Technical University of Munich
TUM School of Engineering and Design

Modeling and Simulation of Physical Systems with Variable Structure

Andrea Neumayr

Complete reprint of the dissertation approved by the TUM School of Engineering and Design of the Technical University of Munich for the award of the

Doktorin der Ingenieurwissenschaften (Dr.-Ing.).

Chair: Prof. Dr.-Ing. Hans-Georg Herzog

Examiners:

1. Hon.-Prof. Dr.-Ing. Martin Otter
2. Prof. Dr. rer. nat. Martin Arnold

The dissertation was submitted to the Technical University of Munich on 22.01.2025 and accepted by the TUM School of Engineering and Design on 17.06.2025.

Acknowledgments

This thesis is the result of my research at the Institute for System Dynamics and Control of the German Aerospace Center (DLR) at Oberpfaffenhofen, Germany. In this regard, I would like to thank Prof. Johann Bals for affording me the opportunity to conduct this exciting research at his institute.

My deepest thanks and appreciation go to my open-minded and inspiring supervisor, Prof. Martin Otter, a luminary in the field of Modelica. Prof. Otter gave me absolute research freedom and never-ending encouragement.

Thank you to all of my current and former colleagues who created such a pleasant working atmosphere, gave moral support and helped me with any and all kinds of problems along the way. A special thank you to Dr. Gerhard Hippmann for his amazing code-reviews and for the many fruitful discussions that we had. Thank you also to Dr. Dirk Zimmer, an expert on modeling and Modelica for providing many invaluable suggestions and questions.

I would like to express my gratitude to the Modelica community for providing a welcoming and supportive environment, especially at conferences, along with inspiring presentations and delightful discussions. In this regard, I would especially like to thank Dr. Hilding Elmqvist for the rich and rewarding scientific discourse.

Thank you to all my friends for standing by my side at all times and for regularly uplifting my mood. A very special thank you to my friend of many years, Dr. Andrea Kulmhofer-Bommer, whose guidance helps me to constantly improve my way of scientific writing, not only for this thesis. I would like to give credit to Richard Fornefeld for answering all my questions regarding the English language.

I am deeply grateful to my family for their unconditional love, support, patience and for encouraging me in everything I do. Finally, I am forever grateful to my partner, Dr. Andreas Knoblach, for his unwavering belief in me and for keeping me in balance. I cherish the many invaluable discussions with him and his outstanding LaTeX- and Git-support. Thank you for being part of my life!

Munich, January 2025

Andrea Neumayr

Abstract

The modeling and simulation of multidisciplinary, physical systems present several fascinating practical applications. One complex example of such applications is collision handling. Real-world phenomena are approximated to prevent, to reduce, or to analyze collision situations. For example, space robots capture, manipulate and position spacecraft modules. While doing so, several problems can emerge. To address these, two methods of collision handling with variable-step solvers are considered in this thesis:

1. Distance and force computation.
2. A novel approach to variable structure systems.

The first method consists of four collision handling steps using distance and force computation. One of these steps computes point contact details and a penetration depth or the Euclidean distance between two potentially-colliding objects. If these objects penetrate, the resulting force is determined using elastic material laws based on the theory of Hertz for elastic shapes. This method will most likely only work reliably for continuously varying point contacts and penetration depths. Hence, each collision handling step must ensure this condition is met.

The second method uses a novel approach to variable structure systems to minimize or completely eliminate time-consuming distance calculations. Systems with variable structure are a hypernym for models in which the number of equations vary during simulation. Current state-of-the-art publications for variable structure systems require prior knowledge of all occurring models to switch between them or to reprocess and recompile the entire model. The new general approach allows a varying number of equations, which leads to so-called segments, without having to recompile the code. The user needs to be sufficiently familiar with the multidisciplinary system and its required conditions to interact via certain (user-defined) commands to trigger new segments. These segments need not be known in advance. This new method is validated and seems very promising for future applications. Moreover, it allows the combination of both collision handling methods to achieve the optimal benefits of each. This opens up a wide range of exciting new possibilities.

Contents

List of Abbreviations

Abbreviation	Description	Page
1D	1-dimensional	
2D	2-dimensional	
3D	3-dimensional	
AABB	Axis-Aligned Bounding Box	52
AST	Abstract Syntax Tree	17
BDF	Backward Differentiation Formula	21
BVH	Bounding Volume Hierarchy	53
DAE	Differential Algebraic Equation	11
DFS	Depth-First Search	29
DoF	Degrees of Freedom	
EPA	Expanding Polytope algorithm	58
FEM	Finite Element Method	57
GJK	Gilbert-Johnson-Keerthi	58
iff	if and only if	
inv	invariant	121
k-DOP	k-Discrete Orientation Polytope	52
MPR	Minkowski Portal Refinement	59
OBB	Oriented Bounding Box	52
ODE	Ordinary Differential Equation	12
PCM	Polygonal Contact Model	57
pw	piecewise	116
QSS	Quantized State Systems	4
var	variant	122

List of Symbols

For better readability and simplicity the dependencies are skipped, and some shortcuts are used throughout this thesis e.g., $\delta_{\mathrm{d}} = \delta_{\mathrm{d}}(A, B), A, B \in \mathcal{C}, A \neq B$ or $z_{\mathrm{AB}} = z_{\mathrm{AB}}(t), t \in \mathbb{R}_0$. Here, dependencies of A, B or t are omitted.

Symbol	Description	Page
a	Scalar	
$\boldsymbol{a}$	Vector	
$\emptyset$	Empty set	
∂	Boundary of set	
$\mathbb{N}_0$	Set of nonnegative integer natural numbers, including 0	
$\mathbb{R}$	Set of real numbers	
$\mathbb{R}_0$	Set of nonnegative real numbers, including 0	
$\mathbb{R}_{>0}$	Set of nonnegative real numbers, without 0	
$\mathbb{R}^n$	Set of n vectors with elements in $\mathbb{R}$	
$\mathrm{conv}(\cdot)$	Convex hull of a polytope	
$\mathrm{vertex}(\cdot)$	Vertices of a polytope	
$\lvert\cdot\rvert$	Absolute value	
$\lVert\cdot\rVert$	Euclidean norm or 2-norm	
$\boldsymbol{a} \cdot \boldsymbol{b}$	Scalar or inner product	
$\boldsymbol{a} \times \boldsymbol{b}$	Cross product or vector product	

Julia Mathematical Functions

Symbol	Description	Page
`abs()`	Absolute value	
`cross(a,b)`	Cross product or vector product	
`dot(a,b)`	Scalar or inner product	
`norm()`	Euclidean norm or 2-norm	
`normalize()`	Normalized vector	
`sign(a)`	Signum: 0 if $a == 0$ and $\frac{a}{\lvert a\rvert}$ otherwise	

Collision Handling with Variable-Step Solvers

Symbol	Description	Page
$A \in \mathcal{C}$	Collision object	43
$A, B \in \mathcal{C}$	Collision pair, potentially-colliding pair	43
$A \ominus B$	Minkowski Difference	43
$A \oplus B$	Minkowski Sum	43
$\mathcal{C}$	Collision set of all potentially-colliding objects	43
cor	Coefficient of restitution	91
d	Damping coefficient	89
δ_{broad}	Euclidean distance in broad phase	53
δ_{d}	Euclidean distance, separation distance, distance	44
δ_{mpr}	Signed distance in narrow phase	76
δ_{p}	Penetration depth	44, 81
$\dot{\delta}_{\text{p}}$	Penetration velocity	89
E	Young's modulus of solid material	91
$E^{\star}$	Combined Young's moduli and Poisson's ratios	89
$\boldsymbol{e}_n$	Unit vector normal to the contacting surfaces	89
$\boldsymbol{e}_{t,\text{reg}}, \boldsymbol{e}_{\omega,\text{reg}}$	Regularized direction vectors in direction of the relative tangential and relative angular velocity	89
$\boldsymbol{f}_n$	Contact force in normal direction	89
$\boldsymbol{f}_t$	Contact force in tangential direction	89
k_{red}	Elastic contact reduction factor	89
μ_k	Kinetic/sliding friction force coefficient	89
μ_r	Rotational resistance torque coefficient	89
$n_{\mathcal{C}}$	Number of potentially-colliding objects	43
n_{pp}	Number of potentially-colliding objects after preprocessing step	51
ν	Poisson's ratio of solid material	91
R_{geo}	Geometry dependent coefficient	89
$\boldsymbol{\tau}_{\omega}$	Contact torque	89
z_{AB}	Zero-crossing function	82
z_{d}	Zero-crossing function for separated objects	83
z_{p}	Zero-crossing function for penetrating objects	84

Variable Structure Systems

Symbol	Description	Page
$\boldsymbol{c}_f$	Flow variables	116
$\boldsymbol{c}_p$	Potential variables	116

Symbol	Description	Page
$\boldsymbol{f}_{\mathrm{c}}$	Mathematical description of an acausal component	116
$\boldsymbol{f}_{\mathrm{c},1}$, $\boldsymbol{f}_{\mathrm{c},2}$	Pre-translated functions	118
$\boldsymbol{f}_{\mathrm{c,eq}}$	Set of equations to be solved with the system	118
$\mathtt{m}$	Internal memory	120
$\boldsymbol{p}$	Parameters	116
t	Time	116
$\boldsymbol{u}$	Inputs	116
$\boldsymbol{w}$	Local variables	116
$\boldsymbol{x}$	Continuous states	116
$\boldsymbol{y}$	Outputs	116
$\boldsymbol{z}$	Event indicators	116

1 Introduction

Modeling and simulation are powerful tools for understanding and predicting real-world phenomena. In science and engineering, it is crucial to test hypotheses, to experiment, to explore, to analyze, and to gain insights without the need for physical experiments. Thus, physical systems are approximated, resulting in corresponding mathematical models. Approximation focuses only on the essential aspects and neglects the less important ones. The model's accuracy depends on the underlying assumptions, simplifications and available data.

At present, there is no suitable simulation tool to adequately simulate multidisciplinary systems with collision handling and variable structure systems with a varying number of degrees of freedom. With existing simulation tools, these problems (e.g., space applications like a two-stage rocket or a relocatable space robot in Figure 1.1) can only be simulated by experts with a lot of know-how. To overcome this limitation, Modia (Elmqvist, Henningsson, and Otter, 2017) a modeling and simulation environment is used. Modia is inspired by the equation-based modeling language Modelica (Modelica Association, 2021) for which several open-source and commercial tools are available. Modia is a domain-specific extension of the Julia programming language (Bezanson et al., 2017). Thus, Modia benefits from the fast-growing, feature-rich Julia ecosystem.

In both simulation environments, Modia and Modelica, physical systems are mathematically described by Differential Algebraic Equations (DAEs) that are

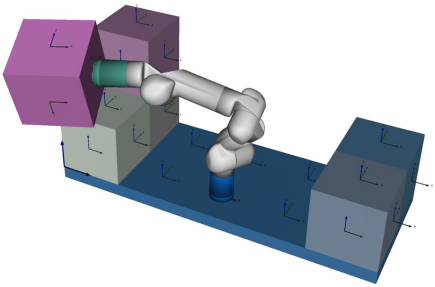


Figure 1.1: Relocatable space robot as part of the MOSAR space project (Letier et al., 2019). This space application highlights variable structure systems in Section 12.3.

automatically transformed to Ordinary Differential Equations (ODEs). This conversion occurs only once, and before the simulation is executed. The resulting ODEs are retained and solved during the simulation. If the structure of the model changes – known as a variable structure system – so does its underlying mathematical description with DAEs.

Collision handling is a modeling and simulation application. It is used to prevent collisions (e.g., autonomously driving cars), to reduce collisions (e.g., robotics applications), or to analyze collisions (e.g., gripping processes). Collision handling is a complex task during which several problems can arise. To address these, two methods of collision handling are considered in this thesis:

1. Distance and force computation with variable-step solvers.
2. A novel approach to variable structure systems.

The latter minimizes or completely eliminates the need for time-consuming collision calculations. In order to gain a better understanding of these two methods, short introductions to collision handling and variable structure systems follow.

Introduction to Collision Handling with Variable-Step Solvers

One constraint of the real world is that two solid objects cannot occupy the same point in space at the same time. In simulation environments, this condition is generally not imposed and a collision occurs if two objects have at least one point in common. Collision handling consists of two aspects: collision detection and collision response. Collision detection is the mathematical problem of determining whether and where two geometric objects are intersecting (Bergen, 2003; Ericson, 2004). Collision response is a kinetic problem involving the motion of two or more objects during a collision (Bourg and Bywalec, 2013).

Collision handling is widely researched with extensive literature available, especially in the field of game physics and virtual reality (e.g., Bergen and Gregorius, 2010; Millington, 2010; Bourg and Bywalec, 2013; Szauer, 2017). However, in the real world, collision avoidance is needed for autonomously driving cars and in robotics applications. Modeling and simulating the collisions between geometrical objects is complex and there are many methods and algorithms dedicated to particular purposes. A survey of collision detection methods for convex and concave rigid bodies, as well as for deformable shapes, is given in e.g., Avril, Gouranton, and Arnaldi (2009) and Mainzer (2015). In computer games, it is important that collisions of many objects are supported in real-time and that the resulting simulation appears reasonably realistic. Game engines, like Bullet Physics SDK (Bullet, online; Coumans, online), Unity (Unity Technologies, online), and Unreal Engine (Epic Games, online) usually make tradeoffs in terms of physics. Erez, Tassa, and Todorov (2015) point out that PhysX (PhysX, online) and Havok (Havok, online) neglect Coriolis forces. They also compare a number of physics engines and identify that each of these performs best for the case it is designed for.

When simulating real-world problems, in contrast, it is important that collision models reflect the actual physical behavior and are validated using data from real experiments. Therefore, the simulation results must be closer to reality than in a game.

Regardless of the procedure used, there are $n_{\mathcal{C}}$ objects in a collision set $\mathcal{C}$ that could potentially collide. A distance calculation of all potential collision pairs would result in $\frac{n_{\mathcal{C}}(n_{\mathcal{C}}-1)}{2}$ collision tests. For reducing $O(n_{\mathcal{C}}^2)$ complexity, collision handling is performed in four steps[1]:

1. Preprocessing (see Chapter 5)
   The mechanical structure is mapped onto an internal tree structure by a once-off additional preprocessing step. This reduces the number of potential collision pairs (Neumayr and Otter, 2018).
2. Broad Phase (see Chapter 5)
   An efficient and simple selection procedure is performed. Therefore, all objects are approximated by simple bounding volumes. Cheap intersection tests check if those bounding volumes overlap or not (e.g., Bergen, 2003). There are several different possible bounding volumes. In this thesis, Axis-Aligned Bounding Boxes (AABBs) are used.
3. Narrow Phase (see Chapter 6)
   An exact and computationally expensive intersection test is performed only for objects whose bounding volumes intersect. Depending on the approach, there are several different algorithms. In this thesis, an enhanced version of the Minkowski Portal Refinement (MPR) algorithm is used to calculate the signed distance, contact normal and contact points: It is based on previous publications to the MPR algorithm (Snethen, 2008; Olvång, 2010; Kenwright, 2015; Neumayr and Otter, 2017).
4. Response Calculation (see Chapter 7)
   The collision response depends on the chosen collision approach. For a comparison of these approaches see Otter, Elmqvist, and López (2005) and Hofmann et al. (2014). In this thesis, the collision behavior is calculated using elastic material laws, based on the penetration depth and the contact points of two penetrating objects. It is possible to apply a force and/or torque at the contact point, such as a spring-damper force element (e.g., Neumayr and Otter, 2017). Furthermore, Neumayr and Otter (2019b) propose a different approach. A normal and a tangential force are applied at the contact point as function of the penetration depth and its derivative, such as a spring-damper force element. This leads to a novel elastic-response characteristic.

Step 1 is executed only once. The three remaining collision handling steps must be executed each time the underlying mathematical formulation – a hybrid DAE system – is solved. The time of a collision, i.e., touching time, is not known in

[1] This is an overview, details, definitions and explanations are given in the referred chapters.

advance, but the model behavior and the resulting forces change drastically when a collision occurs or ends. In order to obtain an accurate physical simulation, variable-step solvers are used to determine the exact start and end time of a collision. This means that each collision handling step needs to guarantee to be usable with variable-step solvers. If there are many collisions, the use of Quantized State Systems (QSS) solvers (Cellier and Kofman, 2006; Floros et al., 2011) might be more appropriate.

Introduction to Variable Structure Systems

Variable structure systems are very common in physical modeling. The dynamic structural changes of physical systems are caused e.g., by mechanical elements that break apart, systems with clutches, or the reconfiguration of robot models. Variable structure systems can be used to minimize or to eliminate collision calculations, as shown in Chapter 12. In the context of object-oriented and equation-based modeling structural changes result in a varying number of variables (both algebraic variables and states) and also a varying number of equations during simulation. The underlying mathematical description represented by DAEs changes if the structure of the model is changed. DAEs are symbolically transformed to ODEs, by structural-analysis methods like the Pantelides algorithm (Pantelides, 1988) or Pryce's Σ-method (Pryce, 2001). The resulting ODEs are retained and solved during the simulation with standard numerical methods. It is difficult to handle dynamic changes of the structure properly and efficiently without having to reanalyze the equations and recompile the code, or to manually initialize a new model.

In the literature, variable structure systems are also known as multi-mode models. Multi-mode modeling defines model components with state machines, where the number of component equations change whenever a transition to another state occurs (see e.g., Elmqvist, Matsson, and Otter, 2014). Nytsch-Geusen et al. (2006) expand the Modelica language to allow structural changes during simulation to activate and deactivate different parts of the model. Therefore, the entire model is decomposed into a finite set of modes. This is only a viable solution if the structural changes are modeled at the top-level, but not if they are triggered by single components.

A transition of structure can lead to Dirac impulses. Benveniste et al. (2019), Caillaud, Malandain, and Thibault (2020) and Benveniste et al. (2022) extend the structural analysis with Pantelides algorithm and Pryce's Σ-method for multi-mode models. In Benveniste et al. (2019) it is demonstrated how a multi-mode Modia model is treated. For a particular model, the code is regenerated and recompiled on-the-fly with a special prototype during a state transition. It is then initialized in the new states, even if Dirac impulses occur.

Höger (2014) also works on Pryce's Σ-method for variable structure modeling. Zimmer (2010) uses a runtime interpreter to transform the DAE equations at runtime, when the structure changes. Limitations of this approach are that im-

pulsive behavior is not supported and simulation time is one or more orders of magnitude greater than when compiled code is used. Pepper et al. (2011) describe the semantics of variable structure modeling with state machines. Mehlhase (2014) provides a Python-based approach during which transitions can be made between predefined models. Elmqvist, Matsson, and Otter (2014) and Mattsson, Otter, and Elmqvist (2015) propose a high-level description of multi-mode models in Modelica by extending the clocked synchronous state machines to continuous-time state machines.

Tinnerholm, Pop, and Sjölund (2022) provide a Julia-based implementation of Modelica called OpenModelica.jl which supports variable structure systems. A distinction is made between explicit and implicit variable structure systems. For explicit variable structure systems, the transition between states is explicitly encoded by the user. Thus, all equations and variables are known beforehand. Both the compiler and the simulation runtime need to process the whole model at once. For implicit variable structure systems, predefined events trigger a recompiling of the model on-the-fly during simulation.

All current proposals for variable structure systems require prior knowledge of all models – for all modes – to switch between these models during simulation. If this knowledge is not available, and whenever the equation structure changes, the entire model is reprocessed and its code is regenerated and recompiled (or interpreted). In this thesis, a new general approach for dealing with variable structure systems is introduced, in which variables can appear and disappear during simulation. There is no need for regenerating and recompiling code when the number of equations and states are varying at events. The method is presented in a generic, mathematical way and can be applied to declarative, equation-based modeling languages, such as Modia and Modelica. The transition between the modes, referred to as segments, is triggered by specific commands. Both the number of variables and equations can vary from segment to segment. Several novel features are introduced to overcome current limitations for variable structure systems.

- The sizes of array equations can be changed after code generation and before simulation starts. A simple example is given in Chapter 9.
- For dynamically varying the number of states during simulation, the new concept of predefined acausal components is introduced in Chapter 10. The automated procedure of analyzing the structure of DAEs, and transforming them to ODEs, results in causal equations (assigned equations) and acausal equations (not assigned equations). The same procedure can be applied to acausal components as well. Furthermore, the component equations are also split into causal and acausal equations. The causal equations of the acausal components become pre-translated functions. The states computed by these pre-translated functions are hidden in memory and are directly passed to the solver. However, the acausal equations of components must be solved as part of the entire model.

In this new approach, predefined acausal components cannot be designed for arbitrary (useful) connection scenarios. This means, they cannot be used if one or more of their pre-translated functions need to be differentiated and not all equations of an entire model are reprocessed when the model structure changes. Yet, a considerable number of systems can still be practically handled. However, users must be acquainted with the system and its required conditions, and provide significant information about the model in advance, in order to trigger a new segment. Interaction with the model is achieved using specific commands that are tailored to the predefined acausal components. This allows great flexibility when creating customized interactions for the component and the model. The use of predefined acausal components scales well for large systems and leads to efficient simulations because code is not regenerated and recompiled on-the-fly.

1.1 Contributions

Notable contributions are made to collision handling with variable-step solvers with distance and force computation, to variable structure systems, and to the modeling and simulation environment Modia and multibody module Modia3D.

These contributions pertain to collision handling with variable-step solvers and are preliminarily published in Neumayr and Otter (2017), Neumayr and Otter (2018), Neumayr and Otter (2019b) and Neumayr and Otter (2020). A once-off additional preprocessing step that analyzes the mechanical structure is invented to reduce the number of potential collision pairs. Several considerable enhancements to the MPR algorithm, which is used in the narrow phase are introduced and discussed in detail. The novel concept of the *signed distance* is crucial for collision handling with variable-step solvers. The signed distance is the Euclidean distance (when both objects are separated) or the penetration depth (when both objects are penetrating). In addition, to ensure the existence of a Euclidean distance between separated pairs, the Euclidean distance between their AABBs in the broad phase is used. Moreover, to avoid unnoticeable penetrations of shapes, loose-fitting AABBs are proposed. It is also proven that the signed distance is suitable as a zero-crossing function that is used to detect the transition between penetration and non-penetration, and vice-versa by variable-step solvers. The zero crossing itself corresponds to the touching time. An associated zero-crossing function exists for each collision pair. Regardless of how many potential collision pairs actually exist, a new method is proposed to reduce the number of zero-crossing functions to two. To avoid unphysical behavior in the response calculation, several restrictions are introduced to ensure a continuous penetration depth. A novel force and torque formulation is developed by combining and enhancing existing response formulations.

The most noteworthy contribution is a significant new approach to variable structure systems. This is preliminarily published by Neumayr and Otter (2023a). It is a general approach that is discussed from the mathematical- and from the

algorithmic side. It further allows a varying number of states before simulation starts, as well as during simulation. In doing so, customized action commands interact with special components. These commands trigger the addition or removal of the components' states during simulation, without having to recompile the model's code. Moreover, benchmarks are established for a transportation scenario. These are preliminarily published by Neumayr and Otter (2023b). They show that time-consuming collision handling can be eliminated or reduced to a minimum. Furthermore, it is possible to combine both collision handling and segmented simulations. This allows collision handling to be turned on for segments of particular interest and turned off for segments that are of no interest. As a proof of concept for variable structure systems, the flipping of a kinematic chain (and vice-versa) during simulation is realized by Neumayr and Otter (2024). This opens up a wide range of exciting new applications in space robotics, industrial and scientific areas.

Furthermore, other contributions relevant to the modeling environment Modia and Modia3D are highlighted. These are preliminarily published by Neumayr and Otter (2018), Neumayr and Otter (2019a), Elmqvist, Otter, Neumayr, and Hippmann (2021) and Neumayr and Otter (2024). The open-source multibody module Modia3D is modular and has a customizable component-based design pattern. Algorithms are introduced to group objects according to their properties and to analyze them efficiently. These algorithms are inspired by the new additional preprocessing step of collision handling. Among others, the combination and symbolic transformations of Modia and Modia3D represent crucial developments for variable structure systems.

1.2 Outline

This thesis is devoted to collision handling and two different approaches are discussed: one with distance and force computations and the other with variable structure systems. It is divided into three parts:

Part I – Fundamentals. Basic concepts of equation-based modeling with DAEs, multibody equations, and variable structure systems are explained in Chapter 2. An overview about the open-source modeling and simulation environment Modia, which describes physical systems by DAEs, is presented in Chapter 3. The open-source multibody module Modia3D, for mechanical systems, is closely integrated with Modia and is symbolically transformed. Modia3D is a predefined acausal component.

Part II – Collision Handling with Variable-Step Solvers. Three state-of-the-art collision handling steps, as well as a once-off additional preprocessing step, are used. Basic concepts needed for these steps are introduced in Chapter 4. Each step is exhaustively debated in Chapters 5 to 7. Each step is further developed so that it can be used with variable-step solvers. The adaptation of Modia3D and

critical considerations concerning collision handling with variable-step solvers are discussed in Chapter 8.

Part III – Variable Structure Systems. The possibility of varying the number of states before simulation is shown in Chapter 9. A new general approach that allows the varying of the number of states during simulation is introduced in Chapter 10. The adaptation of Modia3D to variable structure systems is discussed in Chapter 11. In contrast to the theoretical concepts mentioned in this thesis, detailed overviews of selected practical sample applications are presented in Chapter 12. Several critical deliberations are made and inherent issues of the two approaches are elaborated. Based on that, as well as the questions arising from it, specific areas that require further research are identified in Chapter 13. Appendix A supplies the interested reader with some mathematical definitions. The MPR algorithm is examined with several selected collision models. These illustrative models are explained briefly in Appendix B.

Part I

Fundamentals

2 Equation-Based Modeling

In equation-based modeling, a physical system is described with Differential Algebraic Equations (DAEs). The whole model or its individual components are represented by a set of equations and associated variables. All equations from all components are collected in one set when the whole model is compiled. A prerequisite for any object-oriented modeling approach is that the behavior of the whole system can be derived from the behavior of its components. General laws for the connections between the components make the equations of each component applicable and reusable. Thus, pairs of flow and potential variables are introduced by adding equations to the system that represent the connections between components.

- Flow variable: The sum of the inflowing inputs and the outflowing outputs must equal 0 at a connection point.
- Potential variable: The values of all inputs and outputs at a connection point are equal.

Depending on the domain, there are specific flow and potential pairs, e.g., voltage and current in electricity, force and velocity in translational mechanics, torque and angular velocity in rotational mechanics (see e.g., Zimmer, 2010). The final system contains a mixture of purely algebraic equations and differential equations. Thus, it is called a DAE system.

This chapter outlines the main concepts of equation-based and object-oriented modeling languages, like Modelica and Modia. The Modelica language is standardized, declarative, and equation-based (Modelica Association, 2021). There are open-source and commercial tools supporting Modelica (Modelica Association, online). The Modelica language is widely used in scientific and industrial applications for modeling, simulation, and design of physical systems. Modelica defines large sets of differential, algebraic, and discrete equations in a standardized, user-friendly manner. Based on Modelica, Modia is a revised approach towards equation-based modeling. Modia3D is a multibody module for mechanical systems and belongs to the Modia modeling environment. Modia3D serves as a testbed for collision handling with variable-step solvers and variable structure systems. For both approaches, state events are crucial.

2.1 Differential Algebraic Equations

Physical systems are mathematically described by Differential Algebraic Equations (DAEs). DAEs occur in the mathematical modeling of a wide variety of problems in engineering and science such as in multibody and flexible body mechanics, electrical circuit designs, optimal control, compressible and incompressible fluid dynamics, molecular dynamics, chemical kinetics, and chemical process control.

The general representation of a DAE is given by

$$\boldsymbol{F}(\dot{\boldsymbol{x}}_{\mathrm{DAE}}, \boldsymbol{x}_{\mathrm{DAE}}, \boldsymbol{u}, t) = 0, \tag{2.1}$$

where $\boldsymbol{x}_{\mathrm{DAE}}(t)$ are the unknown variables and $\boldsymbol{u}(t)$ is the input vector. Both are dependent on time $t \in \mathbb{R}$. $\boldsymbol{F}$ represents the equations of the system. (See e.g., Brenan, Campbell, and Petzold, 1996; Cellier and Kofman, 2006)

There are two standard ways of solving general DAEs (2.1): First, a system of DAEs (2.1) can be solved numerically with DAE solvers such as DASSL (Brenan, Campbell, and Petzold, 1996) or IDA from the Sundials suite (Hindmarsh et al., 2005; Hindmarsh, Serban, and Collier, 2015). This approach has some limitations. For this reason, there are solvers for DAEs with a particular structure (Arnold, 2017).

Second, a system of DAEs $\boldsymbol{F}$ (2.1) can be transformed into Ordinary Differential Equations (ODEs) in explicit state-space form

$$\dot{\boldsymbol{x}}_{\mathrm{ODE}} = \boldsymbol{f}(\boldsymbol{x}_{\mathrm{ODE}}, \boldsymbol{u}, t), \tag{2.2}$$

and solved with ODE solvers. $\boldsymbol{x}_{\mathrm{ODE}}$ are the states of the system. Typically, $\boldsymbol{x}_{\mathrm{ODE}}$ is a subset of $\boldsymbol{x}_{\mathrm{DAE}}$. The non-trivial transformation from an implicit DAE system to an explicit ODE system can be performed symbolically and automated by any compiler of equation-based modeling languages. If the structure of the physical system changes during simulation – known as a variable structure system – so does its underlying mathematical description represented by DAEs and its corresponding ODEs. Therefore, it would be required to execute the computationally expensive transformation and compilation from DAEs to ODEs again.

Of course, for simple DAE systems one could transform a DAE into an ODE manually, by using substitution and Gaussian elimination until all unknowns are assigned and inserted to compute the states of the ODE.

Since this is not feasible for larger systems, this procedure must be automated and generalized for compilers to analyze the structure of DAEs. There are several steps which need to be performed by the compiler, each can be solved by various algorithms. The symbolic transformation steps are outlined briefly below; for an extended description with some examples see Cellier and Kofman (2006).

1. Causalization (Sorting): Initially, starting from the DAE, all equations are acausal. This means, the equal sign has to be interpreted in the sense of an equality, rather than in the sense of an assignment (Cellier and Kofman,

2006).[1] An algorithm is needed to assign equations and unknowns. This means finding causal equations where its unknown can be evaluated directly from its assigned equation. If an unknown appears in more than one equation, there must be rules to decide which variable to solve for. Equations are causal if they have been sorted horizontally. If the equations have been sorted vertically, they are in an executable sequence. This executable sequence is referred to as sort order. The set of equations and its unknowns can be regarded as a bipartite graph[2], where each equation and each unknown are represented as nodes. The node of an equation and an unknown is connected via an edge, if the unknown appears in the equation. The idea is finding a matching and thus making the equations causal. Pantelides (1988) proposed the most popular causalization algorithm. The procedure creates an overdetermined equation system, by differentiating equations and variables and adding them to the system. During the process, it selects the equations to be solved (Cellier and Kofman, 2006).

2. Algebraic Loops (System of Algebraic Equations): Causalizing DAE systems is not always straight forward because there might be equations left, which need to be solved together. These remaining acausal equations are called algebraic loops or system of algebraic equations. DAEs with algebraic loops often appear e.g., in multibody systems containing closed kinematic loops. The algebraic loop equations need to be solved together. If all algebraic loop equations are linear, the unknowns can be solved for by Gaussian elimination. If they are nonlinear, Newton iteration is needed to solve them. The DAE translator needs to extract such loops and generate code for a numerical solution.
3. Tearing Algorithm: It is a symbolic algorithm for general algebraic loops to reduce the size of an equation system. The set of equations is torn apart by assuming that one variable or potentially several variables are known. These variables are called tearing variables. The equations from which the tearing variables are to be computed are called residual equations. The selection of tearing variables is not arbitrary. Finding the minimum number of tearing variables for an algebraic loop is an np-complete problem. Still, there are heuristic procedures for finding a small number of tearing variables (Cellier and Kofman, 2006).
4. Structural Singularities: Structural singularities occur when so-called DAE constraint equations are present in the system. This means that some state variables depend algebraically on each other. In other words, their

[1] In a physical sense, causal means that: The present states are dependent from the states of the past, but are independent from the states of the future. Causality in the context of modeling describes the computational flow.

[2] In other words, a bipartite graph is a graph with two disjoint sets of nodes. The nodes within one set are not connected via edges.

initial conditions cannot be chosen independently. To tackle this issue, the constraint equations are symbolically differentiated (Pantelides, 1988), and added to the equation set. In the process of differentiation so-called pseudo-derivatives also known as dummy-derivatives are introduced (Mattsson and Söderlind, 1993). This procedure is explained at the basis of some examples by Cellier and Kofman (2006).

After these symbolic transformations, the last step is to generate code, e.g., Julia or C. Subsequently, the code is compiled.

2.2 Multibody Equations

Multibody systems are mechanical systems consisting of several linked rigid or flexible bodies that can be displaced in translation and rotation. The connections between these bodies allow complex motions and interactions. In this thesis, only rigid bodies are considered. Multibody systems with kinematic loops are mathematically described by the equations of motion (see e.g., Arnold, 2017):

$$\begin{aligned} \dot{\boldsymbol{q}} &= \boldsymbol{v} \\ \boldsymbol{M}(\boldsymbol{q},t)\dot{\boldsymbol{v}} + \boldsymbol{G}^T(\boldsymbol{q},t)\boldsymbol{\lambda} + \boldsymbol{h}(\boldsymbol{q},\boldsymbol{v},t) &= \boldsymbol{\tau} \\ \boldsymbol{0} &= \boldsymbol{g}(\boldsymbol{q},t), \end{aligned} \tag{2.3}$$

where $\boldsymbol{q}$ are the generalized coordinates of the joints of the spanning tree (e.g., the angle of a revolute joint), $\boldsymbol{v}$ are the derivatives of $\boldsymbol{q}$, $\boldsymbol{\tau}$ are the generalized forces in the joints of the spanning tree (e.g., the driving torque of a revolute joint), $\boldsymbol{\lambda}$ are the generalized forces/torques in the cut-joints[3], $\boldsymbol{M} = \boldsymbol{M}^T$ is the positive definite mass matrix, $\boldsymbol{g}$ are the kinematic constraint equations of the cut-joints on position level, $\boldsymbol{G} = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{q}}$ are the partial derivatives of the constraint equations with respect to $\boldsymbol{q}$ and has full row rank, and $\boldsymbol{h}$ are applied generalized forces. This DAE (2.3) gives rise to numerical problems when integrating it directly. Instead, the method of Gear, Leimkuhler, and Gupta (1985) and Gear (1988) can be used to transform it to a DAE (2.4) (see Otter and Elmqvist, 2017; Neumayr and Otter, 2019a; Neumayr and Otter, 2024)

$$\begin{aligned} \boldsymbol{0} &= \dot{\boldsymbol{q}} - \boldsymbol{v} + \boldsymbol{G}^T(\boldsymbol{q},t)\dot{\boldsymbol{\mu}}_{int} \\ \boldsymbol{0} &= \boldsymbol{M}(\boldsymbol{q},t)\dot{\boldsymbol{v}} + \boldsymbol{G}^T(\boldsymbol{q},t)\dot{\boldsymbol{\lambda}}_{int} + \boldsymbol{h}(\boldsymbol{q},\boldsymbol{v},t) - \boldsymbol{\tau} \\ \boldsymbol{0} &= \boldsymbol{g}(\boldsymbol{q},t) \\ \boldsymbol{0} &= \boldsymbol{G}(\boldsymbol{q},t)\boldsymbol{v} + \boldsymbol{g}^{(1)}(\boldsymbol{q},t), \end{aligned} \tag{2.4}$$

with much more beneficial numerical properties, where:

[3] In multibody modeling, a cut-joint is a specialized joint designed to handle kinematic loops in mechanical systems. Otherwise, the system might become over-constrained due to the closed kinematic loop.

1. The derivative of the constraint equations $\mathbf{0} = \boldsymbol{g}(\boldsymbol{q}, t)$ are added as new equations.
2. New unknowns $\dot{\boldsymbol{\mu}}_{int}$ are introduced to stabilize the DAE.
3. The generalized constraint forces $\boldsymbol{\lambda}$ are replaced by $\dot{\boldsymbol{\lambda}}_{int}$ the derivatives of its integral.

In this thesis, the focus is on the special case of tree-structured multibody systems where (2.3) and (2.4) simplify to the DAE

$$\begin{aligned} \dot{\boldsymbol{q}} &= \boldsymbol{v} \\ \boldsymbol{M}(\boldsymbol{q}, t)\dot{\boldsymbol{v}} + \boldsymbol{h}(\boldsymbol{q}, \boldsymbol{v}, t) &= \boldsymbol{\tau}. \end{aligned} \tag{2.5}$$

These equations can be transformed into the ODE

$$\begin{aligned} \dot{\boldsymbol{q}} &= \boldsymbol{v} \\ \dot{\boldsymbol{v}} &= \boldsymbol{M}^{-1}(\boldsymbol{q}, t)\left(\boldsymbol{\tau} - \boldsymbol{h}(\boldsymbol{q}, \boldsymbol{v}, t)\right) \\ &= \boldsymbol{f}_{\text{mbs}}(\boldsymbol{q}, \boldsymbol{v}, \boldsymbol{\tau}, t). \end{aligned} \tag{2.6}$$

These ODEs (2.6) are built up by the multibody module Modia3D, are symbolically transformed by Modia, and are essential for variable structure systems.

2.3 Variable Structure Systems

A model with different structural properties which change during simulation, such as a varying number of differential equations, are referred to as variable structure systems. Variable structure systems are very common in engineering problems.

- The structural changes are triggered by ideal switching processes, e.g., ideal diodes in electric circuits, rigid mechanical elements that break apart, systems with clutches, or the reconfiguration of robot models.
- The model has a variable number of variables, e.g., social or traffic simulations with a variable number of agents or entities.
- For efficiency reasons a variable structure is used, e.g., a more detailed model of a curved beam at the buckling point and less detailed elsewhere.
- User interaction creates variability in the structure, e.g., the user is allowed to create or connect certain components during simulation.

A variable structure system is a fairly general term that applies to a number of different modeling designs, e.g., adaptive meshes in finite elements, or discrete communication models for flexible computer networks. In the sense of equation-based modeling, a structural change means a change in the set of variables and a change in the equations between those variables during the simulation. These

changes can lead to significant changes in the structure of the model and are triggered by state events.

As the structure of the model changes, so does its underlying mathematical description with DAEs. In solvers for equation-based languages like Modelica and Modia it is difficult to deal with these changes because DAEs are automatically transformed to ODEs once at the beginning of the simulation. The resulting ODEs are solved during the simulation. In Part III of this thesis, one approach is presented to overcome this limitation.

2.4 State Events

Generally, events cannot only happen at predefined time points, but also when functions of continuously varying state variables meet certain conditions. For example, state events occur when a contact from a collision starts and ends or a structural change is triggered when dealing with variable structure systems. Event conditions are usually given implicitly, i.e., as zero-crossing functions. A state event occurs when an associated variable passes through zero. More than one zero-crossing function can be associated with a single event type. During the simulation, the zero-crossing functions must be checked continuously. For this purpose, many numerical ODE solvers provide a so-called root option. A vector contains the variables to be tested. These variables are constantly monitored during the simulation. If one of them passes through zero, an iteration is started to determine the zero-crossing time with a prespecified accuracy. Since it is not known when the event conditions will become true, the step size to hit them accurately cannot be reduced. Instead, some sort of iteration (or interpolation) mechanism is required to locate the event time. Thus, when an event condition is triggered during the execution of an integration step, it affects the step size control mechanism of the integration algorithm. Therefore, a variable-step solver is used, i.e., the step size of the solver is varying. The step size is smaller when an event occurs and larger when it does not. The continuous simulation is forced to iterate (or interpolate) to the earliest zero crossing within the current integration step, with e.g., Regula Falsi, or the Golden Section. One problem occurring with solvers is the so-called ghosting effect. That is, an object moves through another object during a time step (see Hofmann et al., 2014). One solution is to reduce the maximum step size of the solver in such a case.

State events are not considered in physics engines of computer games. The reason is, those engines are intended for real-time interactive simulations. Thus, they use fixed-step size solvers. This means that the step size is fixed during the simulation and events are not detected precisely. State events are vital for collision handling with variable-step solvers and variable structure systems.

3 Overview of Modia and Modia3D

An overview of the equation-based modeling language Modia (see Section 3.1), and the multibody module Modia3D (see Section 3.2) is provided. A general approach is applied that symbolically transformes and solves the occuring systems of linear and algebraic equations (see Section 3.3) when combining equation-based and multibody modeling.

3.1 Modia

Modia (Elmqvist, Henningsson, and Otter, 2017; Elmqvist, Otter, Neumayr, and Hippmann, 2021) is an open-source modeling and simulation environment for declarative, equation-based models to simulate physical systems described by DAEs. Based on Modelica, Modia is a new approach towards equation-based modeling with a very similar semantics to Modelica. Modia is a domain-specific extension of the Julia programming language (Bezanson et al., 2017). Modia utilizes the advantages of Julia's[1] powerful language features such as multiple dispatch, modern data structures with type inference, and a just-in-time-compiler. Benchmarks with Julia demonstrate a similar performance like C.

Contrary to other modeling languages, Modia models are defined directly with the Julia language, using some predefined helper functions to define the model's Abstract Syntax Tree (AST). So, no other language is needed to be parsed and converted to Julia code. Therefore, the user can benefit of other packages of the feature-rich Julia ecosystem. Modia consists of a set of Julia packages, most importantly of the packages Modia.jl (Elmqvist and Otter, online[b]) and Modia3D.jl (Neumayr, Otter, and Hippmann, online[b]). Modia offers several interfaces to Julia plot packages, like PyPlot.jl (JuliaPy, online) an interface to the Matplotlib (Hunter, 2007) plotting library from Python and several plot packages of Makie.jl (Danisch and Krumbiegel, 2021). Many more details on plotting and models are given in the Modia documentation (Elmqvist and Otter, online[a]).

Modia's very simple language semantics defines models with hierarchical collections of name/value pairs. This unified scheme is used for models, variables, equations, hierarchical modifiers, inheritance and replaceable components.

[1] Julia starts counting indices at 1 and brackets `[]` are used to get a specific entry of an array or dictionary (`firstEntry = myArray[1]`). The `::` operator is used to attach type annotations (`myVariable::Float = 42.0`). The `:` operator indicates a symbol (`:mySymbol`). The `@` operator indicates a macro (`@error`). Unicode characters (e.g.,δ) are supported.

This chapter is a concise introduction to the Modia language as needed to understand the code snippets used in this thesis. This shortened and modified overview of Modia is already published in Elmqvist, Otter, Neumayr, and Hippmann (2021, Section 2) and Neumayr and Otter (2023a, Appendix A). It is about how to define variables, models, connectors, and components and how to merge them and define connections.

3.1.1 Variables and Models

Variables `Var` are implicitly defined by their references in equations and can have attributes.

```
1 name = Var(attribute=value, ...)
```

`Var` is a function which takes name/value pairs and creates and returns a corresponding dictionary. The currently introduced real attributes are: `value`, `min`, `max`, `init`, and `start`. Boolean attributes are: `parameter`, `constant`, `input`, `output`, `potential` and `flow`. Parameters are defined with `Par`. It is a shortcut for `Var(parameter=true)`. Thus, the definitions of `T1` and `T2` are equivalent (Lines 3 to 4)[2].

```
2 # T1 and T2 are equivalent variable declarations
3 T1 = Var(parameter=true, value=0.2, min=0)
4 T2 = Par(value=0.2, min=0)
```

To create multiple expressions in Julia, without using the explicit constructor, and unlike the other means of quoting, `:( )` is used (Julia Documentation, online, see quote). Thus, the expressions must be enclosed in so-called quotes `:( )`, if the value contains references to other declared variables in the model. A parameter can also be defined by `name = literal-value`. Moreover, there are reserved names e.g., `time` for the independent variable with unit seconds.

A model is defined as a collection of name/value pairs with the constructor `Model` (Lines 5 to 12). The equations have Julia expressions on both the left and right sides of the equal sign. The entire set of `equations` is enclosed in `:[ ]`. Since an AST is built instead of evaluating the expressions, later processing is possible, e.g., solving the equations symbolically. The Modia-specific operator `der(v)` defines the time derivative $\dot{v} = \frac{dv}{dt}$ of variable v.

```
5 name = Model(
6     <variable-or-component-definition>,
7     ...,
8     equations = :[
9         <equation1>
10        <equation2>
```

[2] For better reference each code snippet is marked with a unique line number on the left-hand side.

```
11          ...]
12 )
```

Compare, e.g., a low pass filter defined as a Modia model (Lines 14 to 22), with the corresponding Modelica model (Lines 24 to 32). The Julia package Unitful.jl (Keller, online) defines units and manages unit inference and verification. The definition of units is done with `u"..."` suffix, compare with Line 15 and Line 25.

```
13 # Modia model
14 LowPassFilter = Model(
15     T = 0.2u"s",
16     u = Var(input=true),
17     y = Var(output=true),
18     x = Var(init=0.0),
19     equations = :[
20         T * der(x) + x = u
21         y = x]
22 )
```

```
23 // Modelica model
24 block LowPassFilter
25     parameter SIunits.Time T = 0.2;
26     input  Real u;
27     output Real y;
28     Real x(start=0.0, fixed=true);
29 equation
30     T * der(x) + x = u;
31     y = x;
32 end LowPassFilter;
```

3.1.2 Connectors and Components

Connectors are models that contain flow variables. A flow variable has the attribute `flow = true`. A potential variable has the attribute `potential = true`. For a specification of flow and potential variables see Chapter 2. Connectors must have the same number of flow and potential variables and the same array size. Connectors must not have any equations. Components are declared by using a model name as a value in a name/value pair. An electrical connector with potential `v` and current `i` is defined in Line 33.

```
33 Pin = Model(v = Var(potential=true), i = Var(flow=true))
```

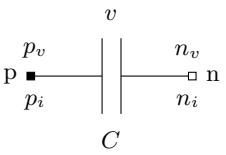


Figure 3.1: Equation-based model of a capacitor with parameter C, state v, connectors p, n with potential variables p_v, n_v (electrical potentials) and flow variables p_i, n_i (electric currents).

An electrical capacitor with two Pins `p` and `n` corresponding to Figure 3.1 is described in Lines 34 to 42. Furthermore, this capacitor is used to discuss a predefined acausal component needed for variable structure systems in Part III, (Section 10.1.4).

```
34 Capacitor = Model(
35     C = 0.1u"F",
36     p = Pin,
37     n = Pin,
38     equations = :[
39         0 = p.i + n.i
40         v = p.v - n.v
41         C*der(v) = p.i]
42 )
```

3.1.3 Merging

Models and variables are defined using hierarchical collections of name/value pairs. A constructor `Map` is used to set and modify parameters of components and attributes of variables, which are also structured in hierarchical collections. For example, the parameter T of the `LowPassFilter` model (Lines 14 to 22) is changed in Line 43.

```
43 lowPassFilter = LowPassFilter | Map(T = Map(value=2u"s", min=1u"s"))
```

The merge operator `|` is an overloaded binary operator of bitwise or with recursive merge semantics. The merging of `equations` is done particularly by concatenating the equation vectors.

