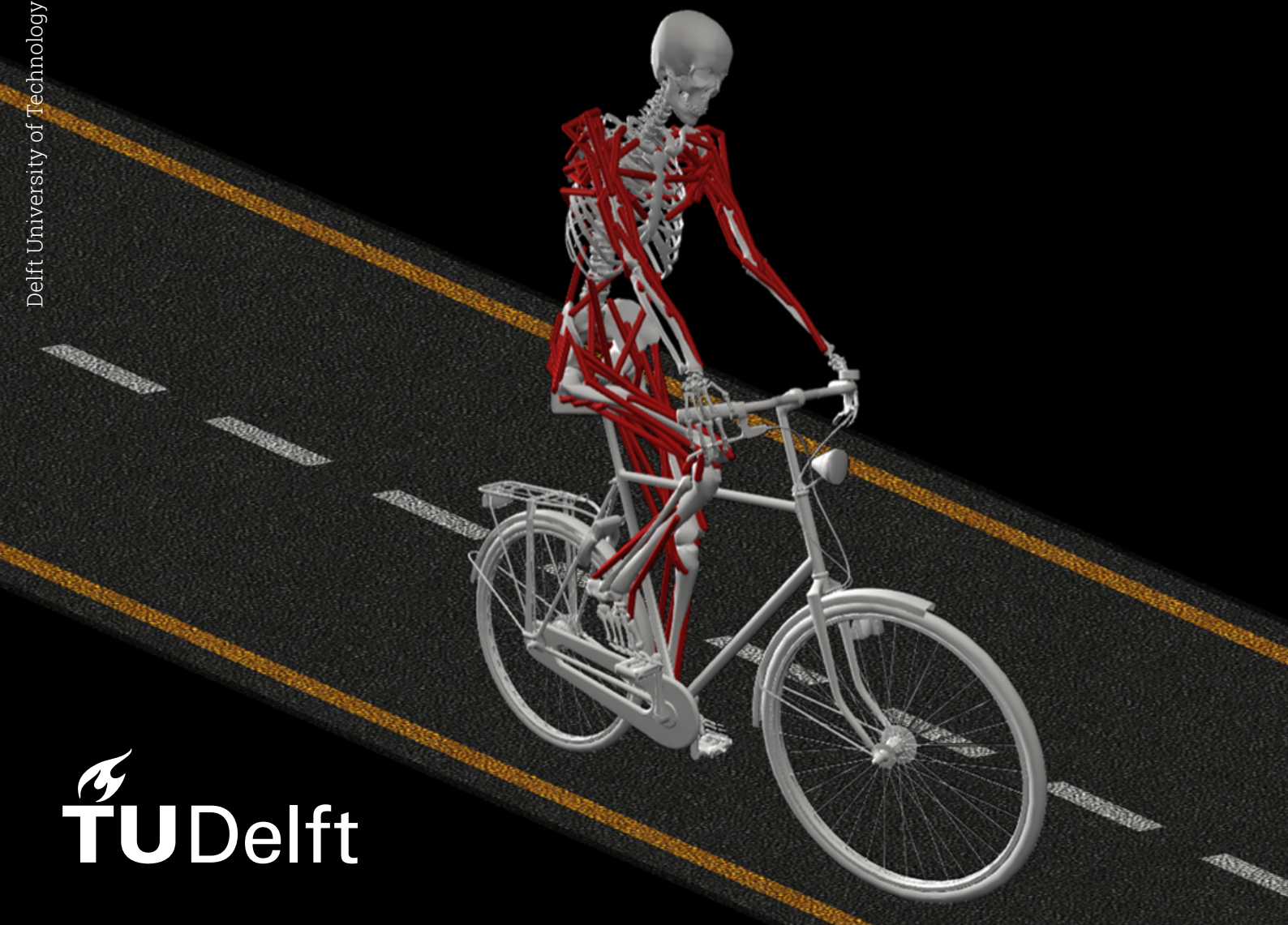
BRiM: A Modular Bicycle-Rider Modeling Framework

ME54035: ME-BMD MSc Thesis

Timo Stienstra



Delft University of Technology

TU Delft

BRiM: A Modular Bicycle-Rider Modeling Framework

by

Timo Stienstra

Student number:	4939476	
Project Duration:	01, 2023 - 08, 2023	
Supervisors:	Dr. S. G. Brockie,	BMechE
	Dr. J. K. Moore,	BMechE
Committee:	Dr. S. G. Brockie,	BMechE
	Dr. R. Happee	
	Dr. J. K. Moore,	BMechE

Cover image: Created by Timo Stienstra using *OpenSim*

TU Delft

Acknowledgments

First of all, I would like to express my profound gratitude to Sam Brockie, my supervisor, for his active guidance throughout this entire project. With the many conversations and code reviews he has really helped me bring this project to the next level. Besides the (too) many conversations about *BRiM*, *SymPy*, and this report, it was also fun and a great learning experience to see the problems he has been working on in *SymPy*. Also, the explanation and help on using *Pycollo*, a Python package he developed, was extremely helpful.

I owe great thanks to Jason Moore, my other supervisor, who helped me get started. By teaching the multibody dynamics course he made me enthusiastic about using *SymPy* to solve dynamics problems. After which he supervised me during my Google Summer of Code within *SymPy*. I also owe him the initial idea for this project and the freedom to investigate where my interests wandered off to. Also, his unrelenting critique and ideas gave me insight into how to improve this report and fix several problems along the way.

I am grateful to all the members of the bicycle lab for the fun conversations and valuable discussions throughout the year. I would like to thank Adam Kewley for the lovely conversation on the usage of *OpenSim* and Oscar Benjamin, a lead developer of *SymPy*, for his detailed explanations of the problems and possible solutions when solving linear systems symbolically.

Last of all I am grateful to my family and friends for the support and wonderful time besides study. Above all, I also want to glorify my God, Creator, and Father, Who has given me the ability to study, be creative in solving problems, and stay motivated.

Usage of AI Tools

In recent years the usage of tools using Artificial Intelligence has increased. During this project the large language model ChatGPT-3.5 developed by OpenAI has been utilized in several ways.

ChatGPT has been employed through its integration into GitHub CoPilot. In my experience it is not (yet) able to make good suggestions when writing a unique function. However, it serves well as an advanced version of autocomplete, which speeds up the process of writing boilerplate code and documentation. Additionally, the online chat has been utilized to generate ideas when brainstorming, like figuring out how to name a code variable. Even this process of having to explain something supports your own thought process.

It has also been utilized in rephrasing certain paragraphs. This enhances readability and spots typos, like Grammarly or a regular spell checker, and it introduces a greater variation in language, like Thesaurus. When rephrasing a paragraph I used the following workflow. Initially, I could provide a summarized version of the desired paragraph in bullet points. ChatGPT's response generally is a long text, which is mostly worthless except that I could use its vocabulary to enrich my own. Next, I would write my own version of the paragraph and provide it to ChatGPT to rephrase it correcting possible spelling errors and improving its readability. Based on this proposal I could improve the paragraph and if necessary do a second or third iteration.

Abstract

Bicycles have been studied extensively over the past 200 years, with mathematical models providing valuable insights into various aspects of bicycle dynamics and rider control. However, the lack of a common framework for creating and sharing bicycle-rider models hinders the development of advanced models, research reproducibility, and dissemination. This thesis addresses this gap by introducing *BRiM*[1]: an open-source modular and extensible framework for creating **B**icycle-**Ri**der **M**odels.

The modular setup of *BRiM* relies on a systematic approach to define a model and form the analytical EOMs. For the involved analytical computations *BRiM* utilizes *SymPy* [44], a Computer Algebra System. The systematic approach consists of four stages. The first stage defines the objects in the system, such as symbols and bodies. Secondly, the kinematic relationships between the objects, such as angular velocities between reference frames, are established. The third and the fourth stages, which are order-independent, specify the loads and constraints acting upon the system. The decoupling *BRiM* required to achieve modularity is enabled through this systematic approach, because computations within a stage are mostly order-independent.

The core of *BRiM* employs the systematic approach within a unified framework for modeling mechanical systems in general. It describes a model using a tree representation, in which a model is defined as an aggregation of smaller submodels. The relationships between submodels are established by parent models, using interchangeable connections to accommodate complex relations, such as tyre models between the ground and a wheel. This application of submodels enables swapping and adding submodels, making the overarching model both modular and extensible. Actuation within *BRiM* can either be specified by attaching prespecified groups of loads to models and connections, or by utilizing the interface provided by the mechanics module in *SymPy*, which offers the flexibility to even manipulate equations in detail.

BRiM applies this generalized framework to create modular bicycle-rider models. Both a stationary bicycle and a modular bicycle based on Moore's convention [49] of the Carvallo-Whipple bicycle [11, 90] have been constructed. These bicycle models are extensible to bicycle-rider models by including an upper and/or lower body. Within the rider models each joint can be actuated by a linear torsional spring-damper. *BRiM* integrates parametrization of models, which provides mappings between symbolic quantities used in equations and experimentally determined values, using the existing open-source *BicycleParameters* library [47]. Additionally, *SymMePlot* [79], a visualization package for symbolically defined mechanical systems, has been developed and integrated within *BRiM* to visualize the created bicycle-rider models.

The effectiveness of *BRiM* is demonstrated through optimization and simulation tasks. Firstly, a real-time forward simulation of a torque-driven upper body bicycle-rider is performed. Secondly, an optimization problem is solved, involving the tracking of a rolling disc along a sinusoidal trajectory while minimizing the control torques. These demonstrations highlight the seamless integration of *BRiM* with other scientific tools and *BRiM*'s potential for practical applications.

In conclusion, *BRiM* fills the gap in bicycle dynamics research by providing a modular and extensible framework for creating and sharing bicycle-rider models. Its systematic approach, unified framework, and integration capabilities enable efficient model development, research reproducibility, and further advancement in bicycle research.

[1] *BRiM* is available at: `https://github.com/TJStienstra/brim`.

Contents

List of Figures

Glossaries

Acronyms

API	Application Programming Interface	10, 11, 25
CAS	Computer Algebra System	iii, ix, 3, 4, 45
CSE	Common Subexpression Elimination	8, 30, 31
DCM	Direct Cosine Matrix	6, 7, 10, 43
DOF	Degrees of Freedom	viii, 37
EOMs	Equations of Motion	iii, viii, 1–4, 8, 9, 12, 22, 23, 25, 29–35, 37, 40, 43–45, 54, 55, 57, 63, 64
OCP	Optimal Control Problem	ix, 39, 40
ODE	Ordinary Differential Equation	ix, 54
UML	Unified Modeling Language	vi, 16, 19, 28
URDF	Unified Robot Description Format	10

Symbols

A_{sr}	$\mathbb{R}^{p\times M+m}$ Matrix relating the dependent generalized speeds to the independent.	55
M	Number of holonomic constraints.	vii, viii
N	Number of coordinates, equals the sum of the number of independent and dependent generalized coordinates.	vii, viii
$\boldsymbol{F}_r^*$	$\mathbb{R}^p$ Vector of all generalized inertia forces.	54, 55
$\boldsymbol{F}_r$	$\mathbb{R}^p$ Vector of all generalized active forces.	54, 55
$\boldsymbol{J}_{\boldsymbol{f},\boldsymbol{x}}$	Jacobian of $\boldsymbol{f}$ with respect to $\boldsymbol{x}$, where $\boldsymbol{J}_{\boldsymbol{f},\boldsymbol{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$.	vii, 54, 55
$\boldsymbol{M}_d$	$\mathbb{R}^{N\times N}$ Dynamic mass matrix.	54, 55
$\boldsymbol{M}_k$	$\mathbb{R}^{N\times N}$ Kinematic mass matrix.	54
$\dot{\boldsymbol{f}}_{vc}(\boldsymbol{q},\boldsymbol{u},\dot{\boldsymbol{u}})$	$\mathbb{R}^{M+m}$ Vector of the velocity constraints.	55
$\dot{\boldsymbol{q}}$	First derivative of $\boldsymbol{q}$ to t in $\mathbb{R}^N$.	vii, 54
$\dot{\boldsymbol{u}}_r$	First derivative of $\boldsymbol{u}_r$ to t in $\mathbb{R}^{M+m}$.	55
$\dot{\boldsymbol{u}}_s$	First derivative of $\boldsymbol{u}_s$ to t in $\mathbb{R}^p$.	55
$\dot{\boldsymbol{u}}$	First derivative of $\boldsymbol{u}$ to t in $\mathbb{R}^N$.	vii, 54, 55
$\boldsymbol{f}_d(\boldsymbol{q},\boldsymbol{u},\dot{\boldsymbol{u}})$	Dynamic differential equations, which describe the dynamic behavior of the system.	54
$\boldsymbol{f}_k(\boldsymbol{q},\dot{\boldsymbol{q}},\boldsymbol{u})$	Kinematical differential equations, which relate the generalized speeds to the generalized coordinates.	54
$\boldsymbol{f}_{vc}(\boldsymbol{q},\boldsymbol{u})$	$\mathbb{R}^{M+m}$ Vector of the velocity constraints.	55
$\boldsymbol{g}_d$	$\mathbb{R}^N$ Dynamic forcing vector.	54, 55
$\boldsymbol{g}_k$	$\mathbb{R}^N$ Kinematic forcing vector.	54
$\boldsymbol{q}_{dep}$	$\mathbb{R}^M$ Vector of the dependent generalized coordinates.	viii

$\boldsymbol{q}_{ind}$	$\mathbb{R}^n$ Vector of the independent generalized coordinates.	viii
$\boldsymbol{q}$	$\mathbb{R}^N$ Vector of all generalized coordinates, typically $\begin{bmatrix}\boldsymbol{q}_{ind}\\ \boldsymbol{q}_{dep}\end{bmatrix}$.	vii, 40, 54, 55
$\boldsymbol{u}_r$	$\mathbb{R}^{M+m}$ Vector of the dependent generalized speeds.	vii, viii, 55
$\boldsymbol{u}_s$	$\mathbb{R}^p$ Vector of the independent generalized speeds.	vii, viii, 55
$\boldsymbol{u}$	$\mathbb{R}^N$ Vector of all generalized speeds, typically $\begin{bmatrix}\boldsymbol{u}_s\\ \boldsymbol{u}_r\end{bmatrix}$.	vii, 54, 55
m	Number of nonholonomic constraints.	vii, viii
n	Number of independent generalized coordinates equals $N - M$.	viii
p	Number independent generalized speeds equals $n-m$, also referred to as the number of Degrees of Freedom (DOF) by Kane and Levinson [26].	vii, viii, 55
t	Time.	vii, 37, 40

software

BRiM	A modular and extensible open-source framework for creating bicycle-rider models proposed in this thesis.	i, iii, vi, 1–4, 10, 11, 15, 16, 18, 21–35, 37–40, 42–46, 56, 57, 61
BicycleParameters	An open-source Python library for generating, manipulating and analyzing the physical parameters of a bicycle and rider from experimental measurements [47].	iii, 29, 34, 37
Matplotlib	An open-source Python package for creating visualizations.	viii, 29, 30, 40
MuJoCo	**Mu**lti-**Jo**int dynamics with **Co**ntact is a popular physics engine created by Todorov, Erez, and Tassa [82].	3, 10
NumPy	An open-source Python library for numerical computations providing arrays and functions for efficient mathematical operations.	viii, 37
Numba	An open-source just-in-time compiler that translates a subset of Python and *NumPy* code into fast machine code.	37
OpenSim	A physics engine aimed at musculoskeletal modelling created by Delp et al. [17], which uses *Simbody* [76] as its backend.	i, 10, 27
Pycollo	An open-source Python package for solving multiphase optimal control problems using direct orthogonal collocation developed by Brockie [10].	i, 33, 39, 40
SD/Fast	A tool that is used as a physics engine with high computational efficiency by symbolically formulating the Equations of Motion (EOMs) and converting them into optimized numeric code.	3
SciPy	An open-source Python library for mathematics, science, and engineering.	ix, 9, 35, 37
SymMePlot	An open-source Python library for plotting objects from `sympy.physics.mechanics` using *Matplotlib* developed during this thesis [79].	iii, vi, 29, 30, 35, 38, 40, 43

SymPy	An open-source Computer Algebra System (CAS) written in Python [44].	i, iii, vi, viii, 3–12, 18, 21, 23, 29–31, 33, 37, 39, 42–45, 54, 55
scikits.odes	An open-source toolkit, which provides access to additional Ordinary Differential Equation (ODE) solvers, which are not implemented in *SciPy*.	9

Glossary

algorithmic differentiation	Techniques to evaluate the partial derivative of a function in a compute program also called automatic differentiation or auto-diff.	3
code generation	Generation on numeric code from by example symbolic mathematical expressions.	8, 29, 37
differentiable physics engine	A physics engine with the addition of also providing gradient information to allow gradient-based optimization and usage in neural networks.	3
direct collocation	A class of direct methods for solving Optimal Control Problems (OCPs). Direct collocation discretizes the OCP converting it to a nonlinear program, which can be solved using a nonlinear programming solver. This is done by parametrizing both the control and the state, approximating them as polynomial splines.	33, 39
generalized coordinate	A variable used to describe the configuration of a mechanical system. Generalized coordinates are employed in minimal coordinate representations.	ix, 5, 7, 8, 30, 31, 33, 54, 61
generalized speed	A variable used to describe the motion of a mechanical system. The employment of generalized speeds is one of the unique features of Kane's method.	ix, 4, 7, 8, 30, 31, 33, 54, 61, 63
kinematic differential equation	An equation in Kane's method to relate generalized speeds to time-derivatives of the generalized coordinates.	54, 61
metaclass	A special class in Python, which defines how classes are created and behave, allowing customization of attributes, methods, and inheritance.	16
minimal coordinate representation	A coordinate representation that minimizes the number of free coordinates used to describe a system. This is done by expressing the position and orientation of the bodies using the coordinates, which describe a joint such a single angle in case of a pin joint. The utilized are coordinates have various synonyms: minimal coordinates, generalized coordinates, reduced coordinates and joint coordinates.	ix
musculoskeletal model	A simplified conceptual representation of a biomechanical system, consisting out of bodies, joints, musculotendons and constraints.	viii
musculotendon	Biomechanical actuator that is a combination of a muscle and tendon. The musculotendon is most of the time modeled as an actuator consisting of three parts (Hill-type): a contractile element, a parallel elastic element, and an elastic tendon in series.	ix, 10, 12, 17, 18, 43

physics engine	A piece of software for simulating physical systems. The core of a physics engine is a function that computes the next state based on the model, the current state and the control forces. This function generally consists out of three steps: forward dynamics, collision resolution, integration.	viii, ix, 3, 4, 10

1

Introduction

In the 200 years of the existence of bicycles, numerous researchers have developed mathematical models to investigate various aspects of bicycle dynamics and rider control [69]. These models have provided valuable insights into self-stability [41], active and passive rider control [49, 70, 75], and the identification of specific eigenmodes [74]. These insights have not only contributed to a better understanding of bicycle behavior but have also informed the development of bicycles with improved stability, handling, and comfort [57]. One of the novel examples using such a mathematical model is a balance assist system to aid riders by providing an assistive steering torque [1].

However, despite the extensive research in this field, a common challenge remains the creation of a mathematical bicycle model. As pointed out by Schwab and Meijaard [69], multiple of the historically developed models actually have had mistakes in their derivation. Nowadays, most researchers seem to use the linearized Whipple model [42] as starting point and extend it to incorporate additional features such as tyre models [36, 40, 57, 69] or rider attachments [49]. This approach is error prone [69], time consuming, hinders the development of more complex models, and reduces research dissemination and reproducibility. To bridge this gap, this project sets out the following goal:

*Develop an open-source modular and extensible framework for creating symbolic **B**icycle-**Ri**der **M**odels, called BRiM.*

In this goal, the framework is an abstract piece of software, which provides generic functionality that can be used by users. The functionality is that a user can describe a model in multiple levels of detail. A component level, where components can be chosen (modular) and where also new components can be attached (extensible). Two examples of modularity are changing the knife-edge wheel approximation of the rear wheel to a toroidal shape and changing the entire bicycle in a bicycle-rider model. An example of extensibility is that a bicycle can be modeled solely, but can also be extended with a rider model resulting in a bicycle-rider model. An intermediate level of describing a model allows the modification of a model using a joints and bodies interface. And a low level, where researchers can examine and manipulate the equations describing a model. A last property of this framework is that it will use symbolics resulting in symbolic Equations of Motion (EOMs). Symbolics refers to the use of mathematical expressions represented by symbols instead of numerical values. While the core of this framework can be applied to mechanical models in general, it will be specifically developed to create **B**icycle-**Ri**der **M**odels. Therefore, the package will also include the implementation of multiple bicycle-rider models and additional utilities, which make use of already existing scientific tools.

This final aim is achieved by means of four intermediate objectives:

- Formulate a systematic approach to describe a mechanical model.
- Design a modular and extensible framework to describe mechanical models in general.
- Apply this framework to create a library of bicycle-rider components.
- Demonstrate the usage of *BRiM*.

The objectives are sorted in chronological order. The first aim is to formulate a systematic approach, which acquires the necessary background knowledge for designing the framework. It provides an overview of how *BRiM* fits into a simulation or optimization and identifies the systematic steps of defining a model to form symbolic EOMs. This analysis and approach are required to design an algorithmic approach, which is closely related to the second objective. The second objective is obtained by first creating the requirements of this core framework. Secondly, several architectural designs are proposed, which satisfy these requirements. After analyzing the concepts on their strong and weak sides, a final design is proposed and implemented forming the core of the *BRiM*. The third goal utilizes this novel core framework to implement various bicycle-rider components resulting in bicycle-rider models. Components are interchangeable parts of the model with which a user can choose what aspects are taken into account, like the type of wheel. To further improve the bicycle-rider models a simple model is benchmarked against laborious manual implementations. Based on these results optimizations of the implemented components are done, while immediately measuring *BRiM*s performance. The fourth intermediate objective further proves the usefulness and versatility of *BRiM*. It shows some of the models that can easily be generated for simulation and optimization tasks.

The outline of this thesis follows the intermediate objectives. Chapter 2 lays the foundation by describing a systematic approach to form the EOMs symbolically using Kane's method in a simulation task. Chapter 3 builds upon this systematic approach to propose a framework to define models modularly, which is the core of *BRiM*. Next, chapter 4 utilizes the framework describing how it can be applied to create bicycle-rider models. Chapter 5 demonstrates *BRiM* by solving a trajectory tracking problem of a rolling disc following a periodic sinusoidal path and by forward simulating an advanced bicycle-rider model. Lastly, chapters 6 and 7 wrap up the thesis with future recommendations and conclusions.

2

Universal Simulation Workflow

This chapter presents a workflow to systematically formulate a model and form its EOMs in a symbolic form using *SymPy*. The primary objective is to provide a foundational understanding of the steps involved in generating the EOMs in a symbolic form. This understanding is crucial for the design of the modular modeling framework inherent to *BRiM*. By decoupling the model formulation process, *BRiM* enables modularity and extensibility, allowing individually-defined models to be combined into a cohesive bicycle-rider model.

As the entire framework uses a symbolic approach, section 2.1 starts with an explanation of symbolics. It discusses what is meant by symbolics and what are the advantages and disadvantages of forming symbolic EOMs. Next, section 2.2 identifies and describes the steps in which a model can be formulated using *SymPy*. Section 2.3 briefly continues on the derivation of the EOMs using *SymPy*'s existing version of Kane's method. Followed by section 2.4, which explains how the generated EOMs are utilized in the simulation. The final section, section 2.5, summarizes the findings and insights of this chapter.

2.1. Usage of Symbolic Equations

Symbolics refers to the use of mathematical expressions represented by symbols instead of numerical values. In the context of bicycle-rider modeling, symbolic equations allow for exact mathematical representations of the system dynamics. To work with symbolic equations, researchers utilize Computer Algebra System (CAS), which is software designed for algebraic manipulation, simplification, and evaluation of expressions. Popular CASs include *Maple* [39], *Mathematica* [24], *Symbolics.jl* [21] and *SymPy* [44]. In the approaches of simulating multibody systems, symbolics contrasts numerics. While most bicycle researchers rely on handcrafted EOMs [69], most physics engines actually use fast numeric algorithms [19], which compute the EOMs every timestep [82].

The distinction between symbolics and numerics lies in the nature of the final results obtained before evaluation. Symbolic computations yield sets of analytically exact expressions, while numerics yield numerical results. Symbolics offer several advantages. Firstly, symbolic expressions enable further expression manipulation, value substitution, expression reuse, and differentiation. Differentiation is especially useful for optimization tasks, as the analytic computation of the gradient allows for higher-order optimization methods. Secondly, Rosenthal and Sherman [64] also argue that handcrafted symbolic EOMs have the highest reachable computational performance.

Recent advancements however are blurring the boundaries between symbolics and numerics. With algorithmic differentiation one can compute the partial derivatives of a piece of code [20]. This is, among others, used in the creation of differentiable physics engines, which also offer gradients of the EOMs allowing for faster optimization and usage within neural networks [23, 89]. The accuracy of these gradients, particularly in scenarios involving contact dynamics, continues to be an area of improvement [95]. An additional disadvantage seen in algorithmic differentiation is the added computation and memory costs when running the simulation [45]. Similarly, symbolics has additional computation costs when formulating the model. As for the difference in computation speed, Todorov, Erez, and Tassa [82] argue in their comparison between *MuJoCo* and *SD/Fast* that modern-day compilers mostly seem to bridge

this gap. Despite the closing gap in performance and features, the choice to use symbolics is motivated by a few additional factors. First, symbolics offers transparency, allowing for a clear understanding of the underlying equations and enabling greater flexibility in manipulating and interpreting the resulting expressions. The last of which leads to the second advantage, namely that there is a scope for the symbolic approach to result in more performant code [21].

Out of the several CASs mentioned before, *SymPy* [44] has been chosen as backend for *BRiM*, because it has several advantages. Firstly, it is open-source, which aligns with the open-source values of *BRiM*. Secondly, it provides a large feature set, including a dedicated module for defining mechanical problems. Thirdly, it is implemented in Python, which is both easy to learn and widely used. Additionally, its compatibility with various scientific libraries facilitates seamless integration with existing tools. This overall empowers researchers to publicly develop and share models to be combined with numerical solvers, visualization tools, optimization libraries, and other bicycle or biomechanical research tools.

2.2. Model Formulation Workflow

In order to write a modular framework to formulate a model using *SymPy*, it is necessary to explore a systematic approach to define a model. This involves breaking down the process into smaller decoupled steps. This division makes an implementation of a model easier to understand, develop and maintain. Additionally, it aids brainstorming on how *BRiM* can allow more decoupling, control, and customization of models.

