UniVerMeC – A Framework for Development, Assessment and Interoperable Use of Verified Techniques

Von der Fakultät für Ingenieurwissenschaften,
Abteilung Informatik und Angewandte Kognitionswissenschaft
der Universität Duisburg-Essen

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften

genehmigte Dissertation

von

Stefan Kiel
aus
Frankenberg (Eder)

1. Gutachter: Prof. Dr. Wolfram Luther
2. Gutachter: Prof. Dr. Jürgen Wolff von Gudenberg
Tag der mündlichen Prüfung: 27.01.2014

UNIVERMEC – A FRAMEWORK FOR DEVELOPMENT, ASSESSMENT AND INTEROPERABLE USE OF VERIFIED TECHNIQUES

STEFAN KIEL

With Applications in Distance Computation, Global Optimization, and Comparison Systematics

Stefan Kiel: *UniVerMeC – A Framework for Development, Assessment and Interoperable Use of Verified Techniques,* With Applications in Distance Computation, Global Optimization, and Comparison Systematics

ABSTRACT

Verified algorithms play an important role in the context of different applications from various areas of science. An open question is the interoperability between different verified range arithmetics and their numerical comparability. In this thesis, a theoretical framework is provided for interoperable handling of the arithmetics without altering their verification features. For this purpose, we formalize the arithmetics using a heterogeneous algebra. Based on this algebra, we introduce the concept of function representation objects. They characterize a mathematical function by inclusion functions in different arithmetics and allow for representing particular features of the function (e.g., differentiability). The representation objects allow us to describe problems by functional and relational dependencies so that different verified methods can be used interchangeably. On this basis, we develop a new verified distance computation algorithm which can handle non-convex objects. Furthermore, a verified global optimization algorithm is adapted so that it can use the different methods made accessible by the function representation objects. Moreover, we formalize and improve interval-based hierarchical structures which are used by both algorithms. To evaluate our approach, we provide a prototypical implementation and perform fair numerical comparisons between the different arithmetics. Finally, we apply the framework to relevant application cases from biomechanics and modeling, simulation and control of fuel cells. We demonstrate that our implementation supports parallel computations on the CPU and GPU and show that it can be extended by interfacing additional external IVP solver libraries.

ZUSAMMENFASSUNG

Verifizierte numerische Verfahren spielen in zahlreichen Anwendungskontexten eine wichtige Rolle. Hierbei ist die Interoperabilität zwischen den verschiedenen verwendeten Wertebereichsarithmetiken sowie ihre numerische Vergleichbarkeit eine offene Frage. In dieser Arbeit wird ein theoretisches Rahmenwerk zur Verfügung gestellt, welches die Interoperabilität zwischen den Arithmetiken unter Beibehaltung der Verifikationseigenschaften sicherstellt. Hierzu werden die Arithmetiken mittels einer heterogenen Algebra formalisiert und darauf aufbauend das Konzept der "Funktionsrepräsentationsobjekte" eingeführt. Damit können mathematische Funktionen mittels Inklusionsfunktionen in unterschiedlichen Arithmetiken charakterisiert und ihre Eigenschaften (z.B. Differenzierbarkeit) dargestellt werden. Mit Hilfe der Repräsentationsobjekte ist es möglich Problemstellungen, die mittels funktionaler und relationaler Beziehungen repräsentierbar sind, derart zu beschreiben, dass unterschiedliche verifizierte Methoden eingesetzt werden können. Aufbauend hierauf wird ein neuer Algorithmus zur verifizierten Abstandsberechnung zwischen nicht konvexen Körpern entwickelt. Weiterhin wird ein verifizierter globaler Optimierungsalgorithmus derart angepasst, dass er die durch Funktionsrepräsentationsobjekte zugänglich gemachten, unterschiedlichen Methoden flexibel nutzen kann. Beide Algorithmen greifen auf in das Rahmenwerk integrierte Hilfsdatenstrukturen zur intervallbasierten hierarchischen Zerlegung zurück, die im Kontext dieser Arbeit formalisiert und verbessert werden. Zur Evaluation wird der Gesamtansatz prototypisch implementiert. Hierbei werden nicht nur numerische Vergleiche zwischen den unterschiedlichen Arithmetiken durchgeführt, sondern auch relevante Anwendungsfälle aus der Biomechanik und der Simulation von Brennstoffzellen behandelt. Weiterhin wird demonstriert, dass die vorliegende Implementierung parallele Berechnungen auf der CPU und GPU unterstützt und durch die Anbindung weiterer externer AWP Löserbibliotheken flexibel erweiterbar ist.

PUBLICATIONS

Most ideas, figures and algorithms appeared previously in the following publications:

- R. Cuypers, S. Kiel, and W. Luther. "Automatic Femur Decomposition, Reconstruction, and Refinement Using Superquadric Shapes." In: *Proceedings of the IASTED International Conference*. Vol. 663. 2009, p. 59
- E. Auer, R. Cuypers, E. Dyllong, S. Kiel, and W. Luther. "Verification and Validation for Femur Prosthesis Surgery." In: *Computer-assisted proofs - tools, methods and applications*. Ed. by B. M. Brown, E. Kaltofen, S. Oishi, and S. M. Rump. Dagstuhl Seminar Proceedings 09471. Schloss Dagstuhl, 2010. URL: http://drops.dagstuhl.de/opus/volltexte/2010/2513
- E. Dyllong and S. Kiel. "Verified Distance Computation Between Convex Hulls of Octrees Using Interval Optimization Techniques." In: *PAMM* 10.1 (2010), pp. 651–652. ISSN: 1617-7061
- E. Auer, A. Chuev, R. Cuypers, S. Kiel, and W. Luther. "Relevance of Accurate and Verified Numerical Algorithms for Verification and Validation in Biomechanics." In: *EUROMECH Colloquium 511*. Ponta Delgada, Azores, Portugal, 2011
- S. Kiel. "Verified Spatial Subdivision of Implicit Objects Using Implicit Linear Interval Estimations." In: *Curves and Surfaces*. Ed. by J.-D. Boissonnat, P. Chenin, A. Cohen, C. Gout, T. Lyche, M.-L. Mazure, and L. Schumaker. Vol. 6920. Lecture Notes in Computer Science. Springer, 2012, pp. 402–415
- E. Dyllong and S. Kiel. "A Comparison of verified distance computation between implicit objects using different arithmetics for range enclosure." In: *Computing* 94 (2 2012), pp. 281–296. ISSN: 0010-485X
- S. Kiel. "YalAA: Yet Another Library for Affine Arithmetic." In: *Reliable Computing* 16 (2012), pp. 114–129
- E. Auer, S. Kiel, and A. Rauh. "Verified Parameter Identification for Solid Oxide Fuel Cells." In: *Proceedings of the 5th International Conference on Reliable Engineering Computing*. 2012
- S. Kiel, W. Luther, and E. Dyllong. "Verified distance computation between non-convex superquadrics using hierarchical

space decomposition structures." In: *Soft Computing* 17.8 (2013), pp. 1367–1378. ISSN: 1432-7643

- S. Kiel, E. Auer, and A. Rauh. "Use of GPU Powered Interval Optimization for Parameter Identification in the Context of SO Fuel Cells." In: *Proceedings of NOLCOS 2013 - 9th IFAC Symposium on Nonlinear Control Systems*. 2013. DOI: 10.3182/20130904-3-FR-2041.00169

- S. Kiel, E. Auer, and A. Rauh. "An Environment for Testing, Verification and Validation of Dynamical Models in the Context of Solid Oxide Fuel Cells." In: *Reliable Computing* 19.3 (2014), pp. 302–317

Table 1: List of major text and figure adaptions from previous publications.

SECTION / FIGURE	PUBLICATIONS
Sect. 3.3.2, 3.3.3.2, 3.3.5	[Kie12b]
Sect. 7.1.1-7.1.4	[DK12; KLD13]
Sect. 7.3.1	[KAR14]
Sect. 8.1.1	[DK12]
Sect. 8.1.2	[KLD13]
Sect. 8.2	[AKR12; KAR13; KAR14]
Fig. 2, 35, 44	[KAR13]
Fig. 10	[Kie12a]
Fig. 12, 13	[Kie12b]
Fig. 14, 27, 32, 42	[KLD13]
Fig. 25, 46	[KAR14]
Fig. 31	[DK10]
Fig. 33, 40	[DK12]

Sections containing major portions of updated and adapted text published previously are listed in Tab. 1. The table also identifies the sources of previously published figures.

DANKSAGUNG

Die Erstellung dieser Arbeit erfolgte zu wesentlichen Teilen im Rahmen des durch die Deutsche Forschungsgemeinschaft geförderten Projekts “Intervallbasierte Verfahren für adaptive hierarchische Modelle in Modellierungs- und Simulationssystemen”. An erster Stelle danke ich Frau Dr. Eva Dyllong und Herrn Prof. Dr. Wolfram Luther, die als Projektleiterin beziehungsweise Lehrstuhlinhaber die Durchführung des Forschungsvorhabens ermöglichten und unterstützten. Herzlichen Dank auch an Herrn Prof. Dr. Jürgen Wolff von Gudenberg für die Übernahme des Korreferats. Mein besonderer Dank an Frau Dr. Ekaterina Auer, die mich nicht nur als Kollegin stets unterstützte, sondern auch die Anwendung der entwickelten Methoden im Kontext von Festoxidbrennstoffzellen anregte und im Rahmen einer Kooperation ermöglichte. In diesem Zusammenhang danke ich auch Herrn Dr. Andreas Rauh für die zur Verfügung gestellten Festoxidbrennstoffzellenmodelle. Mein herzlicher Dank geht an die Mitarbeiter des Lehrstuhls für Computergrafik und Wissenschaftliches Rechnen für die stets kollegiale Atmosphäre und das freundliche Arbeitsumfeld. Und nicht zuletzt ein großes Dankeschön an meine Eltern und Familie, die mich immer in jeglicher Hinsicht unterstützten.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ALGORITHMS

ACRONYMS

AA affine arithmetic

AD algorithmic differentiation

API application programming interface

CPU central processing unit

CSG computer solid geometry

DAG directed acyclic graph

ED elemental differentiability

FIFO first in, first out

FP floating-point

FPA floating-point arithmetic

FRO function representation object

GPU graphics processing unit

GPGPU general purpose computations on the GPU

GUI graphical user interface

IA interval arithmetic

ILIE implicit linear interval estimation

IVP initial value problem

KKT Karush-Kuhn-Tucker

LDB linear dominated bounder

LIFO last in, first out

MPI message-passing interface

ODE ordinary differential equation

PDE partial differential equation

QFB quadratic fast bounder

SFINAE substitution failure is not an error

SIMD single instruction multiple data

SIMT single instruction multiple threads

SISD single instruction single data

SOFC solid oxide fuel cell

SQ superquadric

THR total hip replacement

TM Taylor model

NOTATION

Generally, the notation used in this these follows the standardized interval notation introduced in [Kea+02]. We denote scalars and vectors by lower case letters (i.e., x, y) and matrices by upper case ones (i.e., M, T). Interval quantities are bold-faced (i.e., $\mathbf{x}, \mathbf{y}$); affine forms

are marked by a hat (i.e., $(\hat{x}, \hat{y})$); and Taylor models by a breve (i.e., $\breve{x}, \breve{y}$). Additionally, we use calligraphic letters (i.e., $\mathcal{M}, \mathcal{S}$) for sets and fractured letter (i.e., $\mathfrak{T}, \mathfrak{N}$) for tuples. Individual entries of a tuple $\mathfrak{T} = (\mathsf{E}_1, \ldots, \mathsf{E}_n)$ can be accessed by the shorthand notation $\mathfrak{T}.\mathsf{E}_i$ (for the i-th element).

1 INTRODUCTION

Numerical computations on a computer

Today, numerical computations are normally carried out on a digital computer using floating-point arithmetic (FPA). Despite the fact that the results are not exact and that this finite number system is not flawless, a direct floating-point (FP) implementation of an algorithm based on the real-number system can yield satisfactory results. However, such a behavior is by no means guaranteed and the produced results might turn out to be fatally wrong. Thus, it is usually necessary to adapt the algorithms carefully to the FP number system. Even so, the result is only an approximation of the true one.

Verified methods

The need for guaranteed results arose in many application areas very early on. This led, for example, to the use of interval arithmetic (IA) (developed by Moore [Moo66] and others) as a calculus for automatically bounding the approximation or rounding errors. The approach produces bounds that are guaranteed to enclose the real number result. A method that handles rounding errors, does not introduce approximation errors, and does not neglect solutions is called *verified* because it guarantees the correctness of results obtained on a computer.

Error sources

Aside from the above-mentioned numerical errors, we have to cope with additional error sources in practical applications. The two most important ones are modeling errors and uncertain (input) parameters. Modeling errors arise because the formal model on which our computations are based usually does not correspond to the real world exactly. Additionally, model parameters are not known exactly in general. On the one hand the values of these parameters might be known only up to a certain degree of accuracy, for example, if they come from measurements; on the other hand, the parameters might be given in form of stochastic distributions (which in turn, can have uncertain characteristics). Handling and propagating both uncertainty and numerical errors through complex computations can become quite complicated if only FPA is used. IA or range arithmetics in general can handle both directly.

Sophisticated verified arithmetics

Even though IA is employed in many different application areas nowadays, its naive application often produces correct but, due to wide bounds, useless results from the practical point of view. Over the time, several more sophisticated verified arithmetics and techniques were proposed to supplement or replace IA, for example, centered forms [Neu90], affine arithmetic (AA) [CS90], ellipsoid arithmetic [Neu93], generalized IA [Han75], Taylor models (TMs) [Ber95b] or consistency techniques which try to exploit domain knowledge to

tighten bounds. Often, improved tightness of enclosures is achieved at the cost of a higher computational effort per operation.

1.1 PROBLEM MOTIVATION

Branch and bound algorithms

Branch and bound algorithms are a common problem solving strategy in the scope of interval computations, for example, for distance computation [DG07c], global optimization [HW04] or path planning [Jau01]. Their basic principle is the subdivision of the search region into box-shaped subregions. Besides this basic principle, additional techniques are applied, such as heuristics for box selection or subdivision directions, more sophisticated enclosure techniques, or interval contractors. Without these accelerating devices, the performance of branch and bound algorithms might be poor due to their exponential complexity.

Selection of accelerators

For good results it is necessary to choose the kind of an accelerator depending on the problem at hand. However, even experienced users might have difficulties selecting them in a satisfactory way. That is, they have problems finding the method delivering bounds in a reasonable computational time and at acceptable memory cost that are also sharp enough to be useful. From a theoretical point of view, this question was investigated only for few range bounding techniques over sufficiently small intervals[1]. In practice, where intervals cannot become arbitrarily small during subdivision due to memory and runtime constraints, such theoretical considerations are merely guidelines despite having applications in special cases.

Interoperability problems

Matters are further complicated by the fact that the actual speed of a numerical method on a computer is determined not only by its theoretical properties but also by its actual implementation. While a lot of implementations are freely available for verified techniques, they are usually not interoperable with each other. Thus, employing techniques provided by different libraries requires substantial effort on the users' part. They do not only need to cope with different interfaces, but also to develop a problem definition and a solving strategy abstract enough to be used with these.

Consequences of interoperability problems

Due to these difficulties in employing and combining different techniques, a single one is usually used throughout the entire application. This voluntary restriction might lead to inferior results. Worse, the problems with interoperability of existing software packages might prevent a practice-oriented user, who is mainly interested in the results, from employing verified computations entirely even if they would suit his needs. Additionally, there is a lack of practical experience as to which method performs well for a specific problem be-

1 Neumaier [Neu03] gives an overview of the different definitions for the approximation orders of range enclosure techniques. He also notes that obtaining analogous results for wide intervals is difficult.

cause there are next to no comparisons between the techniques. The task becomes more complicated if modern hardware architectures are taken into account: Depending on the actual problem structure, it can be favorable to tackle the problem or its parts with graphics processing units (GPUs) instead of employing the standard central processing unit (CPU). These highly parallel many-core processors can often improve the problem solving time. However, they require a special treatment. Implementations well suited for the CPU may not perform well on a GPU. Usually, problems are not solved entirely on the GPU. In case of a hybrid CPU/GPU approach additional interoperability efforts are necessary to cope with the different hardware architectures. For example, libraries employed for IA on the CPU are not necessarily interoperable with GPU interval libraries. To summarize, there is no framework that would allow for selection of adequate verified algorithms and for handling interoperability problems between existing software on the CPU, and even less those arising in a hybrid CPU/GPU computation environment.

1.2 OBJECTIVES

Theoretical foundation

The first main objective of this thesis is to provide the theoretical foundations for a software framework that tackles the interoperability problems. This is done by carefully analyzing common techniques employed in existing interval algorithms and by deriving appropriate interfaces that allow for using these techniques as interchangeable *and* interoperable software components in different algorithms. The framework should take care of the interoperability between different verified techniques automatically and employ well-tested software packages as its basis. Thus, users would be able to specify a problem once and then employ various techniques interchangeably to solve the problem. Additionally, they should get a fair comparison of the performance in the respective problem domains. Furthermore, it should be possible to combine the available techniques to improve solutions for a problem. Here, we also aim to provide a theoretical foundation for conditions under which techniques are allowed to interact with each other (i.e., in such a way as still to produce verified results). While the use of the GPU is not in the main focus of this thesis, we will also discuss some strategies to enable the use of verified interval techniques in a mixed CPU/GPU environment.

Extensible software framework

The theoretical foundations are then practically applied to develop the *proof of concept* software framework UniVerMeC that tackles the interoperability problem as the second objective. Since it is only a proof of concept, we limit the implementation to a set of selected common techniques. However, the framework should be extensible which necessitates application of modern software engineering techniques. That is, we implement an open platform to which new tech-

niques can be added. As more and more users provide reference implementations of their new techniques (algorithms, contractors, range arithmetics, etc.) inside the framework, the task of fair comparisons between new developments becomes less difficult. Currently, authors of a new development are often forced to make their own implementation of the techniques they want to compare with, because, for example, a reference implementation is not available publicly or cannot be employed easily in the software system of the authors. This results in comparisons with easy-to-implement techniques or with inferior implemented references. A uniform framework, where both the new technique and the reference implementation of existing techniques are available, would help authors and end users in equal measure. The tedious task of providing comparisons would be simplified for software developers. End users would benefit from the better results and at the same time from getting the reference implementations inside a uniform framework.

Evaluation of the approach

The final main objective is to evaluate the approach and its practical realization by applying the developed tool to several *real world* examples. We focus on geometric computations since the combination of continuous and discrete data makes them especially susceptible to rounding and other computational errors [Yap97]. In this scope, we discuss improvements in interval-based hierarchical space decomposition structures for geometric objects and show how they can be used for uniformly handling different geometric modeling techniques. Furthermore, we present a novel algorithm for computing a verified bound on the distance between non-convex objects that applies the developed hierarchical space decomposition structures. This application also demonstrates that the encapsulation provided by the framework allows users to employ the same algorithm with several techniques for range enclosures. This makes it possible to compare them fairly and, thus, give users a better overall database for choosing the "right" technique. At least to our knowledge, such comparisons based on the same problem and algorithm implementation were not available until now.

As another application of the distance computation algorithm, we highlight verification of important subprocesses in an automatic support tool for total hip replacement (THR). In this scope, we apply our algorithm to derive the distances between the femur bone and the femur shaft. They are modeled using non-convex superquadrics (SQs) or non-convex polyhedrons, that is, smooth and non-smooth models.

The second large use-case area is modeling, simulation and control of solid oxide fuel cells (SOFCs). We show how thermal SOFC models can be realized in the unified notations of the framework. After that, we demonstrate how to identify model parameters by using a modular variant of the global optimization algorithm by Hansen and Walster [HW04]. Besides this, we show how to speed up the process

using the GPU. Moreover, we interface different external libraries or solvers with UNIVERMEC, and use them for parameter identification or for simulating the SOFCs.

1.3 RELATED WORK

Since this thesis covers a rather broad field, the complete discussion of related work would be too lengthy. Therefore, more specific aspects will be discussed in the subsequent chapters. Here, we touch upon the general topics of combination and comparison of different techniques.

Contractor programming

The question of how different techniques for verified computations can be combined or compared has recieved some attention lately. For example, Chabert and Jaulin [CJ09] proposed, under the name *contractor programming*, a framework for the flexible and easy reconfiguration of constraint programming solvers. Their idea is to split a classical interval-based solver into components: a paver and one or several contractors. The paver handles the bisection, manages interval boxes and calls the contractors. Users can alter the behavior of the solver by altering the list of contractors of the paver. Contractors are accelerating devices for branch and bound algorithms as explained in the previous section (e.g. interval Newton, hull consistency). The authors claim that it is possible to implement much more flexible solvers than existing ones. The IBEX library and the QUIMPER language [Cha] are implementations of their ideas.

GLOBTLAB

Another environment that allows for flexible configuration of the solving process and user configuration of different techniques is the GLOPTLAB software developed by Domes [Dom09]. It is a MATLAB program for solving quadratic constraint satisfaction problems. Users can configure the solving process of GLOPTLAB by providing an user-defined strategy. This strategy combines the already implemented solving techniques (accelerating devices) inside the branch and bound algorithm and can be adapted to the current problem. Users can extend the tool by adding their own accelerating devices.

Inclusion representation framework

Again in the context of constraint programming, Vu, Sam-Haroud, and Faltings [VSHF09] propose a theoretical framework called *inclusion representation*. This framework can be used to unify different range arithmetics (e.g. IA or AA). Their formalism allows us to derive common definitions for inclusion functions or natural extensions for all arithmetics. The authors use it to demonstrate how constraint programming can benefit from more sophisticated range arithmetics.

VERICOMP

Auer and Rauh [AR12] developed an online platform called VERICOMP. On the one hand, the platform allows for comparing different verified initial value problem (IVP) solvers using a common set of test problems or a user-defined problem. On the other hand, the platform

tries to recommend solver settings for custom problems automatically on the basis of data that was collected on similar problems.

Environment for comparing global optimization codes

Domes, Fuchs, and Schichl [DFS11] developed an environment for testing global optimization codes. It provides interfaces to different solvers. The solvers can be called with a selection of test problems managed by the environment. Furthermore, it can be used to evaluate the results of the solvers almost automatically. A graphical user interface allows for handling of the environment.

Relation with UniVerMeC

While all of the above mentioned software systems overlap with UniVerMeC at one point or another, they all are more specialized towards specific problems (e.g., building a constraint solver or comparing solvers). In their specific domains, they provide additional features that are not offered by UniVerMeC. However, the idea and structure of our tool is more general. It provides problem-dependent modeling layers (Sect. 5) that allow for applying the framework in different domains. In this way, UniVerMeC can be the basis for integrated environments in those domains. For example, both TreeVis (Sect. 8.1) and VeriCell (Sect. 8.2) use the framework as a common code base but target different problem domains. Moreover, tools for comparisons such as VERICOMP can be implemented on top of UniVerMeC to reduce the amount of necessary work to develop them[2]. In conclusion, UniVerMeC fills a conceptual gap because, on the one hand, it can be used to compare existing techniques in a fair manner, and, on the other hand, the framework can be employed as the basis for creating domain dependent verified problem solving environments much more easily.

1.4 STRUCTURE

Requirements

The thesis is structured as follows. In Chap. 2, the conceptual basis for this thesis is provided. Here, requirements for the software, the overall software architecture, possible use-cases, and the organization of user input and output are discussed.

Arithmetics

The following five chapters are organized using the software structure as a guideline: Each of them discusses one layer of the software. In Chap. 3, an overview of the theory and existing implementations of the arithmetics considered inside the framework is given. Furthermore, we analyze how to unify different arithmetics to allow for an interchangeable use while still preserving the property of verification. As a use-case, we describe YalAA, an AA library developed in the scope of the thesis, which already implements some of the guidelines on the arithmetic library level. Besides this, we discuss the automatic generation of adapters and cast operators for the arithmetics as well as the extension of the arithmetic layer to GPU computations.

2 For example, uniform function representation or interfaces to different solvers are already available in UniVerMeC.

Function representation in UNIVERMEC

Problems solved within the developed framework are usually described by functional and relational dependencies. Chap. 4 is concerned with the data type independent, homogeneous representation and implementation of functions in the mathematical sense inside UNIVERMEC. Furthermore, the notion of interval contractors is introduced and the contractor techniques implemented in the framework are explained.

Modeling layer

In Chap. 5, the problem dependent layers in the middle of the framework structure are described. In particular, we give details on the layer devoted to different geometric modeling types supported by the framework, the layer for IVPs and the layer for optimization problems.

Hierarchical decompositions

Chapter 6 discusses common utility data structures used in interval-branch-and-bound algorithms. We describe uniform interfaces for interval trees used in the scope of geometric problems. Furthermore, we introduce the enhanced variants developed in the scope of the thesis, for example, trees for parametric surfaces or trees with integrated contractors. As for non-geometric algorithms, we discuss uniform interfaces and implementations for multisection schemes and box management data structures.

Algorithms

Now we are ready to move to algorithms, which use the tools described in the previous chapters. In Chap. 7, the algorithms currently supplied with the framework are outlined. In particular, the novel verified distance computation algorithm between geometric objects in our uniform representation and the modular verified global optimization algorithm for inequality constrained problems. Furthermore, we discuss how to interface existing solvers, for example, IPOPT [WB06] for optimization or VNODE-LP [Ned06] for IVPs, so that they work on the uniform problem description.

Practical applications

In Chap. 8, we apply the newly developed and implemented algorithms for different purposes. First, we compare different verified techniques in the scope of distance computation. After that, we discuss applications of the framework in the areas of hip replacement surgery and parameter identification and simulation of SOFCs. Finally, a summary of the main results and possible future research directions are given in Chap. 9.

2 UNIVERMEC SOFTWARE

Chapter structure

This chapter gives an overview of the software framework developed in the scope of this thesis. First, it discusses and specifies requirements for the framework with a focus on verification, interoperability, and extensibility. After that, an overview on the architecture of the framework is given. Then, we discuss use-cases for which UniVerMeC can be applied. Finally, we illustrate the input and output paradigm with an example.

2.1 REQUIREMENTS

UniVerMeC framework

UniVerMeC (*Uni*fied Framework for *Ver*ified Geo*Me*tric Computations) is intended to be an integrated platform for providing various verified techniques, for example, IA, AA, TMs, or interval contractors. The goal is to make them available in a uniform environment. Furthermore, the framework should offer uniform abstractions for the supplied different verified techniques, so that it is possible to implement algorithms in the framework that do not depend on a concrete technique but only on its abstraction. The resulting algorithms are highly configurable and flexible. Moreover, the framework should allow users to combine different techniques and to configure interactively[1] the algorithms to adapt them to the their problem domains. In this way, users are encouraged to try out different techniques and to choose the one that fits best.

Optimization goals of users

An important aspect is the fact that each user wants to solve his/her problem with different optimization goals in mind. For example, the following objectives might be of interest: the tightness of the result enclosure, minimization of the CPU time or the memory usage, employment of software in real time environments, or exploitation of computational power of special hardware co-processors, such as GPUs. UniVerMeC should feature a modular architecture to allow users to configure algorithms according to their needs. Note that we do not support real time computations in UniVerMeC. In an inherently modular and object-oriented environment, where subsystems can be transparently replaced, it is hardly possible to meet real time requirements without restricting the flexibility. Thus, UniVerMeC only supports offline computations. Flexibility is the main requirement of UniVerMeC. It is met through the use of abstraction at different levels as shown in Sect. 2.2. However, flexibility comes at the price of a runtime and memory overhead. Less general solutions tai-

1 at runtime

lored towards a specific scenario might have a higher efficiency in their problem domain.

Fair comparisons

Another requirement for UniVerMeC is to allow fair comparisons of different techniques. To make a comparison fair and meaningful, it is important to fix basic conditions. For example, the test hardware, the problem description, the measures, and the algorithms have to be the same except for the techniques to be compared. Especially in the context of numerical comparisons, comparisons are often made useless by the fact that their result depends on the employed implementation. If a new technique is developed, it is as often as not compared to an own implementation instead of an optimized implementation available somewhere else. Another frequently encountered mistake is to compare to a basic method and not to the best one available. For example, a new range enclosure method is only compared to naive interval evaluation and not to more sophisticated ones such as centered forms or AA. Sometimes comparisons focus on one artificial benchmark criterion but not on performance in more complex and more realistic scenarios. Such comparisons can be considered unfair and are not very useful. As a uniform platform, UniVerMeC already provides fixed basic conditions for the test. To allow fair comparisons between sophisticated well-tested implementations of the techniques, UniVerMeC should not reimplement them but allow users to interface existing libraries with it and to exchange these libraries transparently. Therefore, an important requirement is to provide interoperability between techniques supplied by third party libraries.

Summary of requirements

To summarize, the framework should fulfill the following requirements:

- Verification inside the framework
- Support for standards
- Integration and interoperable use of different verified techniques
- Expandability and flexible combination of techniques
- Configurable algorithms to support different optimization goals
- Applicable in different problem domains

These requirements are described in detail below.

2.1.1 *Verification Requirements*

Validation and verification assessment cycle

As mentioned above, a requirement for the UniVerMeC platform is to provide verified computations. To define what we mean by *verified* in the context of UniVerMeC, we consider the validation and verification assessment cycle [AL09] shown in Fig. 1. It consists of three

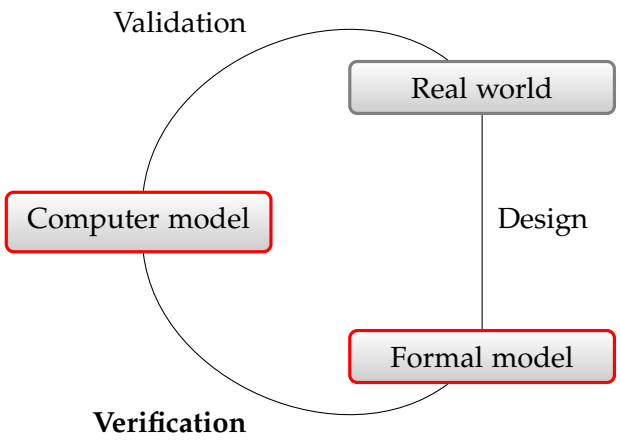


Figure 1: The validation and verification assessment cycle [AL09]. UniVerMeC only performs numerical result verification, which is of use during the transformation of the formal model into a computerized one.

main components: the real world, the formal model, and the computer model. Usually, the real world is analyzed with respect to the application domain in the design step to create the formal model. In this phase, the aspects relevant to the current application domain are identified to integrate them into the formal model. In the second step, the formal model itself is transformed into a computer-based model. This usually means implementing a concrete computer program. At this step, verification can be performed. There are several verification kinds, for example, "code verification", "formal verification", or "result verification". In the third step, the computer model is then validated, that is, the resemblance between the model and the real world is checked.

Basis of verified software

By using UniVerMeC, we are able to cover the verification step and take care of some aspects of validation. The primary goal is, however, to provide numerical algorithms with automatic result verification. They can be characterized by the following "design principle" [Rum10]:

> "Mathematical theorems are formulated whose assumptions are verified with the aid of a computer."

One way to achieve this is to use numerical algorithms in conjunction with rigorous arithmetics and fixed point theorems to account for rounding errors and to obtain verified results. Often, it is not sufficient just to replace the standard FPA by a rigorous one. In addition, we have to account for the approximation errors of the algorithms. Verification techniques provided by UniVerMeC cope with numerical errors introduced by the use of finite arithmetics and with approximation errors inside the algorithms. Although, we do not handle modeling errors, it is possible to account for modeling errors in

parameters inside the framework by introducing interval uncertainties into model descriptions. Most methods supplied by UniVerMeC can cope with interval uncertainties in principle, owing to the use of range arithmetics.

Taxonomy for numerical verification

To classify software tools further, Auer and Luther [AL09] introduced a taxonomy for numerical verification with four classes. The degree of verification in each class increases from the fourth to the first class. Basically, no guarantees about the result are given if an application belongs to the fourth class, whereas the first one demands full numerical or analytical verification, employment of code verification, and appropriate handling of uncertainties. The two classes in-between require the use of IEEE 754 arithmetic and a guarantee of stability, for example, by sensitivity analysis (the third class). Additionally, "relevant subsystems" should be "implemented using tools with result verification" (the second class). Complete definitions are to be found in [AL09]. According to this taxonomy, UniVerMeC fulfills at least the requirements for Class 2. In fact, it can be classified somewhere between Class 1 and 2 because it provides full result verification, and model uncertainties can be handled by range arithmetics. However, no code verification for its implementation is provided.