3.1.4 Connections

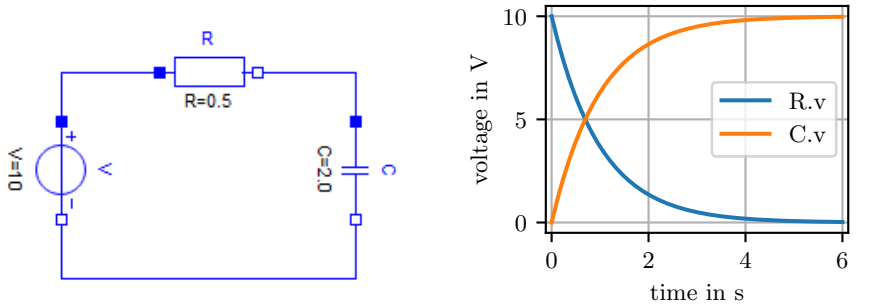
Connections are described as a special equation of the form:

```
44 connect = :[
45     (<connect-reference-1>, <connect-reference-2>),
46     (<connect-reference-3>, <connect-reference-4>),
47     ...]
```

A `connect-reference` is either a `component-instance-name` or a `connect-instance-name`. To be more precise, a `connect-instance-name` is either a connector instance, an input variable or an output variable. All corresponding potentials of the connected connectors are set equal. The sum of all incoming corresponding flows into the model is set equal to the sum of corresponding flows into the sub-components, i.e., the same semantics as in Modelica.

In Lines 48 to 62, a filter with electrical component models[3] is instantiated, parameters are set and connections are defined, as Figure 3.2a depicts. The filter model is instantiated, simulated, and the voltages of the resistor and of the capacitor are plotted (Figure 3.2b). The Julia macro `@instantiateModel` symbolically transforms the model, generates and compiles Julia code. Symbolic transformation is done with standard algorithms of object-oriented modeling languages and with

[3] Modia.jl, v0.12.0, examples/FilterCircuit.jl



a) Filter. b) Voltage of resistor and capacitor.

Figure 3.2: Electric circuit consisting of a constant voltage, a resistor, and a capacitor (Elmqvist and Otter, online[a]).

extensions as described by Otter and Elmqvist (2017). One simulation of the instantiated model is performed with function `simulate!` using Modia's default solver `CVODE_BDF`. This is one solver from the Julia package DifferentialEquations.jl (Rackauckas and Nie, 2017; SciML, online), which contains a large set of solvers. `CVODE_BDF` utilizes the Backward Differentiation Formula (BDF) solver of package CVODE from the Sundials suite (Hindmarsh et al., 2005; Hindmarsh, Serban, and Collier, 2015). Various keyword arguments for the simulation run can be defined, e.g., the stop time is set to 6 s. Parameters and initial values can be provided by a hierarchical `Map` which is merged with the current values via the `merge` keyword. The simulation results are stored within the instantiated model and are plotted with function call `plot` depending on the selected plot package. More details about this filter model are given in the Modia documentation (Elmqvist and Otter, online[a]).

```
48 using Modia
49 include("$(Modia.path)/models/Electric.jl") # include electrical components
50 Filter = Model(
51     R = Resistor | Map(R=0.5u"Ω"),,
52     C = Capacitor | Map(C=2.0u"F"),
53     V = ConstantVoltage | Map(V=10.0u"V"),
54     connect = :[
55         (V.p, R.p)
56         (R.n, C.p)
57         (C.n, V.n)]
58 )
59 filter = @instantiateModel(Filter)
60 simulate!(filter, stopTime=6.0u"s")
```

```
@usingModiaPlot   # use selected plot package
plot(filter, ("R.v", "C.v"))
```

Summarizing, the open-source modeling language Modia – the simulation environment used in this thesis – is a domain-specific extension of the Julia language. Modia benefits from the Julia ecosystem which is convenient for users too.

3.2 Modia3D

The open-source Julia package Modia3D.jl (Neumayr, Otter, and Hippmann, online[b], v0.12.0) is a multibody module for 3-dimensional (3D) mechanical systems. Modia3D belongs to the Modia modeling ecosystem. It is targeted for solvers with adaptive step size control to compute results close to real physics. Modia3D includes collision handling using the Minkowski Portal Refinement (MPR) algorithm and collision response for elastic contacts (Part II, Section 8.1). Moreover, Modia3D is designed as a predefined acausal component to support variable structure systems (Part III, Chapter 11). Modia3D is inspired by the generic component-based design pattern of modern game engines allowing modular and customizable definitions of 3D systems: A coordinate system located in 3D is used as a container, called Object3D, with optional components (geometry, solid and collision properties, visualization data, light, camera, etc.), see e.g., Nystrom (2014), Unity (Unity Technologies, online), Unreal Engine (Epic Games, online) three.js (Three.js, online).

Neumayr and Otter (2018) describe the modular and customizable component-based design pattern of Modia3D. To communicate with Modia, the user interface has changed in the meantime, so that constructors and functions take keyword arguments as inputs. The integration of Modia3D within Modia is described in Elmqvist, Otter, Neumayr, and Hippmann (2021) and Neumayr and Otter (2024). A Modia3D package documentation (Neumayr, Otter, and Hippmann, online[a]) and an installation guide (Neumayr, online) are publicly available on the web under MIT licenses.

3.2.1 Modia3D Components

The core component of Modia3D is an Object3D. It is the basis for modular and customizable 3D modeling. An Object3D is a coordinate system moving in 3D space with associated optional features, see Figure 3.3 and Table 3.1. An Object3D's position and orientation is defined relative to an optional parent Object3D by translation and rotation. An Object3D knows its parent and its optional children. An exception of this is the Object3D with feature `Scene`. It has no parent and is the root of all other Object3Ds in the multibody tree. For its uniqueness, only one Object3D is allowed to have a Scene. It is called `world` Object3D. This root defines a global inertial system. The feature `Visual` is for 3D animation only and defines

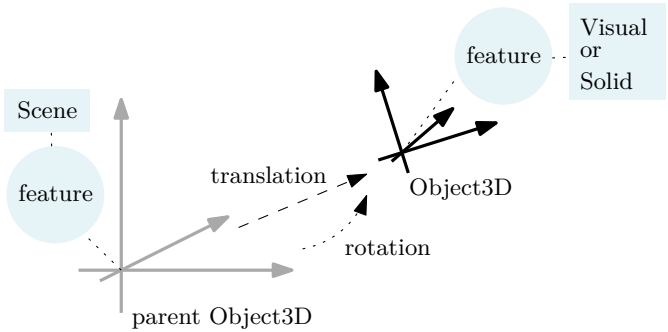
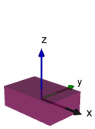
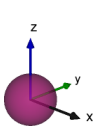
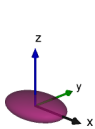
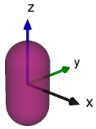
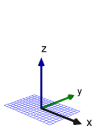
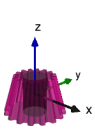
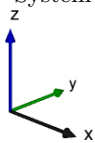
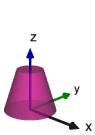
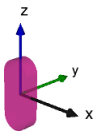
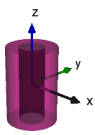
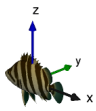
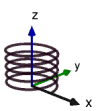


Figure 3.3: Object3D defined relative to its parent with translation and rotation. An Object3D can have exactly one optional feature: Scene, Visual, or Solid.

Table 3.1: Shapes supported by Solid and Visual features, and by Visual feature only.

Solid and Visual Shapes				Visual Shapes		
Box	Sphere	Ellipsoid	Capsule	Grid	GearWheel	Coordinate-System
						
Cone	Beam	Cylinder	FileMesh	Spring	TextShape	Modelica-Shape
						

shapes (see Definition 4.1) such as box, sphere, cylinder, beam, and 3D mesh with visualization properties. The feature `Solid` defines solid bodies. It has mass properties and can be considered in collision situations if keyword `collidable = true` is set. A Solid can have a shape and visualization properties.

For simplifying the user interface of Modia3D, the keyword `shape` is used for Solid and Visual features. Thus, there exist shapes which are used as Visual and Solid features, and shapes which are used as Visual feature only (Table 3.1). The geometric form (Definition 4.1) coincides with the shapes used for Solid feature. An Object3D is a collision object (Definition 4.4) if and only if (iff) it has a Solid feature with a shape, material properties and the keyword `collidable = true` is set. As an abbreviation Scene, Solid, or Visual stands for an Object3D with feature Scene, Solid, or Visual. The description of all possible keywords of `Object3D`, `Scene`, `Visual` and `Solid` can be found in the Modia3D documentation (Neumayr, Otter, and Hippmann, online[a]).

The design of Modia3D is modular, customizable and reusable, due to the modularity of feature combinations and of Object3Ds. Shapes are used as Solid feature with mass properties which can be used in collision situations or just for visualization purposes.

3.2.2 Modia Interface

This section outlines how the internal Modia3D structures and functions are interfaced to Modia models. The following concept is generic and not specific to Modia3D.

Multibody components are defined as very simple Modia components. These Modia components are Julia functions with keyword arguments. All Modia3D multibody components, except joints, are defined as Modia parameters `Par` (Line 4, Page 18). This means, all keyword arguments are treated as parameters. In Lines 63 to 65 a function `Object3D` with an arbitrary number of keyword arguments `kwargs...` is defined. These arbitrary keyword arguments, special predefined keywords[4] such as `_constructor = :(Modia3D.Composition.Object3D)`, and additional predefined keywords are passed on to `Par`. A dictionary containing all kinds of keywords is created. This can be regarded as a generalization of the concept of External Objects in Modelica. It is a generic Modia approach and not specific to multibody systems. All multibody components contain enough information to be used for variable structure systems.

```
63 Object3D(; kwargs...) = Par(;
64     _constructor = :(Modia3D.Composition.Object3D{FloatType}),
65     _path = true, kwargs...)
66 Scene(; kwargs...) = Par(;
67     _constructor = :(Modia3D.Composition.Scene{FloatType}), kwargs...)
68 Visual(; kwargs...) = Par(;
```

[4] All special predefined keywords in Modia and Modia3D start with `_`.

```
69     _constructor = :(Modia3D.Shapes.Visual), kwargs...)
70 Solid(; kwargs...)  = Par(;
71     _constructor = :(Modia3D.Shapes.Solid{FloatType}), kwargs...)
```

During code generation, Modia processes certain keywords of a parameter e.g., to access the parameter value in the generated Julia function. Some keywords, such as the Modia3D keywords, are ignored during code generation. Before simulation starts, all parameters are evaluated. For this purpose, the hierarchical dictionary of parameter definitions is recursively traversed to evaluate parameter expressions and propagated parameters.

For example, in the upcoming pendulum model Lines 98 to 102 (Page 27) the constructor of an `Object3D` is called. In a first step, `Object3D(feature = Solid())` is replaced by `Modia3D.Object3D(feature = Modia3D.Solid())`. In a second step, this constructor is executed and returns a reference to a Julia object that is associated with key `obj1`. After evaluating the parameters, the complete Modia3D data structure of this model is instantiated and is available in the dictionary of the evaluated parameters.

Modia3D is a predefined acausal component. This is essential to use Modia3D for variable structure systems. To understand how Modia3D is implemented, the main implementation concepts of predefined acausal components are already declared below. All mathematical details and definitions of predefined acausal components are explained in Chapter 10, as well as, a more detailed application. A predefined acausal component is a Modia `Model` with two special predefined keywords: `_buildFunction` and `_initSegmentFunction`.

- `_buildFunction`: Before symbolic transformation, the hierarchical dictionary of the model is run through. For each sub-model, the function defined by `_buildFunction` is executed once. First, this function defines additional model variables and equations which are merged with the corresponding model. Second, it returns an instance of the Julia structure, which acts as the internal memory of the component.
- `_initSegmentFunction`: This function is called by the simulation engine before the model is initialized and before the model is re-initialized for variable structure systems. In both cases, it is necessary to redefine all local variables of the predefined acausal component, including states, zero-crossing functions, and initial values for the newly defined states.

Only if these two keywords are set, it is a predefined acausal component. Otherwise, it is an ordinary Modia model.

When dealing with 3D models the top-level Modia model must be a `Model3D` (Lines 73 to 78). The special function `buildModel3D!` (Line 75) is called to add a few equations to the models depending on the used multibody components e.g., different joints used in the actual 3D model. It recursively traverses the model, i.e., a hierarchical dictionary, and collects all information about the joints used.

Based on this information, `initSegmentModel3D!` (Line 77) builds, among others, the internal tree structure of Modia3D with the so-called super-objects.

```
72 # predefined acausal component
73 Model3D(; kwargs...) = Model(;
74     # called once before symbolic transformation
75     _buildFunction = :(Modia3D.buildModel3D!),
76     # called before each new simulation segment
77     _initSegmentFunction = Par(functionName = :(Modia3D.initSegmentModel3D!)),
78         kwargs...)
```

3.2.3 Joints with Invariant Variables

Modia3D offers two types of joints:

1. Joints with Invariant Variables that cannot be changed during simulation. This joint type is discussed in the upcoming section.
2. Joints with Variant Variables that can be changed during simulation. This joint type is discussed in Section 11.1.

The first type of joints contains Modia equation sections with invariant variables, including invariant states. These joints are visible for Modia and cannot be removed or added during simulation. The interface to the Modia3D functionality is designed to define differential equations only on the Modia side in Modia equation sections.

For example, the multibody component `RevoluteWithFlange` in Lines 79 to 92 is defined as a Modia `Model`. It is a revolute joint with a Modia 1-dimensional (1D) rotational `flange`. Additionally, it consists of parameters `obj1, obj2, axis` and local variables `phi, w` that are initialized with zero. To realize variable structure systems, the two equations `phi = flange.phi` and `w = der(phi)` are the acausal part of a revolute joint. The causal part is defined with parameter `_constructor` together with all parameters defined with keyword `Par`. For further information to causal and acausal equations see Chapter 10.

```
79 Flange = Model(phi=Var(potential=true), tau=Var(flow=true))

80 RevoluteWithFlange(; obj1, obj2, axis=3, phi=Var(init=0.0), w=Var(init=0.0)) =
81     Model(;
82         _constructor = Par(value=:(Modia3D.Revolute),
83                            _jointType=:RevoluteWithFlange),
84         obj1   = Par(value = obj1),
85         obj2   = Par(value = obj2),
86         axis   = Par(value = axis),
87         flange = Flange,
88         phi    = phi,
89         w      = w,
90         equations = :[
91             phi = flange.phi
92             w   = der(phi)])
```

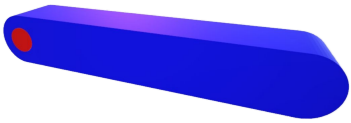


Figure 3.4: Pendulum consists of a solid beam. A red cylinder visualizes the revolute joint.

The model of a simple pendulum[5] (Lines 93 to 112 and Figure 3.4) with damping in its joint uses Modia3D components, especially Object3Ds with its different features and combines equation-based components of Modia. The remaining elements of the pendulum use predefined equation-based models of a small Modia library which corresponds to the Modelica.Mechanics.Rotational library. To model the damping in the joint, a rotational 1D damper is connected to a fixed point and the flange of a revolute joint (Lines 79 to 92).

```
93 using Modia3D
94 include("$(Modia3D.modelsPath)/AllModels.jl") # Modia rotational library
95 Pendulum = Model3D(
96     # multibody components
97     world = Object3D(feature=Scene()),
98     obj1  = Object3D(feature=Solid(solidMaterial="Steel", collidable=true,
99                 shape=Beam(length=1.0, width=0.2, thickness=0.2, ...))),
100    obj2  = Object3D(parent=:obj1, translation=[-0.5, 0, 0],
101                 feature=Visual(shape=Cylinder(diameter=0.1, length=0.21),
102                 visualMaterial = VisualMaterial(color="Red"))),
103    rev   = RevoluteWithFlange(obj1=:world, obj2=:obj2),
104    # equation-based components
105    damper = Damper | Map(d=100.0),
106    fixed  = Fixed,
107    connect = :[
108        (damper.flange_b, rev.flange),
109        (damper.flange_a, fixed.flange)]
110 )
111 pendulum = @instantiateModel(Pendulum, ...)
112 simulate!(pendulum, stopTime=3.0)
```

During the parameter evaluation, a special action is taken for parameters with key `_constructor`: A constructor call is assembled from the constructor name and any defined parameters. For example, the `RevoluteWithFlange` definition of Lines 79 to 92 results in the constructor call of Line 113. This constructor is called on the fly resulting in an instance of Julia struct `Revolute`. The call returns a reference `ref` to the created instance. A statement such as `rev = RevoluteWithFlange()` in Line 103 is a key/value pair. The key is `rev` and the value is an instance of a `Model` dictionary. This value is replaced by an instance of a parameter dictionary. So, the

[5] Modia3D.jl, v0.12.0, test/Tutorial/Pendulum3.jl

generated instance of the revolute joint is stored as a parameter. The evaluated parameters are displayed with e.g.,`simulate!(logEvaluatedParameters = true)`.

```
113 rev = Modia3D.Joints.Revolute(obj1, obj2, axis=3)
```

The keys of other instances are referenced in the argument list, e.g.,`RevoluteWithFlange(obj1 = :world)`. During parameter evaluation, symbols such as `:world` are searched for on the left side of the equal signs. They are then replaced by the corresponding value of this keyword. For example, `:world` is replaced by the constructor call `Modia3D.Object3D(feature = Modia3D.Scene())`. Once all parameters are evaluated, all keyword arguments of multibody components contain a reference to the instantiated Julia objects.

The overall model is traversed when the model is instantiated. When traversing, each predefined acausal component can inject equations into the model definition which are used in symbolic transformation. Alias variables are eliminated. The set of all equations is generated, as sketched in (Lines 116 to 126) for the pendulum model (Lines 93 to 112).

Function `openModel3D!` creates an instance of the multibody system. It contains, e.g., the generated instance of the revolute joint. The instantiated top-level `model` is passed as an argument. So, the function `openModel3D!` has access to the complete model definition. Function `setStatesRevolute!` stores the current values of the angles and angular velocities of all revolute joints of the multibody model. These variables are states in the Modia equations, due to their definition in Lines 79 to 92. Function `setAccelerationsRevolute!` stores the angular accelerations of all revolute joints in the multibody model. Further function calls basically construct the multibody equations (2.5) in residual form. So, $\boldsymbol{f}_{\text{gen}} = \boldsymbol{M}(\boldsymbol{q}, t)\dot{\boldsymbol{v}} + \boldsymbol{h}(\boldsymbol{q}, \boldsymbol{v}, t) - \boldsymbol{\tau}$. The set of equations is transformed symbolically, i.e., equations are differentiated, sorted and simplified. The result is the Julia `getDerivatives` function. It is compiled into code that is called by the `simulate!` function.

```
114 # _x    ... states vector from solver
115 # model ... instantiated simulation model
116 function getDerivatives(_x, model, time)
117     ...
118     rev.phi = _x[1]
119     rev.w   = _x[2]
120     # Equations injected with buildModel3D!
121     _mbs1 = Modia3D.openModel3D!(model, _x, time)
122     _mbs2 = Modia3D.setStatesRevolute!(_mbs1, rev.phi, rev.w)
123     _mbs3 = Modia3D.setAccelerationsRevolute!(_mbs2, der(rev.w))
124     _genForces = Modia3D.computeGeneralizedForces(_mbs3)
125     ...
126 end
```

To recap, the first type of joints with invariant variables containing Modia equations are introduced once. They cannot be added or removed during simulation.

The second type of joints with variant variables can be added or removed during simulation. This type of joints is a crucial invention to support variable structure systems, see Section 11.1.

3.2.4 Internal Tree Structure: Super-Objects

The objects of 3D models need to be handled efficiently during simulation. Therefore, a preprocessing step for building up Modia3D's internal execution scheme (internal tree structure, data structure) based on the Modia3D's model definitions is introduced. All information about multibody system components (e.g., Object3Ds, joints, solids, ...) and their functionality (e.g., collision properties) is sorted and mapped to an internal tree structure. This resulting multibody tree is a directed acyclic graph with a unique root. It can be efficiently evaluated during simulation starting from its unique root and uses the parent-child relationship. Moreover, this execution scheme includes definitions of the states of the multibody systems and of their initial values which are deduced from the used joints. Object3Ds are grouped into so-called super-objects (Neumayr and Otter, 2019a). Super-objects have the following characteristics:

- Super-objects are disjunct via joints.
- A super-object consists of rigidly connected Object3Ds.
- The root of a super-object is an Object3D which is freely moving or which has a joint.
- The super-object's root is the parent of all other Object3Ds in a super-object.

To fulfill the above characteristics, all Object3Ds belonging to a super-object are rigidly re-connected to its super-object's root. Based on the features of Object3Ds in super-objects, different actions are performed: For example, all Object3Ds in the same super-object cannot collide with each other, but they can collide with all other Object3Ds that are enabled for collision handling. A common mass, common inertia tensor, and common center of mass are computed for a super-object considering the mass properties of all Object3Ds inside this super-object. Two corresponding examples of super-objects are given in Figures 3.5 and 3.6.

The algorithm for grouping the Object3Ds is based on the Depth-First Search (DFS) by Tarjan (1972) and Hopcroft and Tarjan (1974). The DFS is extended to the augmented DFS by Neumayr and Otter (2019a) leading to the internal data structure with super-objects. The super-objects themselves are sorted in DFS order. All rigidly connected Object3Ds belonging to a super-object are also sorted in DFS order. Cut-joint and kinematic loops are not supported yet. Without any further assumptions, the grouping of the 12 Object3Ds of Figure 3.6, leads to five general super-objects in Figure 3.5. A pseudo code for grouping all Object3Ds of the multibody tree into super-objects is given in Lines 127 to 151. This algorithm needs two different arrays.

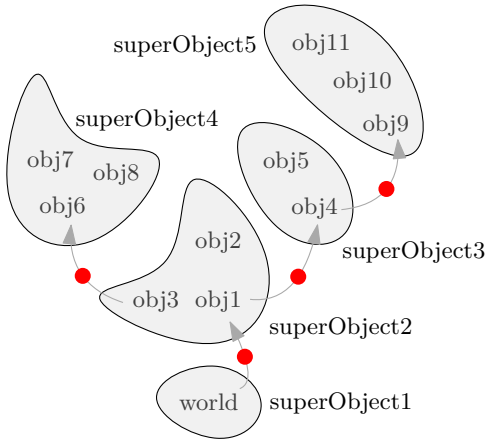


Figure 3.5: Five general super-objects. The unique root of the multibody tree is world. The roots of the super-objects are world, obj1, obj4, obj6, obj9.

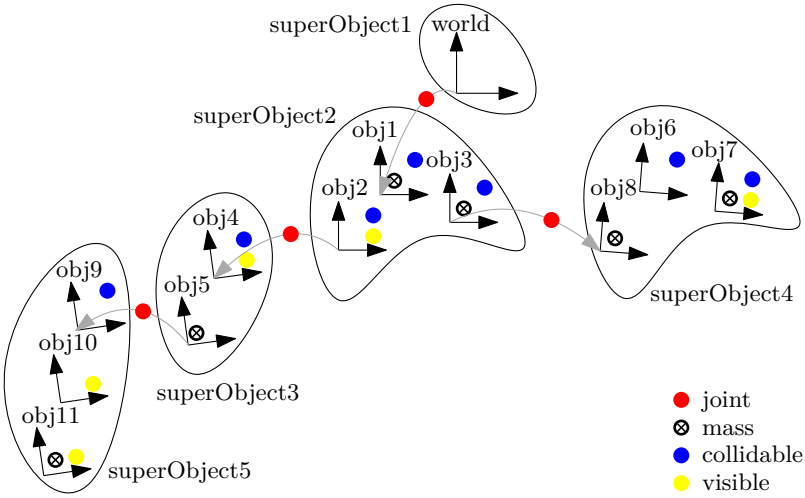


Figure 3.6: Internal execution scheme at initialization. 12 Object3Ds with different properties are defined: They are allowed to collide, can have a mass, and are visible. They are grouped into five super-objects, which are disjunct via joints.

- `actRootChildren`: Starting from the root of the actual super-object all rigidly attached children and rigidly attached children's children and so on, are traversed in DFS order. They are stored in `actRootChildren` and are rigidly re-attached to the actual super-object's root. If free moving children or children with joints are encountered they are stored in `roots`.
- `roots`: Free moving Object3Ds and Object3Ds with joints are roots of super-objects.

```
function augmentedDFS!(world::Object3D)
    push!(roots, world)
    actPos = 1, nPos = 1
    while actPos <= nPos
        superObj = SuperObjs() # actual super-object
        root = roots[actPos]   # actual root
        if root != world
            # group root according to its features
            group(superObj, root)
        end
        # store root's children in actRootChildren or roots
        # re-attach root's children to if stored in actRootChildren
        rootsOrActRootChildren!(superObj, root, root)
        while length(actRootChildren) > 0
            child = pop!(actRootChildren)
            # group child according to its features
            group(superObj, child)
            # store child's children in actRootChildren or roots
            # re-attach child's children to root if stored in actRootChildren
            rootsOrActRootChildren!(superObj, root, child)
        end
        nPos = length(roots)
        actPos += 1
        ...
end; end
```

To conclude, grouping Object3Ds into super-objects is a preprocessing step. Super-objects are used for collision handling, e.g., to speed up the broad phase. For more details see Section 8.1. This internal tree structure is extended in Chapter 11 to support variable structure systems.

3.2.5 Animation

Modia3D provides a generic interface to visualize simulation results with 3D renderers. This supports the user in interpreting the simulation results through visualization, besides plotting the results. For further details on animation see Neumayr, Otter, and Hippmann (online[a]).

Both, the free Community and the Professional edition of the DLR visualization library (Kümper, Hellerer, and Bellmann, 2021) are supported, allowing rendering during simulation and the creation of videos in various formats.

Another option is to automatically generate a JSON Object Scene format 4 file when the simulation finishes (Three.js, online). This file can be imported into the three.js editor, which allows adaptable inspection of the animation and provides several options for rendering the scene with different cameras and lighting options. In addition, the animation can be exported to the standard glTF file format or its binary glb version, for which many viewers are available. The initial configuration can also be exported in obj, ply or stl format.

Furthermore, an interesting feature of Microsoft Office 2019 (e.g., Word or PowerPoint) is the importing and rendering of these file formats. While only a static display is possible with Office 2019, the current Office 365 subscription also supports the playback of the animation sequence.

3.3 Systems of Algebraic Equations

The equation-based modeling language Modia symbolically transforms DAEs (2.1) to ODEs (2.2) with symbolic transformation algorithms. These algorithms are already published by Otter and Elmqvist (2017), Elmqvist, Otter, Neumayr, and Hippmann (2021) and Neumayr and Otter (2024). To start with, a general approach that solves systems of linear equations together with systems of algebraic equations is outlined. Then, this general approach is applied to a specific application to demonstrate details. This application combines equation-based modeling and multibody modeling.

After symbolic transformations a Julia function called `getDerivatives` is generated and compiled. This function calculates the derivatives $\dot{\boldsymbol{x}}$. The `getDerivatives` function is dependent from its application, see Lines 116 to 126 (Page 28), Lines 203 to 263 (Page 36), and Lines 576 to 586 (Page 113).

Physical models often lead to linear equation systems. Modia generates highly efficient code to solve them numerically during execution of the model. Assume, structural analysis identifies a nonlinear equation system

$$\mathbf{0} = \boldsymbol{g}(\boldsymbol{w}, \boldsymbol{u}), \tag{3.1}$$

with unknown local variables $\boldsymbol{w}$ and known variables $\boldsymbol{u}$ that are computed prior to this statement. This equation system can be transformed to

$$\boldsymbol{w}_1 = \boldsymbol{g}_1(\boldsymbol{w}_{\text{eq}}, \boldsymbol{u}) \tag{3.2}$$

$$\mathbf{0} = \boldsymbol{g}_{\text{eq}}(\boldsymbol{w}_1, \boldsymbol{w}_{\text{eq}}, \boldsymbol{u}), \tag{3.3}$$

with the tearing algorithm of Otter and Elmqvist (2017). In (3.2), (3.3) the unknowns and equations are split into an explicitly evaluable part $\boldsymbol{w}_1$, and a system of algebraic equations with unknowns $\boldsymbol{w}_{\text{eq}}$. If $\boldsymbol{g}$ is linear in the unknowns $\boldsymbol{w}$, equation (3.3) can be (conceptually) rearranged into a linear equation system

$$0 = \boldsymbol{A}(\boldsymbol{w}_1, \boldsymbol{u})\boldsymbol{w}_{\text{eq}} - \boldsymbol{b}(\boldsymbol{w}_1, \boldsymbol{u}). \tag{3.4}$$

In (3.4), $\boldsymbol{A}, \boldsymbol{b}$ are functions of the explicitly solved variables $\boldsymbol{w}_1$ and the known variables $\boldsymbol{u}$. The equation has to be solved for variables $\boldsymbol{w}_{\text{eq}}$. In the worst case, $\boldsymbol{A}$ has n^2 elements ($n = \dim(\boldsymbol{w}_{\text{eq}})$). Therefore, the rearranged code size is $O(n^2)$. If the number of iteration variables n grows, the code size increases quadratically.

```
152 # Initialize memory m (m.w_eq = 0, ...)
153 while true
154     w_eq = m.w_eq
155     w_1 = g_1(w_eq, u)
156     m.r = g_eq(w_1, w_eq, u)
157     if lEqIteration(m); break; end
158 end
```

Instead, another approach with code size $O(n)$ is to generate the code in Lines 152 to 158 and Lines 159 to 186. Together they construct and solve the linear equation system (3.4). Residuals $\boldsymbol{r}$ are computed and stored in the memory m. The linear equation system is solved to compute $\boldsymbol{w}_{\text{eq}}$ and $\boldsymbol{w}_1$ from this solution.

```
159 function lEqIteration(m)
160     n = length(m.w_eq)
161     if m.mode == QUIT
162         return true
163     elseif m.mode == COMPUTE_B
164         # compute b with w_eq = 0
165         # r = A*0 - b => b = -r
166         m.b = -copy(m.r)
167         m.j = 1
168         m.w_eq = e_1
169         m.mode = COMPUTE_A
170     else # m.mode == COMPUTE_A
171         # compute column j of A with w_eq = e_j
172         # r = A*e_j - b => A[:,j]
173         m.A[:,j] = m.r + m.b
174         if m.j != n
175             m.j += 1     # j+1
176             m.w_eq = e_j # j+1-th unit vector
177         else
178             # solve linear equation system
179             # A*w_eq = b
180             m.w_eq = m.A \ m.b
181             m.mode = QUIT
182         end
183     end
184     m.r = zeros(n)
185     return false
186 end
```

The function `lEqIteration` in Lines 159 to 186 is called in a while loop from Lines 152 to 158. It iteratively computes vector $\boldsymbol{b}$, matrix $\boldsymbol{A}$, and finally $\boldsymbol{w}_{\text{eq}}$, depending on the actual mode (`COMPUTE_B, COMPUTE_A, QUIT`). All vectors $\boldsymbol{b}, \boldsymbol{r}, \boldsymbol{w}_{\text{eq}}$,

matrix $\boldsymbol{A}$, column counter j, and the actual mode are stored in a memory m, and are updated when needed. To compute vector $\boldsymbol{b}$, the first mode is `COMPUTE_B`. The residuals $\boldsymbol{r}$ are computed with $\boldsymbol{w}_{\text{eq}} = \boldsymbol{0}$. This allows to set $\boldsymbol{b} = -\boldsymbol{r}$. To compute matrix $\boldsymbol{A}$, the next mode is `COMPUTE_A`. To iteratively calculate the columns of $\boldsymbol{A}$, the residuals are computed with $\boldsymbol{w}_{\text{eq}} = \boldsymbol{e}_j$ that is the j-th unit vector from $j = 1, \ldots, n$. When the n-th column of $\boldsymbol{A}$ is computed, so $\boldsymbol{A}$ is known, the linear equation system is solved for $\boldsymbol{w}_{\text{eq}}$. One final iteration of the while loop is needed to evaluate $\boldsymbol{w}_1$.

To keep the description of `lEqIteration` in Lines 159 to 186 simple, two special cases are not shown. Symbolic transformations analyze if $\boldsymbol{A}$ is a function of parameters $\boldsymbol{p}$, it remains unchanged after initialization. At initialization, the LU-decomposition of $\boldsymbol{A}$ is computed once and stored in a memory m. During simulation, only an inexpensive backwards solution is applied to calculate the solution. If the residual equation's size is one, a simple division is performed. A linear equation solver is not required.

Modia utilizes the linear equation solver from the RecursiveFactorization.jl (JuliaLinearAlgebra, online) package. This package employs a recursive left-looking LU-algorithm for dimensions up to $n = 500$ by Toledo (1997). Performance tests demonstrate a beneficial speed-up compared to the linear standard solver from OpenBLAS (OpenBLAS, online) which is otherwise used. The DifferentialEquations.jl (Rackauckas and Nie, 2017; SciML, online) package's ODE and DAE solvers are employed to handle the generated `getDerivatives` function. The `getDerivatives` function is automatically called as required by the interface of the chosen solver.

One effective technique for DAE solvers significantly increases the simulation speed. It is applicable when both the linear equation system's size n exceeds a certain limit ($n \geq 50$) and the unknowns $\boldsymbol{w}_{\text{eq}}$ are a subset of the DAE states' derivatives. The relevant DAE state derivatives are used as solutions $\boldsymbol{w}_{\text{eq}}$ of the linear equation system. The residuals $\boldsymbol{r}$ are used for the DAE solver. For each model evaluation, the linear equation system's residuals are calculated only once eliminating the need of solving a linear equation system. At events (including initialization), the linear equation system is constructed and solved, providing consistent initial conditions for the DAE solver.

An application of a one-arm robot[6] with a drive train in Lines 187 to 202 demonstrates the proposed approach. Symbolic transformation of the model yields the `getDerivatives` function in Lines 203 to 263. The one-arm robot model comprising a motor, ideal gear, and a cascaded P-PI controller, actuates the flange of a revolute joint. The corresponding Modelica object-diagram is illustrated in Figure 3.7. Individual instances `world, body, rev` of multibody components are specified and integrated with `ramp, ppi, wSensor, motorInertia, gear` of equation-based Modia components.

```
187 Servo = Model3D(
188     world = Object3D(feature=Scene()),
```

[6] Modia3D.jl, v0.12.2, test/Robot/ServoWithRampAndRevolute.jl

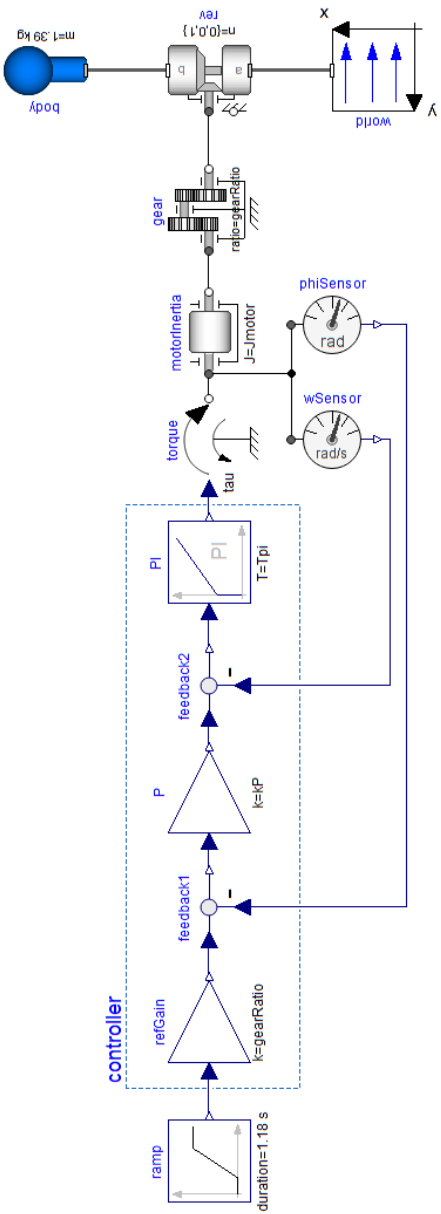


Figure 3.7: A single revolute joint of a manipulator rotates around the z-axis and is driven by a servo motor via an ideal gear. The revolute angle is controlled by a cascaded P-PI controller, that tracks the reference ramp.

```
189     body  = Object3D(feature=Solid(...)),
190     rev   = RevoluteWithFlange(obj1=:world, obj2=:body, axis=3,
191         phi=Var(init=0.0), w=Var(init=0.0)),
192     ramp         = Ramp,
193     ppi          = Controller,
194     wSensor      = UnitlessSpeedSensor,
195     motorInertia = Inertia,
196     gear         = IdealGear,
197     connect = :[
198         (ramp.y, ppi.refGain)
199         (gear.flangeB, rev.flange)
200         ...])
201 servo = @instantiateModel(Servo)
202 simulate!(servo, stopTime=...)
```

All relevant parameters are queried in Line 206 of the generated `getDerivatives` function. The states $\_x$ supplied by the solver are assigned to their respective model variables in Lines 208 to 210. Subsequently, all explicitly solved equations are present in Lines 213 to 226[7]. An algebraic loop is present since the motor inertia is connected via an idealized gear model to the revolute joint. This algebraic loop is between equations of the Modia components `motorInertia` and `gear` and the Modia3D components `world, body`, and `rev`. To solve this algebraic loop within the sorted equations, a new memory m is allocated in Line 236. The stored data is initialized with zero values. Residuals are iteratively calculated in a while loop (Lines 237 to 257), solving multibody equations (2.6), and the equations of components `motorInertia, gear` with `lEqIteration` to determine iteration variable $\boldsymbol{w}_{\text{eq}}$.

```
203 # _x    ... states vector from solver
204 # model ... instantiated simulation model
205 function getDerivatives(_x, model, time)
206     < get parameters: startTime, duration, kRefGain, gearRatio, ...>
207     # states
208     rev.phi  = _x[1]
209     rev.w    = _x[2]
210     ppi.PI.x = _x[3]
211     # explicitly solved equations
212     # f1 from eq (2.6)
213     der(rev.phi) = rev.w
214     ppi.refGain.u = ramp(time, startTime, duration)
215     ppi.refGain.y = kRefGain * ppi.refGain.u
216     motorInertia.phi = gearRatio * rev.phi
217     wSensor.flange.phi = motorInertia.phi
218     ppi.P.u = ppi.refGain.y - wSensor.flange.phi
219     ppi.P.y = kP * ppi.P.u
220     der(motorInertia.phi) = gearRatio * der(rev.phi)
```

[7] Julia allows variable names such as `der(rev.phi)`. So, the Modia variable names can be directly used as variable names in the generated function.

```
221     der(wSensor.flange.phi) = der(motorInertia.phi)
222     wSensor.w = der(wSensor.flange.phi)
223     ppi.PI.u = ppi.P.y - wSensor.w
224     der(ppi.PI.x) = ppi.PI.u / Tpi
225     motorInertia.flangeA.tau = kpi * (ppi.PI.x + ppi.PI.u)
226     motorInertia.w = der(motorInertia.phi)
227     # open 3D model
228     _mbs1 = openModel3D!(model, _x, time)
229     # set states in revolute joints
230     _mbs2 = setStatesRevolute!(_mbs1, rev.phi, rev.w)
231     begin
232     # new memory m: m.A=zeros(1,1),
233     # m.b=zeros(1), m.w_eq=zeros(1),
234     # m.r=zeros(1), m.j=0
235     # m.mode = COMPUTE_B
236     m = initlEqIteration(model)
237     while true
238         # explicitly solved equations
239         der(rev.w) = m.w_eq[1]
240         der(der(rev.phi)) = der(rev.w)
241         der(der(motorInertia.phi)) = gearRatio * der(der(rev.phi))
242         der(motorInertia.w) = der(der(motorInertia.phi))
243         motorInertia.a = der(motorInertia.w)
244         gear.flangeA.tau = -Jmotor * motorInertia.a + motorInertia.flangeA.tau
245         gear.flangeB.tau = -gearRatio * gear.flangeA.tau
246         # set acceleration in joints
247         _mbs3 = setAccelerationsRevolute!(_mbs2, der(rev.w))
248         # f2 from eq (2.6): compute generalized
249         # forces in joints from position,
250         # velocity, acceleration, collisions
251         _genForces = computeGeneralizedForces(_mbs3)
252         # compute residual vector
253         if m.mode != QUIT
254             m.r[1] = _genForces[1] + gear.flangeB.tau
255         end
256         if lEqIteration(m); break; end
257     end
258     # report derivatives to solver
259     model.der_x[1] = der(rev.phi)
260     model.der_x[2] = der(rev.w)
261     model.der_x[3] = der(ppi.PI.x)
262     return nothing
263 end
```

In summary, the most fundamental concepts for equation-based modeling with the modeling language Modia, the multibody package Modia3D, and how both are symbolically transformed have been introduced. Especially, the Modia3D package is a suitable testbed for collision handling with variable-step solvers (see Part II) and variable structure systems (see Part III).

Part II

Collision Handling with Variable-Step Solvers

4 Basic Concepts for Collision Handling

Collision handling is widespread, especially in the field of game physics and virtual reality. In computer games, it is important that collisions of many objects are detected in real-time and that the resulting simulation appears reasonably realistic. Therefore, computer games use fixed-step size solvers that do not detect events precisely.

When simulating real-world problems, in contrast, it is important that collision models reflect the actual physical behavior. The collision handling approach discussed in this thesis is tailored to variable-step solvers to determine the exact start and end time of a collision to obtain an accurate physical simulation. Furthermore, it is not known in advance which objects could potentially collide with each other. It is only known, that an object is a collision object and may collide. A standard collision handling approach has three steps: broad phase, narrow phase, and response calculation. An additional once-off preprocessing step that analyzes the mechanical structure is invented to reduce the number of potential collision pairs in the upcoming three steps. The algorithms and approaches of each collision handling step are mindfully chosen. They are adapted, so that they synthesize with each other and are applicable with variable-step solvers. The broad phase is a simple selection procedure. All collision objects are approximated by simple bounding volumes, e.g., Axis-Aligned Bounding Boxes (AABBs). Only, if those bounding volumes overlap, the narrow phase is executed. The narrow phase is a computationally expensive intersection test. In this thesis the Minkowski Portal Refinement (MPR) algorithm is chosen to compute a (unique) contact point on each object and a penetration depth (when both objects are penetrating). This algorithm is enhanced to compute the Euclidean distance (when both objects are separated). The novel concept of the *signed distance* (Euclidean distance or penetration depth) is used as zero-crossing function and is crucial for collision handling with variable-step solvers. The MPR algorithm is based on the Minkowski Difference and on support mappings. The Minkowski Difference transforms a shape-to-shape distance problem into a shape-to-point distance problem. Support mappings are implicit representations of geometries. They are needed to find the farthest vertex in a search direction. Moreover, support mappings are also applied to create the bounding volumes (AABBs) in the broad phase. To avoid unphysical behavior in the response calculation, several restrictions are introduced to ensure a continuous penetration depth. A novel force and torque formulation is developed by

combining and enhancing existing response formulations. In this thesis, the collision response is calculated using elastic material laws, based on the penetration depth, the contact points of the two penetrating objects, material data, and geometries.

Some basic concepts needed for the briefly introduced collision handling approach, especially in the broad and narrow phase with the Minkowski Portal Refinement (MPR) algorithm are discussed in the upcoming chapter.

The term geometric figure or shape is used inconsistently and often remains undefined in the literature. Kendall (1984) says a shape is "what is left when the effects of location, size, and rotation have been filtered out". Bergen (2003) gives an illustrative taxonomy of primitive shapes. Based on that, the following definitions are used in this thesis.

Definition 4.1: (Geometric form, shape).
A geometric form or a shape is pure geometric information with a fixed scale. Information about translation, rotation and material properties are filtered out.

Definition 4.2: (Object).
An object consists of a time invariant shape with time-dependent and continuous translation and rotation. In addition, it has solid material properties e.g., mass, Young's modulus, Poisson's ratio.

Remark 4.1: In other words, a time invariant shape is not deformable. Furthermore, the first part of Definition 4.2 corresponds to the definition given by Bergen (2003): "An object is a closed bounded nonempty set of points in Euclidean space $\mathbb{R}^3$." For collision handling step 2 and step 3, the broad and narrow phase, it is sufficient that an object consists of a shape with translation and rotation. Furthermore, for collision handling step 4, the response calculation, solid material properties are needed. Since in this thesis an object has solid material properties, it is also referred to as Solid as in the literature.

The set of objects (Definition 4.2) is a compound of the set of (convex) polytopes, where a 2-dimensional (2D) polytope is referred to as polygon and a 3D one as polyhedron. A (convex) polytope $S \subset \mathbb{R}^n$ is defined as the convex hull $S = \text{conv}(P)$ of a finite subset $P = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k\} \subset \mathbb{R}^n$ (Grünbaum, 2013). A polytope is bounded and closed, and thus compact (Brondsted, 2012). A simplex is the convex hull of an affinely independent set of points i.e., simplices of one, two, three and four vertices are points, line segments, triangles, and tetrahedrons, respectively (Bergen, 2003). The other part of convex object is the set of convex quadratics, like spheres, cones and cylinders in $\mathbb{R}^3$. Those convex quadratics are again bounded and closed.

An object can change over time. While in general, this might include deformation of objects, this thesis is restricted to rigid motion. This means translations and rotations are allowed but no deformations, reflections, or uniform/nonuniform scalings.

Definition 4.3: (Collision Set).
Let the collision set $\mathcal{C}$ be the set of all potentially-colliding convex or concave time-dependent objects in a collision environment. Let $n_{\mathcal{C}}$ be the cardinality of the collision set $\mathcal{C}$, in other words it corresponds to the number of potentially-colliding objects in this set.

Definition 4.4: (Collision Object).
An object $A \in \mathcal{C}$ is called collision object. For time $t \in \mathbb{R}_0$, the translation and rotation of $A(t)$ changes continuously due to rigid motion.

Definition 4.5: (Collision Pair).
Two objects $A, B \in \mathcal{C}, A \neq B$ are a potentially-colliding pair or a collision pair.

4.1 Minkowski Difference

The collision detection algorithm is based on a mathematical concept called the Minkowski Difference. To begin with, the Minkowski Sum is introduced.

Definition 4.6: (Minkowski Sum). (e.g., Bergen, 2003)
The Minkowski Sum $A \oplus B$ of two sets A and B is defined as the set of

$$A \oplus B = \{\boldsymbol{a} + \boldsymbol{b} : \ \boldsymbol{a} \in A, \ \boldsymbol{b} \in B\}. \tag{4.1}$$

The set of Minkowski Sum is the addition of all points in A added to all points in B.

However, for distance computation, the focus is on subtracting two sets. The subtraction of the Minkowski Sum is referred to as the Minkowski Difference.

