accelerations, and forces for joint $i$, respectively. $\boldsymbol{\tau}_i$ is more precisely the force that is transmitted through joint $i$.

If the joint positions for a joint $i$ are nonzero, it means that the joint has moved from its initial position. The transformation from its initial position is then represented by the joint motion transform $\boldsymbol{X}_{\boldsymbol{J}i}$. This is a spatial transformation on the form given by (2.20) where the entries $\boldsymbol{R}$ and $\boldsymbol{r}$ are joint type dependent.

### Joint Models

This thesis is primarily intended for use with robotic manipulators, where prismatic and revolute joints are the most prevalent. Thus only the descriptions of these joint models are presented and available in u2c.

As mentioned previously, the joint defines the relative motion between two bodies, and is joint type dependent. The relative motion can be described by the joint space matrix $\boldsymbol{S}$, defining the space of the joint motion. The matrix defines a mapping $\boldsymbol{S} : \mathbb{R}^n \to \mathtt{M}^6$. The joint space matrices for revolute joints, $\boldsymbol{S}_R$, and prismatic joints, $\boldsymbol{S}_P$, around or along the coordinate axes are:

$$\boldsymbol{S}_{Rx} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \boldsymbol{S}_{Ry} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \boldsymbol{S}_{Rz} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

$$\boldsymbol{S}_{Px} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \boldsymbol{S}_{Py} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \boldsymbol{S}_{Pz} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

Joints may also rotate about or translate along axes that do not coincide with the reference frame axes. This axis must then be described as a normalized vector. The joint space matrix for a revolute joint thus becomes:

$$\boldsymbol{S}_R = \begin{pmatrix} x \\ y \\ z \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

and for a prismatic joint:

$$\boldsymbol{S}_P = \begin{pmatrix} 0 \\ 0 \\ 0 \\ x \\ y \\ z \end{pmatrix},$$

where $x, y, z$ are the values of the normalized vector.

One can thus express the joint velocities, accelerations, and forces through the joint space matrices. The joint force defines the constrained relative force between the parent body and the child body it is connecting:

$$\boldsymbol{\tau}_i = \boldsymbol{S}_i^T \boldsymbol{f}_i \tag{2.48}$$

The joint velocity defines the constrained relative motion between the parent body and child body of the connecting joint:

$$\boldsymbol{v}_{\boldsymbol{J}i} = \boldsymbol{S}_i \dot{\boldsymbol{q}}_i \tag{2.49}$$

The joint acceleration is defined as the derivative of the joint velocity:

$$\begin{aligned} \boldsymbol{a}_{\boldsymbol{J}i} &= \boldsymbol{S}_i \ddot{\boldsymbol{q}}_i + \dot{\boldsymbol{S}}_i \dot{\boldsymbol{q}}_i \\ &= \boldsymbol{S}_i \ddot{\boldsymbol{q}}_i + (\mathring{\boldsymbol{S}}_i + \boldsymbol{v}_i \times \boldsymbol{S}_i)\dot{\boldsymbol{q}}_i \\ &= \boldsymbol{S}_i \ddot{\boldsymbol{q}}_i + \boldsymbol{c}_{\boldsymbol{J}i} + \boldsymbol{v}_i \times \boldsymbol{v}_{\boldsymbol{J}i} \end{aligned} \tag{2.50}$$

There are two factors affecting $\dot{\boldsymbol{S}}_{\boldsymbol{i}}$: there is the change of the joint space over time itself and the change of the joint space due to the motion of the frame of body $i$. The change of motion of body $i$ is represented by its spatial velocity $\boldsymbol{v}_i$. Hence, (2.50) indicates that the rate of change of the joint space due to the motion of body $i$ is given by $\boldsymbol{v}_i \times \boldsymbol{S}_i \dot{\boldsymbol{q}}_i = \boldsymbol{v}_i \times \boldsymbol{v}_{\boldsymbol{J}i}$, where $\boldsymbol{v}_i\times$ denotes the spatial motion cross product explained in section 2.2.7. Further, $\boldsymbol{c}_{\boldsymbol{J}i}$ represents the change of the joint space due to time itself and is given by:

$$\begin{aligned} \boldsymbol{c}_{\boldsymbol{J}i} &= \mathring{\boldsymbol{S}}_i \dot{\boldsymbol{q}}_i \\ &= \Big(\frac{d\boldsymbol{S}_i}{dt} + \sum_{j=1}^{n} \frac{d\boldsymbol{S}_i}{dq_j}\dot{q}_j\Big)\dot{\boldsymbol{q}}_{\boldsymbol{i}} \end{aligned} \tag{2.51}$$

$\mathring{\boldsymbol{S}}$ is in Featherstone (2008) referred to as the apparent derivative, and from (2.51) the following is observed:

$$\mathring{\boldsymbol{S}} = \frac{d\boldsymbol{S}_i}{dt} + \sum_{j=1}^{n} \frac{d\boldsymbol{S}_i}{dq_j}\dot{q}_j \tag{2.52}$$

It should be mentioned that this relatively complicated equation for joint acceleration applies to a general case and is almost never needed in practice. In the case of only considering prismatic and revolute joints, the joint space is not affected over time. Hence, the so-called apparent derivative is zero: $\mathring{\boldsymbol{S}} = 0 \Rightarrow \boldsymbol{c}_{Ji} = 0$. This simplifies (2.50) to:

$$\boldsymbol{a}_{\boldsymbol{J}i} = \boldsymbol{S}_i \ddot{\boldsymbol{q}}_i + \boldsymbol{v}_i \times \boldsymbol{v}_{\boldsymbol{J}i} \tag{2.53}$$

### Joint Numbering

In a loop-free multi-body system, the connection of bodies via joints can be seen as a directed graph where the bodies are nodes and the joints are the directed edges. Not considering the root body, there are the same number of bodies $\boldsymbol{B}_i$ as joints $\boldsymbol{J}_i$ with $i = 1, \ldots n$.

Joint $i$ connects body $i$ to its parent body. Parent bodies are denoted $\lambda_i$ and the parent array $\lambda$. The bodies are numbered such that $\lambda_i < i$ always holds, and for kinematic chains the numbering is such that $\lambda_i = i - 1$. Additionally, $\kappa(i)$ being the set of bodies that influences body $i$, this numbering has the property that $j < i$, $\forall j \in \kappa(i)$. Similarily, $\nu(i)$ being the set of indices of the bodies in the subtree starting at body $i$, the numbering results in $j > i$, $\forall j \in \nu(i)$.

Another useful set of indices is $\mu(i)$ which contains all indices of the joints of the subtree starting at $i$. For kinematic chains this simplifies to one index $\mu_i = i + 1$.

# CHAPTER 3

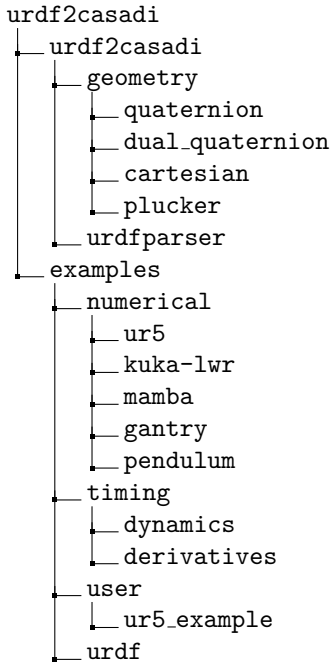
# Implementation

*This chapter is based on the Implementation chapter from the specialization project associated with this thesis. Modifications have been made according to changes during the latter part of the project.*

## 3.1 The Library Structure

For the reader to get a better perspective of the library and the work that has been done, the reader is encouraged to have a look at the open-source implementation of u2c. Throughout the report, it is explained where the relevant work can be found and viewed in the repository. The URL to the open-source implementation of urdf2casadi is: https://github.com/lillmaria/urdf2casadi.

The following direction tree describes the overall folder structure of the library:




This chapter focuses on the implementation of u2c, which is centered in the `urdf2casadi`-folder. The folder contains one module, the `urdfparser`, that inhabits the functionality of u2c and the implementation of the rigid body dynamics algorithms, and a submodule, the `geometry`-folder. This is further explained in the latter, and the reader is encouraged to view the corresponding parts of the open-source library at Github, through the URL given above.

### 3.1.1 `geometry`

The geometry module has been divided into four geometry submodules. All submodules contain the same functionality in the sense that they provide functionality for finding the geometric aspects of the multi-body system, such as transformation matrices, rotation matrices, and inertia matrices. The difference is in what way this information is represented. The `quaternion`-module provides the geometrics in quaternion notation and the `dual_quaternion`-module in dual quaternion notation. Further, the `cartesian`-module represents the original 3D notation where velocities and accelerations are represented as two 3D vectors, and the standard $4 \times 4$ transformation matrices are used. These three submodules were originally implemented for generating the forward kinematics.

In association with the implementation of the rigid body dynamics algorithms, spatial algebra and Plücker coordinates were required. The `plucker`-module contains in many ways the same functionality as the other geometric submodules, given on a spatial form using Plücker coordinates. Functions for finding spatial transforms, such as $\boldsymbol{X_J}$ and $\boldsymbol{X_T}$, as well as spatial inertia and the spatial cross

products are provided, to mention some. A list of the functionality implemented in this module is shown in the directory tree below. This functionality is implemented according to the spatial algebra presented in Chapter 2. Input variables are not included in the tree as the purpose is to give a brief overview of the functionality.

```
geometry
└── plucker
    ├── motion_cross_product()
    ├── force_cross_product()
    ├── XT()
    ├── XJT_prismatic()
    ├── XJT_revolute()
    ├── spatial_inertia_matrix_I0()
    └── spatial_inertia_matrix_Ic()
```

### 3.1.2 urdfparser

The main module of u2c is the `urdfparser`. The module has one class which contains all the necessary functions to provide a robot's dynamics. It is here found functions for parsing the URDF to a robot model and functions for retrieving the robot's dynamics. The implementation of the rigid body dynamics algorithms are thus found here.

The class of the `urdfparser` has the name `URDFparser`. It contains one instance variable, the *robot_description*, which provides a description of the kinematic structure of the robot, based on the information provided by the URDF. By using the `geometry` module and the information retrieved from the URDF, the `URDFparser` provides methods for obtaining symbolic functions of the robot's dynamics and forward kinematics. The directory tree below shows where to find the `URDFparser` class, as well as the methods provided by the class, with the input variables omitted for brevity.

```
urdf2casadi
└── urdf2casadi
    └── class URDFparser
        ├── from_file()
        ├── from_server()
        ├── from_string()
        ├── get_joint_info()
        ├── get_n_joints()
        ├── _model_calculation()
        ├── _apply_external_forces()
        ├── get_gravity_rnea()
        ├── get_coriolis_rnea()
        ├── get_inverse_dynamics_rnea()
        ├── get_inertia_matrix_crba()
        └── get_forward_dynamics_aba()
```

```
|
|__ _get_M()
|__ _get_C()
|__ get_forward_dynamics_crba()
|__ get_forward_kinematics()
```

The class consists of three types of methods:

**Load-methods**
The class provides three methods for loading the robot description from a URDF. The user can choose to load the robot description directly from a URDF file, from a URDF string, or ROS parameter server.[1]

**Private-methods**
The private methods are the methods starting with _. They are not intended to be accessed by users and are used internally in other methods to provide information. They make use of the *robot_description* and the `geometry`-module to retrieve this information.

**Get-methods**
The get-methods represent the functionality provided to the user. They require two inputs from the user: the name of the *root* body and the name of the *tip* body. Based on this input, the get-methods provide the kinematics and dynamics from the root body to the tip body.

## 3.2 Loading the Robot Description

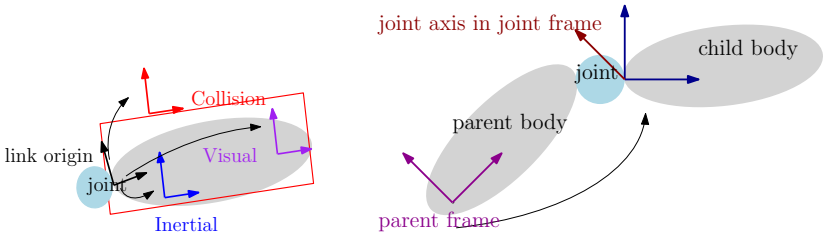
As explained in the previous section, the load methods are essential to get a description of the robot. The input from the user is given in the form of a URDF. In this section, the URDF is presented, and the method for preserving and using its information in u2c is provided.

### 3.2.1 The URDF

The URDF provides a kinematic description of a robot based on a kinematic tree structure using an XML format with different tags for links, joints, and transmissions. The links represent the bodies of the rigid multi-body system. To illustrate, an example of a link tag for a UR5 robot's base link is presented together with an illustrative figure of the information provided by the link tag in Figure 3.1a.

```
<link name="base_link">
    <visual>
      <geometry>
        <mesh filename="package://ur_description/meshes/ur5/
        visual/base.dae"/>
```

[1]The ROS parameter server is a collection of values that can be accessed upon request while ROS is running.




**(a)** Link information. **(b)** Joint information.

**Figure 3.1:** Information provided by the URDF (illustration based on URDF documentation).

```
      </geometry>
      <material name="LightGrey">
        <color rgba="0.7 0.7 0.7 1.0"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <mesh filename="package://ur_description/meshes/ur5/
        collision/base.stl"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="4.0"/>
      <origin rpy="0 0 0" xyz="0.0 0.0 0.0"/>
      <inertia ixx="0.00443333156" ixy="0.0" ixz="0.0"
      iyy="0.00443333156" iyz="0.0" izz="0.0072"/>
    </inertial>
  </link>
```

One can observe that the link tag provides information about the name of the link, its visual, collision, and inertial features. This information is visualized in Figure 3.1a. In the case of the development of u2c, the relevant information is the inertial tag as it gives the inertia and body frame. Further, the joint tag for the joint connecting the base link of the UR5 to its child body is presented:

```
<joint name="shoulder_pan_joint" type="revolute">
    <parent link="base_link"/>
    <child link="shoulder_link"/>
    <origin rpy="0.0 0.0 0.0" xyz="0.0 0.0 0.089159"/>
    <axis xyz="0 0 1"/>
    <limit effort="150.0" lower="-6.28318530718"
    upper="6.28318530718" velocity="3.15"/>
    <dynamics damping="0.0" friction="0.0"/>
```

```
</joint>
```

The joint tag provides relevant joint information such as its type, the axis of rotation or translation, its frame origin position (xyz) and orientation (rpy), as well as its parent and child body. This information is illustrated in Figure 3.1b.

Hence, the URDF provides the information necessary for obtaining a robot's kinematics and dynamics. This includes inertial properties for all bodies, here denoted links, as well as body reference frames, joint location frames, and joint types and axes. From this information, one can obtain the joint motion frames, spatial transforms from parent bodies to child bodies, and spatial inertias. With this information, one can further retrieve the robot's dynamics using the rigid body dynamics algorithms, further explained in section 3.3.

It should be mentioned that the URDF has some shortcomings. The main shortcoming is that the URDF does not support closed kinematic chains, such as Stewart platforms and delta robots, which restricts the usage of u2c to kinematic trees. Further, URDF only allows one robot tag. In practice, this means that the URDF does not support multiple robots, and if one were to represent multiple robots, it is needed to combine them manually as a single robot element. The URDF information is also only provided for a nominal robot, and in practice further calibration may be required for each particular robot instance as the inertial information may differ if tools are attached to the robot or the information is incorrect.

### 3.2.2 urdf_parser_py

As mentioned above, the `URDFparser` class provides three load-methods for the user to load the robot description. These methods make use of the open-source library *urdf_parser_py*, which is part of the standard ROS packages. The library parses the information provided by the URDF to a Python class structure called `robot`. After parsing the URDF to a `robot` class structure, the instance contains the information provided by the URDF, and the information is easily accessible through the `robot` instance. The class instance returned by urdf_parser_py is the instance variable *robot_description* of the `URDFparser` class.

A shortcoming of urdf_parser_py is that it only supports parsing of kinematic chains. Hence, u2c is limited to finding the kinematics and dynamics of kinematic chains when using the urdf_parser_py and cannot find the kinematics and dynamics of trees, such as a dexterous robot hand, even though trees are well-represented by a URDF.

It is also worth noting that the urdf_parser_py package is called urdfdom_py when installed with ROS, and urdf_parser_py when installed independently through pip.

## 3.3 Rigid Body Dynamics Algorithms

In this section, the implementation of the rigid body dynamics algorithms used to retrieve symbolic expressions of the robot's dynamics is presented. These algorithms model the robot as a rigid multi-body system and use spatial vector algebra to retrieve the dynamics. The algorithms presented in this section are:

**Recursive Newton-Euler Algorithm (RNEA)**
The algorithm is used to retrieve the robot's inverse dynamics, i.e. the forces transmitted through the robot's joints, $\boldsymbol{\tau}$, as an expression of the other joint state variables $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$, $\ddot{\boldsymbol{q}}$ such that $\boldsymbol{\tau}_{\text{ID}} = \text{RNEA}(\textit{robot description}, \boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}, \boldsymbol{f}^{ext})$. The algorithm is also used to find the Coriolis forces transmitted through each joint, which is found by a call to RNEA using $\ddot{\boldsymbol{q}} = 0$ and $\boldsymbol{f}^{ext} = 0$:
$\boldsymbol{\tau_C} = \text{RNEA}(\textit{robot description}, \boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{0}, \boldsymbol{0})$ Lastly, RNEA can be used to retrieve the gravitational forces transmitted through each joint, found by a call to RNEA using $\dot{\boldsymbol{q}} = 0$, $\ddot{\boldsymbol{q}} = 0$, and $\boldsymbol{f}^{ext} = 0$:
$\boldsymbol{\tau_G} = \text{RNEA}(\textit{robot description}, \boldsymbol{q}, \boldsymbol{0}, \boldsymbol{0}, \boldsymbol{0})$

**Articulated Body Algorithm (ABA)**
The algorithm is used to retrieve the robot's forward dynamics, i.e. the joint accelerations $\ddot{\boldsymbol{q}}$ as an expression of the other joint state variables $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$, $\boldsymbol{\tau}$ such that $\ddot{\boldsymbol{q}} = \text{ABA}(\textit{robot description}, \boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{\tau}, \boldsymbol{f}^{ext})$.

**Composite Rigid Body Algorithm (CRBA)**
The algorithm is used to obtain the joint state inertia matrix $\boldsymbol{M}$ as an expression of the joint positions $\boldsymbol{q}$ such that $\boldsymbol{M} = \boldsymbol{M}(\boldsymbol{q})$. This algorithm can further be used to obtain the forward dynamics by combining it with the equation of motion for a multi-body system.

### 3.3.1 The Model Calculation Routine

The algorithms mentioned above find the dynamic properties of the robot based on the robot description and the joint state variables by modeling the robot as a rigid multi-body system. Thus, the algorithms start with the same preparatory step: obtaining the necessary joint space matrices, spatial transforms, and spatial intertias. The procedure for obtaining these quantities is called the *model calculation routine* and is implemented in the `_model_calculation()`-method of the `URDFparser` class. The pseudocode of the model calculation routine is given in Algorithm 1.

**Algorithm 1** Model Calculation Routine

**Input:** $\boldsymbol{q}$, chain
**Output:** $\boldsymbol{X_{JT}}$, $\boldsymbol{I}$, $\boldsymbol{S}$

**for** item **in** chain **do**
  $i = 1$
  **if** item **is** joint **then**
    **if** joint type **is** fixed **then**
      $\boldsymbol{X_T}$ = plucker.XT(xyz, rpy)
    **else if** joint type **is** prismatic **then**
      $\boldsymbol{X_{JT}}_i$ = plucker.XJT_prismatic(xyz, rpy, $\boldsymbol{q}_i$)
      **if** prev_joint **is** fixed **then**
        $\boldsymbol{X_{JT}}_i = \boldsymbol{X_T}\boldsymbol{X_{JT}}_i$
      **end if**
      $\boldsymbol{S}_i$ = [joint.axis[0], joint.axis[1], joint.axis[2], 0, 0, 0]
      $i = i + 1$
    **else if** joint type **is** revolute **then**
      $\boldsymbol{X_{JT}}_i$ = plucker.XJT_revolute(xyz, rpy, $\boldsymbol{q}_i$)
      **if** prev_joint **is** fixed **then**
        $\boldsymbol{X_{JT}}_i = \boldsymbol{X_T}\boldsymbol{X_{JT}}_i$
      **end if**
      $\boldsymbol{S}_i$ = [joint.axis[0], joint.axis[1], joint.axis[2], 0, 0, 0]
      $i = i + 1$
    **end if**
    prev_joint = joint.type
  **end if**

  **if** item **is** link **then**
    $\boldsymbol{I}_i$ = plucker.spatial_inertia_matrix_I0(ixx, ixy, ixz, iyy, iyz, izz, mass,xyz)
    **if** prev_joint **is** fixed **then**
      $\boldsymbol{I}_i = \boldsymbol{I}_{i-1} + \boldsymbol{X_T}_i^T \boldsymbol{I}_i \boldsymbol{X_T}_i$
    **end if**
  **end if**
**end for**
**return** $\boldsymbol{X_{JT}}$, $\boldsymbol{I}$, $\boldsymbol{S}$

In Algorithm 1, the joint positions, $\boldsymbol{q}$, are given as input in the form of CasADi `SX` symbols. The other input, the chain, is a dictionary containing the joints and links that form the robotic chain. It contains relevant information extracted from the URDF, such as the type and axis of the joints, and inertial properties of the links.

Further, one can observe the procedure constructs the joint space matrix for each joint and uses the `plucker` geometry module to construct the transforms from the parent body frame to the joint location frame $\boldsymbol{X_T}_i$, and the transforms

from the joint location frame to the joint motion frame $\boldsymbol{X_{J}}_i$ in one calculation. The transform denoted $\boldsymbol{X_{JT}}_i$ thus denotes the total transform from the parent body frame to the joint motion frame of body $i$, in the former chapter given by ${}^i\boldsymbol{X}_{\lambda_i} = \boldsymbol{X_J}_i\boldsymbol{X_T}_i$. Inertial properties of the link are used by the `plucker`-module to construct the spatial inertias.

From the pseudocode, one can also observe that fixed joints are treated as an extension of the parent body, such that the parent body and child body of the connecting fixed joint are represented as a single rigid body. Spatial algebra makes this easy, as the total inertia of the two connected rigid bodies is found by adding the two inertias together, and the latter inertia is needed to be transformed into the same frame as the first, as seen in line 27. Further, the total transform is found by multiplying the two spatial transforms together, where the spatial transform from the parent body frame to the joint motion frame using a fixed joint becomes $\boldsymbol{X_T}$. This can be observed in line 10 and line 17.

The procedure returns three lists that represent the modeling of a rigid multi-body system: $\boldsymbol{X}$ containing the transforms from reference frames of parent bodies to their child body reference frame, $\boldsymbol{I}$ containing the spatial inertia matrices, and $\boldsymbol{S}$ containing the joint space matrices.

In the following sections, the implementation of the three rigid body dynamics algorithms is explained. These are implemented in the following get-methods of the `URDFparser`:

```
class URDFparser
├── get_gravity_rnea()
├── get_coriolis_rnea()
├── get_inverse_dynamics_rnea()
├── get_forward_dynamics_aba()
├── get_inertia_matrix_crba()
└── get_forward_dynamics_crba()
```

The methods describe which dynamic properties they return as well as the initials of the algorithms used to obtain the result. The implementation of these algorithms is now presented, and the following can be assumed for each of the methods:

- Before starting the algorithm calculation, the model calculation procedure has been performed. From this, ${}^i\boldsymbol{X}_{\lambda_i}$ are retrieved, and can further be used to obtain[2] ${}^{\lambda_i}\boldsymbol{X}_i^*$, ${}^i\boldsymbol{X}_0$, and ${}^i\boldsymbol{X}_0^*$ where the implementation requires it.
- Whenever the joint state variables, ($\boldsymbol{q}$, $\dot{\boldsymbol{q}}$, $\ddot{\boldsymbol{q}}$, $\boldsymbol{\tau}$), are used, they are declared as CasADi `SX` matrices. The `SX` data type is used for minimal overhead as explained in section 2.1.2.
- The algorithms are implemented under the assumption that the robot mechanism is a kinematic chain. This assumption can be made as the urdf_parser_py only supports kinematic chains. The effect of this will be further explained where it affects the implementation.

[2] By using (2.22), (2.23), and (2.12).

### 3.3.2 Recursive Newton-Euler Algorithm

RNEA is an efficient algorithm, mostly used for calculating the inverse dynamics of a rigid multi-body system. The algorithm complexity is $O(n)$ where $n$ is the number of bodies in the rigid multi-body system. The reason for its efficiency is that the algorithm exploits recurrence relations and hence offers a systematic way to keep away from unnecessary repetition in calculating sequences of related quantities.

The algorithm is computed using the following steps:

1. Position, velocity, and acceleration of all bodies are computed.
2. The forces required on the bodies to produce the acceleration obtained in step 1 are computed using the equation of motion for a rigid body.
3. The results from step 1 and 2 are used to compute the forces transmitted over each joint.

**Step 1** consists of finding the quantities $\boldsymbol{v}_i$ and $\boldsymbol{a}_i$ for all bodies $i = 1, \dots n$. The quantities can be found recursively by the formulas:

$$\boldsymbol{v}_i = \boldsymbol{v}_{\lambda i} + \boldsymbol{v}_{\boldsymbol{J}i} \tag{3.1}$$

$$\boldsymbol{a}_i = \boldsymbol{a}_{\lambda i} + \boldsymbol{a}_{\boldsymbol{J}i} \tag{3.2}$$

From (3.1) one can observe that the spatial velocity can be found by adding the velocity of the joint connecting body $i$ to its parent body $\lambda_i$ to the velocity of the parent body $\lambda_i$, where the joint velocity $\boldsymbol{v}_{\boldsymbol{J}i}$ is given by (2.49). The initial value is given as $\boldsymbol{v}_0 = 0$ since body $\boldsymbol{B}_0$ is fixed. From (3.2) one can observe that the spatial acceleration of body $i$ can be found by the same approach as the spatial velocity of body $i$, where the acceleration of the joint $i$ is given by (2.53) and its initial value is $\boldsymbol{a}_0 = -\boldsymbol{a}_g$, $\boldsymbol{a}_g$ being the gravity constant. By setting $\boldsymbol{a}_0 = -\boldsymbol{a}_g$ a uniform gravitational field is modeled so that one does not need to account for the gravitational forces in the external forces acting on the system. Using this trick, $\boldsymbol{a}_i$ is not the true acceleration of body $i$ as it is offset by the gravitational acceleration vector. It should be mentioned that the gravitational forces are set optionally by the user of u2c. If the user does not explicitly define the gravitational vector, the gravitational forces are set to zero: $\boldsymbol{a}_0 = \boldsymbol{0}$.