Literature showcases several systematic approaches for automatic model formulation and derivation of the EOMs. Rosenthal and Sherman [64] employ Kane's method to form the EOMs symbolically. Featherstone [19] discusses various algorithms to form the EOMs, both implicitly and explicitly, as is used by several numeric physics engines [60, 82]. Casius, Bobbert, and Van Soest [12] use Newton-Euler to form the EOMs. They focus more closely on the actual formulation of the model itself in 2D using a human sprint cycling model as an example. However, for biomechanical simulations in 3D they do advise using Kane's method, which is also backed up by Yamaguchi [93]. The rest of this section explains a systematic approach to formulate a model with *SymPy*, such that in section 2.3 the EOMs can be formed using Kane's method [26].

To explain each of the steps in formulating the model, a rolling disc will be used as an example, shown in figure 2.1. The rolling disc is defined as an infinitesimally thin disc, rolling on the ground without lateral and longitudinal slip. The disc itself is defined with respect to the ground with a subsequent yaw-roll-pitch body-fixed rotation. Its contact point is defined to be in the ground plane at $q_1\hat{n}_x + q_2\hat{n}_y$ from the origin. As for the generalized speeds used in Kane's method, those are defined as $u_i = \dot{q}_i$ for $i = 1, \ldots, 5$. While the ground is treated as the Newtonian body, the disc has a specified mass, inertia and radius. To control the disc three time-varying torques are used: T_{drive} acts about the rotation axis of the disc, T_{steer} acts about the axis going through the contact point and the center of the disc, and T_{roll} acts about the axis perpendicular to both the normal of the ground and the rotation axis.

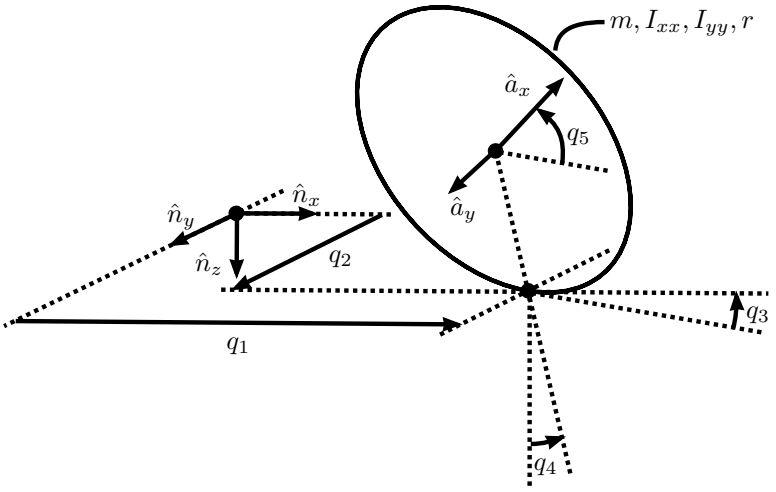


Figure 2.1: Free body diagram of a rolling disc.

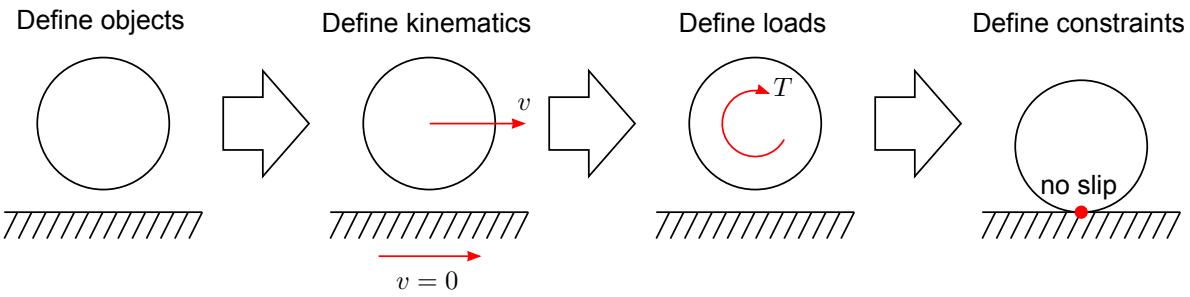


Figure 2.2: Illustration of the identified steps in formulating a model in *SymPy*. The image shows a simplified example of a rolling disc. The first step defines the ground and the disc. The second creates the kinematic relationships by defining the velocities. The third defines a torque acting upon the disc. And the fourth defines a no-slip constraint between the ground and the disc.

The steps that were identified in formulating a model are illustrated in figure 2.2. Each of those steps are further elaborated on in the subsections.

2.2.1. Define Objects

The first step is to define all objects. A first set of objects are symbols to represent various quantities. Within the mechanics module of *SymPy*, `sympy.physics.mechanics`, there are two kinds of symbols: symbols and dynamic symbols. Symbols denote constants, like mass, dynamic symbols denote time-varying functions, like generalized coordinates. In the code snippet below each of these quantities is created for the rolling disc.

```
# Import the sympy and its mechanics module under the aliases sm and me
import sympy as sm
import sympy.physics.mechanics as me

g, m, r, Ixx, Iyy = sm.symbols("g m r Ixx Iyy")
q1, q2, q3, q4, q5, u1, u2, u3, u4, u5 = me.dynamicsymbols("q1:6 u1:6")
T_drive, T_steer, T_roll = me.dynamicsymbols("T_drive T_steer T_roll")
```

Listing 2.1: Code defining the symbols used in the rolling disc model.

Other objects to be created are the reference frames, points and bodies. The code snippet below constructs the reference frames and points manually. This clearly shows that a rigid-body is mainly a data object keeping references to its intertial properties, frame, and center of mass. They can also be instantiated automatically when creating a body.

```
N = me.ReferenceFrame("ground_frame")
A = me.ReferenceFrame("disc_frame")
O = me.Point("origin")
P = me.Point("disc_center")
CP = me.Point("contact_point")
disc = me.RigidBody(
    name="disc",
    masscenter=P,
    frame=A,
    mass=m,
    inertia=me.Inertia.from_inertia_scalars(P, A, Ixx, Iyy, Ixx)
)
ground = me.RigidBody("ground", O, N)
```

Listing 2.2: Code defining the reference frames, points, and bodies of the rolling disc model.

These objects together form the building blocks of the mechanical system. Therefore, all other stages depend on the define objects stage. As *SymPy* uses graph representations for its symbolic

expressions[1] as well as for the relationships between reference frames and points, this step can be seen as creating the nodes of all the graphs[2]. The edges ought to be defined in the next stages.

2.2.2. Define Kinematics

Where the define objects step only creates the two base components, namely reference frames and points, the define kinematics stage defines the edges in the graphs they are used in. To store the kinematic relationships, the `sympy.physics.mechanics` module makes use of six graphs in total: three for describing the orientation, angular velocity, and angular acceleration of the reference frames, and three for describing the position, velocity, and acceleration of the points. An edge in a graph describes the relationship between two nodes. In the case of the points position graph, an edge describes the position of one point with respect to another.

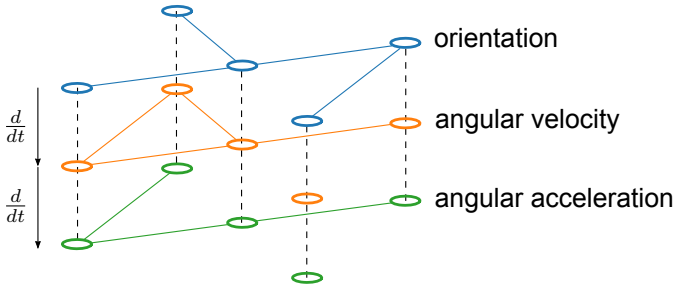


Figure 2.3: Visualization of the kinematic relations structure used in `sympy.physics.mechanics` for the reference frames. The image shows three graphs, which are related by the time derivative. The circles represent the nodes, i.e. reference frames and the edges represent the existing relationships. An edge in the top graph describes the orientation between two frames using a Direct Cosine Matrix (DCM). An edge in the middle graph represents the angular velocity of one frame with respect to another. The bottom graph shows the defined angular accelerations between frames.

These graphs are also interrelated, as illustrated in figure 2.3 for the reference frames. The reference frames use three bidirectional graphs. Notable in this case, is the fact that only the graph describing the orientation between reference frames ought to be connected and acyclic. Connectivity is a required property, as the orientation from each frame with respect to the Newtonian frame ought to be defined. The graph must be acyclic to ensure its consistency. On the other hand, the angular velocity and angular acceleration graph can both be disconnected and cyclic. The reason is that both can be derived from another graph: the angular velocity from the orientation graph and the angular acceleration from the angular velocity by taking the time derivative. Points use a similar graph structure, where a bidirectional connected acyclic graph is used for the positions between points. The velocity of points is stored with respect to reference frames. Consequently, a point's velocity in a frame can only be computed if and only if there is a connected point, whose velocity is already known in the particular frame. The acceleration again equals the time derivative of the velocity.

From these graph descriptions, several important conclusions can be drawn. First and foremost that it does not matter in what order relationships are defined within a graph, as long as the final result of the graphs is as follows:

- The orientation graph of the reference frames must be connected and acyclic.
- The position graph of the points must be connected and acyclic.
- The velocity of a least one point must be defined in the Newtonian reference frame.

This implies that one can work on different parts of the same graph separately in random order without requiring any knowledge about the other parts, which is generally advantageous for decoupling. However, there is an order-dependency between graphs. The position graph of the points is dependent on the orientation graph of the reference frames, because it defines its relations by means of the reference frames' unit vectors. The orientation graph of the reference frames on the other hand is entirely independent of any other graph[3]. Another important conclusion with respect to biomechanical modeling

[1] Checkout the *SymPy* documentation on *Advanced Expression Manipulation* for more details.

[2] The implications of these insights will be discussed in section 2.2.2.

[3] The future recommendations, chapter 6, propose to split up the "define kinematics" step into a "define orientations" and "define positions" step because of this dependency structure.

is that the actual formulation of the kinematics matches closely to the mobilizer formulation proposed by Seth et al. [72], which is used in *Simbody* [76]. This can be seen in the usage of DCMs for the orientation between frames and position vectors between points[4].

The code snippet below establishes the minimal required definition of the kinematics of the rolling disc.

```
21 A.orient_body_fixed(N, (q3, q4, q5), "ZXY")
22 CP.set_pos(O, q1 * N.x + q2 * N.y)
23 P.set_pos(CP, r * me.cross(A.y, me.cross(A.y, N.z)).normalize())
24 O.set_vel(N, 0)
```

Listing 2.3: Code establishing the kinematic relationships between the rolling disc and the ground.

2.2.3. Define Loads

With the kinematics defined, the next step is to define the loads. The term loads is used as a general term for both forces and torques. A load within *SymPy* is defined as a vector, which is associated with a location. For a force this location must be a point[5] and for a torque the location must be a frame, because points are used to describe changes in position and reference frames to describe rotations.

The reason for loads to be after the "define kinematics" step, is that loads may make use of relationships between points and frames and not the other way around. An example of a load depending on a kinematic relationship is a spring between two moving points. However, if one were to be using non-contributing loads, like a normal force in the model, then it is required in Kane's method to also introduce an auxiliary speed in the direction the load is acting. In this case, it is best to introduce this speed in the define kinematics step, as it should propagate through the kinematic relationships.

The process of defining loads is illustrated in the code below. Here the total applied torque and the force due to gravity acting upon the disc are computed.

```
25 torque = me.Torque(
26     frame=A,
27     torque=(T_drive * A.y
28             + T_steer * P.pos_from(CP).normalize()
29             + T_roll * me.cross(A.y, N.z).normalize())
30 )
31 force = me.Force(P, m * g * N.z)
```

Listing 2.4: Code computing the force and torque acting upon the rolling disc.

2.2.4. Define Constraints

The last step in the model formulation is the definition of the constraints. Within the field of dynamics, one generally distinguishes two types of constraints: holonomic and nonholonomic constraints. Holonomic constraints are defined as equations that are functions of the generalized coordinates and possibly time. Nonholonomic constraints are non-integrable equations that are linear functions in both the independent and dependent generalized speeds [50]. Generally, constraints can be seen as equations, which would create some sort of a cycle in some of the graphs used by reference frames and points. A clear example is the closing joint of a four-bar linkage. In this example, the constraints define "an extra positional relation" between two points, which are already connected via the open-loop chain.

Listing 2.5 shows that the nonholonomic constraints are computed by subtracting two alternative computations of the velocity of the wheel's center. However, it is important to note that this computation should not use the points velocity graph[6], because the velocity of some points can be computed based on different theoretical constructs. An example of such a point is the contact point. In pure-rolling it can

[4] An initial step towards a mobilizer implementation in the joints framework in `sympy.physics.mechanics` can be found in #23920

[5] *SymPy* uses a point, while it is in fact a line to which the vector is bounded.

[6] The assumption is here that the previously made definitions are not known.

be defined as having a velocity in the ground's frame[7], but also as having zero velocity[8].

```
N_v_P1 = P.pos_from(O).dt(N)
N_v_P2 = me.cross(A.ang_vel_in(N), P.pos_from(CP))
nonholonomic_constraints = [
    (N_v_P1 - N_v_P2).dot(N.x),
    (N_v_P1 - N_v_P2).dot(N.y),
]
```

Listing 2.5: Code computing the nonholonomic constraints ensuring pure-rolling of the rolling disc.

As the constraint definition step depends the strongest on the previous steps, it has been chosen as the last step. However one should note that the "define loads" and "define constraints" steps are not dependent on each other, so they could be swapped.

2.3. Kane's Method

The mechanics module in *SymPy* offers multiple automated methods, which can form the EOMs of a system. One of which is Kane's method. There are two major advantages of Kane's method over classical methods such as Newton-Euler, Lagrange and Hamilton. Firstly, it is designed to be systematic, while also resulting in simpler EOMs. Secondly, it can directly deal with nonholonomic systems without having to use Lagrange multipliers [26].

Due to the systematic implementation and integration of Kane's method in *SymPy*, the user only requires to specify its inputs. For the rolling disc example from the previous section these are shown in listing 2.6. An important remark is that Kane's method not only uses generalized coordinates but also generalized speeds, which are defined as linear functions of the derivatives of the generalized coordinates. Generalized speeds are used to specify the velocity of the system, like generalized coordinates are used to specify its configuration. For more information on Kane's method and its implementation in *SymPy* see appendix A.

```
t = me.dynamicsymbols._t  # Symbol for time utilized by dynamicsymbols
kane = me.KanesMethod(
    frame=N,  # Inertial frame
    q_ind=[q1, q2, q3, q4, q5],  # Independent generalized coordinates
    u_ind=[u3, u4, u5],  # Independent generalized speeds
    kd_eqs=[u1 - q1.diff(t), u2 - q2.diff(t), u3 - q3.diff(t),
            u4 - q4.diff(t), u5 - q5.diff(t)],  # Kinematic differential equations
    u_dependent=[u1, u2],  # Dependent generalized speeds
    velocity_constraints=nonholonomic_constraints,  # Velocity constraints
    bodies=[disc, ground],  # Bodies within the system
    forcelist=[force, torque],  # Loads acting upon the system
)
kane.kanes_equations()  # Forms the EOMs
```

Listing 2.6: Code utilizing *SymPy*'s Kane's method to compute the EOMs.

2.4. Code Generation and Integration

The result of Kane's method are the analytically exact EOMs. For simulation it is necessary to efficiently evaluate these to obtain the acceleration. Therefore, it is required to generate compilable and executable code from the analytical EOMs of the system. *SymPy* contains several functions for code generation to different languages, such as C, Fortran, Octave and Python[9]. An important optimization within code generation is Common Subexpression Elimination (CSE). CSE determines what subexpressions are used in multiple locations and replaces them by a temporary variable, which only has

[7] The contact point's location changes with respect to time, which can be computed by differentiating the position vector w.r.t. to the origin in the ground's frame.

[8] The contact point is also the instantaneous center of rotation and has therefore a zero velocity in both the ground's as well as the disc's frame.

[9] For more information refer to the *Code Generation page* in the online *SymPy* documentation.

to be computed once. Thereby the number of operations, e.g. summations and multiplications, in the EOMs can be significantly reduced.

After code generating the EOMs, the function can be passed to an integrator with the initial conditions and inputs to simulate the system. There are many integrators available in Python, two popular libraries are *SciPy* and *scikits.odes*. Integrators from both of these libraries are utilized in this project.

2.5. Summary

This chapter offers several insights into developing a modular framework by identifying a systematic approach to define a model. Based on which the EOMs can be computed using an algorithmic implementation of Kane's method in *SymPy*. The systematic approach consists of four steps:

1. "Define objects" (section 2.2.1). Creates the objects, such as symbols and reference frames, without defining any relationships between them.
2. "Define kinematics" (section 2.2.2). Establishes relationships between the objects' orientations/-positions, velocities, and accelerations.
3. "Define loads" (section 2.2.3). Specifies the forces and torques acting upon the system.
4. "Define constraints" (section 2.2.4). Computes the holonomic and nonholonomic constraints to which the system is subject.

From the perspective of the graphs, it can also be seen that these steps are decoupled and internally mostly order-independent[10]. "Define objects" corresponds to the creation of the nodes. "Define kinematics" corresponds to the creation of the edges. And both "define constraints" and "define loads" only make use of the graphs to define additional properties of the system.

A last consequence of the graph perspective is the identification of three properties that a valid system must satisfy:

- The orientation graph of the reference frames must be connected and acyclic.
- The position graph of the points must be connected and acyclic.
- The velocity of a least one point must be defined in the Newtonian reference frame.

[10] The reason why it is not entirely order-independent is further clarified in chapter 6.

3

Modular Mechanics Modeling

BRiM Core

To aid the development of a modular framework to create bicycle-rider models, a universally applicable framework is proposed forming the core of *BRiM*. This chapter focuses on this framework which allows for multiple levels of describing a model. At a high level, models can be specified using components, providing the modularity to select a specific wheel or tyre model in a single statement. For features not supported at the component level, an intermediate level allows the use of joints and bodies to represent the model. The framework also allows for a low-level description, enabling the manipulation of expressions and the use of custom DCMs between frames. The combination of these three is possible, as they all form symbolic building blocks defined using *SymPy*, while building upon the model formulation workflow, as discussed in the previous chapter. This chapter focuses on the explanation of the high level, while building upon and supporting the other two levels. Though this project did also involve redesigning the bodies and joints interface and introducing a special `System` object in *SymPy*'s mechanics module, it has been excluded from this report. An explanation of the joints framework is available in *SymPy*'s online documentation[1]. An explanation of the `System` object has not been made public yet[2].

Different physics engines use different formats to describe models. A popular format to describe robots is the Unified Robot Description Format (URDF) [83], but it suffers from several limitations such as the lack of support for musculotendons. Alternative file formats like the XML-based *osim* format in *OpenSim* [17] and *MJCF* in *MuJoCo* [82] have been introduced to address some of these limitations and optimize for their respective physics engines. However, creating model files in either of these formats is challenging [83]. All of these file formats primarily focus on the intermediate-level description using joints, bodies, and actuators. Although *MJCF* can be considered a hybrid between a modeling format and a programming language, it lacks the full power and versatility of a general-purpose programming language [16].

This chapter is structured as follows. Section 3.1 explains the requirements of a modular framework. After which section 3.2 explains how modularity, in general, can be viewed from a physics perspective. Building upon this understanding, section 3.3 proposes two architectures with a discussion of their advantages and disadvantages. Based on these proposals a new design is derived consisting of three fundamental components, further explained in section 3.4. Finally, the application of the proposed design is exemplified in section 3.5, where the framework is utilized to model the dynamics of a rolling disc.

[1] The joints framework explanation page can be found at: `https://docs.sympy.org/dev/modules/physics/mechanics/joints.html`. The API reference is available at: `https://docs.sympy.org/dev/modules/physics/mechanics/api/joint.html`.

[2] I have written a full description in its docstring. However, the object is not publicly available at the time of writing this thesis, due to it still being experimental.

3.1. Requirements

To investigate the solution space for the required modular framework, it is necessary to define a set of requirements. These requirements were identified by informally interviewing researchers and users of *SymPy*. The requirements, also called features, have been classified into three categories:

- Functional requirements (section 3.1.1): These describe desired functionalities of the framework.
- Performance requirements (section 3.1.2): These specify aspects that the framework should be optimized.
- Constraints (section 3.1.3): These identify limitations and restrictions to which the framework should be subjected.

The prioritization of the requirements is further classified using the agile method, called *MoSCoW* [14], where each of the capital letters stands for a different category:

- M (Must): Requirements that are deemed essential and must be satisfied for the framework's success.
- S (Should): Requirements that are important and should be satisfied to meet the framework's objectives.
- C (Could): Requirements that are desirable and could be satisfied if resources permit.
- W (Won't): Requirements that will not be satisfied in the current scope of the framework.

To enhance clarity and understanding of the requirements, user stories are utilized. User stories provide concise explanations of the wishes and needs expressed by different stakeholders. The identified stakeholders for the framework are:

- Developers: Individuals who work at the core of *BRiM*, responsible for maintaining and developing the framework.
- Modelers: Users who utilize the core of *BRiM* to create new models, such as a new bicycle-rider model. They possibly contribute to the other modules in *BRiM*.
- End-users: Users who use and combine the models created within *BRiM* for their specific purposes.

3.1.1. Functional Requirements

The following functional requirements for the core have been identified:

- It must be possible to create a **library of components**.

  As a modeler I want to be able to create a set of components like a knife edge wheel and toroidal-shaped wheel, which can be easily reused in different systems.

- It must be **modular**.

  As an end-user I want to be able to relatively simply change out the model of a wheel from a knife edge wheel to a toroidal-shaped wheel.

- It must be **extensible**.

  As an end-user I want to add the model of a rider on top of a bicycle, after modeling a bicycle and rider separately.

- It must be possible to create a new component of a type, which is automatically **compatible** with other **models**.

  As a modeler I want my newly created flexible rear frame to automatically be compatible with the bicycle models which the rigid rear frame is also compatible with.

- It must be **compatible** with the existing `sympy.physics.mechanics` objects.

  As an end-user I want the option to add other loads like gravity to the model using the `sympy.physics.mechanics` *Application Programming Interface (API).*

- It must be possible to choose what set of **prespecified loads** should be applied.

  As an end-user I want the option to specify in a single statement set of musculotendons should be applied to the model.

- It should support choosing a different **convention** for the same model

  As a modeler I want to be able to implement a Whipple bicycle model according to Moore's convention and to Meijaard's convention, which can be instantiated from the Whipple bicycle class.

- It should have a method to get a **description of symbols**.

  As an end-user I want to easily be able to figure out a description of a symbol used in the EOMs.

- It could have the possibility to **customize symbols**.

  As an end-user I want the ability to use the same symbol for the radius for the front and the rear wheel.

- It could have smart methods to **find suitable components**.

  As an end-user I want to be able to easily find what wheel models I can use in a certain bicycle model as the rear wheel.

3.1.2. Performance Requirements

The following performance requirements for the core have been identified:

- It must strive for **decoupling of models**.

  As a developer I aim for decoupling, because it solves problems where objects have to be made compatible.

- It should be as **intuitively** as possible to follow the general principles and thought processes used in defining a unique model and forming the EOMs.

  Everyone wants the framework to be as intuitive as possible.

- It should minimize the amount of **boilerplate** code when defining a component.

  As a modeler I want to write as less code as possible and only write those things which actually characterize my model.

- It should minimize the **number of operations** in the EOMs without sacrificing computation speed[3].

  As an end-user I want my EOMs to be as efficient as possible.

- It could minimize the **computation time** for defining the model and forming the EOMs.

  As an end-user I want my equations to be formed as fast as possible.

3.1.3. Constraints

- It must use ***SymPy*** and specifically build upon its module `sympy.physics.mechanics`.

  As a developer I have chosen for SymPy and want full compatibility with its mechanics module, because everyone wants to utilize its functionality.

- **Symbols** used by different components must be **unique**.

  As an end-user I want the rear wheel and front wheel radius to have a unique symbol by default, even though they may be instances of the same model.
  As a modeler I want an easy method to make sure the symbols between different models are different, while still using sensible names for each symbol.

[3] The reason to constrain the number of operations with the computation speed is to avoid the use of time-consuming simplification routines.

3.2. Moving System Boundaries

An important concept in each design of the framework as well as in modeling systems in general is the definition of a system boundary. A system boundary represents a demarcation line that separates the system under consideration from the world. It defines the scope and extent of the system, encompassing all the components and interactions that are relevant to the specific analysis or modeling task. Choosing a good system boundary is seen as a crucial step, as it can vastly simplify the modeling process by abstracting away unnecessary details.

Furthermore, a system boundary enables modular modeling by facilitating the decomposition of complex systems into smaller, more manageable subsystems. Each subsystem can be treated independently within its respective system boundary, simplifying the modeling of individual components. Once the subsystems have been defined separately, they can be merged to form the complete system. This modular approach not only enhances the understanding and organization of the model but also promotes reusability and flexibility in system design. A visual representation of system boundaries applied to a bipedal robot model is depicted in figure 3.1.

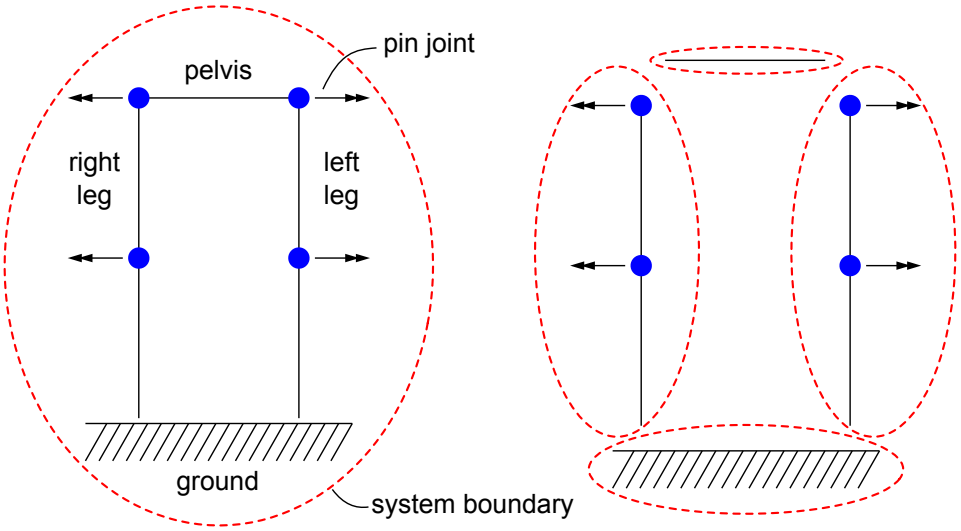


Figure 3.1: Illustration of how system boundaries can be drawn differently to split up a model into different parts. The bipedal robot is split up into a separate ground, pelvis, left leg, and right leg system. Thereby each smaller system can be modeled separately before combining them later on when drawing a system boundary, which unites all of the smaller systems.

In practical applications, it is important to consider interactions that occur at the system boundaries. For instance, if a system boundary represents a single link of a pendulum, it becomes necessary to specify the loads applied by the pin joints on the link. These loads can be treated as unknowns within the subsystem. However, another subsystem will be having the same unknown loads in the opposite direction. Therefore, both subsystems do not need to model these interactions. The system which unites the subsystems will have to specify the exact interaction.