Definition 4.7: (Minkowski Difference). (e.g., Bergen, 2003)
The Minkowski Difference $A \ominus B$ of two sets A and B is defined as the set of

$$A \ominus B = \{\boldsymbol{a} - \boldsymbol{b} : \ \boldsymbol{a} \in A, \ \boldsymbol{b} \in B\}. \tag{4.2}$$

This is the set of all vectors from a point of B to a point of A.

Remark 4.2: Both sets must be defined relative to the same coordinate system. Therefore, both, $\boldsymbol{a}$ and $\boldsymbol{b}$ are absolute translation vectors of points in A and B, respectively (Bergen, 2003).

4.1.1 Properties of the Minkowski Difference

Some important properties of the Minkowski Difference and the Minkowski Sum, which are also applicable to the Minkowski Difference, are considered below. The proofs of the stated theorems are given in Bergen (2003).

Theorem 4.1: *(Bergen, 2003, Theorem 2.1) The Minkowski Sum of two convex sets A and B is convex.*

Theorem 4.2: *(Bergen, 2003, Theorem 2.2) Let A and B be two polytopes. Then, the Minkowski Sum of these two polytopes A and B is a convex polytope*

$$A \oplus B = \operatorname{conv}\{\boldsymbol{a} + \boldsymbol{b} : \boldsymbol{a} \in \operatorname{vertex}(A),\ \boldsymbol{b} \in \operatorname{vertex}(B)\}.$$

Where $\operatorname{vertex}(A) \subseteq A$ *is the set of all vertices of A, the same holds true for* $\operatorname{vertex}(B) \subseteq B$*. This means $A \oplus B$ is the convex hull of all combinations of both polytopes.*

Remark 4.3: The statements of Theorems 4.1 and 4.2 can also be made for the Minkowski Difference (Bergen, 2003).

Theorem 4.3: *(Bergen, 2003, p. 36) For any pair of convex objects there exists a unique point in $A \ominus B$ that is closest to the origin.*

Theorem 4.4: *(Bergen, 2003, p. 36) Two objects intersect iff Minkowski Difference contains the origin*

$$A \cap B \neq \emptyset \equiv \mathbf{0} \in A \ominus B. \tag{4.3}$$

Since two objects intersect they have a common point, and therefore the vector from this common point to itself is contained in $A \ominus B$, which is the zero vector or the origin.

Definition 4.8: (Distance, Separation distance). (see e.g., Ong and Gilbert, 1996; Bergen, 2003)
The distance $\delta_{\mathrm{d}} > 0$ between two separated objects $A, B \in \mathcal{C}, A \neq B$ is most naturally defined as the shortest Euclidean distance

$$\delta_{\mathrm{d}}(A, B) = \min\{\|\boldsymbol{a} - \boldsymbol{b}\| :\ \boldsymbol{a} \in A,\ \boldsymbol{b} \in B\}. \tag{4.4}$$

Remark 4.4: The distance δ_{d} is commutative because $\delta_{\mathrm{d}}(A, B) = \delta_{\mathrm{d}}(B, A)$. Other synonyms for the distance between two separated objects are: they are disjunct, they are not in contact, they do not overlap, two objects are separated or a Euclidean distance can be found.

Definition 4.9: (Penetration depth). (Bergen, 2003)
The penetration depth of an intersecting pair $A, B \in \mathcal{C}, A \neq B$ is the length of the shortest vector over which one of the objects needs to be translated in order to bring the pair in touching contact

$$\delta_{\mathrm{p}}(A, B) = \inf\{\|\boldsymbol{r}\| : \boldsymbol{r} \notin A \ominus B\}. \tag{4.5}$$

Remark 4.5: The penetration depth δ_{p} is commutative because $\delta_{\mathrm{p}}(A, B) = \delta_{\mathrm{p}}(B, A)$. δ_{p} is the simplest definition that is based on pure geometric properties at the actual time instant. With a significant increase in complexity, there are several definitions of penetration depth which take object rotations (Ong, 1995) or take movement of contact points Zhang, Kim, and Manocha (2014) into account. Other synonyms for two penetrating objects are: overlapping objects, objects in contact, and intersecting objects.

The definition of two touching objects (Definition 4.10) is a special case of Theorem 4.4. This definition shows the relationship between the Euclidean distance (Definition 4.8) and penetration depth (Definition 4.9). Both are translational distances.

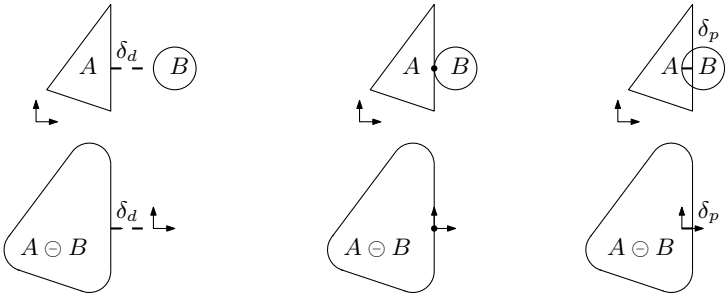
Definition 4.10: (Touching). (Ong and Gilbert, 1996)
Two objects are touching iff the origin is on the boundary of the Minkowski Difference

$$A \cap B \neq \emptyset \equiv \mathbf{0} \in \partial(A \ominus B). \tag{4.6}$$

Two objects are touching iff the Euclidean distance (4.4) $\delta_{\mathrm{d}} = 0$ as well as the penetration depth (4.5) $\delta_{\mathrm{p}} = 0$ are zero.

Remark 4.6: The definition of two touching objects (4.6) is a special case of two intersecting objects (4.3) because $\partial(A \ominus B) \subset A \ominus B$.

Figure 4.1 shows two objects A, B which are not in contact, touching and penetrating and their corresponding Euclidean distance and penetration depth and their resulting Minkowski Difference $A \ominus B$.



a) No contact. b) Touching. c) Penetrating.

Figure 4.1: Two objects A, B in different contact situations (upper row) with their corresponding Minkowski Difference $A \ominus B$ (lower row).

The Minkowski Difference (Definition 4.7) is applicable if both objects are convex. If not, the convex hull (see Section 4.3) of the concave object is taken. If two objects intersect, there is at least one point in object A that coincides with at least one point in object B. By subtracting these points from each other, the result will be the zero vector (the origin). Remember: forming Minkowski Difference means subtracting points (4.2). Both objects overlap if $A \ominus B$ contains the origin. Otherwise, if two objects do not intersect, no point from object A will be equal to any point of object B and $A \ominus B$ will not contain the origin (zero vector). Theorem 4.3 states, that there exists exactly one unique point on the boundary of $A \ominus B$ with the shortest distance to the origin (zero vector), if the origin is not contained. The uniqueness of the point of $A \ominus B$ closest to the origin does not imply that the distance between two convex objects is realized by a unique pair of points. In other words, there may exist multiple contact points $\boldsymbol{a} \in A$ and $\boldsymbol{b} \in B$ so that (4.4) is minimal. If both objects are penetrating, this shortest distance is equivalent to the penetration depth (see Definition 4.9), otherwise, it is equivalent to the Euclidean distance (see also Definition 4.8, Theorems 6.1 and 6.3). A similar discussion can be found in (Bergen, 2003; Snethen, 2008; Kenwright, 2015).

The Minkowski Difference with its useful properties reduces complexity. Instead of calculating the shortest distance between two convex objects or their convex hulls, the Minkowski Difference transfers the determination of the shortest distance, into the much simpler problem of determining the shortest distance between a convex object $A \ominus B$ to the origin. This transforms a shape-to-shape distance problem into a shape-to-point distance problem.

4.2 Support Mappings and Support Points

The calculation of the Minkowski Difference could become very time-consuming for complex geometries. Hence, instead of calculating each point in $A \ominus B$, only some well-chosen points are subtracted. Therefore, support mappings are introduced.

4.2.1 Support Mappings

Support mapping fully describes the geometry of a polytope A and can be viewed as the implicit representation of an object (Bergen, 2003; Snethen, 2008; Kenwright, 2015). It is a mathematical function that maps a vector $\boldsymbol{n}$, which is the search direction to a vertex on the boundary of polytope A that lies farthest in that direction. If multiple vertices satisfy this requirement, any vertex can be chosen as long as the same vertex is always mapped to a given search direction. Support mapping does not necessarily return the closest vertex in the search direction if it is not the farthest one (see Figure 4.2). It is worth mentioning that the search direction does not need to be a unit vector. For support mappings, the focus is only on the farthest vertex and not on the distance.

The input arguments of function `supportMapping` (Line 264) are:

- A search direction $\boldsymbol{n}$.
- A geometry `obj`. It is a polytope (a given collection of vertices) or a primitive geometry.

The support point computation depends on the shape and it is transformed into global coordinates with absolute translation vector `r` and absolute rotation matrix `R`. The return value of function `supportMapping` is the vertex on the object that lies farthest in that direction. This vertex is called support point.

```
264 supportMapping(obj, n) = obj.r + obj.R'*supportPoint(obj, obj.R*n)
```

Several support point mappings for primitive shapes and polytopes are stated in Bergen (2003), Snethen (2008) and Olvång (2010) and the concept of compound support mappings is further introduced. For example, support point computation for a box in Lines 266 to 271.

```
265 # support point mapping for box
266 function supportPoint(obj::Box, n)
267     x = 1/2*obj.lengthX
268     y = 1/2*obj.lengthY
269     z = 1/2*obj.lengthZ
270     return (sign(n[1])*x, sign(n[2])*y, sign(n[3])*z)
271 end
```

Support mappings for compound shapes like capsule, cone, frustum of a cone, and beam are not given in the common literature. As part of this thesis, support mappings for these compound shapes have been derived. Their implementation can be found online[1]. Moreover, for non-smooth shapes a newly invented smoothing radius is considered, see Section 6.2.2.

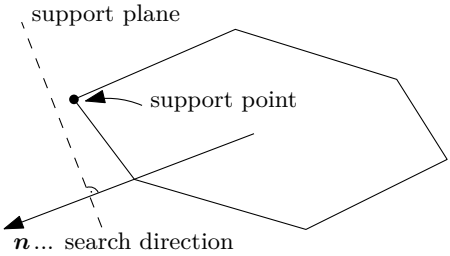


Figure 4.2: Support mappings can be imagined as a normal plane (support plane) to the search direction $\boldsymbol{n}$. The support plane slides along the search direction towards the object until they touch. This vertex is the support point.

[1] Modia3D.jl, v0.10.4, src/Shapes/boundingBoxes.jl

4.2.2 Support Points

It is easier to call `supportMapping` twice and find a support point on object A in search direction $\boldsymbol{n}$ (Line 273) as well as on object B in search direction $-\boldsymbol{n}$ (Line 274) and subtract both, then to directly compute a support point $\boldsymbol{p}$ in the Minkowski Difference $A \ominus B$. The support point (Line 275) lies in the Minkowski Difference $\boldsymbol{p} \in A \ominus B$ because it is the difference of $\boldsymbol{a} - \boldsymbol{b}$ where $\boldsymbol{a} \in A$ and $\boldsymbol{b} \in B$ (see Definition 4.7).

```
272 function support(A, B, n)
273     a = supportMapping(A, n)    # farthest point on boundary A in direction n
274     b = supportMapping(B, -n)   # farthest point on boundary B in direction -n
275     p = a - b                   # support point in Minkowski Difference
276     return SupportPoint(p, a, b, n)
277 end
```

Function `support` (Lines 272 to 277) computes and creates a new `SupportPoint` structure inspired by Kenwright (2015). A `SupportPoint` structure is specified by a search direction $\boldsymbol{n}$. All corresponding data to the search direction $\boldsymbol{n}$, its belonging support point $\boldsymbol{p} \in A \ominus B$, point $\boldsymbol{a} \in A$ in direction $\boldsymbol{n}$, and point $\boldsymbol{b} \in B$ in direction $-\boldsymbol{n}$ are stored in `SupportPoint` (Lines 278 to 283).

```
278 struct SupportPoint{T}
279     p::SVector{3,T}    # support point
280     a::SVector{3,T}    # point on object A
281     b::SVector{3,T}    # point on object B
282     n::SVector{3,T}    # support normal vector
283 end
```

If the distance between the support point on the boundary and the origin of the Minkowski Difference is minimal in direction of unit vector $\boldsymbol{n}$, the closest distance between objects A and B is found.

Remark 4.7: In the pseudo code snippets, `ri` refers to a `SupportPoint` structure (Lines 278 to 283). When referring to a support point in the code it is indicated by `ri.p` or `p`. For simplicity $\boldsymbol{r}_i$ also corresponds to the same support point in $A \ominus B$.

```
284 ri = support(A, B, n)        # i = 1, 2, 3, 4
```

This is a preparation for the MPR algorithm used in the narrow phase (see Chapter 6). This algorithm iteratively constructs up to four support points in a given search direction.

4.3 Convex Hull

Another use case of support mappings is generating convex hulls of their responding shapes, whether they are convex or concave shapes (see Bergen (2003) and Figure 4.3). For the mathematical description of a convex hull, see Appendix A. In this

thesis, for a concave shape, the distance computation is performed in the narrow phase with respect to its convex hull. One could think of another application: the distance between the convex hulls of two concave objects can be useful for collision avoidance algorithms, as well as for cheap, precise tests whether contact is possible for deformable objects (see e.g., Pungotra, 2010).

a) 2D shape. b) 3D shape.

Figure 4.3: Convex hulls of a concave 2D and a concave 3D shape.

4.4 Approximate Convex Decomposition

Not only convex objects collide but also concave ones. For this purpose, convex hulls are used for the decomposition of concave objects. An approximate convex decomposition is the composition of convex hulls by decomposing a concave shape into several concave shapes. In other words, a concave shape is divided into multiple shapes with their convex hulls. An approximate convex decomposition is the composition of these convex hulls (see Figure 4.4). Exact convex decomposition

a) 2D shape. b) 3D shape.

Figure 4.4: Approximate convex decompositions of a concave 2D and a concave 3D shape.

algorithms are NP-hard[2]. This can be done with V-HACD (see Mamou, Lengyel, and Peters, 2016; Mamou, online). In Modia3D the convex parts of one concave shape are rigidly attached, so no collisions are possible between them. This reduces the number of potential collision pairs (see Section 5.1). For a collision application with convex decompositions see Appendix B.

In summary, the most important concept for collision handling is the Minkowski Difference and it is introduced in Chapter 4. The Minkowski Difference and its properties are used for describing the relationship between a potentially-colliding pair. Those relationships are: objects are not in contact, objects are penetrating and are touching. For efficiently computing some well-chosen points in the Minkowski Difference and for generating convex hulls of their responding shapes support mappings and support points are introduced. These basic collision concepts are needed for the upcoming four collision handling steps, especially in the broad and narrow phase.

[2] For a suitable approximation of e.g., a hollow sphere or pipe more granular convex decompositions are needed.

5 Preprocessing and Broad Phase

Collision handling is performed in four steps. This chapter discusses the first two steps. The first step of collision handling – preprocessing – is executed only once (see Section 5.1). In contrast, the second collision handling step – the broad phase – is executed each time an update is requested by the solver (see Section 5.2).

5.1 Preprocessing

The preprocessing step, which is the first collision handling step, analyzes the mechanical structure of the 3D-model by mapping its components to an internal tree structure and reduces the number of collision pairs in the overall collision configuration (Neumayr and Otter, 2018; Neumayr and Otter, 2019a). This once-off additional preprocessing step reduces the number of potential collision pairs to n_{pp} and speeds up the upcoming broad phase. This leads to $n_{\text{pp}} \leq \frac{n_{\mathcal{C}}(n_{\mathcal{C}}-1)}{2}$ potential collision pairs.

There are two preprocessing rules:

1. Rigidly attached objects cannot collide with each other.
2. Objects connected by a joint cannot collide with each other if the joint specific option `canCollide` is set to `false` by the user (default setting).

Neumayr and Otter (2018) point out the two preprocessing rules and Neumayr and Otter (2019a) show how to implement this preprocessing step.

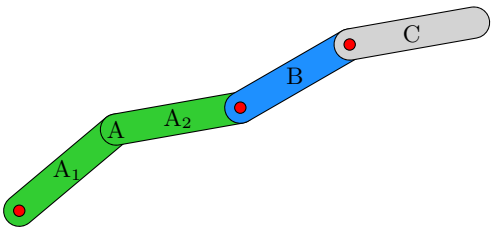


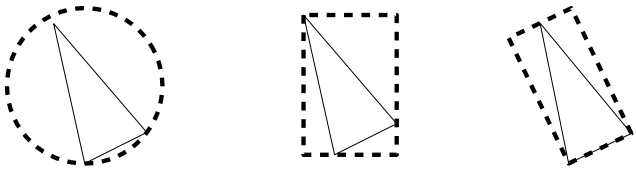
Figure 5.1: Preprocessing for rigidly attached objects and joints. The red circles represent revolute joints. Objects A_1, A_2 are rigidly connected. If both preprocessing rules are applied, two potential collision pairs $\{A_1, C\}$, $\{A_2, C\}$ remain.

For example, in Figure 5.1 the objects A_1, A_2 are rigidly connected, so when the first preprocessing rule is applied A_1 cannot collide with A_2, but both objects can still collide with all other objects. Applying the second preprocessing rule, the red cylinders denote revolute joints, instead of six, only two potential collision pairs $\{A_1, C\}$, $\{A_2, C\}$ remain and are checked in the broad phase.

5.2 Broad Phase

The broad phase, which is the second collision handling step, is an efficient and simple selection procedure: complex objects are approximated by simple bounding volumes, leading to $O(n_{\mathcal{C}}^2)$ cheap intersection tests (e.g., Bergen, 2003), without preprocessing. Only if the bounding volumes overlap the narrow phase is executed (see Chapter 6).

For the following literature review, it is assumed that no additional preprocessing step is performed that reduces the number of collision pairs. The use of bounding volumes is a tradeoff between additional storage and computational overhead, so performance is improved only when the bounding volumes are disjoint with high probability. Different types of bounding volumes are: bounding spheres, Axis-Aligned Bounding Boxes (AABBs), Oriented Bounding Boxes (OBBs), and k-Discrete Orientation Polytopes (k-DOPs) (Bergen, 2003). Further variations are bounding ellipsoids, bounding capsules, and bounding cylinders. Some bounding volumes are displayed in Figure 5.2. Bounding spheres are the simplest and the most commonly used, but often not the most tightly fitting bounding volumes. AABBs need more storage than spheres, but their intersection tests are faster. For an average object, the smallest AABB fits as badly as the smallest bounding sphere (Bergen, 2003). AABBs are aligned along three coordinate axes. Support mappings (Section 4.2) are used to easily create them (Bergen, 2003; Neumayr and Otter, 2018). An OBB is a rectangular bounding box with an arbitrary rotation, thus an AABB is a special case of an OBB (Gottschalk, Lin, and Manocha, 1996; Bergen, 2003). An OBB needs more storage and intersection testing cost than an



a) Bounding sphere. b) AABB. c) OBB.

Figure 5.2: Bounding volumes.

AABB, but for many objects it is the most tight-fitting approximation. A k-DOP generalizes an AABB. It has k arbitrary directions. It creates a convex polytope that generally better fits than an AABB (Bergen, 2003).

Furthermore, the approximated objects can be placed in a Bounding Volume Hierarchy (BVH) so that all direct and indirect children of a node cannot penetrate, if a collision is not possible for the node e.g., sphere trees (Hubbard, 1996), OBB trees (Gottschalk, Lin, and Manocha, 1996), k-DOP trees (Zachmann, 1998) and AABB trees (Bergen, 2003). Typically, $O(n_{\mathcal{C}} \log(n_{\mathcal{C}}))$ collision tests are being performed in this phase. Karras (2012) shows that the construction of BVHs can be parallelized by processing all nodes simultaneously using octrees and k-d trees. Extension of Morton codes (Vinkler, Bittner, and Havran, 2017) for 3D significantly improve the quality of the fastest available BVH construction algorithms LBHV by Karras (2012) and ATRBVH by Domingues and Pedrini (2015) while not increasing the computation time.

If the bounding volumes overlap, the signed distance between these two potentially-colliding objects is calculated in the narrow phase in Chapter 6. Otherwise, the objects cannot penetrate and the Euclidean distance δ_{broad} between two bounding volumes is calculated instead (see Definition 5.1).

Definition 5.1: (δ_{broad}). (Neumayr and Otter, 2019b)
Let $A, B \in \mathcal{C}, A \neq B$ be two objects. The Euclidean distance between two non-overlapping bounding volumes is

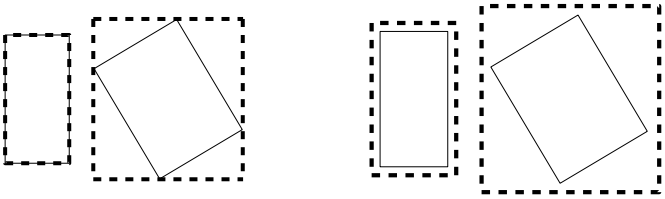
$$\delta_{\text{broad}}(A, B) > 0 \quad A, B \in \mathcal{C}, A \neq B.$$

In this thesis, the broad phase is realized with AABBs, since their implementation and intersection tests are simple, and they are extendable to BVH (which has not been done yet). The determination of δ_{broad} is just a simple geometric calculation, between two axis aligned boxes. Figure 5.3 is a 2D representation of AABBs.

```
285 supportMappingAABB(obj, axis, dir) =
286     obj.r[axis] + obj.R[:,axis]'*supportPoint(obj, dir*obj.R[:,axis]))

287 function AABB(obj)
288     xmin = supportMappingAABB(obj, 1, -1)
289     xmax = supportMappingAABB(obj, 1, +1)
290     ymin = supportMappingAABB(obj, 2, -1)
291     ymax = supportMappingAABB(obj, 2, +1)
292     zmin = supportMappingAABB(obj, 3, -1)
293     zmax = supportMappingAABB(obj, 3, +1)
294 end
```

Support mappings (see Section 4.2) are used for creating AABBs (Bergen, 2003; Neumayr and Otter, 2018). Therefore, no object specific AABB function is needed. An AABB of an object is calculated by calling the `supportMappingAABB` function 6 times (once per axis `axis = 1, 2, 3` and direction `dir = -1, 1`) for computing



a) Tight-fitting AABBs. b) Loose-fitting AABBs.

Figure 5.3: Loose-fitting AABBs are used to compute zero-crossing functions to detect transitions with variable-step solvers.

a support point in the reference coordinate system of the object. Thus, the object's absolute translation and absolute rotation is needed.

Tight-fitting AABBs are not useful when zero-crossing functions shall be computed because if some surfaces or edges of an object are also parallel to an axis, and these objects would incidentally collide, they are already penetrating each other (see Figure 5.3 and Neumayr and Otter (2018)). Therefore, it will not be possible for the variable-step solver to detect the transition (see Section 6.3) between penetration and non-penetration. Hence, to avoid such scenarios, loose-fitting AABBs are introduced. Therefore, each edge length of the tight-fitting AABB gets enlarged by a specific factor of the longest edge length, see Figure 5.3b.

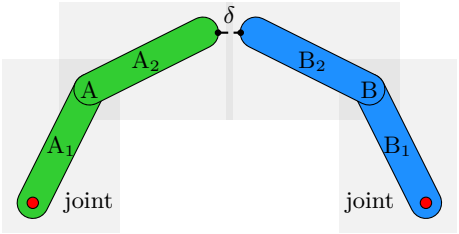


Figure 5.4: Preprocessing: The two preprocessing rules reduce the number of potential collision pairs from six to four. Broad Phase: AABBs of A_2 and B_2 overlap which leads to one potentially-colliding pair $\{A_2, B_2\}$. Narrow Phase: Perform a computationally expensive intersection test for $\{A_2, B_2\}$, no collision is detected.

In Figure 5.4 there are four potentially-colliding objects A_1, A_2, B_1, B_2 and each has its AABBs shown as a grey box. There are two rigidly attached objects: A consists of A_1 and A_2, and B consists of B_1 and B_2. The joints (red cylinders) connect A and B to the ground. The two preprocessing rules reduce the number

of potential collision pairs from six to four. These four potential collision pairs are examined in the broad phase to determine whether their AABBs overlap or not. The AABBs of A_2 and B_2 overlap. Consequently, the narrow phase has to be performed for one potentially-colliding pair $\{A_2, B_2\}$, no collision is detected.

To summarize, Chapter 5 discusses the first two steps needed for collision handling with variable-step solvers. The additional first step – the preprocessing step – is only performed once and makes use of the multibody setup. The second collision handling step – the broad phase – is executed each time an update is requested by the solver, regardless of changes in the objects' translation and rotation. In this thesis, the broad phase is based on AABBs. To avoid unnoticeable penetrations loose-fitting AABBs are proposed. It is checked if the AABBs are overlapping. If they do so, the narrow phase (see Chapter 6) is executed.

6 Narrow Phase

The narrow phase – the third collision handling step – is an exact and computationally expensive intersection test. The narrow phase is only executed if the broad phase detects that two objects could potentially collide. Depending on the approach, there are several different intersection tests, which could be performed in the narrow phase.

There are low-level intersection tests.[1] These basic intersection tests could be used in the broad phase. But in the narrow phase they are not applicable because too many object types combinations have to be implemented, like a sphere or box against a plane or box. In this thesis, any collision between two arbitrary geometries is considered.

Keeping this in mind, one could think of discretizing the volume of the geometries involved using Finite Element Method (FEM) for analyzing any arbitrary collision situation and the resulting deformation with best quality, but this results in huge computational effort. An interesting approach that points in this direction is discretizing the contact area for shapes consisting of triangles, the Polygonal Contact Model (PCM) (Hippmann, 2004a; Hippmann, 2004b). Hippmann (2004b) compares some of these approaches, like e.g., theory of elasticity, FEM, half space approximation, and more, for their suitability in multibody programs, each tailored to a specific application or which can only insufficiently represent the physical processes.

A compromise between the greatest possible versatility in handling arbitrary geometries and yet sufficient accuracy in dynamic simulation is computing penetration depth between convex shapes or their convex hull. Based on the Minkowski Difference $A \ominus B$ and support mappings, Snethen (2008) proposes the Minkowski Portal Refinement (MPR) algorithm to detect whether two convex objects penetrate. If this is the case, he suggests computing an approximation of the penetration

[1] Some low-level intersection tests are mentioned in the following list (see e.g., O'Rourke, 1998; Bergen, 2003; Ericson, 2004):

- Closest point computations, e.g.,: closest point on plane to point, closest point on line segment to point, closest point on triangle to point, closest point on convex polyhedron to point, and more.
- Testing primitives, e.g.,: separating-axis test, testing sphere or box or cone against plane, testing sphere against triangle or polygon, and more.
- Intersecting lines, rays, and (directed) segments, e.g.,: intersecting ray or segment against sphere or box or triangle, and more.

depth. The MPR algorithm can be used to compute a lower and upper bound on the closest Euclidean distance of two non-penetrating convex objects (Neumayr and Otter, 2017). The MPR algorithm in 3D is much simpler than the often used Gilbert-Johnson-Keerthi (GJK: Gilbert, Johnson, and Keerthi, 1988) and Expanding Polytope algorithm (EPA: Bergen, 2003) because it operates with simpler geometries. The drawback is that MPR may only compute an approximation of the penetration depth in some situations.

For all three algorithms, the (unique) contact point found in the Minkowski Difference is transformed back to the (potentially ambiguous) contact points on shapes A and B using the barycentric coordinates of the contact point with respect to the final simplex or polytope. For further information to barycentric coordinates (see Bergen, 2003). In the following, these three algorithms are looked at in more detail.

Gilbert-Johnson-Keerthi (GJK) Algorithm

The Gilbert-Johnson-Keerthi (GJK: Gilbert, Johnson, and Keerthi, 1988) algorithm is used for intersection tests between two convex shapes. It computes the shortest distance between Minkowski Difference $A \ominus B$ and the origin. The approach is conceptually simple by using support mappings to construct appropriate simplices consisting of one to four vertices on the boundary of the shape. In the 3D case a simplex is a point, a line, a triangle, or a tetrahedron. The simplices are iteratively updated to move closer and closer to the origin. For each iteration it is checked if the simplex contains the origin. If so $A \ominus B$ must contain the origin, too. For penetrating shapes, the GJK algorithm stops with a simplex that has the origin in its interior. If the origin is outside the shape and no further progress is possible, then the closest distance of the shape from the origin corresponds to the closest distance of the final simplex from the origin. To summarize, for non-penetrating shapes the GJK algorithm computes the closest distance. For penetrating shapes, the algorithm detects that they do so and the Expanding Polytope algorithm (EPA) computes the penetration depth.

Expanding Polytope Algorithm (EPA)

The Expanding Polytope algorithm (EPA: Bergen, 2003) computes the penetration depth between two convex shapes, if the origin is in the interior of the resulting $A \ominus B$. The final simplex of the GJK algorithm is used as the initial simplex for EPA. It expands the simplex to a polytope where in every iteration a new support point vertex is added. If no further progress is possible, the penetration depth is the closest distance of the final polytope from the origin.

The GJK and EPA algorithms are conceptually simple, but the implementation is non-trivial and elaborate due to the many special cases (e.g., handling 1, 2, 3, 4-simplices in all situations) and due to various numerical problems, that might occur when selecting the next simplex or computing a termination condition. This

might be the reason why no robust open-source implementation with a permissive free license seems to be available for the 3D-case.[2]

Minkowski Portal Refinement (MPR) Algorithm

Based on the Minkowski Difference, Snethen (2008) and Snethen (online[a]) propose the Minkowski Portal Refinement (MPR) algorithm, also known as XenoCollide, to detect whether two convex shapes intersect. If this is the case, an approximation of the penetration depth is computed. The MPR algorithm in 3D is much simpler than GJK with EPA because it uses a geometric approach and operates basically with triangles. The drawback is that the MPR algorithm may only compute an approximation of the penetration depth in some situations. Olvång (2010) compared several millions randomized benchmarks of various object types for MPR, GJK and EPA. These comparisons were not conclusive, which algorithm is the most suitable, it depends on the context. Olvång (2010) recommends using the MPR algorithm instead of GJK and EPA algorithms, since both are heuristic methods. Kenwright (2015) offers a compact pseudo code for the MPR algorithm and shows that the MPR algorithm can also be used to compute the closest distance of non-penetrating convex shapes.

The BSD-licensed open-source C-library libccd (Fišer, online) provides an implementation of the MPR algorithm. It is used for example in the open-source zlib-licensed Bullet Physics SDK. However, libccd does not compute the Euclidean distance of non-penetrating shapes and can therefore not be used as basis of the investigation of this thesis.

In the narrow phase, the MPR algorithm is used to calculate the contact information consisting of a (unique) contact point on each shape, a penetration depth or Euclidean distance, and a contact normal between the two colliding convex shapes or their convex hull. In this thesis, several improvements to the MPR algorithm are made to apply it to collision handling with variable-step solvers, see Section 6.1. The properties of the improved MPR algorithm are discussed in Section 6.2. So, that the Euclidean distance or the penetration depth is appropriate as zero-crossing function to detect collision events, see Section 6.3.

6.1 Improved MPR Algorithm in 3D

The Minkowski Portal Refinement (MPR) algorithm uses the Minkowski Difference to transform a shape-to-shape distance problem into a shape-to-point distance problem. In other words, to calculate the distance between two shapes, the distance between the Minkowski Difference and the origin is calculated instead. As discussed in Section 4.1, if the Minkowski Difference encloses the origin objects are penetrating, if not, there is no contact, if the origin lies on the boundary of the Minkowski

[2] The GJK and EPA based software SOLID (Bergen, online) is available under GPL license and accompanying Bergen (2003).

Difference objects are touching. The idea of the MPR algorithm is to construct a tetrahedron that lies in the Minkowski Difference. If objects are penetrating, this tetrahedron encloses the origin, otherwise objects are not penetrating, see Figure 6.1. One point ($\boldsymbol{r}_0$) of the tetrahedron is in the interior of the Minkowski Difference. Three points ($\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3$) of the tetrahedron are on the boundary of the Minkowski Difference and span up a portal. These portal points are iteratively refined until no further progress is possible and the closest distance δ from the portal to the origin is found.

The novel concept of the *signed distance* combines Definitions 4.8 to 4.10 and is crucial for collision handling with variable-step solvers.

Definition 6.1: (Signed Distance).
Let $A, B \in \mathcal{C}, A \neq B$ be two potentially colliding convex objects, or their convex hull. The signed distance δ is

$$\delta = \begin{cases} \delta_\mathrm{d} & \text{Euclidean distance} & \text{if } A, B \text{ are disjunct,} \\ 0 & & \text{if } A, B \text{ are touching,} \\ -\delta_\mathrm{p} & \text{penetration depth} & \text{if } A, B \text{ are penetrating.} \end{cases} \tag{6.1}$$

The MPR algorithm in 3D is based on publications of Snethen (2008), Olvång (2010) and Kenwright (2015). In this thesis several considerable enhancements to the MPR algorithm are discussed exhaustively. To highlight these enhancements, the unique line numbers of Julia like pseudo code snippets are marked in red. The open-source implementation of the improved MPR algorithm can be found in Modia3D.jl under an MIT (expat) license. Neumayr and Otter (2017) already published parts of the improved version. In the meantime, further improvements

a) Objects are penetrating. b) Objects are disjunct.

Figure 6.1: If the tetrahedron encloses the origin (visualized as coordinate system), objects are penetrating, otherwise objects are disjunct. The origin ray, from the interior point $\boldsymbol{r}_0$ to the origin intersects the valid portal ($\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3$). The signed distance δ is the length between the origin and the intersection point of the ray with the portal.

are made which are compared to common literature in Section 6.1.5. Additionally, accuracy and termination tolerances are investigated in Section 6.1.4. The MPR algorithm can be structured into three phases (see Olvång, 2010).

1. The first phase is constructing an initial portal (Section 6.1.1).
2. The second phase is finding a valid portal (Section 6.1.2).
3. The third phase is iteratively refining the portal (Section 6.1.3) until no further progress is possible and the closest distance from the portal to the origin is found.

The MPR algorithm in 3D provides contact information (e.g., contact points, the shortest distance between objects and contact normal) for two potentially-colliding objects. For getting started, Snethen (online[b]) gives a short and simple overview of the MPR algorithm in 2D.

The improved MPR algorithm has the following interface, structure, and output variables:[3]

```
295 function mpr(A, B; mprTol = 1.0e-20, mprIterMax = 120)
296     # phase 1: construct interior and initial portal points
297     r0::SupportPoint = constructR0(A, B)
298     r1::SupportPoint = constructR1(A, B, r0)
299     (isTC1, n2) = checkTC1(A, B, r0, r1)
300     if isTC1
301         δ = -dot(r1.p, r1.n)
302         (a, b, n) = r1.a, r1.b, r1.n
303         return (δ, a, b, n)
304     end
305     r2::SupportPoint = constructR2(A, B, n2)
306     (r2, r3::SupportPoint) = constructR3(A, B, r0, r1, r2)
307     # phase 2: valid portal
308     (r1, r2, r3) = validPortal(A, B, r0, r1, r2, r3, mprIterMax)
309     # phase 3: portal refinement
310     (δ, a, b, n) = portalRefinement(A, B, r0, r1, r2, r3, mprTol, mprIterMax)
311     return (δ, a, b, n)
312 end
```

Input Variables of `mpr`

Function `mpr` (Line 295) provides contact information of the input objects A and B. Both objects are convex or concave objects with known geometries, absolute translations, and centroids. For a concave shape its convex hull is taken (see Section 4.3). To avoid infinite looping, the maximum number of iterations is limited to `mprIterMax`. `mprTol` is the MPR termination tolerance for approximating the distance and terminating the MPR algorithm. For a detailed discussion see Section 6.1.4.

[3] The released and improved MPR algorithm is tuned for efficiency and avoids allocations under Julia. In contrast, the MPR pseudo code in this thesis focuses on readability and comprehensibility.

Output Variables of `mpr` — Contact Information
The computed distance between objects A and B by `mpr` algorithm (Line 295) corresponds to the signed distance δ, see Definition 6.1. The computed distance δ is commutative because δ_d and δ_p are commutative. Thus, `mpr(A,B) = mpr(B,A)`. Contact points $\boldsymbol{a}, \boldsymbol{b}$ are absolute translation vectors on the boundary of object A and on the boundary of object B where the computed signed distance δ is minimal (see Figure 6.2). The unit vector $\boldsymbol{n}$ spans across both contact points $\boldsymbol{b} - \boldsymbol{a} = \delta \boldsymbol{n}$. If the surface around contact point $\boldsymbol{a}$ is differentiable, the contact normal $\boldsymbol{n}$ is perpendicular to object A in point $\boldsymbol{a}$. The same holds for a differentiable surface around contact point $\boldsymbol{b}$. In the following, the three phases of the MPR algorithm are looked at in more detail.

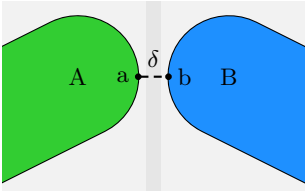


Figure 6.2: Objects A, B are not in contact, but their loose-fitting AABBs overlap (grey boxes, see Section 5.2). The computed signed distance $\delta > 0$ is minimal between contact points $\boldsymbol{a}, \boldsymbol{b}$.

6.1.1 Phase 1: Constructing an Initial Portal

The goal of the first phase is to construct an initial tetrahedron. The initial tetrahedron has to fulfill the following properties:

- The tetrahedron lies in the Minkowski Difference $A \ominus B$.
- The tetrahedron has a non-zero volume.
- The tetrahedron consists of support point $\boldsymbol{r}_0$ in the interior and three further support points $\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3$ on the boundary of $A \ominus B$.

These properties must also be fulfilled by tetrahedrons in the upcoming phases. For further information on support points see Section 4.2.

Definition 6.2: (Origin Ray). (e.g., Snethen, 2008)
The ray drawn from the interior point $\boldsymbol{r}_0$ through the origin (zero vector) of $A \ominus B$ is the origin ray.

Definition 6.3: (Valid Portal). (e.g., Snethen, 2008)
A portal is a triangle spanned by three support points $\boldsymbol{r}_1$, $\boldsymbol{r}_2$, and $\boldsymbol{r}_3$ on the boundary of $A \ominus B$. Moreover, a portal is a valid portal if the origin ray intersects the triangle spanned by three support points.

If the origin of the Minkowski Difference $A \ominus B$ is enclosed by the tetrahedron, objects A and B are penetrating. Otherwise, both objects are not penetrating each other (see Theorem 4.1 - Theorem 4.4). Both situations are shown in Figure 6.1. Furthermore, the origin ray from $\boldsymbol{r}_0$ through the origin of $A \ominus B$ must intersect the portal, otherwise new support points must be constructed, until it is a valid portal (Definition 6.3). The signed distance δ is the length between the origin and the intersection point of the ray with the portal triangle. These principles are illustrated in Figure 6.1, Page 60.

Constructing Interior Point $\boldsymbol{r}_0$

The support point $\boldsymbol{r}_0$ must be an arbitrary interior point of Minkowski Difference $A \ominus B$ and it is stored with `SupportPoint` (Lines 278 to 283, Page 48). It would be sufficient to take any two arbitrary points from the interior of object A as well as from the interior of object B. A convenient choice is the geometric center or center of mass (Snethen, 2008). To be more precise, the geometric center called `centroid` of each object is used (Kenwright, 2015) in this thesis.

```
313 # interior point r0
314 function contructR0(A, B)
315     p = centroid(A) - centroid(B)
316     if norm(r0.p) < ε
317         @error "Centers are overlapping."
318     end
319     a = zeros(3), b = zeros(3) # default values, zero vector
320     r0 = SupportPoint(p, a, b, -p)
321     return r0
322 end
```

If the two geometric centroids coincide, the constructed interior point is exactly the origin of the Minkowski Difference. Therefore, $\|\boldsymbol{r}_0\| = 0$ and a division by zero will occur by normalizing the first search direction in Line 325. In Lines 316 to 318 numerical inaccuracies are considered with ε see Section 6.1.4.

Constructing Initial Portal Points

Starting from the interior point, three linearly independent points are constructed on the boundary of Minkowski Difference $A \ominus B$, forming a triangular portal through which the origin ray could pass. There are many ways of constructing three linearly independent points. Here, the construction instruction by Snethen (2008) and Kenwright (2015) is followed.

Constructing Initial Portal Point $\boldsymbol{r}_1$

The first portal point $\boldsymbol{r}_1$ is found by searching in the normalized direction of the origin ray and by using function `support` (Lines 272 to 277, Page 48).

```
323 # initial portal point r1
324 function constructR1(A, B, r0)
325     e1 = -normalize(r0.p)
326     r1 = support(A, B, e1)
```

```
327     return (r1, e1)
328 end
```

Constructing Initial Portal Point r_2

The search direction for the second portal point r_2 is orthogonal to the plane containing the interior point r_0 and first support point r_1. To intercept a division by zero, Termination Condition (TC1) is introduced and the MPR algorithm is terminated. Otherwise, initial portal point r_2 is constructed.

```
329 # check if TC1 is valid
330 function checkTC1(A, B, r0, r1)
331     n2 = cross(-r0.p, r1.p)
332     TC1 = norm(n2)
333     isTC1 = TC1 < ε          # termination condition 1
334     return isTC1, n2
335 end

336 # initial portal point r2
337 function constructR2(A, B, n2)
338     e2 = normalize(n2)
339     r2 = support(A, B, e2)
340     return r2
341 end
```

Termination Condition (TC1)

To intercept a division by zero while normalizing the search direction (Line 338), and to terminate the MPR algorithm (Lines 300 to 304, Page 61) the termination condition (TC1) (Line 333)

$$\|\boldsymbol{r}_0 \times \boldsymbol{r}_1\| < \varepsilon \tag{TC1}$$

is introduced.

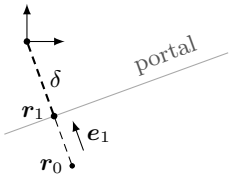


Figure 6.3: (TC1): $\boldsymbol{r}_1$ is on the ray from $\boldsymbol{r}_0$ through the origin. So, $\|\boldsymbol{r}_0 \times \boldsymbol{r}_1\| = 0$.

The cross product of two vectors is the zero vector, iff both vectors are parallel (Bronstein et al., 2001) or at least one vector is a zero vector. If the cross product is zero, its norm is zero as well. This means $\boldsymbol{r}_0$ and $\boldsymbol{r}_1$ are not linearly independent. Closer inspection shows, $\boldsymbol{r}_1$ is on the origin ray from $\boldsymbol{r}_0$ to the origin, as visualized in Figure 6.3. Since $\boldsymbol{r}_1$ is already the closest point to the origin, the signed distance

δ can be calculated directly and the distance calculation will be terminated by the MPR algorithm (Lines 300 to 304, Page 61). This situation occurs whenever the nearest points lie on the line passing through the centers of the two objects, e.g., two spheres are colliding.

Constructing Initial Portal Point r_3

The search direction for the third portal point $\boldsymbol{r}_3$ is orthogonal to the plane containing the interior point and the two support points already found. This means that the search direction is orthogonal to the spanned plane $\boldsymbol{r}_1 - \boldsymbol{r}_0$ and $\boldsymbol{r}_2 - \boldsymbol{r}_0$ and points to the origin. The following lines of code determine whether the resulting collision situation has degenerated into a 2D Minkowski Difference. The distance calculation would then have to be done with a 2D version of the MPR algorithm using a line and not a triangle for the portal. However, computing penetration depth in 3D of a zero volume object (2D object) would not make sense in the context of this thesis.

```
342 # initial portal point r3
343 function constructR3(A, B, r0, r1, r2)
344     n3 = cross(r1.p - r0.p, r2.p - r0.p)
345     if norm(n3) < ε
346         r2 = support(A, B, -r2.n) # change search direction
347         n3 = cross(r1.p - r0.p, r2.p - r0.p)
348         if norm(n3) < ε
349             @error "Shapes are planar and MPR for 2D is not supported."
350         end
351     end
352     if dot(r0.p, n3) > 0
353         n3 = -n3
354     end
355     e3 = normalize(n3)
356     r3 = support(A, B, e3)
357     # check if portal triangle r1, r2, r3 has degenerated into a line segment
358     if norm(cross(r2.p - r1.p, r3.p - r1.p)) < ε
359         r3 = support(A, B, -e3) # change search direction
360         if norm(cross(r2.p - r1.p, r3.p - r1.p)) < ε
361             @error "Shapes are planar and MPR for 2D is not supported."
362         end
363     end
364     return (r2, r3)
365 end
```

When normalizing the search direction $\boldsymbol{n}_3$ a division by zero is intercepted in Line 345 if $\|\boldsymbol{n}_3\| = 0$. Closer inspection of Line 344 shows $\boldsymbol{r}_1 - \boldsymbol{r}_0$ is parallel to $\boldsymbol{r}_2 - \boldsymbol{r}_0$ if $\|\boldsymbol{n}_3\| = 0$. This means that $\boldsymbol{r}_2$ is on the ray from $\boldsymbol{r}_0$ to $\boldsymbol{r}_1$. This situation is depicted in Figure 6.4. In order to proceed, it is tried to find another portal point $\boldsymbol{r}_2$ by searching in the opposite direction $-\boldsymbol{e}_2$ (Line 346). If the newly constructed $\boldsymbol{r}_2$ also lies on the ray, the Minkowski Difference of the two objects is planar (or very thin) (Line 348). Otherwise, the Minkowski Difference is not planar and

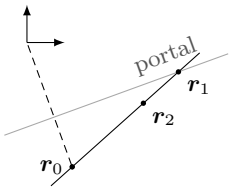


Figure 6.4: r_2 is on the ray from r_0 through r_1. So, $\|(r_1 - r_0) \times (r_2 - r_0)\| = 0$.

normalizing n_3 is allowed. If the newly found n_3 points in the wrong direction, its sign must be swapped.

To ensure the initial portal triangle (r_1, r_2, r_3) has a non-zero area, the cross product $(r_2 - r_1) \times (r_3 - r_1)$ is not allowed to be a zero vector (Line 358). Otherwise, support points r_1, r_2 and r_3 are on one line and the triangle of these three points has a zero area. It helps to take an opposite search direction $-e_3$ and newly compute r_3 (Line 359). The Minkowski Difference is planar or very thin, if the cross product is zero again (Line 360). Thus, the 3D problem of finding a signed distance has degenerated into a 2D problem and an error is triggered. Otherwise, a valid initial portal triangle with a non-zero area consisting of (r_1, r_2, r_3) was found.

6.1.2 Phase 2: Constructing a Valid Portal

Departing from the initial portal points of phase 1 (see Section 6.1.1), the algorithm repeats constructing new portal points until they span a valid portal (see Definition 6.3). This is achieved if the origin ray intersects the portal triangle (r_1, r_2, r_3). It follows that the tetrahedron must contain the origin if the objects overlap. Otherwise, new portal points must be constructed until this condition is fulfilled (see Figure 6.1).