Transforming (3.1) and (3.2) to body $i$ coordinates and using the expressions for $\boldsymbol{v}_{\boldsymbol{J}i}$ and $\boldsymbol{a}_{\boldsymbol{J}i}$ given by (2.49) and (2.53), the formulas used in the implementation of the algorithm become:

$$\boldsymbol{v}_i = {}^i\boldsymbol{X}_{\lambda_i}\boldsymbol{v}_{\lambda i} + \boldsymbol{S}_i\dot{\boldsymbol{q}}_i \tag{3.3}$$

$$\boldsymbol{a}_i = {}^i\boldsymbol{X}_{\lambda_i}\boldsymbol{a}_{\lambda i} + \boldsymbol{S}_i\ddot{\boldsymbol{q}}_i + \boldsymbol{v}_i \times \boldsymbol{S}_i\dot{\boldsymbol{q}}_i \tag{3.4}$$

**Step 2** consists of finding the net force acting on body $i$, denoted $\boldsymbol{f}_i^B$. This force is given by the equation of motion of a rigid body and relates the acceleration of body $i$ obtained in step 1 to the force:

$$\boldsymbol{f}_i^B = \boldsymbol{I}_i\boldsymbol{a}_i + \boldsymbol{v}_i \times^* \boldsymbol{I}_i\boldsymbol{v}_i \tag{3.5}$$

Step 1 and 2 are implemented as one forward pass from $i = 1$ to $n$.

**Step 3** consists of finding the forces transmitted across the joints from the forces acting on the bodies. $\boldsymbol{f}_i$ referres to the force transmitted from parent body $\lambda_i$ to body $i$ through joint $i$, and $\boldsymbol{f}_i^{ext}$ denotes potential external forces acting upon body $i$. The net force on body $i$ is then:

$$\boldsymbol{f}_i^B = \boldsymbol{f}_i + \boldsymbol{f}_i^{ext} - \sum_{j \in \mu(i)} \boldsymbol{f}_j \tag{3.6}$$

(3.6) can be rearranged to provide a recurrence relation for the joint forces:

$$\boldsymbol{f}_i = \boldsymbol{f}_i^B - \boldsymbol{f}_i^{ext} + \sum_{j \in \mu(i)} \boldsymbol{f}_j \tag{3.7}$$

(3.7) is the general equation for the joint forces and $\mu(i)$ is the set of children bodies of body $i$. Expressed in body $i$ coordinate, the expression to be used in the algorithm becomes:

$$\boldsymbol{f}_i = \boldsymbol{f}_i^B - {}^i\boldsymbol{X}_0^* \boldsymbol{f}_i^{ext} + \sum_{j \in \mu(i)} {}^i\boldsymbol{X}_j^* \boldsymbol{f}_j \tag{3.8}$$

It should be mentioned that since URDF only support kinematic chains, $\mu(i) = i + 1$, and (3.8) can be implemented as:

$$\boldsymbol{f}_{\lambda_i} = \boldsymbol{f}_{\lambda_i}^B - {}^i\boldsymbol{X}_0^* \boldsymbol{f}_{\lambda_i}^{ext} + {}^i\boldsymbol{X}_{\lambda_i}^* \boldsymbol{f}_i \tag{3.9}$$

Thus the spatial force across joint $\boldsymbol{f}_i$ for all joints $i = 1, \ldots, n$ can be found by a backward pass from $n - 1$ to 0.

Having computed the spatial force across each joint, the forces across each joint can be found:

$$\boldsymbol{\tau}_i = \boldsymbol{S}_i^T \boldsymbol{f}_i. \tag{3.10}$$

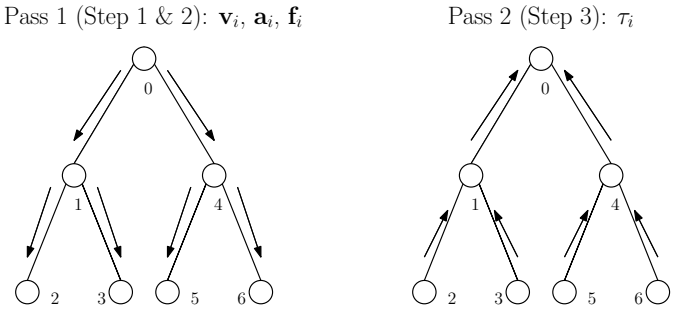



**Figure 3.2:** Passes of RNEA.

Figure 3.2 illustrates how RNEA is performed on a branched kinematic tree illustrated as a graph. The nodes in the graph represent the bodies of the kinematic tree. The pseudocode implementation is shown in Algorithm 2.

**Algorithm 2** Recursive Newton-Euler Algorithm

**Input:** $\boldsymbol{X}$, $\boldsymbol{I}$, $\boldsymbol{S}$
**Output:** $\boldsymbol{\tau}$

1: $\boldsymbol{v}_0 = \boldsymbol{0}$
2: $\boldsymbol{a}_0 = -\boldsymbol{a}_g$
3: **for** $i = 1$ **to** $n$ **do**
4: $\quad \boldsymbol{v}_i = {}^i\boldsymbol{X}_{\lambda_i}\boldsymbol{v}_{\lambda_i} + \boldsymbol{S}_i\dot{\boldsymbol{q}}_{\boldsymbol{i}}$
5: $\quad \boldsymbol{a}_i = {}^i\boldsymbol{X}_{\lambda_i}\boldsymbol{a}_{\lambda_i} + \boldsymbol{S}_i\ddot{\boldsymbol{q}}_{\boldsymbol{i}} + \boldsymbol{v}_i \times \boldsymbol{S}_i\dot{\boldsymbol{q}}_{\boldsymbol{i}}$
6: $\quad \boldsymbol{f}_i = \boldsymbol{I}\boldsymbol{a}_i + \boldsymbol{v}_i \times^* \boldsymbol{I}\boldsymbol{v}_i - {}^i\boldsymbol{X}_0^*\boldsymbol{f}_i^{ext}$
7: **end for**
8: **for** $i = n - 1$ **to** $0$ **do**
9: $\quad \boldsymbol{\tau}_i = \boldsymbol{S}_i^T\boldsymbol{f}_i$
10: $\quad$ **if** $\lambda_i \neq 0$ **then**
11: $\quad\quad \boldsymbol{f}_{\lambda_i} = \boldsymbol{f}_{\lambda_i} + {}^{\lambda_i}\boldsymbol{X}_i^*\boldsymbol{f}_i$
12: $\quad$ **end if**
13: **end for**

One can observe that the two passes illustrated in Figure 3.2 are implemented. In the forward pass, steps 1 and 2 are computed: the spatial forces acting on the bodies are obtained using the equation of motion and exploiting the recurrence relations between the spatial velocities and accelerations. Step 3 is performed in the backward pass where the joint and body forces are calculated.

The original version of the algorithm was formulated using 3D vectors and appears to be very different from this version with 6D vectors. However, the only difference of significance is that this implementation uses spatial acceleration, which differs from the classical acceleration as described in section 2.2.8. Advantages of the spatial version of RNEA are its compact and easy form, and that the algorithm is insensitive to joint type. This is opposed to the original version of RNEA, which requires different expressions for revolute and prismatic joints. This information is embodied in the joint space matrix in spatial vector algebra.

### 3.3.3 Articulated Body Algorithm

By using ABA, one can compute the forward dynamics of a kinematic tree with $O(n)$ operations. The algorithm was first seen in Featherstone (1983) although others have described various variants. The implementation of ABA in u2c is based upon Featherstone's original version.

In the forward dynamics problem, the joint accelerations and the joint constraint forces are considered the unknowns. ABA is based on formulating equations that must be satisfied by the joint acceleration and joint force and propagating these relations to neighbor bodies until the equations are solvable. Propagation algorithms are known for their complexity but more importantly, their efficiency.

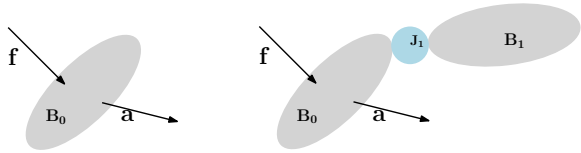
The basis of ABA is to exploit the information captured in the articulated body inertia. The articulated body inertia is the inertia that a body seems to have when being a part of a rigid multi-body system. It is further explained in the following.

#### Articulated Body Equation of Motion

The equation of motion for a single rigid body, denoted $\boldsymbol{B}_0$, is given by:

$$\boldsymbol{f} = \boldsymbol{I}\boldsymbol{a} + \boldsymbol{p} \tag{3.11}$$

The equation describes the relationship between the force applied to the single rigid body, $\boldsymbol{f}$, and the resulting acceleration, $\boldsymbol{a}$. $\boldsymbol{p}$ is the bias force describing the necessary force for the body to produce zero acceleration. The equation of motion for a single rigid body is illustrated in Figure 3.3a.




**(a)** Rigid body.

**(b)** Articulated body composed of two rigid bodies.

**Figure 3.3:** Illustration of the articulated body equation of motion.

To explain the articulated body inertia one can imagine that the single rigid body gets a second body, $\boldsymbol{B}_1$, attached to it and that they are connected by a joint. The system becomes a rigid multi-body system and the whole system can be considered as an articulated body consisting of the original body $\boldsymbol{B}_0$ and the new attached body $\boldsymbol{B}_1$. This case is illustrated in Figure 3.3b. When now applying a force to body $\boldsymbol{B}_0$, the second body $\boldsymbol{B}_1$ will affect the acceleration response. Thus, the equation of motion for body $\boldsymbol{B}_0$ changes to:

$$\boldsymbol{f} = \boldsymbol{I}^A\boldsymbol{a} + \boldsymbol{p}^A \tag{3.12}$$

$\boldsymbol{I}^A$ and $\boldsymbol{p}^A$ are the articulated body inertia and bias force, respectively. The equation describes the relationship between the applied force and the acceleration response for the rigid body $\boldsymbol{B}_0$ while taking into account the dynamic effects of the other body $\boldsymbol{B}_1$ and joint connecting them. The dynamic effects of the attached bodies and joints are captured in the articulated inertia and bias force, which is exploited by ABA.

The articulated body inertia shares some common properties with the rigid body inertia: it is a symmetric, positive definite matrix that represents a mapping from $\mathtt{M}^6$ to $\mathtt{F}^6$ and obeys the same transformation rule. Nonetheless, a significant difference is that the articulated body inertia represents the mapping from acceleration to force in contrast to rigid body inertia, which represents the mapping from velocity to momentum.

### The Articulated Body Algorithm Passes

ABA computes the joint accelerations in three passes:

1. The positions, velocities, and rigid-body bias forces are computed in a forward pass.
2. The articulated body inertia and articulated bias force are computed for all bodies in a backward pass.
3. $\boldsymbol{I}_i^A$ and $\boldsymbol{p}_i^A$ computed in pass 2 are used to compute the joint accelerations in a forward pass.

**Pass 1** consists of calculating the velocity-product accelerations, i.e. $\boldsymbol{v}_i \times \boldsymbol{v}_{\boldsymbol{J}i}$, further denoted $\boldsymbol{c}_i$. The bias forces $\boldsymbol{p}_i$ must also be obtained. As these are functions of the body velocities, it is necessary to compute $\boldsymbol{v}_i$ first. The general equations become:

$$\boldsymbol{v}_{\boldsymbol{J}i} = \boldsymbol{S}_i \dot{\boldsymbol{q}}_i \tag{3.13}$$

$$\boldsymbol{v}_i = \boldsymbol{v}_{\lambda_i} + \boldsymbol{v}_{\boldsymbol{J}i} \tag{3.14}$$

$$\boldsymbol{c}_i = \boldsymbol{v}_i \times \boldsymbol{v}_{\boldsymbol{J}i} \tag{3.15}$$

$$\boldsymbol{p}_i = \boldsymbol{v}_i \times^* \boldsymbol{I}_i \boldsymbol{v}_i - \boldsymbol{f}_i^x \tag{3.16}$$

These equations are similar to the equations of the forward pass of the RNEA. The difference is that it is only the velocity-product part of the acceleration that is considered as $\ddot{\boldsymbol{q}}$ is unknown. Further, the results from (3.16) are used in pass 2 to obtain the articulated bias force and the results from (3.15) are used in pass 3 to obtain $\ddot{\boldsymbol{q}}$.

**Pass 2** consists of computing the articulated inertia matrices and bias forces. A set of articulated bodies, $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_n$ are defined such that $\boldsymbol{A}_i$ contains the bodies in the subtree growing from $\boldsymbol{A}_i$. With this definition of the articulated bodies, one can obtain $\boldsymbol{I}_i^A$ and $\boldsymbol{p}_i^A$ from the children of body $i$. Since an articulated body can consist of only a single rigid body, in which case the articulated body inertia

and bias force are equal to the rigid body inertia and bias force, this provides the starting point for recursive calculation of $\boldsymbol{I}^A$ and $\boldsymbol{p}^A$:

$$\boldsymbol{I}_i^A = \boldsymbol{I}_i + \sum_{j \in \mu(i)} \boldsymbol{I}_j^a \tag{3.17}$$

$$\boldsymbol{p}_i^A = \boldsymbol{p}_i + \sum_{j \in \mu(i)} \boldsymbol{p}_j^a \tag{3.18}$$

As u2c only supports kinematic chains, each body in the kinematic tree has one child. Taking this into consideration, the implementation of (3.17) and (3.18) can be simplified to:

$$\boldsymbol{I}_i^A = \boldsymbol{I}_i + \boldsymbol{I}_{\mu_i}^a \tag{3.19}$$

$$\boldsymbol{p}_i^A = \boldsymbol{p}_i + \boldsymbol{p}_{\mu_i}^a \tag{3.20}$$

Comparing (3.17) and (3.18) to (3.19) and (3.20), respectively, one can observe that for kinematic chains $j = \mu(i) = i + 1$. Since the quantities are obtained in a backward pass, (3.19) and (3.20) are implemented using the ($\lambda_i$, i)-pair rather than the (i, $\mu_i$)-pair. The equations can thus be rewritten:

$$\boldsymbol{I}_{\lambda_i}^A = \boldsymbol{I}_{\lambda_i} + \boldsymbol{I}_i^a \tag{3.21}$$

$$\boldsymbol{p}_{\lambda_i}^A = \boldsymbol{p}_{\lambda_i} + \boldsymbol{p}_i^a \tag{3.22}$$

The quantities $\boldsymbol{I}_i^a$ and $\boldsymbol{p}_i^a$ are a result of propagating $\boldsymbol{I}_i^A$ and $\boldsymbol{p}_i^A$ through joint $i$ and can thus be expressed as a function the joint motion constraint for joint $i$ given by the joint space matrix $\boldsymbol{S}_i$:

$$\boldsymbol{I}_i^a = \boldsymbol{I}_i^A - \boldsymbol{I}_i^A \boldsymbol{S}_i (\boldsymbol{S}_i \boldsymbol{I}_i^A \boldsymbol{S}_i)^{-1} \boldsymbol{S}_i^T \boldsymbol{I}_i^A \tag{3.23}$$

$$\boldsymbol{p}_i^a = \boldsymbol{p}_i^A + \boldsymbol{I}_i^a \boldsymbol{c}_i + \boldsymbol{I}_i^A \boldsymbol{S}_i (\boldsymbol{S}_i \boldsymbol{I}_i^A \boldsymbol{S}_i)^{-1} (\boldsymbol{\tau}_i - \boldsymbol{S}_i^T \boldsymbol{p}_i^A) \tag{3.24}$$

To put the quantities in context, $\boldsymbol{I}_i^A$ and $\boldsymbol{p}_i^A$ express the force across the connecting joint as a function of the child body's acceleration, while the quantities $\boldsymbol{I}_i^a$ and $\boldsymbol{p}_i^a$ express the same force as a function of the parent body's acceleration:

$$\begin{aligned} \boldsymbol{f}_i &= \boldsymbol{I}_i^A \boldsymbol{a}_i + \boldsymbol{p}_i^A \\ &= \boldsymbol{I}_i^a \boldsymbol{a}_{\lambda_i} + \boldsymbol{p}_i^a \end{aligned} \tag{3.25}$$

To summarize, pass 2 of the algorithm consists of calculating the articulated body inertia and bias force for every body in the system by using (3.21) and (3.22) together with (3.23) and (3.24) in a backward pass from $n - 1$ to 0.

**Pass 3** consists of a forward pass to obtain the joint accelerations. The joint accelerations are obtained by exploiting the fact that a force from a body through a joint is constrained such that the force becomes:

$$\boldsymbol{\tau}_i = \boldsymbol{S}_i^T \boldsymbol{f}_i \tag{3.26}$$

Further, by using the equation of motion for an articulated body to express $\boldsymbol{f}_i$, (3.26) can be expressed through the articulated inertia and bias force as:

$$\boldsymbol{\tau}_i = \boldsymbol{S}_i^T(\boldsymbol{I}^A\boldsymbol{a}_i + \boldsymbol{p}^A) \tag{3.27}$$

where $\boldsymbol{a}_i$ can be expressed recursively as $\boldsymbol{a}_i = \boldsymbol{a}_{\lambda_i} + \boldsymbol{a}_{\boldsymbol{J}i}$ such that (3.27) can be expressed as:

$$\boldsymbol{\tau}_i = \boldsymbol{S}_i^T(\boldsymbol{I}^A(\boldsymbol{a}_{\lambda_i} + \boldsymbol{S}\ddot{\boldsymbol{q}} + \boldsymbol{v}_i \times \boldsymbol{v}_{\boldsymbol{j}i}) + \boldsymbol{p}^A) \tag{3.28}$$

From this equation, the joint accelerations can be obtained:

$$\ddot{\boldsymbol{q}}_i = (\boldsymbol{S}_i^T\boldsymbol{I}_i^A\boldsymbol{S})^{-1}(\boldsymbol{\tau}_i - \boldsymbol{S}_i^T\boldsymbol{I}_i^A(\boldsymbol{a}_{\lambda_i} + \boldsymbol{c}_{\boldsymbol{i}}) - \boldsymbol{S}_i^T\boldsymbol{p}_i^A) \tag{3.29}$$

And last, the body accelerations can be found:

$$\boldsymbol{a}_i = \boldsymbol{a}_{\lambda_i} + \boldsymbol{c}_i + \boldsymbol{S}_i\ddot{\boldsymbol{q}}_i \tag{3.30}$$

To summarize, pass 3 obtains the joint accelerations by expressing the joint forces as a product of the joint constraint and the equation of motion for an articulated body. The joint and body accelerations are thus obtained by (3.29) and (3.30), respectively. The quantities that are included in these expressions are found in the former passes.

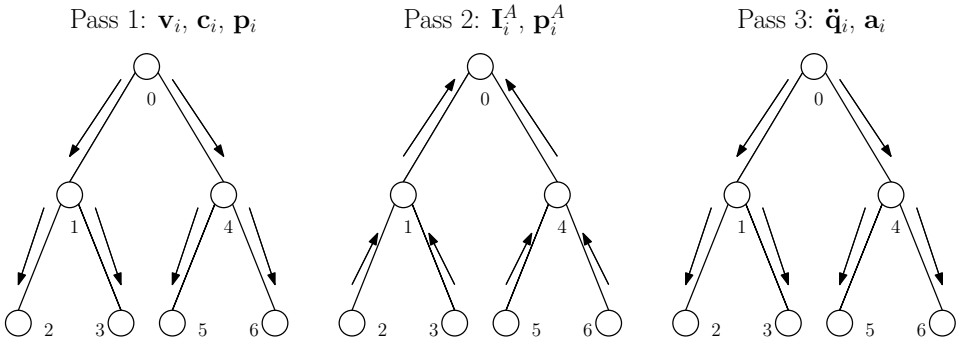



**Figure 3.4:** Passes of ABA.

Figure 3.4 illustrates the three passes performed on a branched kinematic tree represented as a graph where the nodes of the graphs represent the bodies.

### Subexpressions

In the equations explained above, there are several common subexpressions. To make a rather complex implementation simpler, the following subexpressions are defined:

$$\boldsymbol{U}_i = \boldsymbol{I}_i^A\boldsymbol{S}_i \tag{3.31}$$

$$\boldsymbol{D}_i = \boldsymbol{S}_i^T\boldsymbol{U}_i \tag{3.32}$$

$$\boldsymbol{u}_i = \boldsymbol{\tau}_i - \boldsymbol{S}_i^T\boldsymbol{p}_i^A \tag{3.33}$$

$$\boldsymbol{a}_i' = \boldsymbol{a}_{\lambda_i} + \boldsymbol{c}_i \tag{3.34}$$

By using these subexpressions, (3.23), (3.24), (3.29), and, (3.30) simplifies to:

$$\boldsymbol{I}_j^a = \boldsymbol{I}_j^A - \boldsymbol{U}_j \boldsymbol{D}_j^{-1} \boldsymbol{U}_j^T \tag{3.35}$$

$$\boldsymbol{p}_j^a = \boldsymbol{p}_j^A + \boldsymbol{I}_j^a \boldsymbol{c}_j + \boldsymbol{U}_j \boldsymbol{D}_j^{-1} \boldsymbol{u}_j \tag{3.36}$$

$$\ddot{\boldsymbol{q}}_i = \boldsymbol{D}_i^{-1}(\boldsymbol{u}_i - \boldsymbol{U}_i^T \boldsymbol{a}_i') \tag{3.37}$$

$$\boldsymbol{a}_i = \boldsymbol{a}_i' + \boldsymbol{S}_i \ddot{\boldsymbol{q}}_{\boldsymbol{i}} \tag{3.38}$$

### Body Coordinate Equations

The algorithms are implemented in body coordinates and must, therefore, be transformed from the general equations to body coordinate equations. The equations used in the implementation of ABA are given in body coordinates below. They are expressed using the subexpressions given above and sorted by the passes where they are used.

Pass 1 (forward):

$$\begin{aligned}
\boldsymbol{v}_{\boldsymbol{J}i} &= \boldsymbol{S}_i \dot{\boldsymbol{q}}_i \\
\boldsymbol{v}_i &= {}^i\boldsymbol{X}_{\lambda_i} \boldsymbol{v}_{\lambda_i} + \boldsymbol{v}_{\boldsymbol{J}i} \\
\boldsymbol{c}_i &= \boldsymbol{v}_i \times \boldsymbol{v}_{\boldsymbol{J}i} \\
\boldsymbol{p}_i &= \boldsymbol{v}_i \times^* \boldsymbol{I}_i \boldsymbol{v}_i - {}^i\boldsymbol{X}_0^* \boldsymbol{f}_i^x
\end{aligned}$$

Pass 2 (backward):

$$\begin{aligned}
\boldsymbol{I}_{\lambda_i}^A &= \boldsymbol{I}_{\lambda_i} + {}^{\lambda_i}\boldsymbol{X}_i^* \boldsymbol{I}_i^{a\,i} \boldsymbol{X}_{\lambda_i} \\
\boldsymbol{p}_{\lambda_i}^A &= \boldsymbol{p}_{\lambda_i} + {}^{\lambda_i}\boldsymbol{X}_i^* \boldsymbol{p}_i^a \\
\boldsymbol{U}_i &= \boldsymbol{I}_i^A \boldsymbol{S}_i \\
\boldsymbol{D}_i &= \boldsymbol{S}_i^T \boldsymbol{U}_i \\
\boldsymbol{u}_i &= \boldsymbol{\tau}_i - \boldsymbol{S}_i^T \boldsymbol{p}_i^A \\
\boldsymbol{I}_i^a &= \boldsymbol{I}_i^A - \boldsymbol{U}_i \boldsymbol{D}_i^{-1} \boldsymbol{U}_i^T \\
\boldsymbol{p}_i^a &= \boldsymbol{p}_i^A + \boldsymbol{I}_i^a \boldsymbol{c}_i + \boldsymbol{U}_i \boldsymbol{D}_i^{-1} \boldsymbol{u}_i
\end{aligned}$$

Pass 3 (forward):

$$\begin{aligned}
\boldsymbol{a}_i' &= {}^i\boldsymbol{X}_{\lambda_i} \boldsymbol{a}_{\lambda_i} + \boldsymbol{c}_i \\
\ddot{\boldsymbol{q}}_i &= \boldsymbol{D}_i^{-1}(\boldsymbol{u}_i - \boldsymbol{U}_i^T \boldsymbol{a}_i') \\
\boldsymbol{a}_i &= \boldsymbol{a}_i' + \boldsymbol{S}_i \ddot{\boldsymbol{q}}_{\boldsymbol{i}}
\end{aligned}$$

Given the above, the pseudocode of ABA is described in Algorithm 3.