3.3. Graph Structure

From the concept of defining separate subsystems using system boundaries, which can be united, follows a graph structure. Graphs come in different types with varying properties. This section proposes two new designs: a tree graph (section 3.3.1) and a bi-directional cyclic graph, called a flat graph for simplicity (section 3.3.2). Both of these graphs have several advantages and disadvantages. Section 3.3.3 concludes with a discussion to make a decision on what parts of the designs to proceed. To aid in the description of both designs, the bipedal robot presented in figure 3.1 will be used as an example.

3.3.1. Tree Structure

A tree structure is an acyclic directional graph with a hierarchical organization. In this design, a model (parent) is composed out of smaller submodels (children). So each submodel will define its system

within its own system boundary and the overarching parent model will define their interactions. An illustration of the tree structure applied on the bipedal robot is depiced in figure 3.2.

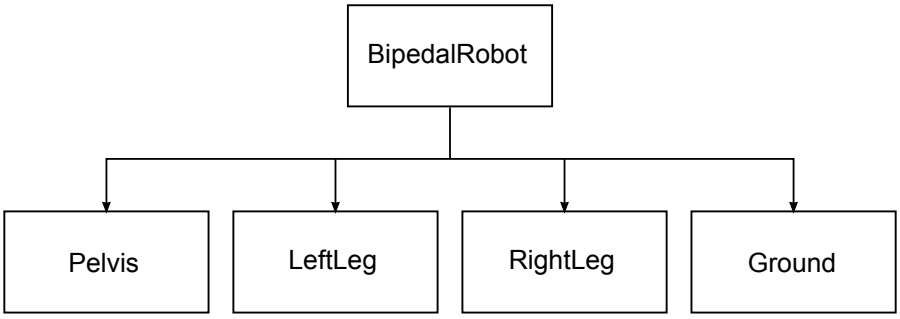


Figure 3.2: Visualization of the tree graph design for the bipedal robot shown in figure 3.1. The *BipedalRobot* is the parent model composed of four submodels.

3.3.2. Flat Graph Structure

In contrast, a flat graph structure represents a bi-directional cyclic graph without a direct hierarchical relationship between the systems. This design choice aligns with the notion that connections between systems in the physical world often describe interactions rather than strict hierarchies. To achieve a flat graph structure, a registry can be employed to automatically establish the connections between components. If one would specify that there should be a connection between the ground and the left leg, then the registry finds a connection that is able to do so. However, while this design does remove the explicit hierarchical relationship, it still necessitates some kind of grouping utility to automatically specify which components should be connected with each other. Figure 3.3 provides a visualization of the flat graph design for the bipedal robot.

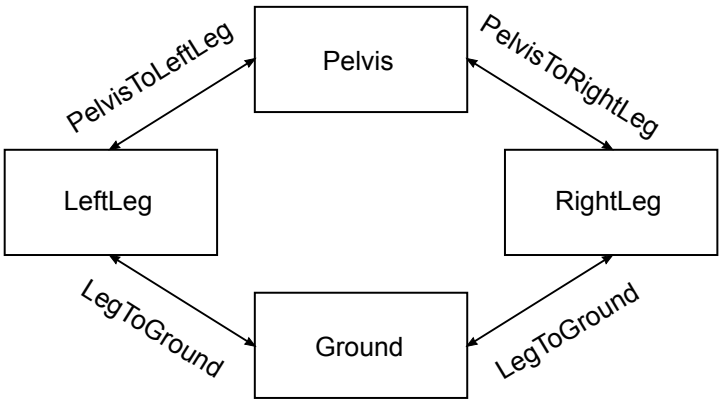


Figure 3.3: Visualization of the flat graph design for the bipedal robot shown in figure 3.1. Each of the models is connected to each other bi-directionally using a connection as an edge.

3.3.3. Concept Evaluation

The decision between a tree structure and a flat graph structure depends on various factors. Both designs offer a method to decouple the different submodels. They also use graph structures for which a wide variety of well-documented algorithms are available including methods to visualize them. Though it must be mentioned that the visualization of trees is more readable and easier to implement. A disadvantageous property shared by both designs is that the choice of the system boundaries is partially subjective. However, the flat graph is more forgiving here, because it can more easily work around the problem by introducing new connections.

The tree graph offers several additional advantages. It aligns with a natural conceptualization by modelers through an intuitive compositional representation. Through its hierarchical structure it also provides a lot of clarity and structure. What on its turn results in improved ease in debugging and maintaining. Overall, the tree structure offers a very user-friendly design.

There are however also several disadvantages. Most of these are found when pairing two submodels with a complex interaction, which relies on several model-unique properties. An example is the interaction between the wheel and the ground. Firstly, it relies on unique properties of how the shape of each is described in order to compute the contact point. If it is a knife-edge wheel, only the radius is utilized. However, if it is a toroidal-shaped wheel, also the transverse radius must be taken into account with a slightly different computation. These are only two examples for the wheel, but the ground may also not be flat giving even more combinations. The second problem already notable in this example is that the computation is complex and should therefore be reusable among different parent models. Thirdly, the exact interaction between the two submodels may in itself also be modular. In the case of the interaction between the wheel and the ground, there is also the definition of the tyre model, which must be modular. A last disadvantage is that some models may require the creation of new properties for compatibility with other models. An example is the addition of a rider lean axis, when adding a rider on top of a rear frame.

The flat graph structure offers other advantages. Contrary to the tree structure, it aligns with the physics perspective, because interactions between systems are similarly defined to interactions within systems. Additionally, it provides greater flexibility, as every model can be connected to another model by introducing a new connection. A huge advantage, which actually solves most of the disadvantages of the tree structure. Lastly, the registry introduced by the flat graph also greatly improves the users' workflow in aiding the search for suitable connections.

However, all of these advantages come at a cost. Because connections are utilized for every interaction, the code base grows exponentially. Also, the increased flexibility is sensitive to cluttering of the code base, since a new connection can be introduced to solve every problem, though this can be reduced by introducing a strict set of guidelines[4]. The last problem is that connections, i.e. edges, are designed to model the interaction only between two models, while one may prefer to describe an interaction between multiple models or between two groups of models and connections.

All in all, it can be seen that there are many advantages on either side. While the tree structure is more user-friendly and intuitive, the flat graph manages to solve several of the issues of the tree structure. However, this comes at the cost of various disadvantages mainly revolving around the problem that it needs a lot of different connections. An optimal solution is created by incorporating strong aspects of both to provide a comprehensive modeling approach. The new design utilizes the tree structure as basis, but reworks the connectors from the flat graph into, so-called connections (section 3.4.2). Additionally, the concept of using a registry to assist users in finding suitable models and connections can also be adopted from the flat graph design.

3.4. Core Components

The core design principles discussed in section 3.3 and the identified requirements outlined in section 3.1 are implemented in *BRiM* through three fundamental components: models (section 3.4.1), connections (section 3.4.2), and load groups (section 3.4.3). The models are the main components, where each model describes a system or subsystem following the tree structure explained in section 3.3.1. The connections can be seen as an utility in of parent models to describe a modular interaction between submodels resembling many characteristics with the connectors from the flat graph structure. The load groups are predefined sets of actuators and loads, which are commonly associated with a specific model or connection.

[4] A discussion of a version of the flat graph structure involving various of these strict guidelines can be found in #16 of *BRiM*

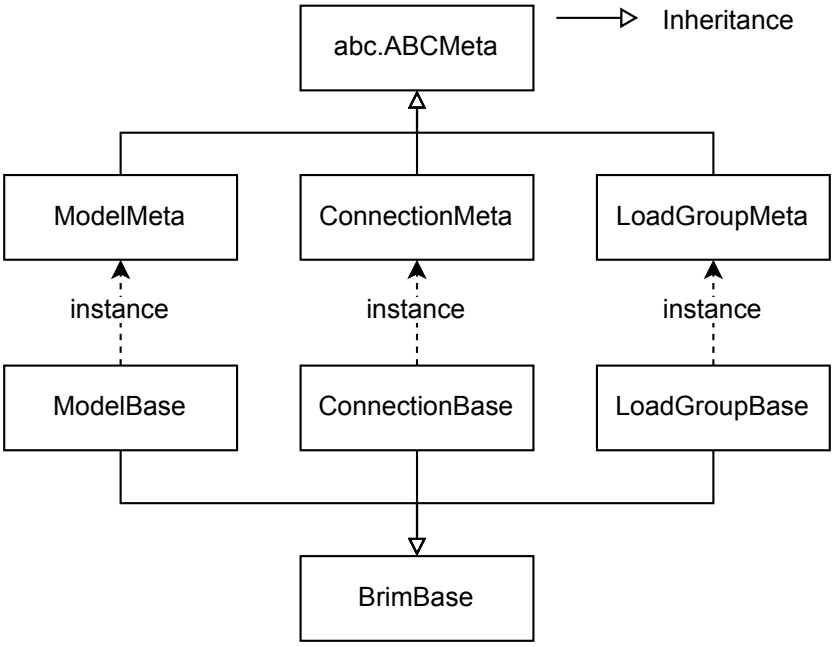


Figure 3.4: UML diagram of the main core components of *BRiM*. The UML diagram shows the three abstract classes[5]: `ModelBase`, `ConnectionBase` and `LoadGroupBase`. Each of these classes has a separate metaclass, which processes the class definition and registers the class in the registry. To increase the similarity between each and remove code duplication, each of the classes also inherits from `BrimBase`.

A Unified Modeling Language (UML) diagram of the core components is shown in figure 3.4. In line with the principles of object-oriented programming *BRiM* uses classes to create models, connections, and load groups. Such that one can instantiate `KnifeEdgeWheel` to create a model of a wheel. These three base classes inherit from `BrimBase`, because they share the four define steps, discussed in section 2.2. Metaclasses[6] are used to improve the modeler's interface by reducing boilerplate and automatically adding the defined classes to the registry. Another option to preprocess a class definition would have been the usage of decorators, similar as in the *dataclasses* library. However, utilizing a metaclass offers two main advantages over using a decorator. Firstly, metaclasses align more closely with the principles of object-oriented programming. Secondly, the application of a metaclass automatically occurs through inheritance, whereas a decorator needs to be manually applied to each class.

3.4.1. Models

A model in *BRiM* is an object that represents a specific system within defined system boundaries. A model can be composed out of multiple submodels, in this case the composed model is called a parent model with respect to the submodels. Each model encapsulates the relations and behaviors of the system, allowing for a modular and hierarchical representation where the parents do not know the details of their submodels. An example of encapsulation is the usage of a rotation axis property in the case of a wheel, as it hides information on how and why it chooses a certain vector.

While the usage of an abstract class for a component ensures a default interface, it is common for bicycle models to have multiple conventions. To accommodate this variability, an addition is made to the core design of *BRiM*. A class method called `from_convention` is introduced, utilizing the registry (see section 3.3.2) to locate the appropriate model based on the class property `convention`.

3.4.2. Connections

A connection object in *BRiM* serves as a utility used by a parent model to establish relationships between two or more submodels. It offers three key features that enhance the modeling framework. Firstly, a connection enables a parent model to define interactions between multiple submodels in a modular manner. For instance, consider a robot arm connected to its base. The type of connection

[5] `abc.ABCMeta` is a metaclass to create abstract base classes.

[6] A metaclass is a special class in Python, which defines how classes are created and behave, allowing customization of attributes, methods, and inheritance.

between the arm and the base can be modular, allowing the parent to choose between fixation, a pin joint, or other options. Secondly, connections enable the modular reusability of complex interactions. An example is the complex interaction between a wheel and the ground. This interaction should not be described separately in each model containing a wheel interacting with the ground. Also creating a separate submodel is not possible, as it would break the tree structure by requiring the same submodel to be used by different parent models. A visualization of this problem is shown in figure 3.5.

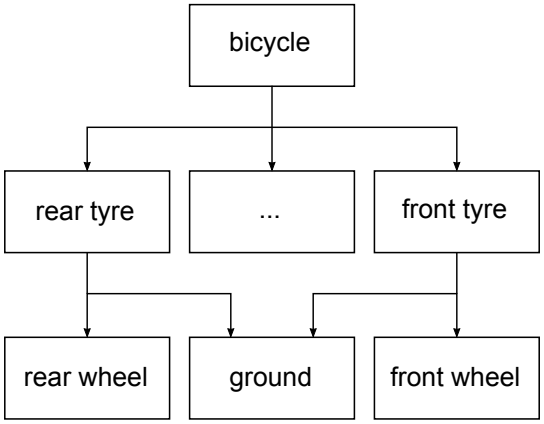


Figure 3.5: Visualization of a broken tree structure caused by the usage of a tyre model object to describe the interaction between a wheel and the ground.

Thirdly, connections provide the capability to incorporate new properties into a model. While the rider's lean axis property may not be necessary when solely modeling a bicycle's rear frame, it becomes a valuable addition when considering the interaction between the bicycle and a rider. By utilizing a connection this property can be easily included in both the rear frame and the rider[7]. This flexibility in adding additional properties to a model contributes to the versatility of the modeling framework. As can be noticed from these features. The problems found in the tree structure, in section 3.3.3, are solved by the connection object.

3.4.3. Load Groups

Most models are by themselves not by definition associated with certain load, but there are set of loads which are commonly applied to certain models. Therefore, it is advantages to introduce a separate object called load groups. A load group allows adding a set of actuators and loads to a model or connection. An example where it is advantageous to use load groups is for a leg, see illustration in figure 3.6. Consider a leg consisting of three bodies: a thigh, shank, and foot. Each of these bodies is connected using a pin joint. The simplest load group would be to use a torque actuator applying a time-dependent torque $T_i(t)$ to each of the pin joints about the rotation axis. A bit more advanced option is to replace this scalar with a linear torsional spring-damper. A further step would be to model musculotendons. Each of these three options can be defined using a load group, which can be applied to the model of the leg. Note that it is also possible to add multiple load groups in parallel, e.g. the time-dependent torque and the linear torsional spring-damper could be applied simultaneously.

[7] In the current implementation, these properties are added to connection objects rather than the model objects themselves.

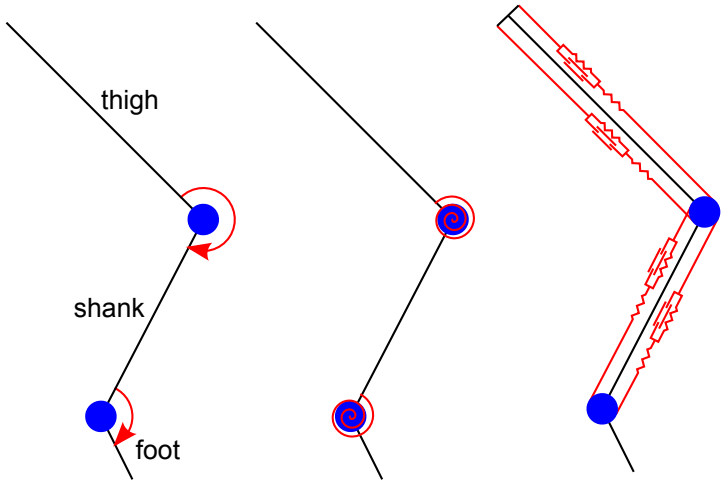


Figure 3.6: Illustration of different load groups one can use for the leg. The left uses time-dependent torques, the middle linear torsional spring-dampers, and the right musculotendons.

3.4.4. Model Definition Workflow

The definition of a model starts with the end-user defining the model's structure. The user specifies the aggregation of each model, i.e. out of what submodels, connections, and load groups each model exists. Having this structure defined the user starts the model definition process:

1. Associate the submodels with the connections. The end-user specified the submodels with respect to the parent models. In this phase the parent models associate the submodels with the connections, to ensure that the connections have access to the required information in the next stages.
2. Run the define steps in order. The four steps key steps identified in section 2.2 were: defining objects, specifying kinematic relationships, determining the loads acting within a system, and imposing constraints on a model. A definition step involves the following sub-steps starting from the root model:
   (a) Define submodels. A model first calls the define step for each of the submodels, causing a depth-first traversal. A depth-first traversal is required as parent models may use properties of submodels and not the other way around.
   (b) The model defines itself. If applicable it triggers the define step of connections. The define steps in connections are initiated by the parent models manually, as there can be dependence from the connection to the parent model and vice versa. An example is that the disc should be oriented before the location of the wheel center with respect to the ground can be computed.
   (c) Define load groups. The model calls the define step for each of the load groups associated with it.[8]

Upon completion of the above steps, all relationships within the model are defined. At this stage, the end-user can export the model to a system instance of `sympy.physics.mechanics`.

3.5. Rolling Disc Example

The rolling disc is a model posing most of the problems one may encounter when designing such a framework. This section discusses the implementation of a rolling disc model in *BRiM*. The same formulation of a rolling disc is used as in section 2.2. The description of the model is found above the free body diagram of a rolling disc, figure 2.1. This section starts with a general overview of all the components in section 3.5.1. The subsections that follow explain each type of component: the models, connections, and loads. A more detailed explanation including exemplifying code is provided in appendix C. Section 3.5.5 finishes by building the model and giving a step-by-step overview of the model definition workflow.

[8] Note that connections and load groups never initiate a define step of a model.

3.5.1. Components Overview

As depicted in figure 3.7, the rolling disc model (`RollingDisc`) consists of two models, one connection and a set of load groups. Each body is represented by a separate model, as the bodies are expected to be modular. In this case, the ground is modeled as a flat ground (`FlatGround`), but one can also use a ground with a slope. The wheel model describes the inertial properties and the shape of the disc, in this case a knife-edge wheel (`KnifeEgeWheel`). To describe the interaction between the wheel and the ground, a non-holonomic tyre model (`NonHolonomicTyre`) is utilized. Reasons for this have also been discussed in section 3.4.2. A load group is used to apply a driving, rolling, and steering torque. Gravity is applied after creating the final system instance.

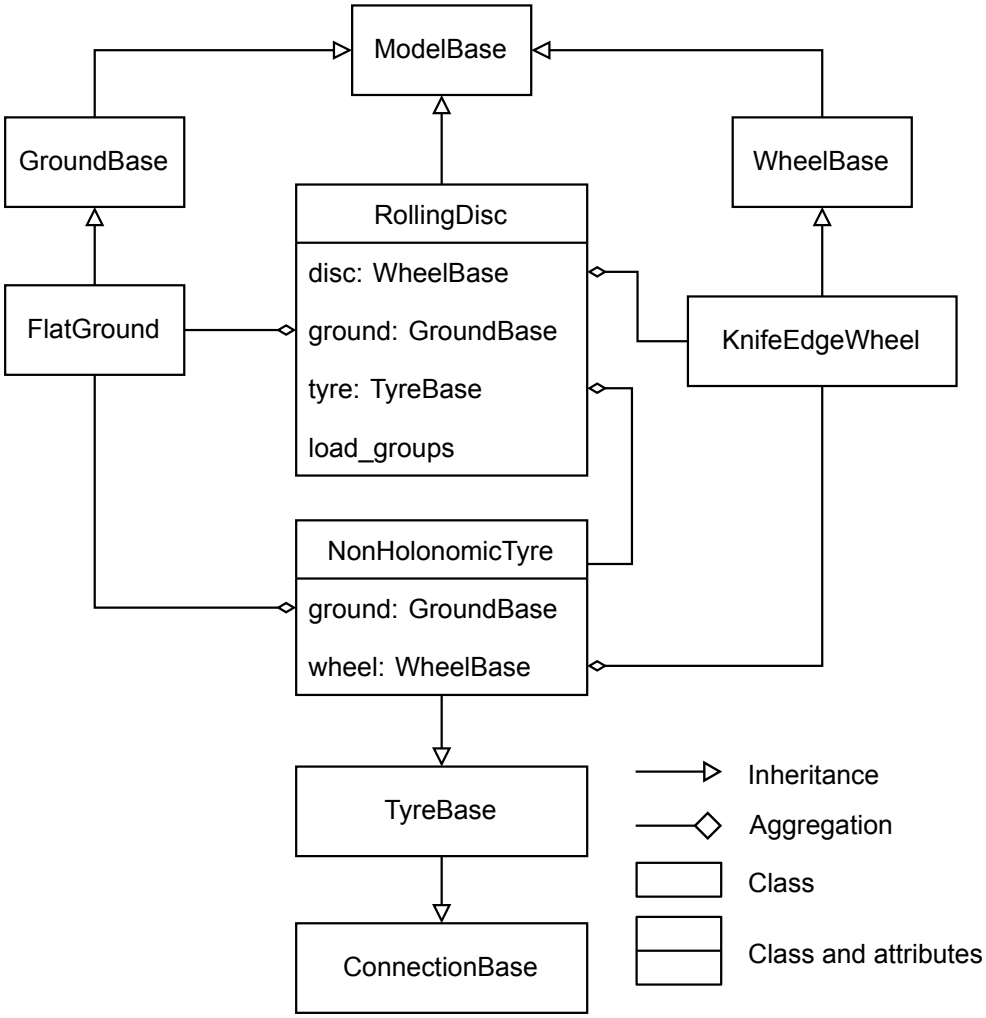


Figure 3.7: UML diagram of the rolling disc.

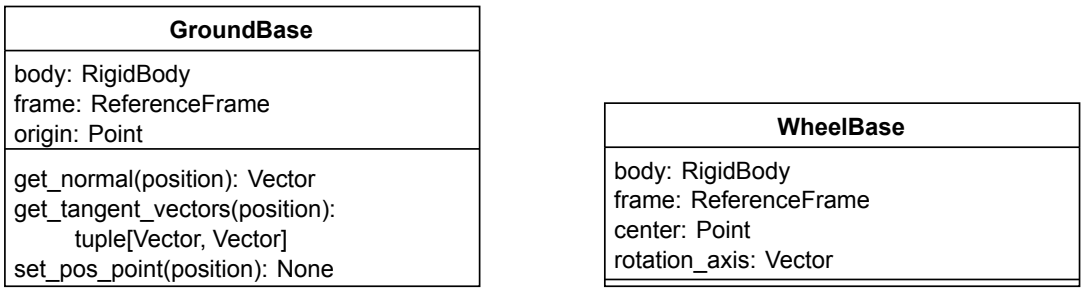
3.5.2. Models

The flat ground inherits from the abstract class `GroundBase`, which defines the interface for every ground model. This approach allows the rolling disc and other models to interact with different ground models in a similar manner. Each ground shares three attributes and three methods, as shown in figure 3.8a. The three attributes, namely a body, frame and origin, are defined by `GroundBase`, such that subclasses only have to implement three methods. A first method gets the positional dependent normal vector (`get_normal`), another gets the positional dependent tangent vectors (`get_tangent_vectors`) and the last sets the position of a point on the ground. One may denote that a normal vector is by definition enough to define a plane locally. However, the tangent vectors method is included to ease the interface for other classes, which may also result in simpler equations.

The flat ground implements these three methods accordingly. Its `get_normal` method returns the unit vector in the negative Z-direction as normal for every position. The X- and Y-unit vectors are always

returned as tangent vectors. In `set_pos_point` the position of a point is set in the XY plane.

Similar to the flat ground, the knife-edge wheel inherits from the abstract class `WheelBase`, shown in figure 3.8b. This abstract class has already created a body and a body-fixed frame to represent the wheel. Besides those, it prescribes two properties to be implemented by subclasses: the rotation axis, and the wheel center. The knife-edge wheel implements both of these with the rotation axis being the Y-unit vector and the center being the center of mass. To describe the shape, the knife-edge wheel only defines a symbol representing the radius. The actual computation involving the shape to compute the contact point is done within the tyre connection.



(a) Properties of the ground base model.

(b) Properties of the wheel base model.

Figure 3.8: Overview of classes' additional attributes and methods. The top cell shows the class name, the second the attributes, and the third the methods.

3.5.3. Connections

`TyreBase` is the abstract base class to describe the interaction between the ground and the wheel. The abstract class to describe the interaction between the ground and wheel is `TyreBase`. It creates for both the ground and wheel a property, which accepts any type of ground and wheel. It also creates a property for the contact point assuming a single contact point by default. Since the computation of the contact point is shared among all tyre models, the abstract class also implements a method to set the position of the wheel's center with respect to the contact point. This method takes into account what type of wheel and ground has been specified. A last property that is specified by the abstract tyre class is one to specify whether the contact point is defined to be in the ground plane by the parent model, in this case the rolling disc model.

With the implementation of `TyreBase` the nonholonomic tyre only has to update some of the define steps. In the method for defining the kinematics it computes the contact point utilizing the implemented method for it from `TyreBase`. The constraints to ensure pure-rolling are set in the "define constraints" method utilizing the properties of the wheel and ground models to perform the computations.

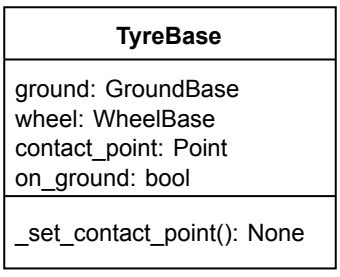


Figure 3.9: Overview of the additional attributes and methods of the tyre abstract base class. The top cell shows the class name, the second the attributes, and the third the methods.

3.5.4. Loads

The load group to apply the control torques to the rolling disc has a straightforward implementation. It only specifies the parent's type, as the `parent` property has already been implemented in the abstract class for load groups. Additionally, it defines the symbols representing the torque quantities when defining the objects and the torques acting upon the wheel in the loads' definition phase.

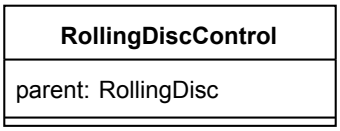


Figure 3.10: Overview of the additional attributes and methods of the rolling disc control load group. The top cell shows the class name, the second the attributes, and the third the methods.

The most convenient method to apply gravity on the system is to use the `apply_gravity` method within `System`, as implemented in `sympy.physics.mechanics`. This does not result in any problems, as there is no dependency on gravity of other model definitions. In most cases, it does actually not form any problems to define a load after exporting the model of *BRiM* to a single system instance, which relates to the fact that the "define loads" stage could be chosen as the last phase (see section 2.2.4). In this case, the torques applied within a load group could also just have been applied afterward.

3.5.5. Model Definition Steps

The user can set up the model using the code below:

```
# Import the required objects
from appendix import FlatGround, KnifeEdgeWheel, NonHolonomicTyre, RollingDisc, RollingDiscControl

rolling_disc = RollingDisc("rolling_disc")
rolling_disc.ground = FlatGround("ground")
rolling_disc.wheel = KnifeEdgeWheel("wheel")
rolling_disc.tyre = NonHolonomicTyre("tyre")
rolling_disc.add_load_groups(RollingDiscControl("control"))
```

Listing 3.1: Code configuring the rolling disc model.

The model is further defined, by calling the define method on the rolling disc model:

```
rolling_disc.define_all()
```

Listing 3.2: Code defining the rolling disc.