To avoid infinite looping, the for-loop iterates until the origin ray intersects the portal or the maximum number of iterations is reached (see Section 6.1.4). For each iteration a new support point is calculated and one of the previous support points is replaced by a new one. When searching for the new support point r_3 (Line 373 and Line 379), the normalization of n_3 cannot lead to a division by zero because this is calculated only if $r_0 \cdot n_3 > 0$ is positive.

```
366 # phase 2: valid portal
367 function validPortal(A, B, r0, r1, r2, r3, mprIterMax)
368     success = false
369     for i=1:mprIterMax
370         n3 = cross(r1.p - r0.p, r3.p - r0.p)
371         if dot(r0.p, n3) > 0
372             r2 = r3
373             r3 = support(A, B, normalize(n3))
```

```
374             continue
375         end
376         n3 = cross(r3.p - r0.p, r2.p - r0.p)
377         if dot(r0.p, n3) > 0
378             r1 = r3
379             r3 = support(A, B, normalize(n3))
380             continue
381         end
382         success = true
383         break
384     end
385     if !success
386         @error "Phase 2 of MPR did not converge.
387                 Please, consider to increase mprIterMax."
388     end
389     return (r1, r2, r3)
390 end
```

6.1.3 Phase 3: Portal Refinement – Determining the Portal Closest to the Origin and Terminating the Algorithm

In the third and final phase, the portal triangle is refined and positioned closer to the origin until no further progress is possible and thus the closest distance to the origin is found. In every iteration a new support point $\boldsymbol{r}_4$ is constructed by searching in the orthonormal direction of the actual valid portal. If support point $\boldsymbol{r}_4$ is the closest point to the origin, the MPR algorithm is terminated with the newly introduced termination condition (TC2). Alternatively, if it is in the plane of the portal it is finished with termination condition (TC3). Otherwise, support point $\boldsymbol{r}_4$ is taken as the vertex spanning three portal candidates $(\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_4), (\boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{r}_4)$

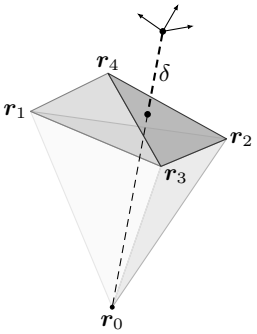


Figure 6.5: Support point $\boldsymbol{r}_4$ is taken as the vertex spanning three portal candidates. The origin ray intersects the refined valid portal $(\boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{r}_4)$.

and $(\boldsymbol{r}_3, \boldsymbol{r}_1, \boldsymbol{r}_4)$. The portal candidate intersected by the origin ray becomes the new refined valid portal (see Definitions 6.2 and 6.3). In Figure 6.5, the triangle $(\boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{r}_4)$ is the new refined portal. If the refinement fails or the maximum number of iterations is reached (Section 6.1.4), the MPR algorithm is terminated with the best found tolerance so far.

```
391 # portal refinement
392 function portalRefinement(A, B, r0, r1, r2, r3, mprTol, mprIterMax)
393     newTol = 42.0      # initialized by an arbitrary positive value
394     for i=1:mprIterMax
395         (r3, r4, e4) = constructR4(A, B, r0, r1, r2, r3)
396         TC2 = norm(cross(r4.p, e4))
397         TC3 = abs(dot(r4.p - r1.p, e4))
398         if TC2 < mprTol        # termination condition 2
399             return finalTC2(r1, r2, r3, r4)
400         elseif TC3 < mprTol    # termination condition 3
401             return finalTC3(r0, r1, r2, r3, r4)
402         else
403             (r1_best, r2_best, r3_best, r4_best, isTC2, isTC3, newTol) =
404                 storeSupportPointsBestTC(r1, r2, r3, r4, TC2, TC3, newTol)
405         end
406         (nextPortal, r1, r2, r3) = refinePortal(r0, r1, r2, r3, r4)
407         if !nextPortal # refinePortal failed, no better solution possible
408             @warn "MPR terminated with mprTol = $newTol."
409             return terminateMPR(r0, r1_best, r2_best, r3_best,
410                                 r4_best, isTC2, isTC3)
411         end
412     end
413     @warn "MPR terminated with mprTol = $newTol and mprIterMax = $mprIterMax.
414            Please, increase mprIterMax and/or mprTol."
415     return terminateMPR(r0, r1_best, r2_best, r3_best, r4_best, isTC2, isTC3)
416 end
```

Constructing Portal Point $\boldsymbol{r}_4$

The specific goal of the refinement phase is determining the distance δ. The search direction for the fourth portal point $\boldsymbol{r}_4$ is orthonormal to the currently valid portal $(\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3)$.

With the procedure already known from Section 6.1.1 (Lines 343 to 365, Page 65), it is checked whether the candidate portal triangle has degenerated into a line segment or not and if normalizing $\boldsymbol{n}_4$ is allowed.

```
417 # construct portal point r4
418 function constructR4(A, B, r0, r1, r2, r3)
419     n4 = cross(r2.p - r1.p, r3.p - r1.p)
420     # check if portal triangle r1, r2, r3 has degenerated into a line segment
421     if norm(n4) < ε
422         r3 = support(A, B, -r3.n) # change search direction
423         n4 = cross(r2.p - r1.p, r3.p - r1.p)
424         if norm(n4) < ε
425             @error "Objects are purely planar. Implement 2D MPR algorithm."
```

```
426         end
427     end
428     if dot(r0.p, n4) > 0
429         n4 = -n4
430     end
431     e4 = normalize(n4)
432     r4 = support(A, B, e4)
433     return (r3, r4, e4)
434 end
```

Termination Condition (TC2)
In case, the new support point $\boldsymbol{r}_4$ is parallel to $\boldsymbol{e}_4$. They are located on the same ray, see Figure 6.6. No further progress is possible. $\boldsymbol{r}_4$ is the closest point to the origin, so no better solution can be found. In order to terminate the third phase of the MPR algorithm, the termination condition (TC2) (Line 398, Page 68)

$$\|\boldsymbol{r}_4 \times \boldsymbol{e}_4\| \leq \texttt{mprTol} \tag{TC2}$$

is introduced. In this particular case, the signed distance δ (Line 436) can be calculated directly, it is negative if shapes are in contact, otherwise not.

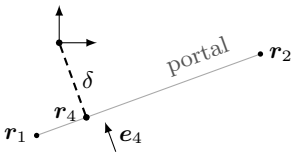


Figure 6.6: (TC2): $\boldsymbol{r}_4$ is parallel to the search direction $\boldsymbol{e}_4$. They are located on the same ray. So, $\|\boldsymbol{r}_4 \times \boldsymbol{e}_4\| = 0$.

Finally, the support point `r4.p` is the closest point to the origin in the Minkowski Difference. It is transformed to a contact point `r4.a` on object A and a contact point `r4.b` on object B with barycentric coordinates (Line 437) and the distance calculation will be terminated by the MPR algorithm. For more information on barycentric coordinates see Bergen (2003).

```
435 function finalTC2(r1, r2, r3, r4)
436     δ = -dot(r4.p, e4)
437     (r4.a, r4.b) = barycentric(r1, r2, r3, r4, e4)
438     return δ, r4.a, r4.b, r4.n
439 end
```

Termination Condition (TC3)
In case, the new support point $\boldsymbol{r}_4$ is in the plane of the portal. The portal and vector $\boldsymbol{r}_4$ are the closest to the origin, to which the ray passes. No further progress is possible. In order to terminate the third phase of the MPR algorithm, the

termination condition (TC3)[4] (Line 400, Page 68)

$$|(\boldsymbol{r}_4 - \boldsymbol{r}_1) \cdot \boldsymbol{e}_4| \leq \texttt{mprTol} \tag{TC3}$$

is used.

a) The closest point is in the interior of the portal.

b) The closest point is on the boundary of the portal.

Figure 6.7: (TC3): The closest point to the origin of the Minkowski Difference, lies in the interior or on the boundary of the portal. The length of the dashed thick line is the shortest distance δ from the portal to the origin.

As can be seen in Figure 6.7, the signed distance is then either the orthogonal distance between the portal and the origin (see Figure 6.7a) or it is the smallest distance between one of the vertices or one of the edges of the triangle (see Figure 6.7b). The implementation of Line 441 can be found in Modia3D.jl.

Finally, `r4.p` is transformed with barycentric coordinates (Line 442) to contact points `r4.a, r4.b`, and the distance calculation will be terminated by the MPR algorithm.

```
440 function finalTC3(r0, r1, r2, r3, r4)
441     (δ, r4.p, r4.n) = distanceToPortal(r0, r1, r2, r3, e4)
442     (r4.a, r4.b) = barycentric(r1, r2, r3, r4, e4)
443     return (δ, r4.a, r4.b, r4.n)
444 end
```

Refining Portal

The new portal point $\boldsymbol{r}_4$ and two old portal points are used to build up a new refined portal triangle which is closer to the origin and is passed by the ray from $\boldsymbol{r}_0$ to the origin (see Figure 6.5). Only, if the tested refined triangle is a valid portal – which means the two tested parallelepipedial products are negative – the support point $\boldsymbol{r}_4$ is replaced by an old portal point which is no longer used. Otherwise, if no better portal can be found, the MPR algorithm terminates with the best fitting solution (see Line 409, Page 68).

[4] Since $\boldsymbol{r}_1$ is the base of the portal triangle, it is also used in the termination condition. It can be shown, that the here used termination condition is equivalent to $|(\boldsymbol{r}_4 - \boldsymbol{r}_3) \cdot \boldsymbol{e}_4|$ like Snethen (2008) and Kenwright (2015) are using.

```
445 function refinePortal(r0, r1, r2, r3, r4)
446     nextPortal = true
447     if isNextPortal(r0, r1, r2, r4)
448         r3 = r4
449     elseif isNextPortal(r0, r2, r3, r4)
450         r1 = r4
451     elseif isNextPortal(r0, r3, r1, r4)
452         r2 = r4
453     else
454         # the signs of all tried combinations of
455         # parallelepipedial products are not negative
456         nextPortal = false
457     end
458     return (nextPortal, r1, r2, r3)
459 end

460 # computes twice a parallelepipedial product
461 isNextPortal(r0,r1,r2,r4) =
462     dot(cross(r2.p - r0.p, r4.p - r0.p), r0.n) < 0 &&
463     dot(cross(r4.p - r0.p, r1.p - r0.p), r0.n) < 0
```

Terminating the MPR Algorithm

There are three reasons why the MPR algorithm terminates. First, in phase 1 the MPR algorithm detects if the centroids of both objects coincide. In some collision cases e.g., two spheres are colliding, the signed distance can be computed directly with termination condition (TC1). Second, in phase 3 the MPR algorithm is terminated if the termination conditions (TC2) or (TC3) are fulfilled with the required MPR termination tolerance. Third, the MPR algorithm is terminated when it is impossible to further refine the portal (see Line 409, Page 68) or when the maximum number of iterations is reached (see Line 415, Page 68). If the latter occurs, the solution closest to the tolerance so far is stored and the respective termination condition has to be defined.

```
464 function terminateMPR(r0, r1_best, r2_best, r3_best, r4_best, isTC2, isTC3)
465     if isTC2
466         return finalTC2(r1_best, r2_best, r3_best, r4_best)
467     end
468     if isTC3
469         return finalTC3(r0, r1_best, r2_best, r3_best, r4_best)
470 end; end
```

Remark 6.1: According to Snethen (2008), in the refinement steps of the algorithm, the refined portal rapidly approaches the boundary of $A \ominus B$. If $A \ominus B$ has a curved boundary, the origin may lie infinitesimally close to this curved boundary. Thus, the refined portals may take an arbitrary number of iterations to pass the origin. The algorithm terminates under these conditions when the portals get sufficiently close to the surface. If the distance between the portal and its parallel support plane remains below a defined tolerance, the MPR algorithm terminates.

6.1.4 Accuracy and MPR Termination Tolerances

When dealing with the MPR algorithm and variable-step solvers it turns out that solvers fail for some complex collision models. In addition, some collision models still require a lot of CPU time even if the MPR algorithm has been optimized for efficiency. So, some of Modia3D's collision models[5] are analyzed in more detail on a standard notebook[6]. These collision models are explained briefly in Appendix B. A closer examination of two collision models (Model 1[7] (Figure B.7) and Model 2[8] (Figure B.8)) reveals that the MPR algorithm performs best with a MPR termination tolerance of 10^{-20}. This requires that the MPR algorithm must be executed with quadruple precision. Moreover, to avoid infinite looping of the MPR algorithm the iterations in each phase of the algorithm are limited. A sufficient default value is chosen on the basis of this evaluation.

Accuracy

Since the MPR algorithm is numerical sensitive, it is performed with quadruple precision (significant 106 bits), also known as Double64 from DoubleFloats.jl package (JuliaMath, online), to increase robustness, reliability, and to guard against overflow. In addition, it speeds up the algorithm. Therefore, for Double64 it is $\varepsilon \approx 10^{-30}$ in Line 471 (for Julia's default Float64 it is $\varepsilon \approx 10^{-14}$).

```
471 ε = 100.0 * eps(T) # precision depends on type of T, e.g., Double64
```

Maximum Iterations

To avoid infinite looping of phase 2 and phase 3 of the MPR algorithm the number of iterations is limited. The limit is `mprIterMax`. To find a sufficient default value for this limit, some of Modia3D's collision models are analyzed with a MPR termination tolerance `mprTol` $= 10^{-20}$ in Table 6.1. It turns out that 120 iterations for each phase and each MPR call are adequate. If fewer iterations are required for these phases, they will be terminated earlier, otherwise a warning is displayed. The maximum number of required iterations to quit phase 2 and phase 3 across all MPR calls of a simulation is called `i_max`.

- Phase 2: It is completed after a maximum of $4-7$ iterations across all MPR calls of the simulations.
- Phase 3: One half of the models quit phase 3 and therefore the MPR algorithm after a maximum of $13-17$ iterations. The other half completes after a maximum of $56-69$ iterations. There are two upwards outlier models, one with 87 and the other with 107 maximum iterations to quit.

To summarize, a suitable default value for the iteration limit that covers both phases is `mprIterMax` $= 120$.

[5] Modia3D.jl, v0.10.2, test/Collision/
[6] Intel(R) Core(TM) i7-9850H CPU @ 2.6 GHz, RAM 32 GB
[7] Modia3D.jl, v0.10.2, test/Collision/BouncingSphereFreeMotion.jl
[8] Modia3D.jl, v0.10.2, test/Collision/BouncingEllipsoid.jl

Table 6.1: Maximum number of iterations `i_max` required to complete phase 2 and 3 across all MPR calls of the simulations.

		`i_max` phase 2	`i_max` phase 3
TwoCollidingBalls.jl	Figure B.9	4	14
BouncingBeams.jl	Figure B.2	5	15
BouncingCapsules.jl	Figure B.1	4	16
Rattleback.jl	Figure B.5	5	17
Billard4Balls.jl	Figure B.10	4	56
BouncingEllipsoid.jl	Figure B.8	5	57
BouncingFrustums.jl	Figure B.4	4	61
BouncingSphereFreeMotion.jl	Figure B.7	5	61
BouncingCones.jl	Figure B.3	5	69
CollidingSphereWithBunnies.jl	Figure B.11	7	87
BouncingEllipsoidOnSphere.jl	Figure B.6	4	107

MPR Termination Tolerance

The MPR algorithm is terminated if (TC2) or (TC3) is less or equal than the MPR termination tolerance `mprTol`. A closer examination of collision models shows that the MPR termination tolerance has a high impact on the simulation time and performs best with a MPR termination tolerance 10^{-20}.

In Figure 6.8 two models are analyzed in more detail both with simulation tolerance 10^{-8}, without visualization and without plotting. The used solver is CVODE_BDF (variable step, variable order, BDF method). For a brief explanation of the models see Appendix B. For Model 1, CPU time (without initialization) and number of solver iterations are at their minimum for $\texttt{mprTol} = 10^{-20}$. Both increase a bit for a decreasing MPR termination tolerance. The same behavior is shown for values up to $\texttt{mprTol} = 10^{-17}$. For MPR termination tolerance between $10^{-8} - 10^{-16}$ the CPU time and number of solver iterations increase significantly (not shown in Figure 6.8, as the information for smaller tolerances is no longer visible). The maximum number of iterations for phase 3 across all MPR calls decrease for an increasing MPR termination tolerance. In general, Model 2 shows the same behavior, except that for very small MPR termination tolerances, the number of solver iterations decreases, and in return the CPU time increases. The maximum number of required iterations increases as well. In other words, if the MPR termination tolerances are too large, the contact detection becomes too inaccurate and the solver needs more iterations that increases the CPU time. If the MPR termination tolerances are too small, there is an unnecessarily high computational effort in the MPR algorithm that increases the CPU time.

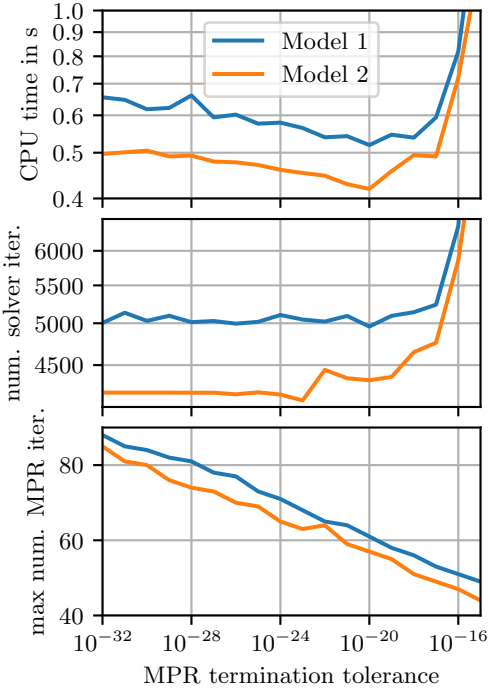


Figure 6.8: Evaluation for different MPR termination tolerances. Model 1 and Model 2 show a quite similar behavior for CPU time, number of solver iterations, and the maximum number of required iterations across all MPR calls to quit phase 3.

To sum up, the default value for the MPR termination tolerance $\texttt{mprTol} = 10^{-20}$ seems to be a good trade-off between CPU time, number of solver iterations calls, and maximum number of required iterations across all MPR calls of the simulations. Executing the MPR algorithm with quadruple precision increases robustness, so variable-step solvers are less likely to fail and it speeds up collision simulations.

6.1.5 Overview of Improvements to the MPR Algorithm

The enhancements of the MPR algorithm in 3D are already published in Neumayr and Otter (2017) and are implemented in Modia3D.jl. In this thesis, the following debate is more extended. The improvements to the discussed MPR algorithm in 3D are based on publications of Snethen (2008), Olvång (2010) and Kenwright

(2015). Versions by Snethen (2008) and Olvång (2010) quit the algorithm when they determine that two objects do not overlap and therefore do not collide. The improved MPR algorithm as well as the version by Kenwright (2015) iterate until the minimal Euclidean distance of shapes that are not in contact or minimal penetration depth of shapes that are in contact is found.

Since the MPR algorithm is numerical sensitive it is performed with quadruple precision (significant 106 bits) (JuliaMath, online) to increase robustness and reliability. Therefore, a small tolerance `mprTol`, e.g., 10^{-20} for terminating the algorithm is possible (see Section 6.1.4).

As mentioned in Section 4.2, the search direction does not need to be of unit length, but in the improved MPR algorithm it does. Therefore, to avoid divisions by zero, some new checks are introduced. These situations are considered in more detail below.

The first check detects if the centroids of both objects coincide. In some collision cases e.g., two spheres are colliding, the signed distance can be computed directly because support point $\boldsymbol{r}_1$ and the interior point $\boldsymbol{r}_0$ are aligned on the origin ray. Thus, termination condition (TC1) is introduced.

Furthermore, to avoid unphysical behavior, the improved MPR algorithm also verifies whether both collision objects have degenerated to a 2D geometry or less e.g., 2D surface, line, point or a very thin 3D geometry. Thus, the resulting Minkowski Difference has also degenerated into a 2D geometry or less. To adequately address such a situation, a 2D MPR algorithm (Snethen, online[b]) would be required. However, due to the termination condition (TC1), there are cases that give a result even in a 2D case. In the context of this thesis only penetration depths in 3D models are regarded.

For terminating the third phase of the MPR algorithm, termination condition (TC2) is introduced if the new support point $\boldsymbol{r}_4$ is located on the same ray as the search direction. So, no better point can be found for refining the portal. Additionally, termination condition (TC3) is improved. Furthermore, it is checked if support point $\boldsymbol{r}_4$ is located in the portal plane or on one of its edges. If the refinement fails or, to avoid infinite looping and the maximum number of iterations is reached, the MPR algorithm is terminated with the best found result so far. Finally, all improvements of the MPR algorithm are discussed in detail.

6.2 Properties of Improved MPR Algorithm

This section discusses properties of the improved MPR algorithm. In Section 6.2.1, two new theorems are introduced based on the circumstance that the improved MPR algorithm also computes the Euclidean distance between non-penetrating shapes. In Section 6.2.2, an approach for small penetrations is established that avoids discontinuous jumps for contact points for continuously moving objects with

respect to time. Based on this approach, an additional definition of penetration depth is given for small penetrations.

6.2.1 Theorems Concerning Improved MPR Algorithm

Neumayr and Otter (2017) have summarized these properties in the following new theorems, which have been further extended in this thesis for (TC2).

Definition 6.4: ($\delta_{\mathbf{mpr}}$). Let $A, B \in \mathcal{C}, A \neq B$ be two convex objects. Assume that computations are performed with infinite precision ($\texttt{mprTol} = 0$, $\texttt{mprIterMax} = \infty$), then the signed distance returned by function `mpr` (Line 295, Page 61) is

$$\delta_{\text{mpr}}(A, B) = \texttt{mpr(A, B; mprTol = 0, mprIterMax = } \infty\texttt{)}.$$

Theorem 6.1 is sufficient so that δ_{mpr} can be used as zero-crossing function for a variable-step solver in Section 6.3.

Theorem 6.1: ***(Contact detection 1).*** *(Neumayr and Otter, 2017)*
Let $A, B \in \mathcal{C}, A \neq B$. For δ_{mpr} with infinite precision the following holds:

1. *$\delta_{mpr} > 0$: A and B are not in contact with each other.*
2. *$\delta_{mpr} = 0$: A and B are touching each other.*
3. *$\delta_{mpr} < 0$: A and B are penetrating each other.*

Proof: This proof is based on Neumayr and Otter (2017). For this proof, properties of the Minkowski Difference are used. From Theorem 4.4 directly follows that two objects A and B penetrate iff $A \ominus B$ contains the origin. Moreover, if they are touching, the origin is on the boundary $\mathbf{0} \in \partial(A \ominus B)$ (see Definition 4.10). Two objects A and B are not in contact iff the origin is not included $\mathbf{0} \notin A \ominus B$. If function `mpr` terminates successfully, the relationship between the origin ray (Definition 6.2) from $\boldsymbol{r}_0$ to the origin through the final portal is always well-defined and the collision situation follows automatically:

Assume function `mpr` terminates due to (TC1): In this case $\boldsymbol{r}_1$ is on the origin ray from $\boldsymbol{r}_0$ through the origin. $\boldsymbol{r}_1$ is the farthest support point that still lies on the boundary of Minkowski Difference. Therefore, if the origin is between $\boldsymbol{r}_0$ and $\boldsymbol{r}_1$, then the origin is in the Minkowski Difference $\mathbf{0} \in A \ominus B$. It follows that objects A and B are penetrating and $\delta_{\text{mpr}} < 0$. If $\boldsymbol{r}_1 = \mathbf{0}$, the objects are touching and $\delta_{\text{mpr}} = 0$ because the origin is on the boundary of $A \ominus B$. If $\boldsymbol{r}_1$ is between $\boldsymbol{r}_0$ and the origin, then the origin is outside $\mathbf{0} \notin A \ominus B$. It follows that objects A and B are not in contact and $\delta_{\text{mpr}} > 0$.

Assume function `mpr` terminates due to (TC2): In this case $\boldsymbol{r}_4$ is the farthest support point on the boundary of the Minkowski Difference in the direction of the origin ray. No further support point is possible because the portal cannot be refined any further. If the origin lies between $\boldsymbol{r}_0$ and $\boldsymbol{r}_4$, then the ray from $\boldsymbol{r}_0$ passes first through the origin and then through $\boldsymbol{r}_4$. Therefore, the origin is inside $\mathbf{0} \in A \ominus B$,

$\delta_{\text{mpr}} < 0$ and objects A and B are penetrating. If $\boldsymbol{r}_4 = \mathbf{0}$ the objects are touching and $\delta_{\text{mpr}} = 0$ because the support point $\boldsymbol{r}_4$, which is the origin, is on the boundary of $A \ominus B$. If $\boldsymbol{r}_4$ lies between the origin and $\boldsymbol{r}_0$, then the origin is outside $\mathbf{0} \notin A \ominus B$. It follows that objects A and B are not in contact and $\delta_{\text{mpr}} > 0$.

Assume function `mpr` terminates due to (TC3): In the case of (TC3), no further support point lies beyond the portal plane in the direction of the origin ray. If the origin is on the same side of the plane as $\boldsymbol{r}_0$, then the ray from $\boldsymbol{r}_0$ passes first through the origin and then through the portal. Therefore, the origin is inside $\mathbf{0} \in A \ominus B$, $\delta_{\text{mpr}} < 0$ and objects A and B are penetrating. If the origin is in the plane, then the origin ray from $\boldsymbol{r}_0$ passes the origin. This is only possible if the origin is on the portal. Since (TC3) implies that the portal is on the boundary, the origin is on the boundary of $A \ominus B$ and the objects are touching $\delta_{\text{mpr}} = 0$. If the origin is on the opposite side of the portal plane than $\boldsymbol{r}_0$, the origin is outside $\mathbf{0} \notin A \ominus B$, so that $\delta_{\text{mpr}} > 0$ and objects A and B are not in contact. ■

Theorem 6.2 is a slight extension of Theorem 6.1 Item 1 and additionally considers δ_{broad} (see Definition 5.1) between two non-overlapping AABBs.

Theorem 6.2: ***(Contact detection 2).*** *(Neumayr and Otter, 2019b)*
Let $A, B \in \mathcal{C}, A \neq B$. For δ_{mpr} with infinite precision and δ_{broad} which is the Euclidean distance between two non-overlapping AABBs, the following holds:

1. *$\delta_{broad} > 0$ or $\delta_{mpr} > 0$: A and B are not in contact with each other.*

Proof: This proof is based on Neumayr and Otter (2019b). For $\delta_{\text{broad}} > 0$ two objects cannot be in contact with each other because their AABBs are not in contact with each other. ■

While Theorem 6.2 defines the collision situation for δ_{broad} and δ_{mpr}, the following theorem defines the relationship between δ_{mpr} (Definition 6.4) and δ_{d} (Definition 4.8). Furthermore, Theorem 6.3 states that the MPR algorithm either returns the closest distance of two non-penetrating objects or an upper bound on the closest Euclidean distance.

Theorem 6.3: ***(The closest distance).*** *(Neumayr and Otter, 2017)*
Let $A, B \in \mathcal{C}, A \neq B$. If $\delta_{mpr} > 0$ with infinite precision, the following holds:

1. *If termination occurred via (TC1) or via (TC2) or via (TC3) where $\boldsymbol{r}_4$ calculated with function `distanceToPortal` (Line 441, Page 70) is inside the portal triangle, then*

$$\delta_{mpr} = \delta_d.$$

2. *If termination occurred via (TC3) and $\boldsymbol{r}_4$ is located on one of the edges or vertices of the portal triangle, then*

$$0 < -\boldsymbol{r}_4 \cdot \boldsymbol{e}_4 \leq \delta_d \leq \delta_{mpr}.$$

Proof: This proof is based on Neumayr and Otter (2017).

Assume 1. holds: In the case of (TC1): $\boldsymbol{r}_1$ is the closest point to the origin and it is parallel to the normalized search direction. In the case of (TC2): $\boldsymbol{r}_4$ is the closest point to the origin and it is parallel to the normalized search direction. In the case of (TC3): $\boldsymbol{r}_4$ is inside the portal. Due to proof of Theorem 6.1, it is the closest point to the origin and it is parallel to the normalized search direction. Since these vectors are in parallel to their normalized search direction, their absolute value is the closest distance, so $\delta_{\text{mpr}} = \delta_{\text{d}}$.

Assume 2. holds: In direction of the origin ray no support point is beyond the portal plane. On the one hand, a lower bound for the closest distance is the projection of $\boldsymbol{r}_4$ to the plane normal: $-\boldsymbol{r}_4 \cdot \boldsymbol{e}_4 \leq \delta_{\text{d}}$. On the other hand, the closest distance δ_{d} cannot be greater than $\|\boldsymbol{r}_4\| = \delta_{\text{mpr}}$. ■

6.2.2 Ensuring Continuous Penetration Depth and Contact Points

For simulating colliding objects, the penetration depth needs to be computed for subsequent points in time. In order to avoid numerical issues in the upcoming chapter (see Chapter 7), the penetration depth, contact normal and contact points must evolve continuously with respect to time.

In three situations, the penetration depth, especially contact points, and contact normal jump discontinuously for continuously dynamically moving objects. In the first situation, both objects are largely overlapping, like a bullet penetrating into an object (see Figure 6.9). Second, for non-smooth objects discontinuous jumps occur on edges and vertices (see Figure 6.10). Both situations are discussed in the following section and an approach for small penetrations is given. Third, a contact area instead of a contact point emerges, e.g., one face of a box collides in parallel with another face of a box. This situation cannot be handled appropriate e.g., with the MPR algorithm, for a more extensive discussion see Section 8.3.

Discontinuities due to Definition of Translational Penetration Depth

The conventionally used definition of penetration depth δ_{p} (Definition 4.9) is based on pure geometric properties. It is the distance of the shortest vector over which the objects need to be translated to bring them into touching contact. The drawback of this definition is that dynamic movement of penetrating objects can lead to unphysical discontinuities in contact points, contact normal and penetration depth e.g., like a bullet penetrating into an object or dynamic collisions on vertices or edges.

Figure 6.9 depicts a situation inspired by a bullet penetrating into an object. It is assumed that, object B moves from left to right, which is the direction of motion (dm), and it penetrates object A. If the penetration depth in direction of motion $\delta_{\text{dm}} \geq 0$ gets large enough, the (translational) penetration depth δ_{p} is no longer aligned in direction of motion and is physically wrong. Furthermore, contact points and contact normals jump discontinuously.

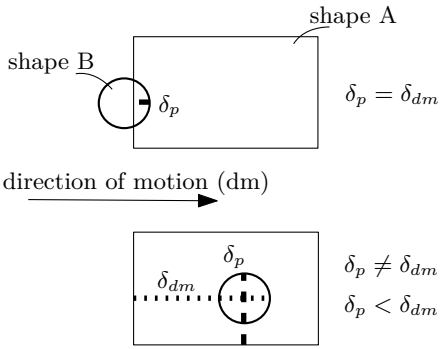


Figure 6.9: Discontinuities in penetration depth computation. Object B penetrates object A in direction of motion. Discontinuity occurs if $\delta_\mathrm{p} \neq \delta_\mathrm{dm}$.

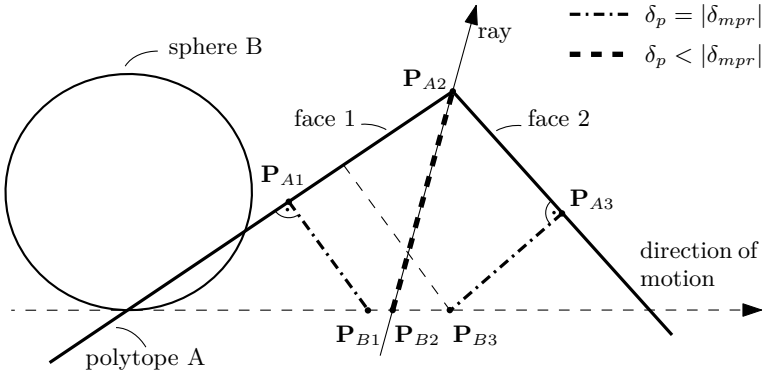


Figure 6.10: Sphere B penetrates polytope A in direction of motion. The occurrence of discontinuous penetration depth is analyzed in more detail using three contact translations.

Discontinuities due to (Improved) MPR Algorithm

In general, for non-smooth objects discontinuous jumps of penetration depth occur on edges and vertices. The occurrence of discontinuous penetration depth is analyzed in more detail for the improved MPR algorithm using three contact translations (see Figure 6.10). Sphere B is sliding in direction of motion from left to right along the dashed arrow and penetrates polytope A, passing three contact translations with its resulting contact points $(\boldsymbol{P}_{A1}, \boldsymbol{P}_{B1}), (\boldsymbol{P}_{A2}, \boldsymbol{P}_{B2}), (\boldsymbol{P}_{A3}, \boldsymbol{P}_{B3})$.

Imagine, the lower point of sphere B is on contact point $\boldsymbol{P}_{B1}$, then the penetration

depth is the orthogonal distance to face 1 of polytope A and $\delta_\mathrm{p} = |\overrightarrow{\boldsymbol{P}_{A1}\boldsymbol{P}_{B1}}| = |\delta_\mathrm{mpr}|$, which in turn is the distance computed by the MPR algorithm. To illustrate discontinuity, in Figure 6.10 the contact point $\boldsymbol{P}_{B3}$ of sphere B is chosen, even though the jump occurs slightly earlier. Point $\boldsymbol{P}_{B3}$ is a point where the distance to face 2 is smaller than the distance to face 1. Then the contact point on polytope A jumps discontinuously from face 1 to face 2 and $\delta_\mathrm{p} = |\overrightarrow{\boldsymbol{P}_{A3}\boldsymbol{P}_{B3}}| = |\delta_\mathrm{mpr}|$.

The improved MPR algorithm introduces a slightly different discontinuity: If the centroids of polytope A and sphere B are on the origin ray, the MPR algorithm terminates due to termination condition (TC1). This occurs, if the lower point of sphere B is on point $\boldsymbol{P}_{B2}$, which also lies on the ray, and therefore $|\delta_\mathrm{mpr}| = |\overrightarrow{\boldsymbol{P}_{A2}\boldsymbol{P}_{B2}}| > \delta_\mathrm{p}$. Hence, the discontinuous jump from face 1 to face 2 occurs earlier. The contact point on the boundary of polytope A as well as δ_mpr changes discontinuously if the lower point of sphere B touches $\boldsymbol{P}_{B2}$.

Ensuring Continuous Penetration Depth and Contact Points

To ensure continuous penetration depth, Zhang, Kim, and Manocha (2014) propose an algorithm where the penetration depth and contact points are continuous with respect to the motion parameters, by taking the movement of the contact points and of past geometric properties into account.

In this thesis, the focus is on collision situations with small penetration depths. Hence, largely overlapping objects are not considered. Furthermore, to avoid discontinuous jumps on edges and vertices smoothing radii are introduced. Neumayr and Otter (2017) have already discussed this approach, which is inspired by Bergen (2003). Collision smoothing radii, with tiny radii r_A and r_B for objects A and B respectively, are introduced for all non-smooth shapes (e.g., box, cylinder, cone, frustum of a cone, and beam). This affects the support mapping computations of Section 4.2. This procedure is performed in two steps for non-smooth shapes.

- The shape is scaled in the support mapping computation. This means it is shrunk by its collision smoothing radius. It is examined for the box in Lines 473 to 478.
- The related smoothing radius is added in search direction again in Lines 479 to 480. This leads to smoothed edges as illustrated in Figure 6.11.

```
472 # support point mapping for box with smoothing radius
473 function supportPoint(obj::Box, n)
474     x = 1/2*obj.lengthX - obj.smoothingRadius
475     y = 1/2*obj.lengthY - obj.smoothingRadius
476     z = 1/2*obj.lengthZ - obj.smoothingRadius
477     return (sign(n[1])*x, sign(n[2])*y, sign(n[3])*z)
478 end

479 supportMapping(obj::Box, n) =
480     obj.r + obj.R'*supportPoint(obj, obj.R*n) + n*obj.smoothingRadius
```

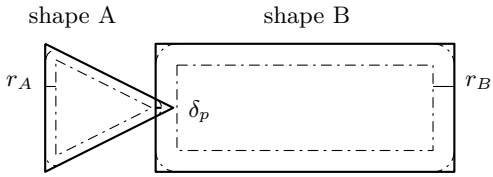


Figure 6.11: Shapes A and B, both with a smoothing radius r_A, r_B are penetrating with δ_p (Neumayr and Otter, 2017).

In Modia3D the user can set a `smoothingRadius` value. To avoid dimension problems, the smoothing value is the minimum of the user's value and 10% of the smallest shape lengths. For no smoothing, the value is set to 0. All support mappings with smoothing radius can be found online[9].

For small penetration depths and smooth collision shapes a relationship between δ_mpr (Definition 6.4) and δ_p (Definition 4.9) is defined.

Definition 6.5: (Penetration depth 2). (Neumayr and Otter, 2019b)
Let $A, B \in \mathcal{C}, A \neq B$ an intersecting pair and $\delta_\mathrm{p} < r_A + r_B$ where $r_A, r_B \in \mathbb{R}_0$ are collision smoothing radii. The penetration depth δ_p is the signed distance $\delta_\mathrm{mpr}(A, B) \leq 0$ computed by the MPR algorithm in the narrow phase with infinite precision

$$\delta_\mathrm{p}(A, B) = |\delta_\mathrm{mpr}(A, B)| \geq 0. \tag{6.2}$$

Assume, in Figure 6.11, shape A's smoothed edge continuously moves over shape B's smoothed edge, and $\delta_\mathrm{p} < r_A + r_B$, then the penetration depth $\delta_\mathrm{p} = |\delta_\mathrm{mpr}|$ is continuous. Still, there are collision situations that cannot be handled with an algorithm that computes point-contacts, due to the loss of contact information. For example, collisions between two parallel surfaces cannot be handled adequately. For a detailed discussion see Section 8.3. Nevertheless, sufficient collision situations can be handled.

The approach presented here is not applicable for all collision situations. It depends on the collision shapes, their relative position and their relative orientation. Furthermore, it is limited to small penetrations and continuously moving objects with respect to time. For a detailed discussion to this topic see "Conclusion to Collision Handling with Variable-Step Solvers (Section 8.3)". This approach and the upcoming section dealing with zero-crossing functions for collision handling with variable-step solvers are a preparation for "Collision Response (Chapter 7)".

[9] Modia3D.jl, v0.10.4, src/Shapes/boundingBoxes.jl

6.3 Zero-Crossing Functions for Collision Handling

The time of a collision is not known in advance, but the model behavior and its resulting forces are changing drastically when a collision occurs. In order to determine the event time, event conditions are implicitly specified in form of zero-crossing functions, see Section 2.4, and are detected with variable-step solvers.

In this thesis, for computing the collision response the penetration depth is needed in Section 7.1. Neumayr and Otter (2017) and Neumayr and Otter (2019b) propose to use the zero crossing of the signed distance δ_{mpr} (Definition 6.4) computed by the improved MPR algorithm as an event condition. Variable-step solvers evaluate zero crossings, i.e., the time when objects touch and trigger an event. This is suitable for systems where many convex objects, or their convex hulls, may potentially collide. Theorem 6.1 and its extension to Theorem 6.2 are sufficient so that function z_{AB} (6.3) can be used as zero-crossing function.

Definition 6.6: (Zero-crossing function for collision handling). (Neumayr and Otter, 2019b)
Let $A, B \in \mathcal{C}, A \neq B$ be two potentially colliding convex objects, or their convex hull, and time $t \in \mathbb{R}_0$. Furthermore, the translations and rotations of objects A, B are time-dependent. Then, function z_{AB}

$$z_{\text{AB}}(t) = \begin{cases} \delta_{\text{broad}}(A,B) & \text{if } \delta_{\text{broad}} > 0 \\ \delta_{\text{mpr}}(A,B) & \text{if } \delta_{\text{broad}} \leq 0 \end{cases}, \quad A, B \in \mathcal{C}, A \neq B, \tag{6.3}$$

is called zero-crossing function with $z_{\text{AB}} = z_{\text{BA}}$. Where $\delta_{\text{broad}}(A,B)$ (Definition 5.1) is the Euclidean distance between two non-overlapping AABBs calculated in the broad phase. Furthermore, $\delta_{\text{mpr}}(A,B)$ is the signed distance computed by the MPR algorithm in the narrow phase (Definition 6.4).

Figure 6.12 shows a zero-crossing function. The zero crossing $z_{\text{AB}}(t) = 0, t = t_{\text{ev}}$ occurs if both objects touch, then t_{ev} is the transition or event time. For $z_{\text{AB}} > 0$ both objects are not in contact and for $z_{\text{AB}} < 0$ both objects penetrate.

It is standard for variable-step solvers with zero-crossing support to evaluate a zero-crossing function $z_{\text{AB}}(t)$ at time instant $t_i + h$ of a completed solver step with step size h. If $z_{\text{AB}}(t_i) \cdot z_{\text{AB}}(t_i + h) \leq 0$ an interval $[t_i, t_i + h]$ is determined in which z_{AB} crosses zero. Several algorithms with guaranteed convergence are known that reduce the interval in which the zero crossing occurs until a specified tolerance for the final interval is reached. For example, the solvers of the Sundials suite use a modified secant method. The DASSL solver uses the method of Brent (1973). In both cases, the interval is successively decreased in each iteration. Here the only required property is the sign of z_{AB}, which is provided by (6.3) according to Theorem 6.2. In case of the method of Brent (1973), convergence is super-linear, if $z_{\text{AB}}(t)$ is smooth.

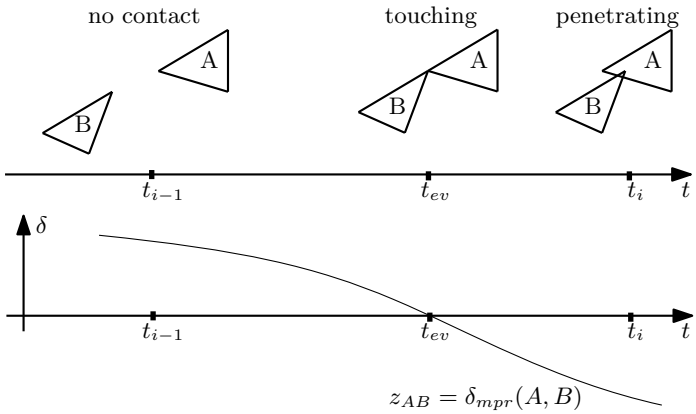


Figure 6.12: Signed distance, compound of distance and penetration depth, between two objects A, B are used as zero-crossing function z_{AB}. The zero crossing $z_{\mathrm{AB}}(t) = 0$ at event instant t_{ev}, where both objects are touching.

Methods treating Zero-Crossing Functions for Collision Handling

The three methods for treating zero-crossing functions used for collision handling require the distances between the collision objects and are considered below. First, a brute force method would be to use the distances between any two collision objects as zero-crossing functions. However, this approach is not practical for a larger number of objects $n_{\mathcal{C}}$ because $O(\dim(\boldsymbol{z})) = O(n_{\mathcal{C}}^2)$. The number of zero-crossing functions, as well as the number of distance computations would grow quadratically with the number of objects, that can potentially collide.

The second method avoids a quadratic growth of the zero-crossing functions. Neumayr and Otter (2017) suggest limiting the number of zero-crossing functions the solver must process. This upper bound $n_{\max}$ of zero-crossing functions is set by the user. Thus, at most $n_{\max}$ collision pairs can be in contact at the same time instant. If more objects get in contact, the simulation is halted with an error, or alternatively, the simulation is halted and restarted with an enlarged $\boldsymbol{z}$ vector. This approach has the disadvantage that an upper bound $n_{\max}$ on the number of zero-crossing functions must be defined by the user, and the algorithm for tracking the smallest $n_{\max}$ values is complicated.

The third method simplifies the idea of Neumayr and Otter (2017) by using only two zero-crossing functions. This idea is preliminarily published by Neumayr and Otter (2019b) and is discussed in the following. For this method the two functions

$$z_{\mathrm{d}} = \min_{A,B\in\mathcal{C},A\neq B}(z_{\mathrm{AB}} \ : \ \forall \text{ pairs } A, B \text{ separated at last event}), \tag{6.4a}$$

$$z_{\mathrm{p}} = \max_{A,B\in\mathcal{C},A\neq B}(z_{\mathrm{AB}} \ : \ \forall \text{ pairs } A,B \text{ penetrated at last event}), \tag{6.4b}$$

are used as zero-crossing functions. Hereby, (6.4a) is the minimum of all zero-crossing functions (6.3) of collision pairs which were not in contact at the last event instant. Therefore, a zero crossing of z_{d} occurs when at least one pair of objects that was not in contact at the last event changes from positive to negative values. Hereby, (6.4b) is the maximum of all zero-crossing functions (6.3) of collision pairs which were in contact at the last event instant. Therefore, a zero crossing of z_{p} occurs when at least one pair of objects that was in contact during the last event separates and thus changes from negative to positive values.

The number of zero-crossing functions is reduced in the second and third methods compared to the first method. The quadratic growth of distance calculations cannot be avoided with all three methods, since the distances and penetration depths between collision objects are used as zero-crossing functions.

Implementation of Zero-Crossing Functions for Collision Handling

The implementation in Modia3D uses basically one dictionary, called `contactDict`. This dictionary stores each collision `pair`, which has been in contact at the last event instant in order to decide whether a computed zero-crossing function (6.3) of two arbitrary objects $A,B \in \mathcal{C}, A \neq B$ should be used in z_{p} or in z_{d}. Each collision `pair` has a unique `pairID`, which is used to identify collision pairs in dictionary `contactDict`. A collision `pair` holds information about contact points on each object, the contact normal, both penetrating objects, and the distance with hysteresis.

Furthermore, the following three internal functions[10] are used for updating both zero-crossing functions (6.4). Before that, distances between all potentially colliding objects are computed in a broad and if necessary in a narrow phase.

- `selectZeroCrossings!`: This function is called at every event instant and computes (6.3) for all potential collision pairs. The actual penetrating pairs are detected and stored in `contactDict`. This selection is kept until `selectZeroCrossings!` is called again.
- `updateZeroCrossings!`: This function is called whenever the solver requests a new zero crossing evaluation and computes (6.3) for all potential collision pairs. Furthermore, contact points and contact normals of the collision pairs stored in `contactDict` are updated.
- `getDistances!`: This function is called for evaluation of the equations of motion and at communication points, i.e., for result output. It updates (6.3), contact points and contact normals for all collision pairs stored in `contactDict`.

The difference between these three functions is also pointed out in Julia pseudo code snippet.

[10] As usual in Julia, function names with a `!` at the end indicate that one or more of the input arguments are changed by the function call.

```
481 if isEvent(...)
482     selectZeroCrossings!(...)
483 elseif isZeroCrossing(...)
484     updateZeroCrossings!(...)
485 else
486     getDistances!(...)
487 end
```

Partitioning into Two Sets

The previously discussed procedure is realized by choosing two functions z_d and z_p (6.4) and those are used as zero-crossing functions for the solver. Because of that, all potential collision pairs with their zero-crossing functions z_AB (6.3) are partitioned into two sets. The first set identifies z_d. In order to identify z_d, the distances between separated objects are compared and the minimum value is stored in variable `noContactMinVal` (Lines 503 to 506).

z_d indicates whether at least one zero crossing from positive to negative i.e., from non-penetrating to penetrating takes place. If z_d is negative, a zero crossing occurs. This means, z_d is the minimum value of all distances between objects which are not in `contactDict`.