2.1.2 *Standard Requirements*

Standard compliance

The trustworthiness of the results produced by a system with or without result verification depends not only on the used methods but also on whether employed hardware or software libraries deliver reliable results with sufficient guarantees. Therefore, the above mentioned taxonomy for numerical verification requires highest verification class software and hardware to conform to the IEEE 754 standard and to an upcoming interval standard, which is currently being developed under the name IEEE P1788.

Non-standardized arithmetics

UniVerMeC complies with these requirements by employing IEEE 754-2008 double precision arithmetic. Furthermore, the uniform interface defined for all arithmetics is oriented towards the upcoming interval standard IEEE P1788. However, UniVerMeC does not provide its own arithmetic implementation but uses already existing ones. Therefore, the extent to which the IA in the framework conforms to the standard depends on the chosen external implementation. Because other range arithmetics such as affine forms or TMs, have no standard currently, the overall standardization in UniVerMeC is more complicated. Still, if implementations orient themselves to the upcoming interval standard, employment in an environment such as UniVerMeC is much easier. An example is our own library YalAA [Kie12b] for AA, which tries to emulate the interface of IEEE P1788 as much as possible.

2.1.3 *Interoperability Requirements*

Interoperability at different levels

Because of the limited standard support in the external tools, UniVerMeC has to ensure the interoperable use of different techniques by itself. According to ISO/IEC 2382-1:1993, interoperability is defined as:

Definition 1 (Interoperability [Isob]) *"The capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units."*

In the scope of UniVerMeC, we are concerned with interoperability at different levels. The first interoperability problems appear directly if we do not want to implement all techniques by ourselves but use already developed libraries. To reach our goal of interoperability we have to overcome the problem that libraries usually do not share the same interfaces even if they implement the same technique or method, that is, we cannot use them interchangeably in a straightforward way. Furthermore, we want to allow users to employ various techniques or combinations of them. Therefore, we have to ensure interoperability between different techniques without loosing the current degree of verification. To conform to the above definition, UniVerMeC should handle interoperability automatically and transparently for the end-user as much as possible. Above we discussed the interoperability with respect to software. However, very similar problems appear if modern many-core-architectures, for example, GPUs, are applied to accelerate computations. Because these platforms currently have a still very limited tool support compared to the CPU, the automatic and transparent handling is not as far-reaching as in the CPU case. However, such problems as transferring CPU intervals (independent of the applied CPU library) to the GPU and back are handled by UniVerMeC automatically, if requested by the user.

Importance of IEEE P1788

The foundation for fulfilling the interoperability requirements will be the use of IEEE 754-2008 FP arithmetic. It is the basic building block for almost all verified techniques employed in UniVerMeC and is used to transfer data between the different libraries. Furthermore, the IEEE 754-2008 data types are available not only on the CPU but also on modern GPUs [WFF11] and can therefore be used for transferring data between host and GPU without loss of information. The common basis for using different techniques will be further discussed in Sect. 3.5 and the integration of the GPUin Sect. 3.7.

2.1.4 *Expandability Requirements*

Integration of new techniques

As already mentioned in the introduction of this thesis, our goal is not to provide a *complete* framework but a *proof of concept*. That is, on the

one hand UniVerMeC should provide a number of techniques that are available to users readily and that allow us to assess whether the approach can be used to solve practical problems in different application areas. On the other hand, that means that UniVerMeC should act as a skeletal structure that users can extend with new techniques to fulfill their needs. Therefore, an important requirement for the developed software is to provide abstract interfaces that can be used to describe a wide class of techniques. Users should be able to implement and integrate new techniques in the framework with the interfaces. This would automatically allow for interaction with the rest of UniVerMeC.

Formal framework

Prior to actual implementation, it is necessary to provide a theoretically motivated set of formalizations and definitions for the basic building blocks such as arithmetics, contractors or types of enclosures employed in verified computations. This theoretical work ensures that the provided abstract interfaces can cover a large number of techniques and should result in a set of well-founded interfaces in the actual software.

2.2 SOFTWARE ARCHITECTURE

Design patterns

UniVerMeC uses modern software engineering techniques to fulfill the requirements outlined above. To decouple parts of the system, we rely mainly on the object-oriented paradigm during the framework's design, but also use other techniques from generative programming [CEoo], for example, template metaprogramming or semi-automatical program code generation. During the design of complex software projects, developers often face the same kind of problems independent of the actual project content. For such a problem, a "well-proven generic scheme for its solution" [Bus+96, p. 8] is called a design-pattern, a technique which was popularized by Gamma et al. [Gam+95]. If possible, UniVerMeC tries to make use of existing design-patterns.

Relaxed layered structure

Design-patterns can be applied to implement concepts at different levels of abstraction. If they are applied at the highest level, where they "express a fundamental structural organization schema for software systems" [Bus+96, p. 12], they are also called *architectural pattern*. A well-known example for an architectural pattern is a layered structure [Bus+96, pp. 31-51]. It decomposes a system into a set of layers, where each layer represents a certain level of abstraction. For decoupling the layers, it also imposes certain restrictions which layers can communicate to each other and defines how communication is performed. Usually, in a strict layered architecture, only adjacent layers can communicate. In this case, lower layers are not aware of the higher ones, and a higher level layer only knows the direct adjacent lower level one. An example of the layered structure pattern in action

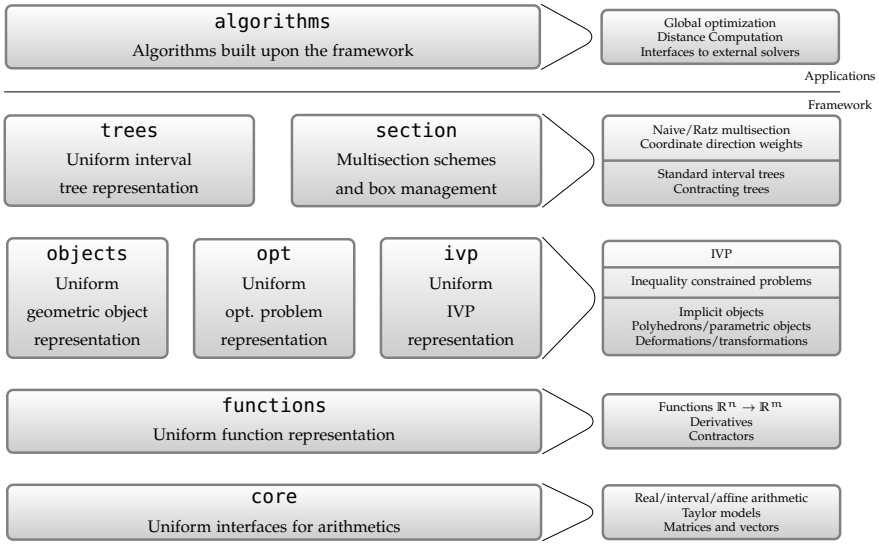


Figure 2: UniVerMeC uses a relaxed layer structure. The figure denotes all layers, their main goals and examples for techniques, and methods they provide.

is the *Open Systems Interconnection Reference Model* [Isoa], which plays an important role in the scope of the design of networking protocols.

UniVerMeC also uses a layered structure, but it does not follow the strict model and relaxes the decoupling between the layers slightly. In the *relaxed model* a higher layer can access all lower layers directly. This looser decoupling results in a performance gain, since a call to a function or service on a lower layer does not need to be passed through intermediate layers at a price of less maintainability [Bus+96, p. 45]. Another important reason for relaxing the layer requirement was that the lower layers of UniVerMeC define interfaces for such important concepts as mathematical functions. From our point of view, it is necessary to make these concepts accessible to all higher layers.

Arithmetic layer

The five layers of UniVerMeC are shown in Fig. 2. At the lowest level, we have the `core` layer, which provides the supported arithmetics of the framework. All arithmetics can be accessed through a uniform interface. Note that arithmetics are not actually implemented at this level, but in external libraries. These do not form an own layer, but are hidden and thereby cannot be accessed by higher layers directly. Furthermore, at this level, vector and matrix classes for all supported arithmetics are provided.

Function representation layer

The second layer provides a uniform representation for scalar-valued functions $f : \mathbb{R}^n \to \mathbb{R}$ and for vector-valued functions $f : \mathbb{R}^n \to \mathbb{R}^m$. It supplies the users with tools that they can use to create their own functions in this representation and ensures that extensions of the functions can be evaluated with all arithmetics supported by the `core` layer. Also, it describes how function-related concepts, such as differentiability, are represented in the framework. Built upon these

functionalities, the layer provides a uniform representation of interval contractors and enclosures.

Modeling layer

The modeling layer lies in the middle of the framework. It is divided into three independent layers: the `objects` layer, which represents geometric objects, the `opt` layer for optimization problems, and the `ivp` layer for IVPs. The sublayers depend on the problem domain. The `objects` layer maps discrete geometric structures, such as polyhedrons and smooth structures (e.g., implicit objects) on a uniform description. The layer describes objects independently of their underlying modeling type. Furthermore, it provides deformations or transformations to alter objects. In general, an optimization problem consists of an objective function, inequality constraints, and bounds on the variables. They are assembled in the `opt` sublayer to provide a uniform representation. The `ivp` sublayer combines the function describing the problem's right-hand side with the related data, for example, initial values or possibly time dependent parameters, into an abstract IVP description. It can be solved by IVP solvers interfaced to the framework.

Decomposition layer

Basically, the fourth layer consists of two separated layers. On the one hand, we have the `trees` part, which defines and supplies hierarchical space decomposition structures working on the geometric objects descriptions generated by the `objects` layer. On the other hand, we have the `section` layer, which provides multisection schemes. They do not work on objects but only on box-shaped regions in a user-defined space. Note that both layers basically provide utility data structures for the algorithms. Consequently, the structures of the `trees` layer are *not* considered to be geometric objects.

Algorithm layer

Algorithms making use of the services and methods provided by UniVerMeC are implemented at the topmost layer. Currently, a verified global optimization method and an algorithm for computing an enclosure of the distance between possibly non-convex objects are available. Furthermore, interfaces to solvers for optimization and IVPs are provided on this level. They make use of the problem description capability of UniVerMeC. Algorithms within the framework can employ them, for example, to speed up the computational process.

Design-patterns inside layers

The description of the layers only gives an overview of the overall structure of UniVerMeC and the most important subtasks solved at specific levels. Inside the layers, further design-patterns are applied, for example, *adapter*, *facade*, or *strategy* [Gam+95]. The facade pattern hides the internal complexity of a layer. It usually generalizes several concepts represented by abstract interfaces in a subsystem by providing a uniform interface. Several facades can be nested to create a hierarchy.

Figure 3 shows the hierarchy of facades in the `objects` layer of UniVerMeC. The most general concept at this level is the *uniform*

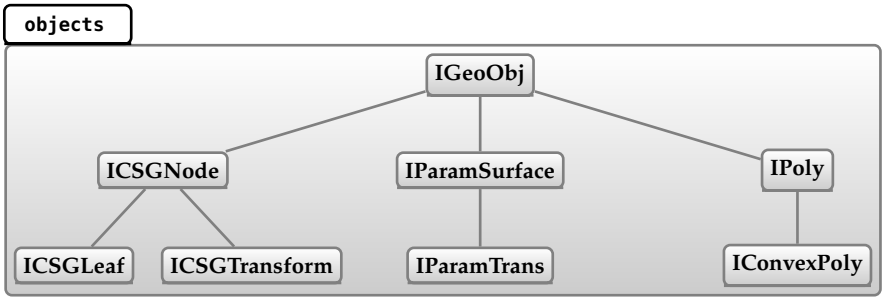


Figure 3: Abstract interfaces forming a hierarchy of facades in the `objects` layer. Interfaces at the bottom of the hierarchy represent more specialized concepts.

object representation provided by the abstract `IGeoObj` interface. It describes an object by an in/out function, that is, a function that returns whether a point or a region belongs to the object (lies inside it) or not. Closed objects can be defined by such a function, independently of whether they are constructed by a single implicit function, by several computer solid geometry (CSG) operations, by a (deformed or non-deformed) parametric function or by a polyhedron. If a method on a higher level is only capable of handling some specific modeling type, for example, an algorithm for simplifying a CSG tree, it can request a more specialized interface as input. Note that even the most specialized interfaces still hide many implementation details. For example, CSG operations can be implemented by different branches of R-functions [Sha91]. The information about what branch is actually used is never propagated to a higher level, but is always hidden as the implementation detail. Since only abstract interfaces pass over layer boundaries, the implementations can be exchanged. More details on the design of the individual layers and the methods they provide are given in the respective chapters later on.

2.3 USE-CASES

Used algorithms

We evaluate our approach during this thesis using several use-cases, which are outlined in this section. The evaluation is mainly based on two algorithms. They are implemented on the top of UNIVERMEC: distance computation (Sect. 7.1) and global optimization (Sect. 7.2). The former computes verified bounds on the distance between two objects of which both can be non-convex. It works solely on the tree representation of the objects and thus is completely decoupled from the underlying modeling type (e.g., polyhedron, implicit object, parametric object). Therefore, the algorithm can compute the distance between objects described by two different modeling types (e.g., between polyhedrons and parametric objects). The global optimization algorithm does not work with tree decompositions but uses the sec-

tion layer for space decomposition. It solves classical inequality constrained optimization problems.

Comparison of range-enclosure methods

The first use-case is the comparison and evaluation of different verified techniques for range enclosure in the scope of distance computation. Classical IA overestimates the range of a function in general (Sect. 3.2.5). Sometimes the overestimation is so large that the obtained results are useless. For example, a branch and bound algorithm can be forced into massive recursion due to clustering effects occurring near (local) minimums [DK94] because of overestimation. Several more sophisticated enclosure methods, for example, AA or TMs were, proposed to overcome these problems. However, a fair comparison between these new techniques is still lacking. Using UniVerMeC, we compared how long it takes to compute an enclosure of the distance between two smooth objects up to a user-specified enclosure width. The framework allows us to use the same implementation of the algorithm for all considered techniques and ensures that the overall overhead inside the framework is always the same, thus producing fair comparison results.

Total hip replacement

In the second use-case, we consider a system for automatic surgery planning for THR [Cuy11], which was developed in the recent project PROREOP [Pro]. It uses possibly non-convex multi-component SQ [Bar81] models or polyhedral models. To ensure that the automatically selected implant fits, several distance computations between two SQs or between an SQ and a polyhedron are carried out. Using UniVerMeC, it is possible to derive verified bounds on the distances even in the non-convex cases and to ensure that the selected implants fit with certainty. Because SQs can be described both parametrically and implicitly, we are also going to compare distance computation between parametrically and implicitly described objects.

Parameter identification of SOFCs

The third use-case for the framework is concerned with the parameter identification for a thermal SOFC model [AKR12] emerging from the research project VerIPC-SOFC. Parameters of a model are usually identified by solving an optimization problem with a quadratic error measure as the objective function to minimize. It is based on the deviation of the simulated values for a considered set of parameters from the measured ones. The goal is to parametrize the model so that it resembles the real SOFC stack as closely as possible. Since the model function is complex[2], it is hard to find a global solution to the problem in general. Furthermore, it is also very expensive to evaluate the function. Therefore, we will also present results that use GPU acceleration.

Combination of different solvers

As a fourth use-case we highlight the possibility to input problems and then solve them using different solvers that have interfaces to UniVerMeC. As an example, we show how different verified and

2 It depends on the sum of differences to over 19000 measurements and the simulated temperatures can be derived only by solving an IVP numerically.

non-verified IVP solvers perform in the simulation of a SOFC model described by the `ivp` layer of UniVerMeC, as well as how we can apply FP optimization algorithms for parameter identification of the SOFCs and then validate their results using verified IVP solvers.

2.4 USER INPUT AND OUTPUT

Model specification in UniVerMeC

Usually, solving a problem starts with creating an appropriate model by analyzing the real world and identifying aspects relevant to the current problem, as represented by the verification and validation assessment cycle in Fig. 1. A model is a simplified and idealized view on the world. Typically it introduces some modeling error. Models in UniVerMeC are represented using one of the model description layers of: `objects`, `opt`, or `ivp`. They are problem dependent. That is, if users want to work on a model type not currently supported, they have to add an additional model description layer. As seen in Fig. 2, all model description layers are placed at the same level in the middle of the framework above the `function` and below the decomposition layer. This positioning is not arbitrary. It enables the model description layers to employ concepts defined at lower levels for describing models. In particular, it is possible to use intervals, vectors, matrices, and mathematical functions to specify a model.

Organization of model input

Now consider a user who wants to input an already existing model into the framework. Often, such a model is composed of entities provided at lower layers of the framework. Because the input in UniVerMeC is organized for each layer separately, it is not possible to input it directly but first to input the parts of the model described at the lower layers. Then all parts are assembled together to form the complete model. In general, we can say that users have to perform the necessary steps to input a model in the reversed data flow order of the framework, that is, they have to start in Fig. 2 at the bottom and proceed to the top. This is a direct result of the use of a relaxed layered architecture because it allows (and forces) us to use concepts[3] defined at lower layers to describe more abstract concepts at higher layers.

Example

To illustrate this approach consider the application of UniVerMeC in the scope of the PROREOP project as outlined in Sect. 2.3. The goal is to calculate a bound on the distance between an implant and the femur-shaft which are both modeled by a multi-component SQ model. If we apply UniVerMeC to the problem, we get the input and outputs as depicted in Fig. 4. In the first step, users can configure options at the arithmetic level, for example, the bounder for converting a TM to an interval, if they are not satisfied with the standard settings. On the

3 Their use is forced because higher level layers do not provide any tools to describe entities which were defined at lowers levels.

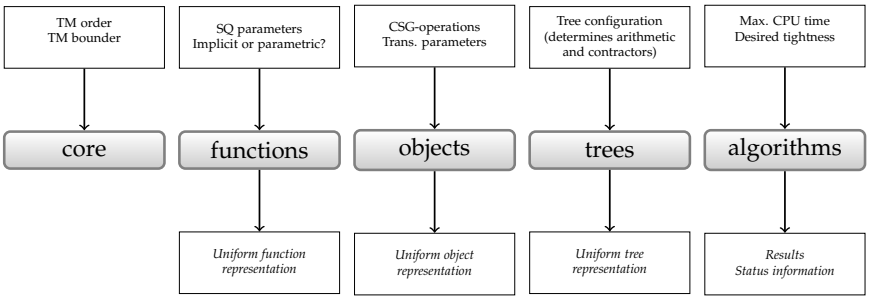


Figure 4: User input and corresponding ouput at the different abstraction levels of UniVerMeC for the distance computation use-case.

next layer, users can input the SQ's parameters[4] and choose between an implicit or parametric description. The returned uniform function representation is used at the next level to define implicit or parametric surfaces or to apply CSG operations, deformations (e.g. tapering and bending), and affine transformations. The result is a uniform object representation. It describes the multi-component SQ model. At the next layer, users can configure the hierarchical space decomposition structures. The tree configuration done here determines what arithmetic to use for a function range enclosure and if contractors or other sophisticated techniques are applied to improve the decomposition's quality. Basically, the trees are just utility data structures for the actual algorithm to calculate the distance. This enables us to decouple the algorithm from the object representation, range bounders and so on. At the top level, users set the algorithm's options, for example, the desired tightness of the bound.

Use of GUIs

While the framework gives an output at every layer, most users will only be interested in the values returned by the algorithms on the topmost layer. Therefore, these parts of the framework are linked into a dynamic library called `libunivermec`, which is then utilized by graphical user interface (GUI) programs as depicted in Fig. 5. They hide the intermediate steps outlined in this section.

TreeVis

The program TreeVis was developed in the scope of project TellHim&S. It provides access to the geometric part of UniVerMeC, that is, the geometric objects layer, the distance computation algorithm, and the hierarchical decomposition structures. It allows users to define implicit or parametric objects based on smooth functions easily by entering the formulas directly or by choosing predefined ones. Furthermore, it can load polyhedrons into the framework. The trees are configured through the user interface graphically and then the distance computation can be run on them. It can visualize the trees and distances using OpenGL. All these steps are carried out on a special

4 Strictly speaking, the SQ parameters, for example, roundness or scaling are FP numbers or intervals and belong to the `core` layer and have to be entered there. We omit this details for clarity reasons.

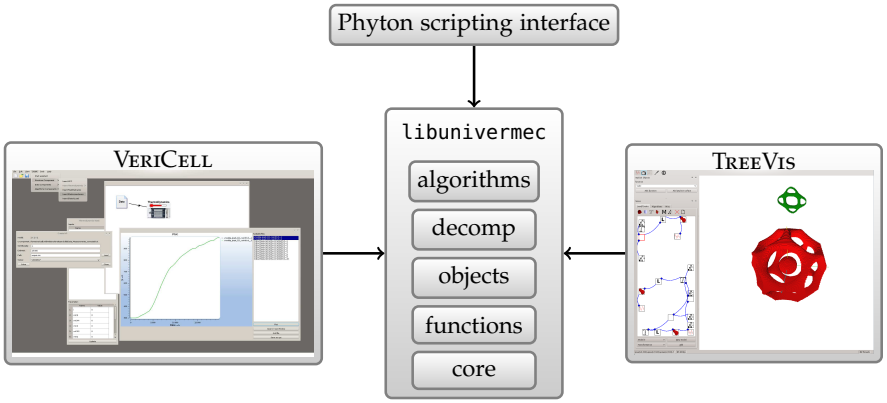


Figure 5: User interfaces of integrated problem solving environments built upon UniVerMeC.

scene graph that takes care of resource management and can be used to store and load configurations.

VeriCell

A second GUI called VeriCell is being currently developed in the project VerIPC-SOFC and an upcoming master thesis [Pus13]. It uses UniVerMeC to represent IVPs that model SOFCs. The framework acts as an interoperability layer to interface external solvers, for example, VNODE-LP [Ned06] or ValEncIA-IVP [RA11]. They are used to perform simulations with models. Furthermore, the VeriCell software offers graphical access to UniVerMeC's global optimization algorithm to identify model parameters.

Scripting interface

Finally, it is also possible to use UniVerMeC from the Python scripting language through an extension module. The scripting interface is well suited for repetitive tasks, such as comparing different techniques. It also allows for the use of the framework without a GUI in comparisons because the visualization may interfere with their results.

2.5 CONCLUSIONS

Six main requirements

In this chapter, we have discussed the requirements that an interoperability framework in the scope of verified computations should fulfill. During this process we have identified six main requirements. We have also outlined some concepts and techniques that help us to fulfill the requirements and build UniVerMeC. In Fig. 6, the six main requirements are directly connected with UniVerMeC. For each requirement some of the techniques and concepts employed to fulfill them and the most important sections of the thesis where these concepts are applied are given. For example, the interoperability requirement between the different arithmetics and their implementation is fulfilled through an adaption of the inclusion representation framework by Vu, Sam-Haroud, and Faltings [VSHF09] (cf. Sect. 3.5.1) and automatic adapter generation (cf. Sect. 3.6). Another important as-

Figure 6: Overview of the requirements of UniVerMeC and their dependencies of each other. For each requirement some of the concepts used to fulfill it are depicted. Additionally, the parts of the thesis where the depicted concepts play an important role are listed.

pect is revealed by the figure: the requirements depend on each other. For example, interoperability and verification necessitate support for IEEE 754-2008 . Also some concepts that are used to fulfill the requirements related to several of them (e.g., object oriented design).

3 ARITHMETICS

Rigorous arithmetics

The purpose of this chapter is twofold: the first four sections give an introduction into the basic arithmetics for verified computations on a digital computer, and the remaining sections discuss the interoperability of the mentioned arithmetics. Arithmetics are the crucial basic building blocks for verified computations. Therefore, different arithmetics and even more variants of them have been proposed during the last 50 years. The chapter introduces the most commonly used rigorous range arithmetics: interval arithmetic, affine arithmetic, and Taylor models. The discussion highlights not only their theoretical aspects but also their different implementations. We will show through the example of AA how to develop a generic library which fits well into existing interval environment and handles numerous variants of AA by employing policy-based design.

Improving range-bounds

It is well-known that, besides special cases, rigorous range arithmetics, such as IA or AA only return a rough outer bound, that is, an over-approximating bound on the range of an expression. Usually, we can improve the bounds by subdividing the considered range that leads to the classic branch and bound algorithms, which are applied, for example, in global optimization. However, their brute force subdivision of the search space leads to an exponential runtime. Another way to obtain sharper bounds is to use different methods for range bounding, because, depending on the expression, either interval or affine arithmetic might provide better bounds for its range. It is even possible to intersect the bounds computed by different methods in order to improve them. This is, for example, offered by LIBAA [Sto] library for IA and AA .

Inclusion representation

Recently, Vu, Sam-Haroud, and Faltings [VSHF09] proposed a general formal framework for using different rigorous range arithmetics simultaneously in the scope of constraint programming. They introduce the notion of a so-called *inclusion representation* and derive on its basis formalizations for inclusion functions and natural extensions independent of the actual arithmetic. Furthermore, they show that their inclusion representation can be applied to IA and different variants of AA. However, they do not discuss the practical consequences of providing an environment that supports different arithmetics and possibly different libraries for each of them. One goal of this chapter is to build a theoretical basis *and* an implementation that solves practical problems while combining different arithmetics.

Chapter structure

The chapter starts with a brief discussion of FPA in Sect. 3.1. It is followed by Sect. 3.2, which introduces IA, the upcoming IEEE

P1788, and possible implementations. AA, its variants, and our own implementation YalAA are discussed throughout in Sect. 3.3. TMs are the last of our four supported arithmetics, and described in Sect. 3.4. Based on the work by Vu, Sam-Haroud, and Faltings, which we outline in Sect. 3.5, we build the theoretical foundation for the arithmetic layer of our software. Moreover, we extend their work by presenting an inclusion representation for TMs. As a preparation for the practical implementation based on the inclusion representation, we formulate a heterogeneous algebra for all arithmetics, which defines a common set of operations, elementary functions and the possibility of operations between different arithmetics. As a foundation for our set of functions, we chose the upcoming IEEE P1788 interval standard. After that, we present possible conversions between the arithmetics, also based on the inclusion representations. Finally, in Sect. 3.6 we combine our theoretical considerations to provide a generator framework for adapters. The adapters should tackle the practical challenges evolving from the use of different arithmetics and libraries provided by different vendors. Finally, Sect. 3.7 discusses interoperability challenges that come into play if we work in a heterogeneous CPU/GPU environment.

3.1 FLOATING-POINT ARITHMETIC

Floating-point system

By default, even a modern computer does not perform arithmetic operations on the real numbers exactly in hardware. Instead, the operands are usually converted to machine number (e.g., FP numbers). With these, operations are performed that only approximate their exact counterpart in general. Modern computers usually contain a FP unit that conforms with the IEEE 754-2008 [Iee] standard. Following Muller et al. [Mul+10, pp. 13-20], we define a FP system as an integer 4-tuple $(\beta, p, e_{\min}, e_{\max})$ with the base $\beta \geqslant 2$, the precision $p \geqslant 2$, and the minimum and maximum exponents $e_{\min}, e_{\max}$. A number x belongs to the FP system if it can be represented by a 3-tuple (s, m, e):

$$x = (-1)^s \cdot m \cdot \beta^e$$

where $s \in \{0, 1\}$ is the sign, e the integer exponent with $e_{\min} \leqslant e \leqslant e_{\max}$ and $m = |M| \cdot \beta^{1-p}$ the significand. It holds $0 \leqslant m < \beta$ and the integral significand M satisfies $|M| \leqslant \beta^{p-1}$. A normalized representation has to comply with $1 \leqslant |m| < \beta$.

IEEE 754-2008 standard

IEEE 754-2008 specifies several FP formats. However, the single and double precision formats (cf. Tab. 2) are the most widely used. In the rest of this work, we denote the set of double precision FP numbers excluding $\pm\infty$ and `NaN` by $\mathbb{F}$. The extended set of FP numbers $\mathbb{F} \cup \{-\infty, \infty, \texttt{NaN}\}$ is denoted by $\overline{\mathbb{F}}$. Accordingly, $\overline{\mathbb{R}}$ denotes the set of extended real numbers $\mathbb{R} \cup \{-\infty, \infty\}$. In general, we will call mem-

Table 2: IEEE 754-2008 single and double precision formats

FORMAT	p	e_{min}	e_{max}
Single	24	-126	127
Double	53	-1022	1023

bers of $\overline{\mathbb{F}}$ that are not in $\mathbb{F}$ *special forms*. The operators $\mathbb{F}(\mathcal{D})$ and $\overline{\mathbb{F}}(\mathcal{D}')$ return the set of all (extended) FP numbers contained in $\mathcal{D} \subseteq \mathbb{R}$ and $\mathcal{D}' \subseteq \overline{\mathbb{R}}$ respectively. The set of FP numbers is finite. Thus, in general, a number $x \in \mathbb{R}$ has to be approximated by a FP number $\tilde{x} \in \overline{\mathbb{F}}$. The approximated number is derived by a mapping $\mathrm{fl} : \mathbb{R} \to \overline{\mathbb{F}}$, which is called a rounding [Rum10, p. 302], [Mul+10, pp. 20-25]. The IEEE 754-2008 provides several definitions for a rounding. The standard mode *round to nearest* chooses the FP number with the minimum deviation from the real result

$$|\mathrm{fl}(x) - x| = \min_{f \in \overline{\mathbb{F}}} |f - x| \ .$$

If the result lies exactly between two FP numbers, the even number is chosen. The two directed rounding modes *round to* $\pm\infty$ are of special interest for our work. The rounding $\mathrm{fl}_{+\infty}(x)$ returns the smallest FP number greater than x, whereas $\mathrm{fl}_{-\infty}(x)$ returns the largest FP number less than x.

Correctly rounded

Unfortunately, the IEEE 754-2008 standard only defines the four basic arithmetic operations $\{+, -, \cdot, /\}$ and square root. They can be defined using a rounding [Rum10, p. 302], [Mul+10, pp. 20-25] by

$$x \tilde{\circ} y \coloneqq \mathrm{fl}(x \circ y) \ ,$$

where $x, y \in \overline{\mathbb{F}}, \circ \in \{+, -, \cdot, /\}$ is the real- and $\tilde{\circ}$ the corresponding FP operation. We call these operations *correctly rounded* because their result is identical to the result of the exact operation rounded to the next FP number under the current rounding. The results obtained for other elementary functions, such as exponential, logarithm or trigonometric functions provided by modern processors and programming environments are often not correctly rounded. In specific, the IEEE 754-2008 does not make any guarantees for them.

3.2 INTERVAL ARITHMETIC

Interval analysis

While interval analysis is older than digital computer systems, the modern use of it in verified numerics on computers is widely attributed to Ramon E. Moore, who published his book *Interval Analysis* in 1966 [Moo66]. IA is capable of deriving rigorous lower and upper bounds on an exact result even in the presence of rounding errors. It

can compute (rigorous) bounds on a function's codomain over some axis-aligned box. Our discussion of the basic properties of interval arithmetic follows [AH83].

3.2.1 *Basic Arithmetic*

Compact real interval

A compact real interval

$$\mathbf{x} = [\underline{x}, \overline{x}] = \left\{ x \;\middle|\; \underline{x} \leqslant x \leqslant \overline{x} \right\}$$

consists of two endpoints $\underline{x}, \overline{x} \in \mathbb{R}$, which we call lower and upper bound respectively. We denote the set of all compact and non-empty real intervals with $\mathbb{IR}$. Furthermore, we define the midpoint of an interval

$$\operatorname{mid}(\mathbf{x}) \coloneqq \frac{\overline{x} + \underline{x}}{2} \;, \tag{1}$$

its width

$$\operatorname{wid}(\mathbf{x}) \coloneqq \overline{x} - \underline{x} \;, \tag{2}$$

its radius

$$\operatorname{rad}(\mathbf{x}) \coloneqq \frac{\overline{x} - \underline{x}}{2} \;, \tag{3}$$

and the midpoint-radius[1] form

$$< \operatorname{mid} \mathbf{x}, \operatorname{rad} \mathbf{x} > \coloneqq \left\{ x \;\middle|\; \operatorname{mid} \mathbf{x} - \operatorname{rad} \mathbf{x} \leqslant x \leqslant \operatorname{mid} \mathbf{x} + \operatorname{rad} \mathbf{x} \right\} = \mathbf{x} \;.$$

An interval is empty if $\underline{x} > \overline{x}$ and unbounded if at least one endpoint is $\pm\infty$. The extended set of intervals $\overline{\mathbb{IR}}$ includes empty and unbounded intervals. As for the FP numbers, we call members of $\overline{\mathbb{IR}}$ those that are not in $\mathbb{IR}$ special forms. Furthermore, the $\mathbb{IR}(\mathcal{D})$ operator returns the set of all real intervals over $\mathcal{D} \subset \mathbb{R}$.