This expands to the following traversal:

1. Call `rolling_disc.define_connections`.
   (a) Perform a depth-first traversal of the submodels to define their connections. In this case, none of the submodels has any connections, so nothing happens.
   (b) The rolling disc model specifies the wheel and ground property of the tyre connection.
2. Call `rolling_disc.define_objects`.
   (a) The ground and wheel model define their objects, such as bodies and symbols.
   (b) The rolling disc defines its own objects and calls the tyre to also define its objects. It also sets the `on_ground` property of the tyre to true, as it will define the contact point to be in the ground plane by definition.
   (c) The control load group defines its objects, which are only symbols in this case.
3. Call `rolling_disc.define_kinematics`.
   (a) The ground and wheel model define their own kinematic relationships. Only the known points fixed to their frames are set to zero.
   (b) The rolling disc defines the orientation of the disc with respect to the ground and locates the contact point in the ground plane. Afterward, it manually calls the tyre to define its kinematics.
      i. The tyre sets the position of the wheel's center with respect to the contact point.

   (c) The control load group has no kinematic relationships to define.

4. Call `rolling_disc.define_loads`.

   (a) The ground and disc have no loads to define.
   (b) The rolling disc defines no loads and calls the tyre to define its loads, which are also none.
   (c) The load group defines the control loads on the disc.

5. Call `rolling_disc.define_constraints`.

   (a) The ground and disc have no constraints to define.
   (b) The rolling disc model has no constraints of its own and calls the tyre connection to define its constraints.
      i. The rolling disc defines the nonholonomic constraints to ensure no slip between the disc and the ground.
   (c) The load group has no constraints to define.

Once the model has been defined, it can be exported to an instance of `System`. Gravity can be applied to the system, and the EOMs can be formed using the following code:

```
10 from sympy import Symbol
11
12 system = rolling_disc.to_system()
13 system.apply_gravity(Symbol("g") * system.z)
14 system.form_eoms()
```

Listing 3.3: Code exporting the model to a single system and forming the EOMs.

This concludes forming the EOMs for the rolling disc using the core of *BRiM*.

3.6. Summary

This chapter proposes a generally applicable framework to formulate a model fulfilling several identified requirements. The most important requirements can be summarized as that the framework must allow for defining and reusing models, each of which can describe a system in varying complexity in an encapsulated manner, such that the models can be attached together to describe a bigger system.

The previous chapter identified how to work independently on different parts of the graphs describing a system. The framework utilizes this principle by dividing a model into smaller modular submodels creating a tree structure. To get the full required functionality, three core components must be used: models to describe (sub)systems, connections to describe modular and reusable interactions between subsystems, and load groups to define sets of forces and torques acting upon a (sub)system.

Overall, the framework is both intuitive as well as useful also for other applications. It is intuitive because dynamicists are used to thinking of complex systems in parts. For example, an excavator consists out of an arm and a moving base, each of which can be further subdivided. And it is useful, as it is a novel method to algorithmically describe a system with multiple layers of accessible abstractions. Thereby forming a strong basis to expand *BRiM* to implement bicycle-rider models.

4

Bicycle-Rider Modeling

BRiM Models

Over the past 150 years, numerous bicycle models have been proposed [36, 69]. These models have been instrumental in research across various areas, including investigations into bicycle stability [57, 71, 73], control strategies [22, 88], and more. However, creating new bicycle models or extending existing ones to account for specific aspects can be a laborious task. As highlighted by Schwab and Meijaard [69], formulating the EOMs for a bicycle model is prone to errors. To assist researchers in finding and creating their own models more easily, it is beneficial to leverage the general framework proposed in chapter 3 to develop bicycle-rider models. Another advantage of using such a modular framework is that it allows to more easily incorporate more complex rider models, which are currently missing in literature.

Where chapter 3 introduces a modular framework for modeling mechanical systems in general, this chapter focuses on utilizing this framework as the core for creating **B**icycle-**Ri**der **M**odels (*BRiM*). In summary, the modular framework employs a tree representation for modeling, allowing a bicycle-rider model to be split into separate models for the bicycle and the rider. Each of these models represents a mechanical system on its own and can be further divided into smaller, more manageable models. Parent models can employ connections to define modular interactions between submodels if desired. Additionally, load groups are utilized to define additional loads for the system.

Building upon this modular framework, this chapter proposes a set of modular models to be implemented including some additional features to improve the usage of *BRiM*. Section 4.1 starts with the additional requirements to *BRiM*, when implementing bicycle-rider models for research purposes. Followed by a description of what methods are used in developing *BRiM* to satisfy the development method requirements in section 4.2. Section 4.3 provides an explanation on which models have been implemented. The two additional features on parametrization and visualization are discussed in sections 4.4 and 4.5. To test the model's performance section 4.6 creates a benchmark comparing *BRiM* to manually generated EOMs using *SymPy*. Finally, section 4.7 summarizes the chapter.

4.1. Requirements

As the bicycle-rider modeling modules of *BRiM* build upon the core, the requirements set in section 3.1 automatically apply. However, there are additional functional requirements for the bicycle-rider modeling modules:

- *BRiM* must have a **component library** that includes pre-implemented models for bicycles in combination with riders. The specifics on the required models is discussed in section 4.3.

  > *As an end-user I want pre-implemented models that I can combine to create my desired bicycle-rider models.*

- *BRiM* should have an utility to import **parametrization** sets of models.

As an end-user I want to be able to get a dictionary mapping all bicycle and rider parameters to a value, based on a bicycle parameter data set.

- *BRiM* could incorporate a **visualization** utility to depict the model.

  As an end-user I want to easily visualize my bicycle and rider, for example using `matplotlib`*.*

There are also several requirements from a researcher's perspective on how *BRiM* should be developed. The user stories in the following list also include researchers as stakeholders combining the stakeholder groups of end-users and modelers. This last set of requirements aims to set the values used in the development of *BRiM*:

- *BRiM* must be **open-source**.

  As a researcher I want to be able to share my models and use other researchers' models. This would improve research dissemination and reproducibility.

- *BRiM* must be as easy as possible to **install**.

  As researcher I want to install BRiM without any hassle.

- *BRiM* should be **backward compatible**.

  As a researcher I want my code to produce the keep working across different versions of BRiM.

- *BRiM* should have a **contributing** guide.

  As a new developer I want to know how I can contribute to BRiM.
  As a researcher I want to contribute my new models to expand the component library of BRiM.

- *BRiM* could use **issue templates**.

  As a researcher I want to know how to best format my bug report or feature request.
  As a developer I want the issues to be properly formatted, such that it is easy to classify and understand an issue.

- *BRiM* could use **pull-request templates**.

  As a developer I want the pull-requests to be properly formatted providing a clear overview what is being modified.

- *BRiM*'s development method must aim to **reduce bugs** and other errors.

  As a researcher I want no bugs, because those may also lead to incorrect results.

- *BRiM*'s models must be as **verified** as possible.

  As a researcher I want the models within BRiM to be valid.

- *BRiM*'s code base should be as **readable** as possible.

  As a researcher I want to be able to understand the code defining my model.
  As a developer I want the code base of BRiM to follow proper formatting guidelines.

- *BRiM* should be as well **documented** as possible.

  As a developer I want to understand the code.
  As a researcher I want to understand the models and know what model I should use.

4.2. Development Method

BRiM encompasses several development tools and values to satisfy the development requirements proposed in section 4.1. Most inspiration has been drawn from the articles about setting up a Python project by Jolowicz [25] and Schmidt [67].

To improve *BRiM*'s availability it has been released open-source on GitHub[1], while only utilizing other open-source tools. *BRiM* uses *Poetry* [58], a modern package management system, to simplify installation. An open request to make *BRiM* available on the Python Package index (PyPi)[2] should further improve *BRiM*'s ease of installation in the future. Because it is highly inefficient to support backwards compatibility for software that is still in an experimental phase, *BRiM* will prefix a zero in its version number as long as it cannot guarantee backward compatibility.

BRiM also has several automated workflows in its Continuous Integration and Continuous Delivery (CI/CD). These workflows ensure that only code, which satisfies various requirements, can be added to *BRiM*. A first workflow checks the code quality using *Ruff* [65], a fast Python linter. This checks the code on simple errors, consistent code style, documentation, and much more. Another workflow runs all of the automated unit tests for different Python versions. These tests are also required to result in a 100% line coverage, such that there is no line in *BRiM*'s code base which is not tested. This is the major method to reduce the number of bugs in *BRiM*. This at the same time impacts the verification of *BRiM*'s models, which is further improved by also numerically testing the Whipple bicycle implementation against literature.

BRiM also automatically builds its documentation using *Sphinx* in one of the workflows and deploys it online[3], such that it can be used as a reference when creating models using *BRiM*. The online API reference is fully auto-generated based on the source code, as the linter already makes sure that everything is documented. To reduce the learning curve of *BRiM*, a separate repository *brim-examples*[4] has been created to provide users of examples on how to use *BRiM*.

4.3. Models

This section discusses the several models that have been implemented in *BRiM*. The bicycle-rider model can be split up into two submodels: a bicycle model and a rider model. Section 4.3.1 first discusses why and how the Whipple bicycle model according to Moore's convention [49] has been implemented. Followed by an overview of the division of the implemented rider model in section 4.3.2. Section 4.3.3 finishes on how these two sets of models, namely the bicycle and the rider, are combined to form a bicycle-rider model.

4.3.1. Bicycle Models

The Carvallo-Whipple bicycle model [11, 91] is widely recognized as the lowest order bicycle model with reasonable experimental validation [31, 49, 69, 75]. To align with previous research *BRiM* must implement a Carvallo-Whipple bicycle. Meijaard et al. [42] introduced a specific parameter set for a linearized Whipple bicycle which is used widely throughout the bicycle industry. However, Peterson [56] points out that there are several problems with using this specific parametrization. When forming the EOMs for a linearized model it is sensible to express the parameters with respect to the reference configuration. However, using the same parameter choice for nonlinear EOMs is cumbersome with the requirements of several intermediate quantities to be introduced. Besides that there is also a coupling between the parameters, meaning that a change of the front wheel radius will also result in changes to other parts of the bicycle. Therefore, it is best to choose a parametrization that is configuration independent and more suitable for nonlinear EOMs. The chosen convention which satisfies these conditions is the one presented by Moore [49].

[1] *BRiM*'s repository is: `https://github.com/TJStienstra/brim`.
[2] https://github.com/TJStienstra/brim/issues/37 is the open issue in *BRiM* on releasing it on PyPi.
[3] *BRiM*'s online documentation is available at `https://tjstienstra.github.io/brim/`.
[4] The *brim-examples* is available at: `https://github.com/TJStienstra/brim-examples`.

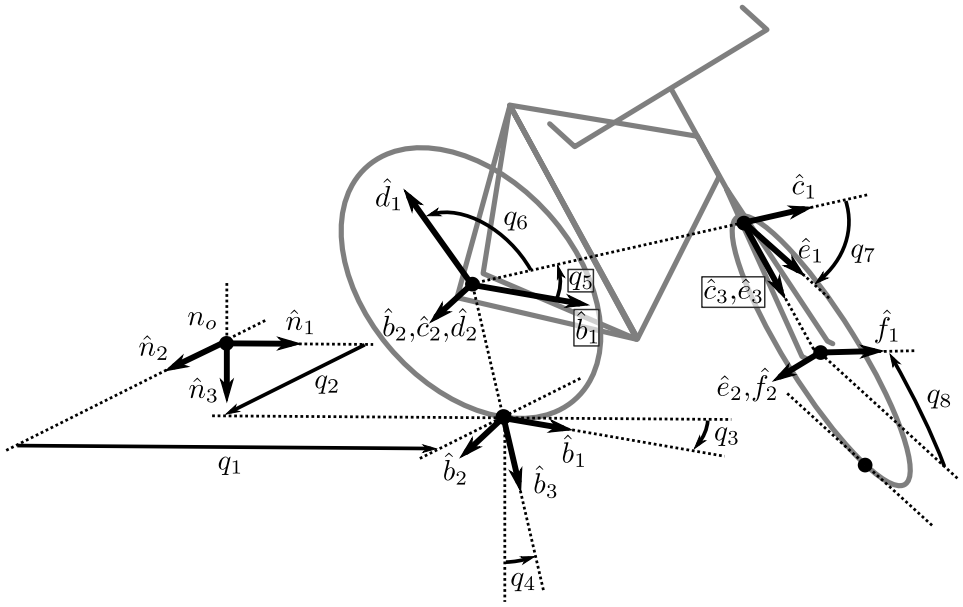


Figure 4.1: General configuration of the Whipple bicycle model based on Moore's convention [49].

An illustration of the general configuration of the Whipple bicycle model, based on Moore's convention, is shown in figure 4.1. The default Whipple bicycle model consists of a ground and four bodies: rear wheel, rear frame, front frame, and front wheel. Each body is assumed to be rigid and interconnected by pin joints. The rear wheel's contact point is defined within the ground plane, and the rear frame is oriented in yaw-roll-pitch rotation relative to the ground. The front wheel's contact point is constrained to the ground using a holonomic constraint, and nonholonomic constraints are applied to both wheels to assume no-slip conditions. Further details on the default Whipple bicycle model's definition are found in [49]. Literature provides several extensions for the Whipple bicycle model. An overview of the most regular modifications is given by Schwab and Meijaard [69] and Limebeer and Sharp [36]. These extensions include effects of among others structural flexibility of the front and rear frame, the transverse radius of a wheel by approximating the wheel with a torus, and tyre models.

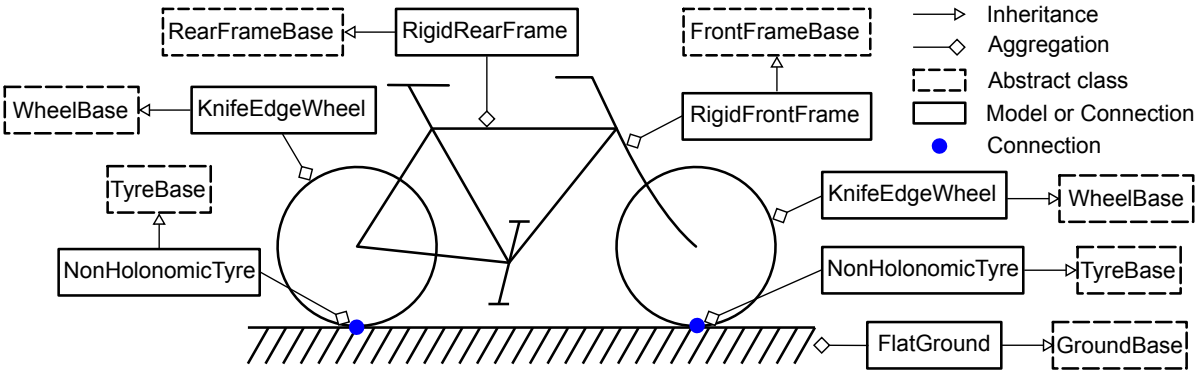


Figure 4.2: Visualization of the components of the Whipple bicycle model according to Moore's convention in *BRiM*.

To facilitate a modular approach it is advantageous to split the bicycle model into separate models for each body, similar to the rolling disc example discussed in section 3.5. This division not only aligns with intuitive modeling but also provides practicality, as most extensions are defined at the body level. For instance, the toroidal wheel extension modifies the wheel shape, and modeling the structural flexibility of the rear frame is best achieved within a rear frame model. This division results in a total of five submodels: ground, rear wheel, front wheel, rear frame, and front frame. Both the rear wheel and front wheel models are wheel models. The interaction between the wheels and the ground can be addressed using a tyre connection. As the connections between the bicycle's four parts consist of simple pin joints, which do not require modification for common extensions, these connections are

handled by the overarching Whipple bicycle model. A visual overview of the division of the Whipple bicycle model is shown in figure 4.2.

4.3.2. Rider Models

The rider models found in literature can overall be subdivided in two categories: upper-body models and pedaling models. Upper-body models generally range in complexity from an inverted pendulum, like [57, 68], to a passive rider model including the arms to incorporate inertial effects on the stability, like [49, 70]. Differently from the upper body models, the pedaling models generally use a stationary bicycle. The pedaling models can be classified into two categories: planar models, like [8, 13, 27, 28, 32, 52, 54, 66, 78, 61, 80, 84, 86], and non-planar models, like [7, 33, 38, 43, 53, 63, 77, 81, 92, 96][5]. Planar models focus solely on leg movements within a single plane, whereas non-planar models account for leg movements that extend beyond a single plane.

A visual overview of the division of the rider model is shown in figure 4.3. The rider is composed of six segments: pelvis, torso, left leg, right leg, left arm, and right arm. This segmentation is intuitive and allows enough freedom to highly simplify parts of the model if required. Modeling the pelvis and torso separately has several reasons. The models in literature show that the lower and upper body are commonly modeled separately. However, if a full-body model is desired, then it is useful to have a single shared component, which is the pelvis. Additionally, *OpenSim* also uses the pelvis as root node in its model description, which follows a tree topology approach [19].

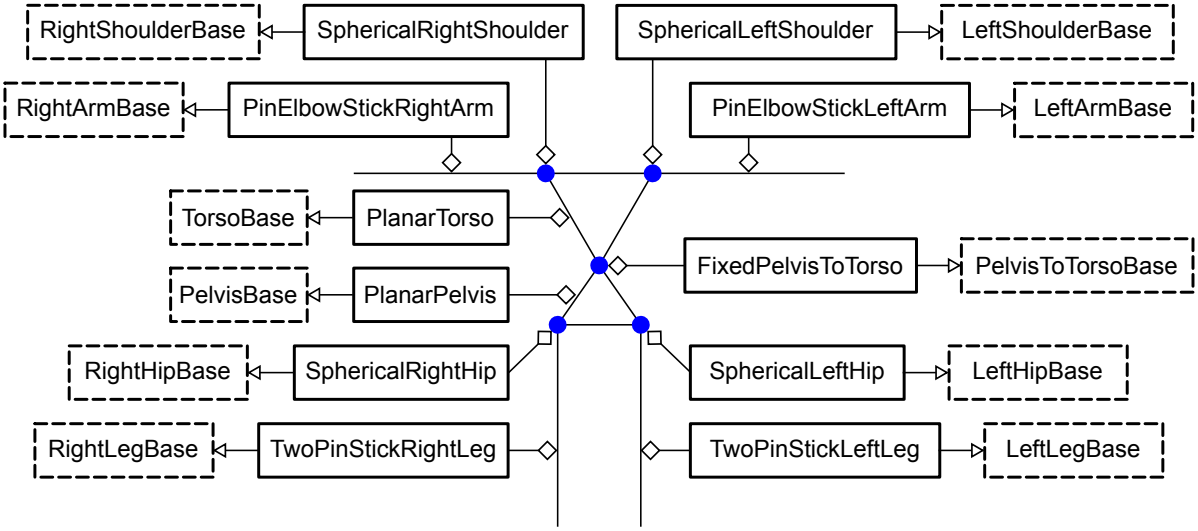


Figure 4.3: Visualization of the components of the rider model in *BRiM*. A legend is shown in figure 4.2.

Connections are used for various joints: hips, shoulders, and one between the pelvis and torso. The reason behind most of them is that information on both models is required, while it is at the same time desirable to have each of these joints modular. An example is the hip joint, which uses the pelvis as well as the leg. It also can be modeled in vastly different ways: a simple pin joint only allowing hip flexion, a spherical joint, or even a custom joint incorporating the translation as well.

[5] Notable is that most of these non-planar pedaling models are created in *OpenSim* [7, 33, 43, 53, 63, 77, 81, 92].

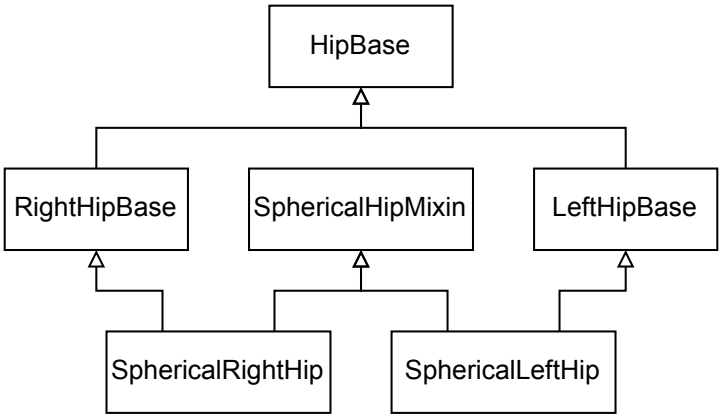


Figure 4.4: UML diagram of the spherical hip joints, to illustrate how symmetry within the rider is modeled.

Due to symmetry of the human body, it has been chosen to use the class design shown in figure 4.4. This UML diagram shows the inheritance of the left and right spherical hip joints from their respective base classes, along with the implementation of shared details through a mixin. To ease the connections between the various body parts and the bicycle, a consistent orientation of the reference frames has been adopted. Specifically, all frames align with the ground frame, with the Z-axis oriented in the inferior direction and the X-axis in the posteroanterior direction.

4.3.3. Bicycle-Rider Models

The bicycle-rider model combines both a bicycle model and a rider model using three connections: one to connect the pelvis with the saddle, one for the pedals and feet, and a last one for the steer and hands. This is also visualized in figure 4.5. As the kinematic chain of the full model should always be connected and acyclic, as explained in section 2.2.2, only one of the connections should describe the orientation and position of the rider with respect to the bicycle. The optimal connection is the one between the rear frame and the pelvis, as the pelvis is included in both lower and upper body models of the rider[6].

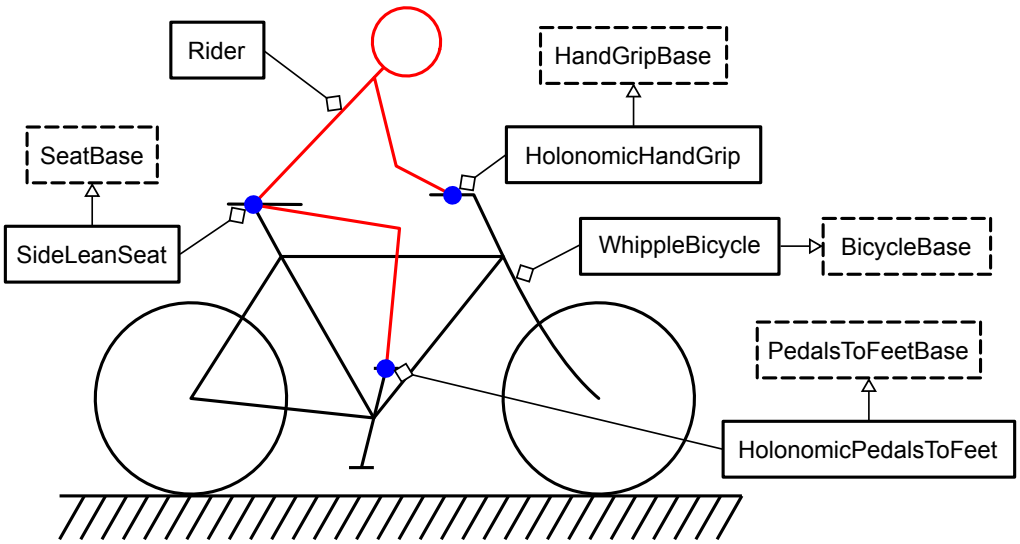


Figure 4.5: Visualization of the components of the bicycle-rider model in *BRiM*. A legend is shown in figure 4.2.

The other two connections can be made in different ways. Two approaches that can be seen in literature to connect the feet to the pedals are: via holonomic constraints [8, 13, 84] and springs [54], possibly including dampers. These same approaches can also be used for connecting the hands to

[6]The other connections could also be used to connect the two kinematic chains.

the front frame. Optionally, additional holonomic constraints can be added to force the arms to hang down [49].

4.4. Parametrizations

As mentioned in the requirements (section 4.1), the integration of parameter sets derived from bicycle-rider measurements is highly advantageous in *BRiM*. Because the EOMs generated by *BRiM* require numerical values of various parameters such as masses and lengths for simulation. While it is possible to manually set each of these parameters, it is common practice to utilize parameter sets based on experimental measurements. One notable library that facilitates this process is the open-source *BicycleParameters* Python library [47] developed by Moore [49]. This library is based on several experimental methods [30, 31, 51] and offers a convenient way to obtain parameter values for bicycles. The *BicycleParameters* library incorporates an object-oriented implementation of Yeadon's model [18], which estimates the inertial properties of human body segments using a geometric method with reasonable accuracy [94].

In *BRiM*, a tree traversal over all objects is performed to obtain a dictionary that maps the symbols to their respective values. During this traversal, each node is provided with the corresponding parameter set object obtained from the *BicycleParameters* library and outputs a mapping. These mappings are combined, populating the dictionary with constants that map to their respective values. This approach allows each model or connection in *BRiM* to implement an independent mapping that aligns with the modularity of the framework.

4.5. Visualization

To facilitate the visual inspection of generated models or simulations, *BRiM* incorporates a simple visualization utility. This utility builds upon *SymMePlot*, an open-source Python package specifically designed and enhanced during this project to visualize mechanical systems created using the mechanics module of *SymPy*. *SymMePlot* utilizes the lightweight and widely used library *Matplotlib* as backend with a tree-based architecture, as depicted in figure 4.6. The root of the tree is a plotter to which various objects can be added, such as bodies, frames, and points. Each of these so called plot objects can utilize other plot objects as children. An example is a frame, which uses three plot objects to represent its unit vectors. The leaf nodes of the tree are artists from *Matplotlib*, which are added to the plot to visualize the system. Overall, this results in a simple and expandable design, which leverages tree traversals for code generation of expressions and computation of coordinates.

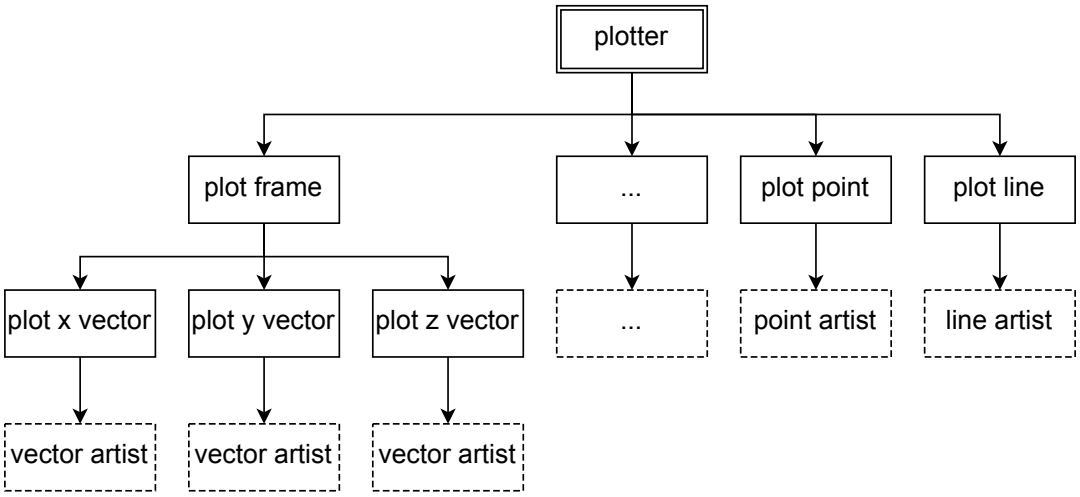


Figure 4.6: Illustration of the tree structure utilized in *SymMePlot*. The double-lined rectangle is the root plotter instance. The single-lined rectangles are the children associated with certain mechanical objects. Their visual representation, the dashed rectangles, are artists from *Matplotlib*.