**Algorithm 3** Articulated Body Algorithm

**Input:** $\boldsymbol{X}$, $\boldsymbol{I}$, $\boldsymbol{S}$
**Output:** $\ddot{\boldsymbol{q}}$

$\boldsymbol{v}_0 = \boldsymbol{0}$
$\boldsymbol{a}_0 = -\boldsymbol{a}_g$
**for** $i = 1$ **to** $n$ **do**
    $\boldsymbol{v}_i = {}^i\boldsymbol{X}_{\lambda_i}\boldsymbol{v}_{\lambda_i} + \boldsymbol{S}_i\dot{\boldsymbol{q}}_i$
    $\boldsymbol{c}_i = \boldsymbol{v}_i \times \boldsymbol{S}_i\dot{\boldsymbol{q}}_i$
    $\boldsymbol{p}_i = \boldsymbol{v}_i \times^* \boldsymbol{I}_i\boldsymbol{v}_i - {}^i\boldsymbol{X}_0^*\boldsymbol{f}_i^x$
**end for**
**for** $i = n - 1$ **to** $0$ **do**
    $\boldsymbol{U}_i = \boldsymbol{I}_i^A\boldsymbol{S}_i$
    $\boldsymbol{D}_i = \boldsymbol{S}_i^T\boldsymbol{U}_i$
    $\boldsymbol{u}_i = \boldsymbol{\tau}_i - \boldsymbol{p}_i^A$
    **if** $\lambda_i \neq 0$ **then**
        $\boldsymbol{I}_i^a = \boldsymbol{I}_i^A - \boldsymbol{U}_i\boldsymbol{D}_i^{-1}\boldsymbol{U}_i^T$
        $\boldsymbol{p}_i^a = \boldsymbol{p}_i^A + \boldsymbol{I}_i^a\boldsymbol{c}_i + \boldsymbol{U}_i\boldsymbol{D}_i^{-1}\boldsymbol{u}_i$
        $\boldsymbol{I}_{\lambda_i}^A = \boldsymbol{I}_{\lambda_i} + {}^{\lambda_i}\boldsymbol{X}_i^*\boldsymbol{I}_i^{a\,i}\boldsymbol{X}_{\lambda_i}$
        $\boldsymbol{p}_{\lambda_i}^A = \boldsymbol{p}_{\lambda_i} + {}^{\lambda_i}\boldsymbol{X}_i^*\boldsymbol{p}_i^a$
    **end if**
**end for**
**for** $i = 1$ **to** $n$ **do**
    $\boldsymbol{a}_i' = {}^i\boldsymbol{X}_{\lambda_i}\boldsymbol{a}_{\lambda_i} + \boldsymbol{c}_i$
    $\ddot{\boldsymbol{q}}_i = \boldsymbol{D}_i^{-1}(\boldsymbol{u}_i - \boldsymbol{U}_i^T\boldsymbol{a}_i')$
    $\boldsymbol{a}_i = \boldsymbol{a}_i' + \boldsymbol{S}_i\ddot{\boldsymbol{q}}_i$
**end for**

### 3.3.4 Composite Rigid Body Algorithm

CRBA is used for obtaining the inertia matrix $\boldsymbol{M}$. To do so, the algorithm computes its values recursively and only computes the nonzero entries, thus exploiting sparsity to make the algorithm more efficient. The sparsity is the reason why the calculation cost of $\boldsymbol{M}$ is only $O(nd^2)$ where $d$ is the depth of the kinematic tree.

By using the equation of motion for the total system, the calculation of the inertia matrix using CRBA can be used to obtain the joint accelerations. In the matter of calculating the forward dynamics, CRBA gives the same result as ABA but uses a different approach. ABA makes a fixed number of passes through the kinematic tree, where for each pass, a fixed number of calculations per body is performed, leading to $O(n)$ complexity. When used to obtain the forward dynamics, CRBA has a worst case of $O(n^3)$ as it has to calculate the elements of an $n \times n$ matrix and then factorize it to solve a set of $n$ linear equations to obtain the acceleration variables. However, it is more accurate to describe the complexity as $O(nd^2)$. In cases where the kinematic tree contains few bodies, $O(nd^2)$ algorithms can match or slightly exceed the speed of $O(n)$ algorithms. Since obtaining the forward dynamics with ABA is the best result for systems containing a large number of bodies, while CRBA can be a better approach for systems with few bodies, both algorithms are implemented in u2c.

#### Rigid Multi-Body Equation of Motion

The equation of motion for a rigid multi-body system is described by Featherstone (2008) as:

$$\boldsymbol{\tau} = \boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C_2}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{f}^{ext}) \tag{3.39}$$

(3.39) gives the relationship between the joint acceleration that the system produces and the joint forces. The inertia matrix $\boldsymbol{M}$ is what relates the joint accelerations $\ddot{\boldsymbol{q}}$ to the joint forces $\boldsymbol{\tau}$. If the system is at rest and there are no forces acting on it, $\boldsymbol{C_2} = \boldsymbol{0}$ and the equation of motion simplifies to $\boldsymbol{\tau} = \boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}}$.

The joint space bias matrix $\boldsymbol{C_2}$ represents the Coriolis and centrifugal forces, gravity terms and the effect of external forces, if any. The matrix thus represents the force that must be applied to the system to produce zero acceleration and is therefore called the joint space bias force. Similar to the Coriolis matrix, computing the joint space bias matrix can be seen as the inverse dynamics problem where $\ddot{\boldsymbol{q}} = \boldsymbol{0}$. Unlike the Coriolis matrix, the effect of gravity and external forces are present in the calculation of the joint space bias matrix.

CRBA calculates the inertia matrix efficiently by exploiting sparsity. By combining the algorithm with the equation of motion and the fact that $\boldsymbol{C_2} = \text{RNEA}(\textit{robot description}, \boldsymbol{q},$ CRBA can be used to obtain the forward dynamics by the following procedure:

1. Calculate $\boldsymbol{C_2}$ using RNEA with $\ddot{\boldsymbol{q}} = 0$.
2. Calculate $\boldsymbol{M}$ using CRBA.
3. Solve $\ddot{\boldsymbol{q}} = \boldsymbol{M}^{-1}(\boldsymbol{\tau} - \boldsymbol{C_2})$.

As can be observed, using CRBA to find the forward dynamics requires to calculate the inverse of the inertia matrix. This makes CRBA computationally expensive for robots with many bodies, and is the reason for the algorithm's $O(n^3)$ worst-case complexity.

### The Inertia Matrix

To understand the computation of the algorithm, one can observe the computation of the kinetic energy of a multi-body system:

$$T = \frac{1}{2}\dot{\boldsymbol{q}}^T \boldsymbol{M}\dot{\boldsymbol{q}} = \frac{1}{2}\sum_{i=1}^{n_B}\sum_{j=1}^{n_B}\dot{\boldsymbol{q}}_i^T \boldsymbol{M}_{ij}\dot{\boldsymbol{q}}_i \tag{3.40}$$

The physical understanding of $\boldsymbol{M}_{ij}$ is that it is the $n_i \times n_j$ submatrix relating the acceleration at joint $j$ to the force at joint $i$. If every joint in the kinematic tree has one degree of freedom, which is the case for u2c, $\boldsymbol{M}_{ij}$ is the $1 \times 1$ matrix on row $i$ and column $j$ of $\boldsymbol{M}$.

Further, the kinetic energy of the system can be viewed as the sum of the kinetic energy of the individual bodies:

$$T = \frac{1}{2}\sum_{i=1}^{n_B}\boldsymbol{v}_i^T \boldsymbol{I}_i \boldsymbol{v}_i \tag{3.41}$$

It can be assumed that all joint space matrices are expressed in the global reference frame (i.e. relative to the base body), which allows (3.41) to be rewritten as:

$$T = \frac{1}{2}\sum_{k=1}^{n_B}\sum_{i\in\kappa(i)}\sum_{j\in\kappa(i)}\dot{\boldsymbol{q}}_i^T \boldsymbol{S}_i^T \boldsymbol{I}_k \boldsymbol{S}_j \dot{\boldsymbol{q}}_j \tag{3.42}$$

In (3.42) the kinetic energy is expressed as a sum over all combinations where body $k$ is supported by both joint $i$ and $j$. Hence, (3.42) can be expressed as:

$$T = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k\in\nu(i)\cap\nu(j)}\dot{\boldsymbol{q}}_i^T \boldsymbol{S}_i^T \boldsymbol{I}_k \boldsymbol{S}_j \dot{\boldsymbol{q}}_j \tag{3.43}$$

And the intersection of $\nu(i)$ and $\nu(j)$ can be written as:

$$\nu(i) \cap \nu(j) = \begin{cases} \nu(i) & \text{if } i \in \nu(j) \\ \nu(j) & \text{if } j \in \nu(i) \\ \emptyset & \text{otherwise} \end{cases} \tag{3.44}$$

By comparing (3.43) to (3.40) the following can be observed:

$$\boldsymbol{M}_{ij} = \sum_{k\in\nu(i)\cap\nu(j)}\boldsymbol{S}_i^T \boldsymbol{I}_k \boldsymbol{S}_j \tag{3.45}$$

While (3.42) obtains the necessary quantities from $k$ up to the root of the graph, (3.43) and (3.45) performs the same computation by obtaining the quantities down the subtrees starting at the joint that is influenced by both $i$ and $j$.

Further, a new quantity $\boldsymbol{I}_i^c$ is introduced and represents the inertia of the subtree where body $i$ is the root of the subtree. The subtree is treated as a single composite rigid body, and this is where the algorithm gets its name. A summation of all the inertias gives the total inertia of the composite rigid body in the subtree:

$$\boldsymbol{I}_i^c = \sum_{j \in \nu(i)} \boldsymbol{I}_j \tag{3.46}$$

(3.46) can be expressed recursively as:

$$\boldsymbol{I}_i^c = \boldsymbol{I}_i + \sum_{j \in \mu(i)} \boldsymbol{I}_j^c \tag{3.47}$$

By utilizing (3.44) together with (3.45) and (3.46), the formula for the inertia matrix can be obtained:

$$\boldsymbol{M}_{ij} = \begin{cases} \boldsymbol{S}_i^T \boldsymbol{I}_i^c \boldsymbol{S}_j & \text{if } i \in \nu(j) \\ \boldsymbol{S}_i^T \boldsymbol{I}_j^c \boldsymbol{S}_j & \text{if } j \in \nu(i) \\ \boldsymbol{0} & \text{otherwise} \end{cases} \tag{3.48}$$

To summarize, (3.47) and (3.48) represent the basis of CRBA for a general kinematic tree. (3.47) gives the formula for obtaining the composite rigid body inertias, and these are further used to calculate the inertia matrix by using (3.48).

### Equations in Body Coordinates

The equations obtained above are given without reference to any particular coordinate system. To express the relevant equations in body coordinates, (3.47) and (3.48) are expanded to:

$$\boldsymbol{I}_i^c = \boldsymbol{I}_i + \sum_{j \in \mu(i)} {}^i\boldsymbol{X}_j^* \boldsymbol{I}_j {}^j\boldsymbol{X}_i \tag{3.49}$$

$$\boldsymbol{M}_{ij} = \begin{cases} \boldsymbol{S}_i^T \boldsymbol{I}_i^c {}^i\boldsymbol{X}_j \boldsymbol{S}_j & \text{if } i \in \nu(j) \\ \boldsymbol{S}_i^T {}^i\boldsymbol{X}_j^* \boldsymbol{I}_j^c \boldsymbol{S}_j & \text{if } j \in \nu(i) \\ \boldsymbol{0} & \text{otherwise} \end{cases} \tag{3.50}$$

To simplify the implementation of (3.50), a new quantity is defined: ${}^j\boldsymbol{F}_i = {}^i\boldsymbol{X}_j^* \boldsymbol{I}_j^c \boldsymbol{S}_j$. The calculation of $\boldsymbol{M}$ requires one instance of ${}^j\boldsymbol{F}_i$ for every $i,j$-pair satisfying $j \in \kappa(i)$, i.e. all joints $j$ where $j < i$ for a kinematic chain. Thus, they can be found recursively:

$${}^{\lambda_j}\boldsymbol{F}_i = {}^{\lambda_j}\boldsymbol{X}_i^{*\,j} \boldsymbol{F}_i \tag{3.51}$$

Given this new quantity, (3.50) can be expressed as:

$$\boldsymbol{M}_{ij} = \begin{cases} {}^{j}\boldsymbol{F}_i^T \boldsymbol{S}_j & \text{if } i \in \nu(j) \\ \boldsymbol{M}_{ij}^T & \text{if } j \in \nu(i) \\ \boldsymbol{0} & \text{otherwise} \end{cases} \tag{3.52}$$

The equations are implemented such that $i$ ranges from $n-1$ to 0 while $j$ develops separately from $i$ through $\lambda_i$ to $\lambda_{\lambda_i}$ until the base is reached. In the implementation, only kinematic chains are considered. In a kinematic chain, $\nu(j)$ represent every body $j > i$. Thus, $j \in \nu(i)$ implies that $i > j$. Another consequence of kinematic chains is that each body only has one connected child body. This body-child pair $(i,\, \mu_i)$ can also be seen as a parent-body pair $(\lambda_i,\, i)$. Hence, the implementation of the general equations (3.49) and (3.52) simplifies to:

$$\boldsymbol{I}_{\lambda_i}^c = \boldsymbol{I}_{\lambda_i} + {}^{\lambda_i}\boldsymbol{X}_i^* \boldsymbol{I}_i^{ci} \boldsymbol{X}_{\lambda_i} \tag{3.53}$$

$$\boldsymbol{M}_{ij} = \begin{cases} {}^{j}\boldsymbol{F}_i^T \boldsymbol{S}_j & \text{if } i > j \\ \boldsymbol{M}_{ij}^T & \text{if } j > i \\ \boldsymbol{0} & \text{otherwise} \end{cases} \tag{3.54}$$

Row $i$ and column $i$ of $\boldsymbol{M}$ can thus be filled by keeping $i$ constant and iterating over all $j$'s where $j < i$:

$j = i$
**while** $\lambda_j \neq 0$ **do**
  $\boldsymbol{F} = {}^{\lambda_j}\boldsymbol{X}_j^* \boldsymbol{I}_i^c \boldsymbol{S}_i$
  $j = \lambda_j$
  $\boldsymbol{M}_{ij} = \boldsymbol{F}^T \boldsymbol{S}_j$
  $\boldsymbol{M}_{ji} = \boldsymbol{M}_{ij}^T$
**end while**

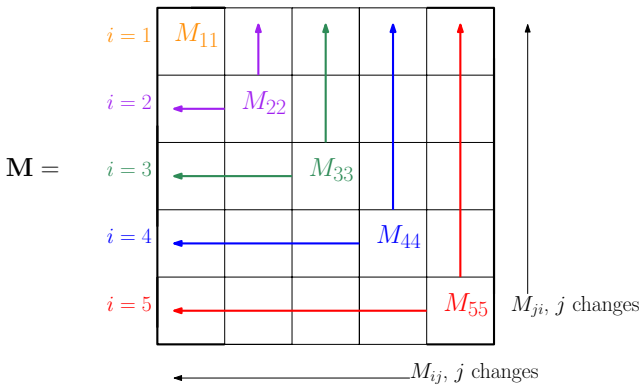



**Figure 3.5:** Illustration of the CRBA approach for computing $\boldsymbol{M}$.

Exploiting this while at the same time iterating $i$ in a backward pass from $n-1$ to 0, all the rows and columns of $\boldsymbol{M}$ are found, starting outermost at $\boldsymbol{M}_{nn}$ and working inwards to $\boldsymbol{M}_{11}$. This is illustrated in Figure 3.5 for a rigid 5-bodied system. The pseudocode implementation of CRBA is given in Algorithm 4.

**Algorithm 4** Composite Rigid Body Algorithm

**Input:** $\boldsymbol{X}$, $\boldsymbol{I}$, $\boldsymbol{S}$
**Output:** $\boldsymbol{M}$

$\boldsymbol{M} = \boldsymbol{0}$
**for** $i = 1$ **to** $n$ **do**
    $\boldsymbol{I}_i^c = \boldsymbol{I}_i$
**end for**
**for** $i = n - 1$ **to** $0$ **do**
    **if** $\lambda_i \neq 0$ **then**
        $\boldsymbol{I}_{\lambda_i}^c = \boldsymbol{I}_{\lambda_i}^c + {}^{\lambda_i}\boldsymbol{X}_i^* \, \boldsymbol{I}_i^{c\lambda_i i} \boldsymbol{X}_{\lambda_i}$
        $\boldsymbol{M}_{ii} = \boldsymbol{S}_i^T \boldsymbol{I}_i^c \boldsymbol{S}_i$
        $j = i$
    **end if**
    **while** $\lambda_j \neq 0$ **do**
        $\boldsymbol{F} = {}^{\lambda_j}\boldsymbol{X}_j^* \boldsymbol{I}_i^c \boldsymbol{S}_i$
        $j = \lambda_j$
        $\boldsymbol{M}_{ij} = \boldsymbol{F}^T \boldsymbol{S}_j$
        $\boldsymbol{M}_{ji} = \boldsymbol{M}_{ij}^T$
    **end while**
**end for**

In the implementation, there are two preparatory steps. The first step is to initialize the inertia matrix, and the second step is to set $\boldsymbol{I}_i^c = \boldsymbol{I}_i$. Then the values of $\boldsymbol{M}$ are obtained as explained above.

Having obtained $\boldsymbol{M}$ with CRBA and $\boldsymbol{C_2}$ with RNEA with $\ddot{\boldsymbol{q}} = 0$, the joint accelerations are easily obtainable for a given $\boldsymbol{\tau}$ by solving $\ddot{\boldsymbol{q}} = \boldsymbol{M}^{-1}(\boldsymbol{\tau} - \boldsymbol{C_2})$. It is worth mentioning that this is a very computationally expensive procedure for robots with a high number of DOF, as this implies that $\boldsymbol{M}$ is a large matrix. This is further demonstrated in Chapter 5.

## 3.4 Resulting Functionality and Usage

In the above sections, it is explained how u2c is structured, and presented how the robot description is loaded from a URDF with the help of the open-source library urdf_parser_py. Last, it was explained how the rigid body dynamics algorithms are used to retrieve the robot's dynamics. The result is that the user of the library can obtain the following, assuming there is a URDF description of the robot:

1. Inverse dynamics using RNEA.
2. Coriolis matrix using RNEA with $\ddot{\boldsymbol{q}} = 0$.
3. Gravitational term using RNEA with $\dot{\boldsymbol{q}} = 0$ and $\ddot{\boldsymbol{q}} = 0$.
4. Forward dynamics using ABA.
5. Forward dynamics using CRBA.
6. Inertia matrix using CRBA.
7. Forward kinematics.

The solution is returned as a just-in-time C-code generated CasADi `SX` function. It can thus be used for numerical evaluation and symbolic evaluation by providing it symbols for the joint state variables, or one can use the CasADi framework to obtain the derivatives of the functions.

For the reader to gain insight into the easy use of the library, an example of usage is given:

```
import urdf2casadi.urdfparser as u2c
robot = u2c.URDFparser()
robot.from_file("./examples/urdf/ur5.urdf")

gravity = [0, 0, -9.81]
id = robot.get_inverse_dynamics_rnea(base_link, tool0, gravity)
id_forearm = robot.get_inverse_dynamics_rnea(base_link, forearm_link, gravity)
fd = robot.get_forward_dynamics_aba(base_link, tool0)
M = robot.get_inertia_matrix_crba(base_link, tool0)
```

In the above example, it is shown how one can find CasADi functions for the inverse and forward dynamics, and the inertia matrix of a UR5 using u2c. The gravity input is optional, and one can also add external forces as an optional input parameter.

As one can see, structuring the library around one class provides easy use. To obtain the kinematics and dynamics of a robot, the user has to import the library, declare a class instance, and load the robot model from a URDF to that instance. After these three preparatory steps, the kinematics and dynamics of the robot can be provided by the library.

An important notice is that the user can obtain the kinematics and dynamics from any root to tip of the robot with one class instance, i.e. by only loading

the URDF once. This is illustrated in the above example, where the inverse dynamics are first found from `base_link` to `tool0`, and then from `base_link` to `forearm_link`. Hence, the `URDFparser` can be seen as a kinematics and dynamics lookup table for the robot.

After obtaining the dynamics functions, they can be numerically evaluated:

```
id_num = id(q_num, qdot_num, qddot_num)
fd_num = fd(q_num, qdot_num, tau_num)
M_num = M(q_num)
```

where `q_num, qdot_num, qddot_num,` and `tau_num` are numerical vectors with length corresponding with the number of DOF of the robot, and their numerical values are within the joint limits.

Having the dynamics functions returned by u2c, the dynamics derivatives are easily obtainable within a few code lines, using the CasADi framework. This is further explained in Chapter 6, and it is also shown in u2c's user example, together with the rest of the functionality provided by u2c. The user example is found in the following directory:

```
urdf2casadi
└── examples
    └── user
        └── ur5_example
```

The user example presents all the functionality provided by u2c, using a UR5 as the example robot, and is a part of the documentation associated with u2c, which is further explained in Chapter 7.

# CHAPTER 4

# Numerical Results

## 4.1 Numerical Tests

The algorithm implementation of u2c is verified by comparing the numerical evaluation of the returned CasADi functions against the well-established numerical libraries: KDL, RBDL, and PyBullet. As briefly mentioned in Chapter 1, these libraries are developed for distinct purposes and hence provide distinct functionality.

The aforementioned libraries are numerical, as opposed to u2c. This restricts the user to the built-in functions rather than being able to take as many partial derivatives as necessary for the controller formulation. However, it is chosen to compare u2c against these libraries for several reasons. KDL is a well-established library in the ROS community, PyBullet is widely used within machine learning, and RBDL is, similar to u2c, implemented based on the Featherstone (2008). Further, these libraries all provide Python bindings and loading URDFs, as u2c. Table 4.1 summarizes the dynamics provided by each library.

**Table 4.1:** Dynamics functionality provided by the libraries.

| | u2c | KDL | RBDL | PyBullet |
|---|---|---|---|---|
| $\boldsymbol{G}$ | ✓ | ✓ | ✓ | ✓ |
| $\boldsymbol{C}$ | ✓ | ✓ | ✓ | ✓ |
| ID | ✓ | | ✓ | ✓ |
| $\boldsymbol{M}$ | ✓ | ✓ | ✓ | ✓ |
| FD | ✓ | | ✓ | |

The numerical results are obtained by generating 1000 samples of configurations, velocities, and accelerations or torques, uniformly distributed within the joint limits. The numerical tests are conducted on a variety of robots: a 2-DOF pendulum, a 6-DOF UR5, a 7-DOF KUKA-LWR, and a 16-DOF snake robot. These robots are chosen for the tests as they represent different kinematic constructions and have various number of DOF, i.e. qualities that may affect the numerical results.

For further implementational details, the numerical test scripts can be found in the library path given by the directory tree:

```
urdf2casadi
└── examples
    └── numerical
        ├── pendulum
        ├── ur5
        ├── kuka
        └── snake
```

## 4.2 2-DOF pendulum

**Table 4.2:** Numerical differences between libraries for the 2-DOF pendulum for 1000 random samples.

| | KDL/u2c | RBDL/u2c | PyBullet/u2c |
|---|---|---|---|
| $\boldsymbol{G}$ $[N]$ | $1.30 \cdot 10^{-12}$ | $4.14 \cdot 10^{-11}$ | $5.91 \cdot 10^{-04}$ |
| $\boldsymbol{C}$ $[N]$ | $1.18 \cdot 10^{-12}$ | $7.37 \cdot 10^{-11}$ | $6.60 \cdot 10^{-04}$ |
| ID $[Nm]$ | | $7.42 \cdot 10^{-11}$ | $1.24 \cdot 10^{-03}$ |
| $\boldsymbol{M}$ $[kgm^2]$ | $6.50 \cdot 10^{-13}$ | $4.08 \cdot 10^{-12}$ | $2.77 \cdot 10^{-04}$ |
| FD $[m/s^2]$ | | $1.58 \cdot 10^{-11}$ | |

Table 4.2 shows the numerical differences between the libraries for the 2-DOF pendulum. The results show that u2c and KDL have the most similar results with a difference of $10^{-12}$ for all the dynamic parameters, indicating a difference of $10^{-15}$ per sample. Further, u2c and RBDL also give very similar results, with the difference being one order of magnitude higher than KDL and u2c. This minor difference is probably due to KDL and u2c having more similar data types than RBDL and u2c. Last, it is seen that PyBullet and u2c have a relatively bigger difference, with an order of magnitude of $-4$ and $-3$. This difference is further discussed later on.

It can be seen that common for all the library comparisons, is that the numerical difference tends to a minor increase from $\boldsymbol{G}$, to $\boldsymbol{C}$, and up to ID. This is most likely due to the additional variables introduced in the calculation of RNEA, thus leading to an extended expression to numerically evaluate.

## 4.3 6-DOF UR5

**Table 4.3:** Numerical differences between libraries for the 6-DOF UR5 for 1000 random samples.

| | KDL/u2c | RBDL/u2c | PyBullet/u2c |
|---|---|---|---|
| $\boldsymbol{G}$ $[N]$ | $4.42 \cdot 10^{-12}$ | $1.96 \cdot 10^{-07}$ | $1.83 \cdot 10^{-03}$ |
| $\boldsymbol{C}$ $[N]$ | $1.03 \cdot 10^{-11}$ | $4.30 \cdot 10^{-07}$ | $3.12 \cdot 10^{-03}$ |
| ID $[Nm]$ | | $4.41 \cdot 10^{-07}$ | $4.12 \cdot 10^{-03}$ |
| $\boldsymbol{M}$ $[kgm^2]$ | $1.40 \cdot 10^{-12}$ | $1.46 \cdot 10^{-08}$ | $2.32 \cdot 10^{-03}$ |
| FD $[m/s^2]$ | | $7.88 \cdot 10^{-07}$ | |

The numerical differences for the 6-DOF UR5, given in Table 4.3, yield similar results as for the 2-DOF pendulum: u2c and KDL have the most similar results, and the numerical results of PyBullet deviates from u2c, KDL, and RBDL. Further, it can be seen that the numerical differences between u2c and RBDL are four orders of magnitude higher, while the differences between u2c and PyBullet are one order of magnitude higher. Whether this is a matter of the robot's kinematics, or the increase in DOF, is hard to tell at this point.

## 4.4 7-DOF KUKA-LWR

**Table 4.4:** Numerical differences between libraries for the 7-DOF KUKA-LWR for 1000 random samples.

| | KDL/u2c | RBDL/u2c | PyBullet/u2c |
|---|---|---|---|
| $\boldsymbol{G}$ $[N]$ | $2.92 \cdot 10^{-12}$ | $4.97 \cdot 10^{-12}$ | $3.87 \cdot 10^{-04}$ |
| $\boldsymbol{C}$ $[N]$ | $3.81 \cdot 10^{-12}$ | $1.36 \cdot 10^{-11}$ | $2.08 \cdot 10^{-04}$ |
| ID $[Nm]$ | | $1.18 \cdot 10^{-11}$ | $5.48 \cdot 10^{-04}$ |
| $\boldsymbol{M}$ $[kgm^2]$ | $2.07 \cdot 10^{-12}$ | $1.54 \cdot 10^{-08}$ | $1.71 \cdot 10^{-04}$ |
| FD $[m/s^2]$ | | $2.22 \cdot 10^{-10}$ | |

Table 4.4 shows that the numerical differences for the 7-DOF KUKA-LWR are more similar to the numerical differences of 2-DOF pendulum than those for the 6-DOF UR5. This indicates that the bigger numerical differences obtained for the UR5 are a result of the robot's kinematics rather than the number of DOF of the robot.