Basic operations

A basic arithmetic operation $\circ \in \{+, -, \cdot, /\}$ is defined for two intervals $\mathbf{x}, \mathbf{y} \in \mathbb{IR}$ as:

$$\mathbf{x} \circ \mathbf{y} = \left\{ x \circ y \;\middle|\; x \in \mathbf{x}, y \in \mathbf{y} \right\} \;. \tag{4}$$

1 All intervals in this work are considered to be in endpoint form if not specified otherwise.

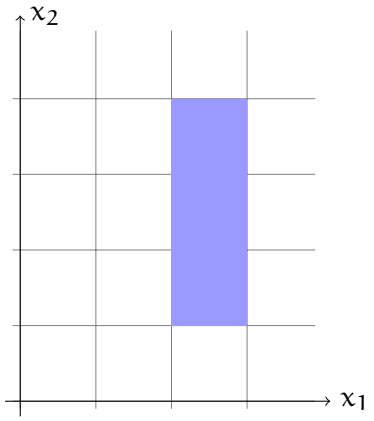


Figure 7: The rectangle represented by the two dimensional interval vector $([2,3],[1,4])$.

The division is only defined if $0 \notin \mathbf{y}$ holds. Explicit endpoint formulas can be derived for all operations:

$$\begin{aligned}
\mathbf{x}+\mathbf{y} &= \left[\underline{x}+\underline{y}, \overline{x}+\overline{y}\right] \\
\mathbf{x}-\mathbf{y} &= \left[\underline{x}-\overline{y}, \overline{x}-\underline{y}\right] \\
\mathbf{x}\cdot\mathbf{y} &= \left[\min\left(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\right), \max\left(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\right)\right] \\
1/\mathbf{x} &= \left[\frac{1}{\overline{x}}, \frac{1}{\underline{x}}\right] \quad , \qquad \text{for } 0 \notin \mathbf{x} \ .
\end{aligned} \tag{5}$$

The result of all operations (5) is again an interval. Therefore, $\mathbb{IR}$ is closed under the basic arithmetic operations. We call a function $F : \mathbb{IR}^n \supset \mathcal{D} \to \mathbb{IR}^m$ an inclusion function for $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}^m$ over an interval box $\mathbf{x} \in \mathbb{IR}^n$ if the inclusion property

$$F(\mathbf{x}) \supseteq \left\{ f(x) \;\middle|\; x \in \mathbf{x} \right\} \tag{6}$$

holds. An interval vector $\mathbf{x} \in \mathbb{IR}^n$ is an axis-aligned n-dimensional cuboid, which we will also call a box. Geometrically it is a rectangle in two dimensions (cf. Fig. 7).

Machine intervals

On a computer, we work with machine intervals, the endpoints of which are FP numbers. We denote the set of all bounded and non-empty machine intervals with $\mathbb{IF}$. If we include empty and unbounded intervals, we obtain $\overline{\mathbb{IF}}$. The $\mathbb{IF}(\mathcal{D})$ and $\overline{\mathbb{IF}}(\mathcal{D}')$ operators return the set of all machine intervals over $\mathcal{D} \subseteq \mathbb{R}, \mathcal{D}' \subseteq \overline{\mathbb{R}}$. For machine intervals, the basic operations also obey the inclusion property, that is, $\mathbf{x} \circ \mathbf{y} \subseteq \mathbf{x} \boxdot \mathbf{y}$ holds for the real operations $\circ \in \{+, -, \cdot, /\}$ and the respective machine operation $\boxdot$. The more strict definition given in (4) does not hold in general. Usually, this is achieved by using directed roundings for the formulas (5). That is, the result of a real operation is rounded towards $-\infty$ and $+\infty$ for the lower and upper bound respectively. Note that (4) can yield a member of $\overline{\mathbb{IF}}$ in general even if all operands are members of $\mathbb{IF}$.

Elementary functions

Usually, not only the basic operations (5) but also elementary functions, such as sine, cosine, logarithm, or exponential, are needed in a practical computation. Interval extensions of these satisfying (6) can be constructed, for example, through polynomial series expansions or by considering their monotonicity features. Ready to use functions are provided by interval libraries, for example, C-XSC [HK04] or FILIB++ [Ler+06]. One problem with these implementations is that they do not necessarily provide the same set of functions or the domain on which the interval extension is defined. These incompatibilities are tackled by the upcoming interval standard IEEE P1788 (cf. Sect. 3.2.3), where a set of elementary standard functions, including their domains, has been defined. While to our knowledge no library yet fully supports the functions required by IEEE P1788, we can assume in the remainder of this work that usable interval extensions of the common elementary functions are available.

3.2.2 *Natural Interval Extension*

Natural extension

An easy way to construct a function satisfying the inclusion property (6) is the *natural interval extension*.

Definition 2 (Natural interval extension) *Let an inductively described function*[2] $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}^m$ *be given. If we replace all quantities in* f *by their respective interval counterparts, we obtain an inclusion function* F *for* f*, which we call the natural interval extension of* f*.*

Properties of the natural extension

The natural interval extension F is inclusion monotonic

$$\mathbf{x} \subset \mathbf{y} \implies F(\mathbf{x}) \subset F(\mathbf{y}) \tag{7}$$

and thin

$$F(\mathbf{x}) = f(x) \tag{8}$$

for $\mathbf{x} = [x, x]$. Furthermore, if every variable appears only once in the expression describing f, and if all used operations and elementary functions are continuous, then F is minimal. This means that for a given $\mathbf{y} \in \mathbb{IR}^n$

$$\neg\exists \mathbf{x} \in \mathbb{IR} : F(\mathbf{y}) \setminus \mathbf{x} \neq \emptyset \wedge \forall y \in \mathbf{y} : f(y) \in \mathbf{x} \ , \tag{9}$$

that is, F computes the smallest box satisfying the inclusion property (6) [Jau+01, Theorem 2.2].

2 That is, the function is given by a finite composition of the basic operations $\{+, -, \cdot, /\}$, elementary functions, variables and constants.

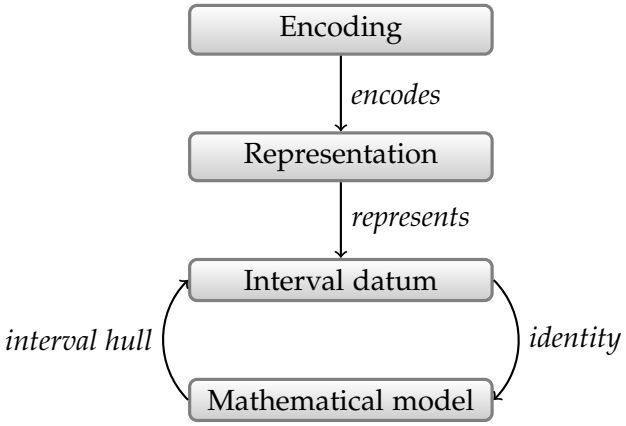


Figure 8: The layers of IEEE P1788 as defined in [P17, pp. 13-14].

3.2.3 *P1788 - Interval Standard*

IEEE P1788

Currently, a standardization process for IA takes place in the IEEE P1788 working group with the goal to create an interval standard similar to IEEE 754-2008 for FP arithmetic. Because a final version is not available at the moment, the following paragraph outlines only parts important for UNIVERMEC. It is based on the draft version 8.4 [P17].

IEEE P1788 layers

In Fig. 8 the four layers (or levels) of IEEE P1788 are shown. They are similar to the layers in the IEEE 754-2008 standard. The bottom layer defines the mathematical theory on which the standard is based, that is, the real interval $\mathbb{IR}$ system over real numbers system $\mathbb{R}$ and corresponding arithmetic operations[3]. At the second layer, the *interval datums* are introduced approximating the mathematical objects from the first layer by a finite set. Every interval datum is a real interval, but a real interval can be in general only represented with the help of the interval hull operation on the interval datum layer. The third layer describes the representation of an interval datum, for example, by two FP numbers. Finally, intervals are encoded as bit strings on the uppermost layer, with no further requirements.

Set of elementary functions

In the scope of UNIVERMEC, the set of standard operations and elementary functions required by IEEE P1788 [P17; WK09] is of a special importance[4]. The text of the standard specifies not only a set of elementary functions but also their definition domains and ranges. This is important for functions, for example, the power function, for which different definitions and thus different definition domains and function ranges make sense depending on the concrete application context[5]. The heterogeneous algebra concept from Sect. 3.5 is based

3 The IEEE P1788 standard draft allows for certain *flavors* of intervals [P17, pp. 17-18]. We restrict our discussion to the set–based flavor [P17, pp. 27-68] which is used throughout this work.

4 This part of the text of the standard has already passed the voting in the IEEE P1788 group.

5 See [HNW13] for a discussion of possible variants of the interval power function.

on the specifications of the IEEE P1788 standard. This concept is used to provide a uniform set of elementary functions for all range arithmetics supported by UniVerMeC and to ensure interoperability between them.

Midpoint and radius

An important aspect discussed in the scope of IEEE P1788 is the representation of intervals in the midpoint-radius form instead of the endpoint-form. The motion [VH12] included into the current standard draft [P17, p. 60], defines the midpoint and the radius of an interval to ensure that a verified conversion to the midpoint-radius form is easily possible. While we limit our discussion to the endpoint representation of intervals in the scope of this thesis, we use the definitions from IEEE P1788 for the midpoint and the radius in the remainder of this work:

Definition 3 (Midpoint [P17; VH12]) *For an interval* $\mathbf{x} \in \mathbb{IF}$, $\operatorname{mid}(\mathbf{x})$ *is the mathematical midpoint* $\frac{\underline{\mathbf{x}}+\overline{\mathbf{x}}}{2}$ *rounded to nearest, ties to even. The midpoints of* $(-\infty, \infty)$, $(-\infty, a]$, *and* $[a, \infty]$ *for* $a \in \mathbb{F}$ *are defined as* o, *the smallest representable FP number, and the biggest representable FP number respectively.*

Definition 4 (Radius [P17; VH12]) *For an interval* $\mathbf{x} \in \overline{\mathbb{IF}}$ *the radius* $\operatorname{rad}(\mathbf{x})$ *is the smallest number in* $\overline{\mathbb{F}}$ *so that* $\mathbf{x}$ *is contained in* $[\operatorname{mid}(\mathbf{x}) - \operatorname{rad}(\mathbf{x}), \operatorname{mid}(\mathbf{x}) + \operatorname{rad}(\mathbf{x})]$.

Decorations

Another interesting concept in IEEE P1788 is the exception handling realized by *decorated intervals*. A decorated interval consists of an interval and a decoration. The decoration is mainly intended for tracking properties of inductively defined functions through their computational graphs. Formally, a decoration in the context of the set-based flavor[6] describes the properties of a pair $(f, \mathbf{x})$ where $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}$ and $\mathbf{x} \in \overline{\mathbb{IR}}^n$ [P17, p. 40]. For the set–based flavor considered in this thesis, the decorations in Tab. 3 are supported. The ordering of the decorations is quality based. Following IEEE P1788 we can say that "decorations above *trv* are 'good' and those below are 'bad'" [P17, p. 40]. Note that the *com* decoration implies for machine arithmetic[7] that no overflow happened [P17, p. 44]. An important advantage over global flags used in IEEE 754-2008 is that a decoration is directly attached to a datum. However, decorations require more memory and are computationally more expensive. To reduce these penalties, IEEE P1788 offers an optional computation mode with compressed intervals with less recorded information [P17, pp. 45-46]. To our knowledge, no IA library besides JInterval [NZ14] implements the decoration concept of IEEE P1788 currently. However,

6 Every IEEE P1788 conforming interval flavor has its own set of decorations and propagation rules. These must comply with the basic principles of the decoration system given in [P17, pp. 21-23]. Our discussion is limited to the decoration system of the set-based flavor [P17, pp. 40-46].

7 the second level in IEEE P1788

Table 3: Decorations supported in IEEE P1788 [P17, p. 40].

DECORATION	DEFINITION
com	"$\mathbf{x}$ is a bounded, nonempty subset of $\mathcal{D}$; f is continuous at each point of $\mathbf{x}$; and the computed interval $F(\mathbf{x})$ is bounded."
dac	"$\mathbf{x}$ is a nonempty subset of $\mathcal{D}$, and the restriction of f to $\mathbf{x}$ is continuous"
def	"$\mathbf{x}$ is a nonempty subset of $\mathcal{D}$"
trv	"always true [...]"
ill	"Not an Interval [...]"

even the decoration support of JINTERVAL is based on an older draft of IEEE P1788 . For AA, we implemented an adaption of decoration-based error handling in our library YALAA (cf. Sect. 3.3.5, [Kie12b]). It is based on a previously discussed decoration variant for intervals (e.g., [Hay10]).

3.2.4 *Implementations*

C-XSC

For IA , a wide range of ready to use libraries is available. C-XSC [HK04] is a collection of tools for scientific computations. It offers a complete IA library supporting many elementary functions, as well as (staggered) multi-precision arithmetic or IA in the complex plane. Furthermore, it offers its own vector and matrix classes and a toolbox containing algorithms, for example, for solving interval systems of equations, for computing slopes, or for global optimization.

PROFIL/BIAS

PROFIL/BIAS [Knü94] also implements vector and matrix classes besides standard IA library functionality. Its goal is to provide fast basic routines for implementing verified interval-based programs.

filib++

FILIB++ [Ler+06] is another C++ library for IA. In contrast to C-XSC or PROFIL/BIAS, it implements only the arithmetic and does not offer additional functions, such as matrix or vector classes. FILIB++ uses a template based approach that allows the user to exchange the base data type[8].

Boost.Interval

BOOST.INTERVAL [BMP06] is part of the well-known BOOST [Boo] project. Similar to FILIB++, it focuses on providing basis interval functionalities. It can be parametrized and adapted to users' needs through templates. BOOST.INTERVAL consequently follows the *policy-based design* principle [Ale01, pp. 3-21]. However, in contrast to the

8 The type used for representing the upper and lower bounds of an interval.

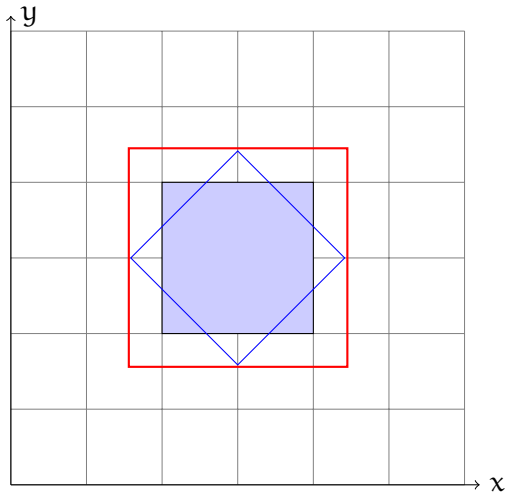


Figure 9: A geometrical representation of the wrapping effect. An interval vector (light blue) is rotated by 45 degree (blue) and is then enclosed again in an axis aligned box (red), increasing its area.

other mentioned libraries, it requires an external library for verified evaluation of elementary functions, such as sine or cosine.

3.2.5 *Overestimation*

Dependency problem

In general, the interval extension overestimates the true range of a function. Depending on the input variables' width and the expression describing a function, the overshoot can be quite large. The two main causes of overestimation are called the *dependency problem* and the *wrapping effect*. The former evolves from the fact that IA treats every occurrence of a variable as independent. Hence, the quality of an interval evaluation of a function directly depends on the expression used. Consider, for example, $f_1(x) = x^2 + 3x$ and $f_2(x) = \left(x + \frac{3}{2}\right)^2 - \frac{9}{4}$, both describing the same function but with and without multiple variable occurrences. Using the natural interval extensions, we get for $\mathbf{x} = [-3, 3]$ $f_1(\mathbf{x}) = [-9, 18]$ and $f_2(\mathbf{x}) = [-2.25, 18]$.

Wrapping effect

The other main cause of overestimation is the wrapping effect. Geometrically, it is caused by a rotation and reenclosure of the results (cf. Fig. 9) by an axis-aligned box. Lohner [Loho1] gives the following non-formal definition, which is applicable not only to IA :

> "It [the wrapping effect] is the undesirable overestimation of a solution set of an iteration or recurrence which occurs if this solution set is replaced by a superset of some 'simpler' structure and this superset is then used to compute enclosures for the next step which may eventually lead to an exponential growth of the overestimation."

Especially during iterative procedures where the effect causes overestimation in every step, it can render results useless. It occurs in many

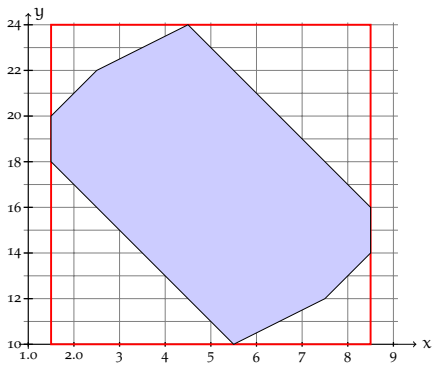


Figure 10: The joint range of the two partially dependent affine forms $5 + 0.5\epsilon_1 + 1.0\epsilon_2 + 0.0\epsilon_3 + (-2.0)\epsilon_4$ and $17 + 1.0\epsilon_1 + 1.0\epsilon_2 + 1.0\epsilon_3 + 4.0\epsilon_4$ is a zonotope (light blue). It is tighter than the corresponding interval box (red).

types of algorithms. Examples given by Lohner are matrix-vector iterations, solving of IVPs or algorithmic differentiation.

Reduction of overestimation

The dependency problem can often be weakened by the use of more sophisticated enclosure techniques, such as AA or TMs, which are described in the next section. They usually cannot cope with the wrapping effect by themselves but need to be combined with wrapping reducing techniques [Neu03], for example, coordinate transformations, zonotopes, or rearranging of algebraic expressions [Loh01].

3.3 AFFINE ARITHMETIC

Affine arithmetic

Affine arithmetic is a range arithmetic for verified numerics developed by Comba and Stolfi [CS90] with a focus towards computer graphic applications. Currently, it receives attention not only in computer graphics but also, for example, in verified global optimization or circuit design [Kie12a; Kno+09; LHB02; Mes02; MT06; NMH10].

3.3.1 *Basic Model*

Affine forms

Affine arithmetic tries to limit the overestimation during function evaluation by tracking first-order dependencies. An affine form $\hat{x} = x_0 + x_1\epsilon_1 + \ldots + x_n\epsilon_n$ consists of a central value $x_0 \in \mathbb{R}$, partial deviations $x_i \in \mathbb{R}$, and symbolic noise variables ϵ_i for $i \in \mathbb{N}^+$. It is assumed that $\epsilon_i \in [-1, 1]$ holds, but the exact value is unknown. The *fundamental invariant of AA* states that, at any point of the computation, there exists an allocation $\epsilon_i^* \in [-1, 1], i = 1, \ldots, n$, of the symbolic noise variables such that $x_0 + \sum_{i=1}^{n} x_i\epsilon_i^*$ equals the value of the exact computation [dS97, p. 44]. Two affine forms are partially linear dependent if they share at least one symbolic noise variable. Figure 10 shows that, in this case, the joint range of two affine forms is not in a box form but a center symmetric polytope.

Operations

In the following, we denote the set of all affine forms over the real numbers by $\mathbb{AR}$. For two affine forms $\hat{x}, \hat{y} \in \mathbb{AR}$ and a scalar $\alpha \in \mathbb{R}$, Comba and Stolfi define the basic affine operations addition, scaling, and translation straightforwardly:

$$\begin{aligned} \hat{x} + \hat{y} &= (x_0 + y_0) + (x_1 + y_1)\epsilon_1 + \ldots + (x_n + y_n)\epsilon_n \ , \\ \alpha\hat{x} &= \alpha x_0 + \alpha x_1 \epsilon_1 + \ldots + \alpha x_n \epsilon_n \ , \\ \alpha + \hat{x} &= (x_0 + \alpha) + x_1\epsilon_1 + \ldots + x_n\epsilon_n \ . \end{aligned} \tag{10}$$

In case of nonlinear operations, such as division or multiplication, as well as for elementary functions, such as square root, sine, cosine, it is not possible to carry out affine computations directly. The basic idea for evaluating a nonlinear function f in the affine computational model is to split it into its affine and non-affine parts. Usually, this is done by providing an affine approximation f_a. Afterwards, the nonlinear part is enclosed and a new error term $x_{n+1}\epsilon_{n+1}$ with the new independent noise variable ϵ_{n+1} is appended to the affine form. We will discuss the implementation of nonlinear elementary functions in AA in Sect. 3.3.3 in more detail.

Conversions to and from IA

Formally an interval containing the range of an affine form $\hat{x} = x_0 + \sum_{i=0}^{n} x_i\epsilon_i$ can be obtained by replacing all symbolic noise variables ϵ_i by their interval domain $[-1, 1]$:

$$\mathbf{x} = x_0 + \sum_{i=0}^{n} x_i \cdot [-1, 1] \ . \tag{11}$$

Furthermore, we can derive an affine form $\hat{x}$ enclosing an interval $\mathbf{x}$ by

$$\hat{x} = \operatorname{mid}(\mathbf{x}) + \operatorname{rad}(\mathbf{x})\epsilon_{n+1} \tag{12}$$

where ϵ_{n+1} is a symbolic noise variable that was not used before.

AA with FPA

In practice, we work with FP numbers instead of real numbers to represent the partial deviations in an affine form. In the following, we denote the set of all affine forms over the FP numbers with $\mathbb{AF}$. Here, the affine operations (10) are no longer exact but incur a rounding error, which we have to determine. De Figueiredo and Stolfi [dS97, pp. 51-53] propose to perform the operations three times (with round to nearest and to plus/minus infinity respectively) in order to obtain the exact rounding error. After that, they add it as a new independent error term to the affine form. It is also important to note that, in (12) the midpoint and radius, Def. 3, 4 have to be used to ensure proper inclusions. Due to the use of FP numbers, overflows can occur. Most implementations of affine arithmetic allow us to detect overflows during the computation by providing a special form. For example, two special forms are introduced in the error handling approach proposed in [dS97, p. 46]: The empty affine form and a form

covering the whole real line. The extended set of affine forms over the FP numbers $\overline{\mathbb{AF}}$ contains these special forms in addition to all members of the set $\mathbb{AF}$. It depends on the actual implementation used if and what special forms are available.

3.3.2 *Extended Models*

Classification of improvements

Several authors [Bil08; Kol07; Mes02; MT06; NM11] proposed improvements to the basic AA model. Most of them can be classified into three groups: better implementation techniques, changes in the model without the necessity to add higher-order noise symbols, and these with higher order noise symbols.

Better implementation techniques

An example for the first category is the a posteriori error correction approach [NM11]. Its goal is to reduce the number of rounding mode switches at the expense of precision, which usually results in a faster computation. Another example is the improved multiplication routine described in [Kol07]. These kind of improvements can be integrated into an implementation without too much effort.

Altered computation model

The second category consists of improvements that alter the computation model itself but still retain an *affine* model. Commonly known examples are the AF1 and AF2 forms introduced by Messine [Mes02]. In both models, the number of independent noise symbols is limited to the number of independent input variables, thus preventing the growth of affine forms during a lengthy computation process. They also add special terms for coping with round off or approximation errors.

Higher-order noise symbols

Finally, there are extended models that also track nonlinear dependencies in the computational graph up to a certain degree. Messine introduces in [MT06] an affine computation model with quadratic noise symbols $x_i \epsilon_i^2$. Bilotta [Bil08] considers the general case where noise symbols up to a chosen order n are used.

3.3.3 *Implementation of Elementary Functions*

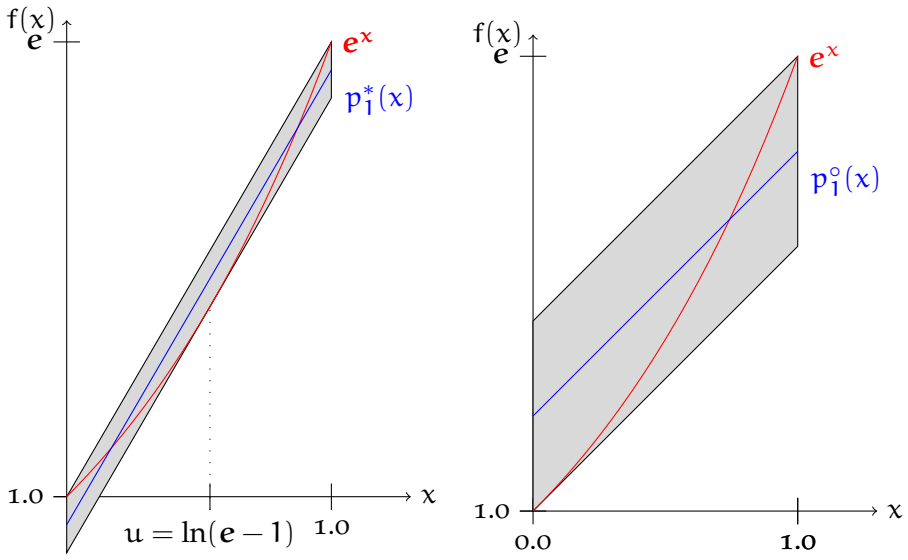
Reference implementation

The overall usability of an AA library heavily depends on the number of provided elementary functions and their quality. At the moment, most publicly available libraries orient themselves on the implementation guidelines given by de Figueiredo and Stolfi [dS97]. The authors also provide a free reference implementation [Sto].

3.3.3.1 *Convex and Concave Functions*

Implementing non-affine functions

In [dS97, pp. 55ff.;64ff.] de Figueiredo and Stolfi propose two methods for implementing a non-affine function: the best affine approximation and the min-range approximation. To simplify matters, they only consider one-dimensional affine approximations $f(\hat{x}) = \alpha\hat{x} + \zeta \pm \delta$, where



(a) Best approximation $p_1^*(x)$ (b) Min-range $p_1^\circ(x)$

Figure 11: Affine approximations for e^x over $[0, 1]$.

δ is the approximation error. However, the affine form $\hat{x}$ is a first-order polynomial in its n symbolic noise symbols $\epsilon_1, \cdots, \epsilon_n$. That is, the authors ignore that $\hat{x}$ itself depends on n independent variables. De Figueiredo and Stolfi that this does not impose a limitation for one-dimensional elementary functions, but they do not give a proof of this.

Best polynomial approximation

The best polynomial approximation $p_n^*(x)$ of degree n to a function $f(x) \in \mathcal{C}^0[a, b]$ is uniquely defined by the equioscillation property [MH03, Theorem 3.4]. It follows from the property that the approximation error is maximal in (at least) $n + 2$ points where it attains the absolute value of $\|f(x) - p_n(x)\|_\infty$. In general, $p_n^*(x)$ cannot be determined analytically but with iterative procedures, such as the Remez algorithm [Mul06, pp. 41-46]. However, a polynomial of degree one is sufficient for AA . De Figueiredo and Stolfi developed a procedure for directly determining $p_1^*(x)$.

Theorem 1 (Best affine approximation [dS97]) *Let $f(x) \in \mathcal{C}^2[a, b]$ be a bounded and convex or concave function. Then $p_1^*(x) = \alpha x + \zeta$ is the best degree one polynomial approximation over $[a, b]$ with*

$$\alpha \coloneqq \frac{f(b) - f(a)}{b - a} \,,$$
$$\zeta \coloneqq \frac{f(u) + f(a) - \alpha(a + u)}{2} \,,$$
$$\delta \coloneqq \left| \frac{f(u) - f(a) - \alpha(u - a)}{2} \right| \,,$$

where u is defined through $f'(u) = \alpha$ and $\delta \geqslant \|f(x) - p_1^*(x)\|_\infty$.

The best-affine approximation minimizes the error term δ and thus preserves as much affine dependencies as possible. Geometrically, it can be visualized as the parallelogram with the minimum area (cf. Fig. 11a). The width on the x-axis is fixed by $\mathrm{wid}\,[a, b]$ and the width on the y-axis is minimized as it is determined by δ.

Min-range approximation

Despite its good properties in preserving affine dependencies, $p_1^*(x)$ overestimates the range. This is visible in Fig. 11a where $p_1^*(x)$ for the exponential function e^x over the domain $[0, 1]$ is shown. The minimum value of e^x is $y = 1.0$ at $x = 0$. However, the minimum value covered by the parallelogram is $y \approx 0.788133$. To reduce the overestimation, de Figueiredo and Stolfi propose another approximation method, which they call *min-range affine approximation*.

Theorem 2 (Min-range affine approximation [dS97]) *Let a convex and bounded function $f(x) \in \mathcal{C}^2[a, b]$ be given. Then $p_1^\circ(x) = \alpha x + \zeta$ is the min-range approximation over $[a, b]$ with*

$$\alpha \coloneqq f'(a) \ ,$$
$$\zeta \coloneqq \frac{f(b) + f(a) - \alpha(a - b)}{2} \ ,$$
$$\delta \coloneqq \left| \frac{f(b) + \alpha(a - b) - f(a)}{2} \right| \ ,$$

where δ is the approximation error. The min-range approximation for a concave function is derived by swapping a and b in the formulas.

As shown in Fig. 11b, the min-range approximation yields a tighter range, in fact the exact range, for the example at the expense of preserving fewer affine dependencies. This results in an increased error.

FP implementation

De Figueiredo and Stolfi also discuss the FP implementation of both techniques. Since $p_1^*(x)$ cannot be determined exactly in finite arithmetic, and, thus, the equioscillation property does not hold, it turns out that the min-range approach needs fewer evaluations of the function to be approximated. This is an important advantage, because the evaluation usually needs to be carried out using computationally expensive IA [9]. In the asymptotic case, both approaches have the quadratic approximation property.

3.3.3.2 *Non-Convex and Non-Concave Functions*

Approximating non-convex functions

The restriction to strictly convex or concave functions prevents implementation of several important standard functions. Consequently, the reference implementation of AA LIBAA [Sto] does not contain functions, such as the sine or the cosine. LIBAFFA [GCH] provides them, but the implementation is not fully verified. Because all arithmetics

9 IEEE 754-2008 does not specify the accuracy of elementary functions (aside from the square root). Usually, neither the results delivered by standard math libraries are correctly rounded, nor rounding modes are respected.

supported in UniVerMeC should offer a common set of verified elementary standard functions, we developed the new AA library YalAA (**Y**et **A**nother **L**ibrary for **A**ffine **A**rithmetic) [Kie12b] to overcome these shortcomings. It offers verified support for non-convex/concave standard functions through the use of Chebyshev interpolation. It can be shown [MH03, Sect. 6.5] that the polynomial approximation obtained in this way is *near-best*. Therefore, the geometry is similar to the best-approximation p_1^* in Fig. 11a.