The second set identifies z_p. In order to identify z_p, all information e.g., contact points on each object, the contact normal, the two penetrating objects, and the distance with hysteresis about penetrating pairs is stored and updated in dictionary `contactDict` (Lines 488 to 502).

z_p indicates whether at least one zero crossing occurs from negative to positive (i.e., from penetrating to non-penetrating) takes place. If z_p is positive, a zero crossing occurs. For detecting that, the maximum value of all distances of collision pairs in `contactDict` is stored in z_p. If the contact set is empty, z_p gets a negative dummy value.

```
488 function updateContactPair!(pair::ContactPair, obj1::Object3D, obj2::Object3D,
489          contactPoint1::Vector, contactPoint2::Vector,contactNormal::Vector,
490          distanceWithHysteresis::Float64)
491     pair.contactPoint1 = contactPoint1
492     pair.contactPoint2 = contactPoint2
493     pair.contactNormal = contactNormal
494     pair.obj1          = obj1
495     pair.obj2          = obj2
496     pair.distanceWithHysteresis = distanceWithHysteresis
497     ...
498 end

499 if contact
500     updateContactPair!(contactDict[pairID], obj1, obj2,
501                        contactPoint1, contactPoint2, contactNormal,
502                        distanceWithHysteresis)
503 else
504     if noContactMinVal > distanceWithHysteresis
```

```
505         noContactMinVal = distanceWithHysteresis
506 end; end
```

Hysteresis for Zero-Crossing Functions

A restart of the integration after an event requires that no zero-crossing function z_{AB} (6.3) is identical to zero, otherwise a zero crossing cannot be detected anymore and solvers invoke an error. However, a zero-crossing function z_{AB} (6.3) might be identically to zero at an event instant e.g., two objects are touching each other at initialization. To avoid this, a hysteresis is added to z_{AB} (6.3). Neumayr and Otter (2019b) realize this similar to FMI Project (2017).

To sum up, the narrow phase and how to calculate the Euclidean distance and penetration depth between two potentially-colliding objects with the MPR algorithm is discussed. This thesis presents an enhanced version of the classical MPR algorithm. The properties of the improved MPR algorithm are summarized in three new theorems. For simulating continuously moving and colliding objects, the penetration depth needs to be computed for subsequent points in time. An approach for small penetration depths that avoids discontinuous jumps in contact points and contact normals is presented. Thus, numerical issues in the elastic-response calculation (see Chapter 7) – the fourth collision handling step – are avoided. Furthermore, the penetration depth respectively the Euclidean distance is used as zero-crossing function. Contact forces and torques are applied if the objects are in contact.

7 Collision Response

The collision detection (collision handling steps 1-3) is a mathematical problem of determining whether and where two geometric objects are intersecting. The collision response known as response calculation is the fourth and final collision handling step. Collision response is a kinetic problem that involves the motion of two or more objects after a collision (Bourg and Bywalec, 2013).

If a collision is detected and collision response is applied, the model's behavior changes drastically. There are several different methods of collision response, each with their own strengths and weaknesses. An overview of methods can be found in Mirtich (1996), Otter, Elmqvist, and López (2005) and Hofmann et al. (2014). The elastic-response calculation is the only one that allows penetrations between objects. It is applicable for rigid body simulations as well as for deformable body simulations. The penetrations need to be small enough, so that they are not noticeable in relationship to the scale of the system. Elastic-response calculation methods vary in details. The basic idea is that between the colliding objects a spring, spring-damper or some other finite force element is present, which generates separating forces. These forces refer to the penetration, i.e., penetration depth or penetration volume between the colliding objects and act at the contact points or contact area. However, finding a satisfactory response characteristic that reflects the physical behavior is challenging. There are different types of elastic-response calculation (Mirtich, 1996; Otter, Elmqvist, and López, 2005; Hofmann et al., 2014).

- Response calculation based on impulses. The collision response is calculated with an idealized approach, applying impulses derived from impact laws like Poisson's hypothesis. The impulses of the compression and decompression phases of an impact are set in relationship to each other. On the one hand, only a few constants are required to describe the impact law. The solver's step size remains unaffected by the response calculation since it occurs in an infinitely small moment of time. On the other hand, idealized impact laws are only valid for stiff collisions. Moreover, the necessary constants cannot be calculated from the material properties of the colliding objects, i.e., they need to be measured. In addition, accurately determining the new initial conditions after an impact presents significant challenges. Especially, when multiple contacts occur simultaneously. There can be either no solutions at all or infinitely many solutions using impulse descriptions. To achieve a physically accurate response, multiple impacts may need to be applied in one of two ways: either sequentially in some cases, or simultaneously in others.

- Response calculation based on penetration depth. It uses simple elastic spring-damper elements, e.g., the spring force is proportional to the penetration depth. This straightforward approach can be used for stiff and soft contacts. It performs reasonably well with several simultaneous contact points. The solver's step size is significantly reduced in the contact phase to capture the rapidly changing contact forces and torques. Spring and damper constants are determined experimentally or calculated from material properties.
- Response calculation based on contact area and penetration volume. The contact force is not only proportional to the penetration depth, but also to the discretized contact area and the discretized penetration volume. The force and torque calculation is more accurate and the material properties are used to calculate the spring constants. In addition, the contact torque can be reasonably calculated.
- Response calculation for special collision situations. The collision response is calculated for special collision situations, e.g., for wheel/road contacts, wheel/rail contacts, and bearing contacts. Specialized approaches tailored to specific contact problems typically offer greater accuracy and applicability than generic solutions.

This thesis introduces an elastic-response calculation method for rigid-body simulation based on penetration depth. It considers several physical aspects and elastic material laws (see Section 7.1). Consequently, solid material properties must be assigned to each collision object. Calculating an appropriate damping factor from the coefficient of restitution requires the initial relative velocity at the start of penetration. Therefore, an event at the start and end of a penetration is mandatory for this approach. Starts and ends of penetrations are identified with zero-crossing functions taking the signed distance into account. The signed distance consists of the penetration depth and the Euclidean distance between the colliding objects. To ensure a physically realistic collision response, the penetration depth, contact normal, and contact points must evolve continuously with respect to time. This condition can be ensured for the elastic-response calculation, as it primarily deals with small penetration depths. Moreover, smoothing the shapes prevent discontinuous jumps at edges and vertices. Consequently, each surface point has a unique normal vector. Furthermore, continuous movement of a surface point induces continuous movement to its corresponding normal vector.

7.1 Force Law

The theory of Hertz (1896) is a nonlinear force law for perfectly elastic and elliptic objects. The force law depends on the penetration depth δ_p and leads to

$$f = \frac{4}{3} E^\star R_\mathrm{geo}^{\frac{1}{2}} \delta_\mathrm{p}^{\frac{3}{2}}, \tag{7.1}$$

where

$$E^\star = \frac{1}{\frac{1-\nu_1^2}{E_1} + \frac{1-\nu_2^2}{E_2}}. \tag{7.2}$$

E_1, E_2 are Young's moduli and ν_1, ν_2 are the Poisson's ratios of the associated objects (see Section 7.1.1). Additionally, R_{geo} is a geometry dependent coefficient (see Section 7.1.2).

Furthermore, to consider energy loss that occurs in mechanical models several different formulations are available which are using a damping coefficient and initial penetration velocity. Based on a comparison by Skrinjar, Slavič, and Boltežar (2018), the approach of Flores et al. (2011) for the introduced response characteristics (7.3) is taken.

Neumayr and Otter (2019b) already published the resulting novel response characteristic for elastic contacts (7.3), see Figure 7.1. A contact force f is calculated using penetration depth δ_{p}, penetration velocity $\dot{\delta_{\text{p}}}$, and various coefficients. The contact force is composed of a normal force $\boldsymbol{f}_n$ (7.3b) and a tangential force $\boldsymbol{f}_t$ (7.3c). Additionally, a torque $\boldsymbol{\tau}_\omega$ (7.3d) counteracting the relative angular velocity is applied to describe the rolling resistance.

$$f = k_{\text{red}} \max\left(0, \frac{4}{3} E^\star R_{\text{geo}}^{\frac{1}{2}} \delta_{\text{p}}^{\frac{3}{2}} \left(1 + d\dot{\delta_{\text{p}}}\right)\right) \tag{7.3a}$$

$$\boldsymbol{f}_n = f \boldsymbol{e}_n \tag{7.3b}$$

$$\boldsymbol{f}_t = -\mu_k f \boldsymbol{e}_{t,\text{reg}} \tag{7.3c}$$

$$\boldsymbol{\tau}_\omega = -\mu_r R_{\text{geo}} f \boldsymbol{e}_{\omega,\text{reg}} \tag{7.3d}$$

$\boldsymbol{f}_n$	Contact force in normal direction.
$\boldsymbol{f}_t$	Contact force in tangential direction.
$\boldsymbol{\tau}_\omega$	Contact torque.
$\boldsymbol{e}_n$	Unit vector normal to the contacting surfaces.
$\boldsymbol{e}_{t,\text{reg}}, \boldsymbol{e}_{\omega,\text{reg}}$	Regularized direction vectors in direction of the relative tangential and relative angular velocity (see (7.8), (7.10) in Section 7.1.3).
δ_{p}	Penetration depth (see Definitions 4.9 and 6.5).
$\dot{\delta_{\text{p}}}$	Penetration velocity (see (7.6) in Section 7.1.3).
μ_k, μ_r	Kinetic/sliding friction force and rotational resistance torque coefficient (see Section 7.1.1).
R_{geo}	Geometry dependent coefficient (see Section 7.1.2).
$E^\star$	Combined Young's moduli and Poisson's ratios (see (7.2)).
d	Damping coefficient (see (7.13) in Section 7.1.4).
k_{red}	Elastic contact reduction factor (see Section 7.1.5).

All coefficients, regularizations, constants, and factors needed for the response characteristic are discussed in the remainder of this section.

In order to always guarantee a positive compressive force and to avoid the unphysical behavior of a pulling force, the normal force f is clamped at 0 (7.3a) (Otter, Elmqvist, and López, 2005).

A short comparison points out the differences of the response characteristics (7.3) to the literature:

- In theory, if both shapes are not-curved (like box and beam) a contact surface can occur. Such a contact surface cannot be described by the MPR algorithm because it only computes point contacts. This is tackled by an artificial geometry dependent coefficient R_{geo} based on each shape's parameters is introduced (for an extended discussion see Section 7.1.2).
- To avoid divisions by zero, direction vectors, the contact start velocity, and the coefficient of restitution are regularized. Therefore, a regularization function with a regularization threshold are introduced (see Section 7.1.3).
- The damping coefficient uses the regularized initial velocity and the regularized coefficient of restitution to avoid an unphysical strong creeping effect (see Section 7.1.4).

An illustrative explanation of how the forces and torques act for two penetrating objects is shown in Figure 7.1. It shows the relationship between the forces $\boldsymbol{f}_n, \boldsymbol{f}_t$, torque $\boldsymbol{\tau}_\omega$ and unit vectors: $\boldsymbol{e}_n$ in normal direction and $\boldsymbol{e}_t, \boldsymbol{e}_\omega$ in direction of the respective relative movement. The intuition is that there is a contact surface with a certain pressure distribution in the normal direction and a stress distribution in the tangential direction. The response characteristics provides an approximation of the resultant normal force $\boldsymbol{f}_n$, resultant tangential force $\boldsymbol{f}_t$, and resultant contact torque $\boldsymbol{\tau}_\omega$. The MPR algorithm calculates an approximation of the contact point, of the penetration depth and of a unit vector $\boldsymbol{e}_n$ that is orthogonal to the contacting surfaces.

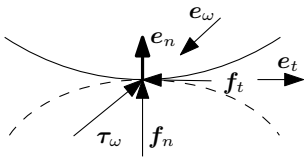


Figure 7.1: Contact normal force $\boldsymbol{f}_n$, contact tangential force $\boldsymbol{f}_t$ (= sliding friction force) and contact torque $\boldsymbol{\tau}_\omega$ between two penetrating objects. $\boldsymbol{e}_n, \boldsymbol{e}_t, \boldsymbol{e}_\omega$ are unit vectors in direction of the respective relative movement.

7.1.1 Solid Material and Material Constants of the Collision Pairs

For collision response an object needs solid material properties to describe the physical behavior in collision situations between a collision pair (see Definitions 4.2, 4.4 and 4.5 and Remark 4.1).

Solid Material Constants

E Young's modulus of solid material in $\mathrm{N\,m^{-2}}$.

ν Poisson's ratio of solid material ($0 < \nu < 1$).

Material Constants of the Collision Pairs

cor Coefficient of restitution ($0 \leq cor \leq 1$). An ideal inelastic collision is defined with $cor = 0$ and an ideal elastic collision with $cor = 1$. The coefficient of restitution is regularized in (7.12) and used for the damping coefficient (7.13).

μ_k Kinetic/sliding friction force coefficient ($\mu_k \geq 0$).

μ_r Rotational resistance torque coefficient ($\mu_r \geq 0$). Its effect is that torque $\boldsymbol{\tau}_\omega$ is computed to reduce the relative angular velocity $\boldsymbol{\omega}_{\mathrm{rel}}$ between the two objects. μ_r can be interpreted as the rolling resistance coefficient if a sphere is rolling on a plane. For simplicity, this coefficient can also be used for a drilling slip.

For each collision object a solid material such as "Steel" or "DryWood" must be assigned, which defines the material parameters E and ν. In addition, for each collision pair that may occur during a simulation their material combinations (e.g., "Steel, Steel" or "Steel, DryWood") must be defined, too. Furthermore, the material constants of these collision pairs (cor, μ_k, μ_r) must be defined as well.

7.1.2 Geometry Dependent Coefficients

The collision response of a collision pair depends on their geometries also known as shapes (see Definition 4.1). The physical behavior is reflected in the coefficient R_{geo}. These coefficients are computed approximately based on the contact theory of Hertz (1896). It is assumed that each of the contacting surfaces can be described by a quadratic polynomial in two variables, defined essentially by its principal curvatures along two perpendicular directions at the point of contact. A characteristic feature is that the penetration volume increases nonlinearly with the penetration depth, unless the two contact surfaces are completely flat. Therefore, the normal contact force changes nonlinearly with the penetration depth. In general, elliptic integrals must be solved, as well as a nonlinear algebraic system of equations to calculate the normal contact force as a function of penetration depth and principal curvatures at the contact point. Antoine et al. (2006) propose an approximate analytical model that replaces elliptic integrals by existing polynomial approximations.

For a numerical integration algorithm with step size control to work reasonably, the contact force must be continuous and continuously differentiable with respect

Table 7.1: Contact radius r_c for different shapes. Additionally, their flatness is given.

Shape	is flat	r_c
Sphere	no	$1/2$ diameter
Capsule	no	$1/2$ diameter
Cone	no	$1/4$ (diameter + topDiameter)
FileMesh	no	$1/2$ shortestEdge
Cylinder	no	$1/2$ min(diameter, length)
Ellipsoid	no	$1/2$ min(lengthX, lengthY, lengthZ)
Box	yes	$1/2$ min(lengthX, lengthY, lengthZ)
Beam	yes	$1/2$ min(length, width, thickness)

Table 7.2: Geometry dependent coefficient R_{geo}.

Shape 1 is flat	Shape 2	R_{geo}
no	yes	$r_{c,1}$
yes	no	$r_{c,2}$
no	no	$r_{c,1}r_{c,2}/(r_{c,1}+r_{c,2})$
yes	yes	$r_{c,1}r_{c,2}/(r_{c,1}+r_{c,2})$

to the penetration depth. This in turn means that the principal curvatures of the contacting surfaces should also be continuously differentiable, which is usually not the case, apart from exceptional cases such as a sphere or an ellipsoid.

In this thesis, only a very rough approximation for principal curvatures of shapes is used, since their determinations are generally complicated and the shapes often have regions of discontinuous curvatures. The contact area of a shape is approximated by a quadratic polynomial with constant mean principal curvature in all directions and at all points on the shape. In other words, a sphere with constant contact radius r_c is associated with any shape used to calculate the coefficient R_{geo}[1]. A default value for the contact radius r_c is determined based on the available data of each shape in Table 7.1. However, this parameter can be overwritten by the user. This allows to use a rough approximation of Hertz' law not only for spheres. Furthermore, for R_{geo} a distinction between flat (box and beam) and curved shapes is made in Table 7.2. If a flat shape collides with a curved shape, the collision radius $r_{c,i},\ i = 1, 2$ of the curved shape is taken for R_{geo}. Only if both shapes are curved, collision radii of both shapes are considered for R_{geo}. In theory, if both shapes are flat R_{geo} tends to infinity. However, such a contact surface cannot be

[1] Modia3D.jl, v0.8.1, src/Shapes/setCollisionSmoothingRadius.jl

described by the used approach. As a remedy, an artificial radius R_{geo} based on the shape's parameters is introduced. While this is physically incorrect, it provides meaningful results in most cases.

7.1.3 Regularization

When calculating the actual values for the response characteristic (7.3), divisions by zero and thus undefined states can occur. To prevent this, a continuous and smooth regularization $\text{reg} : \mathbb{R} \to \mathbb{R}_{>0}$ (7.4) is introduced

$$\text{reg}(v) = \begin{cases} |v| & |v| \geq v_{\min} > 0 \\ \frac{|v|^2}{v_{\min}} \left(1 - \frac{|v|}{3v_{\min}}\right) + \frac{v_{\min}}{3} & \text{otherwise} \end{cases}, \tag{7.4}$$

where $v_{\min} \in \mathbb{R}_{>0}$ is the regularization threshold. Function reg returns $|v|$ if $|v| \geq v_{\min}$. Otherwise, it is a third-order polynomial whose minimum is $\frac{v_{\min}}{3}$ for $v = 0$. It has smooth first and second derivatives at $|v| = v_{\min}$. The regularization for $\text{reg}(v)$ with regularization threshold $v_{\min} = 0.01$ is shown in Figure 7.2.

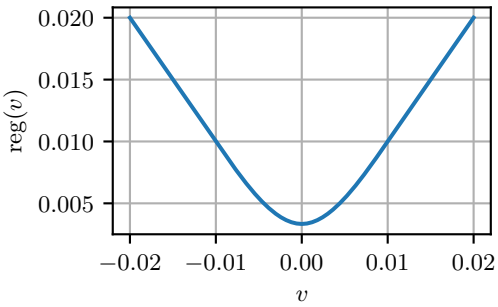


Figure 7.2: Regularization $\text{reg}(v)$ for $v_{\min} = 0.01$.

The regularization reg (7.4) with suitable threshold is used for vectors in direction of the relative tangential and relative angular velocity, and for the contact start velocity. The coefficient of restitution is regularized differently.

Regularized Direction Vectors

The vectors in direction of the relative tangential and relative angular velocity are regularized with (7.4). Note, the absolute value $|\cdot|$ of a vector is the length of $\boldsymbol{v} \in \mathbb{R}^3$.

$$\boldsymbol{v}_{\text{rel}} = \boldsymbol{v}_2 - \boldsymbol{v}_1 \tag{7.5}$$

$$\dot{\delta}_{\text{p}} = \boldsymbol{v}_{\text{rel}} \cdot \boldsymbol{e}_n \tag{7.6}$$

$$\boldsymbol{v}_t = \boldsymbol{v}_{\text{rel}} - \dot{\delta}_{\text{p}}\, \boldsymbol{e}_n \tag{7.7}$$

$$\boldsymbol{e}_{t,\text{reg}} = \frac{\boldsymbol{v}_t}{\text{reg}(|\boldsymbol{v}_t|)} \tag{7.8}$$

$$\boldsymbol{\omega}_{\text{rel}} = \boldsymbol{\omega}_2 - \boldsymbol{\omega}_1 \tag{7.9}$$

$$\boldsymbol{e}_{\omega,\text{reg}} = \frac{\boldsymbol{\omega}_{\text{rel}}}{\text{reg}(|\boldsymbol{\omega}_{\text{rel}}|)} \tag{7.10}$$

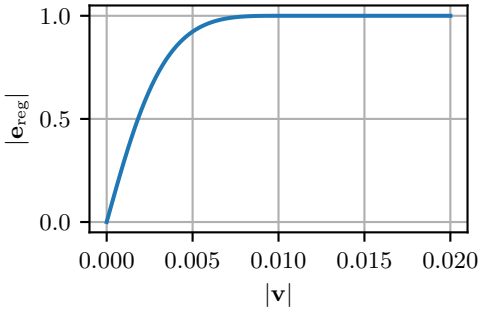


Figure 7.3: Absolute value of a regularized direction vector $|\boldsymbol{e}_{\text{reg}}| = \left|\frac{\boldsymbol{v}}{\text{reg}(|\boldsymbol{v}|)}\right|$ with $v_{\min} = 0.01$.

The regularization thresholds for regularizing the tangential velocity (7.8) and the angular velocity (7.10) are $v_{\min}$ and $\omega_{\min}$. Figure 7.3 shows the absolute value of a regularized direction vector $|\boldsymbol{e}_{\text{reg}}| = \left|\frac{\boldsymbol{v}}{\text{reg}(|\boldsymbol{v}|)}\right|$ with $v_{\min} = 0.01$. If $|\boldsymbol{v}| < v_{\min}$ the regularization is used and $|\boldsymbol{e}_{\text{reg}}| = 0$ if $|\boldsymbol{v}| = 0$. Otherwise, if $|\boldsymbol{v}| \geq v_{\min}$ then $|\boldsymbol{e}_{\text{reg}}| = 1$ is a unit vector.

Regularized Contact Start Velocity

To avoid a division by zero when calculating the damping coefficient d (7.13), the contact start velocity $\dot{\delta}_{\text{p}}^{\,-}$ is regularized with (7.4) and regularization threshold $v_{\min}$

$$\dot{\delta}_{\text{p}_{\text{reg}}}^{\,-} = \text{reg}\left(\dot{\delta}_{\text{p}}^{\,-}\right) > 0. \tag{7.11}$$

A vanishing contact start velocity occurs e.g., if two objects start in touching position.

Regularized Coefficient of Restitution

In reality, all bouncing objects come to rest after a finite number of bounces. In the simulation, an unphysical effect occurs when $cor > 0$, i.e., two objects would bounce infinitely often in finite time until they come to rest. For this reason, cor is reduced

when the velocity at contact start $\dot{\delta}_{\mathrm{p}}^{\ -}$ becomes small. Furthermore, to avoid a division by zero when computing damping coefficient d (7.13), the regularized coefficient of restitution is restricted to a minimum value $cor_{\min}$ (default = 0.001). This leads to the regularized coefficient of restitution cor_{reg}

$$cor_{\mathrm{reg}} = cor + (cor_{\min} - cor)e^{\log(0.01)\frac{|\dot{\delta}_{\mathrm{p}}^{\ -}|}{v_{\min}}}, \tag{7.12}$$

with $0 \leq cor \leq 1$, $\dot{\delta}_{\mathrm{p}}^{\ -} \in \mathbb{R}$, $v_{\min} \in \mathbb{R}_{>0}$, $cor_{\min} \in \mathbb{R}_{>0}$. Figure 7.4 shows this characteristic for several different regularized coefficients of restitution cor_{reg}.

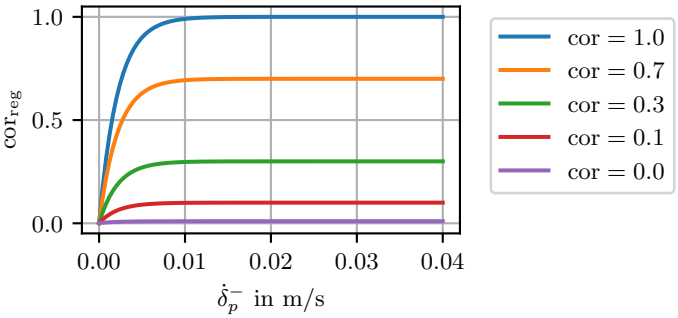


Figure 7.4: Characteristics for several regularized coefficients of restitution.

7.1.4 Damping Coefficient

There are several proposals to compute the damping coefficient as a function of the coefficient of restitution cor and the normal velocity $\dot{\delta}_{\mathrm{p}}^{\ -}$ when contact starts. For a comparison of the different formulations, see Skrinjar, Slavič, and Boltežar (2018). In this thesis, basically the formulation of Flores et al. (2011) is used. The hysteresis-damping coefficient (7.13)

$$d = \min\left(d_{\max}, \frac{8(1 - cor_{\mathrm{reg}})}{5 cor_{\mathrm{reg}} \dot{\delta}_{\mathrm{p}_{\mathrm{reg}}}^{\ -}}\right) \tag{7.13}$$

includes the loss of energy during the contact process. The loss of energy is a function of the coefficient of restitution and the initial penetration velocity. The contact-force model of Flores et al. (2011) is developed for situations occurring between very elastic and very inelastic contacting materials (Machado et al., 2012). It gives similar results with respect to a response calculation with impulses for a wide range of cor values for several experiments performed in Skrinjar, Slavič, and Boltežar (2018).

(7.13) has the following improvements with respect to Flores et al. (2011):

- The regularized initial velocity $\dot{\delta}_{\mathrm{p}_{\mathrm{reg}}}^{-}$ (7.11) is used instead of $\dot{\delta}_{\mathrm{p}}^{-}$ to avoid a division by zero.
- The regularized coefficient of restitution cor_{reg} (7.12) is used instead of cor to avoid a division by zero for $cor = 0$.
- The damping coefficient is limited to $d_{\max} = 2000$ to avoid an unphysical strong creeping effect for collisions with small cor_{reg} values.

The damping coefficient d is shown as function of $\dot{\delta}_{\mathrm{p}}^{-}$ for several cor values in Figure 7.5.

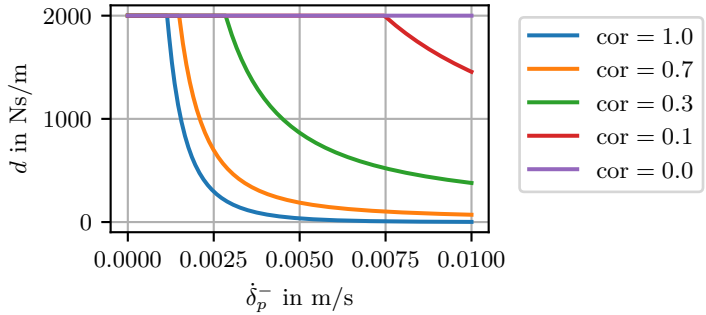


Figure 7.5: Damping coefficient as function of $\dot{\delta}_{\mathrm{p}}^{-}$ and cor. $\dot{\delta}_{\mathrm{p}}^{-}$ from $0.0\,\mathrm{m\,s^{-1}}$ to $0.01\,\mathrm{m\,s^{-1}}$.

7.1.5 Elastic Contact Reduction Factor

The heuristic elastic contact reduction factor k_{red} with $0 < k_{\mathrm{red}} \leq 1$ is introduced in (7.3a). It's default value of 1 is suitable for most physical simulations. The following discussion refers to Neumayr and Otter (2019b). The objective of this heuristic factor is: The application of the elastic-response calculation to hard materials like "Steel" usually leads to stiff ODEs, due to small penetration depths in the order of 10^{-5} m to 10^{-6} m. The penetration depth is implicitly computed by the difference of the absolute positions of the penetrating objects, and these absolute positions are typically error-controlled variables of the solver. This in turn means that, in general, at least a relative tolerance of 10^{-8} must be used for the solver to calculate the penetration depth with 2 or 3 significant digits. To increase the simulation speed, the elastic contact reduction factor k_{red} reduces the stiffness of the contact and thus increases the penetration depth. For example, if k_{red} is set to 10^{-4}, the penetration depth could be on the order of 10^{-3} m, and then a

relative tolerance of 10^{-5} might be sufficient. In many cases, the essential response characteristic is not changed (only the penetration depth is larger), but simulation speed is significantly improved. The heuristic elastic contact reduction factor is clearly a violation of the contact physics. However, especially for impulse like contacts it affects mainly only the penetration depth, but not the overall response.

7.2 Comparison Between Elastic and Impulsive Collision Response

In this section, a comparison is made between the elastic-response characteristic and an impulsive collision response. It shows the intention behind the development of the force law in Flores et al. (2011), and the enhanced response characteristics in (7.3).

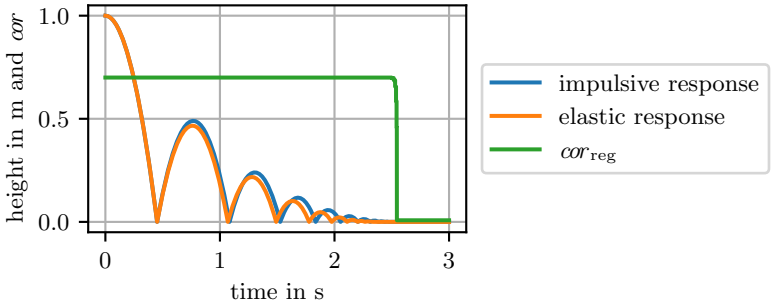


Figure 7.6: Bouncing ball with impulsive collision response and the elastic-response characteristic of (7.3).

In Figure 7.6, the height of a bouncing ball for elastic and impulsive collision responses are displayed. Additionally, the corresponding regularized coefficient of restitution cor_{reg} (7.12) is displayed. The red curve is the collision response if using the elastic-response calculation of (7.3) with $cor = 0.7$ and the solid material constants of "Steel" and of "DryWood". The blue curve is the collision answer if the contact force is computed with an impulse for the same cor value. The green curve is the corresponding regularized coefficient of restitution cor_{reg} (7.12) – for small velocities it becomes small. This shows that f in (7.3) leads to a similar reaction if compared to an impulsive response.

To conclude, the novel response characteristics is an elastic-response calculation. It uses elastic material laws and takes several physical aspects into account. The collision response is limited to point contacts with small penetration depths.

8 Applications and Critical Considerations

Insights into the application with Modia3D as well as its impacts on the internal multibody tree follow. Subsequently, several critical deliberations, including limitations and inherent issues of collision handling with variable-step solvers need to be reflected upon. The approaches chosen in each collision handling step will most likely only work reliably for continuously varying point contacts and penetration depths. Hence, each collision handling step must ensure this condition is met.

8.1 Modia3D used for Collision Handling

Modia3D technically realizes collision handling as it is theoretically described in Chapters 4 to 7. It relies on its internal execution scheme that analyzes the multibody system components, see Section 3.2.4, and reduces the number of potential collision pairs. In Modia3D, it is determined during simulation, if Object3Ds collide with each other to compute the collision response. When using the Modelica modeling language for collision handling, a unique collision identifier must be defined for each collision pair (Otter, Elmqvist, and López, 2005; Elmqvist et al., 2015; Bardaro et al., 2017).

In Modia3D, the preprocessing step uses particular knowledge of the mechanical structure to reduce the number of potential collision pairs, see Figure 8.1. The Object3D's shape, translation, and rotation are needed to create loose-fitting AABBs. Only if AABBs are overlapping the improved MPR algorithm computes penetration depth and penetration velocity. For collision response, material properties, e.g., Young's moduli and the Poisson's ratios are needed to describe the physical behavior.

An Object3D is considered in collision situations if keyword `collidable = true` is set. Additionally, it must be a `Solid` with a `shape` and material properties (e.g., Steel). If the Object3D itself has no mass properties, mass properties must be defined for at least one Object3D of the super-object. In Figure 8.1, collision object obj4 has no mass properties but it is sufficient that obj5 has some.

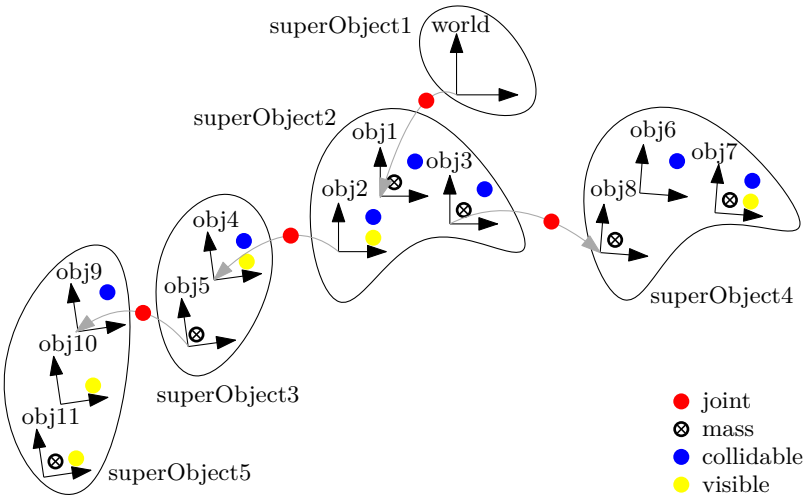


Figure 8.1: Internal execution scheme with two applied collision preprocessing rules. All Object3Ds belonging to a super-object are rigidly attached and cannot collide among themselves e.g., the collision objects: obj1, obj2, and obj3 belong to superObject2 and cannot collide among themselves. The optional rule – super-objects disjunct by a joint cannot collide – can be switched on or off by the user. If it is activated, e.g., superObject2 cannot collide with superObject3 and superObject4 and vice-versa.

8.2 Application: Billiard Game

Neumayr and Otter (2020) model a billiard game[1] with sliding and rolling balls. The cue ball has initial velocity to spread the rack, see Figure 8.2. This model shows the modularity of Modia3D and how to deal with collision handling. The feature of a solid ball is defined once e.g., a solid sphere with collision and mass properties, shape, and visualization material, and is reused for 16 billiard balls in Lines 507 to 526. For preliminary work, see two additional models in Appendix B.

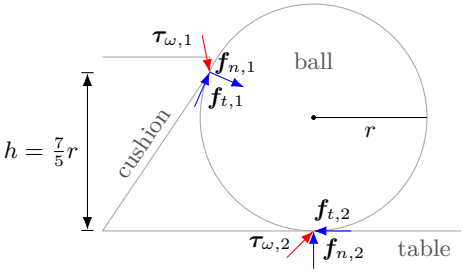
The cushion is arranged properly according to Mathavan, Jackson, and Parkin (2010) in Figure 8.3. To economize collision checks the cushion is simplified. The cushion consists of an upper part with collision features and a lower part without. Both parts are rigidly connected to the table.

Collision handling must be globally enabled for the model. Therefore, `enableContactDetection = true` is set in the Object3D with feature `Scene`. There are several collision objects: 16 billiard balls, the table, and 4 cushions. The 4 cushions

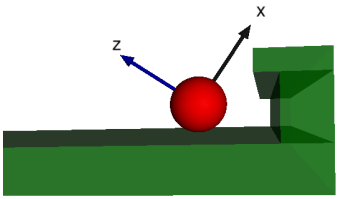
[1] Modia3D.jl, v0.12.0, test/Collision/Billard16Balls.jl

a) At initialization. b) After 1.5 s.

Figure 8.2: Billiard game with 16 balls.



a) Relationship between ball and cushion.



b) Simplified cushion. Billiard ball will collide with upper part of cushion.

Figure 8.3: Corresponding to Mathavan, Jackson, and Parkin (2010) the cushion is arranged and shaped in relationship to the billiard ball radius r. The contact point's height on the rail is $h = \frac{7}{5}r$.

are rigidly connected to the table. So, they belong to one super-object to reduce collision checks. Each billiard ball has 6 Degrees of Freedom (DoF) and is a super-object. There are no joints in this model. The balls collide with other balls, table, and cushions. The mass properties are computed from defined density and their shape.

```
Table = Model(
    table = Object3D(parent=:world, feature=Solid(shape=Box(...),
        solidMaterial="BilliardTable", collidable=true)))
    cushionLower = Object3D(parent=:table, translation=[...],
        feature=Solid(shape=Box(...), solidMaterial="BilliardCushion")),
    cushionUpper = Object3D(parent=:table, translation=[...],
        feature=Solid(shape=Box(...), solidMaterial="BilliardCushion",
        collidable=true)),
    ...)
Ball = Solid(shape=Sphere(...), solidMaterial="BilliardBall", collidable=true)
Billiard = Model3D(
    world = Object3D(feature=Scene(enableContactDetection=true)),
    table = Table,
    cueBall = Object3D(parent=:world, fixedToParent=false, translation=[...],
        velocity=[...], feature=Ball),
    ball1 = Object3D(parent=:world, fixedToParent=false, translation=[...],
        feature=Ball),
    ball2 = Object3D(parent=:world, fixedToParent=false, translation=[...],
        feature=Ball),
    ...)
```

For collision response, the predefine Poisson ratio and Youngs Modulus of `"BilliardBall"`[2] (Lines 527 to 531) is used. The material combination which occur if a billiard ball is in contact with a billiard table is `"BilliardBall, BilliardTable"`[3] (Lines 532 to 538).

```
# Solid material constants
"BilliardBall": {
    "density": 1768.0,
    "YoungsModulus": 5.4e9,
    "PoissonsRatio": 0.34},

# Material constants of a collision pair
"BilliardBall,BilliardTable": {
    "responseType": "ElasticResponse",
    "coefficientOfRestitution": 0.0,
    "slidingFrictionCoefficient": 0.8,
    "rotationalResistanceCoefficient": 0.01
}
```

[2] Modia3D.jl, v0.12.2, palettes/solidMaterials.json

[3] Modia3D.jl, v0.12.2, palettes/contactPairMaterials.json. There are material constants for `"BilliardBall, BilliardBall"` and `"BilliardBall, BilliardCushion"`.

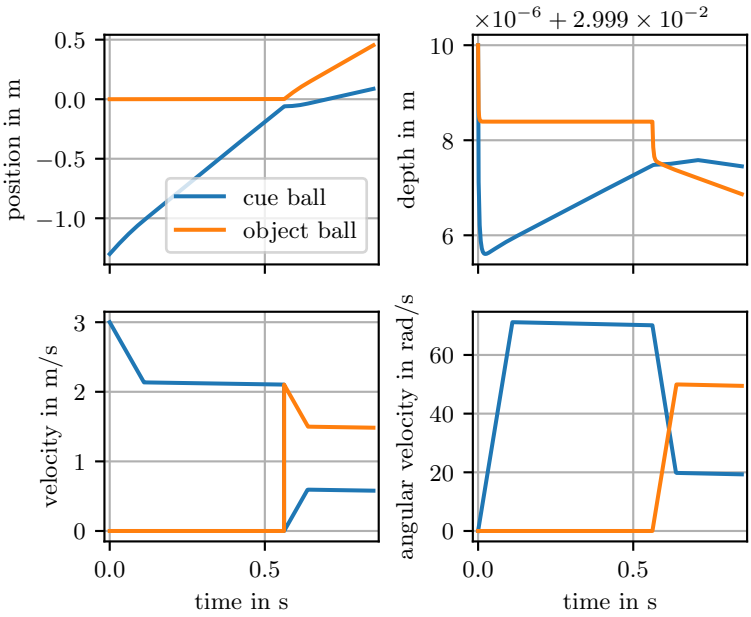


Figure 8.4: The cue ball and object ball subside due to gravity into the table (upper right plot). The cue ball has initial velocity and hits the resting object ball at 0.55 s. The cue ball slides at the beginning and then it rolls. Immediately after impact the cue ball and the object ball slide, and then both roll (lower left and right plots).

The sliding, rolling, and impact of two balls are analyzed in detail with a simplified model[4]. This model consists of a cue ball, an object ball, and a table. Moreover, this particular model is also analyzed for the MPR algorithm in Section 6.1.4, and is briefly discussed in Appendix B, see Figure B.9. Both balls are placed in touching position with the table. There is a gap between the two balls (in x direction), see Figure 8.4. The balls subside immediately into the table due to gravity (in z direction), from 0.03 to about 0.029 996 m. The cue ball has initial velocity (in x direction) and the object ball is at rest. The cue ball slides for about 0.15 s due to the sliding friction force coefficient and the velocity is reduced. At the same time, the sliding friction force acts as a torque around the ball center and forces a rotation of the ball (around the y axis). Without the rotational resistance torque coefficient it would be ideal rolling. At 0.55 s, the cue ball hits the resting object

[4]Modia3D.jl, v0.10.2, test/Collision/TwoCollidingBalls.jl

ball. The coefficient of restitution between the two balls is set to one. Therefore, an ideal elastic collision takes place. This means, the cue ball transfers most of its kinetic energy to the object ball that starts moving with the velocity of the cue ball. The momentum is conserved. Therefore, the cue ball continues rolling and its velocity rises from zero again. Both balls slide and then roll.

Some final remarks on this model: Due to numerical inaccuracies, the resulting contact points and the signed distance may not be aligned to the final normal direction. This results in unphysical behavior. For spheres and ellipsoids, this can be corrected by recalculating the contact point and the resulting signed distance in the final normal direction. For a detailed elaboration see Sections 6.2.2 and 8.3. Moreover, this kind of application would be very time-consuming with a non-convex contact algorithm, e.g., PCM.

In summary, Modia3D's internal execution scheme is especially tailored for collision handling. The user must ensure that an Object3D have collision features. It is therefore a collision object that must be considered in all four collision handling steps. It is determined during simulation if it collides with other collision objects.

8.3 Critical Considerations to Collision Handling with Variable-Step Solvers

Collision handling is a complex task in which several problems can arise. The state-of-the-art collision handling approach consists of three steps: broad phase – approximation by bounding volumes, narrow phase – computation of Euclidean distance or penetration depth, and response calculation – force law. In this thesis, a single preprocessing step is added. For each collision handling step, there are different approaches with comprehensive literature and a number of algorithms to choose from. Each step has its individual advantages and disadvantages which influence the upcoming steps. In order to create awareness, some problems of collision handling in general and of collision handling with variable-step solvers in particular are considered on the basis of well-chosen collision settings.

Preprocessing

Particular knowledge of the mechanical structure is required to reduce the number of potential collision pairs in the upcoming steps. A once-off preprocessing step analyzes the 3D-model.

- To prevent unnecessary collision checks, the first rule – rigidly attached objects cannot collide – is introduced. This rule has no disadvantage.
- The second rule – objects connected by a joint cannot collide – is more ambiguous. On the one hand, this rule avoids undesirable collision computations. Assume the following setting: two collision objects are connected via a joint. Consequently, in the broad phase, their bounding volumes would always overlap near this joint. This causes an execution of all upcoming

collision steps. On the other hand, some collisions might not be detected. Assume the following situation: two collision objects are shaped like two gripping fingers and are connected via a joint. In other words, their end effectors should still collide if necessary. To overcome this limitation, a user-defined keyword is introduced to enable or disable this second preprocessing rule.

Broad Phase

The purpose of the broad phase is to efficiently and simply detect potentially-colliding objects. In this important selection step, each collision object is approximated by simple bounding volumes. Only if the bounding volumes intersect, the computationally expensive narrow phase is executed. There are different types of bounding volumes to choose from. The decision is a tradeoff between additional storage, speed of intersection test, most tightly fitting, and ease of implementation. Based on the discussion in this thesis, Axis-Aligned Bounding Boxes (AABBs) are chosen. They are simple to implement and their intersection tests are fast. Moreover, to further decrease the number of intersection tests performed in the broad phase, AABBs could be extendable to Bounding Volume Hierarchy (BVH) effortlessly. In this thesis, the distance between AABBs is used for zero-crossing functions. Thus, it must be guaranteed that, the transition from non-penetrating to penetrating objects and vice-versa can be always detected by variable-step solvers. In rare cases, if some surfaces or edges of an object are also parallel to an axis, and these objects happen to collide, they already touch each other. To avoid this, the usually tight-fitting AABBs are enlarged by a specific factor of the longest edge to loose-fitting AABBs. This leads to more computationally expensive collision tests in the narrow phase. To further reduce these, more elegant loose-fitting AABBs would be necessary.

Narrow Phase

In the narrow phase, a computationally expensive intersection test is performed. There are several algorithms available to compute contact information for different collision aspects. The decision is a compromise between the greatest possible versatility in handling arbitrary geometries (shapes), computational effort and sufficient accuracy for dynamic 3D simulations. Low-level intersection tests are not adaptable enough because too many combinations of shape types would have to be implemented. A volume discretization is an accurate computation for arbitrary shapes, but it is computationally expensive. These considerations lead to minimizing the computational effort and still obtaining sufficient contact information. The contact information consists of a (unique) contact point on each shape, a penetration depth or Euclidean distance, and a contact normal between the two colliding convex shapes or their convex hull. Still, this contact information is a simplification of collision situations, as some information is lost. The algorithms: Gilbert-Johnson--Keerthi (GJK) algorithm, Expanding Polytope algorithm (EPA), and Minkowski

Portal Refinement (MPR) algorithm all compute the same contact information. Instead of calculating the shortest distance between two convex shapes, they reduce complexity and calculate the shortest distance between one convex shape and the origin. This transforms a shape-to-shape distance problem into a shape-to-point distance problem. In doing so, the two convex shapes are transformed into one corresponding shape in the Minkowski Difference. However, all three algorithms suffer from the same inherent limitations. In this thesis, the MPR algorithm is chosen because it is relatively straightforward to implement.

One major limitation is the restriction to convex shapes and convex hulls of concave shapes. This limitation excludes some collision settings. For example, as the pipe's convex hull is a cylinder, it is impossible to model a ball rolling through a pipe. There are two workarounds for this restriction: approximate the pipe with a convex decomposition or use another, more complex contact detection method.

Another problem arises from the simplification of collision situations and a loss of contact information. Specifically, more contact information is required to describe collisions adequately between two parallel surfaces of boxes, which results in contact surfaces on each box. So, there are an infinite number of contact point pairs that lie on the contact surface. It is obvious that, there exists no unique contact point on each box. Of course, the algorithm will pick one valid solution. If, in addition, the two parallel surfaces change their position due to numerical inaccuracies, this leads to slightly different contact points. This issue also influences the upcoming response calculation because it leads to discontinuous jumps and undesirable behavior of the boxes. For example, the boxes may rotate instead of bouncing if the contact points vary slightly. This can cause the collision simulation to fail. To overcome this limitation and to deal with these types of collision situations appropriately, more contact information must be available. One is to use little helper spheres attached to the surfaces to get a point contact instead of a surface contact. Another is to use the centroids of the emerging contact surfaces as contact points.

The existence of a unique contact point on each shape cannot be guaranteed. Whether unique or ambiguous contact points occur depend on the collision shapes, their relative position and their relative orientation.

- Unique contact points for two colliding boxes do not always exist. Their existence depends on their relative orientation and relative position.
  - As already discussed, there is no unique contact point on each box for two parallel touching surfaces.
  - There is a unique contact point on each box if a vertex touches a surface.
- There are always unique contact points if at least one collision shape is a sphere or an ellipsoid, regardless of their orientation and position.

Another drawback concerns the definition of the translational penetration depth, which can lead to discontinuities for dynamically moving objects. These disconti-

nuities are depicted on the basis of two collision situations that are discussed in more detail for the improved MPR algorithm in Section 6.2.2.