SymMePlot offers several advantages that make it an ideal choice for *BRiM*. Firstly, it features an intuitive and minimal interface, ensuring a user-friendly experience. Secondly, it leverages the widely-

used and well-supported *Matplotlib* library as its backend, providing robust and versatile plotting capabilities. Thirdly, it employs a tree structure to organize all objects, aligning well with the tree structure utilized in *BRiM*. While there are additional advantages to using *SymMePlot*, these three are the most notable ones.

To integrate *SymMePlot* into *BRiM*, a dedicated plotter utility has been developed. This plotter inherits the plotter object from *SymMePlot* and extends its functionality to support the inclusion of *BRiM*'s models in the visualization tree. Currently, *BRiM* employs a centralized setup, where a single function defines for each model how it should be visualized. While this approach offers the advantage of considering other models on the same level of the tree during plotting, it does not align with the decentralized structure utilized by *BRiM*. Therefore, a feature request[7] is open to switch to a decentralized design, where each model and connection describe how it can be visualized. Resulting in a similar implementation to the parametrization.

4.6. Benchmark

As stated in the performance requirements of *BRiM* in section 3.1, the number of operations in the EOMs and their computation time should be minimized. The most important of these is the reduction of the number of operations, because that directly impacts the simulation and optimization time. On the other hand, the computation time to form the EOMs is a cost, which is only paid once. When benchmarking the number of operations in the EOMs it is crucial to not only consider the number of operations before CSE, but especially after CSE. This is because the evaluation and manipulation is optimally based on CSEd expressions. While this section does not review the weighted cost of operations or the numeric stability of the equations, these aspects are important to keep in mind as well.

This benchmark section focuses on two models. Section 4.6.1 benchmarks the implementation of the rolling disc in *BRiM* against several manually generated EOMs using `sympy.physics.mechanics`. This initial benchmark has been utilized to improve the implementation of the tyre connections in *BRiM* to make it more closely match the performance of the best manually generated EOMs. Optimizing the rolling disc is advantageous, because the definition of the rear wheel highly influences the number of operations of the Whipple bicycle model. Section 4.6.2 benchmarks the Whipple bicycle model following Moore's parametrization convention for various implementations, including the one in *BRiM*.

4.6.1. Rolling Disc

The increased number of operations in the EOMs generated using *BRiM* can be attributed to two main reasons. First, certain model simplifications that are applicable only to specific model compositions cannot be made. For instance, modeling a rolling disc requires five generalized coordinates and speeds if the tyre model is not known. However, if it is known that the wheel is modeled as pure-rolling, then only three generalized coordinates and speeds are required. Second, some information is unknown by models, due to decoupling. An example is that a tyre connection does not know all details on how the velocity of the wheel has been set with respect to the ground, therefore it is restricted to mainly depend on the position graph of the points.

The rolling disc benchmark, of which the model is described in section 2.2, identifies three major factors influencing the number of operations after CSE[8][9]:

1. The usage of an intermediate yaw-roll frame with respect to the ground to compute the position of the disc's center from the contact point, instead of a normalized double cross product. This reduces the number of operations in the CSEd EOMs from 445 to 180.
2. Manually setting the velocity of the disc's center in the ground frame based on the instantaneous center of rotation, instead of computing the derivative of the position vector. This further reduces the number of operations in the CSEd EOMs further from 180 to 107.
3. The usage of only three generalized coordinates and speeds in combination with an efficient

[7] #66 is the open feature request in *BRiM*

[8] Significant reductions in the number of operations, as discussed by Mitiguy and Kane [46], can be achieved by choosing efficient generalized speeds, but this reduction is observed only if no intermediate frame is used to determine the position of the disc's center relative to the contact point.

[9] The number of operations mentioned in this list is computed using manually generated EOMs for which the code can be found in the `benchmarks/test_rolling_disc.py` file.

choice of the generalized speeds, instead of five generalized coordinates and speeds. This reduces the number of operations in the CSEd EOMs from 107 to 52[10].

To account for the first influence an additional property has been added to `TyreBase`, by which the parent can optionally specify the axis from the contact point to the disc's center. The second reduction can also be implemented, as the computation of the nonholonomic constraints has been made independent of the velocity graph of the points. The third has not been implemented. Both, because it is system specific, as explained earlier. And, because it would alter the system size in terms of number of generalized coordinates and speeds, which could be counterintuitive and inconvenient for the end-user when comparing multiple models. An overview of the benchmark results is presented in table 4.1.

Model (applied optimizations)	Computation time $\mu \pm \sigma$ (ms)	Number of operations before CSE	Number of operations after CSE
Manual ()	1152 ± 33	19643	445
Manual (1)	244 ± 13	680	180
Manual (1, 2)	216 ± 28	511	107
Manual (1, 2, 3)	135 ± 8	85	52
BRiM (1,2)	278 ± 23	505	103

Table 4.1: Benchmark results for the rolling disc over 50 runs on a *Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz* and *BRiM* commit: ec02a0a7852b15de2f37e0630e2af962f69bba9d.

4.6.2. Whipple Bicycle

This benchmark uses the Whipple bicycle model following Moore's, as explained in section 4.3.1, convention in all cases. The implementation of the model in *BRiM* is compared to two other implementations, each of which also has been validated using the values computed by Basu-Mandal [2]. The first is the manual implementation by Moore [48] himself using `sympy.physics.mechanics`. In this implementation, Moore uses the minimal number of generalized coordinates (4) and speeds (6) required. The second implementation was written by myself at the beginning of this project. This implementation utilizes eight generalized coordinates and speeds, like *BRiM*, while also aiming for EOMs with a low number of operations.

Model	Computation time $\mu \pm \sigma$ (s)	#Operations before CSE	#Operations after CSE
Manual by Moore [48]	3.5 ± 0.2	230789	2198
Manual by myself	6.6 ± 0.3	390554	2389
BRiM	5.8 ± 0.2	468290	2176

Table 4.2: Benchmark results for the Whipple bicycle model over 50 runs on a *Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz* and *BRiM* commit: ec02a0a7852b15de2f37e0630e2af962f69bba9d.

The benchmark results of *BRiM* for the Whipple bicycle model, as presented in table 4.2, reveal interesting insights. While the implementation by Moore [48] emerges as the top performer in most aspects of the benchmark, *BRiM* reaches matching performance in terms of the number of operations after CSE. This outcome holds significant importance, as the number of operations after CSE directly impacts the per-evaluation cost of the EOMs in optimization and simulation. Moreover, *BRiM* achieves this despite having approximately double the number of operations before CSE[11]. Additionally, while the implementation by Moore [48] outperforms *BRiM* in computation speed, it is noteworthy that *BRiM* surpasses my own manual implementation from the beginning of the project, which is most probably due to a strong correlation between the number of operations in the EOMs and the computation time. Overall, these results indicate that while *BRiM* may have a slightly longer computation time than an optimized script for forming the EOMs, it excels in reducing the number of operations after CSE, closely approaching the efficiency of the manual implementation.

[10] In this case, the efficient choice for the generalized speeds is significant, as it causes an extra reduction from 72 to 52.

[11] The reason for this significant difference has not yet been identified.

4.7. Summary

This chapter applies the core of *BRiM* proposed in the previous chapter to create bicycle models, which are extendable with riders. The implemented bicycle model is the Carvallo-Whipple bicycle following the convention by Moore [49]. The bicycle model is broken down into five models: a ground, rear frame, front frame, rear wheel, and front wheel. The tyre models defining the interaction between the wheels and the ground are implemented as connections. The rider model to be attached to the bicycle is composed of six models: pelvis, torso, two legs, and arms. To offer both modularity and increased decoupling between the models, the shoulders and hips are implemented as connections. To combine both the bicycle and rider model into a bicycle-rider model a total of three connections are employed: a connection between the rear frame and pelvis, between the feet and pedals, and between the hands and steer.

To optimize the implementation of these models a benchmark has been created. Through the benchmark of a simple rolling disc the weak spots of the formulation in *BRiM*'s models were identified and solved. The final benchmark of the Whipple bicycle model shows *BRiM*'s excellent performance in generating EOMs having the same level of efficiency of manually optimized equations. These results show how *BRiM* with its bicycle-rider models offers a great potential for solving simulation and optimization tasks.

5

Optimization and Simulation with BRiM

This chapter showcases the effectiveness of *BRiM* for academic research purposes by solving a trajectory tracking problem for a rolling disc and simulating several models including an advanced bicycle-rider model. The basis of the models utilized in this chapter is presented in chapter 4, which applied the framework from chapter 3 to propose a modular structure for defining bicycle-rider models.

This chapter is structured as follows. Section 5.1 starts by simulating the Whipple bicycle, while showing the various features from chapter 4. Section 5.2 expands on this by simulating an advanced upper-body bicycle-rider model, showcasing the comprehensive capabilities of *BRiM* in modeling and analyzing complex mechanical systems. Finally, section 5.3 presents a trajectory tracking problem for a rolling disc. The model of the rolling disc is created using *BRiM* and solved using the direct collocation library *Pycollo*.

5.1. Whipple Bicycle Simulation

Previous studies [2, 49] have shown various derivations of the nonlinear EOMs for the Carvallo-Whipple bicycle model [11, 90]. Basu-Mandal [2] verifies the equivalence of his two derivations of the EOMs by computing the acceleration for arbitrary generalized coordinates and speeds. This section showcases the workflow of deriving the EOMs of the Whipple bicycle using *BRiM* and using them in a forward simulation. The model is also being verified in the test suite of *BRiM* against the values used by Basu-Mandal [2] to prove its mathematical equivalency.

Contrary to the derivation of the EOMs using manual methods, the derivation using *BRiM* is relatively easy. The used model structure is described in section 4.3.1. Listing 5.1 shows the code to compute the EOMs. It starts with importing the necessary objects and configuring the bicycle model. The next step is to establish all relations within the bicycle model and export it to a single instance of `System`. Next, it specifies gravity and a steer torque between the rear and front frame. The only required step before forming the EOMs is to choose which generalized coordinates and speeds are independent and which are dependent. Now that everything is defined Kane's method [26] can be employed to form the EOMs[1].

[1] The implementation of Kane's method in *SymPy* solves the velocity constraints. The default solve method is LU solve. However, as this method is sensitive to zero divisions, due to the usage of symbolics, it is advisable to use a solver based on Cramer's rule. For more information see #24780.

```python
# Import required objects, * imports all objects from the specified module
from brim import *
import sympy as sm
import sympy.physics.mechanics as me

# Configure the bicycle model
bicycle = WhippleBicycle("bicycle")
bicycle.front_frame = RigidFrontFrame("front_frame")
bicycle.rear_frame = RigidRearFrame("rear_frame")
bicycle.front_wheel = KnifeEdgeWheel("front_wheel")
bicycle.rear_wheel = KnifeEdgeWheel("rear_wheel")
bicycle.front_tyre = NonHolonomicTyre("front_tyre")
bicycle.rear_tyre = NonHolonomicTyre("rear_tyre")
bicycle.pedals = SimplePedals("pedals")  # Used to beautify the visualization
bicycle.ground = FlatGround("ground")
# Define the model
bicycle.define_all()

# Create a sympy system object of the model
system = bicycle.to_system()
# Apply additional forces and torques to the system
g = sm.symbols("g")
system.apply_gravity(-g * bicycle.ground.get_normal(bicycle.ground.origin))
steer_torque = me.dynamicsymbols("steer_torque")
system.add_actuators(me.TorqueActuator(
    torque=steer_torque, axis=bicycle.rear_frame.steer_axis,
    target_frame=bicycle.front_frame.frame,
    reaction_frame=bicycle.rear_frame.frame))
# The dependent and independent variables need to be specified manually
system.q_ind = [*bicycle.q[:4], *bicycle.q[5:]]
system.q_dep = [bicycle.q[4]]
system.u_ind = [bicycle.u[3], *bicycle.u[5:7]]
system.u_dep = [*bicycle.u[:3], bicycle.u[4], bicycle.u[7]]

# Simple check to see if the system is valid
system.validate_system()
system.form_eoms(constraint_solver="CRAMER")  # LU solve may lead to zero divisions
```

Listing 5.1: Code building the default Whipple bicycle model and forming its EOMs.

As explained in section 4.4 *BRiM* integrates support for the *BicycleParameters* [47] library to parameterize models. Its usage is shown in the code below, where the model parameters are determined. The currently used parameterization set is that of the *Batavus Browser*, a Dutch style city bicycle. However, one can easily change it to the *Batavus Stratos Deluxe*, a Dutch style sport city bicycle, by changing the string `"Browser"` to `"Stratos"`. Both of these bicycles have been measured by Moore [49].

```python
# Import required libraries under an alias
import bicycleparameters as bp
import numpy as np

data_dir = r"./data"  # Path to the parametrization data
# Load in the parametrization set of the Batavus Browser bicycle
bike_params = bp.Bicycle("Browser", pathToData=data_dir)
# Determine a mapping from the symbols to the numerical values of the bicycle
constants = bicycle.get_param_values(bike_params)
# Add missing values
constants.update({
    g: 9.81,
    # Assign random small values to the pedals, as they are just used for plotting
    bicycle.symbols["gear_ratio"]: np.random.random() * 1E-10,
    **{sym: np.random.random() * 1E-10 for sym in bicycle.pedals.symbols.values()},
})
```

Listing 5.2: Code determining a mapping of the bicycle parameters. Parameterization data has been measured by Moore [49] and can be downloaded from the *BicycleParameters* repository.

The only data, which is now required are the initial conditions and the control inputs. In the code below the initial conditions are set such that the bicycle starts in a rolled position with a forward velocity of $3\,\mathrm{m\,s^{-1}}$. The steering torque is specified as a simple proportional gain controller based on the roll rate. The maximum torque is set to $10\,\mathrm{N\,m}$, which gets applied when the absolute roll rate equals $0.2\,\mathrm{rad\,s^{-1}}$ or higher.

```
# Set all generalized coordinates and speeds to zero
initial_conditions = {xi: 0 for xi in system.q[:] + system.u[:]}
# bicycle.q[4] and bicycle.u[0] are both set to improve the initial guess, when solving
# the constraints.
initial_conditions.update({
    bicycle.q[3]: 0.2,  # Roll angle of the rear frame.
    bicycle.q[4]: 0.314,  # Pitch angle of the rear frame.
    bicycle.u[0]: 3,  # Rear contact point velocity in the X-direction.
    bicycle.u[5]:
        -3 / constants[bicycle.rear_wheel.radius],  # Rear wheel angular velocity.
})

# Specify control as a function in the form f(t, x), where t is time and x is the state.
roll_rate_idx = len(system.q) + system.u[:].index(bicycle.u[3])
max_roll_rate = 0.2
max_torque = 10
controls = {
    steer_torque:
        lambda t, x: max_torque * max(-1, min(x[roll_rate_idx] / max_roll_rate, 1))
}
```

Listing 5.3: Code specifying the initial conditions and the control input.

While there has not been added an appropriate object within *BRiM*'s utilities to run a simulation of a model, an experimental version has been implemented as a utility in the examples repository of *BRiM*[2]. In the code below this object is utilized to solve the initial conditions during the initialization phase and integrate the system over a time span of 2.5 seconds using *SciPy*'s `solve_ivp` function as backend. However, one could also select a DAE solver, like IDA, implemented in *scikits.odes* by using `simulator.solve(np.arange(0, 2.5, 0.01), solver="dae")`.

```
# Import Simulator from the utilities at brim-examples
from utilities import Simulator

# Setup simulator
simulator = Simulator(system)
simulator.constants = constants
simulator.initial_conditions = initial_conditions
simulator.controls = controls
# Initialize simulator: code generates the equations and solves the initial conditions.
simulator.initialize()
# Integrate system with a timestep of 0.01 seconds to get smoother plots.
simulator.solve((0, 2.5), t_eval=np.arange(0, 2.5, 0.01))
```

Listing 5.4: Code integrating the EOMs of the Whipple bicycle model.

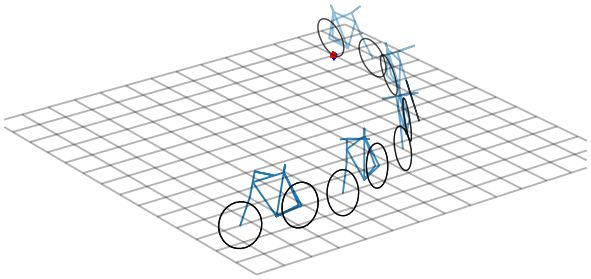
With the system solved visualizations can be created. Due to the development of *SymMePlot* and its integration within *BRiM* (see section 4.5), it is easy to create an animation of the simulation as the code below shows. A time-lapse version of the resulting animation is shown in figure 5.1a.

[2] *BRiM* examples repository is available at: `https://github.com/TJStienstra/brim-examples`.

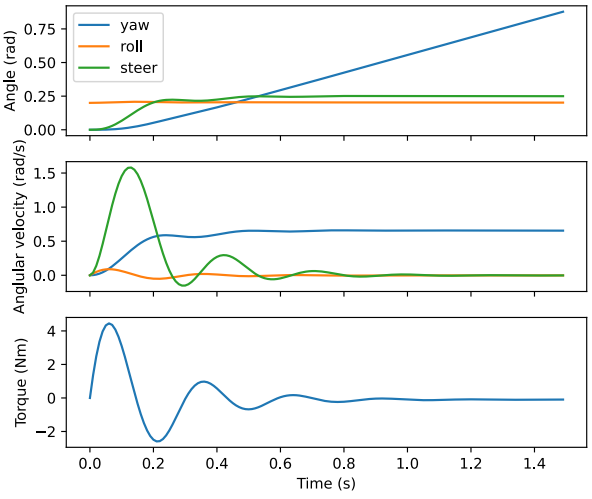
```python
# Import required funtions to create the animation
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from brim.utilities.plotting import Plotter
from scipy.interpolate import CubicSpline

def animate(fi):
    """Update the plot for frame i."""
    plotter.evaluate_system(x_eval(fi / n_frames * simulator.t[-1]), p_vals)
    return *plotter.update(),

# Create a cubic spline to interpolate the state variables.
x_eval = CubicSpline(simulator.t, simulator.x.T)
# Create the figure and the plotter object.
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
plotter = Plotter.from_model(ax, bicycle)
# Set up the plotter for evaluating the coordinates and plot the initial state.
p, p_vals = zip(*simulator.constants.items())
plotter.lambdify_system((system.q[:] + system.u[:], p))
plotter.evaluate_system(simulator.x[:, 0].flatten(), p_vals)
plotter.plot()
# Plot the ground plane.
X, Y = np.meshgrid(np.arange(-1, 6.5, 0.5), np.arange(-1, 6.5, 0.5))
ax.plot_wireframe(X, Y, np.zeros_like(X), color="k", alpha=0.3, rstride=1, cstride=1)
# Orient the figure and set the limits.
ax.set_xlim(X.min(), X.max())
ax.set_ylim(Y.min(), Y.max())
ax.view_init(-157, -143)
ax.set_aspect("equal")
ax.axis("off")
# Create the animation using the animate function.
fps = 30
n_frames = int(fps * simulator.t[-1])
ani = FuncAnimation(fig, animate, frames=n_frames, blit=False)
ani.save("animation.gif", dpi=300, fps=fps)
plt.show()
```

Listing 5.5: Code creating an animation of the simulated Whipple bicycle model.

Figure 5.1 shows the results of the simulation, which are conform to the expected. The bicycle starts with a roll angle to the right and the P controller quickly stabilizes the bicycle resulting in a steady-state turn to the right. The offset in the yaw rate and roll angle is sensible, because the P controller aims for a zero roll rate, not for a straight path. If one were to add a term in the control of the steering torque based on the yaw rate or roll angle, then it should stabilize the bicycle to follow a straight path.



(a) Time lapse of a 2.5 seconds simulation.



(b) Plots of the state and steer torque.

Figure 5.1: Results of the simulation of the default Whipple bicycle model with the parametrization set of the *Batavus Browser*.

The process of defining and simulating the default Whipple bicycle is relatively easy and has a smooth integration with third-party libraries. However, the strength of *BRiM* comes also in modifying and extending the bicycle model. Modifying the bicycle to take into account a toroidal shape of the wheels instead of the knife-edge shape is as simple as changing the utilized class for the wheel from `KnifeEdgeWheel` to `ToroidalWheel`. The only other change is that the constants for the radius and transverse radius of the wheel have to be specified manually[3]. Further extension of the model to include a rider are discussed in the next section.

5.2. Upper-Body Bicycle-Rider Simulation

While the previous section exemplifies the use of *BRiM* to generate the Whipple bicycle model, this section takes a step further by demonstrating the application of *BRiM* to extend it to a sophisticated bicycle-rider model. The model in this section builds upon the rider arms extension by Moore [49]. In his rider extension he rigidifies the torso to the rear frame and models the arms with additional constraints such that their configuration is fully determined by the steer angle. The model utilized in this section goes a step further by introducing side lean of the rider and using linear torsional spring-dampers to control the joints of the rider. By conducting a forward simulation of this advanced model, *BRiM* showcases its versatility and capabilities in handling complex and detailed models. Thereby, highlighting the potential of *BRiM* in generating models that lead to valuable insights into the behavior and control of bicycle-rider models.

5.2.1. Model Description

The model presented in this section combines the Whipple bicycle model, as discussed in section 4.3.1, with an upper-body model. An overview of the components of the model is shown in figure 5.2. The connection between the rear frame and the pelvis of the rider is established through pure side lean, actuated by a spring-damper. The arms of the upper body consist of two bodies: the upper arm and the forearm. The shoulder is modeled as a spherical joint, while the elbow is represented as a pin joint. Both joints are actuated using spring-dampers. Additionally, six holonomic constraints are employed to connect the hands to the steer. The resulting model has a total of ten Degrees of Freedom (DOF), seven holonomic constraints, four nonholonomic constraints, and nine linear torsional spring-dampers for actuation. The torque T_i exerted by each spring-damper is defined as:

$$T_i(t) = -k_i(t)(q_i(t) - q_{eq,i}(t)) - c_i(t)\dot{q}_i(t) \tag{5.1}$$

where k_i denotes the stiffness coefficient, c_i the damping coefficient, q_i the joint angle and $q_{eq,i}$ the equilibrium angle. Each of these values is set by the controller. The controller used in the forward simulation is simple, with constant stiffness and damping coefficients. Also, the equilibrium angles of the spring-dampers for shoulder abduction and rotation are set at constant values. The equilibrium angle for the side lean equals the opposite of the roll angle of the bicycle, such that the rider tries to stay upright. As for the shoulder and elbow flexion, those are controlled based on the roll angle such that they steer the bicycle toward the falling direction.

5.2.2. Workflow

To generate the results, the model is first built using *BRiM*, which takes approximately 85 seconds. Next, the model parameters are retrieved using a parameter set from *BicycleParameters*, and the initial conditions are determined. To perform the forward simulation and evaluate the EOMs, it is essential to convert the symbolic expressions into numeric functions for fast evaluation. This process takes around 390 seconds[4]. An additional method to accelerate the evaluation is to compile this function using *Numba*, which converts the native Python function into machine code, taking an additional 435 seconds[5]. For numerical integration, the widely used function `solve_ivp`[6] from *SciPy* is employed, which utilizes an explicit Runge-Kutta method of order 5(4) with adaptive time-stepping by default. The

[3] The parameterization by Moore [49] does not measure the transverse radius and is therefore together with the radius not obtained when running `get_param_values`.

[4] The code generation utilized is integrated within *SymPy* itself and creates a standard Python function with *NumPy*.

[5] The compilation using *Numba* enhanced the forward simulation speed by approximately four times.

[6] For longer simulations, it is recommended to use a DAE solver to mitigate the accumulation of constraint violations. However, in this short simulation, the constraint errors remain sufficiently low.

forward simulation following this setup, while covering a time span of 2.5 seconds, requires only 0.7 seconds to complete[7].

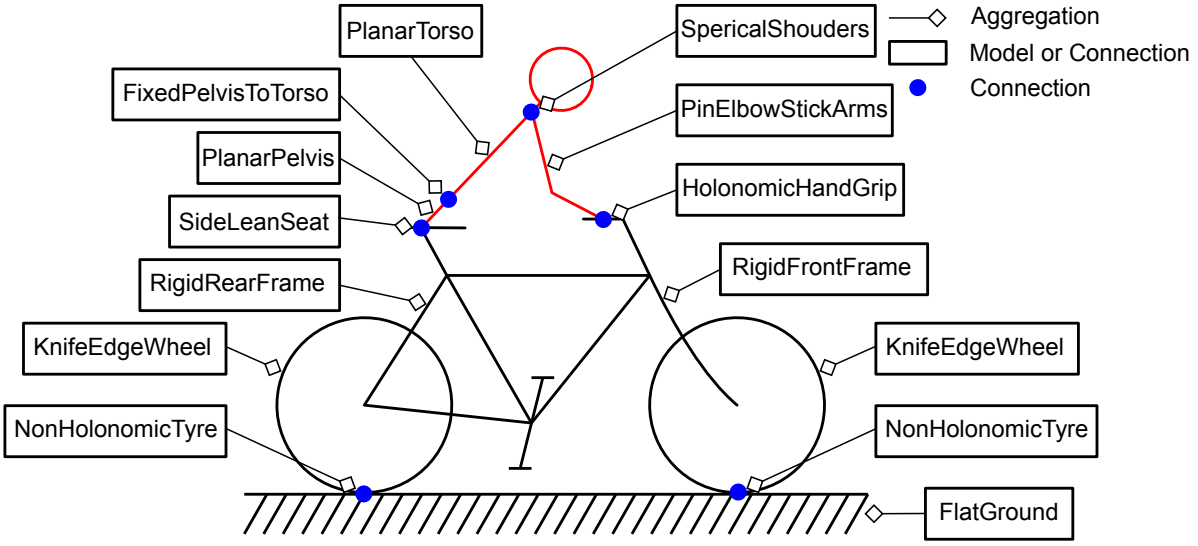
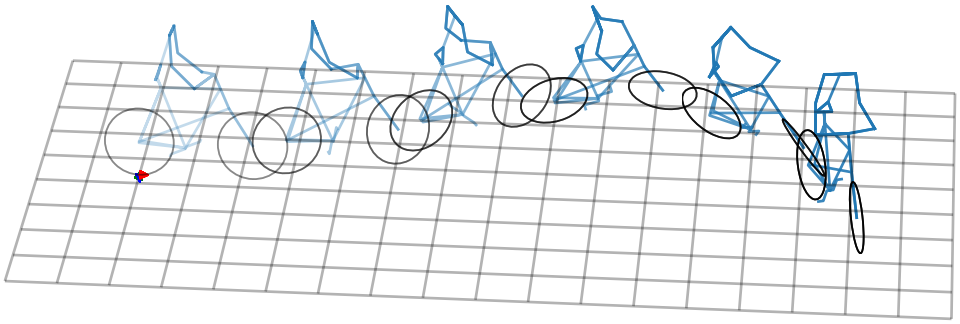


Figure 5.2: Visualization of the components of the simulated upper-body bicycle-rider model in *BRiM*.