Chebyshev interpolation

The Chebyshev nodes x_k are defined as

$$x_k = \cos\left(\frac{\pi(2k+1)}{2n+2}\right) \ , \qquad k = 0, \ldots, n \ ,$$

and are the roots of the Chebyshev polynomials

$$T_i(x) = \cos i\theta \ , \qquad \text{if } x = \cos\theta \ ,$$

see [MH03]. A function $f : [-1, 1] \to \mathbb{R}$ can be approximated using the n-th degree Chebyshev interpolant

$$p_n^c(x) = \frac{c_0}{2} + \sum_{k=1}^{n} c_k T_k(x)$$

with the Chebyshev coefficients[10]

$$c_i = \frac{2}{n+1} \sum_{k=0}^{n} f(x_k) T_i(x_k) \ .$$

For approximating a function on a general finite interval $[a, b]$, a linear transformation to $[-1, 1]$ is necessary. The new Chebyshev nodes are obtained through the inverse transformation as

$$x'_k = \frac{1}{2}\left((b-a)x_k + a + b\right)$$

so that the coefficients are equal to

$$c'_i = \frac{2}{n+1} \sum_{k=0}^{n} f(x'_k) T_i(x_k) \ .$$

The new interpolant is given as

$$p_n^c(x) = \frac{c_0}{2} + \sum_{k=0}^{n} c'_k T_k(x)$$

10 Using the Chebyshev nodes as interpolation points ensures under mild assumptions (Dini-Lipschitz condition) uniform convergence of the series (see [MH03, Sect. 6.1] for details).

for $x \in [-1, 1]$. We can transform any $x' \in [a, b]$ using a linear transformation $t : [a, b] \to [-1, 1]$ with

$$t(x') = \left(\frac{2x' - (a + b)}{b - a} \right) \quad .$$

Therefore, the final polynomial for x' is

$$p_n^c(x') = \frac{c_0}{2} + \sum_{k=0}^{n} c_k' T_k \left(\frac{2x' - (a + b)}{b - a} \right) \quad .$$

Error bounds

We want to compute an affine approximation of the form $p_1^c(x) = \alpha\hat{x} + \zeta$ with an error bound δ over the domain $\boldsymbol{x} = [a, b]$ of $\hat{x}$ in finite precision. This is a polynomial of degree one, so that we have to compute only the coefficients c_0' and c_1'. We calculate enclosures

$$\boldsymbol{\alpha} = \frac{2c_1'}{b - a}$$

and

$$\boldsymbol{\zeta} = \frac{c_0'}{2} - \frac{a + b}{b - a}$$

for α and ζ in IA . After that, we compute the midpoints of $\boldsymbol{\alpha}, \boldsymbol{\zeta}$ and shift the rounding error into $\delta = \delta_1 + \delta_2$ according to

$$\delta_1 = \frac{1}{2} \left(\operatorname{len}(\hat{x}) \operatorname{wid} \boldsymbol{\alpha} + \operatorname{wid} \boldsymbol{\zeta} \right) \quad .$$

Here, $\operatorname{len}(\hat{x})$ denotes the number of noise symbols in $\hat{x}$. A bound for the approximation errors can be derived using Lagrange's remainder formula:

$$|\delta_2| = \left| \frac{R(\boldsymbol{x})}{2} \right| = \frac{1}{2} \left| \frac{(\operatorname{wid} \boldsymbol{x})^2 f''(\boldsymbol{x})}{16} \right| \quad ,$$

if the second derivative f'' is available. Since the central value x_0 is always the midpoint of the interval $[a, b]$ enclosing the affine form, the linear transformation t will always map x_0 to zero in exact arithmetic. Therefore, instead of the direct affine transformation $\hat{y} = \alpha\hat{x} + \zeta$, we use the following formula:

$$\begin{aligned} y_0 &= \zeta \\ y_i &= \alpha x_i' \end{aligned}$$

Alternatives to Lagrange's remainder

In order to bound $R(\boldsymbol{x})$, we use the natural interval extension of $f''(\boldsymbol{x})$. Because of the dependency effect, we sometimes have to exploit other features of f'', such as monotonicity, in order to get sharp bounds on $R(\boldsymbol{x})$ (e.g., for the inverse tangents). Another problem oc-

curs if a function is not two times differentiable over its domain. In this case, we cannot use the Lagrange remainder formula. In YalAA, this is only the case for the inverse sine and cosine functions. Both functions have the domain $[-1, 1]$, and their respective second derivatives $\pm\frac{x}{(1-x^2)^{3/2}}$ are undefined at the endpoints. For both functions, we calculate

$$e(x) = \|f(x) - p_1(x)\|_\infty \quad ,$$

that is, the exact error of our approximation. The maximum error can occur either at the endpoints of $[a, b]$ or at a local maximum. The local maximum is x^* such that $f'(x^*) - p_1'(x^*) = 0$. If we solve this equation, we get $x_{1|2} = \pm\sqrt{1 - \frac{1}{c_1^2}}$ as candidates for the maximum. We have to evaluate $e(x)$ at all 4 points a, b, x_1, x_2 with IA in order to derive a verified upper bound. This technique is much more expensive than the ordinary Lagrange remainder, hence we only apply it near the endpoints, where the derivatives behave poorly.

3.3.4 *Implementations*

Publicly available implementations

Several libraries for AA are publicly available. Examples are the reference implementation LIBAA [Sto] (written in pure C) or LIBAFFA [GCH], AAFLIB [Han] and, YalAA [Kie12b] (C++ object-oriented implementations). YalAA was developed in the scope of this thesis.

Libaa

LIBAA covers only the standard affine model and provides a purely C application programming interface (API), which is not well suited for integration into an object-oriented C++ environment. Since it is exclusively based on the implementation guidelines given by de Figueiredo and Stolfi [dS97], it is limited to strictly convex or concave elementary functions, which excludes important standard functions such as sine or cosine. For elementary functions, the implementation is not always fully verified, as it uses `fesetround` to set directed rounding modes for evaluating them through the standard FP environment. However, the rounding modes are not considered by the implementations of standard functions in general[11].

Libaffa and aaflib

Both LIBAFFA and its fork AAFLIB are written in C++ and provide object-oriented interfaces. In contrast to LIBAA, they not only offer convex and concave standard functions but also provide implementations for functions, such as sine or cosine. However, both libraries do not always deliver fully verified results[12]. The libraries implement the standard affine computation model.

YalAA

To overcome these shortcoming of currently available implementa-

11 Actually, the behavior is implementation-defined according to the C11 standard [C11a, §F.10].

12 For example, LIBAFFA does not perform any rounding modes changes or a posteriori error correction for the exponentiation function.

tions, we developed YalAA. It offers a verified common set of elementary functions oriented with IEEE P1788 as a general guideline. The implementation recommendations of de Figueiredo and Stolfi were used where possible. Otherwise, we applied our the Chebyshev interpolation approach. A further goal was to create a sufficiently flexible library allowing for support of extended affine computation models. Section 3.3.5 describes the design of YalAA in more detail.

Support in UniVerMeC

Currently, UniVerMeC offers a support for libaffa, which could be used as a reference for integrating aaflib in a straightforward way. However, neither of them provide completely verified results. Libaa is not supported because the non-object-oriented approach is difficult to incorporate in a purely object-oriented environment. Additionally, UniVerMeC supports YalAA. This library will be used in the rest of this work because of the above mentioned limitations of the other libraries (cf. also Tab. 4).

3.3.5 *Architecture of* YalAA

Policy-based design

YalAA makes heavy use of template metaprogramming techniques, for example, of *policy* and *trait classes* [Ale01; Mye95]. They allow for flexible adaption of the library to changing requirements. A trait class is a class template offering type definitions, constants, and static methods. Usually, it needs to be specialized for different data types and offers, in this way, a uniform interface for them. In YalAA, it is mainly used as a compile-time replacement for the adapter pattern. A policy class encapsulates a part of the functionality of a system and makes it available through a uniform interface. It can be interpreted as a static compile-time variant of the well-known strategy pattern. As the functionalities chosen are normally orthogonal to each other, it is possible to use arbitrary combinations of policy classes. Thus, providing a wide range of functionality in a library without implementing each feature combination separately. This policy-based design technique can be seen as a generator in the sense of generative programming and, thus, weakens the library scaling problem, which describes the increased amount of work necessary if every possible combination of features has to be implemented manually [CEo0, p. 333].

Architecture of YalAA

The architecture of YalAA is shown in Fig. 12. The library is structured around the template class `AffineForm`, which represents an affine form, and offers the public interface to the user. This class template is parametrized by four policy classes and, additionally, by the base type `T` and the interval type `IV`. `IV` should represent intervals in the endpoint form of the base type `T`. While `T` is the type for partial deviations (usually `double`, but other types like exact rational numbers are possible), the latter determines the underlying interval type. Since YalAA uses a trait class to access it, any existing interval library

Table 4: Overview over AA libraries. The listed elementary functions are oriented on IEEE P1788 .

	LIBAA	LIBAFFA	AAFLIB	YALAA
Prog. Language	C	C++	C++	C++
Affine Models	AF0	AF0	AF0	AF0, AF1, AF2
Verified	✓	×	×	✓
Supported Functions				
sqr	✓	✓	✓	✓
pown	×	✓	✓	✓
pow	×	✓	✓	✓
sqrt	✓	✓	✓	✓
exp	✓	✓	✓	✓
log	×	✓	✓	✓
sin	×	✓	✓	✓
cos	×	✓	✓	✓
tan	×	✓	✓	✓
asin	×	×	×	✓
acos	×	×	×	✓
atan	×	×	✓	✓
atan2	×	×	×	×
sinh	×	✓	×	✓
cosh	×	✓	×	✓
tanh	×	✓	✓	✓
asinh	×	✓	×	✓
acosh	×	✓	×	✓
atanh	×	✓	×	✓
abs	✓	✓	×	✓
min	✓	×	×	×
max	✓	×	×	×

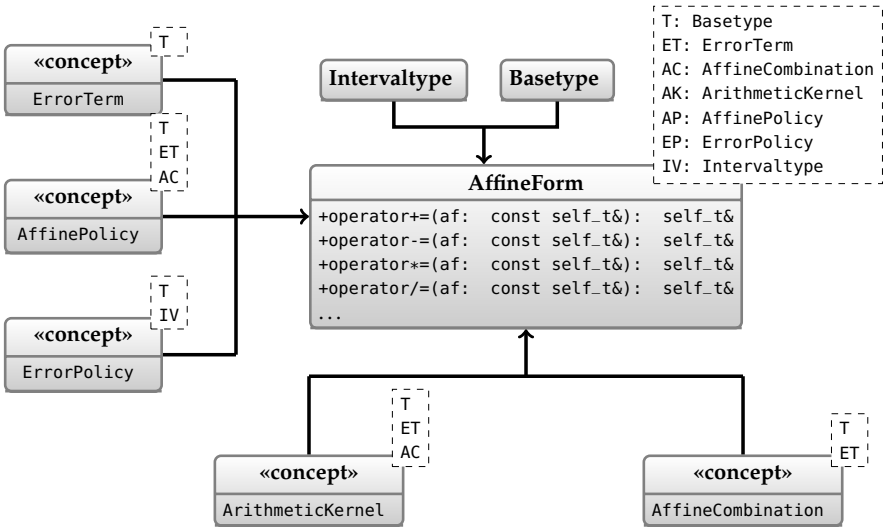


Figure 12: Basic architecture of YALAA

can be integrated seamlessly. Furthermore, users can perform mixed computations between affine forms and types `T` and `IV` in the basic operations. This is important because AA is often employed inside existing IA environments.

ErrorTerm

The policy class `ErrorTerm` represents an error term $x \cdot \epsilon$. It provides an ordering on the error terms and is responsible for generating new noise variables. The latter operation has to be thread-safe if the library is used in a multi-threaded environment. Furthermore, the class stores the unique identifier for noise symbols and so limits their maximum number. Such operations are possibly performance critical. Therefore, users can adapt them to their current needs by changing the policy.

AffineCombination

The `AffineCombination` policy class is responsible for storing an affine combination consisting of the central value and the error terms. Further, it provides the affine operations addition, scaling and translation. Any operation performed in YALAA is broken down into these. Note that no rounding control, error correction, or handling is necessary at this particular level because those are only the plain basic operations.

ArithmeticKernel

The core component of YALAA is the `ArithmeticKernel` policy. It provides an implementation of all supported operations and elementary functions and performs the affine part of them. Furthermore, it calculates bounds on the approximation and rounding error and tracks exceptional situations occurring during the computation, for example, domain violations or overflows. This information is propagated through the `ArithmeticError` class, which acts as a uniform interface between possibly different implementations of `ArithmeticKernel` and the rest of YALAA. Since the verified implementation of an operation depends heavily on the underlying type, `ArithmeticKernel` has to be specialized for each base type `T`.

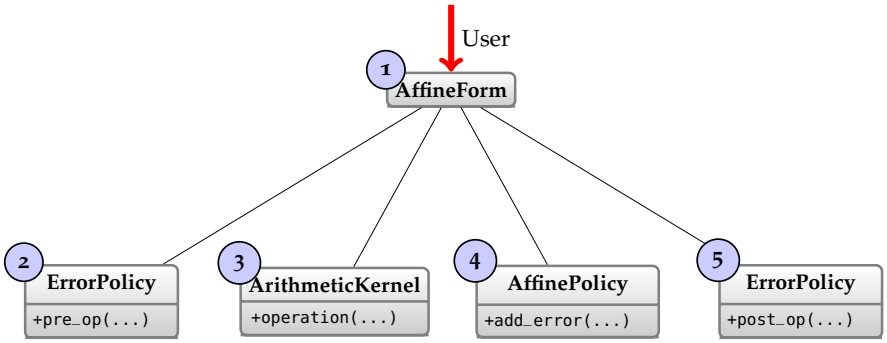


Figure 13: Interaction of YALAA's policy classes

AffinePolicy

The `AffinePolicy` class is responsible for handling rounding and approximation errors, creating new affine forms, and introducing uncertainty into the computation. Utilizing these three methods, we can implement the usual affine model and the extended AF1 and AF2 models just by exchanging the policy class[13].

ErrorPolicy

The `ErrorPolicy` is responsible for providing the special value type `special_t` for affine forms, for example, the empty set or the whole real line. Moreover, it handles information about exceptional situations passed on by the `ArithmeticKernel`. Using the `ErrorPolicy` approach, we can implement different concepts for error handling, for example, the common error handling for AA described in [dS97, p. 46] or even a decoration-like approach currently being discussed for intervals in the IEEE P1788 standardization group.

Process of a function evaluation

The complete process for evaluating an elementary function or a basic operation on an affine form is outlined in Fig. 13. In the first step, the user calls an operation. Then a check for special affine forms in the input arguments is performed by `ErrorPolicy::pre_op`. If none are present, the actual operation is called in the `ArithmeticKernel` class. The `AffinePolicy` handles the rounding and approximation errors. In the last step, `ErrorPolicy::post_op` checks if any errors occured during the operation.

3.4 TAYLOR MODELS

Taylor forms

Taylor models (TMs) are a technique initially introduced by Berz and his group [Ber95b; MB03]. Similar ideas have been discussed before under various names. They can be combined together with TMs to the family of *Taylor forms*. Neumaier [Neu03] gives a throughout discussion of their history. The TM technique has been applied in various contexts with a focus towards, but not limited to, verified solving of IVPs. Our discussion in the remainder of this section follows the work [Ebl06].

13 The special noise symbols introduced by AF1/AF2 also need support at the level of the basic operations, which the `AffineCombination` policy class provides.

3.4.1 *Basic Model*

Taylor model

The idea behind TMs is to compute the range of a function by exploiting Taylor's theorem. Based on it, a polynomial approximation and an interval bound on the error are constructed.

Definition 5 (Taylor model) *Let $P_{f,n}$ be the n-th order Taylor polynomial of a function $f \in \mathcal{C}^{n+1}(\mathcal{D}), \mathcal{D} \subset \mathbb{R}^m$ and $\mathbf{x} \in \mathbb{IR}^m$ with $\mathbf{x} \subset \mathcal{D}$. An interval $\mathbf{i}^{(f,n)}$ is a n-th order remainder bound if $\forall x \in \mathbf{x} : f(x) - P_{f,n}(x) \in \mathbf{i}^{(f,n)}$ holds. The pair $\left(P_{f,n}, \mathbf{i}^{(f,n)}\right)$ is called a Taylor model and $\mathbf{x}$ is called its domain.*

In the remainder of this thesis, we denote the set of TMs over the real numbers by $\mathbb{TR}$. The operator $\mathbb{TR}(\mathcal{D})$ returns all TMs over $\mathcal{D} \subset \mathbb{R}$.

Basic operations

An arithmetic for this model has been developed by Berz's group. For two TMs $\check{x}_1 = \left(P_{f_1,n}, \mathbf{i}^{(f_1,n)}\right)$, and $\check{x}_2 = \left(P_{f_2,n}, \mathbf{i}^{(f_2,n)}\right)$ we can define the addition using

$$\check{x}_1 + \check{x}_2 = \left(P_{f_1+f_2,n}, \mathbf{i}^{(f_1+f_2,n)}\right), \quad \text{with} \quad P_{f_1+f_2,n} = P_{f_1,n} + P_{f_2,n} \ , \quad \text{and} \quad \mathbf{i}^{(f_1+f_2,n)} = \mathbf{i}^{(f_1,n)} + \mathbf{i}^{(f_2,n)}.$$

Multiplication of two polynomials of degree n results in a polynomial of the degree $2n$ at most, which is split into two parts with degrees $\leqslant n$ and $> n$, respectively:

$$P_{f_1,n} \cdot P_{f_2,n} = P_{f_1 f_2,\leqslant n} + P_{f_1 f_2,>n}$$

The resulting TM consists of the polynomial $P_{f_1 f_2,\leqslant n}$ and a remainder bound that also encloses the codomain $W(P_{f_1 f_2,>n})$ of $P_{f_1 f_2,>n}$:

$$\begin{aligned} \check{x}_1 \cdot \check{x}_2 = \left(P_{f_1 f_2,n}, \mathbf{i}^{(f_1 f_2,n)}\right) \ , \quad \text{with} \quad P_{f_1 f_2,n} &= P_{f_1 f_2,\leqslant n} \ , \\ \text{and} \quad \mathbf{i}_{f_1 f_2,n} &= W(P_{f_1 f_2,>n}) + \\ & \quad W(P_{f_1,n})\mathbf{i}^{(f_2,n)} + \\ & \quad W(P_{f_2,n})\mathbf{i}^{(f_1,n)} + \\ & \quad \mathbf{i}^{(f_1,n)}\mathbf{i}^{(f_2,n)} \end{aligned}$$

Definitions for division and elementary functions can be found in [Mak98]. An important advantage of the Taylor basis is that, for an operation $\circ$ or elementary function e, the polynomial part P' of $\check{x}_1 \circ \check{x}_2$ or $e(\check{x}_1)$ equals the truncated Taylor series expansion for this operation or elementary function.

Conversions from and to intervals

An interval $\mathbf{x} \in \mathbb{IR}$ can be converted to a TM by

$$\check{x} = (\text{mid}\,(x), \text{rad}\,(\mathbf{x})) \quad . \tag{13}$$

To determine the result of a computation in a practically comprehensive form, it is necessary to compute an interval bounding the range

of a TM. Here, the main challenge is to bound the range of the polynomial part. Different approaches are available. They range from simple interval evaluation or the use of the Horner scheme to techniques developed especially for TMs, for example, the *linear dominated bounder (LDB)* or the *quadratic fast bounder (QFB)* [MB05b] algorithms.

FP implementation

Analogous to IA and AA , the set of all TMs over the FP numbers is denoted as $\mathbb{TF}$ and the extended set of all TMs $\overline{\mathbb{TF}}$. Rigorous enclosures in the case of FPA and rounding errors are described, for example, in [Ebl06, pp. 80-87 104-111]. In finite arithmetic, the midpoint and radius operations in (13) have to comply with Def. 3 and 4.

3.4.2 *Implementations*

COSY

The reference implementation for TMs is COSY [BM06], which was developed by Berz and his group. It is written in FORTRAN but offers an interface for access from C/C++ programs. A posteriori error correction approach described in [RMB05] ensures that COSY is relatively fast despite the high number of basic FP operations that the Taylor arithmetic requires. The COSY implementation requires a special license. COSY uses its own memory management system, which makes employment in an environment like UNIVERMEC difficult.

Riot

RIOT is a C++ object-oriented implementation of the TM approach. In contrast to COSY, it is available as free software and offers a memory management that can be employed in a more straightforward way in object-oriented environments. However, RIOT is slower compared to COSY and does not provide newer algorithms such as the QFB. The number of independent TMs in a computation is limited to eight.

3.5 ABSTRACT ALGEBRA AND HIERARCHY

Building blocks for functions

An important goal for UNIVERMEC is to provide interoperable access to (at least) the arithmetics described above. That is, it should support a uniform way of specifying functions, so that they can be evaluated with every available arithmetic. The most functions we consider can be described by a finite composition of certain building blocks, for example, basic operations and elementary functions. This section specifies these building blocks more formally as an algebra for all arithmetics. Furthermore, the interoperability between the supported arithmetics is discussed.

3.5.1 *Universal Inclusion Representation*

Combination of different arithmetics

In general, the use of various inclusion arithmetics simultaneously to improve the tightness of enclosures of functions is not new. For example, de Figueiredo and Stolfi [dS97, pp. 75f.] propose combined use of interval and affine arithmetic. Consequently, their reference imple-

Table 5: Representation of arithmetics supported by UNIVERMEC in the framework of Vu, Sam-Haroud, and Faltings [VSHF09].

	T	V_T	D_T	$f_T(V_T)$
Reals	(x)	(x)	$[x, x]$	$id(x)$
IA	(a, b)	(x)	$[a, b]$	$id(x)$
AA	$(x_0, x_1, \ldots, x_n)$	$(\epsilon_1, \ldots, \epsilon_n)$	$[-1, 1]^n$	$x_0 + \sum_{i=1}^{n} x_i \epsilon_i$
TMs	$(a_1, \ldots, a_n, e_1, e_2)$	(x, e)	$[-1, 1]^n$	$e + \sum_{i=1}^{n} a_i x^i$

mentation of AA supports a hybrid computation mode that performs every operation twice with interval and affine arithmetic and allows for intersecting the results.

Inclusion representation

More recently, Vu, Sam-Haroud, and Faltings [VSHF09] proposed the use of different inclusions techniques in the scope of numerical constraint propagation. For this purpose, they introduce the abstract notion of an inclusion representation.

Definition 6 (Inclusion representation [VSHF09]) *Let $\mu : \mathcal{R} \to 2^{\mathcal{A}}$ be a function and $\mathcal{R}$ a non-empty set. The pair $(\mathcal{R}, \mu)$ is an inclusion representation for a set $\mathcal{A}$ if there exists a function $\zeta : 2^{\mathcal{A}} \to \mathcal{R}$ so that*

$$\begin{aligned} &(i) \quad \mu(\zeta(\emptyset)) &&= \emptyset \\ &(ii) \quad \forall S \subseteq \mathcal{A} &&: \; S \subseteq \mu(\zeta(S)) \end{aligned}$$

holds.

This rather abstract definition is specialized by the authors for the class of real representations. In this case, $\mathcal{R}$ contains tuples of real numbers and for $\forall T \in \mathcal{R}$

$$\mu(T) = \{f_T(V_T) | V_T \in D_T[V_T]\}$$

holds. V_T is a sequence of variables, D_T the sequence of associated domains, and $f_T : D_T \to \mathbb{R}$ a function. It is shown by the authors how to represent various inclusion techniques in their framework; see Tab. 5. Although they give no advice on how to integrate TMs into their framework, they can be represented as shown in the table for a degree n TM of one variable over the canonical interval[14] $[-1, 1]$ with the remainder bound e. The representation could be easily extended to involve m variables. All considered arithmetics in UNIVERMEC are represented according to this inclusion framework.

Inclusion functions

Based on this formal definition, the authors derive a generalized notion for an inclusion function (which is characterized e.g. in the interval case by (6)).

14 Most existing software for TMs uses a linear transformation to ensure that the polynomial part of the model lies inside this interval.

Definition 7 (Inclusion function [VSHF09]) *A function* $F : \mathcal{R}_X \to \mathcal{R}_Y$ *is called an inclusion function for* $f : X \to Y$ *if*

$$(S \subseteq \mu_X(T)) \Rightarrow (\{f(x)|x \in S\} \subseteq \mu_Y(F(T)))$$

holds for all $S \subseteq X$ *and* $\forall T \in \mathcal{R}_X$. *Here,* $(\mathcal{R}_X, \mu_X), (\mathcal{R}_Y, \mu_Y)$ *are inclusion representations for* X *and* Y.

Among the considered arithmetics the above definition cannot be used for computations in finite-precision without range-arithmetics. For simplicity, we will nevertheless use the term *inclusion function* with regard to all base sets (including $\mathbb{F}, \overline{\mathbb{F}}$). Based on the general notion of an inclusion function, it is now possible to derive straightforward generalizations of the natural interval extension (Def. 2) for all supported arithmetics.

Definition 8 (Generalized natural extension [VSHF09]) *A function* $F : \mathcal{D} \subseteq \mathcal{R}^n \to \mathcal{R}^m$ *is called the natural extension of* $f : \mathcal{D} \subseteq \mathbb{R}^n \to \mathbb{R}^m$ *in the inclusion representation* $(\mathcal{R}, \mu)$, *if it is constructed by replacing all variables, basic operations, constants, and elementary functions in* f *by their respective counterparts in* $(\mathcal{R}, \mu)$.

In the above definition it is assumed that the counterparts of the elementary functions themselves comply to Def. 7. Although this framework supplies formal definitions for inclusion representations and functions, a definition of the basic building blocks, that is, the set of operations and elementary functions, is still missing. They are defined in the following.

3.5.2 *Heterogeneous Algebra*

Base set of functions

The goal of this section is to specify a set of operations and elementary functions that is available for every supported arithmetic. Furthermore, it defines what functions and operations are allowed for a certain kind of arithmetic as well as what combinations between different arithmetics are possible. The upcoming standard IEEE P1788 acts as a basis for the set of standard functions, their definitions, and domains. Following the interval standard, we first define a set of point functions and then extend them to the different arithmetics that UniVerMeC supports using the notation of generalized inclusion functions according to Def. 7.

Heteregenous real algebra

Formally, we define the set of basic operations and elementary functions by specifying a *heterogeneous algebra*. We use the algebra notion of Birkhoff and Lipson [BL70], who define it as a 2-tuple $(\mathcal{S}, \mathcal{F})$ where "$\mathcal{S}$ is a family of non-void sets S_i of different types of elements" and "$\mathcal{F} = \{f_\alpha\}$ is a set of finitary operations". In our case, the family $\mathcal{S}$ consists of the arithmetic base sets mentioned earlier, that is, $\mathcal{S} := \{\mathbb{R}, \mathbb{IR}, \mathbb{AR}, \mathbb{TR}\}$. Our algebra supports the basic operations

Table 6: Basic operations and elementary functions considered in the heterogeneous algebra in UniVerMeC. The set of functions and their domains are taken from the IEEE P1788 standard [P17, p. 24].

OPERATION / FUNC. e	DEFINITION	DOMAIN $\mathcal{D}(e)$
$x \circ y$ with $\circ \in \{+, -, \cdot\}$		$\mathbb{R} \times \mathbb{R}$
x/y		$\mathbb{R} \times \mathbb{R} \setminus \{0\}$
$\mathrm{abs}(x)$		$\mathbb{R}$
$\mathrm{acos}(x)$		$[-1, 1]$
$\mathrm{acosh}(x)$		$[1, \infty)$
$\mathrm{asin}(x)$		$[-1, 1]$
$\mathrm{asinh}(x)$		$\mathbb{R}$
$\mathrm{atan}(x)$		$\mathbb{R}$
$\mathrm{atanh}(x)$		$\mathbb{R}$
$\mathrm{cos}(x)$		$\mathbb{R}$
$\mathrm{cosh}(x)$		$\mathbb{R}$
$\mathrm{exp}(x)$	e^x	$\mathbb{R}$
$\mathrm{log}(x)$	$\log_{10}(x)$	$\mathbb{R}^+ \setminus \{0\}$
$\mathrm{neg}(x)$	$-x$	$\mathbb{R}$
$\mathrm{pown}(x, p)$	x^p	$\mathbb{R} \times \mathbb{Z}^+ \cup \mathbb{R} \setminus \{0\} \times \mathbb{Z}^-$
$\mathrm{pow}(x, y)$	$\begin{cases} e^{y \ln(x)} & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$	$\left(\mathbb{R}^+ \setminus \{0\} \times \mathbb{R}\right) \cup \left(\{0\} \times \mathbb{R}^+ \setminus \{0\}\right)$
$\mathrm{sin}(x)$		$\mathbb{R}$
$\mathrm{sinh}(x)$		$\mathbb{R}$
$\mathrm{sqr}(x)$	x^2	$\mathbb{R}$
$\mathrm{sqrt}(x)$	$\sqrt{x}$	$\mathbb{R}^+$
$\mathrm{tan}(x)$		$\mathbb{R} \setminus \left\{(k + 0.5)\pi \mid k \in \mathbb{Z}\right\}$
$\mathrm{tanh}(x)$		$\mathbb{R}$
$\inf(\mathbf{x})$	$\underline{\mathbf{x}}$	$\overline{\mathbb{IR}}$
$\sup(\mathbf{x})$	$\overline{\mathbf{x}}$	$\overline{\mathbb{IR}}$
$\mathrm{mid}\,(\mathbf{x})$	(1)	$\mathbb{IR}$
$\mathrm{wid}\,(\mathbf{x})$	(2)	$\mathbb{IR}$
$\mathrm{rad}\,(\mathbf{x})$	$\frac{\mathrm{wid}\,\mathbf{x}}{2}$	$\mathbb{IR}$
$\mathrm{to\_iv}(\hat{x})$	(11)	$\mathbb{AR}$
$\mathrm{to\_iv}(\check{x})$	*TM → IA conversion* Sect. 3.4 (LDB, QFB, ...)	$\mathbb{TR}$

$\mathcal{F}_C \coloneqq \{+,-,\cdot,/\}$, unary elementary functions $\mathcal{F}_E \coloneqq \{\text{acos}, \text{acosh}, \text{asin}, \text{asinh}, \text{atan}, \text{atanh}, \cos, \cosh, \exp, \log, \text{neg}, \sin, \sinh, \text{sqr}, \text{sqrt}, \tan, \tanh\}$, and binary elementary functions $\mathcal{F}_B \coloneqq \{\text{pown}, \text{pow}\}$. In Tab. 6, the natural domain $\mathcal{D}^{(e)}$ associated with each individual function e is given (i.e., $e : \mathcal{D}^{(e)} \to \mathbb{R}$ for $e \in \mathcal{F}_C \cup \mathcal{F}_E \cup \mathcal{F}_B$). The first part of the set $\mathcal{F}$ of basic operations is formally defined as

$$\begin{aligned}\mathcal{F}^{(1)} \coloneqq &\left\{ e_{\mathbb{M}} : \mathbb{M}\left(\mathcal{D}^{(e)}\right) \to \mathbb{M} \;\middle|\; e \in \mathcal{F}_C \cup \mathcal{F}_E, \mathbb{M} \in \mathcal{S} \right\} \cup \\ &\left\{ \text{pow}_{\mathbb{M}} : \mathbb{M}\left(\mathcal{D}_1^{(\text{pow})}\right) \times \mathbb{M}\left(\mathcal{D}_2^{(\text{pow})}\right) \to \mathbb{M} \;\middle|\; \mathbb{M} \in \mathcal{S} \right\} \cup \\ &\left\{ \text{pown}_{\mathbb{M}} : \mathbb{M}\left(\mathcal{D}_1^{(\text{pown})}\right) \times \mathcal{D}_2 \to \mathbb{M} \;\middle|\; \mathbb{M} \in \mathcal{S} \right\}\end{aligned}$$

where $e_{\mathbb{M}}, \text{pow}_{\mathbb{M}}, \text{pown}_{\mathbb{M}}$ are inclusion functions in the sense of Def. 7; for e, pow, pown in the respective arithmetic and $\mathcal{D}_i^{(p)}$ is the i-th component of the natural domain for $p \in \{\text{pow}, \text{pown}\}$ for $i = 1, 2$. Furthermore, we want to allow the use of basic operations with mixed operand types if this does not incur any information loss. Therefore, we define the second part of the set of operations as $\mathcal{F}^{(2)} \coloneqq \mathcal{F}^{(2')} \cup \mathcal{F}^{(2'')}$ with

$$\begin{aligned}\mathcal{F}^{(2')} \coloneqq \Bigg\{ \tilde{\circ} : \mathbb{M}_1\left(\mathcal{D}_1^{(\circ)}\right) \times \mathbb{M}_2\left(\mathcal{D}_2^{(\circ)}\right) \to \mathbb{M}_1 \;\Bigg|\; \\ \circ \in \{+,-,\cdot,/\}, \mathbb{M}_1 \in \mathcal{S}, \mathbb{M}_2 \in \{\mathbb{R}, \mathbb{IR}\}, \mathbb{M}_1 \neq \mathbb{M}_2 \Bigg\} ,\end{aligned}$$

and

$$\begin{aligned}\mathcal{F}^{(2'')} \coloneqq \Bigg\{ \tilde{\circ} : \mathbb{M}_1\left(\mathcal{D}_1^{(\circ)}\right) \times \mathbb{M}_2\left(\mathcal{D}_2^{(\circ)}\right) \to \mathbb{M}_2 \;\Bigg|\; \\ \circ \in \{+,-,\cdot,/\}, \mathbb{M}_2 \in \mathcal{S}, \mathbb{M}_1 \in \{\mathbb{R}, \mathbb{IR}\}, \mathbb{M}_1 \neq \mathbb{M}_2 \Bigg\}\end{aligned}$$

where $\tilde{\circ}$ is an inclusion function in the sense of Def. 7; for $\circ \in \{+,-,\cdot,/\}$ and $\mathcal{D}_1^{(\circ)}, \mathcal{D}_2^{(\circ)}$ are the first and second component of the natural domain of the operation according to Tab. 6. The implementation is straightforward if we apply type conversion. Each $r \in \mathbb{R}$ can be treated as the point-interval $[r, r]$, and each interval x can be treated as an affine form or a TM, by applying the conversion formulas (11) or (13) respectively. In practice we sometimes need operations that are defined for some of the arithmetics. Therefore, in addition to F_C, F_E and, F_B we consider a further set $\mathcal{F}_I \coloneqq \{\text{inf}, \text{sup}, \text{rad}, \text{wid}, \text{mid}\}$ exclusively for intervals and introduce

$$\mathcal{F}^{(3)} \coloneqq \left\{ e : \mathcal{D}^{(e)} \to \mathbb{R} \;\middle|\; e \in \mathcal{F}_I \right\}$$

Table 7: Additional functions for the FP algebra to handle the special forms. All functions map into the set $\mathbb{B} = \{\text{true}, \text{false}\}$.