## 4.5 16-DOF snake

**Table 4.5:** Numerical differences between libraries for the 16-DOF snake for 1000 random samples.

| | KDL/u2c | RBDL/u2c | PyBullet/u2c |
|---|---|---|---|
| $\boldsymbol{G}$ $(N)$ | $2.52 \cdot 10^{-12}$ | 0.0 | $2.37 \cdot 10^{-11}$ |
| $\boldsymbol{C}$ $(N)$ | $2.22 \cdot 10^{-10}$ | $1.73 \cdot 10^{-11}$ | $1.61 \cdot 10^{-10}$ |
| ID $(Nm)$ | | $1.29 \cdot 10^{-10}$ | $1.96 \cdot 10^{-10}$ |
| $\boldsymbol{M}$ $(kgm^2)$ | $6.46 \cdot 10^{-11}$ | $2.38 \cdot 10^{-11}$ | $5.33 \cdot 10^{-11}$ |
| FD $(m/s^2)$ | | $1.29 \cdot 10^{-10}$ | |

The numerical results for the 16-DOF snake, given in Table 4.5, show that the difference between u2c and KDL, and u2c and RBDL remain similar to the results for the other robots. An unexpected result is that the numerical differences between PyBullet and the other libraries are more similar for the 16-DOF snake than for the other robots, despite the increase in DOF. Previously, the numerical differences between PyBullet and the other libraries were of order of magnitude $-4$, and $-3$, while for the 16-DOF snake it is of order of magnitude $-10$ and $-11$, depending on the dynamics. This is a significantly more accurate result, and substantiates that the numerical differences are not affected by the number of DOF.

## 4.6 Discussion

From the results given, it can be concluded that u2c, KDL, and RBDL have sufficiently similar results, where u2c and KDL most likely have more similar data types, leading to a smaller difference in results.

It can further be concluded that the difference in results seems to be independent of the number of DOF. This is substantiated by the fact that PyBullet gives a much more accurate result for the 16-DOF snake than for the other robots, which have fewer DOF.

After some experimentation with the URDFs, it was discovered that the differences in the results are a matter of integers versus decimals. While the pendulum, UR5, and the KUKA-LWR use decimal numbers to describe the robot's kinematics, the 16-DOF snake URDF consists mostly of integers. It was discovered that by changing the description of the other robots to use integers as well, the same accuracy as for the snake was achieved.

It was thus speculated whether the inaccuracy of PyBullet was a matter of single versus double precision floats, or a floating point cancellation issue. The developers of PyBullet was informed, and it was found a conversion from a double precision float to a single precision float in their URDF parser. The bug is now fixed by the developers of PyBullet, and the change is submitted to their open-source library.

# CHAPTER 5

# Timing Results

Rigid body dynamics are often used in real-time tasks within robotics research. To achieve the best possible solution, either if it is for optimal control, trajectory optimization, estimation, or simulation purposes, the dynamics are needed to be updated as often as possible as the input parameters change rapidly. The update frequency of the dynamic parameters depends on how fast the dynamics can be evaluated. Thus, the efficiency of the numerical evaluation of the CasADi functions returned by u2c is a critical aspect of this project.

A thorough, two-folded timing investigation is conducted. The first part focuses on the timing results performed on three different robots: the 2-DOF pendulum, the 6-DOF UR5, and the 16-DOF snake robot. In the second part, experimental URDFs are used to analyze how the evaluation times evolve when increasing the number of DOF from 1 to 60. The timing investigation attempts to find what affects the evaluation times. The results are compared against the three other well-established libraries: KDL, RBDL, and PyBullet. These libraries are, as previously mentioned, numerical and thus expected to be more efficient than the symbolic functions returned by u2c.

It is explained how the timing investigation is performed, followed by a presentation and discussion of the results. In the matter of displaying the results graphically, colors that maximize the distance of the color map are chosen. The timing scripts used to obtain these results can be viewed in the `timing`-folder of u2c:

```
urdf2casadi
└── examples
    └── timing
        ├── robot_comparison
        └── 60dof
```

## 5.1 Timing Tests

The timing tests are performed on a Ubuntu 16.04, 3.5 GHz x-12 Intel Xeon CPU processor in a Python 2.7 environment, and the -OFast compiler flag is used for optimization of the generated C-code. The Python module *timeit* is used to measure the evaluation times of the libraries. timeit is a module that provides the ability to measure the execution time of a code snippet while avoiding common traps for execution time measuring. timeit automatically repeats the execution of the code snippet many times while disabling the garbage collector, and picks the most accurate timer for the given OS. By repeating the code a large number of times, timeit eliminates the influence of other tasks on the machine, e.g disk flushing, and OS scheduling, and thus obtains an accurate timing result.

There are mainly three types of time when measuring code execution: wall-clock time, CPU time, and system time. Wall-clock time measures the elapsed time of the code execution, while CPU time measures how much time the CPU spends on the execution. The CPU time is often shorter than wall-clock time for single core computers as the CPU may be executing other instructions at the same time. On multi-core computers, wall-clock time may be shorter than CPU time as the code is executed in parallel. The third type of time is called system time and represents the time spent in the kernel, and is usually time spent servicing system calls.

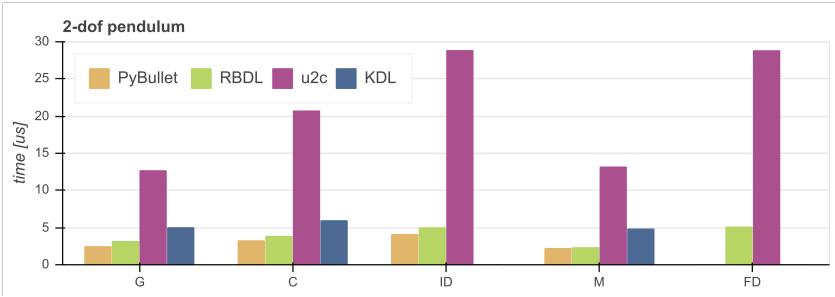
The most accurate timer provided by Python for a Ubuntu OS measures wall-clock time. A common trap using wall-clock time is that other processes may interfere, resulting in the wall-clock time including the time spent waiting for CPU capacity. To avoid this, the timeit module is used to repeat the timing of 1000 numerical evaluations 100 times, and then find the median evaluation time of the median timing batch. To minimize CPU interference it is made sure that no other processes are running on the computer while running the timing tests.

## 5.2 Robot Comparison

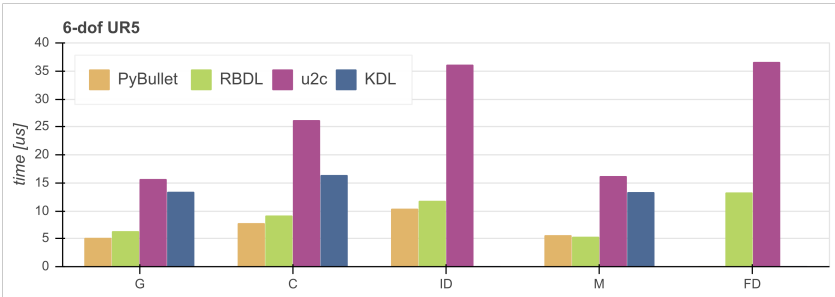
In this section, evaluation times of the dynamics for three robots are evaluated: the 2-DOF pendulum, the 6-DOF UR5, and the 16-DOF snake robot. These robots are chosen so that one can evaluate the timing performance of u2c for robots of various DOF, and thus gain insight into how the number of DOF affects the timing performance.

The URDF description of the 2-DOF pendulum is a pan-tilt pendulum used in ROS tutorials and the UR5 URDF description is taken from ROS's original description of a UR5 robot. The description of the snake robot is constructed from the parameters describing the Eelume snake robot presented in Borlaug et al. (2018).

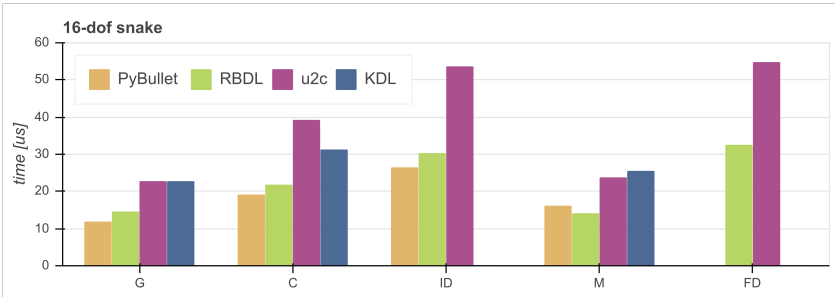
Hereafter, the timing results of these robots are evaluated, followed by an investigation of what mainly affects the evaluation times of u2c.




**(a)** Median evaluation times for pendulum.




**(b)** Median evaluation times for UR5.




**(c)** Median evaluation times for snake.

**Figure 5.1:** Median evaluation times for the dynamics for PyBullet, RBDL, u2c, and KDL.

From Figure 5.1 one can observe that u2c uses overall a longer evaluation time than PyBullet, RBDL, and KDL. This is a natural consequence of u2c generated functions supporting symbolic CasADi data types, and thus, some overhead is expected. Yet, the evaluation times of u2c are at most one order of magnitude higher than the numerical libraries, which is the case for the 2-DOF pendulum, observed in Figure 5.1a. For the UR5, u2c's evaluation times are the same order of magnitude as the evaluation times of the numerical libraries, except for $\boldsymbol{G}$, $\boldsymbol{C}$, and $\boldsymbol{M}$ for PyBullet and RBDL, which is one order of magnitude faster. Lastly, the timing results for the snake yield evaluation times of the same order of magnitude

for u2c and the numerical libraries. It is also seen that u2c excels over KDL for evaluation of the inertia matrix at this point.

The above thus implies that u2c becomes relatively faster compared to the numerical libraries for an increasing number of DOF. This indicates that the overhead related to CasADi seems to be rather constant with regard to number of DOF. It can further be observed from Figure 5.1 that it is mainly the dynamics functions that consist of only one input variable, i.e. $\boldsymbol{G}$ and $\boldsymbol{M}$, that are becoming relatively faster, while the dynamics functions that consist of three symbolic variables, i.e. ID and FD, have longer evaluation times, compared to the numerical libraries. This may indicate that the overhead related to CasADi is due to an overhead related to the CasADi symbolic data types. However, one must also consider the fact that dynamics expressions consisting of several input variables are more likely to encompass a higher number of operations. This may also affect the evaluation times as this implies an expression graph of a higher amount of nodes for CasADi to evaluate. In the following, it is investigated how the number of operations, and the number of input variables, affect the timing efficiency of u2c.

### 5.2.1 Impact of Number of Operations

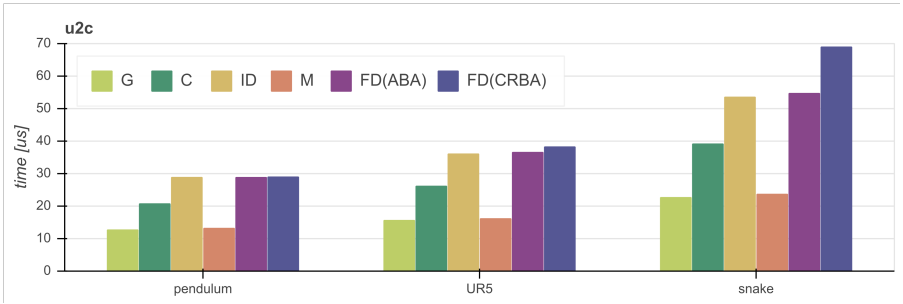



**Figure 5.2:** Median evaluation times for the dynamics for u2c.

**Table 5.1:** Median evaluation times for u2c.

| | pendulum | ur5 | snake |
|---|---|---|---|
| G | 12.64 µs | 15.46 µs | 22.01 µs |
| C | 20.68 µs | 26.02 µs | 37.45 µs |
| ID | 28.81 µs | 36.02 µs | 54.22 µs |
| M | 13.14 µs | 16.09 µs | 22.67 µs |
| FD (crba) | 28.93 µs | 38.19 µs | 68.92 µs |
| FD (aba) | 28.78 µs | 36.50 µs | 54.92 µs |

**Table 5.2:** Number of operations for the dynamics expressions.

| | pendulum | ur5 | snake |
|---|---|---|---|
| G | 18 | 199 | 224 |
| C | 79 | 648 | 572 |
| ID | 110 | 696 | 710 |
| M | 34 | 872 | 1217 |
| FD (crba) | 132 | 2354 | 14786 |
| FD (aba) | 142 | 1948 | 2272 |

To evaluate how the number of operations affects the evaluation times of u2c, one can observe how the evaluation times of ID and FD evolve for the different robots. As both the ID function and the FD functions encompass three input variables, but a different number of operations, their difference in evaluation time should represent the effect of the difference in the number of operations.

By looking at the evaluation times given in Figure 5.2 and Table 5.1, in the light of the information provided by Table 5.2, one can observe that the evaluation times of u2c are not heavily affected by an increase in the number of operations. To illustrate, one can observe that the evaluation times for ID and FD for the 2-DOF pendulum are approximately the same while the number of operations varies from 110 to 142. For the 6-DOF UR5, the number of operations varies more. ID consists of 696 operations, FD (ABA) of 1948 operations, and FD (CRBA) of 2354 operations. Still, the evaluation times remain approximately the same. One can observe that the evaluation time of FD (CRBA) is two microseconds larger than the evaluation time of ID and FD (ABA). However, for such a small increase in evaluation time, it is hard to tell whether this increase is due to the additional operations, normal variability in the evaluation time, or interfering processes during the timing tests.

For the 16-DOF snake robot, the gap in the number of operations between ID and FD (ABA) to FD(CRBA) is higher due to CRBA's $O(n^3)$ worst-case complexity. While there is an increase of 1562 operations from ID to FD (ABA), there is an increase of 12514 operations from FD (ABA) to FD (CRBA). The reason for this high increase in the number of operations for FD using CRBA is that it must invert a $16 \times 16$ inertia matrix. From Figure 5.2 and Table 5.1 one can observe how the increase of 1562 operations does not remarkably affect the evaluation time, as the evaluation times of ID and FD(ABA) for the 16-DOF snake are approximately the same. But the increase of 12514, is noticeable on the efficiency. The evaluation time of FD with CRBA is 14 microseconds larger than the evaluation time of FD with ABA for the 16-DOF snake.

Thus, one can conclude that the overhead related to an increase in the number of operations is present, but is negligible in cases where the robot does not have a high number of DOF. It should also be mentioned that in the case where the increase in the number of operations does affect the evaluation time remarkably, i.e. for FD using CRBA, the user is recommended to use FD obtained with ABA

instead.

### 5.2.2 Impact of Number of Input Variables

To investigate how additional symbolic variables affect the evaluation times of the CasADi functions, one can exploit the fact that $\boldsymbol{G}$, $\boldsymbol{C}$, and ID are retrieved using the same algorithm, namely RNEA, but with a different number of input variables. As previously mentioned, their relation is given by:

$$\begin{aligned} \text{ID} &= \text{RNEA}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}, \boldsymbol{f}^{ext}) \\ \boldsymbol{C} &= \text{RNEA}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{0}, \boldsymbol{0}) \\ \boldsymbol{G} &= \text{RNEA}(\boldsymbol{q}, \boldsymbol{0}, \boldsymbol{0}, \boldsymbol{0}) \end{aligned}$$

where $\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}$ are symbolic `SX` vectors of length consistent with the robot's number of DOF.

By evaluating the inverse dynamics with $\boldsymbol{f}^{ext} = 0$, the increase in evaluation time from $\boldsymbol{G}$, to $\boldsymbol{C}$, to ID shows the impact of additional input variables. One should bare in mind that the difference in the number of operations between $\boldsymbol{G}$, $\boldsymbol{C}$, and ID, for the pendulum, UR5, and snake are at most 600 operations, which the above concluded does not affect the evaluation time noticeably. Thus, it is reasonable to assume that the increase in evaluation time from $\boldsymbol{G}$, to $\boldsymbol{C}$, and up to ID is due to the additional input variables required. For comparison, it is also seen how additional input variables affect KDL, RBDL, and PyBullet.

**Table 5.3:** Median evaluation times for $\boldsymbol{G}$, $\boldsymbol{C}$, and ID, for the pendulum, UR5, and snake.

| | $\boldsymbol{G}$ | $\boldsymbol{C}$ | ID |
|---|---|---|---|
| pendulum | 12.64 µs | 20.68 µs | 28.81 µs |
| UR5 | 15.46 µs | 26.02 µs | 36.02 µs |
| snake | 22.01 µs | 37.45 µs | 54.22 µs |

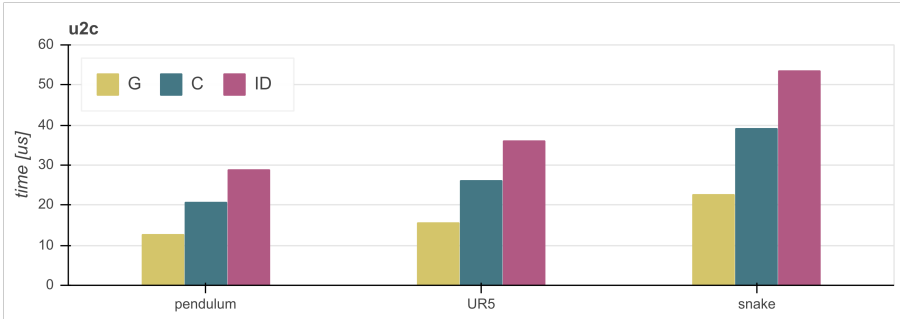
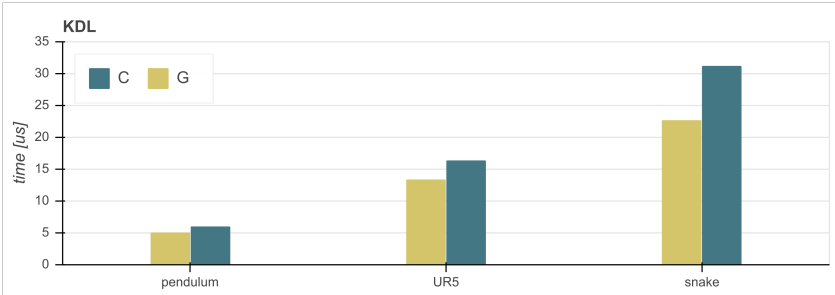


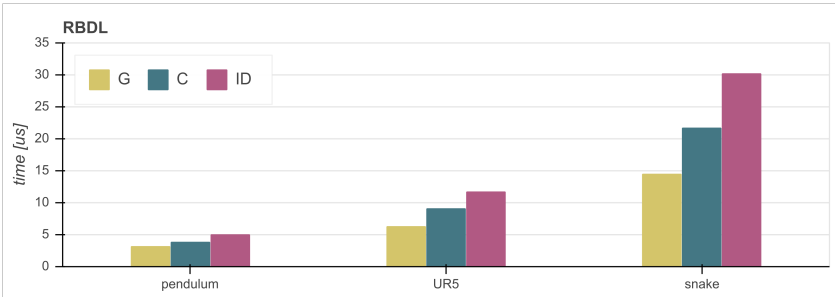

**Figure 5.3:** Median evaluation times for $\boldsymbol{G}$, $\boldsymbol{C}$, and ID for the pendulum, UR5, and snake.

Figure 5.3 and Table 5.3 show that the increase in evaluation times from $\boldsymbol{G}$, to $\boldsymbol{C}$, to ID seem to increase with a constant factor linear to the number of symbolic variables. One can observe that this constant factor increases with the number of DOF, which is reasonable as it is thus required vector variables with a higher number of elements.

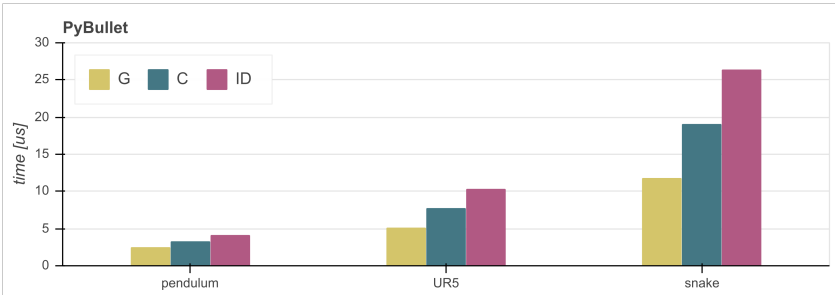
For the 2-DOF pendulum, the evaluation time increases with 8 microseconds when going from one input variable, i.e. $\boldsymbol{G}$, to two input variables, i.e. $\boldsymbol{C}$, and another 8 microseconds up to three input variables, i.e. ID. For the 6-DOF UR5, this increase is approximately 11 microseconds, and 16 microseconds for the 16-DOF snake. Hence, it is reasonable to assume that the increase in evaluation time from $\boldsymbol{G}$ to $\boldsymbol{C}$, to ID, is due to overhead related to the CasADi data types, and the size of the overhead seems to have one constant factor and another factor increasing with the number of DOF. This can be assumed as, although the snake has eight times as many DOF as the pendulum, the constant factor, which the overhead increases with when adding a new input variable, is only twice as big.




**(a)** Median evaluation times for KDL.




**(b)** Median evaluation times for RBDL.




**(c)** Median evaluation times for PyBullet.

**Figure 5.4:** Median evaluation times for $\boldsymbol{G}$, $\boldsymbol{C}$, and ID for the pendulum, UR5, and snake.

By comparing the evaluation times of the numerical libraries given in Figure 5.4, to the evaluation times of u2c, one can observe that the numerical libraries are in general more affected by an increase in DOF. When it comes to the increase in evaluation times from $\boldsymbol{G}$, to $\boldsymbol{C}$, to ID due to additional variables, the numerical libraries seem less affected. One can also observe that the increase from $\boldsymbol{G}$ to $\boldsymbol{C}$, to ID, increases when the number of DOF increases. Whether this increase is due to longer expressions to evaluate, or due to a slight overhead related to the data types, is beyond the author's knowledge of the libraries. Most likely, it is a combination of these. It is reasonable to assume that the overhead related to numerical data

types is not too significant as the evaluation times for the 2-DOF pendulum are remarkably small.

### 5.2.3 Conclusion

Based on the above, one can conclude that the evaluation times of the dynamics functions returned by u2c seem to be most affected by the overhead related to the number of input variables. Further, it can be concluded that this overhead is most present for robots of few DOF, as the results indicate that the overhead is two-folded: one part that seems constant with the addition of another input variable, and another that seems to increase with the number of DOF, as this affects the number of elements.

It is also shown that moderate changes in the number of operations, which is the case for the dynamics obtained with $O(n)$-complexity algorithms for the robots used in the above, do not remarkably affect the evaluation times. The results indicate that more extensive changes in the number of operations, e.g. FD using CRBA for the 16-DOF snake, can affect the evaluation times.

To conclude, an increase in the number of operations may affect the evaluation times of u2c if the increase is significant enough. Further, the increase in evaluation times due to an increasing number of operations seems to be more significant for the numerical libraries as u2c gets relatively more efficient when the DOF increase. It can also be concluded that the overhead associated with the CasADi variables seems to even out as the DOF increase. This was shown as when the number of DOF is eight times larger, i.e. when going from the 2-DOF pendulum to the 16-DOF snake, the overhead associated with a new input variable only doubled.

Because of these findings, further investigations have been performed on more extensive increases in the number of DOF.

## 5.3 60 DOF Analysis

As the above results imply that evaluation times of u2c get relatively faster compared to the numerical libraries when increasing the number of DOF, an interesting test case is to see how the evaluation times evolve for more extensive increases in the number of DOF. For this investigation, 60 experimental URDFs have been generated, where the first URDF represents a robot of 1 DOF and the last represents a robot of 60 DOF. The parameter values of the URDFs are based on the parameter values of the UR5 used previously, such that the 60-DOF robot is the equivalent of connecting ten UR5 robots as one giant robotic arm.

### 5.3.1 Gravity

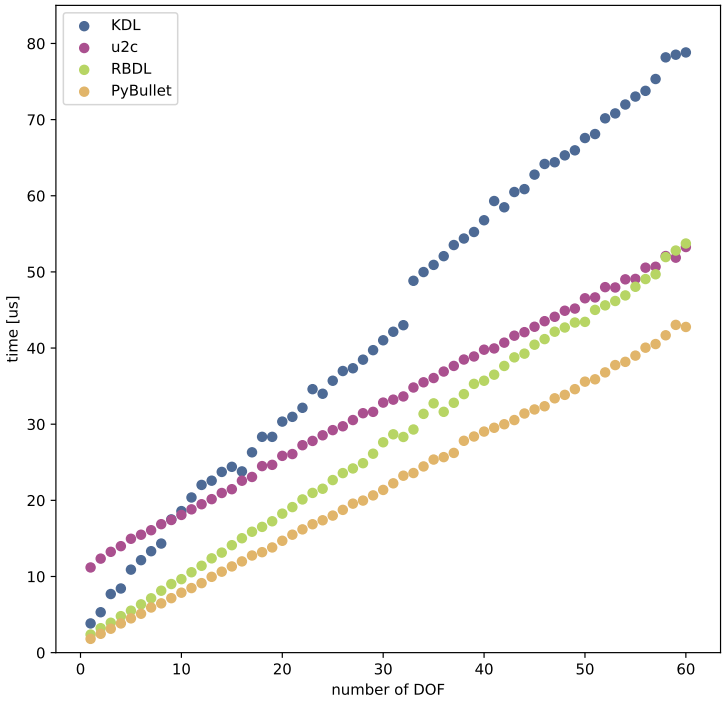



**Figure 5.5:** Median evaluation times for gravity from 1 to 60 DOF.

From Figure 5.5 one can observe that u2c starts out being the least efficient library. As the number of DOF increases, u2c excels over KDL and eventually RBDL. This

**Table 5.4:** Number of operations for $\boldsymbol{G}$ for an increasing number of DOF.

| number of DOF: | 2 | 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|---|---|
| number of operations | 26 | 429 | 883 | 1351 | 1829 | 2246 | 2726 |

implies that the overhead related to a high number of DOF for the numerical libraries is more significant than the overhead related to a high number of DOF for u2c. Further, Figure 5.5 shows that KDL and RBDL are most affected by an increase in DOF as they have the highest slope. u2c has the lowest slope but has a higher offset as a result of the overhead previously described. PyBullet has a slope that is comparable to u2c, but slightly higher. u2c is thus the library with the least variability in evaluation time, meaning that it is the library that is least affected by an increase in DOF. This is due to CasADi providing operations with minimal overhead, in combination with $\boldsymbol{G}$ only encompassing one input variable.