- The definition of the penetration depth is based on pure geometric properties. There is no information about dynamic movement. Thus, discontinuities occur, for instance when a bullet is penetrating an object. If the bullet is too far inside the object, the penetration depth is no longer aligned with the penetration in the direction of motion. This leads to a discontinuous jump of the penetration depth and contact points. There are algorithms to overcome this limitation, which take the movement of past geometric properties into account. To prevent this issue, only small penetrations are considered in this thesis.
- Discontinuities appear for non-smooth shapes, and the penetration depth jumps on edges and vertices, e.g., a sphere rolling uniformly over an edge or vertex of a box. Again, the penetration depth and contact points jump discontinuously. To avoid this, collision smoothing radii are introduced to smooth these edges.

Furthermore, in this thesis several major improvements to the MPR algorithm are made. For a detailed discussion of these enhancements, see Section 6.1.5. Among others, four considerable findings are highlighted here once more.

- The improved MPR algorithm verifies if one or both collision objects are degenerated into a 2D geometry. Since, this work is devoted to the behavior of 3D objects, there is no need for a 2D collision algorithm.
- To avoid infinite looping and once the maximum number of iterations is reached or, if further refinement fails, the improved MPR algorithm is terminated with the best found solution so far. Otherwise, the last solution, which is not necessarily the best solution, is chosen. In some rare situations, this led to discontinuous jumps of contact points.
- Due to numerical issues, the resulting contact points and the signed distance may not be aligned to the final normal direction. For shapes such as spheres and ellipsoids, with locally bijective contact points to their normal, this can be remedied by recalculating the contact point in the final normal direction and the resulting signed distance. Further research needs to be conducted for the other kind of shapes.
- Since the MPR algorithm is numerically sensitive, it is performed with quadruple precision to increase robustness and reliability when variable-step solvers are used.

Collision Response

Some critical considerations to the fourth collision handling step – the collision response – are highlighted. As always, depending on the application, there are different force laws to choose from. In this thesis, the force law considers several

physical aspects. It uses elastic material laws that are based on the theory of Hertz for elastic shapes. The force law is based, among others, on the penetration depth and the penetration velocity. The physical behavior and material of the shapes are reflected in some coefficients. Especially one coefficient, a rough approximation for principal curvatures, is computed approximately based on the contact theory of Hertz. A sphere with constant radius can be associated with any shape. This artificial radius is used to calculate the coefficient. The default value of this radius is based on the geometry of the shape. The problem with this simplification is that not every shape has an associated radius. So, with a cylinder, for example, it is a compromise between diameter and length. To address this issue, the user is allowed to change this value.

A continuous penetration depth must be ensured, otherwise small changes and discontinuities during a solver step, can result in totally different contact points, and thus an unphysical behavior.

In summary, the collision handling method presented here, which uses variable-step size solvers and the use of penetration depth, will most likely only work reliably for continuously varying point contacts and penetration depth. In the here presented approach, only solid material and collision material pairing must be defined by the user.

Part III

Variable Structure Systems

9 Varying Number of States before Simulation

In traditional object-oriented modeling languages, especially Modelica, variable types and array sizes are precisely defined. Although a state is a vector, it is decomposed into scalars when transformed from a DAE to an ODE. This information is used for symbolically analyzing the structure of DAEs to generate and compile code. Thus, it is impossible to vary the length of a state afterwards.

Contrary, Modia's symbolic and simulation engine can handle models where the number of states can be varied after the model code has been generated and compiled and before the simulation starts. To that end, Modia takes full advantage of the Julia language. Julia has a very extensive type system with type inference. The goal of the Modia language is to use all these underlying variable types of Julia instead of replicating them. As a result, the Modia language and the Modia symbolic transformations do not have the complete information about the variable types until it is determined in the Julia compiler inference run. In particular, variable sized matrices and vectors of parameters and variables defined with `init` or `start` attributes can be varied before simulation starts. The upcoming procedure is already published by Neumayr and Otter (2023a).

A Modia model defines a set of unknown variables, a set of equations, and a set of known variables (parameters). It is not necessary to know the type of the variables. An unknown variable can also be an instance of a mutable Julia structure. The basic requirement is that the number of unknown variables and the number of equations must be equal. For example, if a variable is a vector, there must be an equation to calculate this vector. Whether this requirement is fulfilled or not, can be only determined when the model code is generated or even only during execution. Also, a variable is usually treated as one symbol and the associated equation as one symbol equation, even if the symbol is an array (Lines 543 to 544). An array equation must be defined for an array variable. Furthermore, an array must be declared with an init or start array value if its size cannot be inferred. A model with correct array handling is given in Lines 539 to 545. Contrary, in Lines 546 to 552, the array handling is wrong. The reason is the sizes cannot be inferred and there is no array equation.

```
539 # Correct code
540 m1 = Model(
541     v = Var(init=zeros(2)), # size of v cannot be inferred --> init needed
542     equations = :[
```

```
        a = der(v)          # size of a can be inferred
        m*a = [2.0, 3.0]]   # array equation for a
)

# Wrong code
m2 = Model(
    equations = :[
        wd = der(w)     # sizes of w, wd cannot be inferred
        m*wd[1] = 2.0   # no array equation for wd
        m*wd[2] = 3.0]
)
```

Consequently, the size of a variable does usually not affect symbolic transformation or code generation. The equations generated are basically the same regardless of whether a variable is a scalar or has, e.g., 10,000 elements. All this differs from current Modelica tools, where variables and equations are usually decomposed before symbolic transformation takes place. This means, an array with 10,000 elements is replaced by 10,000 scalars, so that 10,000 symbols are used in symbolic transformations.

The array sizes of parameters and of variables defined with `init` or `start` attributes can be varied in Modia after code generation, unless they are defined as static arrays. A model with static and variable array equations is given in Lines 553 to 573. Matrix `A1` and vector `y1` are arrays with static size. Such arrays and dimensions cannot be varied after compiling if `@instantiateModel` was called. Contrary, matrix `A2` and vector `y2` are standard Julia arrays with variable sizes. Their dimensions can be varied after compiling with the `merge` attribute of the `simulate!` command.

```
using StaticArrays # needed for statically sized SVector and SMatrix
LinearODEs = Model(
    # matrix A1 and vector y1 with static size
    A1 = parameter | SMatrix{2,2}([-1.0  0.0;
                                   0.0 -2.0]),
    y1 = Var(init = SVector{2}(1.0, 2.0)),
    # matrix A2 and vector y2 with variable size
    A2 = parameter | [-1.0  0.0;
                       0.0 -2.0],
    y2 = Var(init=[1.0, 2.0]),
    equations = :[
        der(y1) = A1*y1     # static array equation
        der(y2) = A2*y2]    # variable array equation
linearODEs = @instantiateModel(LinearODEs)  # generate and compile code
simulate!(linearODEs, stopTime = 2,
    # vary sizes of parameter A2 and variable y2
    merge = Map(
        A2 = [-1.0  0.0  0.0;
               0.0 -2.0  0.0;
               0.0  0.0 -3.0],
        y2 = [1.1, 2.1, 3.1]))
```

When executing the linear ODE model (Lines 553 to 573) Modia generates the function `getDerivatives` (Lines 576 to 586). Since the model uses static and variable sized arrays, the generated function `getDerivatives` allocates memory on the stack and on the heap.

```
574 # _x    ... states vector from solver
575 # model ... instantiated simulation model
576 function getDerivatives(_x, model, time)
577     ...
578     p = model.evaluatedParameters   # hierarchical dictionary
579     A1::SMatrix{2,2,Float64,4} = p[:A1]  # p[:A1] is the value of symbol :A1
580     A2::Matrix{Float64}        = p[:A2]
581     y1 = SVector{2,Float64}(_x[1], _x[2])
582     y2 = model.x_vec[1]
583     der(y1) = A1 * y1
584     der(y2) = A2 * y2
585     ...
586 end
```

Due to Julia's multiple dispatch, it is distinguished at compile time that `A1 * y1` (Line 583) is a static array and `A2 * y2` (Line 584) is a variable array. This follows from: `A1` and `y1` are static arrays and operator `*` is overloaded. Therefore, a static `SVector` array `der(y1)` is generated and allocates memory on the stack, i.e., fast memory. Operations on `SVector`s are efficiently implemented in `StaticArrays.jl` package (JuliaArrays, online). New arrays needed for calculations are automatically rebuilt on the stack. For example, the statically sized state vector `y1` is always newly regenerated by using the corresponding elements from the model state vector `_x` in the `SVector` constructor.

In contrast, auxiliary memory `model.x_vec[1]` is allocated for the state vector `y2` when the `merge` attribute has been processed (Line 582). Before calling function `getDerivatives`, the corresponding elements of the model state vector `_x` are copied into `model.x_vec[1]`. The generated array equation code does not depend on the array sizes of the variables involved.

On the one hand variable sized arrays, can vary their dimensions after compiling and before simulation starts. But they allocate new memory which might reduce efficiency. On the other hand, static sized arrays are implemented efficiently but their dimensions cannot be varied. In summary, static and variable sized arrays with their pros and cons are discussed above.

10 Varying Number of States during Simulation

A new general method for handling equation-based models with variable structure systems is introduced. States and other variables can be added and removed during simulation, without regenerating and recompiling the model code. This new approach is established in a generic mathematical way. Subsequently, it is specified to be used in Modia and Modia3D. In principle, the method can also be used for other modeling systems, e.g., in an extended version of Modelica. This noteworthy method is already published by Neumayr and Otter (2023a).

To simplify the description and focus on the new technique, time discrete systems, time events, event iteration, and super-dense time (see e.g., Modelica Association, 2021, Appendix B; FMI Project, 2014, Section 3.1) are not discussed in this thesis. However, in the Modia package these features are included.

This chapter relies on the fundamental concepts for equation-based modeling, especially symbolic transformation, see Chapter 2. In Section 10.1 so-called predefined acausal components are introduced. These are acausal components with pre-translated functions, internal memory and remaining equations. A novel procedure in Section 10.2 utilizes these predefined acausal components. This procedure allows to dynamically vary the number of variables, including algebraic variables and states as well as the number of equations during simulation. The re-initialization and the redefinition of states and their start values is performed on-the-fly, without recompiling.

10.1 Predefined Acausal Components

Predefined acausal components[1] are based on acausal components, see Section 10.1.1. The equations of an acausal component can be split into causal and acausal partitions consisting of pre-translated functions and remaining equations, see Section 10.1.2. These pre-translated functions calculate states, their derivatives and event indicators which are hidden in an internal memory. This memory with the hidden states is crucial to be utilized for variable structure systems. Acausal

[1] Neumayr and Otter (2023a) refer to these components as acausal built-in components. Neumayr and Otter (2023b) decide to rename them to predefined acausal components to be more descriptive.

components with pre-translated functions, internal memory and remaining equations are called predefined acausal components, see Section 10.1.3. An illustrative application of a predefined acausal component is discussed in Section 10.1.4. Another application of a heat transfer in a rod shows how to split the equations into causal and acausal partitions, see Section 10.1.5.

10.1.1 Acausal Components

The basis of simulating variable structure systems are acausal components. An acausal component can be connected with other components via inputs, outputs, and connectors containing pairs of potential variables and flow variables. At events, variable values can change discontinuously.

The mathematical description of an acausal component (10.1)

$$\mathbf{0} = \boldsymbol{f}_\mathrm{c}\left(\dot{\boldsymbol{x}}, \boldsymbol{z}, \boldsymbol{w}, \boldsymbol{c}_p, \boldsymbol{c}_f, \boldsymbol{y}, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, t\right), \tag{10.1}$$

is a set of DAEs, where

$$t \in \mathbb{R} \quad \text{time,} \tag{10.2a}$$
$$\boldsymbol{p} \in \mathbb{R}^{n_p} \quad \text{parameters,} \tag{10.2b}$$
$$\boldsymbol{u}(t) \in C^0_\mathrm{pw}(\mathbb{R})^{n_u} \quad \text{inputs,} \tag{10.2c}$$
$$\boldsymbol{x}(t) \in C^1_\mathrm{pw}(\mathbb{R})^{n_x} \quad \text{continuous states,} \tag{10.2d}$$
$$\boldsymbol{y}(t) \in C^0_\mathrm{pw}(\mathbb{R})^{n_y} \quad \text{outputs,} \tag{10.2e}$$
$$\boldsymbol{c}_f(t) \in C^0_\mathrm{pw}(\mathbb{R})^{n_c} \quad \text{flow variables,} \tag{10.2f}$$
$$\boldsymbol{c}_p(t) \in C^0_\mathrm{pw}(\mathbb{R})^{n_c} \quad \text{potential variables,} \tag{10.2g}$$
$$\boldsymbol{w}(t) \in C^0_\mathrm{pw}(\mathbb{R})^{n_w} \quad \text{local variables,} \tag{10.2h}$$
$$\boldsymbol{z}(t) \in C^0_\mathrm{pw}(\mathbb{R})^{n_z} \quad \text{event indicators,} \tag{10.2i}$$

with minimal smoothness requirements[2][3]. Variables in greenish color are assumed to be known: time t and states $\boldsymbol{x}$ are provided by the solver, $\boldsymbol{p}$ are parameters that get constant values before the simulation starts, $\boldsymbol{u}$ are inputs and are provided externally to the component. Unknowns are $\dot{\boldsymbol{x}}, \boldsymbol{z}, \boldsymbol{w}, \boldsymbol{c}_p, \boldsymbol{c}_f, \boldsymbol{y}$. Figure 10.1 depicts the acausal component's interface.

[2] Definitions: Let $\mathbb{R}$ be the set of real numbers and assume $k \in \mathbb{N}_0$. $C^k(\mathbb{R})$ is the space of functions which are k times continuously differentiable in $\mathbb{R}$. This means, $C^1(\mathbb{R})$ is the function space of 1 time continuously differentiable functions. $C^0(\mathbb{R})$ is the function space of continuous functions. Furthermore, due to events there are discontinuous jumps. $C^1_\mathrm{pw}(\mathbb{R})^n$ is the space of piecewise (pw) 1 time continuously differentiable functions in n dimensions, as well as $C^0_\mathrm{pw}(\mathbb{R})^n$ is the space of pw continuous functions in n dimensions. See e.g., Steinbach (2007) for further details.

[3] Higher smoothness might be required (see e.g., Campbell, Linh, and Petzold, 2008). It depends on the component's connection and on the structure of the equations (10.1).

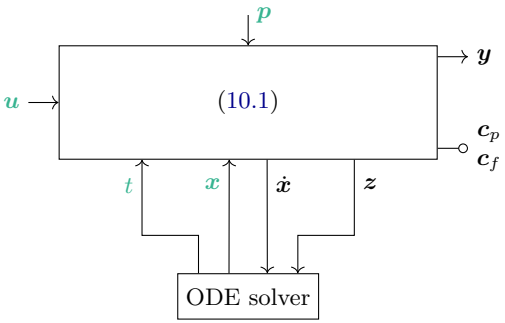


Figure 10.1: Mathematical description of an acausal component with (10.1) and (10.2). It can be connected with other components via its interface for inputs, outputs, and connectors.

To solve (10.1) for its unknowns, n_c equations are missing. These n_c missing equations are provided in (10.3) when connecting the component via the connection equations of $\boldsymbol{c}_p, \boldsymbol{c}_f$. The connected potential variables are set equal and the sum of the connected flow variables is set to zero. A connector has only corresponding pairs of potential and flow variables to ensure that each connection of acausal components is globally balanced. In other words, the number of equations and the number of unknowns of any set of connected components are equal, assuming that each component is locally balanced. A component is locally balanced (Olsson et al., 2008), if

$$\begin{aligned} \dim(\boldsymbol{c}_p) &= \dim(\boldsymbol{c}_f) = n_c, \\ \dim(\boldsymbol{f}_c) &= n_y + n_c + n_w + n_z. \end{aligned} \tag{10.3}$$

The equations of all components of a model, together with the connection equations form a set of DAEs[4]. The set of DAEs is transformed into a set of ODEs and is solved by an ODE solver. In each model evaluation, the time t and the continuous states $\boldsymbol{x}$ are provided to the model by the solver. Using the symbolically transformed equations, the derivative of the states $\dot{\boldsymbol{x}}$ and the event indicators $\boldsymbol{z}$ are calculated and returned to the solver.

10.1.2 Acausal Components with Pre-translated Functions

Elmqvist (2019, pp. 7–10) proposes a generic method to split the equations of an acausal component (10.1), (10.2) into causal and acausal partitions. The causal

[4]The equations in the `equation` section of a Modelica model or the equations in the `equations` vector of a Modia model belong to the system of algebraic equations.

partitions are sorted (causalized), pre-translated and always evaluated in the same order, regardless of the component's connection to other components. The unknowns are evaluated. In contrast, the sorted order of the acausal partition depends on the component's actual connection.

Assume, potential and flow variables are present in a component. These or parts of them are often supplied externally. To prepare for all these cases, n_c dummy equations

$$\mathbf{0} = \boldsymbol{g}_{\mathrm{c}}(\boldsymbol{c}_p, \boldsymbol{c}_f) \tag{10.4}$$

are defined, where each element of each argument appears in each equation of (10.4). So, (10.4) has full incidence in all of its arguments. Mathematically speaking, (10.4) defines a large number of potential connection possibilities of a component. Let $\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, t$ be known. (10.1) and (10.4) are sorted by using only structural information[5]. This leads to the system of algebraic equations (10.5)

$$\begin{aligned} \mathbf{0} &= \boldsymbol{f}_{\mathrm{c,eq}}(\ldots) \\ \mathbf{0} &= \boldsymbol{g}_{\mathrm{c}}(\ldots), \end{aligned} \tag{10.5}$$

where $\boldsymbol{f}_{\mathrm{c,eq}}$ is subset of $\boldsymbol{f}_{\mathrm{c}}$. Since (10.4) has full incidence, all equations of (10.4) are included in (10.5). Moreover, (10.5) represents the acausal part of the component. These equations are needed to calculate all arguments of (10.4), i.e., the potential and flow variables.

All other sorted equations do not depend on how the component is connected. Equations that appear in the sorted order before $\boldsymbol{f}_{\mathrm{c,eq}}$ can be included in a function $\boldsymbol{f}_{\mathrm{c},1}$ (10.6a). Equations that appear in the sorted order after $\boldsymbol{f}_{\mathrm{c,eq}}$ can be included in a function $\boldsymbol{f}_{\mathrm{c},2}$ (10.6c). Within functions $\boldsymbol{f}_{\mathrm{c},1}, \boldsymbol{f}_{\mathrm{c},2}$, there may be local linear and/or nonlinear algebraic equations to solve. Removing equations for $\boldsymbol{g}_{\mathrm{c}}$ (10.4) from the sorted equations leads to the mathematical description of a general acausal component (10.6)

$$(\dot{\boldsymbol{x}}_1, \boldsymbol{z}_1, \boldsymbol{w}_1, \boldsymbol{c}_{p_1}, \boldsymbol{c}_{f_1}, \boldsymbol{y}_1) = \boldsymbol{f}_{\mathrm{c},1}(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, t) \tag{10.6a}$$

$$\mathbf{0} = \boldsymbol{f}_{\mathrm{c,eq}}(\dot{\boldsymbol{x}}_{\mathrm{eq}}, \boldsymbol{z}_{\mathrm{eq}}, \boldsymbol{w}_{eq}, \boldsymbol{c}_{p_{\mathrm{eq}}}, \boldsymbol{c}_{f_{\mathrm{eq}}}, \boldsymbol{y}_{\mathrm{eq}}, \dot{\boldsymbol{x}}_1, \boldsymbol{z}_1, \boldsymbol{w}_1, \boldsymbol{c}_{p_1}, \boldsymbol{c}_{f_1}, \boldsymbol{y}_1, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, t) \tag{10.6b}$$

$$(\dot{\boldsymbol{x}}_2, \boldsymbol{z}_2, \boldsymbol{w}_2, \boldsymbol{c}_{p_2}, \boldsymbol{c}_{f_2}, \boldsymbol{y}_2) = \boldsymbol{f}_{\mathrm{c},2}(\dot{\boldsymbol{x}}_{\mathrm{eq}}, \boldsymbol{z}_{\mathrm{eq}}, \boldsymbol{w}_{eq}, \boldsymbol{c}_{p_{\mathrm{eq}}}, \boldsymbol{c}_{f_{\mathrm{eq}}}, \boldsymbol{y}_{\mathrm{eq}}, \dot{\boldsymbol{x}}_1, \boldsymbol{z}_1, \boldsymbol{w}_1, \boldsymbol{c}_{p_1}, \boldsymbol{c}_{f_1}, \boldsymbol{y}_1, \boldsymbol{x}, \boldsymbol{u}, \boldsymbol{p}, t). \tag{10.6c}$$

Functions $\boldsymbol{f}_{\mathrm{c},1}, \boldsymbol{f}_{\mathrm{c},2}$ can be translated once in advance. So, these two functions are pre-translated. The remaining equations $\boldsymbol{f}_{\mathrm{c,eq}}$ are solved with the overall model. The sorted order of (10.6) will be kept. The unknown variables $\dot{\boldsymbol{x}}, \boldsymbol{z}, \boldsymbol{w}, \boldsymbol{c}_p, \boldsymbol{c}_f, \boldsymbol{y}$ of (10.1) are split into three parts, e.g., $\boldsymbol{y} = (\boldsymbol{y}_1, \boldsymbol{y}_{\mathrm{eq}}, \boldsymbol{y}_2)$. All inputs of $\boldsymbol{f}_{\mathrm{c},1}$ are

[5] In other words, a variable occurs or does not occur in an equation (see e.g., Elmqvist and Otter, online[a], Sections 1.1–1.4).

known, so the unknowns of (10.6a) are evaluated straightforwardly. After equations $\boldsymbol{f}_{\mathrm{c,eq}}$ are solved together with equations of the overall model, all inputs of $\boldsymbol{f}_{\mathrm{c},2}$ are known to evaluate the unknowns of (10.6c). The yellow variables will be moved into an internal memory in Section 10.1.3. Figure 10.2 depicts the interface of the acausal component with pre-translated functions and remaining equations.

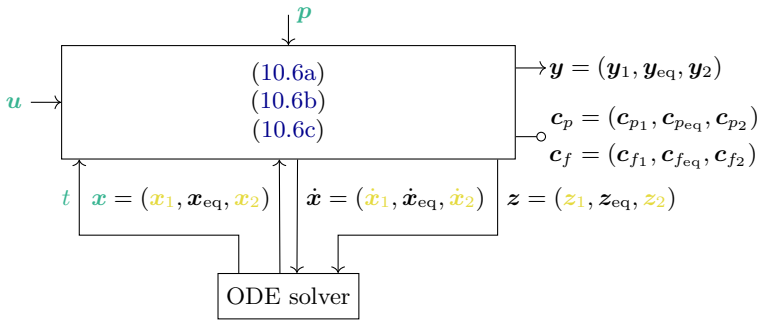


Figure 10.2: Description of an acausal component with pre-translated functions $\boldsymbol{f}_{\mathrm{c},1}$ (10.6a), $\boldsymbol{f}_{\mathrm{c},2}$ (10.6c) and remaining equations $\boldsymbol{f}_{\mathrm{c,eq}}$ (10.6b). This component has the same interface as Figure 10.1 and can be connected via $\boldsymbol{u}, \boldsymbol{y}, \boldsymbol{c}_p, \boldsymbol{c}_f$. Unknown variables $\dot{\boldsymbol{x}}, \boldsymbol{z}, \boldsymbol{w}, \boldsymbol{c}_p, \boldsymbol{c}_f, \boldsymbol{y}$ are split into three parts.

It is impossible to find a general partitioning with pre-translated functions which is suitable for all possible connections. For example, an input/output block is a component without potential and flow variables. On the one hand, the inputs are usually externally supplied. The outputs are calculated from the component equations. Partitioning is performed for this general situation. All equations are sorted and evaluated for the unknowns. Thus, all code can be pre-translated. On the other hand, if the inverse of an input/output block is to be determined, the outputs are externally supplied. The inputs are calculated from the component equations. It may not be possible to determine this inverse model with a pre-translated block. Additional differentiations of equations and derivatives of some inputs might be required. Furthermore, if the whole code of a pre-translated block is contained in one function, a system of algebraic equations can occur when the block is connected. It is difficult to decide whether all causal code should be in one function or whether it is split into more functions, as the example of (FMI Project, 2014, page 73, Figure 5) shows. So, it is impossible to split an acausal component into causal and acausal partitions that are suitable for all possible connections. Rather, the most important use cases of component's connections must be known. Then such a partitioning can be created for these applications.

The advantage of an acausal component like in (10.6) is that $\boldsymbol{f}_{\mathrm{c},1}, \boldsymbol{f}_{\mathrm{c},2}$ can be translated once in advance. This allows the symbolic transformation of an overall

model and generating and compiling of code to be done much more efficiently than with the original formulation of (10.1).

10.1.3 Predefined Acausal Components

Predefined acausal components are acausal components with pre-translated functions, internal memory and remaining equations. The pre-translated functions calculate states, their derivatives and event indicators to store and hide them in an internal memory. This means that they are no longer visible in the model equations. However, they are still passed on to the solver. Furthermore, states and event indicators are split into invariant and variant parts. The communication between them, the functions of predefined acausal components, sorted and solved equations, and the solver are described below. These are prerequisites to be used for variable structure systems.

The mathematical description of a general acausal component with pre-translated functions (10.6) is modified. Therefore, the yellow variables of (10.6) are stored in an internal memory $\mathtt{m}$ of a program. These variables are referred to as hidden variables. This leads to the description of a general predefined acausal component (10.7)

$$(\boldsymbol{w}_3, \boldsymbol{c}_{p_1}, \boldsymbol{c}_{f_1}, \boldsymbol{y}_1) = \boldsymbol{f}_{\mathrm{c},1}(\mathtt{m}, \boldsymbol{x}_{\mathrm{eq}}, \boldsymbol{u}, \boldsymbol{p}, t) \tag{10.7a}$$

$$\mathbf{0} = \boldsymbol{f}_{\mathrm{c,eq}}(\dot{\boldsymbol{x}}_{\mathrm{eq}}, \boldsymbol{z}_{\mathrm{eq}}, \boldsymbol{w}_{eq}, \boldsymbol{c}_{p_{\mathrm{eq}}}, \boldsymbol{c}_{f_{\mathrm{eq}}}, \boldsymbol{y}_{\mathrm{eq}}, \boldsymbol{w}_3, \boldsymbol{c}_{p_1}, \boldsymbol{c}_{f_1}, \boldsymbol{y}_1, \mathtt{m}, \boldsymbol{x}_{\mathrm{eq}}, \boldsymbol{u}, \boldsymbol{p}, t) \tag{10.7b}$$

$$(\boldsymbol{c}_{p_2}, \boldsymbol{c}_{f_2}, \boldsymbol{y}_2) = \boldsymbol{f}_{\mathrm{c},2}(\dot{\boldsymbol{x}}_{\mathrm{eq}}, \boldsymbol{z}_{\mathrm{eq}}, \boldsymbol{w}_{eq}, \boldsymbol{c}_{p_{\mathrm{eq}}}, \boldsymbol{c}_{f_{\mathrm{eq}}}, \boldsymbol{y}_{\mathrm{eq}}, \boldsymbol{w}_3, \boldsymbol{c}_{p_1}, \boldsymbol{c}_{f_1}, \boldsymbol{y}_1, \mathtt{m}, \boldsymbol{x}_{\mathrm{eq}}, \boldsymbol{u}, \boldsymbol{p}, t), \tag{10.7c}$$

with pre-translated functions $\boldsymbol{f}_{\mathrm{c},1}$ (10.7a), $\boldsymbol{f}_{\mathrm{c},2}$ (10.7c), an internal memory $\mathtt{m}$ and remaining equations $\boldsymbol{f}_{\mathrm{c,eq}}$ (10.7b) that are solved with the overall model. The sorted order of (10.7) will be kept. Unlike in (10.6), $\boldsymbol{f}_{\mathrm{c},1}, \boldsymbol{f}_{\mathrm{c},2}$ are no longer mathematical functions since the memory $\mathtt{m}$ is both an input and an output argument to these functions. The memory $\mathtt{m}$ is exchanged between $\boldsymbol{f}_{\mathrm{c},1}$ and $\boldsymbol{f}_{\mathrm{c},2}$. States x_1, x_2 are copied from the solver to the memory $\mathtt{m}$ by $\boldsymbol{f}_{\mathrm{c},1}$. Moreover, $\boldsymbol{f}_{\mathrm{c},1}, \boldsymbol{f}_{\mathrm{c},2}$ evaluate state derivatives $\dot{x}_1, \dot{x}_2$ and event indicators z_1, z_2 and copy them from the memory to the solver. A memory is added to the component's interface in Figure 10.3.

There are issues with (10.7). The states $\boldsymbol{x}$, as well as the output variables $\dot{x}_1, z_1, w_1$ of function $\boldsymbol{f}_{\mathrm{c},1}$ are (possibly) present in $\boldsymbol{f}_{\mathrm{c,eq}}$. It would then not be possible to add or remove these variables during the simulation without code regenerating. If this happens, retransformations are needed. An issue of (10.7) is that parts of the equations in $\boldsymbol{f}_{\mathrm{c,eq}}$ are calculated using the memory. The variables $\dot{x}_1, z_1, w_1, x_1, x_2$ in $\boldsymbol{f}_{\mathrm{c,eq}}$ are no longer visible. It may be necessary to hide some of these variables in (new) local algebraic variables $\boldsymbol{w}_3$, which are returned by $\boldsymbol{f}_{\mathrm{c},1}$ and used in $\boldsymbol{f}_{\mathrm{c,eq}}$. Another issue of predefined acausal components is that $\boldsymbol{f}_{\mathrm{c},1}, \boldsymbol{f}_{\mathrm{c},2}$ cannot be differentiated due to their internal memory.

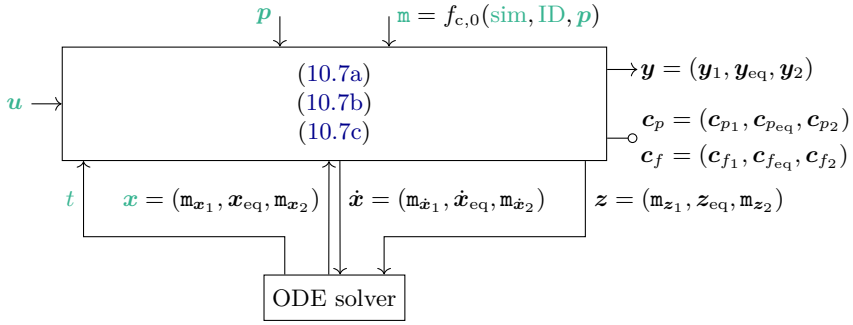


Figure 10.3: Description of a predefined acausal component with pre-translated functions $\boldsymbol{f}_{\mathrm{c},1}$ (10.7a), $\boldsymbol{f}_{\mathrm{c},2}$ (10.7c), an internal memory $\mathtt{m}$, and remaining equations $\boldsymbol{f}_{\mathrm{c,eq}}$ (10.7b). It has the same interface as Figures 10.1 and 10.2. Moreover, $\mathtt{m} = f_{\mathrm{c},0}(\mathrm{sim}, \mathrm{ID}, \boldsymbol{p})$ is an instance of the predefined acausal component. It is constructed before simulation starts with a given reference to the simulation engine sim, a unique identification ID of the instance, and parameters $\boldsymbol{p}$. The hidden state derivatives $\mathtt{m}_{\dot{\boldsymbol{x}}_1}$ are evaluated in $\boldsymbol{f}_{\mathrm{c},1}$ and $\mathtt{m}_{\dot{\boldsymbol{x}}_2}$ in $\boldsymbol{f}_{\mathrm{c},2}$. In some cases, it is necessary to introduce new local algebraic variables e.g., to hide state variables in $\boldsymbol{f}_{\mathrm{c,eq}}$.

The first prerequisite towards variable structure systems is that the states x_1, x_2, their derivatives $\dot{x}_1, \dot{x}_2$, and event indicators z_1, z_2 are hidden in a memory and are not visible in the model equations. Moreover, the code parts for the pre-translated functions $\boldsymbol{f}_{\mathrm{c},1}, \boldsymbol{f}_{\mathrm{c},2}$ exist only once, regardless of the number of instances from the predefined acausal component. Additionally, the used arrays operate on memory allocated once, rather than in each model evaluation. Consequently, the effort for symbolic transformations can be drastically reduced.

The second prerequisite for variable structure systems is explained in the following: The variables (states $\boldsymbol{x}$ and event indicators $\boldsymbol{z}$) communicated between the solver, the sorted and solved equations and the functions of the predefined acausal components are split into an invariant and variant part: $\boldsymbol{x} = (\boldsymbol{x}^{\mathrm{inv}}, \boldsymbol{x}^{\mathrm{var}})$, $\boldsymbol{z} = (\boldsymbol{z}^{\mathrm{inv}}, \boldsymbol{z}^{\mathrm{var}})$, see Figure 10.4. Before the simulation starts, the dimensions of the invariant parts are defined. During simulation, the dimensions of the variant parts can vary at events. The variables are characterized by the following attributes:

- invariant (inv): Variable name, type and number of dimensions are fixed before code generation. All variables defined and used in an equation section are invariant variables. This includes all input/output arguments of the called functions. The dimensions of invariant variables (e.g., length of a vector) can be varied before simulation starts. The solver returns $\boldsymbol{x}^{\mathrm{inv}}$ to the model function, which contains the sorted and solved equations. The

elements of $\boldsymbol{x}^{\text{inv}}$ are copied into the elements of the invariant state variables of the model. The calculated derivatives of the invariant state variables are copied into $\dot{\boldsymbol{x}}^{\text{inv}}$ and the calculated invariant event indicators are copied into $\boldsymbol{z}^{\text{inv}}$ before the model function is returned.

- variant (var): At events, new variables can appear and existing variables can disappear. The type and number of dimensions of a variant variable cannot be varied after it has been introduced for the first time in a simulation run. However, its dimensions (e.g., the length of an array) can be varied at events. All model variables defined in pre-translated functions of predefined acausal components are variant variables. Before using variant state variables of a model in a predefined acausal component's function, their elements are retrieved from the solver's vector $\boldsymbol{x}^{\text{var}}$. After calculating the derivatives of the variant state variables of a model, they are copied into the solver's vector $\dot{\boldsymbol{x}}^{\text{var}}$.

As defined, all variables in the sorted and solved equations must be invariant variables. This includes all input/output arguments of the predefined acausal component functions. Under these constraints it is possible to sort and solve the

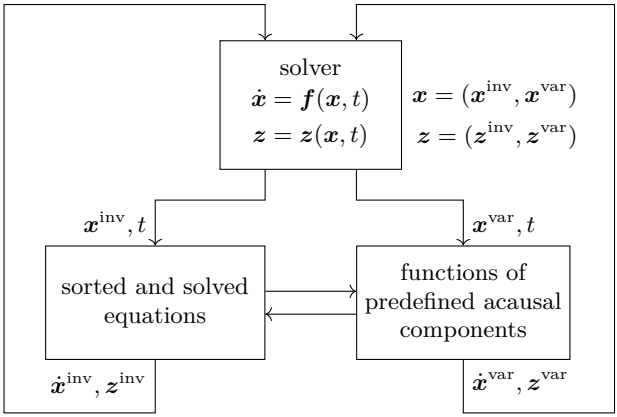


Figure 10.4: Communication between the solver, the sorted and solved equations, and the functions of the predefined acausal components. States $\boldsymbol{x}$ and event indicators $\boldsymbol{z}$ are split into an invariant and a variant part: $\boldsymbol{x} = (\boldsymbol{x}^{\text{inv}}, \boldsymbol{x}^{\text{var}})$, $\boldsymbol{z} = (\boldsymbol{z}^{\text{inv}}, \boldsymbol{z}^{\text{var}})$. The variant parts consist of the states $\boldsymbol{x}_{j,1}, \boldsymbol{x}_{j,2}$ and event indicators $\boldsymbol{z}_{j,1}, \boldsymbol{z}_{j,2}$ defined and used in the causal partitions of all predefined acausal components j, see (10.7). Before simulation starts, the dimensions of the invariant parts are fixed. During simulation, the dimensions of the variant parts can vary at events.

equations of all components as well as predefined acausal components to generate and translate code. Furthermore, two conditions must be fulfilled:

1. The symbolic transformation algorithms treat an array as one symbol (see Chapter 9).
2. No function of a predefined acausal component is differentiated.

As a consequence, all names, types and number of dimensions of all interface variables of a component ($\boldsymbol{u}, \boldsymbol{y}, \boldsymbol{p}, \boldsymbol{c}_p, \boldsymbol{c}_f$ of (10.1)) must be defined before code generation starts and cannot be varied afterwards.

Summarizing, there are two prerequisites towards variable structure systems. First, predefined acausal components are acausal components using pre-translated functions with an internal memory and remaining equations to be solved with the overall system. Second, states and event indicators are split into invariant and variant parts. Furthermore, the variant parts are hidden in an additional memory and are not visible in the model equations, but they are still passed on to the solver.

10.1.4 Application: Capacitor

For a better understanding of predefined acausal components, an academic example of a simple electrical capacitor is discussed. For comparison, there are three different versions of a capacitor model (Figure 10.5). All three versions use the three equations (10.8) for an equation-based capacitor model

$$\begin{aligned} 0 &= p_i + n_i \\ v &= p_v - n_v \\ C\,\dot{v} &= n_i, \end{aligned} \tag{10.8}$$

with potential variables p_v, n_v (electrical potentials) and flow variables p_i, n_i (electric currents), parameter C, and state v (difference between electrical potentials).

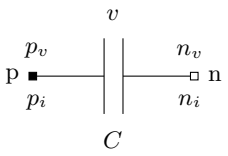


Figure 10.5: Equation-based model of a capacitor with connectors p, n.

Version 1: It is a pure equation-based model with equations (10.8). This version is discussed in Section 3.1.2, Lines 34 to 42 (Page 20).

Version 2: (Table 10.1, Left): It is an acausal component with purely mathematical functions, as introduced in Section 10.1.2.

Version 3: (Table 10.1, Right): It is a predefined acausal component with functions of a program with internal memory, as described in Section 10.1.3. It is implemented in form of pseudo code snippets. Function $f_{c,0}$ of the predefined acausal component (Table 10.1, Right) is called once during setup of the simulation run. This function allocates a data set or structure containing the internal memory $\mathtt{m}$ for the component and copies the parameters to this memory.

The formulation of a capacitor as a predefined acausal component (Version 3) has no benefits. This is because the equation part has 4 equations and the function bodies are tiny. Whereas the pure equation-based model (Version 1) consists only of 3 equations. If two capacitors, which are both defined as predefined acausal components, are also connected in parallel, both capacitors supply the respective state w_1, w_2 from function $f_{c,1}$. This results in an overdetermined equation system with three equations for two unknowns, as the parallel connection introduces a further equation $w_1 = w_2$. Such a model is therefore rejected. Due to the parallel

Table 10.1: The capacitor is defined as: (Left): a component with mathematical functions. (Right): a predefined acausal component with functions of a program with an internal memory $\mathtt{m}$ to hide the state and the state derivative of the component in the equation section.

	Component with math. functions	**Predefined Acausal Component** with functions of a program and memory
Equation section	$w = f_{c,1}(v)$ $w = p_v - n_v$ $0 = p_i + n_i$ $\dot{v} = f_{c,2}(C, n_i)$	$w = f_{c,1}(\mathtt{m})$ $w = p_v - n_v$ $0 = p_i + n_i$ $f_{c,2}(\mathtt{m}, n_i)$ # no return argument
Function definitions		**function** $f_{c,0}(\text{sim}, \text{ID}, C)$ # called once # allocate new memory $\mathtt{m}$ $\mathtt{m}_{\text{sim}} :=$ sim; $\mathtt{m}_{\text{ID}} :=$ ID; $\mathtt{m}_C = C$ **return** $\mathtt{m}$
	function $f_{c,1}(v)$ **return** v	**function** $f_{c,1}(\mathtt{m})$ # copy $\mathtt{m}_v$ from states in $\mathtt{m}_{\text{sim}}$ for $\mathtt{m}_{\text{ID}}$ **return** $\mathtt{m}_v$
	function $f_{c,2}(C, n_i)$ **return** n_i/C	**function** $f_{c,2}(\mathtt{m}, n_i)$ $\mathtt{m}_{\dot{v}} := n_i/\mathtt{m}_C$ # copy $\mathtt{m}_{\dot{v}}$ into state derivatives # of $\mathtt{m}_{\text{sim}}$ for $\mathtt{m}_{\text{ID}}$

connection, w_2 can be calculated from w_1. Thus, w_2 cannot be a state as defined in the predefined acausal component. In a more realistic simulation model, a capacitor would be implemented as in Version 1.

To sum up, a simple capacitor model is used to illustrate the principle of predefined acausal components. Furthermore, the difference between a component with pure mathematical functions and a component with functions and additional memory is shown.

10.1.5 Application: Heat Transfer in a Rod

A more realistic application – heat transfer in a rod – illustrates how to split the equations into equations which can be evaluated independently of the component's connection and an equation system which is symbolically transformed with all other equations of the overall model, i.e., in a causal and in an acausal part. Furthermore, it demonstrates how users can create their own predefined acausal components. The generic procedure is exemplified for an equation-based model of a one-dimensional heat transfer in a rod[6] with n discretized elements. This application does not use segmented simulation (Section 10.2) but it would be straightforward to extend. Neumayr and Otter (2023a) and Neumayr and Otter (2023b) already published the upcoming considerations.

All equations and parameters for this model can be found in Modelica by Example by Tiller (online). Parameters are

$$k_1 = \frac{\frac{\lambda}{\Delta x}}{c\varrho\Delta x}, \tag{10.9}$$

$$k_2 = \frac{2\lambda A}{\Delta x}, \tag{10.10}$$

where ϱ is the density, c is the specific heat capacity, and λ is the thermal conductivity. L is the length of rod. The rod is discretized in space by elements $V_i = \Delta x \cdot A$, $i = 1, \ldots, n$ of equal lengths Δx and equal areas A. In the center of element V_i, a temperature T_i with initial values $T_i(t = t_0) = T_0$ in each element is

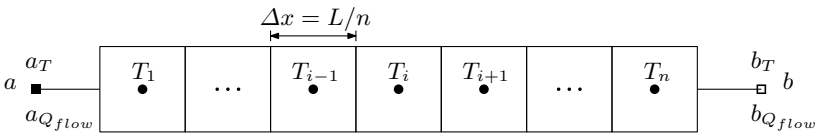


Figure 10.6: Space discretized partial differential equation of one-dimensional heat transfer in a rod with insulated surface.

[6] Modia.jl, v0.12.0, models/HeatTransfer.jl

defined. This results in a temperature array $\boldsymbol{T} = [T_1, T_2, \ldots, T_n]$. a, b are thermal connectors with potential temperatures a_T, b_T and heat flow rates $a_{Q_{\text{flow}}}, b_{Q_{\text{flow}}}$. The discretized rod is shown in Figure 10.6.

The equations for the thermal connectors a, b (10.11)

$$a_{Q_{\text{flow}}} = k_2(a_T - T_1) \tag{10.11a}$$

$$b_{Q_{\text{flow}}} = k_2(b_T - T_n) \tag{10.11b}$$

are the acausal and invariant part of the predefined acausal component. These equations (10.11) are symbolically transformed with all other equations of the overall model, since there is no prior knowledge of whether $a_{Q_{\text{flow}}}$ or a_T and $b_{Q_{\text{flow}}}$ or b_T respectively are treated as known variables.

The equations of the state derivatives $\dot{T}_i$ (10.12) of the node temperatures

$$\dot{T}_i = k_1 \begin{cases} 2(a_T - T_1) - (T_1 - T_2) & i = 1 \\ (T_{i-1} - T_i) - (T_i - T_{i+1}) & i = 2, \ldots, n-1 \\ (T_{n-1} - T_n) - 2(T_n - b_T) & i = n \end{cases} \tag{10.12}$$

are the causal part of the predefined acausal component because these equations do not depend on how the rod is connected. The potential temperatures a_T, b_T are inputs to this equation system. Thus, (10.12) is always evaluated by providing $T_i,\ i = 1, \ldots, n$ from the solver and a_T, b_T as inputs. Temperature derivatives $\dot{T}_i, i = 1, \ldots, n$ are calculated from (10.12) and communicated directly to the solver. Therefore, the number of discretized elements and thus the number of state derivatives equations can vary. These equations (10.12) are passed directly to the solver and are hidden from the sorted and solved equations of Modia.

In the here discussed application (Lines 587 to 602), the rod is completely insulated on the right-hand side and has a fixed temperature on the left-hand side. This means, the rod is connected on the left side with a fixed temperature source $T = 220\,°\text{C} = 493.15\,\text{K}$ (Line 589), and on the right side with a fixed heat flow source with $Q_{\text{flow}} = 0$ (Line 591) via thermal connectors (Lines 595 to 597). The initial temperature is $T_0 = 0\,°\text{C} = 273.15\,\text{K}$ in each discretized element. The model is symbolically transformed (Line 600) and Julia code is generated. For this predefined acausal component, it is possible to vary the number of discretized elements n, after compiling but before simulation starts. For this model, the number of discretized elements is varied from 5 to 8 (Line 602). This varies the dimension of the temperature array, and the number of state derivates (10.12). Now, the whole model is known and (10.11) can be solved. The rod is connected on the left side with a fixed temperature source. Therefore, the potential temperature a_T is known and T_1 is also provided since it is a state. Thus, $a_{Q_{\text{flow}}}$ is computed from equation (10.11a). On the right side of the rod, the heat flow rate $b_{Q_{\text{flow}}} = 0$ is known. During symbolic transformation, equation (10.11b) is transformed to $b_T := T_n$, since $b_{Q_{\text{flow}}}$ is known and T_n is a state. The temperature behavior for each discretization element is displayed in Figure 10.7.

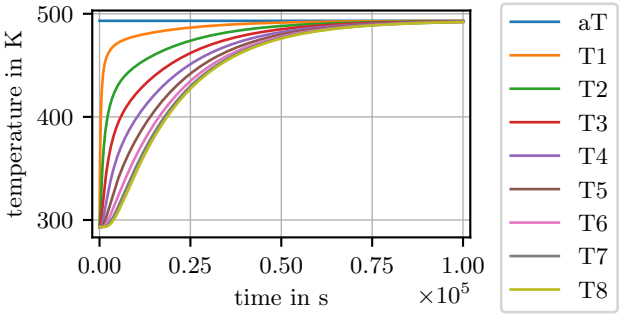


Figure 10.7: Temperatures at the temperature source and in the 8 discretized elements of heated rod.

```
587 HeatedRod = Model(
588     # temperature source
589     fixedT = FixedTemperature | Map(T=493.15),
590     # heat flow source
591     fixedQflow = FixedHeatFlow | Map(Q_flow=0.0),,
592     # insulated rod with 5 elements
593     insRod = InsulatedRod | Map(T0=273.15, n=5),
594     # connecting the components
595     connect = :[
596         (fixedT.port, insRod.portA),
597         (insRod.portB, fixedQflow.port)]
598 )
599 # generate and compile Julia code
600 heatedRod = @instantiateModel(HeatedRod)
601 # vary to 8 elements and simulate model
602 simulate!(heatedRod, stopTime = 1e5, merge=Map(insRod = Map(n=8))
```

A predefined acausal component is specified at the beginning of the model with two necessary parameters that define functions for symbolic transformation and simulation (Lines 603 to 613).