FUNCTION e	DEFINITION	DOMAIN $\mathcal{D}_e$
is_special(x)	true iff $x \in \overline{\mathbb{F}} \setminus \mathbb{F}$	$\overline{\mathbb{F}}$
is_empty(x)	true iff x is NaN	$\overline{\mathbb{F}}$
is_unbounded(x)	true iff $x = \pm\infty$	$\overline{\mathbb{F}}$
is_special($\mathbf{x}$)	true iff $\mathbf{x} \in \overline{\mathbb{IF}} \setminus \mathbb{IF}$	$\overline{\mathbb{IF}}$
is_empty($\mathbf{x}$)	true iff $\underline{x} > \overline{x}$	$\overline{\mathbb{IF}}$
is_unbounded($\mathbf{x}$)	true iff $\underline{x} = -\infty \vee \overline{x} = \infty$	$\overline{\mathbb{IF}}$
is_special($\hat{x}$)	true iff $\hat{x} \in \overline{\mathbb{AF}} \setminus \mathbb{AF}$	$\overline{\mathbb{AF}}$
is_empty($\hat{x}$)	*optional, implementation dependent*	$\overline{\mathbb{AF}}$
is_unbounded($\hat{x}$)	*optional, implementation dependent*	$\overline{\mathbb{AF}}$
is_special($\check{x}$)	true iff $\check{x} \in \overline{\mathbb{TF}} \setminus \mathbb{TF}$	$\overline{\mathbb{IF}}$
is_empty($\check{x}$)	*optional, implementation dependent*	$\overline{\mathbb{TF}}$
is_unbounded($\check{x}$)	*optional, implementation dependent*	$\overline{\mathbb{TF}}$

into our algebra. Furthermore, we consider an outer function to_iv (cf. Tab. 6) that maps an affine form or a TM to an interval

$$\mathcal{F}^{(4)} \coloneqq \left\{ \text{to\_iv} : \mathcal{D}^{(\text{to\_iv})} \to \mathbb{IR} \right\} \quad .$$

This leads to the final definition of our heterogeneous algebra $\mathfrak{EA}$ for real computations as:

$$\mathfrak{EA} \coloneqq \left(\mathcal{S}, \bigcup_{1 \leqslant i \leqslant 4} \mathcal{F}^{(i)} \right) \quad . \tag{14}$$

Corresponding FP algebra

The real algebra can be assigned a corresponding FP algebra which is used for the actual computations. To obtain it, we have to perform two steps: First we replace the real domain sets of all mappings by their respective FP counterparts, that is, $\mathbb{R}$ by $\mathbb{F}$, $\mathbb{IR}$ by $\mathbb{IF}$ and so on. Then we replace the range sets of the mappings by their respective *extended* FP counterparts, that is, $\mathbb{R}$ by $\overline{\mathbb{F}}$, $\mathbb{IR}$ by $\overline{\mathbb{IF}}$ and so on. This is crucial to be able to handle overflows that might occur during computations in finite precision. Note that rounding issues have to be handled carefully during the actual implementation. UNIVERMEC relies here solely on the concrete implementation of the arithmetic that is employed. To make it easier for users to handle the extended sets $\overline{\mathbb{F}}$, $\overline{\mathbb{IF}}$, $\overline{\mathbb{AF}}$, and $\overline{\mathbb{TF}}$, we extend the algebra $\mathfrak{EA}$ by the set of functions specified in Tab. 7. The function *is_special* allows users to check whether something went wrong during the computation. This function is mandatory for all arithmetics because, if a library introduces special forms to handle overflows, domain violation, and similar er-

rors, there probably exists a function to tell these forms apart from the "normal" forms. The more specialized functions *is_empty* and *is_unbounded* can be used to check whether the result is empty or unbounded (which is possibly caused by an overflow) respectively. UniVerMeC guarantees the existence of the three functions and, if possible maps them to the respective routines provided by the underlying arithmetic library. However, the actual semantics can be different in concrete implementations since they are not standardized in general. Because we do not make any assumptions about the behavior of the underlying library, it might choose, for example, to call the `exit` function or to throw an exception if an overflows occurs. In UniVerMeC, we can guarantee only that users can rely on the uniform functions in the framework if a library provides special forms. Formally, we define the set of operations

$$\mathcal{F}^{(5)} \coloneqq \left\{ e : \overline{\mathbb{M}} \to \mathbb{B} \;\middle|\; e \in \mathcal{F}_\mathsf{F}, \overline{\mathbb{M}} \in \{\overline{\mathbb{F}}, \overline{\mathbb{IF}}, \overline{\mathbb{AF}}, \overline{\mathbb{TF}}\} \right\}$$

where $\mathcal{F}_\mathsf{F} \coloneqq \{\text{is\_special}, \text{is\_empty}, \text{is\_unbounded}\}$ is defined according to Tab. 7 and $\mathbb{B} \coloneqq \{\text{true}, \text{false}\}$. That is, the FP algebra $\mathfrak{FA}$ for the finite arithmetics is defined as:

$$\mathfrak{FA} \coloneqq \{\mathcal{S} \cup \{\mathbb{B}\}, \mathcal{F} \cup \mathcal{F}^{(5)}\} \ . \tag{15}$$

Note that there are no assumptions about the behavior of functions in $\mathfrak{FA}$ if their natural domain is violated, for example, if $x \in \overline{\mathbb{IF}} \setminus \mathbb{IF}$ is used as input[15]. In this case, the outcome is implementation-defined, that is, undefined in the general case. The algebra specification given here is well founded in the upcoming IEEE P1788 standard and most existing arithmetic implementations conform to this algebra specification in general. More details on practical considerations along with respective implementations are given in Sect. 3.6.

3.5.3 *Arithmetic Hierarchy and Conversions*

Conversion operators

The algebras above implicitly define a hierarchy between the different arithmetics. It determines what guarantees we can make for conversions between them. Figure 14 shows two types of conversions inside the hierarchy for which we can make guarantees: lossless (solid lines) and enclosure preserving (dotted lines) conversions. A lossless conversion means that no information about the solution area is lost.

Definition 9 (Lossless conversion between real algebras) *Let* $\mathbb{M}_1$, $\mathbb{M}_2$ *be two support sets of the (exact) real algebra with their respective*

15 The domains of all functions are restricted to the non-extended sets $\mathbb{F}$, $\mathbb{IF}$, $\mathbb{AF}$ and $\mathbb{TF}$.

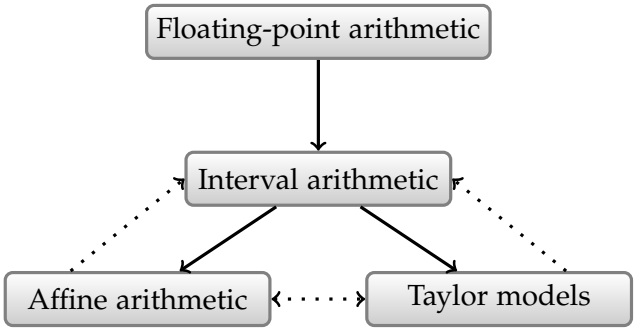


Figure 14: Arithmetic hierarchy in UniVerMeC. Lossless conversions are represented by solid lines and enclosure preserving conversions by dotted lines.

real inclusion representations $(\mathbb{M}_1, \mu_{\mathbb{M}_1})$ *and* $(\mathbb{M}_2, \mu_{\mathbb{M}_2})$. *A mapping* $\varphi : \mathbb{M}_1 \to \mathbb{M}_2$ *is called lossless conversion if*

$$\mu_{\mathbb{M}_1}(x) = \mu_{\mathbb{M}_2}(\varphi(x))$$

holds for $\forall x \in \mathbb{M}_1$.

The possibility for the lossless conversions shown in Fig. 14 follows directly from the algebra definitions and (12), (13). Applying these formulas on a computer using FPA can result in rounding errors widening the enclosed area. Therefore, we use the following definition in the case of the FP algebras.

Definition 10 (Lossless conversion between FP algebras) *Let* $\mathbb{M}_1, \mathbb{M}_2$ *be two support sets of the finite FP algebras. A mapping* $\varphi : \mathbb{M}_1 \to \mathbb{M}_2$ *is called lossless if its corresponding mapping in the respective (exact) real algebra is lossless.*

In the case of enclosure preserving conversions, the requirement is less strict. We only require that no covered area is lost.

Definition 11 (Enclosure preserving conversion) *Let* $\mathbb{M}_1, \mathbb{M}_2$ *be two support sets of either of the exact or the finite algebra with their respective real inclusion representations* $(\mathbb{M}_1, \mu_{\mathbb{M}_1})$ *and* $(\mathbb{M}_2, \mu_{\mathbb{M}_2})$. *A mapping* $\varphi : \mathbb{M}_1 \to \mathbb{M}_2$ *is called enclosure preserving conversion if*

$$\mu_{\mathbb{M}_1}(x) \subseteq \mu_{\mathbb{M}_2}(\varphi(x))$$

holds for $\forall x \in \mathbb{M}_1$.

In some cases it might be necessary to perform a cast between arithmetic types even if this incurs an information loss. For these cases, a forced conversion is supported:

Definition 12 (Forced conversion) *Let* $\mathbb{M}_1, \mathbb{M}_2$ *be two support sets of either the exact or the finite algebra. A mapping* $\varphi : \mathbb{M}_1 \to \mathbb{M}_2$ *is called forced conversion if and only if it*

Table 8: Mapping of FPA, IA, AA and TMs to their concrete scalar, vector and matrix types in UniVerMeC.

ARITHMETIC	SET	SCALAR	VECTOR	MATRIX
FPA	$\overline{\mathbb{F}}$	`mreal`	`rvector`	`rmatrix`
IA	$\overline{\mathbb{IF}}$	`interval`	`ivector`	`imatrix`
AA	$\overline{\mathbb{AF}}$	`aaf`	`avector`	`amatrix`
TMs	$\overline{\mathbb{TF}}$	`taylormodel`	`tvector`	`tmatrix`

1. *is a lossless conversion if such a conversion exists,*
2. *is an enclosure preserving conversion if such a conversion exists.*

All three conversion types may return a special form if an error (e.g. overflow) occurs.

3.6 IMPLEMENTATION OF THE ARITHMETIC LAYER

Adapter classes

UniVerMeC does not provide any arithmetic implementations of its own. Instead it relies on the existing implementations, for example, C-XSC, filib++, YalAA, or Riot, which have proved their merits over the years. To ensure that these libraries match the common interface defined by the heterogeneous algebra, the framework provides a number of adapter classes, closely following the well-known adapter pattern [Gam+95, pp. 139-150]. The adapters are simple classes that do not belong to any class hierarchy. That is, they do not implement some sort of abstract arithmetic interface because it would introduce a call to an overloaded function[16] at the lowest level inside the framework for every arithmetic operation. Instead the adapters are generated semi-automatically. The automation has several benefits. It is less error prone than the manual implementation of each adapter. Furthermore, the differences between the adapters for two libraries are usually small. Hence, the automatic generation reduces the amount of work.

Automatic generation

To automate the adapter creation, a generator is applied. The generator is a technique from the field of generative programming. It is used to generate the implementation of a program automatically from certain specifications. According to Czarneski and Eisenecker [CEoo, pp. 339-341], a generator can be implemented by various techniques, for example, the compiler framework itself (e.g. preprocessor, template meta-programming) or an external code generator. They

16 In this case, a dynamic dispatch is performed during the call, that is, the function called actually is determined depending on the runtime and not on the static type of an object. In C++, this is done by looking into the virtual function table of the object which incurs a small runtime penalty in comparison to static calls [Lip96, pp. 139-144].

Table 9: Conditions that the M4 macro package takes for granted for the underlying types at each level in the hierarchy. The conditions are a direct result of the heterogeneous algebra definition and arithmetic hierarchy.

TYPE(S)	MACRO	REQUIREMENTS ON UNDERLYING TYPE
`mreal`	`REGISTER_REAL`	Lossless conversion from `double` Lossless conversion to `double`
`interval`	`REGISTER_REAL`	Lossless conversion from `mreal` Upper and lower bounds lossless convertible to `mreal`
various	`REGISTER_OTHER`	Lossless conversion from `mreal` and `interval` Enclosure preserving conversion to `interval`

identify four tasks for a generator: It should ensure the *validity of its input*, then use standard settings to *complement* the specification, and then *optimize* it. Finally, it should *create the implementation*. Because the wrappers created automatically in UNIVERMEC are very short, the optimization step is not applicable. The generator automatically creates C++ types that correspond to the heteregenous algebra. That is, it ensures that the signatures of the generated types match the signatures defined in (15). Table 8 shows the C++ types in UNIVERMEC and their corresponding support sets. Basically, every generated type is a composition of an *underlying type* supplied by an external library that implements the actual arithmetic and the method signatures defined by the algebra definition. The task of the generator is then to implement all methods in such a way that they call the corresponding method of the external library.

Basically, the generator has to to provide an adapter skeleton and to carry out text replacements inside it. For these tasks, macro processors are specialized languages that read input streams and copy them to the output. During this process, the stream is tokenized and macros matching specific user-defined tokens are expanded. Therefore, we implemented the generator as a set of scripts in the POSIX standardized M4 [KR77] language[17]. In contrast to the much simpler macro processor integrated in the C/C++ language environment, M4 supports sophisticated control structures, for example, conditional macro expansion and recursion through rescanning of tokens. Therefore, it greatly simplifies the process of creating a flexible system for automatic adapter generation.

17 We chose M4 despite its age because it is lightweight, available on almost every *NIX-operation system, and already used inside the build–system of UNIVERMEC, thus eliminating the need for another language.

```
dnl List of function names in the *external* library
define('UNARY_STANDARD_FUNCTIONS_interval', ''abs', 'acos',
    'acosh', 'asin', 'asinh', 'atan', 'atanh', 'cos', 'cosh',
    'exp', 'log10', 'sin', 'sinh', 'sqr', 'sqrt', 'tan',
    'tanh', 'Inf', 'Sup', 'mid'')

dnl Code block for performing inf/sup
define(B_PERFORM_UNARY_FUN_inf_interval, 'return
    mreal(_double(Inf(args.m_adapt)));')
define(B_PERFORM_UNARY_FUN_sup_interval, 'return
    mreal(_double(Sup(args.m_adapt)));')

dnl Code block for the integer power function
define('B_PERFORM_BINARY_FUN_pown_interval','arg.m_adapt =
    power(arg.m_adapt,e); return arg;')

dnl Registers the interval type
REGISTER_INTERVAL('cxsc::interval','','// C-XSC Headers
#include <interval.hpp>
#include <imath.hpp>
')
```

Listing 1: Excerpt from the M4 macros to register an interval type using C-XSC as the underlying library.

Design of adapter macros

The design of the macro package is based on the arithmetic hierarchy (see Fig. 15). For every hierarchy level, it provides a macro that creates the types on this level: `REGISTER_REAL`, `REGISTER_INTERVAL`, and `REGISTER_OTHER`. The reason for this division is that the underlying type has to fulfill different requirements depending on its position in the arithmetic hierarchy. The requirements are listed in Tab. 9. The package ensures that exactly one adapter for FP numbers and one adapter for intervals is registered. At the top level, it allows the registration of an arbitrary number of adapters. Currently, we only consider affine forms and TMs. Moreover, the common macro `REGISTER_OTHER` allows for easy addition of new types, which can be either a new kind of arithmetic, for example, Chebyshev models[18], or variants of an existing arithmetic, for example, the AF1 or AF2 forms for AA. Therefore, it ensures the extensibility of the framework at this level. Note that adding a new arithmetic might also require that users extend the theoretical basis (e.g. the inclusion representations and algebra) to ensure that the framework still produces correct results.

Listing 1 shows an excerpt[19] of the M4 macros a user has to write to introduce the C-XSC interval type into UniVerMeC. Listing 2 shows

18 Currently, several researchers seek to provide an alternative to the classical TMs by using the Chebyshev basis [Dze12; Jol11]. Chebyshev models might offer better enclosures due to better properties of Chebyshev polynomials. When the work is finished, the corresponding implementations could be included in our framework easily.

19 A few minor parts were left out for clarity.

```
define('UNARY_STANDARD_FUNCTIONS_interval', ''Abs', 'ArcCos',
    'ArCosh', 'ArcSin', 'ArSinh', 'ArcTan', 'ArTanh', 'Cos',
    'Cosh', 'Exp', 'Log10', 'Sin', 'Sinh', 'Sqr', 'Sqrt',
    'Tan', 'Tanh', 'Inf', 'Sup', 'Mid', 'Diam', 'rad'')

dnl Code block for the integer power function
define('B_PERFORM_BINARY_FUN_pown_interval','arg.m_adapt =
    Power(arg.m_adapt, e); return arg;')

dnl Hull Operator
define('B_PERFORM_OP_|_interval_interval', 'm_adapt =
    Hull(m_adapt, other.m_adapt);')
dnl Intersection operator
define('B_PERFORM_OP_&_interval_interval',
    'if(!Intersection(m_adapt, m_adapt, other.m_adapt))
    m_adapt = INTERVAL(1, -1);')

dnl Registers the interval type
REGISTER_INTERVAL('INTERVAL','','// PROFIL/BIAS Headers
#include <Interval.h>
#include <Functions.h>
')
```

Listing 2: Excerpt from the M4 macros to register an interval type using PROFIL/BIAS as the underlying library.

the macros necessary for the PROFIL/BIAS library. The first `define` specifies the names of standard functions as used by the *library*. They are employed to automatically create calls to the corresponding library functions in the function bodies of the adapter. The bodies of these functions are generated by expanding macros of the form `B_PERFORM_UNARY_FUN_<function name>_<type>` for functions and `B_PERFORM_OP_<operation name>_<type left>_<type right>` for operators. Normally, these macros are created automatically based on the function names list provided by the user and the assumption that the external library provides operator overloading for its data type. If the automatic generation is not suitable, users can suppress it by specifying these macros manually. This has to be done, for example, for the `inf, sup, mid` functions in case of the C-XSC library, because it does not return `double` (which we can cast to our `mreal` type) but its own type (`cxsc::real`) that encapsulates a `double`. After providing the manual macro overloads, the `REGISTER_INTERVAL` function is called to register the new type in the adapter generator framework. The call takes as arguments the underlying type and the library headers to be included. The information in the adapter generator framework is not only utilized to generate the adapters but also to create additional support classes in the framework automatically. This ensures that the framework can be extended with new arithmetic types easily. The process is depicted in Fig. 15. The trait classes generated automatically provide operations for casting between the arithmetic

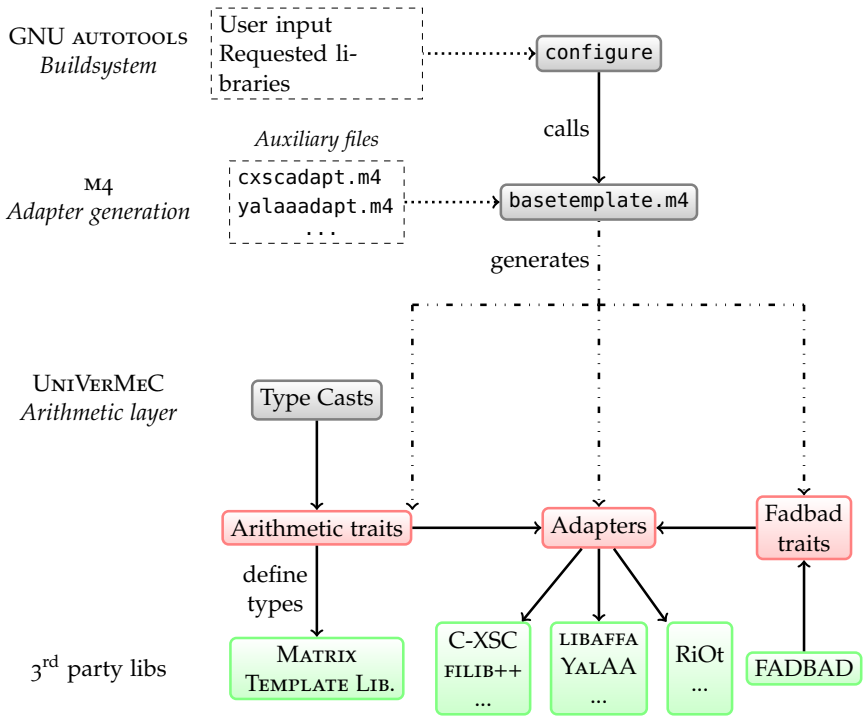


Figure 15: The arithmetic layer of UNIVERMEC and its semi-automatic generation with the help of the build system and the M4 macro processor. Automatically generated parts are drawn in red and external libraries in green.

types according to the hierarchy in Fig. 14. These are used by the actual cast operators `lossless_cast`, `enclosure_cast` and `forced_cast` based on Def. 10, 11, or 12 respectively. Type definitions for the corresponding vector and matrix types of the framework are also contained in trait classes. UNIVERMEC uses the MATRIX TEMPLATE LIBRARY 4 [GL], which allows for a uniform handling of vectors and matrices of different base types and ensures common interfaces for all vector and matrix classes. In the last step, the M4 generation layer creates suitable FADBAD++ [Sta97] trait classes for all generated adapters. FADBAD++ is a free library for algorithmic differentiation (AD), (cf. Sect. 4.1) including the computation of Taylor coefficients. It can be applied to arbitrary arithmetic types.

3.7 GPU-POWERED COMPUTATIONS

GPU computations

So far we did not discuss interoperability problems that occur if the computation is *not* carried out on the same hardware architecture, that is, a IEEE 754-2008 compliant CPU. Recently, the use of highly parallel many-core architectures has attracted much attention since they are now available at a relative low cost in form of modern GPUs. Historically, GPUs were highly specialized hardware for rendering. In the early days, they used merely a non-programmable

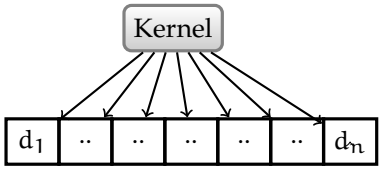


Figure 16: In the stream processing model the same kernel is applied to all data elements $d_1, \ldots, d_n$. The order in which the data elements are processed is not defined.

fixed-function pipeline. As computer graphics became more sophisticated, the need for more flexible GPUs arose. This lead to the introduction of the domain specific shader languages. First, their expressiveness was limited and tailored to computer graphics. Over time, the languages were extended and many of their restrictions disappeared. Owing to this, the field of general purpose computations on the GPU (GPGPU) appeared. While the first GPGPU programs were written using the computer graphics specific shader languages, special languages like OpenCL (*Open* Computing *L*anguage) [Ope] or CUDA (*C*ompute *U*nified *D*evice *A*rchitecture) [NVI12] were developed later. The former is a vendor independent standard maintained by the Khronos group. The latter was developed as a specific language for NVIDIA cards originally. We use CUDA in the remainder of this thesis because, in our opinion, it still has a better overall tool support. However, the CUDA and OpenCL programming models are so similar in general that porting our implementation to OpenCL should not be too difficult.

Stream processing model

Both languages are based on the so-called *stream processing* model, which is shown in Fig. 16. It consists of a *stream* that is composed of many data elements and a *kernel*. The kernel is applied to all elements of the stream in parallel. This model imposes several restrictions on the tasks for which it can be efficiently applied. First, the task needs to be parallelizable, that is, the kernel should be able to process the members of the stream independently. Furthermore, the model is efficient only if the stream is large enough. Another restriction is imposed by the fact that, currently, the stream processing is done on an extra device such as the GPU with its own memory. Because we have to transfer the stream first from the main memory to the GPU, it is important that the kernel is computationally expensive. Otherwise, the gain in time from using the parallel architecture will be negated by the input/output time needed to transfer the stream and the results from the main memory to the GPU and back.

A GPU consists of *stream multiprocessors* and can run hundreds of threads simultaneously. Compared to CPU threads, the GPU threads are much more lightweight. In contrast to common CPU single in-

struction single data (SISD) architectures[20], GPUs employ a single instruction multiple data (SIMD) model. In the scope of NVIDIA CUDA [NVI12, p. 61], it is also referred to as single instruction multiple threads (SIMT) so as to emphasize that the architecture applies to threads and that programmers do not need to cope with diverging branches manually, as is the case in some traditional SIMD architectures. The divergences occur if a branching condition, for example, an `if` statement, is evaluated differently by two threads. It is problematic because the execution of diverging branches can lead to a serialized execution[21] on a GPU. This prevents the utilization of a great part of the computational power of a GPU which depends mainly on parallelization. Therefore, problems that inhabit a large amount of branching are not well suited for GPU computations in general.

Range computations on the GPU

In connection with computer graphics applications, interval and affine arithmetic received some attention from the point of view of GPU parallelization [BN13; FPC10; Hij+10; Kno+09]. However, most of these publications focus on real-time ray-tracing and not on deriving verified results on the GPU. Currently, the adaption of interval-based algorithms for the GPU is obstructed by the lack of appropriate software libraries on the GPU. The Boost.Interval GPU [Hwu11] library is an exception. However, it is outdated and does not make use of the recent improvements in the IEEE 754-2008 support for modern GPUs [WFF11]. An improved version of the library is currently supplied with the CUDA toolkit. Like its predecessor, it is limited to the basic operations, which means that only rational functions can be evaluated with it. To our knowledge, there is no verified GPU AA or TM library publicly available. A goal of an ongoing master thesis at the university of Duisburg-Essen is to provide a GPU powered implementation of YalAA [Gö13].

GPU algebra

For these reasons, we limit our support for elementary functions in the algebra with regard to the GPU to the basic operations (including square root), and FP and IA . That also implies that the GPU support does not fit well into our general arithmetic based on the heterogeneous algebra and has to be handled separately. We denote the set of IEEE 754-2008 (extended) double numbers and intervals on the GPU by $\mathbb{F}'$ ($\overline{\mathbb{F}}'$) and $\mathbb{IF}'$ ($\overline{\mathbb{IF}}'$) respectively. Because the formal derivation of the GPU algebra $\mathfrak{GA}$ is analog the CPU FP algebra (15), we omit it and only state its definition:

$$\mathfrak{GA} := \left(\{\overline{\mathbb{F}}', \overline{\mathbb{IF}}'\}, \{\mathrm{sqrt}, \mathrm{pown}, +, -, \cdot, /\}\right). \tag{16}$$

20 Note that a modern CPU often has more than one core and thus can process multiple threads in parallel. Extensions such as MMX, 3DNow! or SSE add SIMD instructions to the CPU, which we do not consider at first.

21 Whether the execution has to be serialized depends on implementation details. For current NVIDIA cards, it depends on whether the threads are in the same group of threads (a so-called *wrap*), which share certain resources on the card.

Similar to the CPU algebra, the supported basic operations and elementary functions (cf. Tab. 6) of $\mathfrak{GA}$ are required to be inclusion functions according to Def. 7. Note that both CPU and GPU algebras ultimately use the IEEE 754-2008 double type as base type. That is, we can use this type to transfer data from the CPU to the GPU and back safely. These transfers do not incur any rounding or loss of information. Thus, they are suitable for rigorous computations.

We do not maintain a complex framework of automatically generated adapter classes for the GPU but use a slightly enhanced version of the interval type supplied with the CUDA framework that fits our algebra definition. The reasons are the following:

1. The GPU library support is still very limited, and we are not aware of any third party libraries that we could support.
2. In contrast to the CPU arithmetic framework, we consider only FPA and IA.
3. We do not provide support for libraries like FADBAD++ or MTL4 on the GPU and thus do not need additional adapter classes.
4. GPU development is moving very fast. Therefore, it seems too early to provide a fixed framework similar to that on the CPU.

In fact, we do not provide extensive support tools for GPU programming on the arithmetic layer because GPU routines are still implemented on a low level. Therefore, GPU programs are often tailored towards a specific problem. Other aspects of the GPU support in UniVerMeC are discussed in the next chapter. They concern the automatic transfer of large data sets (streams) to the GPU so that it can process them effectively.

3.8 CONCLUSIONS

Heterogeneous algebra

In this chapter, we have presented an approach for handling different rigorous range-arithmetics in a uniform environment. It is based on the theoretical framework of inclusion representations by Vu, Sam-Haroud, and Faltings, which allows for performing mixed computations with different arithmetics to improve inclusions in the scope of contraint propagation. We extended this framework to cover a broader range of theoretical problems. However, the larger part of our contribution in this chapter was devoted to solving the practical problems of using different arithmetics simultaneously. For this purpose, we presented a heterogeneous algebra that defines common sets of compositions and elementary functions and is supported in the upcoming IEEE P1788 standard. The approach helps to overcome two shortcomings of existing software: First, arithmetics can now be used interchangeably to evaluate functions. Second, the algebra formalize and restrict the ways two different arithmetic types

can interact with each other in the same computation. The concept of using more than one arithmetic type is refined by arranging them in a hierarchy and deriving formal conditions for lossless and enclosure preserving conversions. While the actually applied conversion algorithms are well-known, the formalized conditions for the conversion types based on the inclusion representations are new. An important advantage is that our approach can be extended to a new arithmetic by merely providing an inclusion representation for it. The heterogeneous algebra can be extended for a new arithmetic in most cases straightforwardly.

Practical implementation

The second part of this chapter was devoted to the practical considerations that have to be taken into account if different range arithmetics are applied in a rigorous computation environment. Our goal was to provide a software that implements the theoretical framework by employing existing libraries instead of developing a conforming implementation for each arithmetic. We reached the goal by applying methods from generative programming to provide an adapter generator framework. Compared to traditional approaches, which use a fixed set of libraries, our new technique is much more flexible. The approach is vendor independent, that is, we support the use of different arithmetics provided by different libraries. Another advantage of the automatic generation is that additional libraries can be incorporated without too much effort on the users' side, if these libraries follow interface conventions used by most arithmetic types in C++. Furthermore, similar to its theoretical foundation, the practical adapter generation framework can be extended easily if necessary.

4

FUNCTIONS IN UNIVERMEC

Function representation

This chapter discusses functions in the context of UniVerMeC. Function is understood in a mathematical sense, that is, as a mapping $\mathbb{R}^n \to \mathbb{R}^m$. An important feature of our framework is the capability to encapsulate such functions into objects that can be passed inside the framework. By using these objects, it is possible to evaluate inclusion functions for all supported arithmetics. As mentioned in Chap. 2, one of our design goals was to allow users as much freedom as possible in entering functions into the system and high flexibility with respect to their internal representation. For example, it should be possible to enter functions into the framework as closed form analytical expressions (e.g. $f(x, y, z) = x^2 + y^2 + z^2 - 1$) or as C++ code blocks, which are allowed to contain constructs such as `while` or `if` directives depending on the function variables.

Feature system

Such flexibility of our approach comes at a price. For example, we cannot assume that all functions are differentiable. To be able to exploit such special characteristics of functions, we introduce a feature system: a set of orthogonal functionalities[1] that are associated with a concrete function. Our software features are realized by interfaces providing access to them. While UniVerMeC supplies standard implementations for the feature interfaces, users are always free to implement these interfaces themselves. For example, derivatives can be obtained through UniVerMeC's standard implementation by algorithmic differentiation, but users can also choose to provide the derivatives manually. Both custom implementations by users and our standard implementations have to obey the formal definitions for the interfaces that we derive in Sect. 4.4.1.

Limitations

This generic approach makes the framework more complex. Additionally, it becomes more difficult to optimize algorithms because nothing is known about the internal representation of functions. The implementation of techniques that depend on specific internal function representations, and which cannot work with the black boxes that UniVerMeC provides, becomes very complicated. Depending on the intended application domain and solving techniques, other more specific implementations may be better suited. An example is the IBEX library [Cha], which provides functionality that partially overlaps with UniVerMeC (cf. Sect. 1.3). It implements ideas from contractor programming [CJ09] and allows users to adapt algorithms by combining one or several contractors with a bisection scheme in

1 That is, the functionalities do not interact and arbitrary combinations of them are possible. [Ale01, pp. 19-20]

the scope of constraint programming or global optimization. In IBEX, a function is defined as an expression which is in turn represented as a directed acyclic graph (DAG). Evaluations of the function and its derivatives are carried out on the DAG and are restricted to IA.

Comparison with IBEX

In this respect, the IBEX approach to function representation is more limited than that of UNIVERMEC. Both approaches have advantages and disadvantages. On the one hand, contractors can be optimized towards the function representation with DAGs, which might lead to faster algorithms. This is not possible for the function representation in UNIVERMEC because nothing is known about its internal structure. Another positive argument in favor of IBEX is that, contracting techniques that require forward/backward propagation such as HC4 [Ben+99] can work on a DAG basis automatically. On the other hand, certain functions cannot be represented by DAGs easily, whereas UNIVERMEC can handle them in a straightforward way. Examples are the in/out function for polyhedrons realized with ray intersection (cf. Sect. 5.1) or the objective functions for the parameter identification optimization problem for SOFCs (cf. Sect. 8.2). Moreover, it is possible to extend UNIVERMEC so that it provides a DAG structure for functions (where it is reasonable) through its feature system.