The above finding represents the positive and negative aspects of using CasADi's symbolic `SX` variables, explained in section 2.1.2. On the downside, the use of `SX` variables comes with an overhead. However, `SX` variables are made to perform operations with minimal overhead, and thus, the CasADi functions returned by u2c are not profoundly affected by the increase in the number of operations.

### 5.3.2 Coriolis

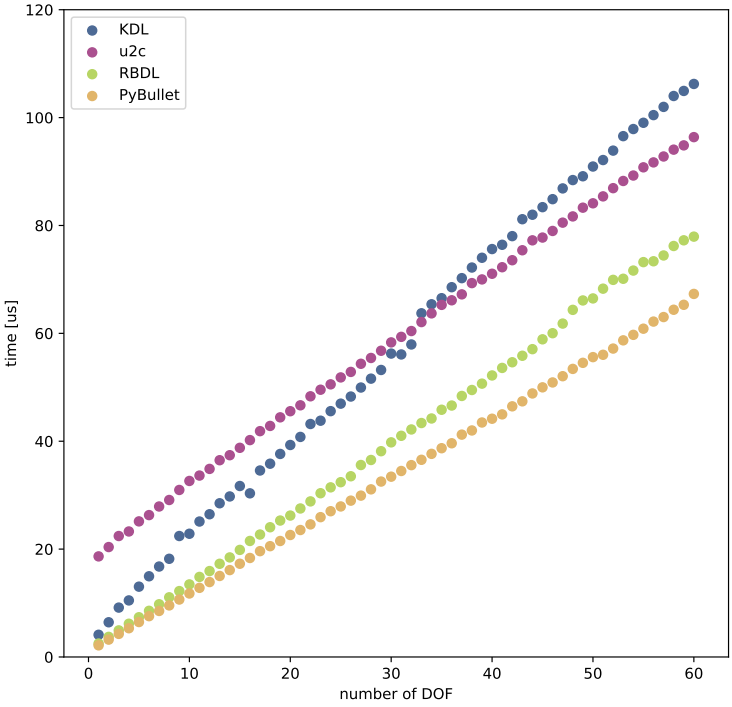



**Figure 5.6:** Median evaluation times for Coriolis from 1 to 60 DOF.

Similar to $\boldsymbol{G}$, Figure 5.6 shows that u2c starts out as the least efficient library, and becomes relatively faster as the number of DOF increases. Yet, while KDL, RBDL, and PyBullet seem to maintain their relative timing compared to those of G, one can observe that u2c has become relatively slower for the evaluation times of $\boldsymbol{C}$, relative to $\boldsymbol{G}$. By comparing Table 5.5 to Table 5.4, one can observe that $\boldsymbol{C}$ contains more operations than $\boldsymbol{G}$, in addition to $\boldsymbol{C}$ encompassing two input variables. Section 5.2.2 illustrated that additional input variables make the function more vulnerable to an increase in DOF. Also, it was found that the increase in overhead is reinforced by an increase in the number of operations. This is substantiated by noting that the slope of u2c in Figure 5.6 is higher than the slope of u2c in Figure 5.5. One can also observe that u2c excels over KDL for a higher number of DOF than for gravity, and remains less effective than RBDL. Evaluation times for u2c still remain the same order of magnitude as the other libraries, which can be considered sufficient.

**Table 5.5:** Number of operations for $\boldsymbol{C}$ for an increasing number of DOF.

| number of DOF: | 2 | 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|---|---|
| number of operations: | 103 | 1309 | 2766 | 4258 | 5784 | 7157 | 8683 |

### 5.3.3 Inverse Dynamics

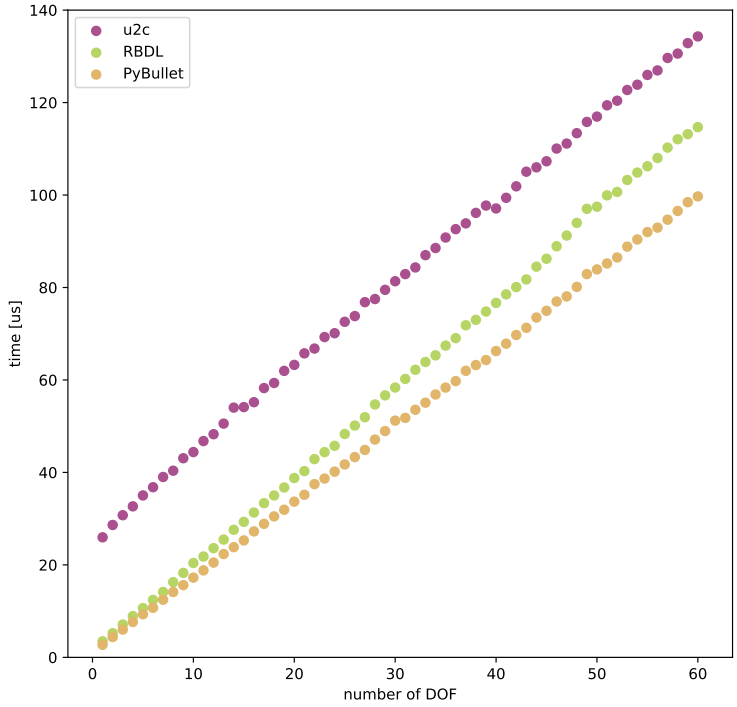



**Figure 5.7:** Median evaluation times for ID from 1 to 60 DOF.

**Table 5.6:** Number of operations for ID for an increasing number of DOF.

| number of DOF: | 2 | 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|---|---|
| number of operations: | 134 | 1378 | 2854 | 4366 | 5912 | 7305 | 8851 |

By evaluating Table 5.6, one can observe that the number of operations for ID is approximately the same as for $\boldsymbol{C}$. The number of operations for ID is just slightly higher, with a maximum of a couple of hundreds operations more. The potential increase in evaluation times for u2c can thus be assumed to be due to the additional input variable required for ID, as it is shown previously that increases in the number of operations of such size do not remarkably affect u2c's evaluation times.

Figure 5.7 shows an increase in evaluation time for all libraries compared to $\boldsymbol{C}$. It can be observed that the increase in evaluation times for u2c is not relatively higher than the evaluation times for RBDL and PyBullet. Thus, it seems that the overhead related to the additional variable is approximately the same as RBDL's and PyBullet's overhead, which can be assumed to come from the additional variable and the small increase in the number of operations.

### Gravity versus Coriolis versus Inverse Dynamics

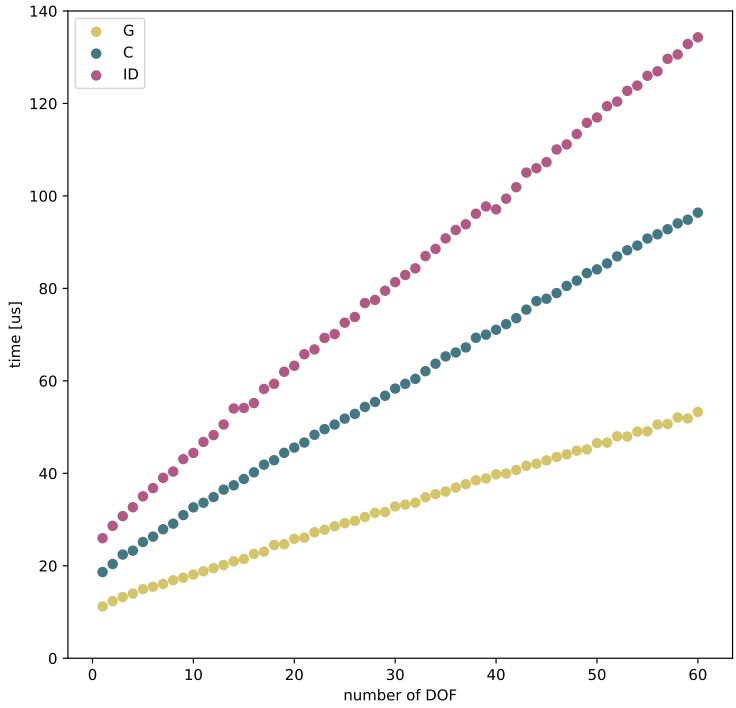



**Figure 5.8:** Median evaluation times for $\boldsymbol{G}$, $\boldsymbol{C}$, and ID for u2c.

**Table 5.7:** Number of operations for $\boldsymbol{G}$, $\boldsymbol{C}$, and ID for an increasing number of DOF.

| number of DOF: | 2 | 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|---|---|
| number of operations (G) | 26 | 429 | 883 | 1351 | 1829 | 2246 | 2726 |
| number of operations (C): | 103 | 1309 | 2766 | 4258 | 5784 | 7157 | 8683 |
| number of operations (ID): | 134 | 1378 | 2854 | 4366 | 5912 | 7305 | 8851 |

By comparing the evaluation times for $\boldsymbol{G}$, $\boldsymbol{C}$, and ID in Figure 5.8, one can observe how the evaluation times evolve for an increasing number of input variables and operations. Table 5.7 shows their corresponding number of operations. As $\boldsymbol{C}$ and ID encompass roughly the same amount of operations, the increase in evaluation time is primarily from the additional input variable needed. The change from $\boldsymbol{G}$ to $\boldsymbol{C}$ is bigger, especially for high number of DOF, which is due to the more extensive increase in the number of operations from $\boldsymbol{G}$ to $\boldsymbol{C}$.

### 5.3.4 Inertia Matrix

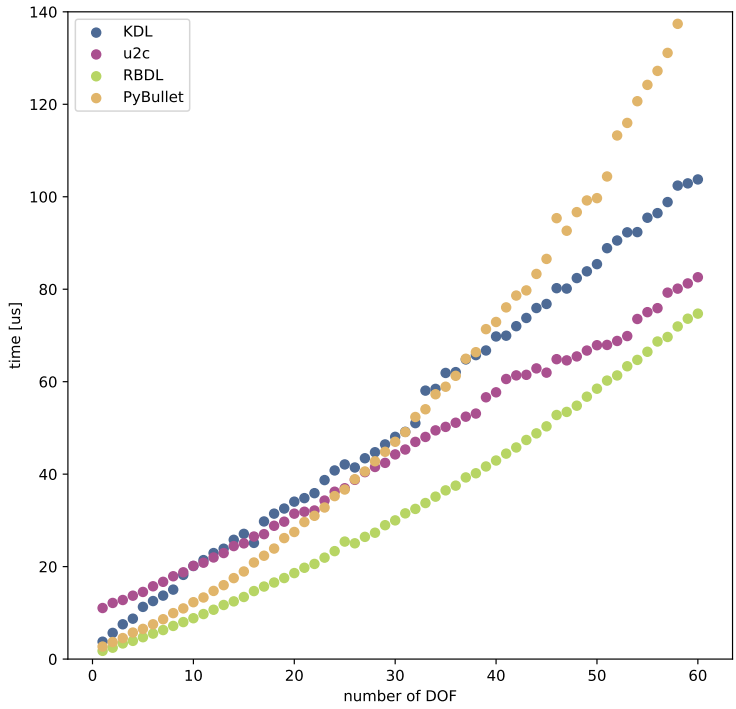



**Figure 5.9:** Median evaluation times for inertia matrix from 1 to 60 DOF.

**Table 5.8:** Number of operations for $\boldsymbol{M}$ for an increasing number of DOF.

| number of DOF: | 2 | 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|---|---|
| number of operations: | 52 | 3055 | 9499 | 17281 | 29320 | 40795 | 56308 |

The inertia matrix is obtained using CRBA, which has $O(n^3)$ worst-case complexity. In practice, this means a more rapid increase in the number of operations as the DOF increase, and that the increase in the number of operations becomes more significant for a higher number of DOF. This can be verified by Table 5.8.

Further, $\boldsymbol{M}$ only encompasses one input variable. This means that the overhead associated with input variables is not too significant. Additionally, it was previously found that the overhead associated with a relatively substantial increase in the

number of operations is present for u2c, but is relatively smaller than the increase in overhead for the numerical libraries. Based on this, the efficiency of u2c should potentially exceed the efficiency of the numerical libraries that are more sensitive to an increase in the number of operations.

Figure 5.9 shows that u2c excels over KDL after 10 DOF, and PyBullet after 24 DOF, substantiating the above. One can also observe that, while PyBullet has been the most efficient library for the evaluation of the dynamics discussed previously, PyBullet here ends up having the longest evaluation times. It can thus be concluded that PyBullet has the most overhead associated with the number of operations. It can further be assumed that PyBullet has little overhead associated with input variables as it has been the most efficient library for ID, where the number of operations is relatively small, and there are three input variables needed. RBDL seems to have some overhead related to the data type of the variables, as it is less efficient than PyBullet for the dynamics with fewer operations. RBDL can further be assumed to have little overhead related to the operations, as it is the fastest for evaluating $\boldsymbol{M}$.

To conclude, due to the high increase in the number of operations combined with only one variable, Figure 5.9 shows that RBDL ends up being the most efficient library, due to little overhead associated with the operations. Second up is u2c, for the same reason. Figure 5.9 shows that u2c seems closer to a linear increase than the others, indicating it is the library least affected by the increase in the number of operations. But u2c is slightly slower than RBDL due to a more significant overhead associated with the data type used. PyBullet ends up being the slowest library as it is most affected by increases in operations.

### 5.3.5 Forward Dynamics

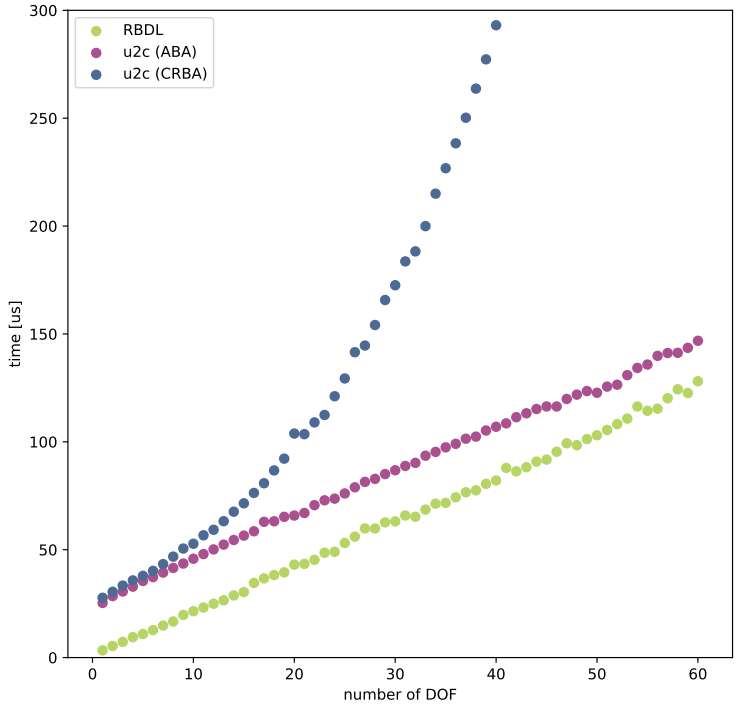



**Figure 5.10:** Median evaluation times for FD from 1 to 60 DOF.

**Table 5.9:** Number of operations for FD (ABA) for an increasing number of DOF.

| number of DOF: | 2 | 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|---|---|
| number of operations: | 209 | 4546 | 10320 | 15947 | 22123 | 27277 | 33253 |

Similar to the inverse dynamics, the overhead associated with three symbolic variables makes FD (ABA) a constant factor slower than RBDL. Further, Figure 5.10 demonstrates that FD (CRBA) works sufficiently for few DOF, while the $O(n^3)$ worst-case complexity makes it an unattractive choice for robots with a high number of DOF. This is due to the enormous amount of operations that occur for the FD (CRBA) expressions, since this approach finds the inverse of a large inertia

matrix. Table 5.9 shows the number of operations for the FD (ABA) expressions. The number of operations for the FD (CRBA) expressions have not been found due to the enormous amount of time this would take.

In Chapter 3, it is mentioned that it is more accurate to describe CRBA as an $O(nd^2)$ algorithm, and that these have the potential to exceed the speed of $O(n)$ algorithms, such as ABA, which is the reason why both approaches have been implemented. One can also observe in section 5.2.1 that for the 2-DOF pendulum, FD (CRBA) contains fewer operations than FD (ABA). Thus, if the overhead related to the number of operations has dominated over the overhead related to the input variables, FD (CRBA) had been faster than FD (ABA). However, as the overhead related to the number of input variables dominates for robots of few DOF, this restricts FD (CRBA) to the same evaluation times as FD (ABA).

### Forward Dynamics (ABA) versus Inverse Dynamics

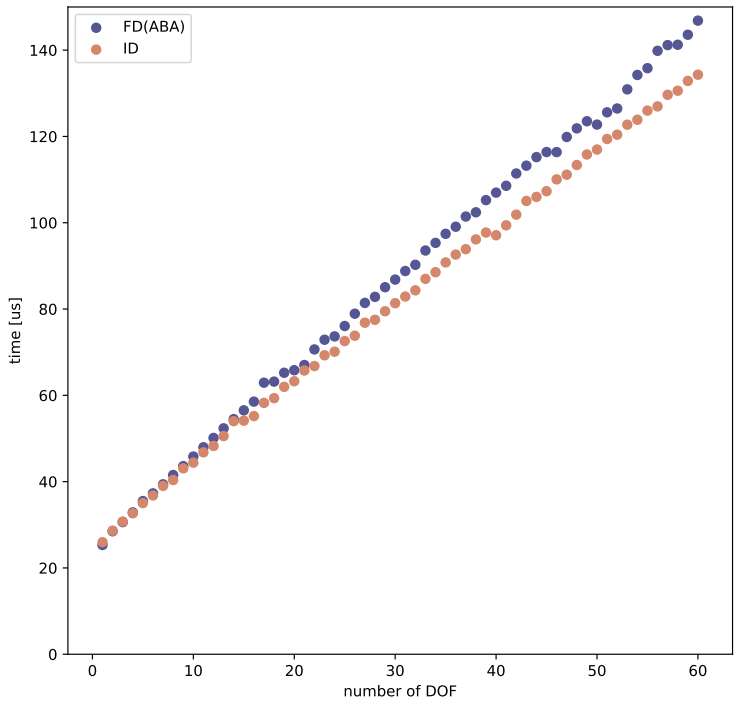



**Figure 5.11:** Median evaluation times for FD (ABA) and ID for u2c.

**Table 5.10:** Number of operations for FD (ABA) and ID for an increasing number of DOF.

| number of DOF: | 2 | 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|---|---|
| number of operations (ID): | 134 | 1378 | 2854 | 4366 | 5912 | 7305 | 8851 |
| number of operations (FD): | 209 | 4546 | 10320 | 15947 | 22123 | 27277 | 33253 |

Table 5.10 shows the number of operations for FD (ABA) and ID. It can be observed that the FD (ABA) expressions contain approximately four times as many operations as the ID expressions. Further, both expressions encompass three symbolic variables. Yet, one can observe from Figure 5.11 that it is not until the robots contain over 20 DOF, that the difference in operations has a noticeable effect on the evaluation times. This once again substantiates that there must be a reasonable difference in operations before it affects the evaluation times, due to the minimal overhead related to the operations.

### 5.3.6 Conclusion

This analysis has shown that u2c has proven strong in situations where there are minimal number of input variables involved, which is the case for both $\boldsymbol{G}$ and $\boldsymbol{M}$. A significant difference between these dynamics is that the number of operations increase rapidly for $\boldsymbol{M}$, while the increase in number of operations is much smaller for $\boldsymbol{G}$. Either way, in these situations u2c excels when compared to several of the numerical libraries, and it was found that u2c is the library with the smallest slope for $\boldsymbol{M}$ and $\boldsymbol{G}$. One can thus conclude that when only one input variable is involved, u2c is less affected by changes in the number of operations.

The analysis also showed that the increase in evaluation times for u2c is more significant from $\boldsymbol{G}$ to $\boldsymbol{C}$ than from $\boldsymbol{C}$ to ID, despite that, in both cases, one input variable is added. It was found that this is due to the more significant increase in the number of operations from $\boldsymbol{G}$ to $\boldsymbol{C}$ than from $\boldsymbol{C}$ to ID, indicating that u2c is more sensitive to increases in operations when there are more input variables involved.

Further, for FD it was shown how the CRBA and ABA approaches are just as good for robots of few DOF. In theory, the CRBA approach may result in fewer operations, but since this is just the case for robots of few DOF, it is the effect of the overhead related to the input variables that dominates, such that the effect of fewer operations is neglected. It was also found that FD using the CRBA approach becomes very inefficient when the DOF increase, due to the enormous increase in the number of operations that occur when having to find the inverse of a large inertia matrix.

## 5.4 Summary

This chapter has found that, despite u2c finding symbolic expressions of the robot's dynamics, the evaluation times remain the same order of magnitude as the evaluation times of the numerical libraries. Whether u2c is more or less efficient than the numerical libraries, depends on two factors: the number of variables involved in the expression and the number of operations that the expression encompasses. The latter mainly depends on the algorithm used to retrieve the expression and the number of DOF of the robot.

By comparing the evaluation times of the dynamics of a 2-DOF pendulum, a 6-DOF UR5, and a 16-DOF snake robot, it was found that u2c is relatively slower for most of the dynamics compared to the numerical libraries, due to the overhead related to the CasADi data types. The results showed that u2c becomes relatively more efficient compared to the numerical libraries for an increasing number of DOF, which indicated that u2c is less affected by an increase in number of operations, as a consequence of the increasing number of DOF.

The 60 DOF analysis substantiated these results. In this analysis, the libraries were exposed to a more extensive increase in the number of operations due to the higher increase in the number of DOF. The results yielded that u2c is the library that is least affected by an increase in the number of operations when only one input variable is involved. When several input variables are involved, u2c is more sensitive to an increase in DOF.

# CHAPTER 6

# Derivatives Timing

Previously the main focus has been rigid body dynamics and its use. However, in the matter of control, optimization, and estimation within robotics, one frequently requires the derivatives of the system's dynamics. Typical situations are optimal control and trajectory optimization, which is becoming the standard approach to control advanced robotic systems, as presented by Posa et al. (2014) and Koenemann et al. (2015). The most computationally expensive part of these optimization algorithms is to compute the dynamics derivatives. At most, up to 90% of the computation time is spent computing these derivatives, according to Carpentier and Mansard (2018).

The derivatives are difficult to derive analytically. The manual process of deriving the derivatives is both complex and error prone. Another matter to consider is that they are often difficult to optimize and to implement efficiently.

Another approach used to derive the dynamics derivatives is finite differences. It can be considered the simplest way to obtain the derivatives, and the method is based on evaluating the input dynamics several times while adding a small increment on the input variables. However, this approach is sensitive to numerical rounding errors, as shown in Carpentier and Mansard (2018). Additionally, in cases where this approach has shown fast enough to be applied on real system, e.g. Tassa et al. (2012) and Koenemann et al. (2015), fine parallelization is required.

To rely on AD, as formerly mentioned in section 2.1.2, for obtaining the derivatives, generally requires intermediate computations. These computations are often hard to avoid or simplify, but code-generation reduces this problem. A known issue in this matter is that it can be a costly process to compose. As CasADi is an AD framework with built-in code-generation functionality, this provides an opportunity to rapidly retrieve the derivatives after obtaining the dynamics.

In this chapter, it is explained how to obtain the dynamics derivatives in the

CasADi framework using u2c, followed by a comparison between the evaluation times of the dynamics and the evaluation times of their derivatives. It is also demonstrated how the number of operation evolves with an increasing number of DOF for the derivatives, and how the evaluation times are affected by an increase in the number of operations.

For implementational details about the derivatives timing tests, they can be found in the following directory:

```
urdf2casadi
└── examples
    └── timing
        └── derivatives
```

## 6.1 Obtaining Dynamics Derivatives

When timing the derivatives, the same approach is used as for timing the dynamics. Before the test, the derivatives are easily obtained using CasADi's built-in Jacobian functionality, where the user can explicitly define which variable one wishes to find the derivative with respect to.

CasADi provides different ways of obtaining derivatives. The `cs.jacobian`-functionality finds the partial derivative of a given CasADi function with respect to a CasADi variable.

The second method allows the user to find the time derivative with respect to the variables explicitly defined by the user. The time derivatives are obtained by using CasADi's `cs.jtimes`-functionality. Based on its inputs, `cs.jtimes` calculates the Jacobian-times-vector product, in a way much more efficient than first creating the full Jacobian and performing a matrix-vector multiplication. This is especially useful as it is often the time derivative that is needed in the formulation of optimal control problems within robotics.

Further, all derivatives expressions are converted to a CasADi-function where the -OFast compiler flag is used for just-in-time compilation of derivatives functions for maximum efficiency. This is the same approach as used in the returned functions of u2c. A more detailed implementational description of how the derivatives are obtained in the CasADi framework is given in the user example of u2c, which can be found in the `user`-folder. The user example is further discussed in the next chapter, and found in the following directiory:

```
urdf2casadi
└── examples
    └── user
        └── ur5_example
```

## 6.2 Derivatives Timing - cs.jacobian

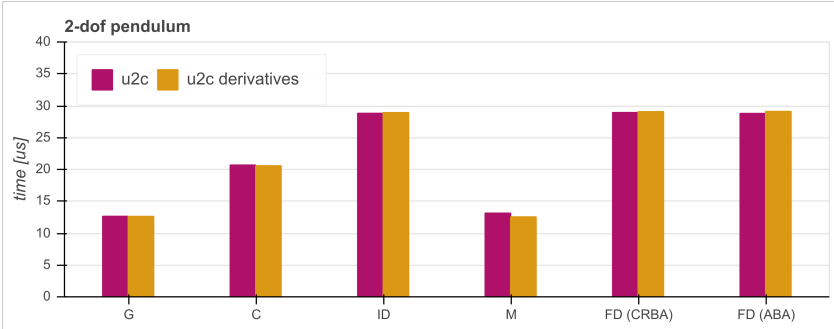
In this section, the evaluation times of the derivatives obtained using CasADi's cs.jacobian-functionality is compared against their related dynamics functions. It is also looked into how the number of operations increases when going from the dynamics expressions to their derivatives and how this affects the evaluation times.