- `_buildFunction` (Line 606): Before symbolic transformation, the hierarchical dictionary of the model is run through. For each sub-model, the function defined by `_buildFunction` is executed once. First, this function defines additional model variables and equations which are merged with the corresponding model. Second, it returns an instance of the Julia structure, which acts as the internal memory of the component.
- `_initSegmentFunction` (Line 608): This function is called by the simulation engine before the model is initialized. Additionally, this function is called before the model is re-initialized in each new simulation segment if the predefined acausal component is used for segmented simulations, see

Section 10.2. In both cases, it is necessary to redefine all local variables of the predefined acausal component model, including states, zero-crossing functions, and initial values for the newly defined states.

Furthermore, the acausal component defines parameters and connectors of the component. Contrary, to a standard Modia component, no equations are defined.

```
603 # predefined acausal component
604 InsulatedRod = Model(
605     # called once before symbolic transformation
606     _buildFunction = Par(functionName = :(buildInsulatedRod!)),
607     # called before each new simulation segment
608     _initSegmentFunction = Par(functionName = :(initSegmentInsulatedRod!)),
609     # parameters
610     L = 1.0, T0 = 293.15, n = 1, A = 0.0004, rho = 7500.0, ...,
611     # connectors
612     portA = HeatPort, portB = HeatPort
613 )
```

The `buildInsulatedRod!` function (Lines 614 to 628) is defined by the first necessary parameter `_buildFunction` of a predefined acausal component. At the beginning this function is executed once. It merges the actual `InsulatedRod` component (Lines 603 to 613) with additional model definitions consisting of two variables and several equations. Finally, it returns the merged model and instantiates the internal memory of the component (Line 627). The predefined acausal component is identified by a unique `ID`. It also identifies the internal memory. The `ID` is specified in the function call. The `$ID` is inside an AST, due to `:[...]` and `$` inserts the actual (literal) value at this place. The temperature states $\boldsymbol{T}$ of the rod are copied into the `InsulatedRodStruct` memory (Line 621). The invariant equations (10.11) at the boundaries are defined in Lines 622 to 624.

```
614 function buildInsulatedRod!(model, ID, ...)
615     model = model | Model(
616         # instance of an internal struct
617         rod = Var(hideResult = true),
618         success = Var(hideResult = true),
619         equations = :[
620             # copy states into rod
621             rod = openInsulatedRod!(m, $ID)
622             # equations at the boundaries (invariant)
623             portA.Qflow = rod.k2*(portA.T - rod.T[1])
624             portB.Qflow = rod.k2*(portB.T - rod.T[end])
625             # function to evaluate state derivatives (variant)
626             success = evaluateStateDerivatives!(m, rod, portA.T, portB.T)])
627         return (model, InsulatedRodStruct())
628 end
```

The state derivatives (10.12) of the temperature are evaluated directly (Lines 630 to 639). They are copied into the state derivative vector of the internal memory.

These equations are independent from the number of discretized elements `n`. Therefore, the number of discretized elements can be changed without regenerating and recompiling.

```
# evaluate state derivatives and copy into memory
function evaluateStateDerivatives!(m, rod, aT, bT)
    T = rod.T, k1 = rod.k1, n = length(T)
    rod.derT[1]     = k1*(2*(aT-T[1])-(T[1]-T[2]))
    rod.derT[n]     = k1*(T[n-1]-T[n]-2*(T[n]-bT))
    for i in 2:n-1
        rod.derT[i] = k1*(T[i+1]-T[i]-(T[i]-T[i-1]))
    end
    # copy into memory
    copy_der_x_segmented_value_to_state(m, rod.ID, rod.derT)
end
```

The `initSegmentInsulatedRod!` function (Lines 640 to 649) is defined by the second necessary parameter `_initSegmentFunction` of a predefined acausal component. It is executed before the model is initialized and before the model is re-initialized in each new simulation segment. If it is executed for the first time, parameters (k_1, k_2) are computed once. Moreover, memory for states and their derivatives is allocated. Finally, new states and their derivatives with units are defined. Note that, even if the `InsulatedRod` component always uses the same definition, the states must be newly defined for each new segment. This predefined acausal component is not used for segmented simulations. So, it is not re-initialized.

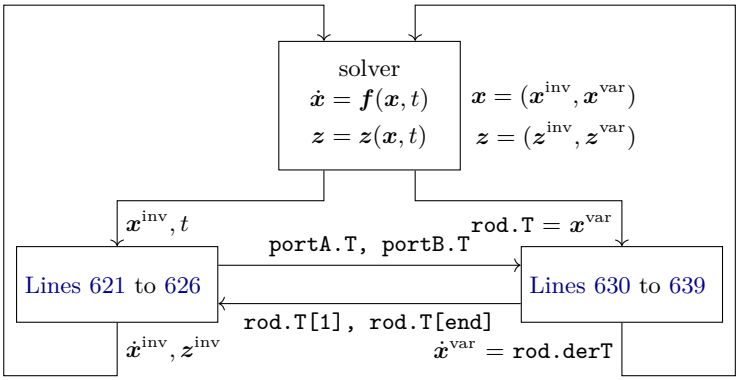


Figure 10.8: Communication between predefined acausal component for heat transfer in a rod, solver, and sorted and solved equations. The inputs and outputs for the heat transfer in a rod are independent of how the rod is connected to other components. Thus, $\boldsymbol{x}^{\text{inv}}$ is not known yet.

```
# called before each new simulation segment
function initSegmentInsulatedRod!(m, ID, ...)
    rod = getInstantiatedSubmodel(m, ID)
    if isFirstInitialOfAllSegments(m)
        # compute parameters once
        # allocate and initialize internal states
    end
    # define new states and state derivatives
    rod.ID = new_x_segmented_variable!(m, ...)
end
```

Figure 10.8 shows the inputs and outputs of the predefined acausal component (compare with Figure 10.4). For the discussed application, there are no invariant states, as the rod has a fixed temperature on one side and is insulated on the other. Still, there could be invariant states for other applications. To sum up, the here discussed procedure is generic and can be customized for suitable components, resulting in predefined acausal components.

10.2 Segmented Simulations

This section introduces a generic procedure for segmented simulation with predefined acausal components. Subsequently, it is specified for Modia. This novel procedure allows to dynamically vary the number of states during simulation, i.e., states may be added or removed. A simulation run is divided into several segments, also called modes. Events trigger the switching to a new segment. The re-initialization of a new segment, including the redefinition of states and their start values is performed on-the-fly. The segments are not necessarily known in advance.

A simulation run starts with mode $i = 1$, see Figure 10.9. The corresponding system is initialized with the initial states $\boldsymbol{x}_{1,0} = (\boldsymbol{x}_0^{\text{inv}}, \boldsymbol{x}_{1,0}^{\text{var}})$. The ODE $\dot{\boldsymbol{x}}_i = \boldsymbol{f}_i(\boldsymbol{x}_i, t)$ of the current mode i is solved until either a termination condition is reached or an event indicator for a full restart $z_{\text{fr}_i,j}$ becomes positive. In the latter case, the system switches to the next mode $i + 1$ with potentially new equations and potentially different variant states than in the previous mode. Initial values

$$\boldsymbol{x}_{i+1,0}^{\text{var}} = \Big(\boldsymbol{x}^{\text{inv}}(t), \boldsymbol{h}_i\big(\boldsymbol{x}_i(t), t\big)\Big) \tag{10.13}$$

in mode $i + 1$ are the invariant states $\boldsymbol{x}^{\text{inv}}(t)$ at the current time instant t of mode i. The initial variant states are computed with the states of mode i with function $\boldsymbol{h}_i\big(\boldsymbol{x}_i(t), t\big)$. It depends on the application of how to define function $\boldsymbol{h}_i$. The re-initialized mode is solved until it is terminated or another full restart event is triggered. The number of segments is usually not known in advance.

Re-initialization is a complex topic because Dirac impulses can occur. Benveniste et al. (2019, Section 4) give more details for a general re-initialization method for

a large class of multi-mode systems. Dirac impulses do not occur in the discussed applications in this thesis, so re-initialization in these cases is straightforward.

Since variables can appear and disappear at events and these modifications are not known in advance, new systems for storing result data are required. Otter (2022) introduces so-called signal tables as a format for exchanging data based on dictionaries and multidimensional arrays with `missing` values. This format is developed and evaluated with the open-source Julia package SignalTables.jl (Otter, online) and is used in Modia.

The general scheme presented in Figure 10.9 is specified for Modia in Figure 10.10. The internal data structure used to efficiently calculate the equations of a predefined acausal component must be updated if the number of equations varies at an event time. This internal data structure is built from the Modia model dictionary, which defines the interface and equations of the predefined acausal component. The internal data structure and the functions for its evaluation are called execution scheme.

At initialization, the model in mode $i = 1$ and its associated execution scheme is defined with the variant states and their initial values. The ODE $\dot{\boldsymbol{x}}_i = \boldsymbol{f}_i(\boldsymbol{x}_i, t)$ of the current segment i is solved for $t \in [t_{i,0}, t_{\text{stop}}]$ with its start values $\boldsymbol{x}_{i,0} = (\boldsymbol{x}_0^{\text{inv}}, \boldsymbol{x}_{i,0}^{\text{var}})$.

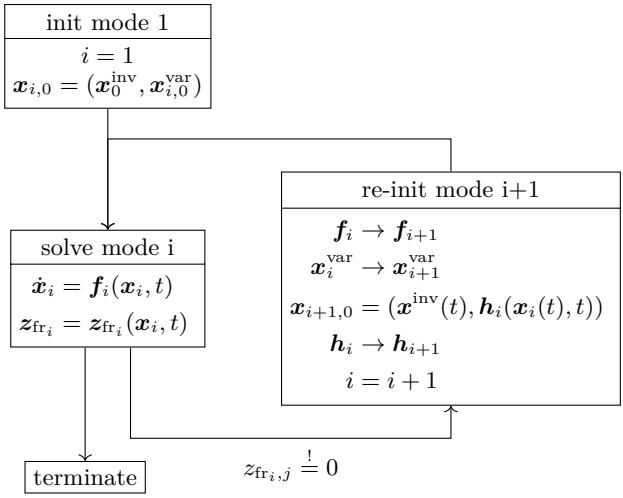


Figure 10.9: State machine of segmented simulation. The first mode $i = 1$ is initialized with its start values. The ODEs of the current mode i are solved until they are terminated or interrupted by a zero-crossing event when $z_{\text{fr}_i,j}$ changes sign. In the latter case, the model is re-initialized in mode $i + 1$. This allows variant variables to appear or disappear.

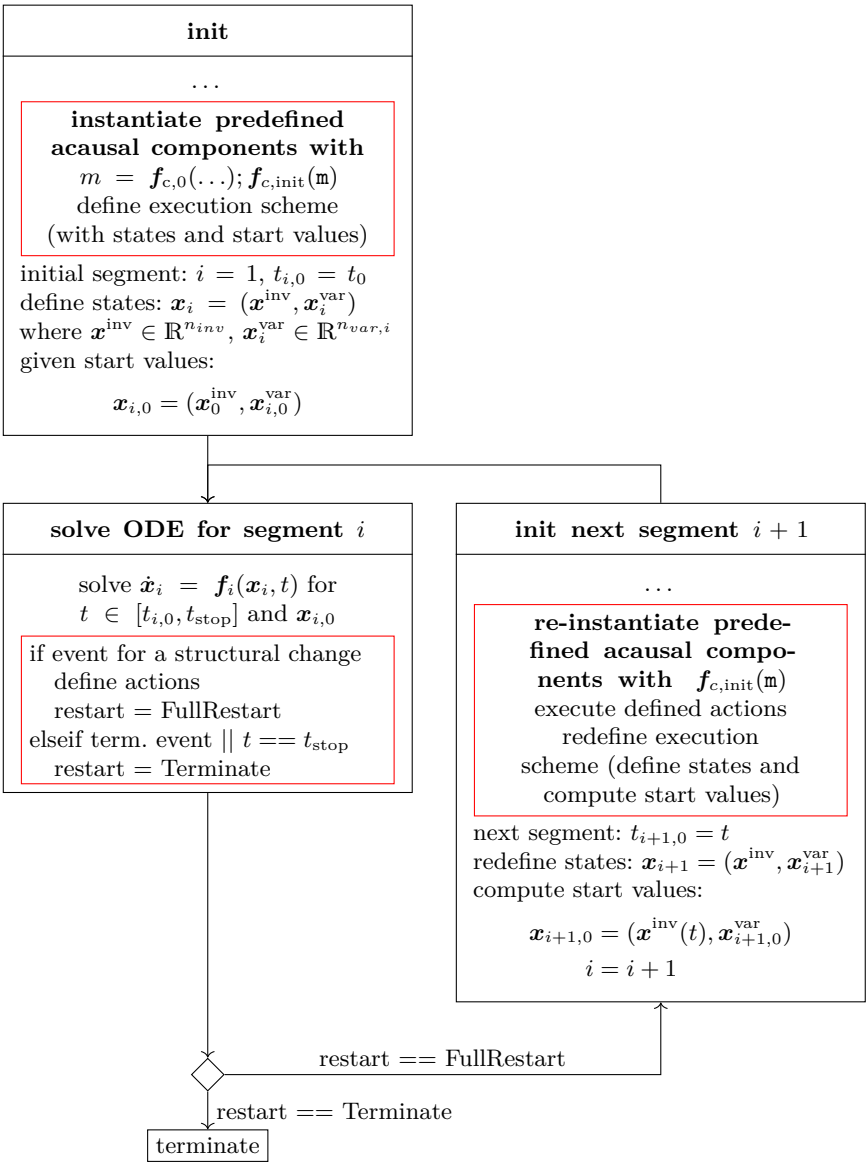


Figure 10.10: Flow chart of segmented simulation as used by Modia.

It is evaluated until t_{stop} is reached or an event for a structural change occurs. In the latter case, the so-called actions are defined and stored internally in the predefined acausal components. Actions define how to construct mode $i + 1$. Furthermore, they deal with the initialization of variant states and define $\boldsymbol{h}_i$. Actions allow users to interact with the model to initialize a new segment. For more details to execution schemes and actions for Modia3D, see Chapter 11. When the predefined acausal component is re-instantiated in mode $i + 1$, the execution scheme is redefined, including new states $\boldsymbol{x}_{i+1} = (\boldsymbol{x}^{\text{inv}}, \boldsymbol{x}^{\text{var}}_{i+1})$ and their start values $\boldsymbol{x}_{i+1,0} = (\boldsymbol{x}^{\text{inv}}(t), \boldsymbol{x}^{\text{var}}_{i+1,0})$. Then the ODE for segment $i + 1$ is solved.

Based on predefined acausal components, a generic state machine for segmented simulation is presented, where the segments need not be known in advance. This state machine is specialized for Modia. To deal with variable structure systems the user is allowed to interact and initialize new states with actions. The procedure discussed here in theory is adapted for Modia3D in Chapter 11.

11 Modia3D as Predefined Acausal Component

Modia3D is designed as a predefined acausal component of Modia to support variable structure systems.

Modia3D offers two types of joints:

1. Joints with Invariant Variables – these joints cannot be changed during simulation, see Section 3.2.3.
2. Joints with Variant Variables – these joints can be exchanged during simulation, see Section 11.1.

The second joint type can be replaced exclusively by another joint from this category. Currently, fixed joints (0 DoF) and free motion joints (6 DoF) are implemented. It is therefore possible to allow an object to move freely or to rigidly attach it to another object during simulation. The switching from free to fixed is only allowed if the relative movement is zero, to avoid Dirac impulses.

The replacement of joints is done with action commands (Section 11.3). Their sequence is defined in an action program. Only these action commands are permitted to add or remove joints from the second joint type. These actions trigger events to initialize new segments. The new states (joints) added during simulation are initialized based on the last known positions, velocities, rotations, and angular velocities. All remaining states are re-initialized with their last known values. Based on the new information about the 3D system, the internal 3D structure is rebuilt and executed until another action for a structural change is initiated. This restructuring is performed with dynamic data structures and is extremely fast. The concept of super-objects is extended (Section 11.2) since it is part of the internal 3D structure.

The interface to Modia is designed as a predefined acausal component (Lines 73 to 78, Page 26). It would go beyond the scope of this thesis to describe it in detail. The generic procedure of a predefined acausal component is exemplified in Section 10.1.5.

11.1 Joints with Variant Variables

The user has access to the second type of joints via action commands. Only these type of joints define variant variables, including variant states, according to

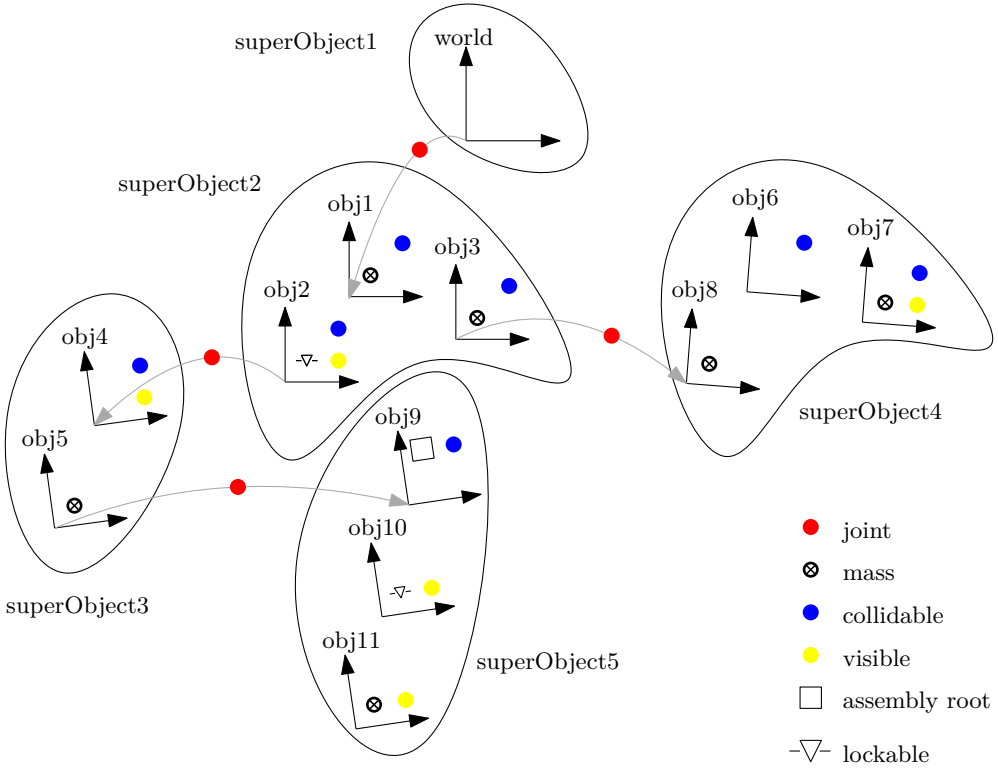


Figure 11.1: Example for an internal execution scheme. Segment 1: Before re-initialization. 12 Object3Ds with different properties are defined: They are allowed to collide, can have a mass, and are visible. They are grouped into five super-objects, which are disjunct via joints. SuperObject5 is an assembly with a lockable Object3D (obj10). Only the assembly root (obj9) is allowed to vary joints and their number of states during simulation. SuperObject2 is able to interact with this assembly via its rigidly attached locking mechanism (obj2).

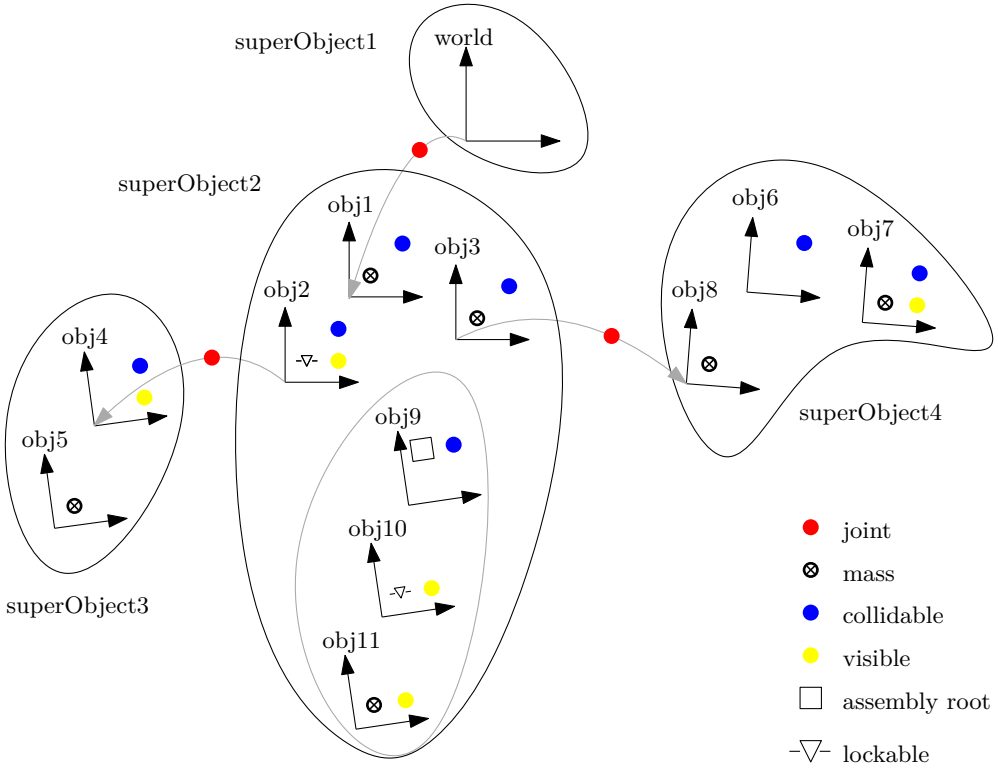


Figure 11.2: Segment 2: After re-initialization. A full restart is triggered with `ActionAttach(obj10, obj2, ...)` to initialize the second segment if both locking mechanisms (obj10 and obj2) are close to each other with negligible relative velocity. The lockable obj10 identifies its assembly root, which is obj9. The joint and states of the assembly root obj9 are removed and all Object3Ds of the assembly are attached to superObject2. This results in four rigidly attached super-objects after re-instantiation.

Figure 10.4, Page 122. The variant states and variant variables are visible only in the predefined acausal component Modia3D and are hidden from Modia. During simulation, only these variables can be added or removed via action commands. For example, an Object3D has an optional keyword `fixedToParent` with a default value of `true`. In this case, the Object3D is rigidly connected to its parent Object3D, this means it has 0 DoF. If the value is set to `false`, the Object3D is allowed to move freely with respect to its parent meaning it has six DoF and 12 variant states. At events, keyword `fixedToParent` can be changed from `false` to `true` and vice-versa.

Additionally, since Modia3D is designed as a predefined acausal component it offers two types of joints. The first type of joints contains Modia equations with invariant variables and states, see Section 3.2.3. The second type of joints defines variant variables with variant states. The user is allowed to replace the second type of joints during simulation with some special actions commands in Section 11.3 to restructure the internal tree structure in Section 11.2.

11.2 Internal Tree Structure: Restructure Super-Objects

At initialization, Modia3D's execution scheme (internal tree structure) is built up based on the Modia3D's model definitions with information about the multibody systems components (Section 3.2.4). The execution scheme is processed during the simulation of the current segment, until one of the defined actions requests a full restart in case of a structural change at an event instant or the simulation is terminated. If a full restart is required, the execution scheme is restructured, as shown in the example of Figures 11.1 and 11.2. Basically, this means that some internal data structures will be changed.

Neumayr and Otter (2023a) extend the concept of super-objects to realize segmented simulation and dynamically changing structures of 3D models. Therefore, an Object3D can be marked to be the root of an assembly or it can be marked to be lockable. An assembly is a special super-object. Only the assembly root is allowed to vary variant joints and their number of states during simulation. It interacts with other assemblies and super-objects via locking mechanisms.

As an example, Figures 11.1 and 11.2 show 12 Object3Ds. In Segment 1 (Figure 11.1) they are grouped into five super-objects before re-initialization. To dynamically switch to the second mode (Segment 2, Figure 11.2) two super-objects are rigidly connected when two respective Object3Ds are close to each other. For this purpose, both corresponding Object3Ds (e.g., obj2 and obj10) are defined as lockable. In addition, this action e.g., `ActionAttach`, must be defined in an action program. If all requirements are fulfilled a full restart is initiated resulting in a new Segment 2. During re-instantiation of Segment 2 the internal data structure

of the Modia3D predefined acausal component is regenerated resulting in four super-objects. This is a very cheap operation in the milli-seconds range.

Rigidly connected Object3Ds can form an assembly by setting `assemblyRoot = true` for the freely moving Object3D, i.e.,`fixedToParent = false`. All rigidly connected children of such an Object3D belong to the assembly. Additionally, any Object3D, whether it is part of an assembly or not, can be a locking mechanism by setting `lockable = true` in the Object3D constructor. In Figure 11.1, superObject5 is an assembly because obj9 is marked as an assembly root. This assembly is able to interact with superObject2 because both have locking mechanisms (obj2 and obj10) with `lockable = true` defined.

11.3 Action Commands

The user is allowed to interact with the model by specific action commands to initialize a new segment. Actions on a Modia3D model and especially on assemblies are executed according to the construction sketched in Lines 650 to 660. A collection of action commands is defined in a Julia function (e.g., `modelProgram`). This function is an input argument of `ModelActions` which returns a reference (e.g., `currentAction`) to an internal data structure. This reference is passed on to `executeActions` which is called in a Modia `equations` section. For applications see Chapter 12.

```
650 function modelProgram(actions)
651     ...  # action commands
652 end

653 myModel = Model3D(
654     world = Object3D(feature=Scene()),
655     ...
656     modelActions  = ModelActions(world=:world, actions=modelProgram),
657     currentAction = Var(hideResult=true),
658     equations = :[
659         currentAction = executeActions(modelActions)]
660 )
```

The action commands in Table 11.1 increase or decrease the number of DoF and therefore trigger an event to initialize a new segment. If the number of DoF increases, new states are defined and their initial values are computed based on the last configuration. Three actions: `ActionAttach, ActionReleaseAndAttach, ActionFlipChain` are only possible if the referenced lockable Object3Ds are close together and the relative velocity and angular velocity are close to zero. Currently, the following cases are treated:

- A freely moving assembly is rigidly connected to an Object3D with `Action-Attach`. This action reduces the number of DoF by 6.

- If an assembly has at least two lockable Object3Ds (objA, objB) and is rigidly connected via objA, this rigid connection is removed and another rigid connection via objB is introduced with `ActionReleaseAndAttach`. This action does not affect the number of DoF, but changes the internal data structure of the super-objects.
- A rigidly connected assembly, i.e., rigid connection to an Object3D is unlocked with `ActionRelease` to get a freely moving assembly. This action increases the number of DoF by 6.
- For `ActionFlipChain` three lockable Object3Ds (objA, objB, objC) are needed. The kinematic chain spanned between two lockable Object3Ds (objA, objB) is reversed. This means, the parent-child relationship is flipped. Hereby, special treatments of joints is required in order to correctly reflect the reversed parent-child relationship. Furthermore, a rigid connection with objA is removed and another rigid connection with objB to objC is introduced. This action does not affect the number of DoF, but changes the internal data structure of the super-objects.
- An assembly which is either freely moving or is rigidly connected to an Object3D is deleted with `ActionDelete`. All Object3Ds of this assembly are removed from the Modia3D model.

Whenever one of these actions is executed, the internal data structure with its super-objects must be restructured because the relationships and connections between parents and their children have changed. As a result of this restructuring, objects may no longer be able to collide with each other or the common mass properties of super-objects may have changed. Other Modia3D actions that do not trigger events for new segments, are listed in Table 11.2.

To initialize the next segment, a full restart is triggered in Figure 11.1, (Segment 1) with `ActionAttach(..., obj10, obj2)`. This only applies if both lockable

Table 11.1: Modia3D actions which trigger events for a structural change and initialize new segments.

Function	Description
`ActionAttach(...)`	Rigidly attaches the specified assembly.
`ActionReleaseAndAttach(...)`	Changes one rigid connection to another rigid connection.
`ActionRelease(...)`	Releases the specified assembly.
`ActionFlipChain(...)`	Reverts the kinematic chain between two lockable Object3Ds. It changes one rigid connection to another rigid connection.
`ActionDelete(...)`	Deletes the specified assembly.

Table 11.2: Modia3D actions.

Function	Description
`EventAfterPeriod(...)`	Triggers an event after a specific period of time.
`ActionWait(...)`	Waits a specific period of time.
`addReferencePath(...)`	Adds a new reference path.
`ptpJointSpace(...)`	Generates a point-to-point trajectory.

Object3Ds (obj2 and obj10) are close to each other. Hereby, the joint connecting obj9 with obj5 is removed and obj10 is rigidly connected to obj1 which is the root of superObject2. The re-instantiation reduces the number of super-objects and states. It results in the execution scheme of Figure 11.2 (Segment 2). All remaining states are re-initialized with their last known values. All action commands of Table 11.1 are extended with an additional keyword `enableContactDetection` defaultly set to `true`. This allows to switch off and on collision detection for the whole scene for the actual segment, this might speed up simulation time, (see Neumayr and Otter, 2023b).

To sum up, Modia3D is adapted to enable segmented simulation. Thus, the internal tree structure of Modia3D is extended to re-instantiate it during a simulation. This procedure can be customized to adapt Modia3D to the user's needs with individual action commands.

12 Applications

Modia3D is designed as a predefined acausal component of Modia. The following applications describe dynamically changing the variable structure during simulation from the user's point of view. Therefore, the user needs to be sufficiently familiar with the system and its required conditions to interact via certain (user-defined) actions that trigger a re-instantiation of a new 3D segment. However, the segments need not be known in advance.

From the user's point of view, a more involved application of segmented simulations deals with a two-stage rocket, where the two stages are separated after a while in Section 12.1. The second application, is an insight using segmented simulation for collision handling. For this purpose, a robot gripping a sphere is modeled in several ways. The outcome is compared and discussed in Section 12.2. In the third application a kinematic chain is reverted to re-initialize a new segment in Section 12.3.

12.1 Segmented Simulation: Two-Stage Rocket

An application of a two-stage rocket[2] highlights Modia3D's actions for segmented simulation. Moreover, it exemplifies how to set up a Modia3D model. This model simulates the separation of the two stages, Figure 12.1. It is already published by Neumayr and Otter (2023a). To demonstrate how to delete Object3Ds, the first

a) At initialization. b) After separating (release).

Figure 12.1: A two-stage rocket[1]. Stage 1 is the lower left part and stage 2 the upper right part after separation.

[1] The visualization data of ESA's VEGA rocket (Arianespace, 2014) is used for the sole purpose of providing a better illustration. The visualization data is taken from Turbosquid.com (online, purchased in 2015).

[2] Modia3D.jl, v0.12.0, test/Segmented/TwoStageRocket3D.jl

stage is removed some time after separation because it is no longer of interest in this scenario. In Figure 12.1b, stage 1 is the lower left part and stage 2 is the upper right part of the rocket.

The thrust of stage 1 drives both rigidly attached stages until separation. Afterwards, stage 2 is driven by its own thrust. Stage 1 has no thrust anymore and loses vertical velocity until it is removed, see Figure 12.2.

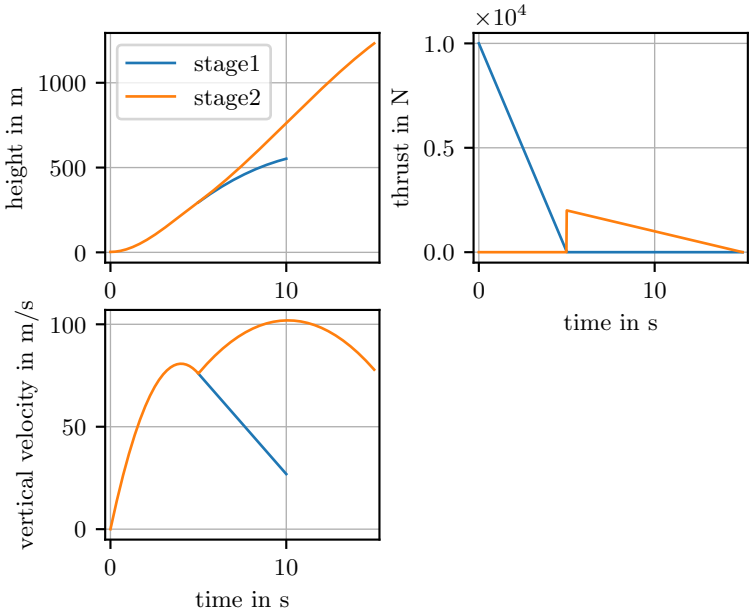


Figure 12.2: If present, the height, vertical velocity and thrusts of both stages are shown. The thrust of stage 1 drives both rigidly attached stages. So, they have the same height and vertical velocity. After separation (5 s), stage 2 gains height and velocity by its own thrust. Stage 1 has no longer a thrust and loses vertical velocity until it is removed after 10 s.

Each rocket stage is modeled using sub-model `RocketStage` (Lines 673 to 688), which consists of not only mass/inertia and lockable Object3Ds at the top and at the bottom but also has a thrust at the bottom. Model `TwoStageRocket` (Lines 689 to 698) builds up the rocket system with a `world` Object3D and two instances of `RocketStage`. To execute the defined actions of the `rocketProgram` (Lines 661 to 671) they are passed to the constructor `ModelActions`. For simulating the

separation process of the two stages, the following actions to dynamically vary DoF during the simulation are needed.

- Segment 1: Initially, the two stages are not rigidly connected.
- Segment 2: At initialization, the top of stage 1 and the bottom of stage 2 are rigidly attached with `ActionAttach(actions, "stage1.top", "stage2.-bottom")`.
- Segment 3: To separate the two stages, an event is triggered after 5 s with `EventAfterPeriod(actions, 5)`. It is needed to release `stage1.top` from `stage2.bottom` with function `ActionRelease(actions, "stage1.-top")`. Furthermore, thrust 1 is switched off.
- Segment 4: To remove stage 1 from the simulation run, another event is triggered after 5 s (so at t = 10 s). Since the movement of stage 1 is no longer of interest in this scenario, `stage1.top` and all Object3Ds connected to it are deleted with `ActionDelete(actions, "stage1.top")`.

For further information to Modia3D's actions see Chapter 11.

```
function rocketProgram(actions)
    # segment 1 (from initialization)
    # segment 2
    ActionAttach(actions, "stage1.top", "stage2.bottom")
    EventAfterPeriod(actions, 5)          # Trigger an event after 5s
    # segment 3
    ActionRelease(actions, "stage1.top")  # Release stage1.top
    EventAfterPeriod(actions, 5)          # Trigger an event after 5s
    # segment 4
    ActionDelete(actions, "stage1.top")   # Delete stage1
end

# L ... length of stage, m ... mass, r_init ... initial translation
RocketStage(; L=1.0, m=100.0, r_init, filename, thrustFunction) = Model(
    # rocket stage with mass points
    body = Object3D(parent=:world, fixedToParent=false, translation=r_init,
        assemblyRoot=true, feature=Solid(
        massProperties=MassProperties(mass=m, ...)),
    # visualization data of a VEGA rocket
    visualVega = Object3D(parent=:body,
        feature=Visual(shape = FileMesh(filename=filename, ...))),
    # lockable Object3Ds at bottom and top
    bottom = Object3D(parent=:body, lockable=true,
        translation=:[0.0, -$L/2, 0.0]),
    top = Object3D(parent=:body, lockable=true,
        translation=:[0.0,  $L/2, 0.0]),
    # thrust is applied at bottom
    thrust = WorldForce(objectApply=:bottom, forceFunction=thrustFunction)
)

TwoStageRocket = Model3D(
```

```
690     world  = Object3D(feature=Scene()),
691     stage1 = RocketStage(L=2.0, r_init=[0,1,0],
692         filename="Interstage1.obj", thrustFunction=thrust1),
693     stage2 = RocketStage(L=1.0, r_init=[0,2.5,0],
694         filename="Interstage2.obj", thrustFunction=thrust2),
695     modelActions = ModelActions(world=:world, actions=rocketProgram),
696     currentAction = Var(hideResult=true),
697     equations = :[currentAction = executeActions(modelActions)]
698 )
```

12.2 Segmented Simulation and Collision Handling: Gripping Robot

This section emphasizes the new approach of segmented simulation to handle collisions. Therefore, several combinations of segmented simulation and collision handling with contact computation are discussed and compared. In all of these applications, a KUKA YouBot robot (youbot-store, online) is gripping and transporting cargo, see Figure 12.3. This robot has a 5 DoF arm and was manufactured in the years 2010–2016. The robot is modeled with Modia, especially its electrical drive trains and controllers. The 3D mechanics is modeled with Modia3D, (see Elmqvist, Otter, Neumayr, and Hippmann, 2021). The following comparison of scenarios is inspired by applications from Neumayr and Otter (2023a) and is preliminarily published by Neumayr and Otter (2023b).

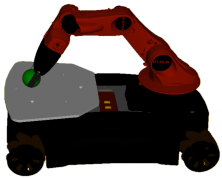


Figure 12.3: YouBot gripping or releasing a sphere on a plate.

Four variants of the following transportation scenario are simulated. In all these scenarios, the robot follows the same trajectory. Initially, the cargo, e.g., a sphere, rests on a plate. It is gripped by the robot's gripper and transported upwards until it is placed down again, where it rests on the plate until it is gripped again. The states of the freely moving sphere, if available, and the absolute position of the sphere center are displayed in Figure 12.4.

Scenario 1 (S1)[3]: This transportation scenario is modeled exclusively with collision handling, i.e., contacts are computed with the MPR algorithm. This

[3] Modia3D.jl, v0.12.2, test/Robot/ScenarioCollisionOnly.jl

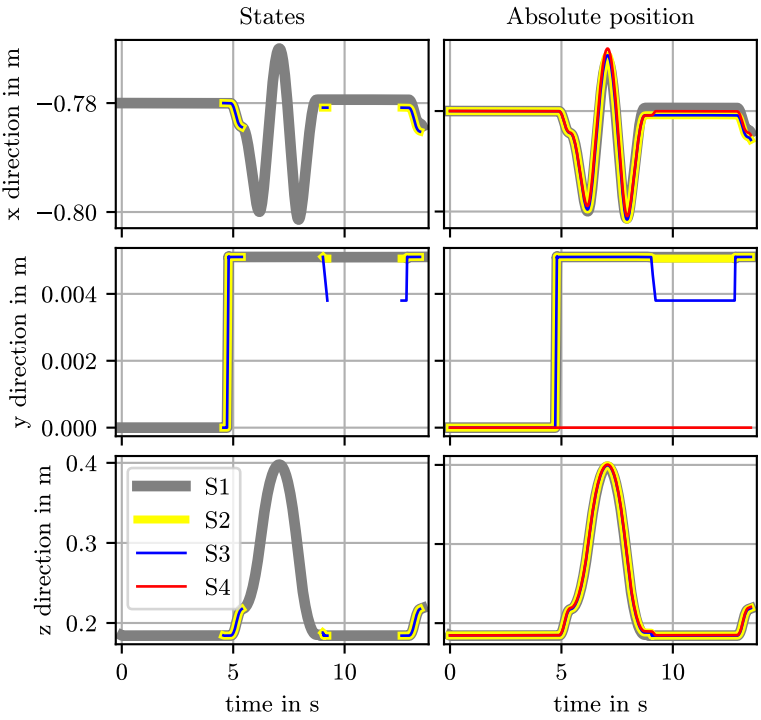


Figure 12.4: States and absolute position of the sphere of all four Scenarios (S1–S4). The states are equivalent to the translation of the sphere center in x, y, z direction with respect to its parent. If the sphere is freely moving, world is its parent. States can only be displayed if they are present. If the sphere is rigidly attached to the plate or gripper, there are no states, and nothing is displayed. The sphere in Scenario 4 (S4) has no states. Therefore, nothing is displayed. The states for Scenarios 2 and 3 (S2, S3) are available and are displayed if the sphere is freely moving.

means, the sphere is in contact with the plate, as well as with the fingers of the gripper. This scenario is similar to Neumayr and Otter (2023a, Scenario 2(b)).

Scenario 2 (S2)[4]: This transportation scenario is modeled with both collision handling with contact computation and segmented simulation (Lines 699 to 738). DoFs are added or removed during simulation: At the beginning, the sphere is rigidly attached to the plate. Shortly before the gripper reaches the sphere, the

[4]Modia3D.jl, v0.12.2, test/Segmented/ScenarioSegmentedCollisionOn.jl

sphere is released (+6 DoF) and collides with the plate. Shortly afterwards it collides with the gripper. After approximately one second, the sphere is rigidly attached to the gripper (−6 DoF). Until the gripper is again close to the plate to release the sphere (+6 DoF), which collides with the plate. Collision handling remains on even if the sphere is rigidly connected to the gripper or plate, as collisions with other bodies can still occur.

Scenario 3 (S3)[5]: This transportation scenario is also modeled with both collision handling with contact computation and segmented simulation, except that collision handling is disabled when the sphere is rigidly connected to the gripper or plate, since it is already known from Scenario 2 that no further collisions will occur. This is deactivated with `enableContactDetection = false` (Lines 699 to 738). Basically, this means that the distance calculations between each collision pair is switched off in these phases.

```
699 # Scenario 2 & 3
700 function robotProgram(actions)
701     # segment 1 (from initialization)
702     addReferencePath(actions, ...)
703     # segment 2
704     # 1. attach sphere to plate, -6 DoF
705     ActionAttach(actions, "sphereLock", "robot.base.plateLock",
706         # enableContactDetection = false)
707     # 2. some movement of robot
708     ptpJointSpace(actions, [
709         # open gripper + move to top
710         # open gripper + move to plate])
711     # segment 3
712     # 3. release sphere off plate, +6 DoF
713     # it collides with plate and gripper
714     ActionRelease(actions, "sphereLock")
715     # 4. gripping via collision handling
716     ptpJointSpace(actions, [
717         # grip
718         # grip + transport a bit      ])
719     # segment 4
720     # 5. attach sphere to gripper, -6 DoF
721     ActionAttach(actions, "sphereLock", "robot.gripper.gripperLock",
722         # enableContactDetection = false)
723     # 6. some movement of robot with sphere
724     ptpJointSpace(actions, [
725         # grip + move to top
726         # grip + transport
727         # grip + move near to plate
728         # open gripper                ])
729     # segment 5
730     # 7. release sphere off gripper, +6 DoF
731     # it collides with plate
```

[5] Modia3D.jl, v0.12.2, test/Segmented/ScenarioSegmentedCollisionOff.jl

```
732     ActionRelease(actions, "sphereLock")
733     # 8. some movement of robot
734     ptpJointSpace(actions, [
735         # open gripper + move to plate])
736     # repeat step 1. - 8.
737     ...
738 end
```

Scenario 4 (S4)[6]: This transportation scenario is modeled exclusively with segmented simulation (Lines 739 to 766). This scenario is similar to Neumayr and Otter (2023a, Scenario 2(a)). Collision handling with contact computation is switched off for this scenario. This means, the sphere is rigidly attached to the plate, when resting, and rigidly attached to the gripper during transportation. Each time the sphere is rigidly connected to the plate or gripper, the segment is re-initialized. Since the relative velocity and angular velocity between the sphere and the gripper is zero, when the sphere is attached to the gripper or attached to the plate, the physics is correctly modeled under the idealized assumption that gripping time is infinitely small. Basically, this means that gripping effects are neglected.

```
739 # Scenario 4
740 function robotProgram(actions)
741     # segment 1 (from initialization)
742     addReferencePath(actions, ...)
743     # segment 2
744     # 1. attach sphere to plate
745     ActionAttach(actions, "sphereLock", "robot.base.plateLock")
746     # 2. some movement of robot
747     ptpJointSpace(actions, [
748         # open gripper + move to top
749         # open gripper + move to plate
750         # grip                        ])
751     # segment 3
752     # 3. attach sphere to gripper
753     ActionAttach(actions, "sphereLock", "robot.gripper.gripperLock")
754     # 4. some movement of robot
755     ptpJointSpace(actions, [
756         # grip + transport a bit
757         # grip + move to top
758         # grip + transport
759         # grip + move near to plate
760         # open gripper                ])
761     # segment 4
762     # 5. release sphere off gripper, attach it to plate
763     ActionReleaseAndAttach(actions, "sphereLock", "robot.base.plateLock")
764     # repeat step 2. - 5.
765     ...
766 end
```

[6]Modia3D.jl, v0.12.2, test/Segmented/ScenarioSegmentedOnly.jl

Table 12.1: Mean $\bar{t}_{\text{sim}}$ and standard deviation σ of the simulation time t_{sim} of all four Scenarios (S1–S4) each for $n = 12$ runs on a standard notebook[7].

	$\bar{t}_{\text{sim}}$	σ
S1	7.816 s	0.123 s
S2	7.255 s	0.075 s
S3	6.863 s	0.388 s
S4	0.397 s	0.016 s

The simulation times t_{sim} of all four scenarios are compared in Table 12.1. The simulation time of Scenario 4 is about 19 times less than that of Scenario 1. This is because Scenario 4 (segmented simulation exclusively) is basically a non-stiff system where the solver can use large step sizes. In addition, the time for reconfiguration of the multibody system, for gripping and releasing, is negligible. Fine-tuning of collision handling during transportation of the gripped freight is no longer required. This makes the simulation faster and more robust. Moreover, the scenario setup is easier. Furthermore, any type of cargo can be transported, regardless of its shape and collision problems are avoided. It is also possible to model gripping operations without frictional contacts but with rigid mechanical connections, such as a bayonet lock. The disadvantage is that the details of the gripping are not modeled, but this can be important, e.g., when developing a control system to perform an assembly task.

Scenario 1 (collision handling exclusively) is a stiff system because the gripper holds the sphere by elastic contact and friction forces, which change during transport. Therefore, the solver must use much smaller step sizes. One limitation of collision handling with the MPR algorithm is that it solely computes point contacts. If the cargo would be a box, it would not be possible to calculate a unique point contact that is continuous over time, for example, because one box and one gripper face or one box and one plate face are parallel to each other during contact (Neumayr and Otter, 2023a, Scenario 3(b)). All these considerations lead to a compromise in modeling the gripping and releasing of the cargo with collision handling, and otherwise rigidly attaching the sphere to the plate or gripper, resulting in Scenario 2 and Scenario 3.

There is not such a big difference in simulation time for Scenarios 1, 2, 3, see Table 12.1. In all three cases, the calculation of the elastic contact response is the limiting factor. This effect is modeled in all these cases. In more realistic scenarios, the approach of Scenario 2 or 3 may pay off, if the number of collision phases is small relative to the remaining actions.