Chapter structure

This chapter consists of two parts: In the first part, we review well-known basic techniques for AD, range-enclosures, and box contraction, which we apply later in the scope of hierarchical space decompositions (cf. Sect. 6.1) and the algorithms (cf. Chap. 7). After that, details are given on one possible realization of a universal function representation in software in the second part.

Section 4.1 contains a brief overview on AD. In Sect. 4.2, we discuss several basic enclosure techniques, including mean-value forms. The first part of the chapter ends with Sect. 4.3 in which we describe interval contractors. We begin the second part of the chapter by determining a theoretical basis for representing functions. After that, we show how the theoretical basis can be realized in software by providing a set of appropriate interfaces. We conclude with a brief overview of standard implementations for the interfaces supplied by UNIVERMEC.

4.1 ALGORITHMIC DIFFERENTIATION

Algorithmic differentiation

Derivatives are required by any algorithms in the context of verified computations. Consider a function f described by a code sequence. Differentiating it by hand or using a computer algebra system requires an analytical expression. Such an expression is sometimes difficult to obtain. Algorithmic differentiation is a well-known technique (cf. [Gri00; Nau12; Ral81][2]) for details) that allows us to obtain the numerical values for derivatives at points also without explicit analytic

2 We base our further discussion on [Gri00] mainly.

expressions. Because AD is based on the chain rule, the method is similar to classic symbolic differentiation. In contrast to numerical methods such as divided differences, it does not introduce additional truncation errors into the computation [Grioo, p. 2].

Evaluation procedure

To apply AD to a function described by a code sequence, we restrict our discussion to functions $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}^m$ over an open set $\mathcal{D}$, which can be expressed as a finite composition of the basic binary operations $\{+, -, \cdot, /\}$ and differentiable elementary functions, such as the sine as defined in the algebra (14) of the arithmetic layer. Note that this restriction excludes conditions from the code sequence that depend directly or indirectly on the variables of f. The code sequence defining the function f with n inputs and m outputs can be evaluated by the following procedure [Grioo, p. 19]:

$$\begin{aligned} v_{i-n} &= x_i \ , & i &= 1 \dots n \ , \\ v_i &= \varphi_i(v_j)_{j \prec i} \ , & i &= 1 \dots l \ , \\ y_{m-i} &= v_{l-i} \ , & i &= m-1 \dots 0 \ , \end{aligned} \tag{17}$$

where x_i are the input variables, y_{m-i} are the outputs, and φ_i are elementary operations as defined above. The relation $\prec$ denotes a direct dependence of v_i on v_j.

Elemental differentiability

Generally, in an AD environment it is assumed that all $\varphi_i \in \mathcal{C}^d(\mathcal{D}_i)$. This assumption is called elemental differentiability (ED) [Grioo, p. 24]. With its help, Griewank formulates and proves the following theorem:

Theorem 3 (Chain Rule) *"Under Assumption ED the set $\mathcal{D}$ of points $x \in \mathcal{D}$ for which the function $y = f(x)$ is well defined by the evaluation procedure [(17)] forms an open subset of $\mathbb{R}^n$ and $f \in \mathcal{C}^d(\mathcal{D}), 0 \leqslant d \leqslant \infty$."*

The theorem states that if all elementary functions are continuous or continuously differentiable, the function f built of them inherits the respective feature in a certain domain.

Forward differentiation

To obtain derivatives through AD, two main methods are distinguished: *forward* and *backward differentiation*. We restrict our discussion to the basics of the former. If a function $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}^m$ is evaluated with the procedure (17), the variables v_i can be seen as nodes in a computational graph. They store the intermediate results from which the function values are computed. In forward mode, each node additionally stores the intermediate values for the directional derivatives. Using the procedure [Grioo, p. 39]

$$\begin{aligned} v'_{i-n} &= x'_i \ , & i &= 1 \dots n \ , \\ v'_i &= \sum_{j \prec i} \frac{\partial}{\partial v_j} \varphi_i(u_i) v'_j \ , & i &= 1 \dots l \ , \\ y'_{l-i} &= v'_{l-1} \ , & m &> i \geqslant 0 \ , \end{aligned} \tag{18}$$

Table 10: Evaluation trace of the implicit equation of an SQ and its directional derivative of x_1.

EVALUATION (17)			FORWARD DIFFERENTIATION (18)		
v_{-2}	=	x_1	v'_{-2}	=	1
v_{-1}	=	x_2	v'_{-1}	=	0
v_0	=	x_3	v'_0	=	0
v_1	=	v_{-2}/a_1	v'_1	=	v'_{-2}/a_1
v_2	=	v_1^{2/ϵ_2}	v'_2	=	$v_1'^{2/\epsilon_2-1}\frac{2}{\epsilon_2}$
v_3	=	v_{-1}/a_2	v'_3	=	v'_{-1}/a_2
v_4	=	v_3^{2/ϵ_2}	v'_4	=	$v_3'^{2/\epsilon_2-1}\frac{2}{\epsilon_2}$
v_5	=	v_2+v_4	v'_5	=	$v'_2+v'_4$
v_6	=	$v_5^{\epsilon_2/\epsilon_1}$	v'_6	=	$v_5'^{\epsilon_2/\epsilon_1-1}\frac{\epsilon_2}{\epsilon_1}$
v_7	=	v_0/a_3	v'_7	=	v'_0/a_3
v_8	=	v_7^{2/ϵ_1}	v'_8	=	$v_7'^{2/\epsilon_1-1}\frac{2}{\epsilon_1}$
v_9	=	v_5+v_8	v'_9	=	$v'_5+v'_8$
v_{10}	=	v_9-1	v'_{10}	=	v'_9
y_1	=	v_{10}	y'_1	=	v'_{10}

where $u_i = (v_j)_{j\prec i} \in \mathbb{R}^{n_i}$ denotes the arguments on which φ_i depends, a directional derivative can be computed.

Example

As an example, consider a superquadric [Bar81] described by an implicit function:

$$f(x_1,x_2,x_3) = \left(\left(\frac{x_1}{a_1}\right)^{\frac{2}{\epsilon_2}} + \left(\frac{x_2}{a_2}\right)^{\frac{2}{\epsilon_2}}\right)^{\frac{\epsilon_2}{\epsilon_1}} + \left(\frac{x_3}{a_3}\right)^{\frac{2}{\epsilon_1}} - 1\ , \quad (19)$$

where $a_1, a_2, a_3, \epsilon_1, \epsilon_2 \in \mathbb{R}$ are model parameters. Table 10 shows how we can evaluate f using (17) while simultaneously computing the directional derivative with respect to x_1.

Forward vs. backward differentiation

In the case of a vector-valued function $f(x) = (f_1(x), \ldots, f_m(x)), x \in \mathbb{R}^n$, the forward differentiation returns the directional derivative with respect to x_i for *all member functions* $f_1, \ldots, f_m$ in *one run*. In contrast, the backward differentiation which roughly speaking works by first evaluating a function f, storing some information during the evaluation and then calculating the derivatives through propagation of so-called *adjoints* beginning from the function values of f backwards returns *all directional derivatives* for *one function* f_i at a time. To summarize, it is advisable to use the forward differentiation if the derivatives of many dependent variables with respect to a few independent variables are required and to use the backward method in the opposite case. Both methods can be implemented through operator over-

loading or source code transformation, whereas operator overloading seems to be more widely used at least in C++ programs.

4.2 VERIFIED FUNCTION ENCLOSURES

As mentioned in Sect. 3.2.5, the evaluation of a function with IA using the natural extension can lead to considerable overestimation of the true range. Often, the natural extension with more sophisticated arithmetics, such as AA or TMs, produces better enclosures. Besides the enhanced arithmetics, using enclosure techniques other than the natural extension can deliver considerably better bounds for the range. In this chapter, we will discuss for them examples briefly, such as the mean-value form and its variants. Details on the implementations available in our framework are given at the end of Sect. 4.4.3.

4.2.1 *Mean-Value Forms*

Mean-value form

Let $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}, f \in \mathcal{C}^1(\mathcal{D})$, be given. Based on the mean-value theorem, an inclusion function F_m for f over $\mathbf{x} \subset \mathcal{D}$ can be derived [AH83, pp. 28-29] as

$$F_m(\mathbf{x}) \coloneqq f(\operatorname{mid}\mathbf{x}) + \nabla F(\mathbf{x}) \cdot (\mathbf{x} - \operatorname{mid}\mathbf{x}) \ , \tag{20}$$

where ∇F is an inclusion function for the gradient of f. F_m is called the *mean-value form*. If F_m is implemented for machine arithmetic, it is sufficient to choose the point $\operatorname{mid}\mathbf{x}$ such that $\operatorname{mid}\mathbf{x} \in \mathbf{x}$ holds. Usually, the mean-value form provides better enclosures for narrow intervals than the natural interval extension because of the quadratic approximation property it possesses. The use of *progressive gradients* [HW04, pp. 126-128] can improve the enclosure quality even more. The idea is to replace certain components of $\mathbf{x}$ by real numbers:

$$F_m(\mathbf{x}) \coloneqq f(\operatorname{mid}\mathbf{x}) + \sum_{i=1}^{n} \frac{\partial}{\partial x_i} F(\mathbf{x}_1, \ldots, \mathbf{x}_i, \operatorname{mid}\mathbf{x}_{i+1}, \ldots, \operatorname{mid}\mathbf{x}_n) \cdot (\mathbf{x}_i - \operatorname{mid}\mathbf{x}_i) \ . \tag{21}$$

Bicentered form

Another way to improve the enclosure is to replace $\operatorname{mid}\mathbf{x}$ by another point $c \in \mathbf{x}$. Such form F_c is called the *centered form*. Actually, the *theorem of Baumann* [Neu90, Theorem 2.3.6] allows us to derive values $c^+, c^- \in \mathbf{x}$ for centered forms leading to the best upper and lower bounds respectively. The intersection

$$F_{c^+}(\mathbf{x}) \cap F_{c^-}(\mathbf{x})$$

is called the *bicentered form*. It is twice as expensive as a normal centered form, but it returns the exact range if f is monotone as shown in [Neu90, p. 59].

Slopes

To improve the results further, the derivative ∇f can be replaced by a *slope* [Neu90, pp. 56-57]. The function $s_f : \mathcal{D} \times \mathcal{D} \to \mathbb{R}^n$ is called a slope for f if the relation

$$f(x) = f(c) + s_f(c, x) \cdot (x - c) \tag{22}$$

holds for certain $x, c \in \mathcal{D}$. The slope is uniquely defined only if $n = 1$. In the multivariate case, slopes are not unique. They can be calculated automatically by slope arithmetic [BHK03], which is similar to AD. An inclusion function for f can be obtained with slopes as follows:

$$F_s(\mathbf{x}) \coloneqq f(\operatorname{mid}\mathbf{x}) + S_f(\operatorname{mid}\mathbf{x}, \mathbf{x}) \cdot (\mathbf{x} - \operatorname{mid}\mathbf{x}) \ , \tag{23}$$

where S_f is an interval inclusion function for s_f.

4.2.2 *Other Enclosure Techniques*

Taylor inclusion function

Because bounding the range of a function is a crucial part of interval computations, many more techniques were proposed aside from the ones discussed in the previous section. For example, a second order *Taylor inclusion function* of f is defined as [Jau+01, p. 35]:

$$F_T(\mathbf{x}) \coloneqq f(c) + \nabla f(c) \cdot (\mathbf{x} - c) + \frac{1}{2}(\mathbf{x} - c) \cdot \nabla^2 F(\mathbf{x}) \cdot (\mathbf{x} - c) \ , \tag{24}$$

where $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}$, $f \in \mathcal{C}^2(\mathcal{D}), c \in \mathbf{x}, \mathbf{x} \in \mathcal{D}$, and $\nabla f, \nabla^2 f$ are the gradient and Hessian matrix of f. The enclosure can be improved further if the progressive gradient technique is applied to both the gradient and the Hessian matrix. In this case, the Hessian matrix is no longer symmetric and all n^2 entries have to be computed. Therefore, it is advisable to use a replacement scheme that retains at least parts of the symmetry [HW04, pp. 128-129].

Exploiting monotonicity

If a function has certain features, specialized procedures for range bounding might be applicable. For example, if a function $f : \mathbb{R} \supset \mathcal{D} \to \mathbb{R}$ increases monotonically over an interval $\mathbf{x} \in \mathcal{D}$, a range enclosure can be computed by

$$F_M(\mathbf{x}) \coloneqq [f(\underline{x}), f(\overline{x})]$$

or by

$$F_M(\mathbf{x}) \coloneqq [f(\overline{x}), f(\underline{x})]$$

if f decreases monotonically. This approach can be extended to the multidimensional case even if f is monotonic only in some of its vari-

ables [HW04, pp. 37-38]. For univariate functions, the *linear boundary value form* [Neu90, pp. 59-60] is of special interest if $0 \in f'(\mathbf{x})$. Specialized procedures are also available for polynomials (cf. [Sta95] for a comparison).

4.3 INTERVAL CONTRACTORS

Solving an equation

In the previous section, we discussed how to get a narrow bound on the range of a function. In this section, the task is to check whether x is a solution to the equation $f(x) = c$ with $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}$ and a certain constant $c \in \mathbb{R}$. Such a check is not easy to perform rigorously on a computer where we work with finite arithmetic[3]. It is *not* sufficient to replace both quantities x, c by intervals $\mathbf{x}, \mathbf{c}$ containing them and then to ascertain whether the implication $F(\mathbf{x}) = \mathbf{c}$ holds. More generally,

$$F(\mathbf{x}) \cap \mathbf{c} \neq \emptyset \implies \exists x \in \mathbf{x}, c \in \mathbf{c} : f(x) = c$$

does *not* hold for $\mathbf{x} \in \mathbb{IR}^n, \mathbf{c} \in \mathbb{IR}$ even in exact arithmetic because the intersection could be non-empty owing to overestimation. To prove that $\mathbf{x}$ contains a solution, we have to apply computational fixed-point theorems (cf. [FL05; FLS04; Moo77]). However, the opposite conclusion is true, that is,

$$F(\mathbf{x}) \cap \mathbf{c} = \emptyset \implies \neg\exists x \in \mathbf{x}, c \in \mathbf{c} : f(x) = c \ . \tag{25}$$

This fact helps us to decide whether a box $\mathbf{x}$ can be discarded without loosing a part of the solution. Because even the smallest intersection interval caused by overestimation forces us to treat the whole $\mathbf{x}$ as a potential solution, algorithms solely relying on the possibility to discard a box according to (25) often do not perform very well.

Contractors

To make better use of the available information, we can try to shrink the box $\mathbf{x}$ by removing its parts that do not solve the equation $f(x) = c$. Following, Jaulin et al. [Jau+01, p. 66], we call such shrinking techniques *contractors*. We use our definition from [KLD13] to define a contractor formally:

Definition 13 (Interval contractor) *Let $\mathbf{x}, \mathbf{c} \in \mathbb{IR}$ and $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}$ be given. A contractor C is an algorithm taking the function f and the intervals $\mathbf{x}$ and $\mathbf{c}$ as parameters, and returning an interval $\mathbf{x}^{(1)}$ such that $\mathbf{x}^{(1)} \subseteq \mathbf{x}$ and $\forall x \in \mathbf{x} \setminus \mathbf{x}^{(1)} : \neg\exists c \in \mathbf{c} : f(x) = c$. That is, it removes parts from $\mathbf{x}$ that do not solve the equality $f(\mathbf{x}) = \mathbf{c}$.*

We use the term *interval* contractor because its output $\mathbf{x}^{(1)}$ is an interval vector. Therefore, the maximum possible contraction that can

3 Even checking such a relation non-rigorously in FPA is not trivial. At the very least, it requires rewriting the equation as an inclusion $f(x) \in [c - \epsilon, c + \epsilon]$ for *some* $\epsilon > 0$ determined by the programmer.

be achieved by it is restricted to the smallest interval hull of the actual solution set. A contractor complying to Def. 13 possesses two of the three properties of a *constraint narrowing operator* identified by Benhamou in [Ben96]: *contractance* ($\mathbf{x}^{(1)}$ does not grow during the contraction) and *correctness* (no solution is lost). In contrast to the definition in [Ben96], we do not require *monotonicity*. That is, it does not follow from Def. 13 that $\mathbf{x}^{(1)} \subseteq \mathbf{x}'^{(1)}$ for solution sets of a contractor C if it is applied to the same relation $f(x) = y$ over different domains $\mathbf{x}, \mathbf{x}'$ such that $\mathbf{x} \subseteq \mathbf{x}'$. However, most implemented techniques will have this property in practice, because it is important for proving the convergence of branch and bound algorithms.

Coping with inequalities

Note that most methods discussed in the remainder of this chapter solve the equation $g(x) = 0$. We can still consider them as contractors in the sense of our definition by setting

$$\mathbf{f}(x) \coloneqq g(x) - \mathbf{c} \ . \tag{26}$$

To fit inequalities into our framework, we use half-open intervals as right-hand sides. That is, we rewrite $f(\mathbf{x}) \leqslant \mathbf{c}$ in $f(\mathbf{x}) = (-\infty, \overline{\mathbf{c}}]$. In the following subsections, we will give a short outline of the basics of several techniques that can be used to realize a contractor that conforms to Def. 13. The section focuses on techniques implemented in UniVerMeC and on improvements that can be provided in our framework relatively easily (for the actual techniques available in the framework, see Sect. 4.4.3).

4.3.1 *One-dimensional Interval Newton Contractor*

Newton iteration

A well-known technique for determining the roots of $f : \mathbb{R} \supset \mathcal{D} \to \mathbb{R}$, which is an at least onetime continuously differentiable function, is the Newton iteration[4]

$$x^{(k+1)} \coloneqq x^{(k)} - \frac{f\left(x^{(k)}\right)}{f'\left(x^{(k)}\right)}$$

where $x^{(k)}$ is a value sufficiently close to a root of f. The used notation and further discussion of the one-dimensional interval Newton operator

$$N\left(x^{(k)}, \mathbf{x}^{(k)}\right) \coloneqq x^{(k)} - \frac{f\left(x^{(k)}\right)}{F'\left(\mathbf{x}^{(k)}\right)} \tag{27}$$

with $\mathbf{x}^{(k)} \in \mathbb{IR}$ and $x^{(k)} \in \mathbf{x}^{(k)}$ is based on [HW04, pp. 169-173]. The *expansion point* $x^{(k)} \in \mathbf{x}^{(k)}$ can be chosen arbitrarily, with a common

4 See [Ueb95, pp. 319ff.] for a derivation and details.

Table 11: Results of the extended interval division of $\mathbf{x}$ and $\mathbf{y} \ni 0$ where 0 lies on one of the boundaries according to [Kulo9].

	$\mathbf{y} = [0,0]$	$\underline{\mathbf{y}} < \overline{\mathbf{y}} = 0$	$0 = \underline{\mathbf{y}} < \overline{\mathbf{y}}$
$\underline{\mathbf{x}} = 0 = \overline{\mathbf{x}}$	$\emptyset$	$[0,0]$	$[0,0]$
$\underline{\mathbf{x}} < 0 \geqslant \overline{\mathbf{y}}$	$\emptyset$	$[\frac{\overline{\mathbf{x}}}{\underline{\mathbf{y}}}, +\infty)$	$(-\infty, \frac{\overline{\mathbf{x}}}{\overline{\mathbf{y}}}]$
$\underline{\mathbf{x}} < 0 < \overline{\mathbf{x}}$	$\emptyset$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$\underline{\mathbf{x}} \geqslant 0 < \overline{\mathbf{x}}$	$\emptyset$	$(-\infty, \frac{\mathbf{x}}{\underline{\mathbf{y}}}]$	$[\frac{\mathbf{x}}{\overline{\mathbf{y}}}, +\infty)$

choice being the midpoint $\operatorname{mid}\mathbf{x}^{(k)}$. The interval Newton iteration is obtained by intersecting with the original box

$$\mathbf{x}^{(k+1)} \coloneqq N\left(x^{(k)}, \mathbf{x}^{(k)}\right) \cap \mathbf{x}^{(k)} \quad . \tag{28}$$

The most important property of the interval Newton operation is that no solution is lost. That is, if $x^* \in \mathbf{x}$ is a root of f, then $x^* \in N(x, \mathbf{x})$. Consequently, if (28) results in an empty interval, this is proof that f contains no root in $\mathbf{x}$ [Ham+95, Theorem 6.1]. Note that, for FP intervals, the theorems hold only if we ensure proper inclusions, for example, by using directed rounding modes. In particular, the evaluations of f and its derivative f' have to be performed with inclusion functions:

$$N\left(x^{(k)}, \mathbf{x}^{(k)}\right) = x^{(k)} - \frac{F\left(\left[x^{(k)}, x^{(k)}\right]\right)}{F'\left(\mathbf{x}^{(k)}\right)}$$

with $\mathbf{x}^{(k)} \in \mathbb{IF}, x^{(k)} \in \mathbf{x}^{(k)}$.

Extended interval division

The interval division in the Newton operator (27) is defined only if $0 \notin F'\left(\mathbf{x}^{(k)}\right)$. However, if f has multiple roots in the interval $\mathbf{x}^{(k)}$, this is not the case. To handle this, we can apply *extended interval division*. For this operation, many different definitions were proposed by various researchers. A thorough discussion of the different approaches can be found in [Che11]. In this thesis, we use the definition by Kulisch [Kulo9] that is used in the upcoming IEEE P1788 standard. To compute the result of $\frac{\mathbf{x}}{\mathbf{y}}$ for $\mathbf{x}, \mathbf{y} \in \mathbb{IR}$, Kulisch basically suggests to remove zero from $\mathbf{y}$ if it is present. Therefore, the division result is

$$\frac{\mathbf{x}}{\mathbf{y}} = \left\{ \frac{x}{y} \;\middle|\; x \in \mathbf{x}, y \in \mathbf{y} \setminus \{0\} \right\} \quad . \tag{29}$$

Detailed rules for obtaining the bounds for the set (29) are given in Tab. 11. It is assumed that the zero lies on the boundary of $\mathbf{y}$. If it lies in the interior, $\mathbf{y}$ can be split into two intervals, $[\underline{\mathbf{y}}, 0]$ and $[0, \overline{\mathbf{y}}]$, and the extended division can be performed with these parts independently. The procedure results in two different intervals, which can

either be processed separately or transformed in one proper interval by taking the convex hull. In this case, the information about the gap can be stored[5]. Note that the intersection in the interval Newton iteration (28) produces bounded interval(s) again in both cases.

Existence proof

Another important feature of the interval Newton operator is the ability to prove that a box contains a simple root. This is the case if

$$N(x, \mathbf{x}) \subset \mathbf{x}$$

holds. This is also true if the function f depends on an interval parameter $\mathbf{p}$[6]. In this case, $\mathbf{x}$ contains a simple zero for each $p \in \mathbf{p}$ [HW04, Theorems 9.6.8/9.6.9].

4.3.2 *Multidimensional Interval Newton Contractor*

Rewriting as linear systems of equations

Let $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}^n, f \in \mathcal{C}^1(\mathcal{D})$ and $\mathbf{x} \in \mathbb{IR}^n$ be given. To find all solutions to the nonlinear system of equations $f(x) = (0, \ldots, 0)^T$ for $x \in \mathbf{x}$, we can use the multidimensional interval Newton operator. Our discussion and notation are based on [HW04, pp. 238-241]. Usually, the one-dimensional relation (27) is rewritten in the multidimensional case as a linear system of equations because the derivative of f is now the Jacobian matrix. The necessity to invert it would restrict the applicability of the Newton operator to cases with a regular Jacobian matrix. For the interval-valued Jacobian matrix $\mathbf{J}_f(\mathbf{x})$, this means that every real matrix $M \in \mathbf{J}_f(\mathbf{x})$ has to be regular. Therefore, to find all solutions x^* in the multidimensional case, the following linear system of equations is usually solved

$$f(x) + \mathbf{J}_f(\mathbf{x})(x^* - x) = 0 \ ,$$

where $x \in \mathbf{x}$ is an expansion point.

Gauss-Seidel method

A common approach to find an enclosure for x^* is to use the iterative Gauss-Seidel method. Usually, the method is applied to system preconditioned with a matrix $B \in \mathbb{R}^{n \times n}$ in such a way as to obtain a diagonally dominant matrix and to decrease its spectral radius, which is important for the convergence of the method. The approximate inverse of mid $\mathbf{J}_f(\mathbf{x})$ is commonly chosen [MKC09, p. 96] for B and can be computed with FPA cheaply. Thus, we actually solve the system[7]

$$\underbrace{B \cdot \mathbf{J}_f(\mathbf{x})}_{:=\mathbf{M}}(x^* - x) = \underbrace{-B \cdot f(x)}_{:=r} \tag{30}$$

5 See, for example, the discussion in [Bee06, pp. 46-47] about the handling of gaps in the recent solver SONIC.

6 In particular, this case occurs if the rewriting rule (26) is applied.

7 If we compute with FPA we have to evaluate $F([x, x])$ instead of $f(x)$. Then the right-hand side is an interval $\mathbf{r}$.

By *solving*, we mean that we determine an interval box that contains the solution set of the system. Generally, this is not the smallest box but just an outer approximation. Finally, through the interval Gauss-Seidel iteration

$$\begin{aligned} N\left(x^{(k)}, \boldsymbol{x}^{(k)}\right)_i &:= x_i^{(k)} + \frac{1}{\boldsymbol{M}_{ii}^{(k)}} \left(r_i^{(k)} - \sum_{j=1}^{i-1} \boldsymbol{M}_{ij}^{(k)} \left(\boldsymbol{x}_j^{(k+1)} - x_j^{(k)} \right) - \right. \\ &\qquad \left. \sum_{j=i+1}^{n} \boldsymbol{M}_{ij}^{(k)} \left(\boldsymbol{x}_j^{(k)} - x_j^{(k)} \right) \right) \quad , \qquad (31) \\ \boldsymbol{x}_i^{(k+1)} &:= N\left(x^{(k)}, \boldsymbol{x}_i^{(k)}\right) \cap \boldsymbol{x}_i^{(k)} \quad , \end{aligned}$$

the i-th line of the multidimensional interval Newton operator and iteration is obtained. Analogously to the one-dimensional case, the extended interval division (29) is used if $0 \in \boldsymbol{M}_{ii}^{(k)}$. Arising gaps are handled similarly to the one-dimensional case. Similarly to the one-dimensional interval Newton method, no solutions are lost during the iteration process in the multidimensional variant. Consequently, we can also use $N(x, \boldsymbol{x})$ to prove that $\boldsymbol{x}$ does not contain a solution or that $\boldsymbol{x}$ contains a unique root [Ham+95, Theorem 13.1]

Hansen-Sengupta method

To improve the bounds, (31) can be altered so that the lines i with $0 \notin \boldsymbol{M}_{ii}$ are computed first. This optimized method is together with the optimization already included in (31) referred to as Hansen-Sengupta method [HW04, pp. 96-99] and used commonly.

Optimal preconditioners

In [Kea96, pp. 120-143], Kearfott studies other choices for the preconditioning matrix B. He introduces two classes that conform with certain optimality conditions he defines: C-preconditioners suitable for contracting a box during the Newton step and E-preconditioners for optimizing gaps appearing during extended interval division. Furthermore, he suggests various optimality criteria for each class of preconditioners, for example, width-optimality of the resulting box. The actual computation of the preconditioning matrices is carried out by solving a linear optimization problem of a special structure. Kearfott advises using the width-optimality preconditioner if the matrix $\boldsymbol{M}$ is dense and small, because its computation is not much more expensive than the inverse of the midpoint matrix.

Alternative procedures

Besides the Gauss-Seidel method, the solution set of (30) can be, for example, bounded with Gaussian elimination or Krawczyk's method. Neumaier gives a thorough discussion of these methods in [Neu90, pp. 124-131 152-166]. However, he proves that the bounds of the Gauss-Seidel iteration are never worse than the Krawczyk's ones for the same preconditioning matrix B [Neu90, Theorem 4.3.5].

4.3.3 *Consistency Techniques*

Consistency techniques

Various other techniques that can act as contractors appeared in the field of constraint programming and are sometimes denoted as *con-*

sistency techniques. In the scope of the thesis, we restrict our discussion to box consistency as described by Hansen and Walster [HW04, pp. 196-203]. An overview of other consistency techniques is to be found in [BG06].

Box consistency

Let $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}, f \in \mathcal{C}^1(\mathcal{D})$, $\mathbf{x} \in \mathbb{IR}^n$ and $\mathbf{y} \in \mathbb{IR}$ be given. The box consistency algorithm aims at removing parts of $\mathbf{x}$ that do not fulfill $f(x) = \mathbf{y}$ for $x \in \mathbf{x}$ and is thus a contractor according to Def. 13. It is based on an one-dimensional interval Newton step. First, it has to map the function f to n one-dimensional interval functions

$$\mathbf{f}_i(x_i) \coloneqq f(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, x_i, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_n) - \mathbf{y} \text{ for } i = 1, \ldots, n$$

suitable for this kind of operator. From the inclusion property of interval arithmetic, it is clear that any zero of f in $\mathbf{x}$ is also contained in $\mathbf{f}_i$. To increase the efficiency of the procedure, Hansen and Walster choose a subinterval $\mathbf{x}_L = \left[\underline{x_i}, c\right] \subseteq \mathbf{x}_i$ with $\underline{x_i} < c \leqslant \overline{x_i}$. The lower bound on $\mathbf{x}_i$ is improved by applying the interval Newton operator $N\left(\underline{x_L}, \mathbf{x}_L\right)$ defined in (27) on $\mathbf{f}_i$. With an appropriately chosen interval $\mathbf{x}_U$, the upper bound can be improved. The procedure is repeated for each component $\mathbf{x}_i$. In [HW04, pp. 201-202], detailed algorithms are given along with an explanation about how to choose the values for c.

4.3.4 *Implicit Linear Interval Estimations*

Thick hyperplanes

Bühler [Büh02] developed implicit linear interval estimations (ILIEs) in the scope of computer graphics to render implicit objects reliably. The boundary of an implicit object is described by the solution set $\mathcal{LS} = \left\{x \mid f(x) = 0, x \in \mathcal{D}\right\}$ of the equation $f(x) = 0$ with an implicit function $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}$. In this context $\mathcal{LS}$ is referred to as the level set of f. ILIEs can be used to tighten the level set and thus act as contractors. Bühler calls a *thick hyperplane*

$$h(x) = \sum_{i=1}^{n} p_i x_i + \mathbf{j} \tag{32}$$

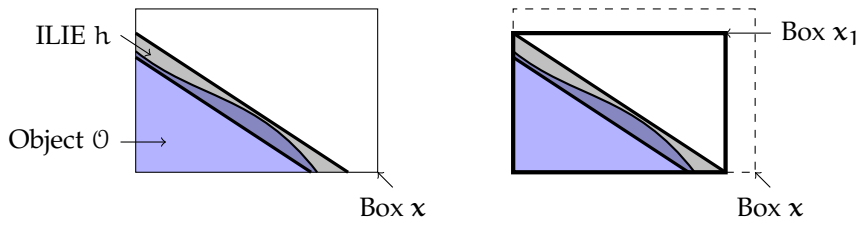
with $p_i \in \mathbb{R}$ and $\mathbf{j} \in \mathbb{IR}$ *implicit linear interval estimation* for f over the box $\mathbf{x} \in \mathbb{R}^n$ if and only if

$$x \in \mathcal{LS} \implies 0 \in h(x)$$

for all $x \in \mathbf{x}$. Thus, the thick hyperplane is by definition a verified linearization of f over $\mathbf{x}$ with respect to the level set. A graphical representation of an ILIE is shown in Fig. 17a.

Contraction

To tighten the solution set for the equation $f(\mathbf{x}) = \mathbf{c}$ over the box $\mathbf{x} \in \mathbb{IR}^n$ with $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}$ and $\mathbf{c} \in \mathbb{R}^n$, the ILIE h is constructed for



(a) The gray shaded ILIE h encloses the boundary of the blue object O (the level set of its implicit function) over $\mathbf{x}$.

(b) Using the ILIE, we can remove points from the box $\mathbf{x}$ that do not belong to the level set resulting in the smaller box $\mathbf{x}_1$.