The derivatives are found with respect to all the variables included in the dynamics expressions:

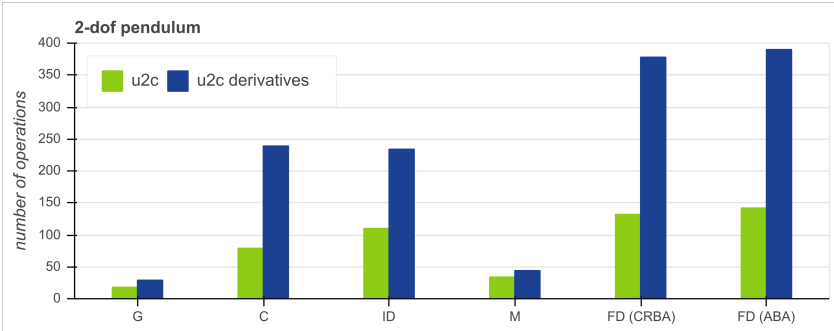
$$\nabla \boldsymbol{G} = \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{q}}$$
$$\nabla \boldsymbol{C} = [\frac{\partial \boldsymbol{C}}{\partial \boldsymbol{q}}, \frac{\partial \boldsymbol{C}}{\partial \dot{\boldsymbol{q}}}]$$
$$\nabla \mathrm{ID} = [\frac{\partial \mathrm{ID}}{\partial \boldsymbol{q}}, \frac{\partial \mathrm{ID}}{\partial \dot{\boldsymbol{q}}}, \frac{\partial \mathrm{ID}}{\partial \ddot{\boldsymbol{q}}}]$$
$$\nabla \boldsymbol{M} = \frac{\partial \boldsymbol{M}}{\partial \boldsymbol{q}}$$
$$\nabla \mathrm{FD} = [\frac{\partial \mathrm{FD}}{\partial \boldsymbol{q}}, \frac{\partial \mathrm{FD}}{\partial \dot{\boldsymbol{q}}}, \frac{\partial \mathrm{FD}}{\partial \boldsymbol{\tau}}]$$

To take the derivative of a matrix with respect to a vector, CasADi reshapes the matrix to a vector, then creates the vector-vector Jacobian matrix.

### 6.2.1 2-DOF pendulum




**(a)** Median evaluation times.



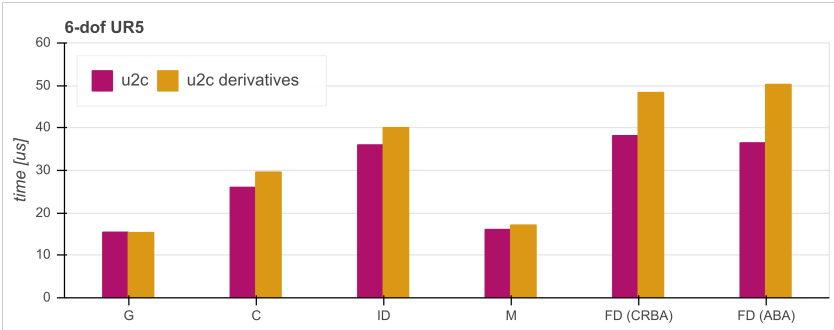

**(b)** Number of operations.

**Figure 6.1:** Median evaluation times and number of operations for the dynamics and their related derivatives for the 2-DOF pendulum.

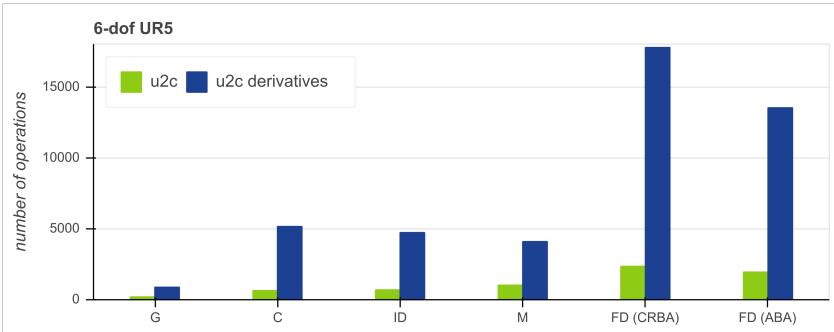
Figure 6.1a shows that the evaluation times of the derivatives obtained with `cs.jacobian` are approximately the same as for their related dynamics functions. It can even be observed that the evaluation time of the derivative of $\boldsymbol{M}$ seems to be a few microseconds faster than the evaluation time of $\boldsymbol{M}$. For the other dynamics parameters, the difference in evaluation time from the dynamics functions to their related derivatives is negligible.

As the dynamics functions and their related derivatives functions encompass the same number of input variables, the potential difference in evaluation time is due to the increase in the number of operations. Figure 6.1b shows the change in the number of operations for the derivatives and their related dynamics. One can observe that FD is the dynamics term most affected by an increase in the number of operations, while $\boldsymbol{G}$ and $\boldsymbol{M}$ is hardly affected at all. Thus, the few microseconds decrease in evaluation time for the derivative of $\boldsymbol{M}$ can be assumed to be due to normal timing variations. It can also be concluded that the increases in operations are of minor importance for the 2-DOF pendulum and do not remarkably affect the evaluation times of the derivatives.

### 6.2.2 6-DOF UR5




**(a)** Median evaluation times.




**(b)** Number of operations.

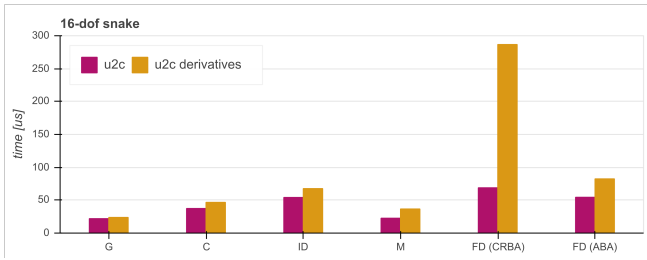
**Figure 6.2:** Median evaluation times and number of operations for the dynamics and their related derivatives for the 6-DOF UR5.

From Figure 6.2a, it can be observed that most of the derivatives for the UR5 have a noticeable increase in evaluation time compared to their related dynamics, although the increase is relatively small. Similar to the pendulum, it is the evaluation time of FD that has the most significant increase, which is approximately 10-15 microseconds for both approaches. $\boldsymbol{G}$ and $\boldsymbol{M}$ are hardly affected by the increase in the number of operations for their derivatives, while it can be observed an increase of a few microseconds for the evaluation times of the derivatives of $\boldsymbol{C}$ and ID.

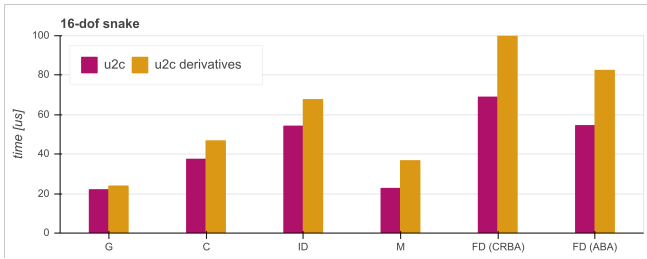
Figure 6.2b shows that the 6-DOF UR5 is more exposed to an increase in the number of operations for the derivatives than the 2-DOF pendulum. The number of operations for the derivatives of UR5 is of order 3 and 4, except for the derivative of $\boldsymbol{G}$, which is still of order 2. This explains why one can observe an increase in evaluation time for all the dynamics derivatives except for $\boldsymbol{G}$. Further, one can see that the increase in the number of operations is most significant for FD, which explains why the increase in evaluation time is biggest for FD. One can also observe that even though the increase in the number of operations is approximately the same for $\boldsymbol{C}$, ID, and $\boldsymbol{M}$, it seems that the evaluation times of $\boldsymbol{C}$ and ID are more

affected by the increase in the number of operations. This is most likely because $\boldsymbol{C}$ and ID encompass more input variables than $\boldsymbol{M}$, and is thus more sensitive to an increase in the number of operations.

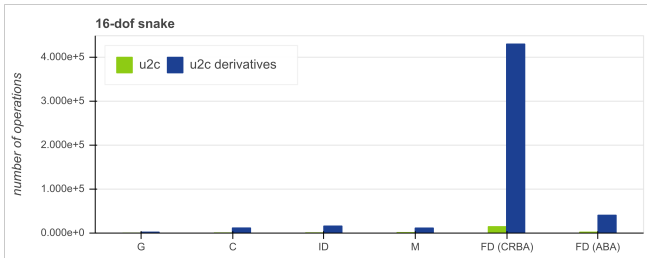
### 6.2.3 16-DOF snake



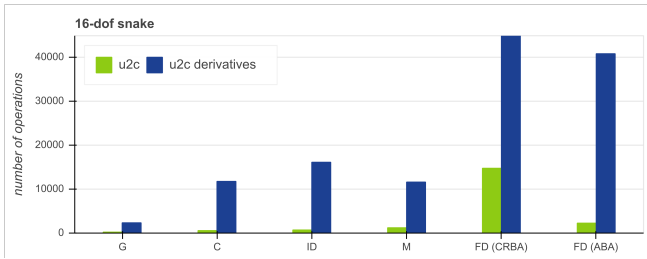

**(a)** Median evaluation times.




**(b)** Median evaluation times zoomed in.




**(c)** Number of operations.




**(d)** Number of operations zoomed in.

**Figure 6.3:** Median evaluation times and number of operations for the dynamics and their related derivatives for the 16-DOF snake.

Figure 6.3 shows the evaluation times for the derivatives and the number of operations, both fully zoomed out, and zoomed in, to get a fuller overview of all the derivatives, as the derivative of FD (CRBA) increases significantly more than the other derivatives.

From Figure 6.3a and b, it can be seen that FD is still the derivative most exposed to an increase in evaluation time. For the other dynamics, Figure 6.3 shows similar results as for the robots discussed earlier, except with a slightly higher increase in evaluation times for the derivatives. This is a result of a higher increase in the number of operations, which can be observed from Figure 6.3d, where the higher increase in the number of operations is due to the increased number of DOF for the snake.

For the other robots, the increase in the evaluation time for the derivative of FD is approximately the same for FD (CRBA) and FD (ABA). For the snake, however, it is seen that FD obtained with CRBA is significantly more exposed to an increase in evaluation time than FD obtained with ABA. From Figure 6.3c, it can be understood that the significant increase in evaluation time for the derivative of FD (CRBA) is due to the major increase in the number of operations. It can be observed that the expression for the derivative of FD (CRBA) consists of over 400 000 operations, in contrast to the expression for the derivative of FD (ABA), which consists of 40 000 operations. It is a known fact that CRBA is more exposed to an increase in DOF, as it has $O(n^3)$ worst-case complexity. Further, the CRBA approach for solving the FD involves solving the equation of motion for $\ddot{\boldsymbol{q}}$, which again implies calculating the inverse of the inertia matrix. For the 16-DOF snake, this means that the inverse of a $16 \times 16$ matrix has to be calculated for the FD, making the derivative of FD more exposed to an increase in the number of operations.

### 6.2.4 Conclusion

The above shows that, on a general basis, dynamics derivatives obtained with `cs.jacobian` result in evaluation times not much higher than the evaluation times of the dynamics themselves, which can be considered a satisfactory result. It is seen that the evaluation times of the derivatives for robots of several DOF increase more relative to their related dynamics, as this causes a more significant increase in the number of operations. Further, the evaluation times of the derivatives of robots with few DOF remain the same as for the related dynamics. It was also seen that the derivatives of the dynamics which inhabit the most input variables are more exposed to an increase in evaluation time.

It can be concluded that for a certain increase in the number of operations, the evaluation times of the derivatives are not profoundly affected, and stay within the same order of magnitude as the evaluation time of its related dynamics function. If the increase in the number of operations exceeds a certain level, this affects the evaluation time to a much higher degree, as seen for the derivative of FD (CRBA) for the 16-DOF snake. Further, FD using the ABA approach has been implemented as it was known that FD using the CRBA approach was not recommended for robots with a high number of DOF, and thus the ABA approach should be the

preferred alternative when obtaining the derivative of FD for such robots.

## 6.3 Derivatives Timing - `cs.jtimes`

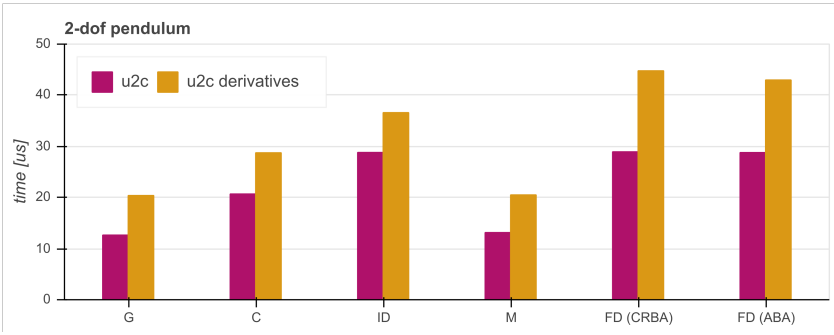
In this section, the evaluation times of the derivatives obtained using CasADi's `cs.jtimes`-functionality is compared against their related dynamics functions.

The derivatives are found with respect to all the variables included in the dynamics expressions, and as formerly mentioned, `cs.jtimes` gives the time derivative of the dynamics:

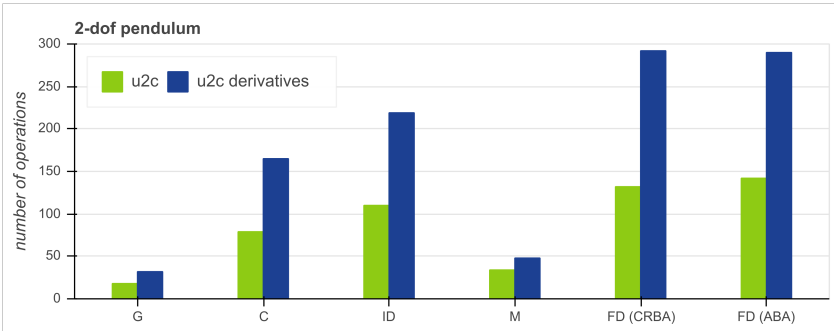
$$\dot{\boldsymbol{G}} = \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{q}}\dot{\boldsymbol{q}}$$
$$\dot{\boldsymbol{C}} = \frac{\partial \boldsymbol{C}}{\partial \boldsymbol{q}}\dot{\boldsymbol{q}} + \frac{\partial \boldsymbol{C}}{\partial \dot{\boldsymbol{q}}}\ddot{\boldsymbol{q}}$$
$$\dot{\mathrm{ID}} = \frac{\partial \mathrm{ID}}{\partial \boldsymbol{q}}\dot{\boldsymbol{q}} + \frac{\partial \mathrm{ID}}{\partial \dot{\boldsymbol{q}}}\ddot{\boldsymbol{q}} + \frac{\partial \mathrm{ID}}{\partial \ddot{\boldsymbol{q}}}\dddot{\boldsymbol{q}}$$
$$\dot{\boldsymbol{M}} = \frac{\partial M}{\partial \boldsymbol{q}}\dot{\boldsymbol{q}}$$
$$\dot{\mathrm{FD}} = \frac{\partial \mathrm{FD}}{\partial \boldsymbol{q}}\dot{\boldsymbol{q}} + \frac{\partial \mathrm{FD}}{\partial \dot{\boldsymbol{q}}}\ddot{\boldsymbol{q}} + \frac{\partial \mathrm{FD}}{\partial \boldsymbol{\tau}}\dot{\boldsymbol{\tau}}$$

The `cs.jtimes` approach naturally includes an additional variable for the time derivative of each input variable of the dynamics parameter. The derivatives functions of $\boldsymbol{G}$, $\boldsymbol{C}$, ID, and $\boldsymbol{M}$ thus encompass one additional input variable when finding the time derivative: $\dot{\boldsymbol{G}} = \dot{\boldsymbol{G}}(\boldsymbol{q}, \dot{\boldsymbol{q}})$, $\dot{\boldsymbol{C}} = \dot{\boldsymbol{C}}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}})$, $\dot{\mathrm{ID}} = \dot{\mathrm{ID}}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}, \dddot{\boldsymbol{q}})$. The time derivative of FD gets two additional input variables: $\dot{\mathrm{FD}} = \dot{\mathrm{FD}}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}, \boldsymbol{\tau}, \dot{\boldsymbol{\tau}})$. Thus, one has to consider the impact of these additional variables on the evaluation times, and not just the increase in the number of operations alone.

### 6.3.1 2-DOF pendulum




**(a)** Median evaluation times.

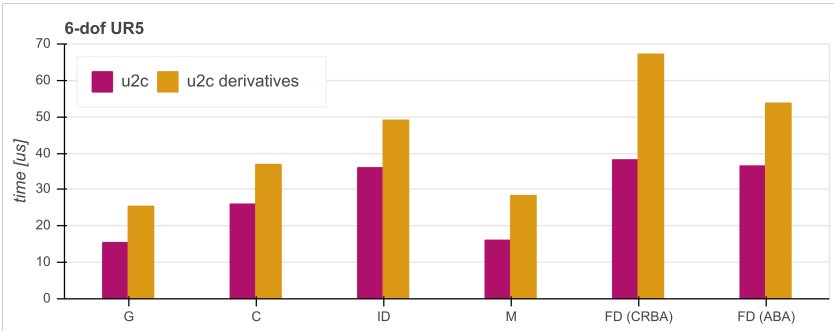



**(b)** Number of operations.

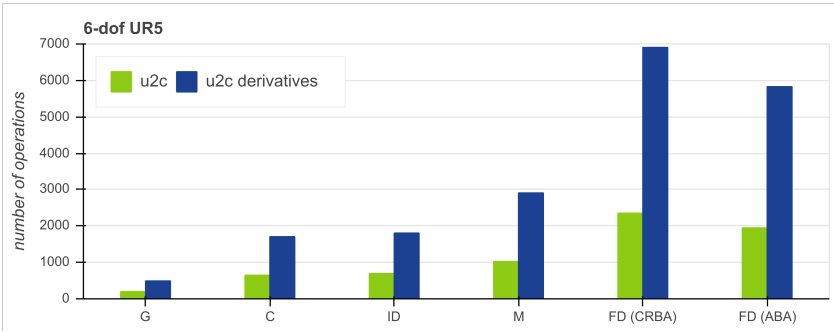
**Figure 6.4:** Median evaluation times and number of operations for the dynamics and their related derivatives for the 2-DOF pendulum.

From Figure 6.4, one can observe that the evaluation times of the dynamics derivatives, where it is still the evaluation times of the FD derivatives that increase the most. By comparing Figure 6.4b to Figure 6.1b one can observe that the derivatives obtained with `cs.jtimes` have fewer operations than the derivatives obtained with `cs.jacobian`. The smaller increase in the number of operations for `cs.jtimes` represents CasADi's efficient calculation of the Jacobian-times-vector product, rather than creating the full Jacobian and performing a matrix-vector multiplication. Further, it was found in section 6.2.1 that the increase in the number of operations for the 2-DOF pendulum does not result in a remarkable increase in evaluation times for the derivatives. Hence, it can be concluded that the increase in evaluation times displayed in Figure 6.4 is mainly due to the additional input variables.

### 6.3.2 6-DOF UR5




**(a)** Median evaluation times.

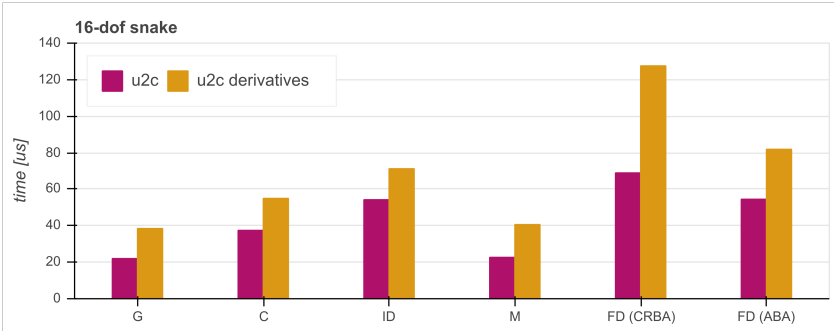



**(b)** Number of operations.

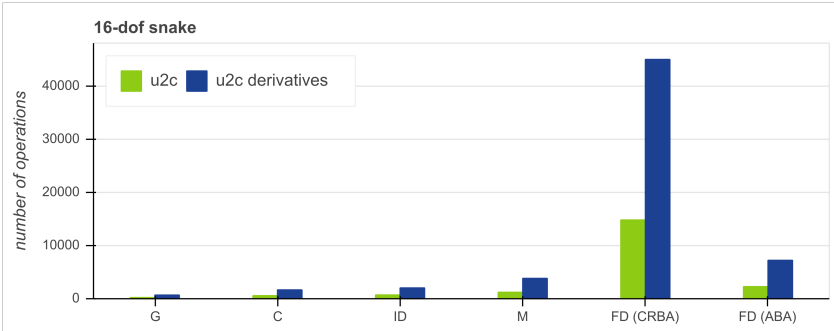
**Figure 6.5:** Median evaluation times and number of operations for the dynamics and their related derivatives for the 6-DOF UR5.

Figure 6.5a shows that the evaluation times of the derivatives increase more for the UR5 than for the pendulum, as expected due to the increase in DOF. One can also observe how the evaluation times of the derivatives of FD (CRBA) tends to increase more rapidly than the evaluation times of the derivatives of FD (ABA). Further, Figure 6.5 shows that, compared to Figure 6.2, the increases in the number of operations are several thousands of operations less than for the derivatives obtained with `cs.jacobian`. One can also observe a tendency of `cs.jtimes` having fewer operations than `cs.jacobian` as the DOF increase. There are thus two conflicting effects affecting the evaluation times. The effect of a lower increase in the number of operations should make the evaluation times increase less, and the effect of having more input variables should make the evaluation times increase more than the derivatives obtained with `cs.jacobian`. Comparing Figure 6.5a with Figure 6.2a, one can see that the effect of the additional input variables is dominant as the increase in evaluation times is slightly bigger for the UR5 using `cs.jtimes`.

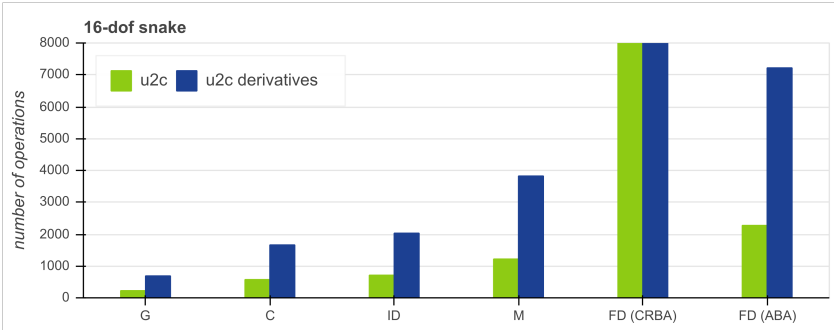
### 6.3.3 16-DOF snake




**(a)** Median evaluation times.




**(b)** Number of operations.




**(c)** Number of operations zoomed in.

**Figure 6.6:** Median evaluation times and number of operations for the dynamics and their related derivatives for the 16-DOF snake.

By evaluating Figure 6.6b, one can observe that the number of operations for the derivative of FD (CRBA) is 40 000, which for the derivative obtained with `cs.jacobian` is one order of magnitude higher. One can also observe, from Figure

6.6c, how the increase in the number of operations for the other dynamics is also lower. By comparing with the other robots, one can observe that the number of operations saved using `cs.jtimes` instead of `cs.jacobian` increases as the DOF increase, indicating that the effect of a lower increase in evaluation times for the derivatives should be strengthened as the DOF increase. Further, it was found in Chapter 5 that the input variables bring a constant overhead, in addition to an overhead related to the increase in symbolic elements of the symbolic vectors. The constant overhead was found to be dominating, leading to the overall overhead associated with input variables decreasing as the DOF increase.

Figure 6.6a substantiates that the effect of the lower increase in the number of operations becomes more dominant, while the effect of the additional input variables becomes less dominant, as the DOF increase. By comparing Figure 6.6a to Figure 6.3a, one can see that the evaluation times of the derivatives are approximately the same for certain derivatives, while the increase in the evaluation times for the other derivatives is smaller than for the robots of fewer DOF, substantiating the above.

### 6.3.4 Conclusion

There are two competing effects that determine the evaluation times of the time derivatives. The additional input variables introduce an extra overhead increasing evaluation times, and the lower number of operations compared to the `cs.jacobian` approach decreases the evaluation times. For robots with a low number of DOF, the number of operations is not as noticeable as the input variable overhead. For robots with higher number of DOF, the evaluation times of `cs.jtimes` are lower than that of `cs.jacobian`, as there are much fewer operations involved. For calculating the time derivative of a dynamics parameter, one would still have to multiply the Jacobian by the time derivatives of the input variables.

## 6.4 `cs.jacobian` versus `cs.jtimes`

From the findings in the previous sections, there is reason to believe that when the robot exceeds a certain number of DOF, the evaluation times of the `cs.jtimes`-derivatives are lower than the evaluation times of the `cs.jacobian`-derivatives. To illustrate, the evaluation times are summarized in Table 6.1 and Table 6.2.