5.2.3. Results

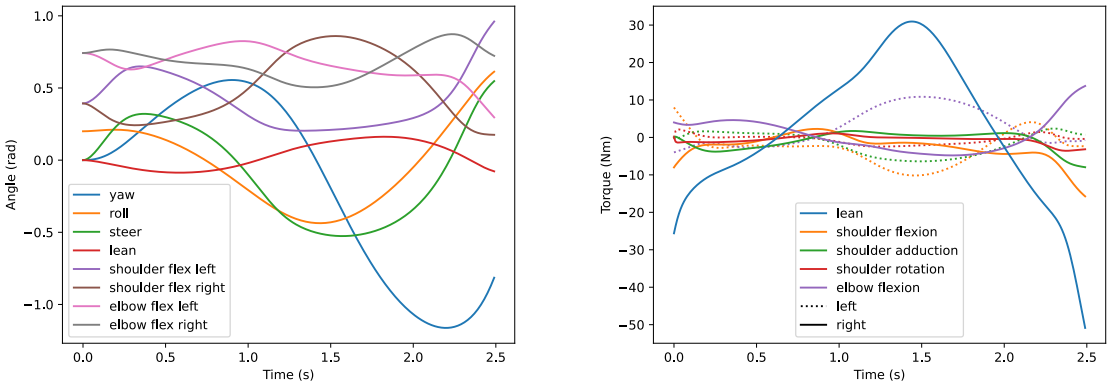
The results of the forward simulation of the bicycle are depicted in figure 5.3. The simulation begins with the bicycle in a rolled position and a relatively low velocity of only 3.4 m/s. As anticipated, the rider steers into the fall direction to stabilize the bike. However, due to the use of a poorly calibrated proportional gain controller, the rider over-steers and ends up having to steer toward the opposite side as it becomes the falling direction. Another observation is that the rider manages to maintain an upright posture for the most part. As figure 5.3c shows, all of this control is accomplished within reasonable torque margins, which is exertable by a human. Overall, these results highlight the value of *BRiM* in facilitating real-time simulations, supporting further research in also rider control and stability.



(a) Time-lapse of the bicycle-rider, created using *SymMePlot*.

Figure 5.3: Results of the forward simulation of a bicycle-rider model.

[7]The approximate computation time of each step of this workflow has been measured in a Jupyter Notebook on an *Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz*.



(b) State angles.

(c) Control torques using spring-dampers.

Figure 5.3: Results of the forward simulation of a bicycle-rider model (cont.).

5.3. Trajectory Tracking of a Rolling Disc

Biomechanical simulation studies are frequently formulated as Optimal Control Problems (OCPs) [9, 85], an example of which are data tracking problems [37, 54]. This section illustrates how *BRiM* can be used to solve OCPs. In particular, a trajectory tracking problem of a rolling disc following a sinusoidal path, because it is an intuitive and relatively cheap problem to solve. Solving this problem not only demonstrates the seamless integration of *BRiM* with other scientific tools, but also lays a foundation for solving similar problems for bicycle or bicycle-rider models.

The numerical methods to solve an OCP can be classified into two categories: indirect methods and direct methods [4, 6]. Indirect methods define optimality conditions, which get discretized and solved numerically [29]. Direct methods on the other hand directly discretize the OCP converting it to a nonlinear programming problem [62]. While indirect methods give a more accurate solution, they are more difficult to construct and solve [29].

Direct methods can be further subdivided into direct shooting and direct collocation methods [62]. Direct shooting methods discretize the control and use forward simulation to evaluate the objective function every iteration, while direct collocation methods discretize both the control and state using polynomial splines [10]. Direct shooting methods are most effective for OCPs with simple control trajectories while requiring high dynamic accuracy [29, 62], but they are sensitive to the initial conditions [9]. In contrast, direct collocation is considered more powerful [62] and has been shown to have faster convergence for complex problems, like those in biomechanical simulations [59].

There are several OCP software programs implementing direct collocation [3, 5, 10, 55, 87]. *Pycollo* stands out among these packages as it has been developed with biomechanical modeling in mind [10]. *Pycollo* is an open-source direct orthogonal collocation library in Python. Direct orthogonal collocation is a type of direct collocation which uses higher order polynomial splines resulting in better accuracy and computational performance [15, 29]. Another huge advantage of *Pycollo* is its integrated support for *SymPy*.

5.3.1. Problem Description

The objective in the OCP is to find the minimal control torques to follow a periodic sinusoidal path with a rolling disc. The rolling disc itself is modeled as a knife-edge wheel subject to pure rolling on flat ground. To control the disc three torques are applied: a driving torque (T_{drive}) about the rotation axis, a steering torque (T_{steer}) about the axis going through the contact point and the center of the disc, and a roll torque (T_{roll}) about an axis perpendicular to both the normal of the ground and the rotation axis. A further explanation of its implementation in *BRiM* is provided in section 3.5.

Lin and Pandy [37] and Park, Caldwell, and Umberger [54] both formulated the objective function of their data tracking problem as multi-objective, namely as the weighted sum of the squared control and squared tracking error. In case of the rolling disc the multi-objective function is defined as:

$$w_t \underbrace{\int_t \Big(\sin\left(q_1(t)\right) - q_2(t) \Big)^2 dt}_{\text{tracking term}} + w_c \underbrace{\int_t \Big(T_{drive}(t)^2 + T_{steer}(t)^2 + T_{roll}(t)^2 \Big) dt}_{\text{control term}} \tag{5.2}$$

where q_1 and q_2 are the x and y-position of the contact point in the ground plane, and w_t and w_c are the tracking and control weights. $\sin\left(q_1(t)\right) - q_2(t)$ is the description of the path that the disc should follow.

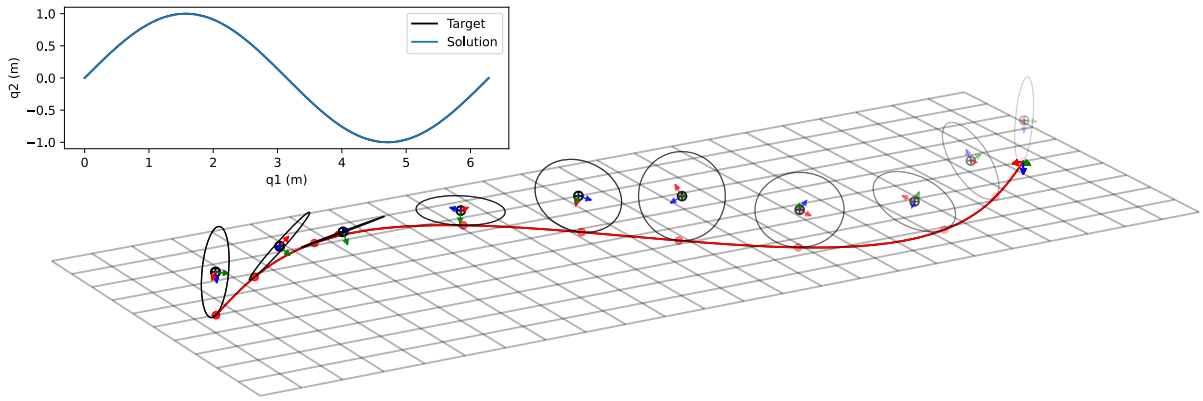
The difficulty with multi-objective functions is the selection of the weights. A too low weight for the tracking would result in a straight line with no torques being applied. And a too high weight on the tracking would result in high tracking accuracy, but also excessive control torques. In the case of the rolling disc the tracking weights have been selected such that w_c equals the aimed tracking term and w_t equals the expected control term. The aimed tracking term w_t can simply be computed by choosing a mean tracking error, which is squared and integrated over time[8]. The tracking weight w_t can either be estimated or experimentally determined as it should approximately be equal to the control term.

5.3.2. Workflow

The OCP has been implemented in the following steps. Firstly, *BRiM* forms the EOMs of the rolling disc model. Next, *Pycollo* solves the OCP using the EOMs as path constraints[9]. Path constraints are constraints that are applied throughout the time interval of an OCP. Finally, *Matplotlib* plots the results, while *SymMePlot* creates an animation to visualize the result[10].

5.3.3. Results

The results of the solved OCP are presented in figure 5.4, yielding the following observations. Firstly, the disc follows the sinusoidal path closely and smoothly. Secondly, both figures are anti-symmetric, as is expected when following a periodic sinusoidal path. Additionally, the driving torque remains close to zero as the disc does not require acceleration in the forward direction. Fourthly, a high roll angle is used, which is in line with intuition when taking a sharp turn at high speed. A last observation, which is consistent with the control literature on bicycles, is that the steering torque is the main contributing load. Even so, that if the optimization is performed with no drive and roll torque, then all graphs will remain highly similar.



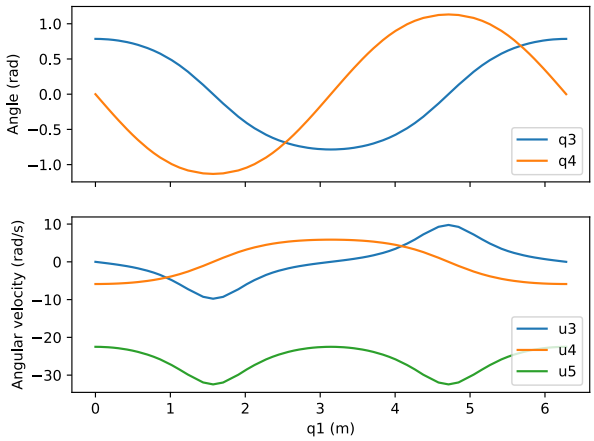
(a) Contact point trajectory and a time-lapse of the rolling disc, created using *SymMePlot*.

Figure 5.4: Results of the trajectory tracking problem of a rolling disc following a periodic sinusoidal path. For reference, the disc has a radius of 0.3 m and is modeled with a uniformly distributed mass of 1.0 kg. The duration of following a single period of the sinusoidal is 1.0 s with an aimed mean tracking error of 0.01 m.

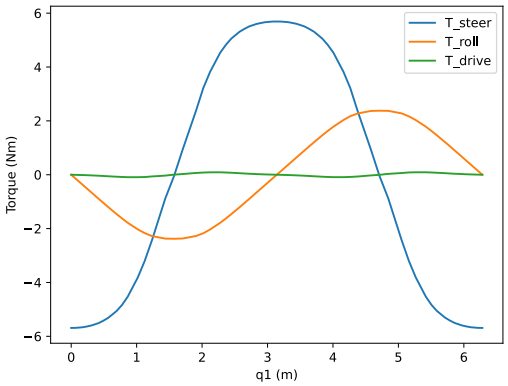
[8]The integration of the squared mean tracking error over time equals the squared mean tracking error multiplied by the integration interval.

[9]The main reason for using path constraints is that solving the analytic EOMs for the accelerations is rather costly resulting is a significant increase in the number of operations. This is especially a problem for more complex systems like a bicycle.

[10]An animation of the results can be found in the brim-examples repository on GitHub.



(b) Graphs of the state. The upper graph shows the yaw (q_3) and roll (q_4) angle. The rotation angle of the disc about its rotation axis (q_5) is left out for improved readability of the figure. The lower graph shows the yaw rate (u_3), roll rate (u_4), and rotation speed (u_5).



(c) Control torques.

Figure 5.4: Results of the trajectory tracking problem of a rolling disc following a periodic sinusoidal path. For reference, the disc has a radius of 0.3 m and is modeled with a uniformly distributed mass of 1.0 kg. The duration of following a single period of the sinusoidal is 1.0 s with an aimed mean tracking error of 0.01 m (cont.).

6

Future Recommendations

BRiM is a novel design that I have made from scratch within a year. Starting with the identification of the requirements to the implementation and demonstration of *BRiM*. Within this process I have had to resolve various problems. Some of which were within *SymPy*, but also many within the design and implementation of *BRiM*.

However, there are still several areas in which improvement can be made. The best method to identify these is through the usage of *BRiM* within research projects, but there are already multiple that I would like to highlight. This chapter categorizes these in three groups. The first group is those, which consider the design and implementation within *BRiM* itself. The second is within *SymPy* and the third is towards third-party tools.

6.1. Recommendations for BRiM

While I have had various fruitful discussions with other researchers along the way, I would still highly recommend critical reviews from more researchers. These reviews should incorporate both the core as well as the models.

Though the core offers a flexible framework, which contributes to the high performance on the benchmarks discussed in section 4.6, it has become more complex than my initial aim. When explaining the structure and implementation of *BRiM* to others, I have noticed that people find it hard to understand. This complexity could be problematic, because it increases the learning curve and is more likely to cause mistakes leading to dysfunctional models. A main cause of the increased complexity is the aim to decouple models, as this requires both an unusual way of thinking by modelers and more code to implement to define a model. I especially experienced decoupling to be a challenge in the implementation of models when applying the optimization of components as discussed in the benchmark section, section 4.6.

When implementing the models I have made various decisions some of which I am still questioning. Within the division of the models there are strong arguments for both implementing the shoulder and the hip as connections as well as for implementing them as part of the arm or leg model. A number of arguments pro the usage of connections are discussed in section 4.3.2. However, my overall preference is to use a model over a connection if possible, which is here the case. Another design choice that I would revise is the choice of the default orientation of reference frames within the rider model. Jason Moore also issued in a review that it would be recommendable to follow the convention used in biomechanics for the orientation of the frames within the rider[1].

Related to reviewing is the further improvement of the test and benchmark suite. I would recommend to refactor the test suite by developing a set of test classes, which simplify and speed up the process of writing tests and thus also prevent possible mistakes by modelers. It would also be valuable to add benchmarks comparing the accuracy of models to experimental data. These benchmarks should rate models on various aspects[2]. This not only increases the validation of the models, but also helps researchers to assemble the perfect bicycle-rider model to fit their needs. To further aid researchers in

[1] #91 is the open issue requesting the change in the orientation of the reference frames within the rider components.

[2] An example is that one can neglect tyre slip on low speed [31].

this choice it is also advisable to enhance the documentation with recommendations on when to use which model.

BRiM currently lacks various common bicycle extensions, as I have focused on enhancing the structure of *BRiM*. Therefore, it is still required to implement several common bicycle extensions such as various tyre models, rear and front frame flexibility, and aerodynamic forces. Especially the implementation of tyre models may bring additional challenges as auxilary forces are required to compute the normal force. This should be possible to implement, as *BRiM* allows the computation of auxilary forces due to its decoupling. However, I have not implemented it due to time restrictions. Another important set of components that are currently missing in *BRiM* are load groups representing musculotendons, because those are still in development within *SymPy*.

A blindside I discovered late in this project is that it might be more advantageous to split the "define kinematics" step into a separate "define orientations" and "define positions" step. The graph representation in section 2.2.2 shows that the definition of the relations between frames is order-independent and independent of the point graphs. The position of points is also order-independent, but does rely on the definition of the frames. Therefore, it seems that invoking the definition of relationships between frames ("define orientations") before invoking the definition of relationships between points ("define positions") leads to complete order-independence. This would mean that the define steps of connections can be called in an automated traversal instead of manually by the parent model. However, this complete order-independence only upholds as long as relationships are not defined based on other relationships out of the same step. Nonetheless, this is possibly violated when redefining (angular) velocities or accelerations to reduce the final number of operations in the EOMs[3].

Another suboptimal design choice has been the centralized integration of *SymMePlot* within *BRiM*, i.e. the description of how each model should be plotted is in a single location. This does not align with the decoupling and extensibility values of *BRiM*. Therefore, I want to change the integration of *SymMePlot* within *BRiM* to decentralized, where each object describes its own visualization[4].

BRiM is aimed at computing the nonlinear EOMs. However, linearized equations are more suitable for some research projects. Therefore, it would be valuable to investigate the usage of *BRiM* in forming linearized EOMs. This can be done by linearizing the nonlinear EOMs[5], but this will result in suboptimal equations compared to literature as those will have been optimized manually. Therefore, it is more recommendable to research if a similar design can be utilized to define a model optimal for forming linearized EOMs.

6.2. Recommendations for SymPy

Within *SymPy* there are multiple ongoing projects from which *BRiM* would highly benefit. The first is the implementation of a separate biomechanics module[6]. This project aims to implement several objects mainly involving the implementation of a symbolic musculotendon model. Once this model and the accompanying objects have been implemented, I would recommend to start using those in *BRiM*.

A second is the integration of a numeric solver within Kane's method. As mentioned in appendix A.0.3, Kane's method performs a linear solve for the velocity constraints. This gives rise to two problems. Firstly, solving a symbolic linear system can easily cause zero divisions, because it is difficult to determine whether an expression could be zero[7]. The second problem is that solving big systems becomes too expensive. To counter the first problem a solver based on Cramer's rule has been implemented, which does not cause a zero division unless the matrix is singular[8]. However, only a numeric solver also solves the second problem by postponing the actual solve to the stage of evaluation.

My last recommendation is to implement the mobilizer joint proposed by Seth et al. [72] to model complex human joints. The current setup I have made in the joints framework within *SymPy* already uses several of the required concepts, like intermediate reference frames for each joint and DCMs to store the orientation. A main topic of research before implementing a mobilizer joint within *SymPy* is whether the entire mechanics module is compatible with the usage of splines within DCM or whether there is a better method to translate the experimentally measured joint movement.

[3] An example is the usage of the velocity one point or two point theorem[26].
[4] #81 is an open draft pull-request to implement this redesign.
[5] *SymPy* has already a linearizer implemented for Kane's method.
[6] #24240 is the active issue tracker of this project in *SymPy*.
[7] For more information see #24780 within *SymPy*
[8] This is true if and only if a division-free method is used to compute the determinant.

6.3. Recommendations for Third Party Tools

With already several third-party tools integrated in *BRiM*, the most important one which is still missing is a `Simulator` object[9]. This object must aim to bridge the gap between the EOMs within *SymPy*'s `System` object and its usage in third-party tools. The simulator must be able to solve the initial conditions automatically and provide numerical functions for third-party tools to evaluate the constraint violations and the EOMs. These functions should be as fast as possible with a variable choice in compilation time.

[9] The reason to call it a `Simulator` is that an initial version has been used for the forward simulations in section 5.2.

7

Conclusions

Bicycle dynamics research has faced the challenge of creating accurate mathematical models for investigating various aspects of bicycle behavior. The main existing approaches seem to either reuse the same well-documented but relatively simple models or to extend or define new models laboriously. Both of these approaches hinder the development of advanced bicycle-rider models and reduce research reproducibility and dissemination. To address this gap, the package *BRiM* has been developed, providing a modular and extendable framework to create symbolic **B**icycle-**Ri**der **M**odels.

BRiM offers a comprehensive solution by leveraging a modular and extendable design to define a model. It uses *SymPy*, an open-source Computer Algebra System, to symbolically describe the model and form the EOMs using Kane's method. With *SymPy* researchers can manipulate equations and modify a model with bodies and joints. *BRiM* also introduces a component-level description by adopting a tree structure, which divides the bicycle-rider model into smaller submodels, e.g. in a bicycle model and a rider model which are both subdivided further. These submodels define relations of the model parts within their respective system boundaries, such as the position of a body with respect to another. Through this design decision it is possible to easily change submodel types, for example from a knife-edge wheel to a toroidal shaped wheel, resulting in a modular overarching parent model. The relations between submodels are defined by the parent model, while utilizing connections to establish the complex relationships in a reusable and modular manner. Additionally, the framework introduces separate load groups, allowing modular choices of loads. Due to the usage of the tree structure, researchers can also extend a model to take into account another aspect, like adding a rider to an already existing bicycle model.

To prevent clashes when algorithmically defining a model *BRiM* utilizes an identified systematic approach to define the relations in and between models. The approach divides the description of a model into four stages, which are run in a depth-first traversal. The first step is to define all objects, like symbols and reference frames. The second phase is to establish the kinematic relationships, such as the position of a body. The third is to define the forces and torques acting on the system. The final step is the employment of the (non)holonomic constraints.

The implemented bicycle model is the Carvallo-Whipple bicycle [11, 90] following Moore's parametrization convention [49]. *BRiM* divides this model into five submodels: a ground, rear frame, front frame, rear wheel, and front wheel. Connections are utilized for the tyre models, as the interaction between a wheel and the ground has a relatively complex description, which should be both reusable and modular to support different tyre models. The rider model is divided into six submodels: pelvis, torso, and two arms and legs. Additional modularity is offered by using connections for both the shoulders and hips. Together the bicycle and rider model can be combined to form a bicycle-rider model by the employment of three connections: one between the rear frame and pelvis, another between the hands and steer, and a last one between the feet and pedals.

The effectiveness of *BRiM* has been demonstrated in several ways. Firstly, by using benchmarks showing that *BRiM* has comparable performance to laboriously manually optimized scripts, which form the EOMs. Secondly, by successfully solving a trajectory tracking problem of a rolling disc with the objective to minimize the applied torques. And thirdly, by showcasing forward simulations of bicycle-rider models. An advanced model even combines the Whipple bicycle model with a leaning upper

body model, which actuates the steer using linear torsional spring-dampers. The real-time simulation of these models and the seamless integration of *BRiM* with other scientific tools show the effectiveness of *BRiM* for simulation and optimization tasks.

Overall, the development of *BRiM* represents a significant advancement in the field of bicycle dynamics research. Its modular and extendable design, combined with its computational efficiency and integration capabilities, addresses the limitations of existing approaches and offers researchers a powerful tool for creating, sharing, and advancing bicycle-rider models. The application of *BRiM* promises to drive further insights into bicycle behavior and contribute to the development of improved bicycle models and rider control strategies.

References

[1] Leila Alizadehsaravi and Jason K. Moore. *Bicycle balance assist system reduces roll and steering motion for young and older bicyclists during real-life safety challenges*. en. Tech. rep. type: article. Engineering Archive, May 2023. DOI: 10.31224/2825. URL: https://engrxiv.org/preprint/view/2825 (visited on 07/10/2023).

[2] Pradipta Basu-Mandal. "Studies On The Dynamics And Stability Of Bicycles". In: Sept. 2007. URL: https://www.semanticscholar.org/paper/Studies-On-The-Dynamics-And-Stability-Of-Bicycles-Basu-Mandal/581e4b0a6da162f5e7100b0ddc53a3980493c87a (visited on 06/05/2023).

[3] Victor Becerra. *PSOPT*. 2011. URL: https://www.psopt.net/ (visited on 07/05/2023).

[4] John T. Betts. *Practical Methods for Optimal Control Using Nonlinear Programming, Third Edition*. Advances in Design and Control. Society for Industrial and Applied Mathematics, Jan. 2020. ISBN: 978-1-61197-618-2. DOI: 10.1137/1.9781611976199. URL: https://epubs.siam.org/doi/book/10.1137/1.9781611976199 (visited on 07/05/2023).

[5] John T. Betts. *SOS: Sparse Optimization Suite*. 2013. URL: https://www.astos.de/products/sos (visited on 07/05/2023).

[6] John T. Betts. "Survey of Numerical Methods for Trajectory Optimization". In: *Journal of Guidance, Control, and Dynamics* 21.2 (Mar. 1998). Publisher: American Institute of Aeronautics and Astronautics, pp. 193–207. ISSN: 0731-5090. DOI: 10.2514/2.4231. URL: https://arc.aiaa.org/doi/10.2514/2.4231 (visited on 07/05/2023).

[7] Rodrigo Rico Bini, Luke Daly, and Michael Kingsley. "Muscle force adaptation to changes in upper body position during seated sprint cycling". In: *Journal of Sports Sciences* 37.19 (Oct. 2019). Publisher: Routledge _eprint: https://doi.org/10.1080/02640414.2019.1627983, pp. 2270–2278. ISSN: 0264-0414. DOI: 10.1080/02640414.2019.1627983. URL: https://doi.org/10.1080/02640414.2019.1627983 (visited on 10/24/2022).

[8] Maarten F. Bobbert et al. "Effect of vasti morphology on peak sprint cycling power of a human musculoskeletal simulation model". In: *Journal of Applied Physiology* 128.2 (Feb. 2020). Publisher: American Physiological Society, pp. 445–455. ISSN: 8750-7587. DOI: 10.1152/japplphysiol.00674.2018. URL: https://journals.physiology.org/doi/full/10.1152/japplphysiol.00674.2018 (visited on 10/24/2022).

[9] Antonie J. van den Bogert, Dimitra Blana, and Dieter Heinrich. "Implicit methods for efficient musculoskeletal simulation and optimal control". en. In: *Procedia IUTAM*. IUTAM Symposium on Human Body Dynamics 2 (Jan. 2011), pp. 297–316. ISSN: 2210-9838. DOI: 10.1016/j.piutam.2011.04.027. URL: https://www.sciencedirect.com/science/article/pii/S2210983811000289 (visited on 06/12/2023).

[10] Samuel Brockie. "Predictive Simulation of Musculoskeletal Models Using Direct Collocation". en. PhD thesis. June 2022. URL: https://www.repository.cam.ac.uk/handle/1810/338642 (visited on 07/05/2023).

[11] Emmanuel Carvallo. *Théorie du mouvement du monocycle et de la bicyclette...* Gauthier-Villars, 1901.

[12] L.J. Richard Casius, Maarten F. Bobbert, and Arthur J. Van Soest. "Forward Dynamics of Two-Dimensional Skeletal Models. A Newton-Euler Approach". In: *Journal of Applied Biomechanics* 20.4 (Nov. 2004), pp. 421–449. ISSN: 1065-8483, 1543-2688. DOI: 10.1123/jab.20.4.421. URL: https://journals.humankinetics.com/view/journals/jab/20/4/article-p421.xml (visited on 06/12/2023).

[13] Giulio Cecchini et al. “Neural Networks for Muscle Forces Prediction in Cycling”. en. In: *Algorithms* 7.4 (Dec. 2014). Number: 4 Publisher: Multidisciplinary Digital Publishing Institute, pp. 621–634. ISSN: 1999-4893. DOI: 10.3390/a7040621. URL: https://www.mdpi.com/1999-4893/7/4/621 (visited on 10/24/2022).

[14] Agile Business Consortium. *The DSDM Agile Project Framework*. DSDM Consortium, Oct. 2014. URL: https://www.agilebusiness.org/dsdm-project-framework.html (visited on 06/16/2023).

[15] Christopher L. Darby, William W. Hager, and Anil V. Rao. “An hp-adaptive pseudospectral method for solving optimal control problems”. en. In: *Optimal Control Applications and Methods* 32.4 (2011). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/oca.957, pp. 476–502. ISSN: 1099-1514. DOI: 10.1002/oca.957. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/oca.957 (visited on 07/05/2023).

[16] DeepMind. *MuJoCo documentation*. Dec. 2022. URL: https://mujoco.readthedocs.io (visited on 12/15/2022).