Figure 17: Application of ILIEs for linearization and pruning.

the rewritten equation $f(\mathbf{x}) - \mathbf{c} = 0$. By formally solving the equation $h(x) = 0$, Bühler obtains the condition

$$x_i^* \in \left(-\mathbf{j} - \sum_{\substack{j=1 \\ j \neq i}}^{n} p_j \mathbf{x}_j \right) \frac{1}{p_i} \ , \qquad \text{for } i = 1, \ldots, n \ ,$$

which all $x^* \in \mathcal{LS}$ have to fulfill if $p_i \neq 0$. The new bounds $\mathbf{x}^{(1)}$ are then obtained straightforwardly by

$$\mathbf{x}_i^{(1)} := \left(-\mathbf{j} - \sum_{\substack{j=1 \\ j \neq i}}^{n} p_j \mathbf{x}_j \right) \frac{1}{p_i} \cap \mathbf{x}_i \ , \qquad \text{for } i = 1, \ldots, n \ . \quad (33)$$

The pruning process is illustrated in Fig. 17b. The technique applied here is called hull consistency (cf. [HW04, pp. 203-208]) in general.

Constructing ILIEs

So far we have not explained how to obtain the ILIE h. Basically, Bühler describes two ways to compute h for a given box $\mathbf{x}$ and a function $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}$. The first possibility is based on AA and the second on TMs. We only will outline the first variant because it is implemented in UniVerMeC. For the sake of simplicity, assume that the affine box $\hat{x}$ associated with the interval $\mathbf{x} \in \mathbb{R}^n$ (obtained by (12)) uses the symbolic noise variables $\epsilon_1, \ldots, \epsilon_n$:

$$\hat{x_i} := \operatorname{mid} \mathbf{x}_i + \operatorname{rad}(\mathbf{x}_i)\, \epsilon_i \text{ for } i = 1, \ldots, n \ .$$

The result of the evaluation of the natural affine extension of f over $\hat{x}$ is an affine form $\hat{f}$ with

$$\hat{f} = f_0 + \underbrace{\sum_{i=1}^{n} f_i \epsilon_i}_{\text{Linear dependencies}} + \underbrace{\sum_{i=n+1}^{l} f_i \epsilon_i}_{\text{Approximation errors}}$$

with a certain $l \geqslant n$. Basically, $\hat{f}$ consists of two sums: The first models the *linear* dependency on the input variables, and the second encloses the nonlinear approximation errors. The values f_i in the linear part are used to compute the normal vector of the oriented

hyperplane L and the distance d of the hyperplane to the origin. The central value f_0 also contributes to d. The thickness is determined by the nonlinear noise part of $\hat{f}$ and contributes together with d to the interval quantity j in (32). For a detailed algorithm, see [Büh02].

4.4 FUNCTION LAYER

Section's structure

This section explains the actual realization of the representation of mathematical functions inside UNIVERMEC. We begin the discussion by providing formal definitions for the representations. After that, we show how to transfer this formal framework to abstract interfaces in our software. Finally, the section concludes with the description of a flexible default implementation of these interfaces.

4.4.1 *Formal Definition*

Function representation objects

UNIVERMEC is not restricted to functions f given in a specific way (e.g., as an analytical expression). Instead, f is characterized by at least one inclusion function for each arithmetic supported by the framework. Formally, f is described by a *function representation object (FRO)*:

Definition 14 (Function representation object) *Let* $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}^m$ *be given. A function representation object of* f *is a tuple* $\mathfrak{O}_{f,n,m} = (\mathcal{I}, \mathcal{F})$ *where*

1. $\mathcal{I} = \{F^{(1)}, \ldots, F^{(q)}\}$ *is the set of inclusion functions where* q *is the number of arithmetics supported in the framework. This set contains one inclusion function* $F^{(i)}$ *(Def. 7) for each (finite) arithmetic.*

2. $\mathcal{F} \subseteq \{\mathfrak{E}_f^{(1)}, \ldots, \mathfrak{E}_f^{(r)}\}$ *is a subset of the set of features where* r *is the number of supported features in the framework. Each* $\mathfrak{E}_f^{(i)}$ *is a tuple of certain form (cf. Def. 15).*

The definition guarantees that each FRO can be evaluated at least with all supported arithmetics[8] using an appropriate inclusion function. Furthermore, a set of *features* can be provided that supplies additional functionalities associated with f. Possible features are, for example, further range-enclosure possibilities or contractors specialized for the given f.

Predefined features

Based on our discussion and our previous work, we identify the following set of features, which are important in our opinion and which we will need later (this set can be extended if necessary):

8 at the moment: FP , IA , AA and TMs

Definition 15 (Predefined Features) *Features are additional functionalities associated with a function f and connected to a FRO $\mathfrak{O}_{f,n,m}$. We define the following features*[9]*:*

1. *A derivative representation object for f is a tuple*
$$\mathfrak{D}_f = \Big(\mathfrak{O}_{\frac{\partial f_1}{\partial x},n,n}, \dots, \mathfrak{O}_{\frac{\partial f_m}{\partial x},n,n}, \quad \mathfrak{O}_{\frac{\partial f_1}{\partial x_1},n,1}, \dots, \mathfrak{O}_{\frac{\partial f_1}{\partial x_n},n,1}, \mathfrak{O}_{\frac{\partial f_2}{\partial x_1},n,1}, \dots, \mathfrak{O}_{\frac{\partial f_2}{\partial x_n},n,1}, \dots, \mathfrak{O}_{\frac{\partial f_m}{\partial x_1},n,1}, \dots, \mathfrak{O}_{\frac{\partial f_m}{\partial x_n},n,1}, \mathcal{I}' \Big)$$
with m FROs for each row of the Jacobian matrix ∂f of f, $m \cdot n$ FROs for each individual entry of ∂f and the set $\mathcal{I}' = \{\partial F^{(1)}, \dots \partial F^{(q)}\}$ of inclusion functions for $\partial f : \mathbb{R}^n \supset \mathcal{D}' \to \mathbb{R}^{m \times n}$, where q is the number of arithmetics supported in the framework.

2. *A Taylor coefficient representation object is a tuple $\mathfrak{TS}_f = \left(\{F_{TS}^{(1)}, \dots, F_{TS}^{(q)}\}, \{F_{\partial TS}^{(1)}, \dots F_{\partial TS}^{(q)}\}\right)$, where $F_{TS}^{(i)}$ are inclusion functions for the Taylor coefficients of the solution to the IVP, with the right side f and $F_{\partial TS}^{(i)}$ inclusion functions for the Taylor coefficients of its Jacobian matrix ∂f.*

3. *A custom enclosure representation object is a tuple $\mathfrak{CI}_f = \left(\{\mathfrak{O}_{f,n,m}^{(1)}, \dots, \mathfrak{O}_{f,n,m}^{(s)}\}\right)$ consisting of a set of s additional FROs of f.*

4. *A custom contractor representation object is a tuple $\mathfrak{C}_f = \left(\{C_f^{(1)}, \dots, C_f^{(t)}\}\right)$ consisting of t contractors (Def. 13), which work only on f.*

5. *A multi-argument function evaluation representation object is a tuple $\mathfrak{MA}_f = \left(\{F_{MA}^{(1)}, \dots, F_{MA}^{(q')}\}\right)$ for the q' arithmetics supporting multi-argument evaluation. The functions $F_{MA}^{(1)}, \dots, F_{MA}^{(q')}$ are inclusion functions for $f_{MA} : \mathcal{D}^l \to \mathbb{R}^{m \cdot l}$ with $f_{MA}(x) \coloneqq (f(x_1, \dots, x_n), f(x_{n+1}, \dots, x_{2n}), \dots, f(x_{l-n}, x_l))$, where l is the number of arguments with $l = n \cdot k, k \in \mathbb{N} \setminus \{0\}$.*

The first feature we define is the derivative representation object $\mathfrak{D}_f$. Its purpose is to "attach" derivatives to an FRO. The partial derivatives of a function f are FROs themselves. This has the important consequence that they can have features too, which allows us to introduce higher-order derivatives into the formal framework. Note that the full derivative of a vector-valued function is matrix-valued. Thus, it can only be characterized by an inclusion function in $\mathfrak{D}_f$

9 Note that the definitions of features reflect the actual reference implementations in UniVerMeC. The users are free to extend the set of features or implement the existing ones in their own way depending on their applications. However, the right way to handle some of the extensions (e.g., decorations from the upcoming IEEE P1788) is through specialized arithmetic functions in the heterogeneous algebra (cf. 3.5.2) and not through the feature system.

Table 12: The most important interfaces of the function layer and the formal concepts that they realize.

INTERFACE	DESCRIPTION	CONCEPT
`IVFunction` `IFunction`	function representation	Def. 14
`IDerivative`	derivatives of a function	Def. 15
`ITaylorCoeff`	Taylor coefficient's of a function	
`ICustomCon`	custom contractors for a function	
`ICustomEncl`	custom enclosures for a function	
`IGPUEval`	multi-argument evaluation	
`IContractor`	contractor representation	Def. 13

because FROs are limited to vector-valued functions within the framework. The Taylor coefficient representation object $\mathfrak{TG}_f$ is only of interest in the context of IVPs with the right-hand side f. Many IVP solvers (e.g., VNODE-LP in our case) need Taylor coefficients of a solution to compute its enclosure. With the third feature, the custom enclosure representation object, it is possible to provide additional inclusion functions for f. They might be of interest if f belongs to a class of functions (e.g., polynomials) for which different specialized techniques for constructing inclusion functions exist (e.g., application of the Horner scheme for polynomials). Similarly to the third feature, the custom contractor representation object is merely a set of contractors that exploit special knowledge about the inner structure of the given function. The multi-argument function evaluation representation object is the last feature we define. It allows us to evaluate the function for different arguments simultaneously.

4.4.2 *Interfaces of the Function Layer*

Connection to the theoretical basis

In this subsection, we discuss the software interfaces giving access to basic concepts defined previously. In Tab. 12, the most important interfaces we will discuss are shown. Each interface has a formal counterpart. We discuss how the implementation maps to the formal foundation and what requirements are induced on the actual classes implementing our interfaces.

Function Interfaces

FROs in software

The two interfaces `IVFunction` and `IFunction` shown in Fig. 18 are the key interfaces of the function layer. `IFunction` is merely a convenience interface for scalar-valued functions. It is derived from

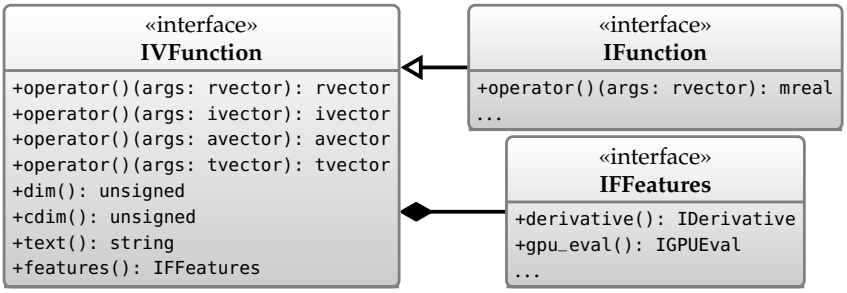


Figure 18: Interfaces `IVFunction` and `IFunction` for representation of functions.

`IVFunction` and overloads[10] the function operators so that they return scalars instead of vectors. In the remainder we discuss only `IVFunction`, but everything applies to the interface `IFunction`. Each instance of `IVFunction` is the implementation of an FRO $\mathfrak{O}_{f,n,m}$. The actual underlying function f is invisible inside UNIVERMEC. Instead, users can access it by evaluating an inclusion function of f for each of the supported arithmetics with the help of the overloaded function operators. In practice, these inclusion functions are often natural extensions (Def. 8) of expressions defining f, but this is not a necessary requirement. Because `IVFunction` is a black-box, users do no know whether the function is described by a simple closed form expression or by a complex algorithm. `IVFunction` provides means for requesting information about the dimension of the function's domain or codomain and, for convenience, an informal description.

Realization of the feature set

The feature set of the underlying FRO is realized through the interface `IFFeatures`, which is accessible from `IVFunction`. Its lifetime is the same as that of `IVFunction`. To extend the set of available features[11], `IFFeatures` needs to be extended with the corresponding code. Currently, all features from Def. 15 are supported. Since features represent fundamental concepts, changes in the interface `IFFeatures` are seldom necessary. Therefore, this is a reasonable trade-off. In the remainder of the subsection, we discuss each feature in some detail.

Derivative Interfaces

Representing derivatives

The representation of derivatives is almost as important as the representation of functions. Figure 19 depicts the interface `IDerivative` responsible for this task in our framework, which reflects a derivative representation object $\mathfrak{D}_f$ (Def. 15). Similarly to $\mathfrak{D}_f$, which is associated with an FRO $\mathfrak{O}_{f,n,m}$ and contained in its feature set, each

10 Note that the return types of the operators of `IVFunction` and `IFunction` are not covariant in the sense of [C11b, § 10.3/7]. The `IVFunction` operators are hidden by those of `IFunction`. The *static* object type and not the polymorphic one (as it would be the case in a standard `virtual` overload) determines what variant is called.

11 the set $\{\mathfrak{E}_f^{(1)}, \ldots, \mathfrak{E}_f^{(r)}\}$ in Def. 14

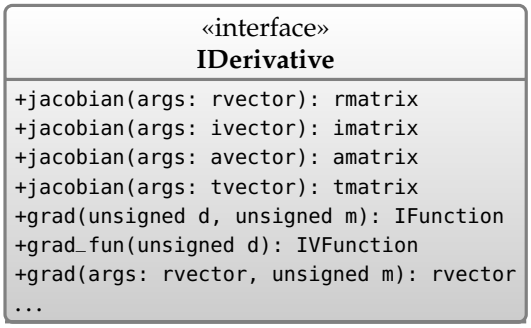


Figure 19: Interface `IDerivative` for accessing derivatives.

`IDerivative` instance is associated with an `IVFunction` object and accessible through its `IFFeatures` interface.

Structure of the Jacobian matrix

The `jacobian` member of the interface allows for evaluating inclusion functions of the Jacobian matrix ∂f of f in each arithmetic supported by UniVerMeC:

$$\text{Function values} \rightarrow \begin{pmatrix} f_1 & \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ f_2 & \frac{\partial f_2}{\partial x_1} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ f_m & \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} \begin{matrix} \leftarrow \texttt{grad\_fun(0)} \\ \\ \\ \leftarrow \texttt{grad(n,m)} \end{matrix} \tag{34}$$

In our realization, the returned Jacobian matrix is extended by the function values of f, which are stored in the first column[12]. For convenience, it is also possible to evaluate the gradient of the i-th member function through the `grad` member, which corresponds to the row in the Jacobian matrix specified by the second parameter.

Higher order derivatives

According to the definition of $\mathfrak{D}_f$, which requires representing rows or single entries of the Jacobian matrix as FROs, `IDerivative` allows to obtain `IVFunction` objects for those. This a special feature that allows for great flexibility inside the framework. The `grad_fun(n)` member returns the n-th row of the Jacobian matrix (and `grad(d,n)` the entry $\frac{\partial f_m}{\partial x_d}$) in the form of an `IVFunction` object. These new function objects behave as normal `IVFunction` instances in UniVerMeC. They can be passed to every algorithm that expects a function object as a parameter. In particular, these functions can also have features of their own, which allow for representing higher-order derivatives.

Example

For example, if $f : \mathbb{R}^n \supset \mathcal{D} \rightarrow \mathbb{R}, f \in \mathcal{C}^2(\mathcal{D})$, is represented by the `IVFunction` object `fun` and `x` is an interval box of the type `ivector`, then we can use:

```
const IVFunction &dfun =
    fun.features().derivative()->grad_fun(0);
std::cout << "Hessian:" << dfun().derivative()->jacobian(x);
```

12 This is convenient because the function values can be retrieved without much additional cost if, for example, AD is used for calculating the derivative values.

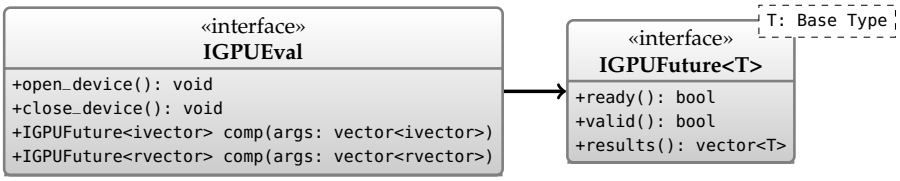


Figure 20: Interfaces `IGPUEval` and `IGPUFuture<T>` for function evaluation on the GPU.

to represent its gradient by a new `IVFunction` object `dfun`. Calling the `jacobian` member of `dfun` would return an interval-based enclosure of the Hessian matrix of f over $\mathbf{x}$. The lifetime of the new object `dfun` is determined by that of the `IVFunction` instance `fun` from which they originated.

GPU Evaluation Interface

GPU evaluation interface

As already mentioned in Sect. 3.7, UNIVERMEC supports computations on the GPU. This is realized at the function layer by providing the interface `IGPUEval`, which corresponds in our formal basis with a multi-argument evaluation representation object $\mathfrak{MA}_f$ (Def. 15). Therefore, each `IGPUEval` object is associated to an `IVFunction` instance and accessible through its feature interface. Figure 20 depicts the `IGPUEval` interface with its four member functions. The `open_device` and `close_device` members should be called prior to the first and after the last GPU function evaluation. They have no counterpart in $\mathfrak{MA}_f$. Their purpose is, for example, to allocate static data (e.g., model parameters) on the GPU before starting the computation.

Asynchronism

Following the definitions of $\mathfrak{MA}_f$, `IGPUEval` provides inclusion functions for f for all supported arithmetics on the GPU (cf. Sect. 3.7). We can evaluate them through the overloaded `comp` functions. Currently, they take either a list of interval vectors or real vectors and return a *future* object for the list of the corresponding inclusion function values. A future object is a kind of handle that refers to values that have not been computed at the time of returning it. It is commonly used in parallel computing (e.g., [HS08, pp. 369-375]). Therefore, a call to `comp` is non-blocking. For GPU computations, UNIVERMEC supplies the `IGPUFuture<T>` interface. It has a template parameter `T` that is chosen depending on whether an interval computation or FP computation is desired. This ensures type safety[13] and extendability with further types (e.g., affine forms) if necessary. With the help of `IGPUFuture<T>`, users can check whether the results have been already computed (the `ready()` member) or whether the future is valid. The `results()` member returns the list of function values and blocks until the computation is finished. Using the future mechanism, it is possible to interleave one or several computations on the GPU, which allows for better utilization of the GPU's resources.

13 The result type is statically determined by the overloaded function call of `IGPUEval`.

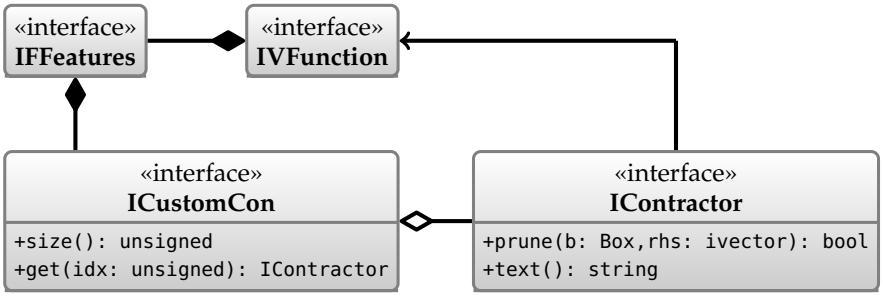


Figure 21: The `IContractor` interface.

Contractor Interface

Representing contractors

Contractors are represented by the `IContractor` interface. Each instance of an `IContractor` stands for one contractor algorithm C, which complies to Def. 13 and is bound to an `IVFunction` object and thus to an FRO $\mathfrak{O}_{f,n,m}$ from our theoretical basis. The structure of `IContractor` is depicted in Fig. 21. The `prune` member takes two interval vectors $\mathbf{b}, \mathbf{r}$ as arguments. It calls the contractor algorithm C to prune $\mathbf{b}$ with respect to the equation $f(b) = r$ for $b \in \mathbf{b}, r \in \mathbf{r}$. Usually, an inclusion function for f is used for this purpose, which is provided by the `IVFunction` object with which the `IContractor` is associated. The `IContractor` interface can be seen as the abstract strategy superclass in a direct application of the *strategy pattern* [Gam+95, pp. 315-323]. As shown in Fig. 21, the argument $\mathbf{b}$ is of the type `Box` instead of the plain interval vector type `ivector`. The purpose of this is that the class `Box` can be extended by further inheritance. Derived classes can store additional information about the region in search space covered by the `Box` object. An example for such a specialization is the `GappedBox` class, which is provided by the `core` layer of UniVerMeC. It stores gaps in the box, that is, parts of the search region covered by the box that do not contain a solution for $f(b) = r$. Such gaps can occur if, for example, the extended interval division (29) is employed in an interval Newton step.

Custom contractors

While some contractors can be associated with almost every function that fulfills certain requirements[14], other contractors need more precise knowledge about a function. They are tightly coupled with a specific function or its implementation and are accessible through the interface `ICustomCon`, which can be obtained from the feature interface of the corresponding function. This interface is merely a container class that stores the respective contractors and reflects the definition of a custom contractor representation object (Def. 15). For example, custom contractors could be used to implement techniques, such as HC4 [Ben+99], which require a forward/backward propagation on a function's computational graph.

14 e.g., the box consistency from Sect. 4.3.3 only requires first-order derivatives

Function Enclosures

Handling of function enclosures

Another important aspect is special enclosure functions for f such as mean-value forms (cf. Sect. 4.2). In our theoretical basis, enclosures are not explicitly modeled, because we can treat them as FROs $\Diamond_{f,n,m}$ with a different set $\mathcal{I}$ of inclusion functions. Therefore, they are realized in UNIVERMEC by providing a new `IVFunction` instance that wraps the original one and performs the enclosure technique for function evaluation.

Example of construction

We illustrate this behavior using the example of the mean-value forms introduced in Sect. 4.2.1. Let $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}, f \in \mathcal{C}^1(\mathcal{D})$, be defined by an expression, and $\Diamond_{f,n,m}$ an FRO for f, the inclusion functions of which are obtained by natural extension. Let further `fun` be an `IVFunction` instance that realizes $\Diamond_{f,n,m}$. A mean-value form enclosure in the framework is a new `IVFunction` object that carries out the following three steps for its inclusion functions evaluated with any of the supported arithmetics:

1. Perform an `enclosure_cast` (Def. 11) to intervals on the arguments, if necessary.
2. Apply (20) using the interval inclusion function of `fun`.
3. Perform a `forced_cast` (Def. 12) to the original argument type, which is, actually, a `lossless_cast` (Def. 10) for all arithmetics except FPA in which the range is obviously lost.

Clearly, the new `IVFunction` is the realization of another FRO for f.

Custom enclosures

Analogously to the `ICustomCon` interface, an `ICustomEncl` interface exists and can be accessed through features of an `IVFunction`. It corresponds to the custom inclusion function representation object from Def. 15 and is just a container that holds the set of additional inclusion functions.

Other Interfaces and Possible Extensions

Taylor coefficients

In addition to the above mentioned interfaces, UNIVERMEC provides the `ITaylorCoeff` interface. It is the computer realization of the Taylor coefficient representation object defined in Def. 15.

Possible extensions

Interesting candidates for the extension of the feature interface are, for example, objects for calculating slopes of a function or for its representation by a DAG that would allow for a generic implementation of techniques such as hull consistency. If these additional functionalities are to be integrated into the framework, it is necessary to alter the interface `IFFeatures` accordingly

4.4.3 *Implementation of the Function Layer*

Functors for representing functions

To make integration of functions into the framework easier, UNIVERMEC supplies standard implementations for all interfaces. These standard implementations can be applied if users provide their functions

```
struct brusselator_t
{
  brusselator_t(const core::arith::mreal &a,
                const core::arith::mreal &b)
    :m_a(b), m_b(b) {}

  template <typename T>
  T operator()(const T& args) const
    {
      T res(2);
      res[0] =  m_a + args[0]*(args[0]*args[1] - m_b - 1);
      res[1] = args[0]*(m_b-args[0]*args[1]);
      return res;
    }

  std::string text() const
    {
      return "Brusselator";
    }

  unsigned cdim() const
    {
      return 2;
    }

  unsigned dim() const
    {
      return 2;
    }

private:
  core::arith::mreal m_a;
  core::arith::mreal m_b;
};
```

Listing 3: A functor for defining a function in UniVerMeC. The concrete functor is the right-hand side of the Brusselator.

in form of a *functor*. A functor is a well-known concept in C++: It is a structure with an overloaded function `operator()`. In contrast to plain function pointers, it also stores context information as member variables.

An Introductory Example

Exploiting templates

As an example consider that the right-hand side of the Brusselator ordinary differential equation (ODE) [Sta97, p.70]

$$\begin{aligned} \dot{x} &= a + x(xy - b - 1) \\ \dot{y} &= x(b - xy) \end{aligned} \tag{35}$$

has to be represented in our framework. In this example, a and b are parameters. A possible implementation as a functor is given in listing 3. The most important member of the structure is the `operator()`, which is realized as a template function in this example. This is

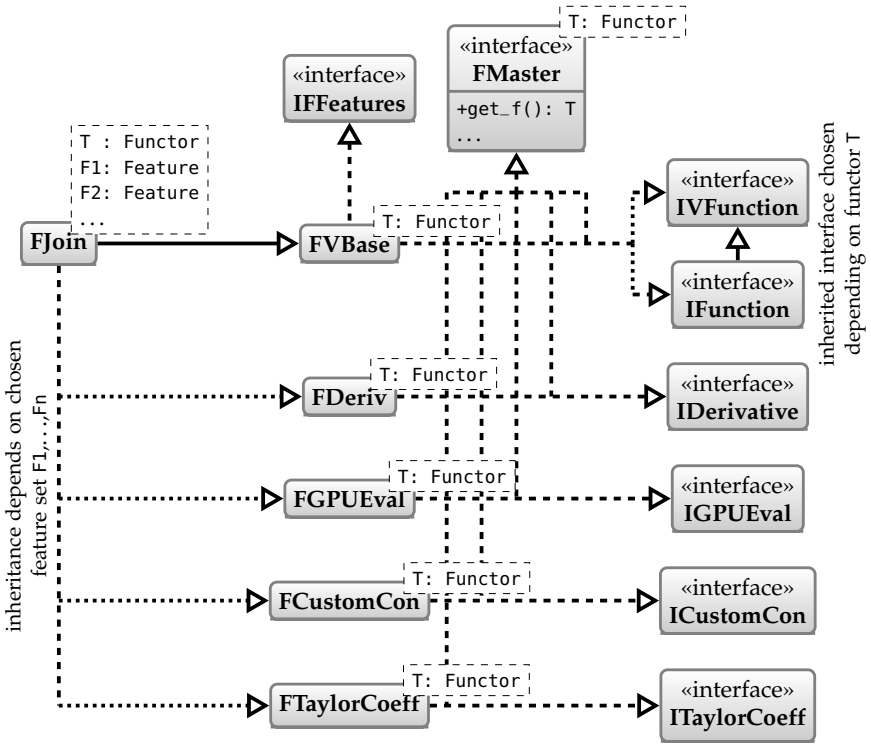


Figure 22: Simplified struture of the uniform function representation layer and its implementation in UNIVERMEC.

the recommended way because it allows for evaluation with all supported arithmetics and to provide derivatives by AD. A further recommendation is for users to restrict operation and function calls inside of the template to the set defined by the algebra (15), because natural inclusion functions for all arithmetics are constructed automatically by template instantiation in this case. This is possible only because the arithmetic layer provides a common set of functions with uniform names. It demonstrates the importance of generating uniform adapters at the arithmetic level automatically. The other members of the structure in listing 3 are just auxiliary functions that return dimension of the function's (co-)domain and a textual description for the functor.

Creating an `IVFunction` instance

With this functor available, an `IVFunction` instance can be generated straightforwardly to evaluate (35). As shown in Fig. 22, the implemention class corresponding to `IVFunction` is the `FVBase` template. To create an `IVFunction` object, it is sufficient to generate a `brusselator_t` functor with the desired parameter values for a, b and then to instantiate `FVBase` with `brusselator_t` as the template parameter:

```
IVFunction* brusselator = new FVBase<brusselator_t>
  (new brusselator_t(1.0, 1.0));
```

The resulting `IVFunction` object could then, for example, be employed to define an IVP, which can be solved with an ODE software, such as

VODE [BBH89], interfaced to our framework (cf. Sect. 7.3). Note that the actual implementation (the functor) is hidden at this point.

Adding features

The function object we have instantiated so far has an empty feature set, that is, there are no means for calculating derivatives. However, if the functor uses a template for the `operator()` and the underlying mathematical function is differentiable, support for the `IDerivative` interface can be added easily:

```
IVFunction* brusselator = new FJoin<brusselator_t,
    FDeriv<brusselator_t>, FTaylorCoeff<brusselator_t> >
  (new brusselator_t(1.0, 1.0));
```

In this case, we join `FVBase` (which is inherited by `FJoin`), `FDeriv` and `FTaylorCoeff`. Consequently, the resulting `IVFunction` instance's feature set contains the `IDerivative` and `ITaylorCoeff` interfaces. An IVP using this more sophisticated `IVFunction` object for its right-hand side could be solved, for example, with ValEncIA-IVP [RA11], which requires first-order derivatives or VNODE-LP [Ned06], which requires Taylor coefficients. Both are interfaced with out framework (cf. Sect. 7.3).

Internal Structure of the Function Layer Implementation

Allowing arbitrary feature combinations

An important aspect of the feature system and its interfaces is that arbitrary combinations of features are possible. In our default implementation, this is achieved basically by associating an individual implemention class with each interface and the ability to combine these classes through multiple inheritance. As shown in Fig. 22, this approach is implemented by the `FJoin` class template. `FJoin` inherits from `FVBase`, which in turn inherits from `IVFunction` or `IFunction`[15]. Furthermore, `FJoin` inherits from each selected feature class. If a feature is requested through the `IFFeatures` interface, the appropriately type-casted `this` pointer of the `FJoin` instance is returned.

The feature is automatically enabled or disabled during compile time using the substitution failure is not an error (SFINAE) principle based on the static type of the `FJoin` template. SFINAE means that if a type substitution of a template parameter is invalid, this is not treated as an error that stops the compilation process. This can be exploited, for example, to *activate* or *disable* functions based on conditions checked on compile time (e.g. through the `std::enable_if` template provided by the standard library in the new C++11 standard [C11b, § 20.9.7.6]). Another feature of our function layer default implementation is that all classes implementing features are decoupled not only from each other but also from `FVBase` and vice versa. That allows us to use another implementation for a feature interface together with the existing default ones. It would be even possible to

15 This decision is made automatically during compile time depending on whether the functor is scalar. A functor is treated as scalar if a function with the signature `interval operator()(const ivector &args) const` exists.

replace the `FVBase` basic class and still use the default feature implementations, if necessary.

Cross delegation

The decoupling is achieved by the `FMaster` template that is inherited by `FVBase` and all feature classes[16]. `FMaster` is an interface that provides access to an instance of the underlying functor using the `get_f()` member. This interface is only implemented by `FVBase`, which stores the actual instance of the functor. If any other feature class, for example, `FDeriv`, requests the functor using its inherited `get_f()` member, the call is forwarded to the only implementation (provided by `FVBase`) inside the final inheritance formed by `FJoin`. This pattern is sometimes referred to as "*cross delegation*" [Cli], because neither `FDeriv` nor `FVBase` know about each other. The call is *cross delegated* transparently to another branch of the (multiple) inheritance graph.

Instances vs. types

Note that we work on *functor instances* instead of *functor types*. Working only on types requires default constructors in every functor in order for each feature class to create its own instance. Alternatively, we could declare all methods `static`. However, these approaches would result in functors the behavior of which is determined at compile time by their static type, without the possibility to parametrize them further at runtime. As an illustration, consider the Brusselator example in listing 3 and its instantiation again. The static type is `FJoin<brusselator_t, FDeriv<brusselator_t>, FTaylorCoeff<brusselator_t> >`. It is determined at compile time. However, if we use instances instead of types, we are free to alter the parameters a, b in (35) at runtime freely.

Function parser

The idea can be extended further by providing a functor that acts as a parser capable of handling closed form expressions. In this case, such expressions can be entered at runtime and result in valid `IVFunction` objects. Internally, the function parser of UniVerMeC is based on `FVBase` and uses Boost.Spirit [GK] to parse the expression and to evaluate the resulting abstract syntax tree recursively. Using `FDeriv`, the resulting function provides derivatives up to a user-defined order[17] automatically. For example, an `IFunction` object for the function $f(x, y, z) = x^2 + y^2 + z^2 - 1$ can be generated using the function parser as follows:

```
IFunction *sphere =
    parser::parse_string("sqr(x0)+sqr(x1)+sqr(x2)-1");
```

FDeriv

Generation of derivative inclusion functions

As mentioned above, the `FDeriv` template is an implementation of the `IDerivative` interface. It provides derivatives for a function $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}^m, f \in \mathcal{C}^d(\mathcal{D})$, defined by a templated functor and

16 Technically, we have to use `virtual` inheritance [C11b, § 10.1] to avoid storing multiple `FMaster` instances inside one `FJoin` instance.