**Table 6.1:** Median evaluation times for the dynamics derivatives obtained with `cs.jacobian` with respect to $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$, $\ddot{\boldsymbol{q}}$ and $\boldsymbol{\tau}$.

| | pendulum | UR5 | snake |
|---|---|---|---|
| $\nabla \boldsymbol{G}$ | 12.63 µs | 15.38 µs | 23.47 µs |
| $\nabla \boldsymbol{C}$ | 20.57 µs | 29.62 µs | 46.78 µs |
| $\nabla$ID | 28.91 µs | 40.08 µs | 67.68 µs |
| $\nabla \boldsymbol{M}$ | 12.54 µs | 17.14 µs | 36.72 µs |
| $\nabla$FD (CRBA) | 29.07 µs | 48.35 µs | 286.73 µs |
| $\nabla$FD (ABA) | 29.10 µs | 50.52 µs | 82.48 µs |

**Table 6.2:** Median evaluation times for the dynamics derivatives obtained with `cs.jtimes` with respect to $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$, $\ddot{\boldsymbol{q}}$ or $\boldsymbol{\tau}$.

| | pendulum | UR5 | snake |
|---|---|---|---|
| $\dot{\boldsymbol{G}}$ | 20.38 µs | 25.43 µs | 38.46 µs |
| $\dot{\boldsymbol{C}}$ | 28.74 µs | 36.92 µs | 54.96 µs |
| $\dot{\text{ID}}$ | 36.66 µs | 49.08 µs | 71.20 µs |
| $\dot{\boldsymbol{M}}$ | 20.50 µs | 28.37 µs | 40.61 µs |
| $\dot{\text{FD}}$ (CRBA) | 44.78 µs | 66.93 µs | 127.73 µs |
| $\dot{\text{FD}}$ (ABA) | 42.98 µs | 53.75 µs | 81.86 µs |

Table 6.1 and Table 6.2 show that even though the `cs.jacobian`-derivatives are much faster for robots of few DOF, this difference becomes smaller as the DOF increase. For the 16-DOF snake the derivatives of FD (ABA), ID, and $\boldsymbol{M}$ have approximately the same evaluation times for both derivatives. There is thus reason to believe that if the DOF continue to increase, the `cs.jtimes`-derivatives exceed the efficiency of the `cs.jacobian`-derivatives. This is due to a higher number of DOF results in a bigger difference in operations between the two types of derivatives. This is displayed systematically in Table 6.3 and Table 6.4.

**Table 6.3:** Number of operations for the dynamics derivatives obtained with `cs.jacobian`.

| | pendulum | UR5 | snake |
|---|---|---|---|
| $\nabla \boldsymbol{G}$ | 29 | 881 | 2347 |
| $\nabla \boldsymbol{C}$ | 239 | 5157 | 11792 |
| $\nabla$ID | 234 | 4731 | 16159 |
| $\nabla \boldsymbol{M}$ | 44 | 4095 | 11635 |
| $\nabla$FD (CRBA) | 378 | 17764 | 431178 |
| $\nabla$FD (ABA) | 390 | 13517 | 40901 |

**Table 6.4:** Number of operations for the dynamics derivatives obtained with `cs.jtimes`.

| | pendulum | UR5 | snake |
|---|---|---|---|
| $\dot{\boldsymbol{G}}$ | 32 | 491 | 687 |
| $\dot{\boldsymbol{C}}$ | 165 | 1707 | 1660 |
| $\dot{\text{ID}}$ | 219 | 1807 | 2029 |
| $\dot{\boldsymbol{M}}$ | 48 | 2909 | 11635 |
| $\dot{\text{FD}}$ (CRBA) | 292 | 6906 | 44940 |
| $\dot{\text{FD}}$ (ABA) | 290 | 5828 | 7218 |

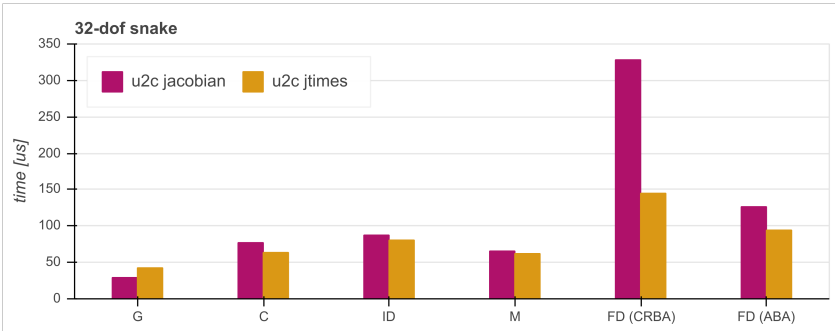



**Figure 6.7:** Median evaluation times and number of operations for the dynamics and their related derivatives for the 32-DOF snake.

To investigate the theory further, a URDF representation of a hypothetical 32-DOF snake robot has been constructed. The results are displayed in Figure 6.7, and substantiate the fact that when a robot exceeds a certain number of DOF, the efficiency gained by fewer operations overcomes the overhead associated with the additional input variables in the functions obtained with `cs.jtimes`. Figure 6.7 shows that this is the fact for all the dynamics except for $\boldsymbol{G}$ for a 32-DOF robot.

The results illustrate that the number of operations can have a significant impact on the evaluation times. Yet, by evaluating Figure 6.7, one can conclude that it has to be a matter of tens and hundreds of thousand operations before it has a significant impact on the evaluation times.

## 6.5 Conclusion

This chapter has illustrated that u2c can be used to derive efficient functions for the dynamics derivatives in the CasADi framework. It is shown that the derivatives can be derived in two different ways, using either `cs.jacobian`, or `cs.jtimes` for efficiently calculating the time derivative.

It is further illustrated that the evaluation times of the derivatives are not much longer than the evaluation times of the dynamics functions themselves. And, except for FD (CRBA), the evaluation times of the dynamics derivatives seem to be of the same order of magnitude as the evaluation times of the dynamics. Thus, it can be concluded that the derivatives, whether they are found by `cs.jacobian` or by `cs.jtimes`, are suitable for use with trajectory optimization, optimal control, and other approaches where dynamics derivatives are needed.

The `cs.jtimes` approach is an efficient approach to calculating the time derivative of the dynamic parameters, and the `cs.jacobian` approach is an easy way of getting the partial derivatives of a dynamic parameter. The comparison between the timing results of `cs.jtimes` and `cs.jacobian` reiterated that the introduction of an additional input variable does have a constant overhead that can be quite significant. Even then, `cs.jtimes` is more efficient than `cs.jacobian` for certain numbers of DOF. This is the intended behavior of `cs.jtimes` from the CasADi developers, as it is intended as a faster subroutine exploiting the directionality of the derivative.

# CHAPTER 7

# Epilogue

*This chapter is based on the Epilogue chapter from the specialization project associated with this thesis. Modifications have been made according to changes during the latter part of the project.*

## 7.1 Discussion

In this section, a discussion regarding the library structure and implementation is presented.

### 7.1.1 C++ versus Python

One of the first choices that had to be made regarding the implementation of u2c, was which programming language to use. CasADi is available for C++, Matlab, and Python with minimal difference in performance. Since Matlab is licensed, many companies and programmers do not use Matlab, and thus this option was excluded. Advantages of using Python are that the Python API is the best documented and proven to be very stable, in addition to Python being a high-level language. The C++ API is also stable but is not ideal for getting started with as there is limited documentation. Additionally, the C++ API lacks the interactivity of other languages like Python and Matlab.

C++ is a compiled language, while Python is an interpreted language. Thus, each platform running C++ code requires a compiler. While for Python, all that is needed is a Python interpreter to be installed so that the executable can be interpreted at run-time. The decisive advantage of Python being an interpreted language is that it becomes a cross-platform language, meaning that if u2c is implemented in Python, the library can be used on all platforms. C++ can be seen as a

cross-platform language in the sense that there exist compilers for many platforms, but not all features or libraries are available on all platforms.

The other difference of importance is that Python is a dynamic language while C++ is not. The meaning of Python being a dynamic language is that under declaration of objects, no type is given and hence the type of this object can change dynamically. This is in contrast to C++, where one has to specify the object type. The consequence of Python being a dynamical language is that declarations normally taken care of during compilation are moved into run-time. The dynamic lookup method can be costly, leading to less efficient code. On the contrary, when executing compiled C++ code, the execution path is more predictable leading to the compiler being able to perform optimization.

To summarize, the leading advantage of C++ is that it is fast and efficient as it is not a dynamic language. However, it is not cross-platform as a compiler is required. The advantages of Python are that it is cross-platform, high level, and has a big community, while the disadvantage is that Python can be slow as it is a dynamic language. The implementation choice thus fell on Python. An essential reason for this choice is that CasADi support C-generation of code, thus making it possible to get the advantages of using Python while at the same time preserving the opportunity to obtain the beneficial functionality of compiled code for function calls of dynamics parameters.

### 7.1.2 The Library Structure

Before starting the implementation of u2c, a thorough investigation had to be made regarding the library structure concerning the resulting usage. Several structures were considered. One option was to structure the library as a suite of functions, such that the user could use these functions as desired. This structure is used by Featherstone's library, spatial_v2, as mentioned in the introduction.

The other option was to use a class structure where one robot is related to one instance of the class, and the class represents the parsing from a URDF to the robot's kinematics and dynamics.

The choice fell on the latter option due to its ease of use. This option provides the user with a way of obtaining the robot's kinematics and dynamics without much prior knowledge of the inner structure of the library. The class provides a suite of methods for loading the robot descriptions and a suite of get-methods that provides the kinematics and dynamics, and this is all information the user needs.

### 7.1.3 Forward Dynamics - ABA versus CRBA

As previously mentioned, u2c provides two different ways of obtaining the forward dynamics: using ABA, or using CRBA and then solving the equation of motion for a rigid multi-body system. Both of them have been implemented as ABA is a better choice for robots of many DOF, while it was speculated that the CRBA approach could be a better option for robots of few DOF, due to its $O(nd^2)$-complexity.

In the timing chapters, it was found that for robots of few DOF, the CRBA approach would sometimes lead to expressions of fewer operations. However, in

these cases, the overhead associated with the symbolic variables would dominate, such that the evaluation times of the forward dynamics, either using the ABA or CRBA approach, remain approximately the same.

It was, therefore, wondered if the CRBA approach should be kept or removed from the library's functionality. During the derivatives investigation, however, it was found that in some use cases, e.g. when wanting to find the derivative of the forward dynamics with respect to only $\dot{\boldsymbol{q}}$ or $\boldsymbol{\tau}$ for robots of few DOF, the CRBA approach was remarkably faster due to significantly fewer operations. Thus, one can conclude that there are not many use cases where the FD CRBA approach is remarkably better than the FD ABA approach. Yet, these scenarios do exist, and both approaches are kept so that the user can investigate for oneself which approach is best suited for one's use case.

### 7.1.4 Documentation

Providing necessary documentation for new users is an important aspect when developing a new library. The documentation of u2c consists of two parts: docstrings describing the functionality of each function and module, and a thorough user example which explains and shows the complete functionality provided by u2c.

#### Docstrings

Python provides docstrings as a form of documenting Python modules, functions, classes, and methods. They represent a convenient way of writing documentation, where the docstrings are specified in the source code to document a specific code segment. The docstrings have of purpose to describe what the code does, and not how.

The functions, classes, methods, and modules of u2c all have docstrings, and are declared using `""" """` just below the class, method or function declaration.

The docstrings are accessible to the user using the `__doc__`-method of the object or using the `help()`-function provided by Python. To illustrate, given that the user has installed u2c using the installation instructions provided in the `README`-file, the docstring documentation is accessible as:

```
import urdf2casadi.urdfparser as u2c
u2c.URDFparser.__doc__
u2c.URDFparser.get_inverse_dynamics_rnea.__doc__
```

Output:

```
'Class that turns a URDF chain to a casadi function.'

'Returns the inverse dynamics as a casadi function.'
```

One can also use the `help()`-function for a more descriptive documentation. For the methods of the class, the `help()`-function also gives the input arguments of the methods. For instance, `help(u2c.URDFparser.get_inverse_dynamics_rnea)` gives the following output:

```
'Help on method get_inverse_dynamics_rnea in module
urdf2casadi.urdfparser:

get_inverse_dynamics_rnea(self, root, tip, gravity=None, f_ext=None)
method of urdf2casadi.urdfparser.URDFparser instance
    Returns the inverse dynamics as a casadi function.'
```

Further, `help(u2c.URDFparser)` returns the docstring of the `URDFparser`-class together with the `help()`-function's description of all the methods provided by the `URDFparser`-class, thus giving a full overview of the class with its class variables, methods, and their input arguments.

#### User Example

The user example can be considered the main source of the u2c documentation. It is developed as a Jupyter notebook, giving the advantage of both showing how the code works, and explaining the code with text, in a systematic way. The user example shows and explains all the functionality provided by u2c, using a UR5 as an example. It also gives a thorough explanation of how to obtain the derivatives of the dynamics in the CasADi-framework, using either `cs.jacobian`, or `cs.jtimes`. Thus, this user example provides all the necessary information and resources a new user needs to understand how to use the library. As previously mentioned, it can be found in the following directory of the library's Github repository:

```
urdf2casadi
└── examples
    └── user
        └── ur5_example
```

The reader is strongly encouraged to have a look at this user example, as it gives an overview of u2c's final functionality and usage.

## 7.2 Further Work

The project has provided an open-source library with the functionality of turning a URDF chain into CasADi functions of the robot's kinematics and dynamics, where one can further easily obtain the dynamics derivatives if needed. Yet, as mentioned, this version of u2c has some limitations, which are associated with the limitations of the urdf_parser_py and the URDF.

### 7.2.1 Generalization

The URDF represents robots with the structure of a branched kinematic tree. However, urdf_parser_py only supports parsing of kinematic chains, thus limiting u2c to only obtaining kinematics and dynamics of kinematic chains. Hence, u2c should, in the long run, be modified to support branched kinematic trees. This

would include going away from urdf_parser_py and develop another parser that parses the full kinematic trees. The algorithms would also have to be modified, but this should be a simple process only involving going away from the assumption that $\lambda_i = i - 1$ and rather use parent arrays. This generalization will not be prioritized in the nearest future but is something that should be modified in the long run.

### 7.2.2 URDF 2.0

The ROS community has already started the discussion of the need for a URDF update. Several forums discuss the construction of a URDF 2.0, where the need for support for closed kinematic chains is in focus. If or when this new version of the URDF is provided, this provides the possibility to expand the functionality of u2c to closed kinematic chains, such as Stewart platforms or delta robots. However, dealing with closed kinematic chains would require significant modification of the algorithm implementations. Thus, this concern should be evaluated at the time of relevance.

## 7.3 Conclusion

In this report, an overview of the research and investigations regarding the development of u2c has been given. This has included a brief presentation of state-of-the-art libraries providing robot kinematics and dynamics, and research concerning the implementation of rigid body dynamics. Further, implementation details of u2c have been presented, and in-depth investigations of the numerical results and timing results for the dynamics and their derivatives have been given.

In the examination of existing libraries it was found that although there is a suite of libraries that generate robot kinematics and dynamics, these are numerical with the exception of SymPyBotics. Even though SymPyBotics does provide symbolic solutions, the framework uses symbolic differentiation, while CasADi uses algorithmic differentiation. In addition, CasADi exploits sparsity, which combined with the algorithmic differentiation makes it a useful framework to obtain fast derivatives of rigid body dynamics for control and optimization. u2c will be especially useful for robotics researchers working with CasADi, but can also be valuable in cases where C code of the symbolic expressions are needed.

The research concerning the implementation of rigid body dynamics algorithms led to the investigation of spatial vector algebra. An introduction to spatial vector algebra for use with rigid body dynamics has been given, and it is showed how the utilization of spatial algebra yield easy and readable code with few code lines and joint type independent algorithms.

The numerical results verified the implementation of the algorithms. The investigation yielded very similar results between u2c, KDL, and RBDL. An inaccuracy related to decimal numbers for PyBullet was also discovered, and has now been fixed by the developers of PyBullet.

Further, the timing investigations showed that, although u2c uses specialized

data types, the evaluation times of u2c are the same order of magnitude as the evaluation times of the numerical libraries. It was found that, due to the overhead related to the CasADi variables and the minimal overhead related to the number of operations, u2c is relatively faster for robots with a high number of DOF. For the same reasons, u2c is relatively faster for the dynamics functions encompassing few input variables.

The timing investigation of the dynamics derivatives yielded evaluation times not much longer than the evaluation times of the dynamics functions themselves. It was also demonstrated how `cs.jtimes` is advantageous when finding the time derivative of the dynamics, saving operations and thus shortening the evaluation times.

To summarize, this report has presented u2c: an open-source library for finding a robot's dynamics and kinematics, and their related derivatives, symbolically in the CasADi framework based on a URDF description of the robot. The functions returned by u2c are numerically accurate and have fast evaluation times, making them suitable for further work within robotics research. The work is summarized in an article accepted to the International Conference on Control, Mechatronics, and Automation (ICCMA), and will be presented at ICCMA 2019, the 7th International, in TU Delft, Netherlands on November 6-8, 2019.

# Appendix

# Robot Dynamics with URDF & CasADi

Lill Maria Gjerde Johannessen[1], Mathias Hauan Arbo[1], and Jan Tommy Gravdahl[1]

*Abstract*— **Fast, accurate evaluation of the dynamic parameters is a key ingredient for accurate control, estimation, and simulation of robots. As these are time-consuming to compute by hand, a software library for generating the rigid body dynamics symbolically can be of great use for robotics researchers. In this paper, we propose a library to efficiently compute and evaluate robot dynamics and its derivatives. Based on a URDF description of the robot's kinematics, three major rigid body dynamics algorithms are used to retrieve the dynamics symbolically in the CasADi framework. To validate the numerical accuracy, the numerical evaluation of the solutions are compared against three other well-established rigid body dynamics libraries, namely RBDL, KDL, and PyBullet. We conduct a timing comparison between the libraries, and we show that the evaluation times of the symbolic expressions are at most one order of magnitude higher than the evaluation times of the numerical libraries. Last, it is shown that the evaluation times of the dynamics derivatives remain of the same order as the evaluation times of the dynamics expressions.**

## I. Introduction

Defining advanced feedback control techniques for robots requires the use of kinematics and dynamics. Oftentimes one requires both the forward or inverse mapping and its derivatives. These can be tedious to compute by hand, and many symbolic solver systems result in functions that have long evaluation time, making them impractical for use in feedback control.

Robotic Operating System (ROS) [1] is a software solution with a growing community among robotics programmers. ROS presents a Universal Robot Description Format (URDF), an XML file describing the robot's kinematics as a kinematic tree of frames with inertial, collision, and visual properties.

A library for using symbolic equations that is growing in popularity among robotics researchers is CasADi [2]. It is an open-source tool for algorithmic differentiation (AD) and numerical optimization. This framework provides the ability to rapidly prototype optimization algorithms and symbolic equations that are close to production ready.

In this paper, we present *urdf2casadi* (u2c), a software library for obtaining functions of the robot's dynamics that can be used with symbolic expressions in the CasADi framework, based on a URDF description of the robot. The library provides forward and inverse dynamics, as well as the Coriolis and gravitational terms, and the inertia matrix. The library is implemented in Python for a cross-platform functionality, and the functions returned by u2c use CasADi's autogenerated C-code to minimize evaluation time.

The library also represents an opportunity to efficiently obtain the derivatives of the dynamics. In standard approaches for controlling complex robotic systems, such as trajectory optimization and optimal control [3, 4, 5], the system dynamics and their derivatives are a crucial part of the optimization problems. Yet, a large amount of the computational time of the optimization algorithms [6] is spent on computing these derivatives.

There are several approaches for evaluating the derivatives. Finite differences is considered the simplest method, but it is sensitive to rounding errors [6], and is dependent on fine parallelization [7, 8]. Further, the manual process of deriving the derivatives is both complex and error-prone, and the analytic derivatives are often difficult to optimize and implement efficiently. By obtaining the dynamics expressions with u2c, the derivatives are easily obtainable using CasADi's built-in Jacobian functionality. Similar to the use of AutoDiff in [6], CasADi uses AD to obtain the derivatives, and the method for calculating the Jacobian involves advanced algorithms exploiting sparsity and symmetry patterns [2]. The result yields efficient calculation of derivatives, making them appropriate for use in robotics research.

### A. Rigid Body Dynamics Libraries

Due to its importance in robotics research, there are several libraries for generating robot dynamics based on a URDF. The *Kinematics and Dynamics Library* [9] (KDL) contains special object types and functions so that one can take a kinematic chain and evaluate the dynamic parameters, i.e the inertia matrix, the Coriolis matrix, and the gravitational force. The routines are real-time safe and implemented in C++.

The *Rigid Body Dynamics Library* (RBDL) [10] is a C++ library inspired by the algorithms presented in Featherstone's *Rigid Body Dynamics Algorithms* [11]. RBDL provides forward dynamics, inverse dynamics, and the dynamic parameters.

PyBullet [12] is an open-source collision detection and rigid body dynamics library, mainly used for physics simulation, robotics, and deep reinforcement learning. It provides the inverse dynamics, and the dynamic parameters.

The aforementioned libraries are numerical. This restricts the user to the built-in function rather than being able to take as many partial derivatives as necessary for the controller formulation. However, it is chosen to compare u2c against these libraries for several reasons. KDL is a well-established library in the ROS community, PyBullet is

[1] Lill Maria Gjerde Johannessen (lmjohann@stud.ntnu.no), Mathias Hauan Arbo, and Jan Tommy Gravdahl are with the Department of Engineering Cybernetics, Norwegian University of Science and Technology (NTNU), Trondheim, Norway

widely used within machine learning, and RBDL is, similar to u2c, implemented based on [11]. Further, these libraries all provide Python bindings and URDF loadings, similar to u2c.

### B. Rigid Body Dynamics

The dynamics of a multi-body rigid system can be described by the equation of motion [13]:

$$\tau = M(q)\ddot{q} + C(q,\dot{q})q + G(q) - \sum_i J_i(q)^T f_i^{ext} \quad (1)$$

where $M$ is the inertia matrix giving the relationship between the generalized joint forces, $\tau$, and the generalized joint accelerations, $\ddot{q}$. $C$ is the Coriolis matrix, encompassing the Coriolis effects, and $G$ encompasses the gravitational forces. For brevity, the dependent variables of $M$, $C$, and $G$ are omitted henceforth and the generalized joints are referred to as joints.

The inverse dynamics (ID) is defined as the joint torques required for the joints to produce a desired joint acceleration for a given joint position, velocity, and external forces:

$$\tau = \mathrm{ID}(model, q, \dot{q}, \ddot{q}, f^{ext}). \quad (2)$$

Similarly, the forward dynamics (FD) is defined as the joint acceleration according to a joint position, torque input and external forces:

$$\ddot{q} = \mathrm{FD}(model, q, \dot{q}, \tau, f^{ext}). \quad (3)$$

u2c provides the forward and inverse dynamics, as well as the Coriolis matrix, the gravitational effects and the inertia matrix using three rigid body dynamics algorithms. The *recursive Newton-Euler algorithm* (RNEA) is used to obtain the inverse dynamics, the Coriolis matrix, and the gravitational force. The *articulated body algorithm* (ABA) is used to obtain the forward dynamics, and the *composite rigid body algorithm* (CRBA) is used to obtain the inertia matrix. The algorithms are implemented using spatial algebra, as presented by Featherstone [11].

## II. Spatial Notation

To give insight into the algorithms, this section gives a brief introduction to spatial vector algebra. Spatial vectors are 6D vectors that contain the linear and angular characteristics of rigid body motion and forces. Spatial vector algebra provides a compact notation to study the dynamics of a multi-body rigid system. A spatial vector contains the information of two 3D vectors and thus replace two or more 3D equations. Hence, dynamics algorithms can be derived quickly and expressed in a compact form leading to efficient computer code.

*1) Spatial Transforms:* The placement of an isolated body $B_i$ with a fixed frame $i$ relative to a world frame 0 is described by a spatial transform denoted ${}^iX_0$. When dealing with multi-body rigid systems, one has to consider several bodies connected through joints. The connecting joint thus represents a constraint for the relative placement between the connection bodies. This constraint is expressed through a quantity referred to as the joint motion matrix, denoted $S_i$. When finding the transform from a child body, denoted $\lambda_i$, to its parent body, $i$, one must consider the placement from the parent body to the connecting joint, $X_J$, and the placement from the joint to the child body, $X_T$. The total transform between two bodies are hence given by ${}^iX_{\lambda_i} = X_J X_T$. The quantity $X_T$ is a fixed placement given by the robot's kinematics, while $X_J$ varies with the joint motion constraint represented by $S_i$.

*2) Spatial Inertia:* When a rigid body has mass, the spatial inertia tensor at an origin $O$ is given by:

$$I_O = \begin{pmatrix} \bar{I}_C + mc\times c\times^T & mc\times \\ mc\times^T & m1 \end{pmatrix}, \quad (4)$$

where $m$ is the mass of the body, $c$ is the body's center of mass, 1 is the identity matrix, and $\times$ represent a spatial cross product operator, which is further explained in the latter. The upper left element of the inertia matrix ($\bar{I}_C + mc\times c\times^T$) is the rotational inertia of the body around $O$. An important advantage of spatial inertia is that if one has to find the total inertia of several bodies, it becomes the sum of all rigid body inertias. To illustrate, if two bodies, having inertia $I_1$ and $I_2$, are rigidly connected and form a composite body, then the total inertia becomes $I_{tot} = I_1 + I_2$.

*3) Motion and force vectors:* Featherstone [11] distinguishes between two groups of spatial vectors, namely *motion* vectors and *force* vectors. Spatial velocity and acceleration belong to the motion group, and spatial force and momentum belong to the force group. Quantities of motion vectors are generally denoted by $m$ and quantities of force vectors are denoted $f$. We distinguished between spatial force transforms and spatial motion transforms such that:

$$m_A = {}^AX_B m_B, \quad (5)$$

$$f = {}^AX^* f_B, \quad (6)$$

where $A$ and $B$ represent two Cartesian frames, ${}^AX_B$ represent the motion transform, and ${}^AX^*_B$ represent the force transform. The relationship between the motion and force transform is that one is the inverse transpose of the other:

$${}^BX^*_A = {}^BX_A^{-T}. \quad (7)$$

The spatial inertia can be seen as a mapping between the two groups as the spatial momentum is given by $h_i = I_i v_i$ and spatial force is given by $f_i = I_i a_i + v_i \times^* h_i$. The spatial force thus correspond to the to the time derivative of the spatial momentum.