[17] Scott L. Delp et al. “OpenSim: Open-Source Software to Create and Analyze Dynamic Simulations of Movement”. In: *IEEE Transactions on Biomedical Engineering* 54.11 (Nov. 2007). Conference Name: IEEE Transactions on Biomedical Engineering, pp. 1940–1950. ISSN: 1558-2531. DOI: 10.1109/TBME.2007.901024.

[18] Christopher Dembia, Jason K. Moore, and Mont Hubbard. “An object oriented implementation of the Yeadon human inertia model”. In: *F1000Research* 3 (Apr. 2015), p. 223. ISSN: 2046-1402. DOI: 10.12688/f1000research.5292.2. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4329601/ (visited on 07/01/2023).

[19] Roy Featherstone. *Rigid Body Dynamics Algorithms*. en. Boston, MA: Springer US, 2008. ISBN: 978-0-387-74314-1. DOI: 10.1007/978-1-4899-7560-7. URL: http://link.springer.com/10.1007/978-1-4899-7560-7 (visited on 12/08/2022).

[20] C. Daniel Freeman et al. *Brax – A Differentiable Physics Engine for Large Scale Rigid Body Simulation*. arXiv:2106.13281 [cs]. June 2021. DOI: 10.48550/arXiv.2106.13281. URL: http://arxiv.org/abs/2106.13281 (visited on 11/30/2022).

[21] Shashi Gowda et al. “High-performance symbolic-numerics via multiple dispatch”. In: *arXiv preprint arXiv:2105.03949* (2021).

[22] Ronald Hess, Jason K. Moore, and Mont Hubbard. “Modeling the Manually Controlled Bicycle”. In: *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 42.3 (May 2012). Conference Name: IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans, pp. 545–557. ISSN: 1558-2426. DOI: 10.1109/TSMCA.2011.2164244.

[23] Taylor A. Howell et al. *Dojo: A Differentiable Physics Engine for Robotics*. arXiv:2203.00806 [cs]. June 2022. DOI: 10.48550/arXiv.2203.00806. URL: http://arxiv.org/abs/2203.00806 (visited on 10/25/2022).

[24] Wolfram Research Inc. *Mathematica*. Champaign, IL, 2023. URL: https://www.wolfram.com/mathematica.

[25] Claudio Jolowicz. *Hypermodern Python*. Jan. 2020. URL: https://cjolowicz.github.io/posts/hypermodern-python-01-setup.

[26] Thomas R. Kane and David A. Levinson. *Dynamics, Theory and Applications*. en. Accepted: 2005-03-14T16:09:48Z. McGraw Hill, 1985. ISBN: 978-0-07-037846-9. URL: https://ecommons.cornell.edu/handle/1813/638 (visited on 01/26/2023).

[27] Hiroyuki Kawai et al. “Closed-Loop Position and Cadence Tracking Control for FES-Cycling Exploiting Pedal Force Direction With Antagonistic Biarticular Muscles”. In: *IEEE Transactions on Control Systems Technology* 27.2 (Mar. 2019). Conference Name: IEEE Transactions on Control Systems Technology, pp. 730–742. ISSN: 1558-0865. DOI: 10.1109/TCST.2017.2771727.

[28] Hiroyuki Kawai et al. “Tracking control for FES-cycling based on force direction efficiency with antagonistic bi-articular muscles”. In: *2014 American Control Conference*. ISSN: 2378-5861. June 2014, pp. 5484–5489. DOI: 10.1109/ACC.2014.6859197.

[29] Matthew Kelly. “An Introduction to Trajectory Optimization: How to Do Your Own Direct Collocation”. In: *SIAM Review* 59.4 (Jan. 2017). Publisher: Society for Industrial and Applied Mathematics, pp. 849–904. ISSN: 0036-1445. DOI: 10.1137/16M1062569. URL: https://epubs.siam.org/doi/abs/10.1137/16M1062569 (visited on 07/04/2023).

[30] J D G Kooijman. “Experimental Validation of a Model for the Motion of an Uncontrolled Bicycle. Master’s thesis”. en. PhD thesis. Delft University of Technology, 2006.

[31] J. D. G. Kooijman, A. L. Schwab, and J. P. Meijaard. “Experimental validation of a model of an uncontrolled bicycle”. en. In: *Multibody System Dynamics* 19.1 (Feb. 2008), pp. 115–132. ISSN: 1573-272X. DOI: 10.1007/s11044-007-9050-x. URL: https://doi.org/10.1007/s11044-007-9050-x (visited on 11/07/2022).

[32] Adrian K. M. Lai et al. “Does a two-element muscle model offer advantages when estimating ankle plantar flexor forces during human cycling?” en. In: *Journal of Biomechanics* 68 (Feb. 2018), pp. 6–13. ISSN: 0021-9290. DOI: 10.1016/j.jbiomech.2017.12.018. URL: https://www.sciencedirect.com/science/article/pii/S002192901730725X (visited on 10/24/2022).

[33] Adrian K. M. Lai et al. “Lower-limb muscle function is influenced by changing mechanical demands in cycling”. In: *Journal of Experimental Biology* 224.3 (Feb. 2021), jeb228221. ISSN: 0022-0949. DOI: 10.1242/jeb.228221. URL: https://doi.org/10.1242/jeb.228221 (visited on 10/24/2022).

[34] Martin Lesser. “A Geometrical Interpretation of Kane’s Equations”. In: *Proceedings: Mathematical and Physical Sciences* 436.1896 (1992). Publisher: The Royal Society, pp. 69–87. ISSN: 0962-8444. URL: https://www.jstor.org/stable/52020 (visited on 02/25/2023).

[35] David Levinson. “The use of AUTOLEV for kinematical analyses of robots”. In: *Guidance, Navigation and Control Conference*. _eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.1990-3386. American Institute of Aeronautics and Astronautics, 1990. DOI: 10.2514/6.1990-3386. URL: https://arc.aiaa.org/doi/abs/10.2514/6.1990-3386 (visited on 01/26/2023).

[36] D.J.N. Limebeer and R.S. Sharp. “Bicycles, motorcycles, and models”. In: *IEEE Control Systems Magazine* 26.5 (Oct. 2006). Conference Name: IEEE Control Systems Magazine, pp. 34–61. ISSN: 1941-000X. DOI: 10.1109/MCS.2006.1700044.

[37] Yi-Chung Lin and Marcus G. Pandy. “Three-dimensional data-tracking dynamic optimization simulations of human locomotion generated by direct collocation”. en. In: *Journal of Biomechanics* 59 (July 2017), pp. 1–8. ISSN: 0021-9290. DOI: 10.1016/j.jbiomech.2017.04.038. URL: https://www.sciencedirect.com/science/article/pii/S0021929017302567 (visited on 07/05/2023).

[38] Yung-Sheng Liu, Tswn-Syau Tsay, and Tsai-Chu Wang. “Muscles force and joints load simulation of bicycle riding using multibody models”. en. In: *Procedia Engineering*. 5th Asia-Pacific Congress on Sports Technology (APCST) 13 (Jan. 2011), pp. 81–87. ISSN: 1877-7058. DOI: 10.1016/j.proeng.2011.05.055. URL: https://www.sciencedirect.com/science/article/pii/S1877705811009696 (visited on 10/24/2022).

[39] *Maple*. 1996. URL: https://www.maplesoft.com/.

[40] J. P. Meijaard and A. L. Schwab. “Linearized equations for an extended bicycle model”. en. In: *III European Conference on Computational Mechanics*. Ed. by C. A. Motasoares et al. Dordrecht: Springer Netherlands, 2006, pp. 772–772. ISBN: 978-1-4020-5370-2. DOI: 10.1007/1-4020-5370-3_772.

[41] J. P. Meijaard et al. “History of thoughts about bicycle self-stability”. en. In: (2011). Accepted: 2011-03-30T17:15:46Z. URL: https://ecommons.cornell.edu/handle/1813/22497 (visited on 07/08/2023).

[42] J.p Meijaard et al. “Linearized dynamics equations for the balance and steer of a bicycle: a benchmark and review”. In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 463.2084 (Aug. 2007). Publisher: Royal Society, pp. 1955–1982. DOI: 10.1098/rspa.2007.1857. URL: https://royalsocietypublishing-org.tudelft.idm.oclc.org/doi/10.1098/rspa.2007.1857 (visited on 10/10/2022).

[43] Mathieu Ménard, Patrick Lacouture, and Mathieu Domalain. “ILIOTIBIAL BAND SYNDROME IN CYCLING: A COMBINED EXPERIMENTAL-SIMULATION APPROACH FOR ASSESSING THE EFFECT OF SADDLE SETBACK”. In: *International Journal of Sports Physical Therapy* 15.6 (Dec. 2020), pp. 958–966. ISSN: 2159-2896. DOI: 10.26603/ijspt20200958. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7727429/ (visited on 11/14/2022).

[44] Aaron Meurer et al. “SymPy: symbolic computing in Python”. In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. URL: https://doi.org/10.7717/peerj-cs.103.

[45] David Millard et al. *Automatic Differentiation and Continuous Sensitivity Analysis of Rigid Body Dynamics*. arXiv:2001.08539 [cs, eess]. Jan. 2020. DOI: 10.48550/arXiv.2001.08539. URL: http://arxiv.org/abs/2001.08539 (visited on 12/05/2022).

[46] Paul C. Mitiguy and Thomas R. Kane. *Motion Variables Leading to Efficient Equations of Motion - Paul C. Mitiguy, Thomas R. Kane, 1996*. 1996. URL: https://journals.sagepub.com/doi/abs/10.1177/027836499601500507?casa_token=e1YlAls8he8AAAAA:SvdKrBIZ_UnpQOsgNqTCLxKAYkYXeDJHxEPPlydCIKO_G_6mcjMC4QTz79dZwV9itkmyGUIK2k8 (visited on 09/16/2022).

[47] Jason K. Moore. *BicycleParameters: A Python library for bicycle parameter estimation and analysis.* 2011. URL: http://pypi.python.org/pypi/BicycleParameters (visited on 07/01/2023).

[48] Jason K. Moore. *Carvallo-Whipple Bicycle Model*. URL: https://pydy.readthedocs.io/en/stable/examples/carvallo-whipple.html.

[49] Jason K. Moore. “Human Control of a Bicycle”. PhD thesis. University of California, Davis, Aug. 2012. URL: http://moorepants.github.io/dissertation/index.html (visited on 10/10/2022).

[50] Jason K. Moore. *Learn Multibody Dynamics*. 2022. URL: https://moorepants.github.io/learn-multibody-dynamics/index.html (visited on 06/13/2023).

[51] Jason K. Moore et al. “A Method for Estimating Physical Properties of a Combined Bicycle and Rider”. In: vol. 4. Jan. 2009. DOI: 10.1115/DETC2009-86947.

[52] Masahiro Nagumo et al. “Iterative Learning Control for FES-Cycling With Antagonistic Biarticular Muscles”. In: *IECON 2020 The 46th Annual Conference of the IEEE Industrial Electronics Society*. ISSN: 2577-1647. Oct. 2020, pp. 279–284. DOI: 10.1109/IECON43393.2020.9254655.

[53] David Pagnon, Mathieu Domalain, and Lionel Reveret. “Pose2Sim: An End-to-End Workflow for 3D Markerless Sports Kinematics—Part 1: Robustness”. en. In: *Sensors* 21.19 (Jan. 2021). Number: 19 Publisher: Multidisciplinary Digital Publishing Institute, p. 6530. ISSN: 1424-8220. DOI: 10.3390/s21196530. URL: https://www.mdpi.com/1424-8220/21/19/6530 (visited on 11/14/2022).

[54] Sangsoo Park, Graham E. Caldwell, and Brian R. Umberger. “A direct collocation framework for optimal control simulation of pedaling using OpenSim”. en. In: *PLOS ONE* 17.2 (Feb. 2022). Publisher: Public Library of Science, e0264346. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0264346. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0264346 (visited on 11/14/2022).

[55] Michael A. Patterson and Anil V. Rao. “GPOPS-II: A MATLAB Software for Solving Multiple-Phase Optimal Control Problems Using hp-Adaptive Gaussian Quadrature Collocation Methods and Sparse Nonlinear Programming”. In: *ACM Transactions on Mathematical Software* 41.1 (Oct. 2014), 1:1–1:37. ISSN: 0098-3500. DOI: 10.1145/2558904. URL: https://dl.acm.org/doi/10.1145/2558904 (visited on 07/05/2023).

[56] Luke Peterson. “Bicycle dynamics: modelling and experimental validation”. PhD thesis. University of California Davis, 2013. URL: https://github.com/hazelnusse/dissertation (visited on 06/05/2023).

[57] Manfred Plöchl et al. “On the wobble mode of a bicycle”. In: *Vehicle System Dynamics* 50.3 (Mar. 2012). Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/00423114.2011.594164, pp. 415–429. ISSN: 0042-3114. DOI: 10.1080/00423114.2011.594164. URL: https://doi.org/10.1080/00423114.2011.594164 (visited on 11/17/2022).

[58] *Poetry - Python dependency management and packaging made easy*. 2023. URL: https://python-poetry.org.

[59] Sina Porsa, Yi-Chung Lin, and Marcus G. Pandy. “Direct Methods for Predicting Movement Biomechanics Based Upon Optimal Control Theory with Implementation in OpenSim”. en. In: *Annals of Biomedical Engineering* 44.8 (Aug. 2016), pp. 2542–2557. ISSN: 1573-9686. DOI: 10.1007/s10439-015-1538-6. URL: https://doi.org/10.1007/s10439-015-1538-6 (visited on 07/05/2023).

[60] Yi-Ling Qiao et al. “Efficient Differentiable Simulation of Articulated Bodies”. en. In: *Proceedings of the 38th International Conference on Machine Learning*. ISSN: 2640-3498. PMLR, July 2021, pp. 8661–8671. URL: https://proceedings.mlr.press/v139/qiao21a.html (visited on 12/05/2022).

[61] Jeffery W. Rankin and Richard R. Neptune. “A theoretical analysis of an optimal chainring shape to maximize crank power during isokinetic pedaling”. en. In: *Journal of Biomechanics* 41.7 (Jan. 2008), pp. 1494–1502. ISSN: 0021-9290. DOI: 10.1016/j.jbiomech.2008.02.015. URL: https://www.sciencedirect.com/science/article/pii/S0021929008000705 (visited on 06/29/2023).

[62] Anil Rao. “A Survey of Numerical Methods for Optimal Control”. In: *Advances in the Astronautical Sciences* 135 (Jan. 2010).

[63] Amy Robinson. “Principles of Neuromusculoskeletal Coordination in Human Cycling”. en. doctoral. Manchester Metropolitan University in collaboration with Simon Fraser University, 2020. URL: https://e-space.mmu.ac.uk/627317/ (visited on 10/24/2022).

[64] Dan Rosenthal and Michael Sherman. “High Performance Multibody Simulations via Symbolic Equation Manipulation and Kane’s Method”. In: *Journal of the Astronautical Sciences* 34 (July 1986), pp. 223–239.

[65] *Ruff - An extremely fast Python linter, written in Rust*. 2023. URL: https://beta.ruff.rs/docs.

[66] Hans H. C. M. Savelberg, Ingrid G. L. Van de Port, and Paul J. B. Willems. “Body Configuration in Cycling Affects Muscle Recruitment and Movement Pattern”. en. In: *Journal of Applied Biomechanics* 19.4 (Nov. 2003). Publisher: Human Kinetics, Inc. Section: Journal of Applied Biomechanics, pp. 310–324. ISSN: 1065-8483, 1543-2688. DOI: 10.1123/jab.19.4.310. URL: https://journals.humankinetics.com/view/journals/jab/19/4/article-p310.xml (visited on 10/24/2022).

[67] Johannes Schmidt. “Setting up Python Projects”. In: *Medium* (Sept. 2022). URL: https://towardsdatascience.com/setting-up-python-projects-part-i-408603868c08.

[68] A. Schwab, J. Kooijman, and J. Meijaard. “Some recent developments in bicycle dynamics and control”. In: Sept. 2008. URL: https://www.semanticscholar.org/paper/Some-recent-developments-in-bicycle-dynamics-and-Schwab-Kooijman/b41b85c2de774792bee89965ffebd08f60d9fad0 (visited on 06/30/2023).

[69] A. L. Schwab and J. P. Meijaard. “A review on bicycle dynamics and rider control”. In: *Vehicle System Dynamics* 51.7 (July 2013), pp. 1059–1090. ISSN: 0042-3114. DOI: 10.1080/00423114.2013.793365. URL: https://doi.org/10.1080/00423114.2013.793365 (visited on 10/13/2022).

[70] A. L. Schwab, J. P. Meijaard, and J. D.G. Kooijman. “Lateral dynamics of a bicycle with a passive rider model: stability and controllability”. In: *Vehicle System Dynamics* 50.8 (Aug. 2012). Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/00423114.2011.610898, pp. 1209–1224. ISSN: 0042-3114. DOI: 10.1080/00423114.2011.610898. URL: https://doi.org/10.1080/00423114.2011.610898 (visited on 06/27/2023).

[71] KA Seffen, GT Parks, and PJ Clarkson. “Observations on the controllability of motion of two-wheelers”. In: *Proceedings of the institution of mechanical engineers, part I: journal of systems and control engineering* 215.2 (2001), pp. 143–156.

[72] Ajay Seth et al. “Minimal formulation of joint motion for biomechanisms”. en. In: *Nonlinear Dynamics* 62.1 (Oct. 2010), pp. 291–303. ISSN: 1573-269X. DOI: 10.1007/s11071-010-9717-3. URL: https://doi.org/10.1007/s11071-010-9717-3 (visited on 01/27/2023).

[73] Amrit Sharma. “Stability Analysis of Bicycles & Motorcycles”. en. In: (2010), p. 220.

[74] R. S. Sharp. “The Dynamics of Single Track Vehicles”. In: *Vehicle System Dynamics* 5.1-2 (Aug. 1976). Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/00423117508968406, pp. 67–77. ISSN: 0042-3114. DOI: 10.1080/00423117508968406. URL: https://doi.org/10.1080/00423117508968406 (visited on 07/08/2023).

[75] Robin S. Sharp. “On the Stability and Control of the Bicycle”. In: *Applied Mechanics Reviews* 61.6 (Oct. 2008). ISSN: 0003-6900. DOI: 10.1115/1.2983014. URL: https://doi.org/10.1115/1.2983014 (visited on 02/20/2023).

[76] Michael A. Sherman, Ajay Seth, and Scott L. Delp. “Simbody: multibody dynamics for biomedical research”. en. In: *Procedia IUTAM*. IUTAM Symposium on Human Body Dynamics 2 (Jan. 2011), pp. 241–261. ISSN: 2210-9838. DOI: 10.1016/j.piutam.2011.04.023. URL: https://www.sciencedirect.com/science/article/pii/S2210983811000241 (visited on 06/12/2023).

[77] Jonathan Sinclair, Philip Stainton, and Benjamin Sant. “The effects of conventional and oval chainrings on patellofemoral loading during road cycling: an exploration using musculoskeletal simulation”. en. In: *Sport Sciences for Health* 14.1 (Apr. 2018), pp. 61–70. ISSN: 1825-1234. DOI: 10.1007/s11332-017-0401-6. URL: https://doi.org/10.1007/s11332-017-0401-6 (visited on 10/24/2022).

[78] Ana Carolina Cardoso de Sousa et al. “A Comparative Study on Control Strategies for FES Cycling Using a Detailed Musculoskeletal Model”. en. In: *IFAC-PapersOnLine*. Cyber-Physical & Human-Systems CPHS 2016 49.32 (Jan. 2016), pp. 204–209. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2016.12.215. URL: https://www.sciencedirect.com/science/article/pii/S2405896316328877 (visited on 10/24/2022).

[79] Timo Stienstra. *SymMePlot: A Visualization Tool for Symbolically Defined Mechanical Systems*. original-date: 2022-04-01T18:58:26Z. 2023. URL: https://github.com/TJStienstra/symmeplot (visited on 07/10/2023).

[80] Shoichiro Takehara, Musashi Murakami, and Kazunori Hase. “Biomechanical Evaluation of an Electric Power-Assisted Bicycle by a Musculoskeletal Model”. In: *Journal of System Design and Dynamics* 6.3 (2012), pp. 343–350. DOI: 10.1299/jsdd.6.343.

[81] Rachel Thompson. “Musculoskeletal Loads during Stationary Cycling and the Effects of Pedal Modifications for Knee Osteoarthritis”. In: *Doctoral Dissertations* (May 2017). URL: https://trace.tennessee.edu/utk_graddiss/4503.

[82] Emanuel Todorov, Tom Erez, and Yuval Tassa. “MuJoCo: A physics engine for model-based control”. In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. ISSN: 2153-0866. Oct. 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[83] Daniella Tola and Peter Corke. *Understanding URDF: A Survey Based on User Experience*. arXiv:2302.13442 [cs]. Mar. 2023. DOI: 10.48550/arXiv.2302.13442. URL: http://arxiv.org/abs/2302.13442 (visited on 06/16/2023).

[84] Brian R. Umberger, Karin G. M. Gerritsen, and Philip E. Martin. “Muscle fiber type effects on energetically optimal cadences in cycling”. en. In: *Journal of Biomechanics* 39.8 (Jan. 2006), pp. 1472–1479. ISSN: 0021-9290. DOI: 10.1016/j.jbiomech.2005.03.025. URL: https://www.sciencedirect.com/science/article/pii/S0021929005001661 (visited on 10/24/2022).

[85] Brian R. Umberger and Ross H. Miller. “Optimal Control Modeling of Human Movement”. en. In: *Handbook of Human Motion*. Ed. by Bertram Müller et al. Cham: Springer International Publishing, 2017, pp. 1–22. ISBN: 978-3-319-30808-1. DOI: 10.1007/978-3-319-30808-1_177-1. URL: https://doi.org/10.1007/978-3-319-30808-1_177-1 (visited on 07/05/2023).

[86] A. J. “Knoek Van Soest and L. J. Richard Casius. “Which factors determine the optimal pedaling rate in sprint cycling?” en-US. In: *Medicine & Science in Sports & Exercise* 32.11 (Nov. 2000), p. 1927. ISSN: 0195-9131. URL: https://journals.lww.com/acsm-msse/Fulltext/2000/11000/Which_factors_determine_the_optimal_pedaling_rate.17.aspx (visited on 06/29/2023).

[87] O Von Stryk. *DIRCOL: A Direct Collocation Method for the Numerical Solution of Optimal Control Problems*. 1999. URL: https://www.informatik.tu-darmstadt.de/sim/forschung_sim/software_sim/dircol_sim/index.en.jsp (visited on 07/05/2023).

[88] Pengcheng Wang, Jingang Yi, and Tao Liu. “Stability and Control of a Rider–Bicycle System: Analysis and Experiments”. In: *IEEE Transactions on Automation Science and Engineering* 17.1 (Jan. 2020). Conference Name: IEEE Transactions on Automation Science and Engineering, pp. 348–360. ISSN: 1558-3783. DOI: 10.1109/TASE.2019.2922068.

[89] Keenon Werling et al. “Fast and Feature-Complete Differentiable Physics for Articulated Rigid Bodies with Contact”. en. In: (June 2021), p. 15.

[90] Francis JW Whipple. “The stability of the motion of a bicycle”. In: *Quarterly Journal of Pure and Applied Mathematics*. 120th ser. 30 (1899), pp. 312–348. URL: http://ruina.tam.cornell.edu/research/topics/bicycle_mechanics/Whipple.pdf (visited on 10/26/2022).

[91] Francis JW Whipple. “The stability of the motion of a bicycle”. In: *Quarterly Journal of Pure and Applied Mathematics* 30.120 (1899), pp. 312–348.

[92] Ross D. Wilkinson, Andrew G. Cresswell, and Glen A. Lichtwark. “Riders Use Their Body Mass to Amplify Crank Power during Nonseated Ergometer Cycling”. English. In: *Medicine and Science in Sports and Exercise* 52.12 (Dec. 2020). Place: Philadelphia Publisher: Lippincott Williams & Wilkins WOS:000588790200014, pp. 2599–2607. ISSN: 0195-9131. DOI: 10.1249/MSS.0000000000002408. URL: http://journals.lww.com/acsm-msse/Fulltext/2020/12000/Riders_Use_Their_Body_Mass_to_Amplify_Crank_Power.14.aspx (visited on 11/14/2022).

[93] Gary Tad Yamaguchi. *Dynamic Modeling of Musculoskeletal Motion*. en. Boston, MA: Springer US, 2001. ISBN: 978-0-387-28704-1 978-0-387-28750-8. DOI: 10.1007/978-0-387-28750-8. URL: http://link.springer.com/10.1007/978-0-387-28750-8 (visited on 06/13/2023).

[94] M. R. Yeadon. “The simulation of aerial movement—I. The determination of orientation angles from film data”. en. In: *Journal of Biomechanics* 23.1 (Jan. 1990), pp. 59–66. ISSN: 0021-9290. DOI: 10.1016/0021-9290(90)90369-E. URL: https://www.sciencedirect.com/science/article/pii/002192909090369E (visited on 07/01/2023).

[95] Yaofeng Desmond Zhong, Jiequn Han, and Georgia Olympia Brikis. *Differentiable Physics Simulations with Contacts: Do They Have Correct Gradients w.r.t. Position, Velocity and Control?* arXiv:2207.05060 [cs]. July 2022. DOI: 10.48550/arXiv.2207.05060. URL: http://arxiv.org/abs/2207.05060 (visited on 12/05/2022).

[96] Lu Zongxing et al. “The Effect of Crank Length Changes from Cycling Rehabilitation on Muscle Behaviors”. en. In: *Applied Bionics and Biomechanics* 2021 (Apr. 2021). Publisher: Hindawi, e8873426. ISSN: 1176-2322. DOI: 10.1155/2021/8873426. URL: https://www.hindawi.com/journals/abb/2021/8873426/ (visited on 10/24/2022).

A

Kane's Method

There were several reasons for Kane's method to be developed. The major one is that classical methods such as Newton-Euler, Lagrange, and Hamilton are labor intensive resulting in computationally expensive EOMs for complex systems. Contrary Kane's method has been designed to be systematic, while resulting in simpler EOMs. With the additional advantage of it being able to directly deal with nonholonomic systems without having to use Lagrange multipliers [26]. This appendix will briefly discuss the implementation of Kane's method within *SymPy*, which has drawn strong inspiration from the implementation in *AUTOLEV* [35].