[7] Intel(R) Core(TM) i7-9850H CPU @ 2.6 GHz, RAM 32 GB

12.3 Segmented Simulation: Relocatable Space Robot

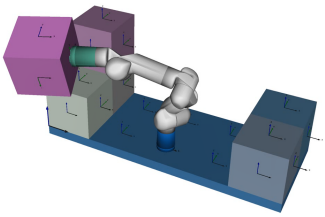
An application of a relocatable space robot (Deremetz et al., 2020) highlights the action for flipping a kinematic chain with segmented simulation. The symmetric, 7 DoF robotic manipulator is part of the MOSAR space project (Modular and Re-Configurable Spacecraft, (see Letier et al., 2019)). The space robot consists of one arm with 7 joints, and two end effectors. It allows to capture, manipulate and position spacecraft modules. The robot is able to relocate itself on the interfaces of the spacecraft or on the modules, marked with coordinate systems in Figure 12.5. The visualization data and trajectory for each joint of the space robot are taken from Reiner (2022). The drive of each joint includes gear dynamics modeled by a spring-damper pair with Modia. The 3D mechanics is modeled with Modia3D.

The following model simulates the described behavior above. In more detail, a robot is placing modules and is walking on a platform, e.g., a spacecraft, using the end effectors of its arm to achieve this in Figure 12.5. The model demonstrates that the robot is capable of gripping the modules with either one of its end effectors. Furthermore, it showcases the robot's ability to alternate between attaching one of its end effectors to the platform. This enables the robot to *walk*. In doing so, the kinematic chain of the robot's joints across its span of arm must be reversed. This effectively means that the parent-child relationship between Object3Ds is flipped. Special treatments of the joints are required in order to correctly implement this.

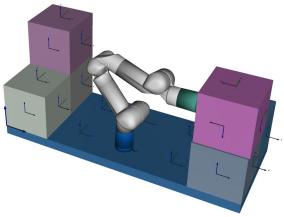
The platform program for the robot and six modules is sketched in Lines 768 to 799. Hereby, segments 9–12 (Lines 778 to 794) correspond to Figure 12.5a – Figure 12.5d. To distinguish between the robot's two end effectors, one is colorized blue while the other is colorized green. The modules are boxes. The two end effectors and the interfaces of the boxes are lockable Object3Ds. In order to interact with the robot and the six boxes using action commands, they are not rigidly attached to the platform during initialization. In segment 2, the blue end effector is rigidly attached to the platform. In segments 3–8, the six boxes are rigidly attached to the platform. In segment 9 and 10, the green end effector is moving and replacing a box. In segment 11, the robot is walking. This means, the attachment to the platform alternates between the blue and the green end effector. The blue one is released and the green one is attached. Additionally, the kinematic chain spanned between the end effectors is reversed. In segment 12, the blue end effector is gripping a box.

The relocatable space robot places two boxes and walks on the platform. This scenario lasts 86 s and the simulation is performed in 2.2 s. This is much faster than real-time, since collision handling with point contacts is neglected. Moreover, it is impossible to represent collisions between two parallel surfaces with a collision algorithm that computes point contact like the MPR algorithm. For a detailed discussion see Section 8.3.

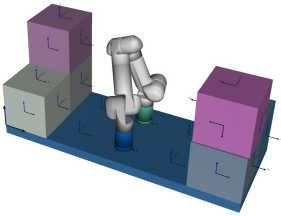
To summarize, this application demonstrates another possibility by not only adding or removing joints, but also reversing an entire kinematic chain.



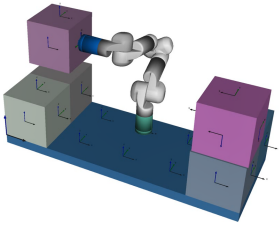
a) Segment 9.



b) Segment 10.



c) Segment 11.



d) Segment 12.

Figure 12.5: Walking robot on a platform. The boxes are placed by two end effectors of a robot arm. Thus, one end effector of the arm is attached to the platform and stationary, while the other one is able to move the boxes. In segment 11, the robot is walking. To do so, the purposes of the rigidly attached end effector and the moving one are reversed.

```
767 function platformProgram(actions)
768     # segment 1 (from initialization)
769     # segment 2
770     # attach blue end effector to platform
771     ActionAttach(actions, "blueLock", "platform.lockX2Y2")
772     EventAfterPeriod(actions, 1e-10)
773     # segment 3 - 8
774     # attach 6 boxes to platform
775     ActionAttach(actions, "boxX1Y1Z1.lockZneg", "platform.lockX1Y1")
776     ...
777     EventAfterPeriod(actions, 7.0)
778     # segment 9
779     # attach box to green end effector
780     ActionReleaseAndAttach(actions, "boxX1Y1Z2.lockXpos", "greenLock")
781     EventAfterPeriod(actions, 17.0)
782     # segment 10
783     # release box off green end effector, attach box to another box
784     ActionReleaseAndAttach(actions, "boxX1Y1Z2.lockZpos", "boxX5Y1Z1.lockZpos")
```

```
785     EventAfterPeriod(actions, 6.0)
786     # segment 11
787     # attach green end effector to platform
788     # flip kinematic chain between blue and green end effector
789     ActionFlipChain(actions, "greenLock", "platform.lockX2Y2", "blueLock")
790     EventAfterPeriod(actions, 14.0)
791     # segment 12
792     # attach box to blue end effector
793     ActionReleaseAndAttach(actions, "boxX1Y2Z2.lockXpos", "blueLock")
794     EventAfterPeriod(actions, 23.0)
795     # segment 13
796     # release box off blue end effector, attach box to another box
797     ActionReleaseAndAttach(actions, "boxX1Y2Z2.lockZpos", "boxX5Y2Z1.lockZpos")
798     ...
799 end
```

To conclude this chapter, all applications combine actions with and without re-initializations of new segments. The application of the two-stage rocket shows not only attaching and releasing components, but it is also possible to delete them if needed, e.g., to avoid computational effort. Moreover, to get deeper observations of applications, it is reasonably simple to extend existing action commands with further features, e.g., to enable or disable collision handling for the actual segment and thus for the whole model. In addition, it is not only possible to add or remove specific connections, it is even possible to revert a kinematic chain. To retrieve the most out of the applications, users must analyze their models for suitable separation and attachment timings and relative positions between components.

13 Conclusion and Outlook

13.1 Conclusions

Conclusion to Collision Handling with Variable-Step Solvers
Collision handling is widely researched with extensive literature but usually exclusively for fixed-step solvers without step size control. There are several algorithms available to compute contact information for different collision aspects. In this thesis a penetration depth or the Euclidean distance and contact points between two potentially-colliding convex objects or their convex hulls are computed. If these objects penetrate, the resulting force is determined using elastic material laws based on the theory of Hertz for elastic shapes. On the one hand, the contact information is sufficient to be used with variable-step solvers and these algorithms are fast. But on the other hand, it is a simplification of collision situations and a loss of contact information. Therefore, not all collision situations can be treated properly. Furthermore, it must be ensured, that there are continuously varying contact points and penetration depths. Hence, it must be ensured that this condition is met. A detailed discussion can be found in Section 8.3.

Conclusion to Variable Structure Systems
A novel approach for variable structure systems is introduced in this thesis. This approach minimizes or completely eliminates time-consuming distance calculations. Therefore, it uses the inherent mathematical structure of acausal components, leading to predefined acausal components. The equations of acausal components are split into acausal and causal parts. The causal equations are pre-translated functions. The states computed by these pre-translated functions are hidden in a memory and are directly passed to the solver. The states of the causal equations of predefined acausal components are allowed to be added or removed during simulation. The acausal equations of the component need to be solved with the whole model. One challenge is to analyze and split the component's equations into these two parts. Another challenge is to interact with the predefined acausal components via suitable action commands. Depending on the action command a new segment (mode) is triggered to add or remove states.

The Modia3D package is designed as predefined acausal component for Modia. Modia3D currently supports some actions for adding or removing rigid connections to free motion joints, which adds or removes 6 Degrees of Freedom (DoF), or changing one rigid connection to another rigid connection. The initiation of a new segment is what all of these actions have in common. The segments themselves do

not need to be known in advance. Furthermore, the extension of functionality with additional actions for other types of joints is an open research question.

This new method is validated and seems very promising for future applications. Moreover, it can be combined with collision handling with variable-step solvers to achieve the optimal benefits of each. This opens up a wide range of exciting new possibilities. However, the limitations for practical applications are to be examined.

13.2 Outlook

In light of the above discussions and limitations, further research is necessary. When it comes to collision handling, the narrow phase in particular can benefit from thinking outside the box. Would it make sense to use two different algorithms in the narrow phase? One algorithm for point contacts and one for non-convex contacts. This would make it possible to address the various issues regarding non-convex shapes, different types of shapes, and collision situations. Is it possible to combine both kinds of algorithms in the narrow phase? Firstly, to figure out how much both shapes overlap with a point contact algorithm and secondly if required use a non-convex contact algorithm. Would it make sense to compute some solver steps with a non-convex contact algorithm, if a discontinuous jump occurs? Perhaps it would be possible to calculate the collisions with the MPR algorithm until some issues are encountered for a specific collision pair and to recalculate the setting with a non-convex contact algorithm. In this case, another force law would be needed as well.

Some new research questions arise for the novel approach for variable structure systems. How are new states initialized during simulation while these are so strongly dependent on the model and applications? For example, in Modia3D the new states are added based on the last known configuration, i.e., position, rotation, and velocity. Another question that arises, is whether it is possible to initialize new states by interpolation. For example, when thinking of increasing the discretization of the rod for a heat transfer by following a known temperature profile. Is it possible to idle parts of the model, and thus states? What does this mean when reactivating it? How to re-initialize it? Should it be based on the last values, before idling it, or is it more like newly introducing states? At the moment, structural changes are triggered by action commands from outside. Currently, it is not possible to initiate a structural change from the model itself, but it is planned to introduce this feature to Modia3D. How can this novel approach be used in Modelica?

A Mathematical Definitions

Some mathematical definitions and theorems are given without proofs.

Implicit and Explicit Functions, and Solvers

The following notations are from Bartsch (2007).

Analytical Notation as Functional Equations

- Implicit notation: $F(x, y) = 0$.
- Explicit notation: $y = f(x)$.
- Parameter notation: $x = x(t), y = y(t)$.

Solvers

- Explicit methods: An explicit solver computes the unknown variables at the current time step, based on values of previous time steps.
- Implicit methods: An implicit solver computes the unknown variables at the current time step, based on values of previous time steps and the current time step. The solution can therefore only be determined iteratively.

Convex Sets and Polytopes

The following definitions and theorems are from Grünbaum and Shephard (1969), Grünbaum and Shephard (1969) and Brondsted (2012) where further details can be found. A subset $S \subseteq \mathbb{R}^n$ is called a convex set, if the line segment

$$\lambda \boldsymbol{x} + (1 - \lambda)\boldsymbol{y} \in S$$

lies entirely in S for all $\boldsymbol{x}, \boldsymbol{y} \in S$ and $0 \leq \lambda \leq 1$.

A point

$$\boldsymbol{x} = \sum_{i=1}^{n} \lambda_i \boldsymbol{x}_i,$$

with $\sum_{i=1}^{n} \lambda_i = 1$ and $\lambda_i \geq 0, \forall i = 1, \ldots, n$, is called a convex combination for $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^n$.

A subset $S \subset \mathbb{R}^n$ is convex iff any convex combination of points from S is again in S. For any set S, the smallest convex set containing S is called the convex hull of S and denoted by $\mathrm{conv}\,(S)$. A convex polytope, or simply a polytope, is the convex hull of a finite set of points. The notation $\mathrm{vertex}(S)$ is used to refer to the vertices of the convex hull of S. Some illustrative explanations in metric space are: closed means that the boundary is part of the object. Bounded means that there exists a sphere of finite radius enclosing the object.

B Applications: Collision Handling with Variable-Step Solvers

The following applications[1] for collision handling with variable-step solvers are analyzed in Section 6.1.4 for the MPR algorithm and are briefly discussed here. It is determined during simulation, if Object3Ds collide with each other and the collision response is computed. Bounding boxes are used for each collision model, but they are only visualized in Figure B.1 with light grey boxes. To get a better impression of colliding objects and their trajectory, their orientation is highlighted with coordinate systems.

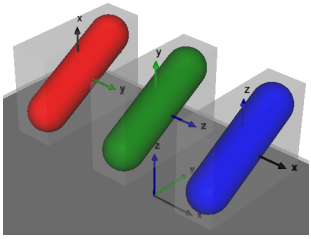


Figure B.1: BouncingCapsules.jl: Three capsules fall downwards due to gravity and have 6 DoF each. They collide with a plate. Each capsule is initialized with a different axis rotation.

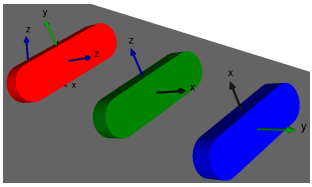


Figure B.2: BouncingBeams.jl: Three beams fall downwards due to gravity and have 6 DoF each. They collide with a plate. Each beam is initialized with a different axis rotation.

[1] Modia3D.jl, v0.10.2, test/Collision/

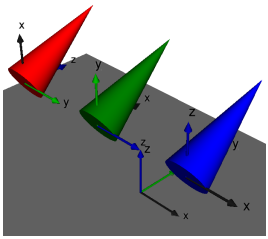


Figure B.3: BouncingCones.jl: Three cones fall downwards due to gravity and have 6 DoF each. They collide with a plate. Each cone is initialized with a different axis rotation.

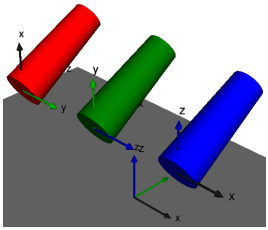


Figure B.4: BouncingFrustums.jl: Three frustums fall downwards due to gravity and have 6 DoF each. They collide with a plate. Each frustum is initialized with a different axis rotation.

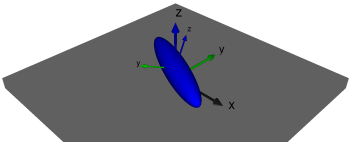


Figure B.5: Rattleback.jl: The rattleback is an ellipsoid that collides with a plate. It has 6 DoF and initial angular velocities in y, z direction. The rattleback rotates counterclockwise and wobbles, until it turns and rotates clockwise and wobbles.

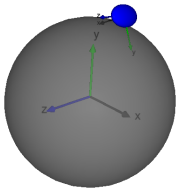


Figure B.6: BouncingEllipsoidOnSphere.jl: An ellipsoid (6 DoF) falls downwards due to gravity. It collides with a sphere. It turned out that this application requires many refinement iterations to determine the closest distance with the MPR algorithm in the narrow phase.

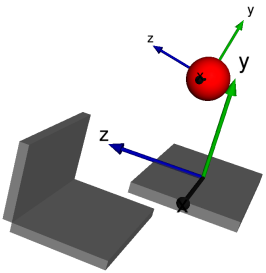


Figure B.7: BouncingSphereFreeMotion.jl: A sphere (6 DoF) falls downwards due to gravity. It collides several times with the upper plate, the wall on the left, and the lower plate again.

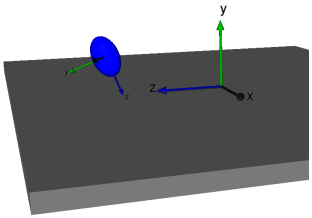


Figure B.8: BouncingEllipsoid.jl: An ellipsoid (6 DoF) falls downwards due to gravity and collides with a plate.

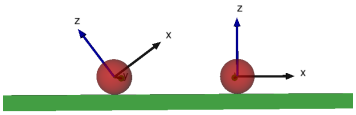


Figure B.9: TwoCollidingBalls.jl: This model analyzes the sliding and rolling of balls. It is preliminary work for a billiard game in Figure B.10. Two balls, each has 6 DoF, lie on a table with a gap between them. The left ball has an initial velocity. The effects of sliding and rolling occur, and after some time it hits the other ball at rest.

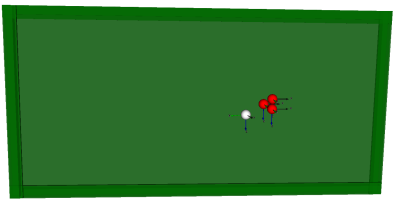


Figure B.10: Billard4Balls.jl: At initialization, four billiard balls are placed in touching position with the table. The balls subside immediately, because of gravity. The cue ball has an initial velocity. The effects of sliding and rolling occur, and after some time it hits the rack at rest. A billiard game with 16 billiard balls is explained in more detail in Section 8.1.

Figure B.11: CollidingSphereWithBunnies.jl: It demonstrates how to handle collisions with concave objects. For the upper bunny the convex hull is taken, for the lower bunny a convex decomposition is used. A sphere (6 DoF) falls downwards due to gravity. It collides with the ear of the grey bunny. To be more precise, it collides with the convex hull of the grey bunny. Then, the sphere collides with the bottom of the lower bunny. The lower bunny is approximated by convex decompositions.

Bibliography

Antoine, J.-F. et al. (Mar. 2006). "Approximate Analytical Model for Hertzian Elliptical Contact Problems". In: *Journal of Tribology* 128.3, pp. 660–664. DOI: 10.1115/1.2197850 (cit. on p. 91).

Arianespace (Apr. 2014). *Vega User's Manual, Issue 4, Revision 0*. Arianespace. URL: https://www.arianespace.com/ (cit. on p. 143).

Arnold, M. (2017). "DAE Aspects of Multibody System Dynamics". In: *Surveys in Differential-Algebraic Equations IV*. Cham: Springer International Publishing, pp. 41–106. DOI: 10.1007/978-3-319-46618-7_2 (cit. on pp. 12, 14).

Avril, Q., V. Gouranton, and B. Arnaldi (2009). "New trends in collision detection performance". In: *VRIC*. Vol. 11, p. 53. URL: https://hal.archives-ouvertes.fr/hal-00412870 (cit. on p. 2).

Bardaro, G. et al. (2017). "Using Modelica for advanced Multi-Body modelling in 3D graphical robotic simulators". In: *Proceedings of the 12th International Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ecp17132887 (cit. on p. 99).

Bartsch, H.-J. (2007). *Taschenbuch mathematischer Formeln*. Vol. 21. Carl Hanser Verlag. ISBN: 978-3-446-40895-1 (cit. on p. 157).

Benveniste, A. et al. (2019). "Multi-Mode DAE Models - Challenges, Theory and Implementation". In: *Computing and Software Science: State of the Art and Perspectives*. Springer, pp. 283–310. DOI: 10.1007/978-3-319-91908-9_16 (cit. on pp. 4, 130).

Benveniste, A. et al. (2022). "Algorithms for the Structural Analysis of Multimode Modelica Models". In: *Electronics* 11.17. DOI: 10.3390/electronics11172755 (cit. on p. 4).

Bergen, G. van den (2003). *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann Publishers. ISBN: 1-55860-801-X (cit. on pp. 2, 3, 42–44, 46–48, 52, 53, 57–59, 69, 80).

Bergen, G. van den (online). *SOLID - Software Library for Interference Detection*. URL: https://github.com/dtecta/solid3 (visited on 02/28/2022) (cit. on p. 59).

Bergen, G. van den and D. Gregorius (2010). *Game Physics Pearls*. AK Peters Ltd. ISBN: 978-1-56881-474-2 (cit. on p. 2).

Bezanson, J. et al. (2017). "Julia: A fresh approach to numerical computing". In: *SIAM review* 59.1, pp. 65–98. DOI: 10.1137/141000671 (cit. on pp. 1, 17).

Bourg, D. M. and B. Bywalec (2013). *Physics for Game Developers: Science, math, and Code for Realistic Effects*. O'Reilly Media, Inc. ISBN: 978-1-449-39251-2 (cit. on pp. 2, 87).

Brenan, K. E., S. L. Campbell, and L. R. Petzold (1996). *Numerical Solution of Initial Value Problems in Differential-Algebraic Equations*. Vol. 14. SIAM. ISBN: 0-89871-353-6 (cit. on p. 12).

Brent, R. (1973). *Algorithms for Minimization without derivatives*. Prentice Hall. ISBN: 0-13-022335-2 (cit. on p. 82).

Brondsted, A. (2012). *An introduction to convex polytopes*. Vol. 90. Springer Science & Business Media. ISBN: 978-1-4612-7023-2 (cit. on pp. 42, 157).

Bronstein, I. N. et al. (2001). *Taschenbuch der Mathematik*. Vol. 5. Verlag Harri Deutsch. ISBN: 3-8171-2005-2 (cit. on p. 64).

Bullet (online). *Bullet Physics Engine*. URL: https://bulletphysics.org (visited on 04/26/2022) (cit. on p. 2).

Caillaud, B., M. Malandain, and J. Thibault (2020). "Implicit Structural Analysis of Multimode DAE Systems". In: *23rd International Conference on Hybrid Systems: Computation and Control*. HSCC '20. ACM. DOI: 10.1145/3365365.3382201 (cit. on p. 4).

Campbell, S. L., V. H. Linh, and L. R. Petzold (2008). "Differential-algebraic equations". In: *Scholarpedia* 3.8. revision #153375, p. 2849. DOI: 10.4249/scholarpedia.2849 (cit. on p. 116).

Cellier, F. E. and E. Kofman (2006). *Continuous System Simulation*. Springer Science & Business Media. ISBN: 978-0-387-26102-7 (cit. on pp. 4, 12–14).

Coumans, E. (online). *Bullet Physics SDK*. URL: https://github.com/bulletphysics/bullet3 (visited on 02/28/2022) (cit. on p. 2).

Danisch, S. and J. Krumbiegel (2021). "Makie.jl: Flexible high-performance data visualization for Julia". In: *Journal of Open Source Software* 6.65, p. 3349. DOI: 10.21105/joss.03349 (cit. on p. 17).

Deremetz, M. et al. (2020). "MOSAR-WM: A relocatable robotic arm demonstrator for future on-orbit applications". In: *71st International Astronautical Congress, IAC 2020*. IAF. URL: https://elib.dlr.de/139962/ (cit. on p. 151).

Domingues, L. R. and H. Pedrini (2015). "Bounding volume hierarchy optimization through agglomerative treelet restructuring". In: *Proceedings of the 7th Conference on High-Performance Graphics*, pp. 13–20. DOI: 10.1145/2790060.2790065 (cit. on p. 53).

Elmqvist, H., T. Henningsson, and M. Otter (2017). "Innovations for Future Modelica". In: *Proceedings of the 12th International Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ecp17132693 (cit. on pp. 1, 17).

Elmqvist, H. et al. (2015). "Generic Modelica Framework for MultiBody Contacts and Discrete Element Method". In: *Proceedings of the 11th International Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ecp15118427 (cit. on p. 99).

Elmqvist, H. (2019). *Modia — A Prototyping Platform for Next Generation Modeling And Simulation Based on Julia.* Jubilee Symposium 2019: Future Directions of System Modeling and Simulation. URL: https://modelica.github.io/Symposium2019/slides/jubilee-symposium-2019-slides-elmqvist.pdf (visited on 12/04/2022) (cit. on p. 117).

Elmqvist, H., S. E. Matsson, and M. Otter (2014). "Modelica extensions for multi-mode DAE systems". In: *Proceedings of the 10th International Modelica Conference*. Linköping University Electronic Press, pp. 183–193. DOI: 10.3384/ECP14096183 (cit. on pp. 4, 5).

Elmqvist, H. and M. Otter (online[a]). *Modia Documentation.* URL: https://modiasim.github.io/Modia.jl/stable (visited on 12/11/2022) (cit. on pp. 17, 21, 118).

Elmqvist, H. and M. Otter (online[b]). *Modia.jl.* URL: https://github.com/ModiaSim/Modia.jl (visited on 12/11/2022) (cit. on p. 17).

Elmqvist, H., M. Otter, A. Neumayr, and G. Hippmann (2021). "Modia - Equation Based Modeling and Domain Specific Algorithms". In: *Proceedings of the 14th International Modelica Conference*. LiU Electronic Press, pp. 73–86. DOI: 10.3384/ecp2118173 (cit. on pp. 7, 17, 18, 22, 32, 146).

Epic Games (online). *Unreal Engine 5 Documentation | Unreal Engine Documentation.* URL: https://docs.unrealengine.com (visited on 04/27/2022) (cit. on pp. 2, 22).

Erez, T., Y. Tassa, and E. Todorov (2015). "Simulation Tools for Model-Based Robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX". In: *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 4397–4404. DOI: 10.1109/ICRA.2015.7139807 (cit. on p. 2).

Ericson, C. (2004). *Real-Time Collision Detection.* Morgan Kaufmann Publishers. ISBN: 1-55860-732-3 (cit. on pp. 2, 57).

Fišer, D. (online). *libccd.* URL: https://github.com/danfis/libccd (visited on 02/28/2022) (cit. on p. 59).

Flores, P. et al. (2011). "On the continuous contact force models for soft materials in multibody dynamics". In: *Multibody system dynamics* 25.3, pp. 357–375. DOI: 10.1007/s11044-010-9237-4 (cit. on pp. 89, 95–97).

Floros, X. et al. (2011). "Automated Simulation of Modelica Models with QSS Methods: The Discontinuous Case". In: *Proceedings of the 8th International Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ecp11063657 (cit. on p. 4).

FMI Project (2014). *Functional Mock-up Interface for Model Exchange and Co-Simulation*. 2.0. URL: https://fmi-standard.org (cit. on pp. 115, 119).

FMI Project (2017). *Functional Mock-up Interface for Model Exchange*. 1.0.1. Modelica Association. URL: https://fmi-standard.org (cit. on p. 86).

Gear, C. W. (1988). "Differential-Algebraic Equation Index Transformations". In: *SIAM Journal on Scientific and Statistical Computing* 9.1, pp. 39–47. DOI: 10.1137/0909004 (cit. on p. 14).

Gear, C. W., B. Leimkuhler, and G. K. Gupta (1985). "Automatic integration of Euler-Lagrange equations with constraints". In: *Journal of Computational and Applied Mathematics* 12, pp. 77–90. DOI: 10.1016/0377-0427(85)90008-1 (cit. on p. 14).

Gilbert, E., D. Johnson, and S. Keerthi (1988). "A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space". In: *IEEE Journal on Robotics and Automation* 4.2, pp. 193–203. DOI: 10.1109/56.2083 (cit. on p. 58).

Gottschalk, S., M. C. Lin, and D. Manocha (1996). "OBBTree: A hierarchical structure for rapid interference detection". In: *Proceedings SIGGRAPH'96*, pp. 171–180. DOI: 10.1145/237170.237244 (cit. on pp. 52, 53).

Grünbaum, B. (2013). *Convex polytopes*. Vol. 221. Springer Science & Business Media (cit. on p. 42).

Grünbaum, B. and G. C. Shephard (1969). "Convex polytopes". In: *Bulletin of the London Mathematical Society* 1.3, pp. 257–300 (cit. on p. 157).

Havok (online). *Havok Physics Engine*. URL: https://www.havok.com (visited on 04/26/2022) (cit. on p. 2).

Hertz, H. (1896). "On the contact of solids - on the contact of rigid elastic solids and on hardness". In: *Miscellaneous papers*, pp. 146–183. URL: https://archive.org/details/cu31924012500306 (cit. on pp. 88, 91).

Hindmarsh, A., R. Serban, and A. Collier (2015). *User Documentation for IDA v2.8.2*. Tech. rep. UCRL-SM-208112. Lawrence Livermore National Laboratory (cit. on pp. 12, 21).

Hindmarsh, A. et al. (2005). "SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers". In: *ACM Transactions on Mathematical Software* 31.3, pp. 363–396. DOI: 10.1145/1089014.1089020 (cit. on pp. 12, 21).

Hippmann, G. (2004a). "An algorithm for compliant contact between complexly shaped bodies". In: *Multibody System Dynamics* 12.4, pp. 345–362 (cit. on p. 57).

Hippmann, G. (2004b). "Modellierung von Kontakten komplex geformter Körper in der Mehrkörperdynamik". PhD thesis. Technische Universität Wien (cit. on p. 57).

Hofmann, A. et al. (2014). "Simulating Collisions within the Modelica MultiBody Library". In: *Proceedings of the 10th International Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ECP14096949 (cit. on pp. 3, 16, 87).

Höger, C. (2014). "Dynamic Structural Analysis for DAEs". In: *2014 Summer Simulation Multiconference*. SummerSim '14. Society for Computer Simulation International. URL: https://dl.acm.org/doi/10.5555/2685617.2685629 (cit. on p. 4).

Hopcroft, J. and R. Tarjan (1974). "Efficient Planarity Testing". In: *Journal of the ACM* 21.4, pp. 549–568. DOI: 10.1145/321850.321852 (cit. on p. 29).

Hubbard, P. M. (1996). "Approximating polyhedra with spheres for time-critical collision detection". In: *ACM Transactions on Graphics* 15.3, pp. 179–210. DOI: 10.1145/231731.231732 (cit. on p. 53).

Hunter, J. D. (2007). "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3, pp. 90–95. DOI: 10.1109/MCSE.2007.55 (cit. on p. 17).

Julia Documentation (online). *Julia Documentation · The Julia Language*. URL: https://docs.julialang.org (visited on 06/19/2023) (cit. on p. 18).

JuliaArrays (online). *StaticArrays.jl*. URL: https://github.com/JuliaArrays/StaticArrays.jl (visited on 12/04/2023) (cit. on p. 113).

JuliaLinearAlgebra (online). *RecursiveFactorization.jl*. URL: https://github.com/JuliaLinearAlgebra/RecursiveFactorization.jl (visited on 10/28/2024) (cit. on p. 34).

JuliaMath (online). *DoubleFloats.jl*. URL: https://github.com/JuliaMath/DoubleFloats.jl (visited on 04/06/2022) (cit. on pp. 72, 75).

JuliaPy (online). *PyPlot.jl*. URL: https://github.com/JuliaPy/PyPlot.jl (visited on 08/03/2023) (cit. on p. 17).

Karras, T. (2012). "Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees". In: *Proceedings of the 4th Symposium on High-Performance Graphics*, pp. 33–37. DOI: 10.2312/EGGH/HPG12/033-037 (cit. on p. 53).

Keller, A. (online). *Unitful.jl*. URL: https://github.com/PainterQubits/Unitful.jl (visited on 12/12/2022) (cit. on p. 19).

Kendall, D. C. (1984). "Statistics, Geometry and the Cosmos". In: *Quarterly Journal of the Royal Astronomical Society* 25, p. 147 (cit. on p. 42).

Kenwright, B. (2015). *Generic Convex Collision Detection using Support Mapping*. Tech. rep. (cit. on pp. 3, 46, 48, 59, 60, 63, 70, 74, 75).

Kümper, S., M. Hellerer, and T. Bellmann (2021). "DLR Visualization 2 Library - Real-Time Graphical Environments for Virtual Commissioning". In: *Proceedings of 14th Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ecp21181197 (cit. on p. 31).

Letier, P. et al. (2019). "MOSAR: Modular spacecraft assembly and reconfiguration demonstrator". In: *15th Symposium on Advanced Space Technologies in Robotics and Automation* (cit. on pp. 1, 151).

Machado, M. et al. (2012). "Compliant contact force models in multibody dynamics: Evolution of the Hertz contact theory". In: *Mechanism and Machine Theory* 53, pp. 99–121. ISSN: 0094-114X. DOI: 10.1016/j.mechmachtheory.2012.02.010 (cit. on p. 95).

Mainzer, D. (2015). "New Geometric Algorithms and Data Structures for Collision Detection of Dynamically Deforming Objects". PhD thesis. Clausthal University of Technology (cit. on p. 2).

Mamou, K. (online). *V-HACD*. URL: https://github.com/kmammou/v-hacd (visited on 02/21/2022) (cit. on p. 50).

Mamou, K., E. Lengyel, and A. Peters (2016). "Volumetric hierarchical approximate convex decomposition". In: *Game Engine Gems 3*. AK Peters, pp. 141–158 (cit. on p. 50).

Mathavan, S., M. R. Jackson, and R. M. Parkin (2010). "A theoretical analysis of billiard ball dynamics under cushion impacts". In: *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science* 224.9, pp. 1863–1873. DOI: 10.1243/09544062JMES1964 (cit. on pp. 100, 101).

Mattsson, S. E., M. Otter, and H. Elmqvist (2015). "Multi-mode DAE systems with varying index". In: *Proceedings of the 11th International Modelica Conference*, pp. 89–98. DOI: 10.3384/ecp1511889 (cit. on p. 5).

Mattsson, S. E. and G. Söderlind (1993). "Index reduction in differential-algebraic equations using dummy derivatives". In: *SIAM Journal on Scientific Computing* 14.3, pp. 677–692 (cit. on p. 14).

Mehlhase, A. (2014). "A Python framework to create and simulate models with variable structure in common simulation environments". In: *Mathematical and Computer Modelling of Dynamical Systems* 20.6, pp. 566–583. DOI: 10.1080/13873954.2013.861854 (cit. on p. 5).

Millington, I. (2010). *Game Physics Engine Development: How to build a robust commercial-grade physics engine for your game*. CRC Press. ISBN: 978-0-12-381976-5 (cit. on p. 2).

Mirtich, B. V. (1996). "Impulse-based dynamic simulation of rigid body systems". PhD thesis. University of California, Berkeley (cit. on p. 87).

Modelica Association (2021). *Modelica® – A Unified Object-Oriented Language for Systems Modeling. Language Specification, Version 3.5*. URL: https://specification.modelica.org/maint/3.5/MLS.pdf (cit. on pp. 1, 11, 115).

Modelica Association (online). *Modelica tools*. URL: https://modelica.org/tools (visited on 12/11/2022) (cit. on p. 11).

Neumayr, A. (online). *Modia3D Installation Guide*. URL: https://github.com/ModiaSim/Modia3D.jl/wiki/Full-Installation-Guide-for-Julia,-Modia3D,-Modia (visited on 03/10/2023) (cit. on p. 22).

Neumayr, A. and M. Otter (2017). "Collision Handling with Variable-step Integrators". In: *Proceedings of the 8th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. EOOLT'17. ACM, pp. 9–18. DOI: 10.1145/3158191.3158193 (cit. on pp. 3, 6, 58, 60, 74, 76–78, 80–83).

Neumayr, A. and M. Otter (2018). "Component-Based 3D Modeling of Dynamic Systems". In: *Proceedings of the American Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ECP18154175 (cit. on pp. 3, 6, 7, 22, 51–54).

Neumayr, A. and M. Otter (2019a). "Algorithms for Component-Based 3D Modeling". In: *Proceedings of the 13th International Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ecp19157383 (cit. on pp. 7, 14, 29, 51).

Neumayr, A. and M. Otter (2019b). "Collision Handling with Elastic Response Calculation and Zero-Crossing Functions". In: *Proceedings of the 9th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. EOOLT'19. ACM, pp. 57–65. DOI: 10.1145/3365984.3365986 (cit. on pp. 3, 6, 53, 77, 81–83, 86, 89, 96).

Neumayr, A. and M. Otter (2020). "Modia3D: Modeling and Simulation of 3D-Systems in Julia". In: *Proceedings of the JuliaCon Conferences* 1.1. DOI: 10.21105/jcon.00043 (cit. on pp. 6, 100).

Neumayr, A. and M. Otter (2023a). "Modelling and Simulation of Physical Systems with Dynamically Changing Degrees of Freedom". In: *Electronics* 12.3. DOI: 10.3390/electronics12030500 (cit. on pp. 6, 18, 111, 115, 125, 138, 143, 146, 147, 149, 150).

Neumayr, A. and M. Otter (2023b). "Variable Structure System Simulation via Predefined Acausal Components". In: *Proceedings of the 15th International Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ecp204 (cit. on pp. 7, 115, 125, 141, 146).

Neumayr, A. and M. Otter (2024). "Combining Equation-based and Multibody Models". In: *Proceedings of the Asian Modelica Conference 2024*. LiU Electronic Press. DOI: 10.3384/ecp217 (cit. on pp. 7, 14, 22, 32).

Neumayr, A., M. Otter, and G. Hippmann (online[a]). *Modia3D Documentation*. URL: https://modiasim.github.io/Modia3D.jl/stable (visited on 11/17/2021) (cit. on pp. 22, 24, 31).

Neumayr, A., M. Otter, and G. Hippmann (online[b]). *Modia3D.jl*. URL: https://github.com/ModiaSim/Modia3D.jl (visited on 11/17/2021) (cit. on pp. 17, 22).

Nystrom, R. (2014). *Game Programming Patterns*. Genever Benning. ISBN: 978-0-9905829-0-8. URL: http://gameprogrammingpatterns.com/ (cit. on p. 22).

Nytsch-Geusen, C. et al. (2006). "Advanced modeling and simulation techniques in MOSILAB: A system development case study". In: *Proceedings of the 5th International Modelica Conference* (cit. on p. 4).

O'Rourke, J. (1998). *Computational Geometry in C*. Cambridge University Press. ISBN: 0-521-64010-5 (cit. on p. 57).

Olsson, H. et al. (2008). "Balanced Models in Modelica 3.0 for Increased Model Quality". In: *Proceedings of the 6th International Modelica Conference*. The Modelica Association, University of Applied Sciences Bielefeld, pp. 21–33. URL: https://www.modelica.org/events/modelica2008/Proceedings/sessions/session1a3.pdf (cit. on p. 117).

Olvång, L. (2010). *Real-time Collision Detection with Implicit Objects*. Tech. rep. Department of Information Technology, Uppsala University,Sweden (cit. on pp. 3, 47, 59–61, 74, 75).

Ong, C. J. (1995). "Properties of penetration between general objects". In: *Proceedings of 1995 IEEE International Conference on Robotics and Automation*. Vol. 3. IEEE, pp. 2293–2298. DOI: 10.1109/ROBOT.1995.525603 (cit. on p. 45).

Ong, C. J. and E. G. Gilbert (1996). "Growth Distances: New Measures for Object Separation and Penetration". In: *IEEE Transactions on Robotics and Automation* 12.6, pp. 888–903. DOI: 10.1109/70.544772 (cit. on pp. 44, 45).

OpenBLAS (online). *OpenBLAS*. URL: http://www.openmathlib.org/OpenBLAS (visited on 12/04/2024) (cit. on p. 34).

Otter, M. (2022). "Signal Tables: An Extensible Exchange Format for Simulation Data". In: *MDPI Electronics* 11.2811, pp. 1–19. DOI: 10.3390/electronics11182811 (cit. on p. 131).

Otter, M. (online). *SignalTables.jl*. URL: https://github.com/ModiaSim/SignalTables.jl (visited on 12/12/2022) (cit. on p. 131).

Otter, M. and H. Elmqvist (2017). "Transformation of Differential Algebraic Array Equations to Index One Form". In: *Proceedings of the 12th International Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ecp17132565 (cit. on pp. 14, 21, 32).

Otter, M., H. Elmqvist, and J. D. López (2005). "Collision Handling for the Modelica MultiBody Library". In: *Proceedings of the 4th International Modelica Conference* (cit. on pp. 3, 87, 90, 99).

Pantelides, C. C. (1988). "The Consistent Initialization of Differential-Algebraic Systems". In: *SIAM Journal on Scientific and Statistical Computing* 9.2, pp. 213–231. DOI: 10.1137/0909014 (cit. on pp. 4, 13, 14).

Pepper, P. et al. (2011). "A Compositional Semantics for Modelica-style Variable-structure Modeling." In: *Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. EOOLT'11, pp. 45–54. ISBN: 978-91-7519-825-5 (cit. on p. 5).

PhysX (online). *PhysX Physics Engine*. URL: https://nvidia.com (visited on 04/26/2022) (cit. on p. 2).

Pryce, J. D. (2001). "A simple structural analysis method for DAEs". In: *BIT Numerical Mathematics* 41.2, pp. 364–394. DOI: 10.1023/A:1021998624799 (cit. on p. 4).

Pungotra, H. (2010). "Collision Detection and Merging of Deformable B-Spline Surfaces in Virtual Reality Environment". PhD thesis. The University of Western Ontario. URL: https://ir.lib.uwo.ca/etd/32 (cit. on p. 49).

Rackauckas, C. and Q. Nie (2017). "DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia". In: *Journal of Open Research Software* 5.1. DOI: 10.5334/jors.151 (cit. on pp. 21, 34).

Reiner, M. J. (2022). "Simulation of the on-orbit construction of structural variable modular spacecraft by robots". In: *Proceedings of the American Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ECP2118638 (cit. on p. 151).

SciML (online). *DifferentialEquations.jl*. URL: https://github.com/SciML/DifferentialEquations.jl (visited on 12/12/2022) (cit. on pp. 21, 34).

Skrinjar, L., J. Slavič, and M. Boltežar (2018). "A review of continuous contact-force models in multibody dynamics". In: *International Journal of Mechanical Sciences* 145, pp. 171–187. DOI: 10.1016/j.ijmecsci.2018.07.010 (cit. on pp. 89, 95).

Snethen, G. (2008). "Xenocollide: Complex collision made simple". In: *Game Programming Gems 7*. Course Technology. Charles River Media, pp. 165–178. ISBN: 978-1-58450-527-3 (cit. on pp. 3, 46, 47, 57, 59, 60, 62, 63, 70, 71, 74, 75).

Snethen, G. (online[a]). *XenoCollide GitHub*. URL: https://github.com/erwincoumans/xenocollide (visited on 03/15/2022) (cit. on p. 59).

Snethen, G. (online[b]). *XenoCollide Website*. URL: http://xenocollide.snethen.com (visited on 01/08/2021) (cit. on pp. 61, 75).

Steinbach, O. (2007). *Numerical approximation methods for elliptic boundary value problems: finite and boundary elements*. Springer: New York, NY, USA. DOI: 10.1007/978-0-387-68805-3 (cit. on p. 116).

Szauer, G. (2017). *Game Physics Cookbook*. Packt Publishing Ltd. ISBN: 978-1-78712-366-3 (cit. on p. 2).

Tarjan, R. (1972). "Depth-first search and linear graph algorithms". In: *SIAM journal on computing* 1.2, pp. 146–160. DOI: 10.1137/0201010 (cit. on p. 29).

Three.js (online). *Three.js docs*. URL: https://threejs.org (visited on 02/03/2023) (cit. on pp. 22, 32).

Tiller, M. (online). *Modelica by Example*. URL: https://mbe.modelica.university/behavior/arrays/oned/ (visited on 01/15/2025) (cit. on p. 125).

Tinnerholm, J., A. Pop, and M. Sjölund (2022). "A Modular, Extensible, and Modelica-Standard-Compliant OpenModelica Compiler Framework in Julia Supporting Structural Variability". In: *Electronics* 11.11, p. 1772. ISSN: 2079-9292. DOI: 10.3390/electronics11111772 (cit. on p. 5).

Toledo, S. (1997). "Locality of Reference in LU Decomposition with Partial Pivoting". In: *SIAM Journal on Matrix Analysis and Applications* 18.4, pp. 1065–1081. DOI: 10.1137/S0895479896297744 (cit. on p. 34).

Turbosquid.com (online). URL: http://www.turbosquid.com/3d-models/vega-rocket-3d-max/655678 (visited on 01/12/2015) (cit. on p. 143).

Unity Technologies (online). *Unity - Manual: Unity User Manual 2021.3 (LTS)*. URL: https://docs.unity3d.com/Manual/index.html (visited on 04/27/2022) (cit. on pp. 2, 22).

Vinkler, M., J. Bittner, and V. Havran (2017). "Extended Morton Codes for High Performance Bounding Volume Hierarchy Construction". In: *Proceedings of HPG'17*. ACM. DOI: 10.1145/3105762.3105782 (cit. on p. 53).

youbot-store (online). *YouBot 3D Model - youBot wiki*. URL: http://www.youbot-store.com/wiki/index.php/YouBot_3D_Model (visited on 11/20/2023) (cit. on p. 146).

Zachmann, G. (1998). "Rapid collision detection by dynamically aligned DOP-trees". In: *Proceedings. IEEE 1998 Virtual Reality Annual International Symposium*, pp. 90–97. DOI: 10.1109/VRAIS.1998.658428 (cit. on p. 53).

Zhang, X., Y. J. Kim, and D. Manocha (2014). "Continuous Penetration Depth". In: *Computer-Aided Design* 46, pp. 3–13. DOI: 10.1016/j.cad.2013.08.013 (cit. on pp. 45, 80).

Zimmer, D. (2010). "Equation-based modeling of variable-structure systems". PhD thesis. ETH Zurich. DOI: 10.3929/ethz-a-006053740 (cit. on pp. 4, 11).

List of Publications

Journal Papers

Neumayr, A. and M. Otter (2023a). "Modelling and Simulation of Physical Systems with Dynamically Changing Degrees of Freedom". In: *Electronics* 12.3. DOI: 10.3390/electronics12030500.

Peer-reviewed Conference Papers

Elmqvist, H., M. Otter, A. Neumayr, and G. Hippmann (2021). "Modia - Equation Based Modeling and Domain Specific Algorithms". In: *Proceedings of the 14th International Modelica Conference*. LiU Electronic Press, pp. 73–86. DOI: 10.3384/ecp2118173.

Neumayr, A. and M. Otter (2017). "Collision Handling with Variable-step Integrators". In: *Proceedings of the 8th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. EOOLT'17. ACM, pp. 9–18. DOI: 10.1145/3158191.3158193.

Neumayr, A. and M. Otter (2018). "Component-Based 3D Modeling of Dynamic Systems". In: *Proceedings of the American Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ECP18154175.

Neumayr, A. and M. Otter (2019a). "Algorithms for Component-Based 3D Modeling". In: *Proceedings of the 13th International Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ecp19157383.

Neumayr, A. and M. Otter (2019b). "Collision Handling with Elastic Response Calculation and Zero-Crossing Functions". In: *Proceedings of the 9th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. EOOLT'19. ACM, pp. 57–65. DOI: 10.1145/3365984.3365986.

Neumayr, A. and M. Otter (2020). "Modia3D: Modeling and Simulation of 3D-Systems in Julia". In: vol. 1. 1. The Open Journal. DOI: 10.21105/jcon.00043.

Neumayr, A. and M. Otter (2023b). "Variable Structure System Simulation via Predefined Acausal Components". In: *Proceedings of the 15th International Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ecp204.

Neumayr, A. and M. Otter (2024). "Combining Equation-based and Multibody Models". In: *Proceedings of the Asian Modelica Conference 2024*. LiU Electronic Press. DOI: 10.3384/ecp217.

Source Code and Documentations

Neumayr, A. (online). *Modia3D Installation Guide*. URL: https://github.com/ModiaSim/Modia3D.jl/wiki/Full-Installation-Guide-for-Julia,-Modia3D,-Modia (visited on 03/10/2023).

Neumayr, A., M. Otter, and G. Hippmann (online[a]). *Modia3D Documentation*. URL: https://modiasim.github.io/Modia3D.jl/stable (visited on 11/17/2021).

Neumayr, A., M. Otter, and G. Hippmann (online[b]). *Modia3D.jl*. URL: https://github.com/ModiaSim/Modia3D.jl (visited on 11/17/2021).