Motion vectors can also operate on both motion and force vectors through spatial cross product operators. They are similar to classic time derivatives, and it is distinguished between spatial motion and force cross products:

$$\dot{m} = v_A \times m_B, \quad (8)$$

$$\dot{f}_A = v_A \times^*_B f_B. \quad (9)$$

$\times^*$ can be viewed as the dual of $\times$, and their relationship is similar to the relationship between $X$ and $X^*$.

Spatial velocity is defined as the time derivative of the spatial transform. Thus, the spatial joint velocity, denoted $v_J$, is found by taking the derivative of the aforementioned quantity $X_J$:

$$v_J = \frac{\partial X_J}{\partial q_i}\dot{q} = S_i\dot{q}_i, \tag{10}$$

where $S_i$ is the aforementioned joint motion matrix. $S_i$ is considered time independent for 1-DOF joints.

Last, the joint acceleration is defined as the derivative of the joint velocity:

$$\begin{aligned} a_{J\,i} &= S_i\ddot{q}_i + \dot{S}_i\dot{q}_i \\ &= S_i\ddot{q}_i + v_i \times S_i\dot{q}_i \\ &= S_i\ddot{q}_i + v_i \times v_{J\,i}. \end{aligned} \tag{11}$$

## III. Rigid Body Dynamics Algorithms

### A. The Recursive Newton-Euler Algorithm

Although various algorithms have been proposed to retrieve ID, RNEA remains the most efficient, whose complexity is $O(n)$ with n being the number of bodies of the robotic system. RNEA was first proposed by [14], and was later renewed by [11] to exploit the advantages of spatial algebra. Compared to [11], u2c explicitly calculates model quantities, i.e ${}^iX_{\lambda_i}$, $S_i$, $I_i$, beforehand in a model calculation routine, and ${}^iX^*_{\lambda_i}$ is found using (7). This accounts for the other algorithm implementations as well. Algorithm 1 shows the compact result of using spatial algebra. As can be observed, RNEA is a two-pass algorithm which propagates the kinematic quantities in a forward pass, followed by retrieving the joint forces in a backward pass.

Although RNEA was originally developed to obtain ID, u2c exploits modifications of RNEA to obtain the Coriolis and gravitational terms as these are subsets of the inverse dynamics problem. ID can be viewed as:

$$ID = \text{RNEA}(model, q, \dot{q}, \ddot{q}, f^{ext}), \tag{12}$$

while $C$ and $G$ can be obtained by:

$$C = \text{RNEA}(model, q, \dot{q}, 0, 0), \tag{13}$$

$$G = \text{RNEA}(model, q, 0, 0, 0). \tag{14}$$

**Algorithm 1** Recursive Newton-Euler Algorithm

**Input:** $X$, $S$, $I$
**Output:** $\tau$

$v_0 = 0$
$a_0 = -a_g$
**for** $i = 1$ **to** $n_B$ **do**
  $v_i = {}^iX_{\lambda_i}v_{\lambda_i} + S_i\dot{q}_i$
  $a_i = {}^iX_{\lambda_i}a_{\lambda_i} + S_i\ddot{q}_i + v_i \times S_i\dot{q}_i$
  $f_i = Ia_i + v_i \times^* Iv_i$ - ${}^iX^*_{\lambda_i}f^{ext}_i$
**end for**
**for** $i = n_B - 1$ **to** 0 **do**
  $\tau_i = S_i^T f_i$
  **if** $\lambda_i \neq 0$ **then**
    $f_{\lambda_i} = f_{\lambda_i} + {}^{\lambda_i}X^*_i f_i$
  **end if**
**end for**

### B. The Articulated Body Algorithm

One of the most efficient algorithms for computing the forward dynamics is ABA, whose complexity is, similar to RNEA, $O(n)$. The algorithm was first presented by [15] although various versions have been proposed since then. ABA does not rely on computing the inverse of the inertia matrix, but is generally more complex than RNEA. It is composed of three main passes, whereas the first pass is a forward recursion collecting the spatial forces acting on the bodies. The quantities obtained in the first pass are used to retrieve the articulated body inertia and the articulated body forces in a backward pass, followed by obtaining the joint and body accelerations in a forward recursion.

### C. The Composite Rigid Body Algorithm

CRBA is used to compute the inertia matrix. The physical interpretation of $M$ is that it relates the force acting on each joint to the acceleration of each joint. By using the definition of the kinetic energy of each body, while treating the bodies as composite rigid bodies, the algorithm recursively calculates each element of the matrix. For further details about CRBA and ABA, we refer to Featherstone's Rigid Body Dynamics Algorithms [11].

CRBA can also be used to solve the forward dynamics problem. [11] presents the equation of motion for a multi-body rigid system as:

$$\tau = M(q)\ddot{q} + C_2(q, \dot{q}, f^{ext}) \tag{15}$$

where $C_2$ encompasses the Coriolis effects, the gravitational force, and the effects of external forces, if any. This quantity can, similar to the Coriolis matrix, be found by a call to RNEA where $\ddot{q} = 0$ and the external and gravitational forces are considered. Thus, FD can be found by solving the equation for $\ddot{q}$:

$$\ddot{q} = M^{-1}(\tau - C_2), \tag{16}$$

where the dependencies of $M$ and $C_2$ are omitted. It should also be mentioned that this method for finding FD has a worst case of $O(nd^2)$ with $d$ being the depth of the kinematic tree. In cases where the kinematic tree contains few bodies, these algorithms can exceed the speed of $O(n)$-algorithms. Thus, this approach has also been implemented in u2c, and both approaches for FD are evaluated in the next section.

## IV. Results

In this section, the performance of u2c with regard to numerical accuracy, efficiency of evaluating the dynamics expressions, and its derivatives, are reported. The tests are performed on a Ubuntu 16.04, 3.5 GHz x 12 Intel Xeon CPU processor in a Python 2.7 environment, and the -OFast compiler flag is used for optimization of the generated C-code. Various robots are used to evaluate the performance: a 2-DOF pendulum, a 6-DOF UR5, and a 16-DOF snake robot.

TABLE I: Numerical differences between libraries for a 6-DOF UR5 robot for 1000 random samples.

| | KDL/u2c | RBDL/u2c | PyBullet/u2c |
|---|---|---|---|
| $G$ $(N)$ | $4.42 \cdot 10^{-12}$ | $1.96 \cdot 10^{-07}$ | $1.83 \cdot 10^{-03}$ |
| $C$ $(N)$ | $1.03 \cdot 10^{-11}$ | $4.30 \cdot 10^{-07}$ | $3.12 \cdot 10^{-03}$ |
| ID $(Nm)$ | | $4.41 \cdot 10^{-07}$ | $4.12 \cdot 10^{-03}$ |
| $M$ $(kgm^2)$ | $1.40 \cdot 10^{-12}$ | $1.46 \cdot 10^{-08}$ | $2.32 \cdot 10^{-03}$ |
| FD $(m/s^2)$ | | $7.88 \cdot 10^{-07}$ | |

### A. Numerical Results

The numerical results for the UR5, displayed in Table I, are obtained by generating 1000 samples of configurations, velocities, and accelerations or torques, uniformly distributed within the joint limits. The result shows that u2c and KDL have very similar results, implying a numerical difference of factor $10^{-15}$ for a single sample of $M$ and $G$. For $C$, the numerical difference is one order of magnitude higher. This increase, from $G$ to $C$, is due to the extra variable that has to be considered in the RNEA calculations, namely $\dot{q}$, when evaluating $C$. One can also observe that the numerical differences between RBDL and u2c, thus also between KDL and RBDL, is of factor $10^{-07}$ and $10^{-08}$. This indicates a numerical difference of at most $10^{-10}$ for a single sample, which can be considered satisfactory. The similarity between KDL and u2c may be due to more similar data types. Further, the numerical results imply a numerical difference of $10^{-06}$ per sample, between PyBullet and the remaining libraries, which may indicate a small inaccuracy for PyBullet. It is speculated that the error is a matter of single versus double precision floats, or a floating point cancellation issue. The developers of PyBullet have been informed about this, but the numerical difference is negligible in most use cases.

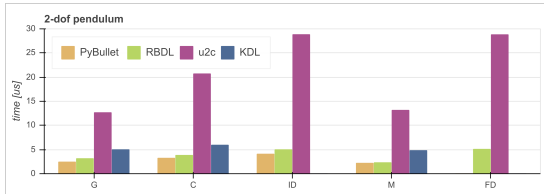
The numerical tests for the 2-DOF pendulum and the 16-DOF snake [16] yielded the same result as for the 6-DOF UR5. Hence, one can assume that the numerical accuracy is independent of the number of DOF.

### B. Timing Results

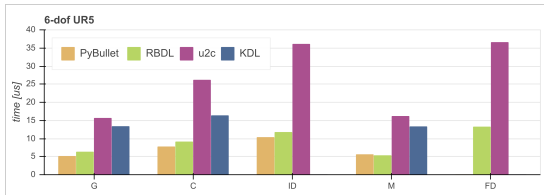
TABLE II: Summary table of number of operations for the dynamics expressions.

| | pendulum | ur5 | snake |
|---|---|---|---|
| G | 18 | 199 | 224 |
| C | 79 | 648 | 572 |
| ID | 110 | 696 | 710 |
| M | 34 | 872 | 1217 |
| FD (CRBA) | 132 | 2354 | 14786 |
| FD (ABA) | 142 | 1948 | 2272 |

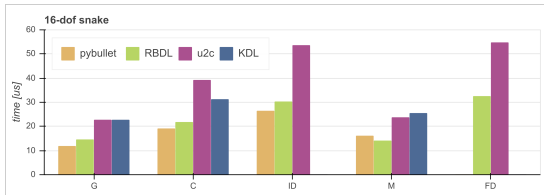
For obtaining the timing results, random configuration, velocity, and acceleration or force vectors are uniformly sampled within the joint limits. Measurements of 1000 samples are stored and the median times spent on evaluating the dynamics are listed in Figure 1. The results show that u2c overall uses a longer evaluation time compared to PyBullet, RBDL, and KDL. This is a natural consequence of u2c generated functions supporting symbolic data types, and thus some overhead is expected. Hence, the functions that only require one symbolic variable, i.e $G$ and $M$, have less overhead than the functions that require two or three symbolic variables, i.e $C$, ID, and FD, resulting in shorter evaluation times for functions of fewer symbolic variables.




(a) Median evaluation times for pendulum.



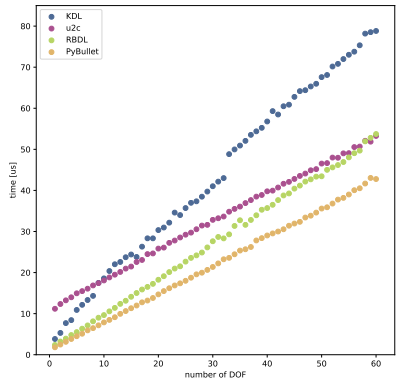

(b) Median evaluation times for UR5.



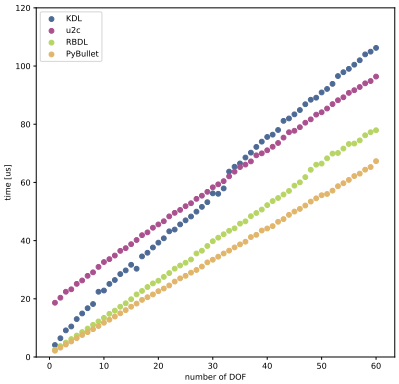

(c) Median evaluation times for snake.

Fig. 1: Median evaluation times for $G$, $C$, ID, $M$, and FD for PyBullet, RBDL, u2c, and KDL.

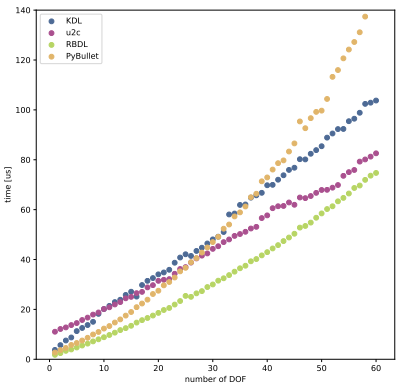
u2c is at most one order of magnitude slower than the numerical libraries, which is the case for the 2-DOF pendulum. While for the 16-DOF snake, u2c and the numerical libraries yield evaluation times of the same order of magnitude. By evaluating Table IV and Figure 1, it is clear that the evaluation times of u2c are mostly affected by the number of symbolic variables, and are less sensitive to the number of operations in the expression. To illustrate, ID and FD(aba) encompass three symbolic variables and are both obtained with $O(n)$-algorithms. FD for the snake requires 1562 more operations than ID. Yet, the difference in evaluation time is only a few microseconds. Hence, it is reasonable to assume that the increase in evaluation time from $G$ and $M$, to $C$, and up to ID and FD is mostly due to the additional symbolic variables required. From Figure 1 it is also seen that the dynamics expressions with a higher number of symbolic variables are more affected by an increase in the number of DOF. This is reasonable as this leads to symbolic vectors with a higher number of elements.




(a) Median evaluation times for $G$.




(b) Median evaluation times for $C$.




(c) Median evaluation times for $M$.

Fig. 2: Median evaluation times for robots of 1 to 60 DOF.

As a result of the CasADi functions being insensitive to an increase in number of operations, and thus also to an increase in DOF, one can observe from Figure 1 that the overall difference in evaluation time between u2c and the numerical libraries decreases as the number of DOF increases. Due to this finding, it was wanted to investigate how the evaluation times evolved for a higher increase in DOF. Thus, we have constructed 60 experimental URDFs and the evaluation times for $G$, $C$, and $M$ from 1 DOF to 60 DOF are shown in Figure 2. For $G$, it is observed that even though u2c starts out as the library with the slowest evaluation times, it ends up being faster than KDL and RBDL. Further, one can observe that the overhead related to an additional variable makes $C$ slower and more sensitive to an increase in DOF, compared to $G$. Yet, u2c still exceeds KDL for robots over 30 DOF and remain the same order of magnitude as RBDL and PyBullet. For $M$ it is shown that u2c exceeds KDL for robots over 10 DOF, and PyBullet for robots over 20 DOF. Hence, we have shown that for the dynamics functions that only encompass a few symbolic variables, the total increase in overhead related to u2c is smaller than the increase in overhead for several of the numerical libraries. This is partly due to the minimal overhead related to the increase in operations, provided by CasADi [2]. For ID and FD, the overhead associated with three symbolic variables prevents u2c from exceeding the numerical libraries, but the evaluation times remain the same order of magnitude as the numerical libraries.

### C. Derivatives Timing

TABLE III: Summary table of median evaluation times for dynamics the derivatives with respect to $q$, $\dot{q}$, $\ddot{q}$ and $\tau$.

| | pendulum | UR5 | snake |
|---|---|---|---|
| derivatives of $G$ | 12.63 µs | 15.38 µs | 23.47 µs |
| derivatives of $C$ | 20.57 µs | 29.62 µs | 46.78 µs |
| derivatives of ID | 28.91 µs | 40.08 µs | 67.68 µs |
| derivatives of $M$ | 12.54 µs | 17.14 µs | 36.72 µs |
| derivatives of FD (crba) | 29.07 µs | 48.35 µs | 286.73 µs |
| derivatives of FD (aba) | 29.10 µs | 50.52 µs | 82.48 µs |

TABLE IV: Summary table of number of operations for the dynamics derivatives.

| | pendulum | UR5 | snake |
|---|---|---|---|
| derivatives of $G$ | 29 | 881 | 2347 |
| derivatives of $C$ | 239 | 5157 | 11792 |
| derivatives of ID | 234 | 4731 | 16159 |
| derivatives of $M$ | 44 | 4095 | 11635 |
| derivatives of FD (crba) | 378 | 17764 | 431178 |
| derivatives of FD (aba) | 390 | 13517 | 40901 |

The derivatives are easily obtained using CasADi's built-in Jacobian functionality, allowing the user to explicitly define which variable one wishes to find the derivative with respect to.

The results, summarized in Table III and Figure 3, show that the evaluation times of the derivatives are not much

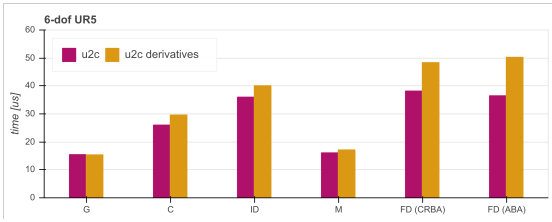



Fig. 3: Median evaluation times of the dynamics and the dynamics derivatives for a UR5.

longer than the dynamics expressions themselves. Figure 3 displays this graphically for a UR5. The evaluation times of the derivatives are the same order of magnitude as the evaluation times of the derivatives, the only exception being the derivative of FD using CRBA for the 16-DOF snake as it must invert a very large inertia matrix.

Table IV shows the number of operations for the dynamics derivatives expressions. As the dynamics functions and their related derivative functions contain the same number of symbolic variables, the change in evaluation time can be assumed to be mainly due to the increase in operations. The evaluation time of the derivative of $G$ for the UR5 increases with 617 operations, which does not remarkably affect the evaluation time. FD using CRBA is the method that is most exposed to an increase in number of operations, as it has a complexity of $O(nd^2)$ and requires three symbolic variables. The derivative of FD using CRBA for the snake increases with 445647 operations, leading to a 217.81 µs increase in evaluation time. This indicates 0.49 µs longer evaluation time for an increase of 1000 operations, and substantiates the fact that the evaluation times of CasADi functions are not heavily affected by an increase in operations. It is the number of symbolic variables involved that is most essential in this matter.

## V. Conclusion

The paper has proposed a software library to efficiently compute the dynamics expressions of a robot based on a URDF description of it's kinematics. To achieve this, we have combined the CasADi framework with the implementation of rigid body dynamics algorithms using spatial algebra. By combining RNEA, ABA, and CRBA, urdf2casadi provides the inverse and forward dynamics expressions, as well as the Coriolis and gravitational terms, and the inertia matrix. Our approach leads to efficient algorithms, resulting in evaluation times comparable to numerical approaches represented by KDL, RBDL, and PyBullet.

By extracting the expressions in the CasADi framework, the dynamics derivatives are easily obtainable using CasADi's Jacobian functionality. We have demonstrated that the evaluation times of the dynamics derivatives are of the same order of magnitude as the evaluation times of the dynamics expressions, thus making them suitable for use within trajectory optimization, optimal control, and other approaches where dynamics derivatives are needed.

We provide the complete open-source Python implementation of this library, thus providing a multi-platform, easily installable library ideal for use in robotics research. More details can be found in Johannessen [17].

## Acknowledgment

The work reported in this paper was based on activities within centre for research based innovation SFI Manufacturing in Norway, and is partially funded by the Research Council of Norway under contract number 237900.

## References

[1] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source robot operating system," in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, (Kobe, Japan), May 2009.

[2] 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*, In Press, 2018.

[3] M. H. Arbo, E. I. Grtli, and J. T. Gravdahl, "On model predictive path following and trajectory tracking for industrial robots," in *2017 13th IEEE Conference on Automation Science and Engineering (CASE)*, pp. 100–105, Aug 2017.

[4] D. Verscheure, M. Diehl, J. De Schutter, and J. Swevers, "On-line time-optimal path tracking for robots," in *2009 IEEE International Conference on Robotics and Automation*, pp. 599–605, May 2009.

[5] 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, 2014.

[6] J. Carpentier and N. Mansard, "Analytical Derivatives of Rigid Body Dynamics Algorithms," in *Robotics: Science and Systems (RSS 2018)*, (Pittsburgh, United States), June 2018.

[7] 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*, pp. 4906–4913, Oct 2012.

[8] J. Koenemann, A. Del Prete, Y. Tassa, E. Todorov, O. Stasse, M. Bennewitz, and N. Mansard, "Whole-body model-predictive control applied to the HRP-2 humanoid," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3346–3351, Sep. 2015.

[9] R. Smits, "KDL: Kinematics and Dynamics Library." `http://www.orocos.org/kdl`, 2014. Accessed: 2018-12-10.

[10] M. L. Felis, "RBDL: an efficient rigid-body dynamics library using recursive algorithms," *Autonomous Robots*, pp. 1–17, 2016.

[11] R. Featherstone, *Rigid Body Dynamics Algorithms*. Springer, 2008.

[12] E. Coumans and Y. Bai, "PyBullet, a Python module for physics simulation for games, robotics and machine learning." `http://pybullet.org`, 2016–2018. Accessed: 2018-12-9.

[13] B. Siciliano and K. Oussama, *Handbook of Robotics*. Springer, 2008.

[14] J. Luh, M. Walker, and R. Paul, "Resolved-acceleration control of mechanical manipulators," *IEEE Transactions on Automatic Control*, vol. 25, pp. 468–474, June 1980.

[15] R. Featherstone, "The Calculation of Robot Dynamics Using Articulated-Body Inertias," *The International Journal of Robotics Research*, vol. 2, no. 1, pp. 13–30, 1983.

[16] I.-L. Borlaug, K. Pettersen, and J. Gravdahl, "Trajectory tracking for an articulated intervention AUV using a super-twisting algorithm in 6 DOF.," *IFAC-PapersOnLine*, vol. 51, no. 29, pp. 311 – 316, 2018. 11th IFAC Conference on Control Applications in Marine Systems, Robotics, and Vehicles CAMS 2018.

[17] L. M. G. Johannessen, "Robot Dynamics with URDF & CasADi," Master's thesis, Norwegian University of Technology and Science, Trondheim, Norway, 2019.

# Bibliography

Andersson, J., Åkesson, J., Diehl, M., 01 2012. CasADi: A Symbolic Package for Automatic Differentiation and Optimal Control. Vol. 87.

Andersson, J. A. E., Gillis, J., Horn, G., Rawlings, J. B., Diehl, M., 2018. CasADi – A software framework for nonlinear optimization and optimal control. In Press, Mathematical Programming Computation.

Beirami, A., Macnab, C. J. B., May 2006. Direct Neural-Adaptive Control of Robotic Manipulators using a Forward Dynamics Approach. In: 2006 Canadian Conference on Electrical and Computer Engineering. pp. 363–367.

Borlaug, I.-L., Pettersen, K., Gravdahl, J., 2018. Trajectory tracking for an articulated intervention AUV using a super-twisting algorithm in 6 DOF**This research was funded by the Research Council of Norway through the Centres of Excellence funding scheme, project No. 223254 NTNU AMOS. IFAC-PapersOnLine 51 (29), 311 – 316, 11th IFAC Conference on Control Applications in Marine Systems, Robotics, and Vehicles CAMS 2018.

Carpentier, J., Mansard, N., Jun. 2018. Analytical Derivatives of Rigid Body Dynamics Algorithms. In: Robotics: Science and Systems (RSS 2018). Pittsburgh, United States.

Chandramouli, A., Manivannan, P. V., April 2018. Inverse dynamics of different upright postures for the developed bio-inspired reconfigurable robot. In: 2018 3rd International Conference on Control and Robotics Engineering (ICCRE). pp. 31–36.

Coumans, E., Bai, Y., 2016–2018. PyBullet, a Python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`, accessed: 2018-12-9.

Dasari, A., Reddy, N. S., Dec 2012. Forward and inverse kinematics of a robotic frog. In: 2012 4th International Conference on Intelligent Human Computer Interaction (IHCI). pp. 1–5.

Featherstone, R., 2007. Robot dynamics. Scholarpedia 2 (10), 3829, revision #91723.

Featherstone, R., 2008. Rigid Body Dynamics Algorithms. Springer.

Featherstone, R., June 2012. spatial_v2 webpage. `http://royfeatherstone.org/spatial/v2/#mcI`, accessed: 2018-12-12.

Felis, M. L., 2016. RBDL: an efficient rigid-body dynamics library using recursive algorithms. Autonomous Robots, 1–17.

Koenemann, J., Del Prete, A., Tassa, Y., Todorov, E., Stasse, O., Bennewitz, M., Mansard, N., Sep. 2015. Whole-body model-predictive control applied to the HRP-2 humanoid. In: 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 3346–3351.

Kucuk, S., Bingul, Z., 10 2006. Link Mass Optimization of Serial Robot Manipulators Using Genetic Algorithm. Vol. 4251. pp. 138–144.

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Roučka, v., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., Scopatz, A., Jan. 2017. SymPy: symbolic computing in Python. PeerJ Computer Science 3, e103.

Posa, M., Cantu, C., Tedrake, R., 2014. A direct method for trajectory optimization of rigid bodies through contact. The International Journal of Robotics Research 33 (1), 69–81.

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A., May 2009. ROS: an open-source robot operating system. In: Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics. Kobe, Japan.

Richert, D., Beirami, A., Macnab, C. J. B., Feb 2000. Neural-adaptive control of robotic manipulators using a supervisory inertia matrix. In: 2009 4th International Conference on Autonomous Robots and Agents. pp. 634–639.

Sansanayuth, T., Nilkhamhang, I., Tungpimolrat, K., Aug 2012. Teleoperation with inverse dynamics control for PHANToM Omni haptic device. In: 2012 Proceedings of SICE Annual Conference (SICE). pp. 2121–2126.

Siciliano, B., Oussama, K., 2008. Handbook of Robotics. Springer.

Smits, R., 2014. KDL: Kinematics and Dynamics Library. `http://www.orocos.org/kdl`, accessed: 2018-12-10.

Song, D. H., Jung, S., Aug 2007. Geometrical Analysis of Inverse Kinematics Solutions and Fuzzy Control of Humanoid Robot Arm under Kinematics Constraints. In: 2007 International Conference on Mechatronics and Automation. pp. 1178–1183.

Sousa, C. D., Aug. 2014. SymPyBotics v1.0. Accessed: 2018-12-10. URL https://doi.org/10.5281/zenodo.11365

Tassa, Y., Erez, T., Todorov, E., Oct 2012. Synthesis and stabilization of complex behaviors through online trajectory optimization. In: 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 4906–4913.

Verscheure, D., Demeulenaere, B., Swevers, J., De Schutter, J., Diehl, M., 2009. Time-optimal path tracking for robots: A Convex Optimization Approach. Automatic Control, IEEE Transactions on 54, 2318 – 2327.