The EOMs formed by Kane's method are second-order Ordinary Differential Equations (ODEs) in the following form.

$$\boldsymbol{f}_k(\boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{u}) = \boldsymbol{M}_k \dot{\boldsymbol{q}} - \boldsymbol{g}_k = \boldsymbol{0} \tag{A.1}$$

$$\boldsymbol{f}_d(\boldsymbol{q}, \boldsymbol{u}, \dot{\boldsymbol{u}}) = \boldsymbol{M}_d \dot{\boldsymbol{u}} - \boldsymbol{g}_d = \boldsymbol{0} \tag{A.2}$$

A.0.1. Kinematic Differential Equations

The first set of equations (equation (A.1)) are the kinematic differential equations. While methods like Lagrange simply use the identity matrix as the kinematic mass matrix $\boldsymbol{M}_k$ and the generalized coordinates $\boldsymbol{q}$ as the kinematic forcing vector $\boldsymbol{g}_k$, Kane's method also makes use of generalized speeds. Generalized speeds ($\boldsymbol{u}$) are used to specify the velocity of the system, similarly to how the generalized coordinates are used to specify its configuration. These generalized speeds must be chosen such that they are linear functions of the time derivatives of the generalized coordinates ($\dot{\boldsymbol{q}}$), which can be uniquely solved for the $\dot{\boldsymbol{q}}$. Therefore, they can be written in the following form:

$$\boldsymbol{f}_k(\boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{u}) = \boldsymbol{0} = \boldsymbol{J}_{\boldsymbol{f}_k, \boldsymbol{u}} \boldsymbol{u} + \boldsymbol{J}_{\boldsymbol{f}_k, \dot{\boldsymbol{q}}} \dot{\boldsymbol{q}} + \boldsymbol{f}_k\left(\boldsymbol{q}, \boldsymbol{0}, \boldsymbol{0}\right)) \tag{A.3}$$

$$\underbrace{\boldsymbol{J}_{\boldsymbol{f}_k, \dot{\boldsymbol{q}}}}_{\boldsymbol{M}_k} \dot{\boldsymbol{q}} = \underbrace{-\left(\boldsymbol{J}_{\boldsymbol{f}_k, \boldsymbol{u}} \boldsymbol{u} + \boldsymbol{f}_k\left(\boldsymbol{q}, \boldsymbol{0}, \boldsymbol{0}\right)\right))}_{\boldsymbol{g}_d} \tag{A.4}$$

where $\boldsymbol{J}_{\boldsymbol{f}, \boldsymbol{x}}$ is defined as the Jacobian of $\boldsymbol{f}$ with respect to $\boldsymbol{x}$.

The advantage of using generalized speeds is that they can simplify the EOMs. Mitiguy and Kane [46] provide several guidelines on choosing the generalized speeds and show an example where it reduces the number of operations of the EOMs by a factor three.

A.0.2. Dynamic Differential Equations

The second set of equations (equation (A.2)) are the dynamic differential equations. These equations equal the sum of the generalized inertia forces ($\boldsymbol{F}_r^*$) with the generalized active forces ($\boldsymbol{F}_r$), shown in equation (A.5). The generalized inertia forces are computed by dotting the derivatives of the momenta of the bodies with the corresponding partial velocities. Similarly, the generalized active forces are computed by dotting the external loads with the corresponding partial velocities. Both dot products can be seen as projections unto the plane representing all possible configurations, called the configuration

tangent space by Lesser [34]. This also closely relates to the well-known D'Alembert's principle. A more detailed explanation of how to compute the generalized forces using *SymPy* is provided by Moore [50].

$$\boldsymbol{F}_r^* + \boldsymbol{F}_r = \boldsymbol{0} \tag{A.5}$$

$$\boldsymbol{J}_{\boldsymbol{F}_r^*, \dot{\boldsymbol{u}}} \dot{\boldsymbol{u}} = -\boldsymbol{F}_r^*\big|_{\dot{\boldsymbol{u}}=\boldsymbol{0}} - \boldsymbol{F}_r \tag{A.6}$$

A.0.3. Constraints

Instead of using holonomic and nonholonomic constraints, Kane's method uses configuration and velocity constraints. Configuration constraints are all constraints in the configuration space making them equivalent to the holonomic constraints. The velocity constraints are all the constraints in the velocity space, thus the velocity constraints comprise the time derivative of the holonomic constraints and the nonholonomic constraints. Within the implementation of Kane's method in `sympy.physics.mechanics` the velocity constraints ($\boldsymbol{f}_{vc}$) are solved creating a matrix A_{sr}[1].

$$\boldsymbol{f}_{vc}(\boldsymbol{q}, \boldsymbol{u}) = \boldsymbol{0} = \boldsymbol{J}_{\boldsymbol{f}_{vc}, \boldsymbol{u}_s} \boldsymbol{u}_s + \boldsymbol{J}_{\boldsymbol{f}_{vc}, \boldsymbol{u}_r} \boldsymbol{u}_r + \boldsymbol{f}_{vc}(\boldsymbol{q}, \boldsymbol{0}) \tag{A.7}$$

$$\boldsymbol{u}_s = \underbrace{-\boldsymbol{J}_{\boldsymbol{f}_{vc}, \boldsymbol{u}_s}^{-1} \boldsymbol{J}_{\boldsymbol{f}_{vc}, \boldsymbol{u}_r}}_{A_{sr}} \boldsymbol{u}_r - \boldsymbol{J}_{\boldsymbol{f}_{vc}, \boldsymbol{u}_s}^{-1} \boldsymbol{f}_{vc}(\boldsymbol{q}, \boldsymbol{0}) \tag{A.8}$$

This matrix A_{sr} is used to map the dependent generalized forces to the independent generalized forces, shown in equations (A.9) and (A.10). Notably $-\boldsymbol{J}_{\boldsymbol{f}_{vc}, \boldsymbol{u}_s}^{-1} \boldsymbol{f}_{vc}(\boldsymbol{q}, \boldsymbol{0})$ is eliminated during the computation of the partial velocities.

$$\boldsymbol{F}_r = \begin{bmatrix} F_{r,1} \\ \vdots \\ F_{r,p} \end{bmatrix} + A_{sr} \begin{bmatrix} F_{r,p+1} \\ \vdots \\ F_{r,n} \end{bmatrix} \tag{A.9}$$

$$\boldsymbol{F}_r^* = \begin{bmatrix} F_{r,1}^* \\ \vdots \\ F_{r,p}^* \end{bmatrix} + A_{sr} \begin{bmatrix} F_{r,p+1}^* \\ \vdots \\ F_{r,n}^* \end{bmatrix} \tag{A.10}$$

The velocity constraints are further incorporated into the EOMs by adding the time derivative of the velocity constraints, known as the acceleration constraints, as additional EOMs. See the equations below.

$$\begin{bmatrix} \boldsymbol{F}_r + \boldsymbol{F}_r^* \\ \dot{\boldsymbol{f}}_{vc}(\boldsymbol{q}, \boldsymbol{u}, \dot{\boldsymbol{u}}) \end{bmatrix} = \boldsymbol{0} \tag{A.11}$$

$$\underbrace{\begin{bmatrix} \boldsymbol{J}_{\boldsymbol{F}_r^*, \dot{\boldsymbol{u}}} \\ \boldsymbol{J}_{\dot{\boldsymbol{f}}_{vc}, \dot{\boldsymbol{u}}} \end{bmatrix}}_{\boldsymbol{M}_d} \underbrace{\begin{bmatrix} \dot{\boldsymbol{u}}_s \\ \dot{\boldsymbol{u}}_r \end{bmatrix}}_{\dot{\boldsymbol{u}}} = \underbrace{\begin{bmatrix} -\boldsymbol{F}_r^*\big|_{\dot{\boldsymbol{u}}=\boldsymbol{0}} - \boldsymbol{F}_r \\ -\dot{\boldsymbol{f}}_{vc}\big|_{\dot{\boldsymbol{u}}=\boldsymbol{0}} \end{bmatrix}}_{\boldsymbol{g}_d} \tag{A.12}$$

[1] In the implementation of `KanesMethod` the inverse is actually computed and called `_Ars`.

B

Auto-Generation of Properties

The metaclasses in combination with the base classes, shown in figure 3.4, reduce the boilerplate when writing a new model in *BRiM*. One of the methods to reduce boilerplate and possible errors by modelers is the auto-generation of properties. A metaclass can automatically specify a required property upon class definition. One can specify these required properties by setting class attributes like `required_model` and `required_connections`. In the code below, a modeler specifies that `MyModel` only requires a ground as submodel.

```
122 from brim.bicycle import GroundBase
123 from brim.core import ModelBase, ModelRequirement
124
125
126 class MyModel(ModelBase):
127     required_models: tuple[ModelRequirement, ...] = (
128         ModelRequirement("ground", GroundBase, "Ground model."),
129     )
```

Listing B.1: Code defining a simple model, which requires a ground as submodel.

Upon running this code, the ground property is automatically defined as shown in the code below.

```
130 class MyModel(ModelBase):
131     required_models: tuple[ModelRequirement, ...] = (
132         ModelRequirement("ground", GroundBase, "Ground model."),
133     )
134
135     def __init__(self, name: str):
136         super().__init__(name)
137         # The ModelBase.__init__ sets the protected attributes to None.
138         self._ground = None
139
140     @property
141     def ground(self) -> GroundBase:
142         """Submodel of the ground."""
143         return self._ground
144
145     @ground.setter
146     def ground(self, model: GroundBase) -> None:
147         if not (model is None or isinstance(model, GroundBase)):
148             raise TypeError(
149                 f"Ground should be an instance of an subclass of GroundBase, but "
150                 f"{model!r} is an instance of {type(model)}.")
151         self._ground = model
```

Listing B.2: Simplified representation of the resulting code, when specifying a required model.

C

Rolling Disc Implementation

This appendix provides a simplified implementation of the rolling disc model described in section 2.2. The implementation is simplified for explanation purposes. A more advanced implementation is available in *BRiM*. While section 3.5 gives an overview of each component to create the rolling disc model, this appendix gives a more in-depth explanation including code. The required imports to run the code in this appendix is provided in listing C.1.

Appendix C.1 starts with an explanation of the implementation of both the abstract base class of the ground model as well as the implementation of the flat ground. Appendix C.2 continues with the implementation of the knife-edge wheel, while inheriting the abstract base class from *BRiM*. The implementation of the tyre connection is explained in appendix C.3. Followed by appendix C.4 with the overarching rolling disc model class. After which appendix C.5 introduces a load group to control the rolling disc. Finally, appendix C.6 shows how the code created in this appendix can be utilized to form the EOMs of the rolling disc.

```
1  from abc import abstractmethod
2  from typing import Any
3
4  from sympy import Expr, Matrix, Symbol, symbols
5  from sympy.physics.mechanics import (
6      Point, ReferenceFrame, RigidBody, System, Torque, Vector, cross, dot,
7      dynamicsymbols, inertia)
8
9  from brim.bicycle import TyreBase, WheelBase
10 from brim.core import ConnectionRequirement, LoadGroupBase, ModelBase, ModelRequirement
```

Listing C.1: Code with the required imports to run the code in appendix C.

C.1. Ground Model Implementation

The flat ground inherits from the abstract class `GroundBase`, which makes sure that each ground shares a total of three properties and three methods. The three attributes, namely a body, frame and origin, are defined by `GroundBase`, such that subclasses only have to implement three methods. One returns the normal vector, another the tangent vectors of the plane and a last one sets the position of a point in plane w.r.t. the origin[1].

When implementing the define steps in a model, connection or load group, then a leading underscore is added: `_define_<step>`. The reason is that these only run the define step for the component itself without traversing the submodels and the load groups. `BrimBase` contains the implementation of the define methods including traversal, which ought to be called by the user. These methods follow the format `define_<step>`.

[1] As explained in section 3.5.2 the normal and tangent vectors method should be position dependent. However, this example simplifies the problem a little by assuming them as position independent.

```python
class GroundBase(ModelBase):
    """Base class for the ground."""

    def _define_objects(self) -> None:
        """Define the objects of the ground."""
        # When overwriting a method from a parent class, it is good practise to call
        # the parent method first. In this case, the _define_objects method of the
        # ModelBase class is called.
        super()._define_objects()
        # Create a rigid body to represent the ground.
        self._body = RigidBody(self.name)
        self._body.masscenter = Point(self._add_prefix("origin"))
        # Create the system object of the ground.
        self._system = System.from_newtonian(self.body)

    def _define_kinematics(self) -> None:
        """Define the kinematics of the ground."""
        super()._define_kinematics()
        # Fixate the origin in the ground frame.
        self.origin.set_vel(self.frame, 0)

    @property
    def body(self) -> RigidBody:
        """The body representing the ground."""
        return self._body

    @property
    def frame(self) -> ReferenceFrame:
        """Frame fixed to the ground."""
        return self.body.frame

    @property
    def origin(self) -> Point:
        """Origin of the ground."""
        return self.body.masscenter

    # The abstractmethod decorators make sure that subclasses have to implement these
    # methods.
    @property
    @abstractmethod
    def normal(self) -> Vector:
        """Normal vector of the ground."""

    @property
    @abstractmethod
    def tangent_vectors(self) -> tuple[Vector, Vector]:
        """Tangent vectors of the ground plane."""

    @abstractmethod
    def set_pos_point(self, point: Point, position: tuple[Expr, Expr]) -> None:
        """Locate a point on the ground."""
```

Listing C.2: Code defining a simplified version of the `GroundBase` model class.

The flat ground implements the abstract methods accordingly. The normal vector is the unit vector in the negative Z-direction. The X- and Y-unit vectors are tangent vectors, which are also used to set the position of a point in the XY plane.

```python
class FlatGround(GroundBase):
    """Flat ground."""

    @property
    def normal(self) -> Vector:
        """Normal vector of the ground."""
        return -self.frame.z

    @property
    def tangent_vectors(self) -> tuple[Vector, Vector]:
        """Tangent vectors of the ground plane."""
        return self.frame.x, self.frame.y

    def set_pos_point(self, point: Point, position: tuple[Expr, Expr]) -> None:
        """Set the location of a point on the ground."""
        point.set_pos(self.origin,
                      position[0] * self.frame.x + position[1] * self.frame.y)
```

Listing C.3: Code defining a simplified version of the `FlatGround` model class.

C.2. Wheel Model Implementation

Similar to the flat ground, the knife-edge wheel inherits from the abstract class `WheelBase`. This abstract class has already created a body (`body`) and a body-fixed frame (`frame`) to represent the wheel. Besides those, it prescribes the implementation of two properties using the `abstractmethod` decorator: the rotation axis (`rotation_axis`), and the wheel center (`center`). In the code below it can be seen that the knife-edge wheel implements these two properties by returning its center of mass as the center of the wheel and the Y-axis of the body-fixed frame as the rotation axis. In the implementation it can also be seen that the wheel uses a dictionary to keep track of its symbols[2]. This symbols dictionary is used to satisfy the wish that a user is able to change a symbol after the define object stage. Besides that the descriptions property, predefined in `BrimBase`, is used to store a description of each symbol. The reason for this feature is that an end-user can easily track down the meaning of each symbol using the `BrimBase.get_description` method[3]. The default definition of each symbol utilizes `BrimBase._add_prefix` to add the name of the instantiated model before each symbol. This way it is ensured that symbols will always be unique as long as every model is given a unique name by the end-user. One should denote that the knife-edge wheel only has to define the symbols to describe its shape, not the shape itself. The actual computation based on the shape to get the contact point position is done within the tyre connection.

[2]The dictionary is defined in `BrimBase` such that every component will be using a similar interface.

[3]The `get_description` uses a breadth-first search through the models, connections, and loads groups.

```python
class KnifeEdgeWheel(WheelBase):
    """Knife-edge wheel."""

    @property
    def descriptions(self) -> dict[Any, str]:
        """Descriptions of the attributes of the wheel."""
        return {
            **super().descriptions,
            self.symbols["r"]: """Radius of the wheel.""",
        }

    def _define_objects(self) -> None:
        """Define the objects of the wheel."""
        super()._define_objects()
        # Change inertia to account for the symmetry of the wheel.
        self.body.central_inertia = inertia(
            self.body.frame, *symbols(self._add_prefix("ixx iyy ixx")))
        self.symbols["r"] = Symbol(self._add_prefix("r"))

    @property
    def center(self) -> Point:
        """Point representing the center of the wheel."""
        return self.body.masscenter

    @property
    def rotation_axis(self) -> Vector:
        """Rotation axis of the wheel."""
        return self.body.y
```

Listing C.4: Code defining a simplified version of the `KnifeEdgeWheel` model class.

C.3. Tyre Connection Implementation

The abstract class for the connection between the ground and wheel is `TyreBase`. It creates for both the ground and wheel a property, which accepts any type of ground and wheel. It also creates a property for the contact point assuming a single contact point by default. Since the computation of the contact point is shared among all tyre models, the abstract class also implements a method to set the position of the wheel's center with respect to the contact point (`_set_pos_contact_point`). This method takes into account what type of wheel and ground has been specified. A last property that is specified by the abstract tyre class is one to specify whether the contact point is defined to be in the ground plane by the parent model, in this case the rolling disc model.

A tyre connection, which enforces pure-rolling using nonholonomic constraints, is implemented in the code below. It inherits from `TyreBase` and re-specifies the class attribute `required_models`. This attribute is processed by the metaclass `ConnectionMeta`. Based on this class attribute it auto-generates properties for the ground and wheel. These were already defined by `TyreBase`, but here they are overwritten to make sure that only an instance of `FlatGround` can be assigned to the ground attribute and similarly for the wheel. Refer to appendix B for more information on the auto-generation of these properties. The method `_define_kinematics` makes use of `_set_pos_contact_point` to define the position of the contact point[4]. In `_define_constraints` the nonholonomic constraints are defined, as explained in section 2.2.4, and a holonomic constraint is defined if the wheel can be off the ground.

[4] `_set_pos_contact_point` is redefined in `NonHolonomicTyre` to show its implementation and to make sure that you can run this code yourself.

```python
class NonHolonomicTyre(TyreBase):
    """Tyre model connection based on nonholonomic constraints."""

    required_models: tuple[ModelRequirement, ...] = (
        ModelRequirement("ground", FlatGround, "Ground model."),
        ModelRequirement("wheel", KnifeEdgeWheel, "Wheel model."),
    )

    def _set_pos_contact_point(self) -> None:
        """Compute the contact point of the wheel with the ground."""
        if isinstance(self.ground, FlatGround):
            if isinstance(self.wheel, KnifeEdgeWheel):
                self.wheel.center.set_pos(
                    self.contact_point, self.wheel.symbols["r"] * cross(
                        self.wheel.rotation_axis, cross(
                            self.ground.normal, self.wheel.rotation_axis)).normalize())
                return
        raise NotImplementedError(
            f"Computation of the contact point has not been implemented for the "
            f"combination of {type(self.ground)} and {type(self.wheel)}.")

    def _define_kinematics(self) -> None:
        """Define the kinematics of the tyre model."""
        super()._define_kinematics()
        self._set_pos_contact_point()

    def _define_constraints(self) -> None:
        """Define the constraints of the tyre model."""
        super()._define_constraints()
        # Get the normal and tangent vectors of the ground at the contact point.
        normal = self.ground.normal
        tangent_vectors = self.ground.tangent_vectors
        # Compute the velocity of wheel center using two different constructs.
        v1 = self.wheel.center.pos_from(self.ground.origin).dt(self.ground.frame)
        v2 = cross(self.wheel.frame.ang_vel_in(self.ground.frame),
                   self.wheel.center.pos_from(self.contact_point))
        # Compute and add the nonholonomic constraints.
        self.system.add_nonholonomic_constraints(
            dot(v1 - v2, tangent_vectors[0]), dot(v1 - v2, tangent_vectors[1]))
        # Add a holonomic constraint if the wheel is not defined to be on the ground.
        if not self.on_ground:
            self.system.add_holonomic_constraints(
                self.contact_point.pos_from(self.ground.origin).dot(normal))
```

Listing C.5: Code defining a simplified version of the `NonHolonomicTyre` connection class.

C.4. Rolling Disc Model Implementation

When implementing a new model like the rolling disc, which combines multiple of *BRiM*'s components, it is required to write a class. This process is similar to that of a component like a wheel, as the rolling disc model itself is also seen as a component. Listing C.6 implements the rolling disc based on the previously stated implementations. It starts by defining what models and connections it requires through class attributes, see appendix B for more information. This automatically creates properties for the ground, wheel, and tyre. Similar to the knife-edge wheel's `symbols` dictionary, see appendix C.2, the rolling disc uses a `q` and `u` mutable matrix to store its generalized coordinates and speeds.

The `RollingDisc` class has to implement all five define steps. In `_define_connections` it sets the ground and wheel attribute of the tyre connection. In `_define_objects` it first initializes a system with the same reference as the ground submodel. After which it lets the tyre connection define its objects and tells it that the contact point will be in the ground plane by definition[5]. Lastly, it also creates matrices to store its generalized coordinates and speeds. In `_define_kinematics` the wheel is oriented and positioned w.r.t. the ground, while adding the generalized coordinates, speeds and kinematic differential equations to the system instance. The last two steps only have to call the tyre to set its loads and constraints.

[5] An example where this is not the case is the front wheel of the Whipple bicycle.

```python
class RollingDisc(ModelBase):
    """Rolling disc model."""

    required_models: tuple[ModelRequirement, ...] = (
        ModelRequirement("ground", GroundBase, "Ground model."),
        ModelRequirement("wheel", WheelBase, "Wheel model."),
    )
    required_connections: tuple[ConnectionRequirement, ...] = (
        ConnectionRequirement("tyre", TyreBase, "Tyre model."),
    )

    @property
    def descriptions(self) -> dict[Any, str]:
        """Dictionary of descriptions of the rolling disc's attributes."""
        desc = {
            **super().descriptions,
            self.q[0]: "Perpendicular distance along ground.x to the contact point.",
            self.q[1]: "Perpendicular distance along ground.y to the contact point.",
            self.q[2]: "Yaw angle of the disc.",
            self.q[3]: "Roll angle of the disc.",
            self.q[4]: "Pitch angle of the disc.",
        }
        desc.update({ui: f"Generalized speed of the {desc[qi].lower()}"
                     for qi, ui in zip(self.q, self.u)})
        return desc

    def _define_connections(self) -> None:
        """Define the connections between the submodels."""
        super()._define_connections()
        self.tyre.ground = self.ground
        self.tyre.wheel = self.wheel

    def _define_objects(self) -> None:
        """Define the objects of the rolling disc."""
        super()._define_objects()
        # Define the system instance with the same reference as the ground.
        self._system = System(self.ground.system.origin, self.ground.frame)
        # Setup the tyre model.
        self.tyre.define_objects()
        self.tyre.on_ground = True
        # Define the generalized coordinates and speeds.
        self.q = Matrix([dynamicsymbols(self._add_prefix("q1:6"))])
        self.u = Matrix([dynamicsymbols(self._add_prefix("u1:6"))])

    def _define_kinematics(self) -> None:
        """Define the kinematics of the rolling disc."""
        super()._define_kinematics()
        # Define the yaw-roll-pitch orientation of the disc.
        self.wheel.frame.orient_body_fixed(self.ground.frame, self.q[2:], "zxy")
        # Define the position of the contact point in the ground plane.
        self.ground.set_pos_point(self.tyre.contact_point, self.q[:2])
        # Define the kinematics of the tyre model.
        self.tyre.define_kinematics()
        # Add coordinates, speeds and kinematic differential equations to the system.
        self.system.add_coordinates(*self.q)
        self.system.add_speeds(*self.u)
        self.system.add_kdes(*(self.q.diff(dynamicsymbols._t) - self.u))

    def _define_loads(self) -> None:
        """Define the loads of the rolling disc."""
        super()._define_loads()
        self.tyre.define_loads()

    def _define_constraints(self) -> None:
        """Define the constraints of the rolling disc."""
        super()._define_constraints()
        self.tyre.define_constraints()
```

Listing C.6: Code defining a simplified version of the `RollingDisc` model class.

C.5. Control Load Group Implementation

A method to apply a drive, roll and steer torque to the wheel of the rolling disc is by utilizing a load group. To define a load group the first step is to set the required type of the parent. In this case, the rolling disc is chosen as parent. Through this definition `LoadGroupBase` automatically makes sure that this load group can only be added to an instance of the `RollingDisc` model, which is accessible in the load group using the `parent` attribute. Another definition by `LoadGroupBase` is that the system attribute refers to the system of the parent component.

The loads within a load group are defined in the define steps. `_define_objects` creates the torque quantities for which a description is given in the `descriptions` property. `_define_loads` actually computes the directions of the torques and adds them to the system. The exemplifying code of this load group is shown in listing C.7.

```
217 class RollingDiscControl(LoadGroupBase):
218     """Rolling disc control load group."""
219     required_parent_type = RollingDisc
220 
221     @property
222     def descriptions(self) -> dict[Any, str]:
223         """Dictionary of descriptions of the rolling disc's controls."""
224         return {
225             **super().descriptions,
226             self.symbols["T_drive"]: "Drive torque.",
227             self.symbols["T_roll"]: "Roll torque.",
228             self.symbols["T_steer"]: "Steer torque.",
229         }
230 
231     def _define_objects(self) -> None:
232         """Define objects."""
233         super()._define_objects()
234         self.symbols.update({
235             name: dynamicsymbols(self._add_prefix(name)) for name in (
236                 "T_drive", "T_roll", "T_steer")
237         })
238 
239     def _define_loads(self) -> None:
240         """Define loads."""
241         super()._define_loads()
242         roll_axis = cross(self.parent.ground.normal, self.parent.wheel.rotation_axis)
243         upward_radial_axis = cross(self.parent.wheel.rotation_axis, roll_axis)
244         self.system.add_loads(
245             Torque(self.parent.wheel.frame,
246                    self.symbols["T_drive"] * self.parent.wheel.rotation_axis +
247                    self.symbols["T_roll"] * roll_axis +
248                    self.symbols["T_steer"] * upward_radial_axis,
249                    )
250         )
```

Listing C.7: Code defining a load group to control the rolling disc using three torques.

C.6. Build Rolling Disc

With all components defined the rolling disc model is built, as shown in listing C.8. It starts with configuring the model by describing out of which components the rolling disc is composed. Next, it defines the entire model using `rolling_disc.define_all()`, refer to section 3.5.5 for a summarizing overview of what happens in each step. With the model defined it is exported to a single system instance on which gravity is applied. The last step before generating the EOMs based on the system is to specify which generalized speeds are independent and which are dependent. The `system.validate_system()` is an optional feature to verify the system for any mistakes, like having a different number of velocity constraints then dependent generalized speeds.

```
# Configure the model by describing the components it consists out of.
rolling_disc = RollingDisc("disc")
rolling_disc.wheel = KnifeEdgeWheel("wheel")
rolling_disc.ground = FlatGround("ground")
rolling_disc.tyre = NonHolonomicTyre("tyre")
rolling_disc.add_load_groups(RollingDiscControl("controls"))
# Run all define steps.
rolling_disc.define_all()
# Export the model to a single instance of System.
system = rolling_disc.to_system()
# Apply gravity.
system.apply_gravity(-Symbol("g") * rolling_disc.ground.normal)
# Define which generalized speeds are indepedent and dependent
system.u_ind = rolling_disc.u[2:]
system.u_dep = rolling_disc.u[:2]
# Run some basic validation of the system before forming the equations of motion.
system.validate_system()
system.form_eoms()
```

Listing C.8: Code building the rolling disc model and forming its EOMs.