17 see below

makes them available through the interface. In this process, AD is used the principles of which we reviewed in Sect. 4.1 briefly. As the concrete AD implementation, we use FADBAD++ [SB] developed by Stauning and Bendtsen. It is a C++ template library, implemented by the means of operator overloading and working with arbitrary arithmetic types through trait classes. FADBAD++ provides computation of derivatives (by forward or backward differentiation) and Taylor coefficients. Because appropriate FADBAD++ trait classes for all arithmetics supported by our framework are automatically generated on the arithmetic layer (cf. Sect. 3.6), inclusion functions for derivatives can be computed for all of them. As an example consider the implicit function of the unit sphere defined by the template

```
template<typename T>
typename T::value_type operator()(const T& args)
{
  return sqr(args[0])+sqr(args[1])+sqr(args[2]) - 1;
}
```

To compute the values of derivatives with forward differentiation with FADBAD++ and IA, we have to wrap our arguments into the `fadbad::F` template, that is, to instantiate the `operator()` template with `mtl::dense_vector<fadbad::F<interval> >` as `T`. Backward differentiation can be performed by using `fadbad::B` as a replacement for `fadbad::F`.

Generation of higher-order derivatives

A distinctive feature of FADBAD++ is the ability to compute higher order derivatives by template nesting. For example, `fadbad::F<fadbad::F<interval> >` is used to obtain the second order derivative. Even mixed nesting of the forward and backward types is supported. However, the derivative order has to be decided upon at compile time because it depends on the static type[18] of the template arguments. With the help of the `FDeriv` class, users can choose the maximum derivative order at compile time for each function individually[19]. The necessary type nesting and instantiations of the template `operator()` for higher-order derivatives as well as the generation of the new `IVFunction` objects for the `grad(d,m)` and `grad_fun(d)` (see Fig. 19) are handled automatically by `FDeriv` itself. We recommend choosing a not too high maximum order because the automatically performed template nesting can increase the compilation time significantly. The default order, which is also used for functions generated through the parser, is set to 3. In our opinion, this is a reasonable setting because higher order derivatives are seldomly used in the verified context aside from Taylor coefficients, which are, however, handled by the `ITaylorCoeff` interface and its implementation `FTaylorCoeff` respectively.

18 which changes depending on the number of template nestings
19 This is realized using third template parameter.

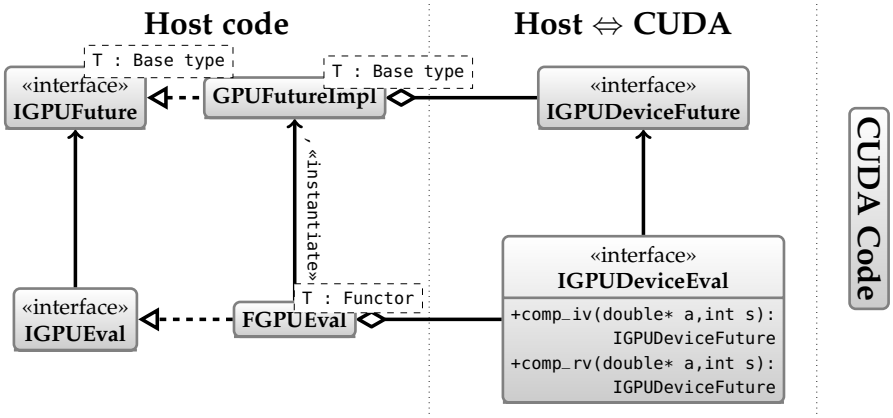


Figure 23: Interaction of the host program with the GPU kernel implementation.

FGPUEval

Memory transfer to the GPU

The GPU integration at the functions layer of UNIVERMEC consists of three parts as shown in Fig. 23. The left part, to which we refer as *host code*, provides implementations for the interfaces of the function layer. In the middle, we have *glue code*, which basically consists of two interfaces imitating the interfaces of the host part with one important difference: They do not take a list of `ivectors` or `rvectors` to process but an array of `doubles`, the type we identified in Sect. 3.7 as suitable for communication between CPU and GPU. It is important to note that the `comp_iv` and `comp_rv` functions of `IGPUDeviceEval` have to fulfill the same requirements as for the `comp` members of `IGPUEval` defined in Sect. 4.4.2. This procedure also decouples the GPU code from the CPU code and makes it independent from libraries employed on the CPU. Some of them, for example, the MATRIX TEMPLATE LIBRARY [GL], can be problematic to parse with the CUDA C compiler. The conversion to the double array is done by the `FGPUEval` class automatically, and the memory layout is configurable. For example, a list of $a^{(1)}, \ldots, a^{(m)}$ d-dimensional `mreal` vectors is converted to

$$a_1^{(1)}, \ldots, a_1^{(n)}, a_2^{(1)}, \ldots, a_2^{(n)}, a_3^{(1)}, \ldots, a_d^{(n)},$$
$$a_1^{(n+1)}, \ldots, a_1^{(2n)}, a_2^{(n+1)}, \ldots, a_d^{(2n)}, \ldots, a_d^{(m)}$$

where $n \in \mathbb{N}$. If $n = 1$ the list is simply flattened, that is, all components of a vector are stored in a row. More interesting is to choose n equal to the *blocksize* of the CUDA kernel doing the actual computation. In this case, coalesced global memory access is possible, which reduces the number of overall memory requests [Cud, pp. 70-72]. Because we do not make any assumptions about it, the actual kernel code is not shown in Fig. 23. The only requirement we pose is for it to implement the `IGPUDeviceEval` and `IGPUDeviceFuture` interfaces. However, we recommend restricting the arithmetic functions on the

Table 13: Implemented contractors and enclosures in UniVerMeC.

NAME	TYPE	DESCRIPTION
`BCContractor`	Contractor	Box consistency (Sect. 4.3.3)
`ILIEEncl`	Both	ILIE (Sect. 4.3.4)
`MeanValueForm`	Enclosure	Mean value form (20)

GPU to the sets described in Sect. 3.7. Non-blocking GPU computations for the `IGPUDeviceFuture` support can be implemented using CUDA streams [NVI12, pp. 31-34].

Example Implementations at the Function Layer

Supplied example implementations

We omit the implementation details of the further classes of the function layer and give a short overview of the implemented contractors and enclosures in UniVerMeC. The available techniques listed in Tab. 13 have been explained already earlier in this chapter. Only the classification of `ILIEEncl` as enclosure and contractor might need some clarification. If we construct an ILIE as described in Sect. 4.3.4, we automatically get an inclusion function[20] (the thick hyperplane itself). Users can access the pruning procedure (33) by a custom contractor. This allows us to utilize both enclosure and contraction features without much overhead induced by the encapsulation.

4.5 CONCLUSIONS

Foundations

In this chapter, we described our approach to representing mathematical functions in our framework for verified computations. We started by giving a short introduction to AD and interval enclosures that we rely on later. In the third section of the chapter, we defined contractors formally and discussed how this contractor notion relates to the well-established definition of a *constraint narrowing operator* by Benhamou [Ben96]. Moreover, we described several further well-known techniques, for example, interval Newton or box consistency that we use as a contractor.

Data-type independent function representation

The final section of the chapter was devoted to the handling of functions (seen as black boxes) in our framework. We started by describing the theoretical basis of our approach. First, we introduced the notion of FROs and extended their functionalities by defining several representation objects for features. After that, we showed how to implement this theoretical basis. We arrived at a generic data-type independent[21] representation of a function f by an interface

20 The range of the inclusion function is equivalent to the inclusion function obtained by AA widely with the exception of rounding errors that occur during the ILIE calculation.

21 with respect to the arithmetic

(`IVFunction`), which characterized `f` by inclusion functions. In this way, we were able to hide the internal representation completely without restricting the ways of actual implementation of functions. The strategy corresponds to the practices in modern software design that propose the wide use of encapsulation. As a second corner stone of our function representation layer, we discussed how to enrich the `IVFunction` interface with additional functionalities by feature representation objects. They are realized as a set of interfaces connected to `IVFunction`. Each of them represents and allows access to a certain functionality coupled with the underlying function `f`. In this scope, we discussed features for derivatives, GPU evaluations and contractors tailored for specific functions. This approach is made even more flexible by handling enclosure functions and derivatives as standard `IVFunction` objects. These have their own feature sets again and cannot be distinguished from "normal" `IVFunction` instances. Therefore, an algorithm, for example, box consistency, which works on an `IVFunction` instance of `f`, can be applied to derivatives of `f` without any extra effort. Additionally, algorithms can dynamically activate accelerating devices (if a function provides a corresponding feature) without additional work on the user's part.

Comparison with IBEX

In this chapter, we also discussed how to integrate standalone contractors in our framework. The provided interface (`IContractor`) allows us to mimic techniques of contractor programming [CJ09] inside UniVerMeC. However, our approach for function representation is more flexible than current implementations of contractor programming, such as IBEX [Cha]. IBEX can be used only with IA. Besides, functions inside IBEX have to be represented DAGs. Both are limitations from which our framework does not suffer. However, it is important to note that the focus of IBEX lies on constraint propagation and that many techniques in this area benefit from the DAG representation. Moreover, note that techniques based on forward/backward propagation (e.g. through operator overloading) cannot be applied to our black-box function representation. However, DAGs or contractors based on them can use the feature system of UniVerMeC.

Supplied standard implementations

Providing support for all interfaces necessary for a full integration of a function into UniVerMeC leads to a considerable amount of work. Therefore, we supply a standard implementation for all our interfaces discussed in the last part of this chapter. Users who want to employ the standard implementation need to provide only their function `f` in form of a functor with a templated[22] function operator. This template is instantiated with every arithmetic type, which generates a natural inclusion function for all arithmetics supported by the `core` layer automatically. Furthermore, it can be used to calculate derivative values by AD. Such level of automatization is only

22 The concept of entering user-specific functions through templates is well established in the verified numerical software, for example, in the packages VNODE-LP [Nedo6] or FADBAD++ [SB].

possible owing to the standardization of the set of available elementary functions for all arithmetics (cf. Sect. 3.5) and the automatically generated adapters (cf. Sect. 3.6). An interesting direction for the future research would be to make the contractors from IBEX available in UniVerMeC. This is easily possible for functions in UniVerMeC, which are generated in the way outlined above.

5 MODELING LAYER

Purpose

The purpose of the modeling layer is to express application domain dependent notions in terms of the basic concepts (e.g. real/interval quantities and functions) defined earlier. Therefore, UniVerMeC does not have a single but many independent modeling layers — one for each application domain. In this chapter, we demonstrate the use of the modeling layer in the scope of three case studies: the representation of geometric objects, initial value problems, and optimization problems. All three case studies emerged from the need to apply our framework to a certain close-to-life problem (cf. Chap. 8). Users are free to alter the supplied model representation to fit their problems. Alternatively, they can introduce new model types to open new application domains.

Structure

To begin we start with a discussion of geometric models in UniVerMeC. After that, we describe the IVP representation supplied with the framework and possibilities to integrate optimization problems into the modeling layer. Finally, we give a short outlook on potential extensions with further modeling types.

5.1 GEOMETRIC MODELS

Geometric objects

Geometric objects are important in many real-life applications. Thus, various notions to describe them exist. For an extensive overview of the different geometric modeling types, see [Ago05a, pp. 156-225]. In our applications, we need support for (non-convex) implicit objects and polyhedrons. Furthermore, we employ parametric descriptions of the implicit objects, if they are available. Limiting our understanding of geometric objects to the above ones, we can define them as below:

Definition 16 (Geometric object) *We call a bounded set $\mathcal{O} \subset \mathbb{R}^n$ geometric object if it is regular. That is, $\mathcal{O}$ is equivalent to the closure of its interior [Ago05a, p. 158]. Further, $\mathcal{O}$ is described either by*

1. *An implicit function $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}$ with $\mathcal{O} = \{x \mid x \in \mathcal{D} \wedge f(x) \leqslant 0\}$. Then $\mathcal{O}$ is called an implicit object.*
2. *A parametric function $p : \mathbb{R}^{n-1} \supset \mathcal{D} \to \mathbb{R}^n$ with $\partial\mathcal{O} = \{p(x) \mid x \in \mathcal{D}\}$. Then $\mathcal{O}$ is called a parametric object.*
3. *"The intersection of a finite number of halfplanes[1]" [Ago05b, p. 31]. Then $\mathcal{O}$ is called a convex polyhedron.*

1 With the help of a hyperplane $h(x)$ in Hesse normal form, we define a halfplane as $\{x \mid h(x) \leqslant 0\}$ [Ago05b, p. 20].

4. *"The union of finitely many convex polyhedra" [Ede95] where each of them has one common halfplane with at least one another polyhedron. Then $\mathcal{O}$ is called a non-convex polyhedron.*

The list of supported modeling types is not mandatory and can be extended in the future[2]. Convention in the rest of our work is that the normal vectors of a geometric object point outwards[3] (if they can be computed). Furthermore, we restrict the domain $\mathcal{D}$ in the case of parametric objects to a box form, that is, $\mathcal{D} \in \mathbb{IR}^{n-1}$.

Affine transformations

Usually, the geometric object representations outlined above describe the object $\mathcal{O}$ with respect to a local coordinate system. To describe objects in $\mathbb{R}^3$ that can occupy any position in space, we apply affine transformations to the local system, namely rotation and translation. In the following, we will use a 3-tuple $(\vec{t}, \vec{r}, \alpha)$ with the translation vector $\vec{t} \in \mathbb{R}^3$, the rotation axis $\vec{r}$ and the rotation angle α to describe the position of $\mathcal{O}$ in a right-handed coordinate system. Note that we can obtain a rotation matrix[4] R from the rotation axis and angle [Ago05b, pp. 112-114]. If the object is described by an implicit function f, the transformed object is described by $f_t := f(t(x))$ with the mapping $t : \mathbb{R}^3 \to \mathbb{R}^3$:

$$t(x) := R^{-1}(x - t) \tag{36}$$

where R^{-1} is the inverse of the rotation matrix[5]. In the case of a parametric function p characterizing $\mathcal{O}$, the transformed object is described by $p_t(x) := R \cdot p(x) + t$ straightforwardly.

Representation in UniVerMeC

Most concepts from Def. 16 can be easily mapped onto notions that UniVerMeC provides. We can represent both implicit functions and parametric functions in a straightforward way using the `IFunction` or `IVFunction` interfaces, respectively. This has the important side effect that inclusion functions for them are available in all arithmetics. The support of derivative computation can be assorted with the geometric notion of normal vectors that play an important role in many algorithms. Both convex and non-convex polyhedrons can be described by an in/out function. In the case of convex polyhedrons, the in/out function can be defined with the help of hyperplanes $h_i : \mathbb{R}^n \to \mathbb{R}$ in Hesse normal form describing the halfplanes of the polyhedron:

$$f(x) := \begin{cases} -1 & \text{each } h_i(x) \leqslant 0 \\ +1 & \text{otherwise} \end{cases} .$$

2 CSG objects generated using implicit objects as primitives are perhaps the first candidate for an extension (see below in this subsection).
3 The definition guarantees this only for implicit objects.
4 This matrix can be combined with the translation vector $\vec{t}$ to obtain a description suitable for the use with homogeneous coordinates.
5 It can be obtained directly by computing the matrix for $-\alpha$.

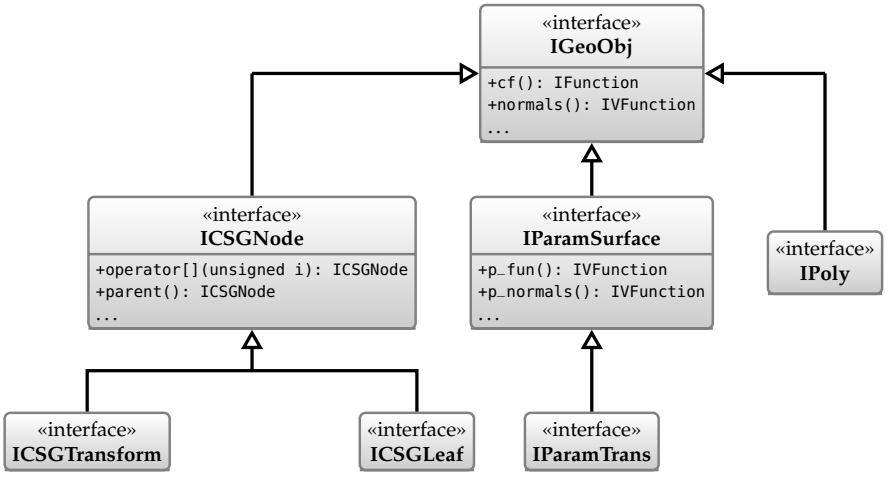


Figure 24: Structure of the geometric model representation layer (simplified).

In case of non-convex polyhedra, more complex algorithms such as ray intersection or algebraic decomposition [Ede95] are necessary.

The geometric modeling layer of UniVerMeC is shown in Fig. 24. The main interface `IGeoObj` is implemented by all geometric objects in the framework. It describes a geometric object by the means of an in/out function:

IGeoObj

Definition 17 (In/out function) *Let a geometric object $\mathcal{O} \subset \mathbb{R}^n$ (Def. 16) be given. We call $f_{inout} : \mathbb{R}^n \to \mathbb{R}$ an in/out function for $\mathcal{O}$ if*

$$x \in \mathcal{O} \iff f_{inout}(x) \leqslant 0$$

holds.

An `IFunction` instance representing an FRO for an in/out function for the geometric object represented by an `IGeoObj` can be retrieved through the `cf()` member. As usual, it provides inclusion functions for all supported arithmetics. The reasoning behind using this representation as the basis for geometric objects was that this description was available for implicit objects and polyhedrons in a straightforward way. Furthermore, all parametric objects we considered in our applications can be described as closed-form implicit expressions. For other object types such an expression can be obtained sometimes by means of implicitization [BJ05]. The `IGeoObj` interface also provides direct support for accessing normal vectors, if they are available. This is realized with the help of an `IVFunction` instance representing a function $f_{normal} : \mathbb{R}^n \to \mathbb{R}^{n+1}$, where the first member function v_1 is equal to f_{inout} and $v_2, \ldots, v_{n+1}$ calculate the n components of the normal vector. Similar to the function returned by the `cf()` member, the normals can be evaluated with every supported arithmetic.

Besides this basic interface, specialized ones are available for each supported modeling type. They allow users to access description spe-

Specialized interfaces

cific features, for example, the parametric function p for a parametric object in `IParamSurface`, or to enumerate the faces of a polyhedron in `IPoly`. Both parametric surfaces and implicit objects can act as leaves of a tree. The purpose is to associate affine transformations (e.g. rotation and translation) with an object. Additionally, the tapering and bending transformations from [JLS00, pp. 41-49] are supported.

CSG operations and R-functions

An interesting topic for our future work would be to extend the framework with respect to CSG operations for implicit primitives. As indicated by the interface names (e.g. `ICSGNode`), the layer is prepared for such an extension. However, providing CSG operations for each supported arithmetic is not an easy task. The implementation of union and intersection using the min and max functions is well-known and straightforward in the interval case (e.g. [Duf92]). However, the use of differentiable functions would be preferable for AA and TMs. A possibility is the representation of the set-theoretical operations intersection and union by R-functions [Sha07; Sha91], which are differentiable nearly everywhere. Fryazinov, Pasko, and Comninos [FPC10] provide a non-verified realization of CSG operations in AA using R-functions and the min-range approximation. In a recent master thesis [Rot12], we investigated how this approach can be integrated into UniVerMeC in a verified manner. Since several questions could not be answered in this scope, for example, how to reduce overestimation in case of the $\mathcal{C}^2$ branch of the R-functions that suffer from the dependency problem, or how to provide similar approximations for TMs, no support for CSG operations is currently provided.

5.2 INITIAL VALUE PROBLEMS

IVPs

ODEs and IVPs are important means of describing physical processes. In this thesis, we limit our discussion to the representation of the IVP type, which is needed later on for the modeling of the temperature of SOFCs (cf. Sect. 8.2). An IVP in our framework has to conform with the following definition

Definition 18 (Initial value problem) *An IVP for an ODE*

$$\dot{x} = f(x, p, u(t)) \tag{37}$$

with $f : \mathbb{R}^{|s|+|p|+|u|} \supset \mathcal{D} \to \mathbb{R}^{|s|} \in \mathcal{C}(\mathcal{D})$, $x \in \mathbb{R}^{|s|}$, $p \in \mathbb{R}^{|p|}$ *and possibly a piecewise constant* $u : \mathbb{R} \to \mathbb{R}^{|u|}$ *is a tuple* $\mathfrak{I} = (f, p, u, x_0, u_0, t_0)$*, where* $x_0 \in \mathbb{R}^{|s|}, u_0 \in \mathbb{R}^{|u|}, t_0 \in \mathbb{R}$ *are the initial values. We call* f *right-hand side of the problem,* p *parameters and* $u(t)$ *control input. Additionally, we assume that* $u(t)$ *if present is constant in each integration time-step so that we can treat* $u(t)$ *as parameter and* (37) *as autonomous.*

The resulting abstract IVP representation corresponding to the above defined tuple is depicted in Fig. 25. In UniVerMeC, it is realized in the form of the `IIVP` interface. Each `IIVP` instance is associated

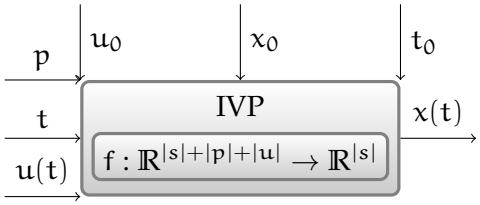


Figure 25: IVP problem type considered in UNIVERMEC.

with an underlying IVP $\mathfrak{I}$. It stores and provides access to all values of $\mathfrak{I}$. The `ivector` class is used to store parameters and initial values. In this way, uncertainty can be represented if necessary[6]. Both $f(x, p, u(t))$ and $u(t)$ are realized as an `IVFunction`. This allows for evaluating them with verified and non-verified arithmetics and thus interface different solver types. Using the feature system of the function, it is also possible to represent derivatives and Taylor coefficients, of the right-hand side, which is important for solvers, such as VNODE-LP [Ned06] and VALENCIA-IVP [RA11], used later on.

5.3 OPTIMIZATION PROBLEMS

Optimization problems

The third type of problem we consider here as an example of real-world applications of our framework are from the area of optimization. They play an important role in many fields of applied science and are used, for example, to determine the *best* (or at least a *good*[7]) set of parameters for a complex system. Usually the goal of solving an optimization problem is to determine such an $x^* \in \mathbb{R}^n$ that a function $\varphi : \mathbb{R}^n \to \mathbb{R}$ attains its minimum. Often x^* has to fulfill certain constraints. More formally, we define an optimization problem by:

Definition 19 (Optimization problem) *An optimization problem*

$$\begin{aligned} &\min_{x \in \mathcal{D}} \varphi(x) \\ &g_i(x) \leqslant 0 \text{ for } i = 1, \ldots, m \end{aligned} \tag{38}$$

consists of a scalar objective function $\varphi : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}$ *and* j *constraints* $g_i : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}$.

The above definition is limited to a special class of optimization problems called *inequality constrained problems*. Our definition and representation scheme can be extended to the more general class of *equality constrained* problems by supplying k equality constraints $h_i(x) = 0$ with $i = j + 1, \ldots, k$. In both cases, the integration of the problem description into our framework is straightforward. The objective functions and the constraints are represented using `IFunction`. This not

6 Otherwise, point intervals can be employed.
7 Depending on whether local or global optimization is performed.

only ensures access to various arithmetics (e.g., for verified and non-verified solvers) but also allows for computing derivatives of the first and the second order that are often required in the scope of optimization (e.g. for Karush-Kuhn-Tucker (KKT) conditions [BV09, pp. 243-246]).

5.4 FURTHER PROBLEM TYPES

Integration of new types

The previous sections of this chapter outlined how different problem types can be integrated into our framework. While the discussed types cover a large range of problem domains, they are not exhaustive. In general, problems that can be described using functions[8] and relational dependencies are well-suited for integration in the modeling layer. It might be more complicated to integrate new models if they work on functions in a way that interferes with the black box nature of our function representation `IVFunction`. For example, the `ICSGTransform` class realizes affine transformations on implicit objects by providing a new `IVFunction` instance which uses formula (36). While in this case the implementation of the inclusion functions is straightforward, special care needs to be taken of features, such as derivatives. If the original function is differentiable and the derivatives are obtained by the means of AD, using AD is not possible for the new wrapping `IVFunction` instance, because the inner details of the original function are hidden in a black box. Instead, the chain rule has to applied manually to provide derivatives, as it is done already in the `FLinChainRule` class supplied by the framework for the linear cases.

Combination of existing types

Another interesting extension of the framework is to combine already integrated modeling types. Parameter identification (cf. Sect. 8.2) that can be treated as an optimization problem is an example application in which such a combination is useful. Often the system for which the parameters need to be identified is described by means of differential equations. In this case, each evaluation of the objective function of the parameter identification problem requires solving an IVP. This combination of different modeling types poses additional challenges. For example, optimization algorithms perform better if derivatives are available. In case of the scenario outlined above, where the objective function depends on the solution to an IVP, derivatives can be obtained by solving a sensitivity problem [Kie+11]. A sensitivity problem is in turn an IVP that depends on the derivatives of the right-hand side of the original problem. Because UniVerMeC is capable of handling higher-order derivatives, the extension of the IVP modeling layer with sensitivity problems can be done in a straightforward way.

8 For which meaningful inclusion functions in the supported arithmetics are available.

6 HIERARCHICAL SPACE DECOMPOSITION

Goals

The goal of this chapter is first to give an overview on hierarchical space decomposition structures and to describe the novel interval trees developed in the scope of this thesis. Such structures are often employed as utility data types in the context of branch and bound algorithms. This chapter consists of two sections: In the first section, we discuss *n-trees* and in the second section *general multisections*. In this thesis we denote by n-trees a special data structure that is bound to a geometric object in contrast to general multisections. N-trees classify[1] a box-shaped region of space with respect to this object.

Interval trees

Hierarchical space decomposition structures are well-known in the context of geometric computations. Quadtrees and octrees [Sam06; Sam90] are the most common ones for two- or three-dimensional data respectively. They are special cases of n-trees. We use a verified extension of them which we call *interval trees*. The variant in this thesis is a refinement of the work by Dyllong and Grimm [DG07d; DG08]. Interval trees are associated with a geometric object $\mathcal{O}$ and subdivide recursively the user-defined starting box into subboxes with respect to $\mathcal{O}$. Each subbox has a color assigned to it, which indicates whether it belongs to $\mathcal{O}$ (black), is disjoint with $\mathcal{O}$ (white), or could not be classified at the current subdivision level (gray). In contrast to the work by Dyllong and Grimm, the focus of this work lies not on the adaptive handling of CSG models but on the handling of different modeling types. Additionally, we provide several extensions, such as the integration of contractors into the trees.

Subpavings

Jaulin et al. [Jau+01, pp. 48-51] introduced a technique similar to n-trees under the name *subpaving*. Its purpose is also the covering of a certain set $\mathcal{S}$ by a non-overlapping collection of interval boxes. Jaulin et al. distinguish between inner and outer approximations, that is, the set $\mathcal{S}$ is approximated by two subpavings $\mathcal{P}_1, \mathcal{P}_2$ of $\mathcal{S}$ where $\mathcal{P}_1 \subseteq \mathcal{S} \subseteq \mathcal{P}_2$. Because the set $\mathcal{S}$ can be related to the geometric object $\mathcal{O}$ associated to an n-tree, $\mathcal{P}_1$ corresponds to the union of all black colored boxes, and $\mathcal{P}_2$ to the union of all boxes associated with black and gray leaf nodes of the tree. Because of this, subpavings and n-trees can be treated as largely equivalent. Especially it is possible to store subpavings using binary trees [Jau+01, pp. 51-52]. The main difference is that the inner and outer approximations are separate entities while n-trees generate both implicitly in one data structure.

General multisections

The second part of the chapter discusses schemes for multisection and memory management for storing the generated subboxes. In

1 Classify, in this sense means to determine whether a box belongs to the object.

contrast to the n-trees, no hierarchical structure that covers the whole search space is created in this case. Instead, heuristically sorted lists store the relevant parts of the search space. Usually, these lists are generated by a multisection scheme that depends on a heuristic for choosing relevant coordinate directions and subdividing points. Because the multisection layer is currently used only by the global optimization algorithm, we limit our discussion of schemes to several well-known techniques from global optimization at this point. Besides, we provide a generic basis that can be extended with further schemes in the future if necessary.

6.1 INTERVAL TREES

In this section, we introduce our notion of *n-trees*. While the formal definition we give in the first subsection includes the classic quad- and octrees, the actual trees with which we work later on are binary trees. Using binary trees makes the integration of certain improvements, such as inversion nodes, easier. We explain the details of the improved structures in Sect. 6.1.2. Our way of handling parametric surfaces with interval trees is introduced in Sect. 6.1.3. Finally, the section is concluded with a short discussion of our actual software implementation in Sect. 6.1.4.

6.1.1 *Formal Definition and Standard Trees*

Trees

As already described an interval tree subdivides the space recursively into colored subboxes. To formalize this notion, we give a definition of a tree first:

Definition 20 (Tree [Knu97, p. 308]) *A tree is a "finite set $\mathcal{T}$ of one or more nodes such that*

1. *there is one specially designated node called the root of the tree,* $\operatorname{root}(\mathcal{T})$*; and*
2. *the remaining nodes (excluding the root) are partitioned into $m \geqslant 0$ disjoint sets $\mathcal{T}_1, \ldots, \mathcal{T}_m$, and each of these sets in turn is a tree. The trees $\mathcal{T}_1, \ldots, \mathcal{T}_m$ are called subtrees of the root."*

Following Knuth [Knu97, p. 311], we call "each root [...] to be the *parent* of the roots of its subtrees and the latter [...] are *children* of their parent". If a subtree $\mathcal{T}_i$ has zero children, we call $\operatorname{root}(\mathcal{T}_i)$ a *leaf node*. Otherwise, it is called an *inner node*. The set of subtrees of tree $\mathcal{T}_i$ is denoted with $\operatorname{ch}(\mathcal{T}_i)$. Furthermore, we abbreviate the root node $\operatorname{root}(\mathcal{T}_i)$ of a subtree $\mathcal{T}_i$ with $\mathfrak{T}_i$.

Interval trees

As mentioned previously, our interval trees resemble those ideas introduced by Dyllong and Grimm [DG07d; DG08]. Here, we first

provide a formal basis covering the ideas from their paper. This definition can be used to describe new tree types introduced later on:

Definition 21 (Tree type) *A n'-tree[2] type $\mathfrak{TT}$ is a 4-tuple $(\mathcal{C}, \psi, \text{split}, \mathcal{M})$ where*

1. *$\mathcal{C} = (C_1, \ldots, C_q)$ is a set of colors*
2. *$\psi : \mathbb{T}_{\mathfrak{TT}} \to \mathcal{W} \subset \mathbb{R}^n$ is an interpretation mapping*
3. *$\text{split} : \mathbb{T}_{\mathfrak{TT}} \to \mathbb{T}^{n'}_{\mathfrak{TT}}$ is a splitting function*
4. *$\mathcal{M}$ is the set of all possible meta-information (stored in the tree nodes)*

and $\mathbb{T}_{\mathfrak{TT}}$ is the set of all n'-trees of type $\mathfrak{TT}$.

The purpose of the tree type is to provide a uniform mechanism for supporting an extended set of node colors. Additionally, the interpretation mapping can be used to associate regions that do not have box form with a tree node. Each interval tree has a unique tree type that allows for interpreting the date stored in the tree:

Definition 22 (Interval tree) *An interval tree $\mathfrak{IT}$ is a tuple $(\mathcal{T}, \mathcal{O}, \mathfrak{O}_{f,n,1}, \mathfrak{TT})$ where $\mathcal{T}$ is a tree (cf. Def. 20), $\mathcal{O} \subset \mathbb{R}^n$ is a geometric object (cf. Def. 16), $\mathfrak{O}_{f,n,1}$ is an FRO (cf. Def. 14) for the in/out function (cf. Def. 17) f of $\mathcal{O}$, and $\mathfrak{TT}$ is the tree type of $\mathfrak{I}$. Each of the subtrees $\mathcal{T}_1, \ldots, \mathcal{T}_m$ must have either zero or n' children. Each node of the interval tree is a 5-tuple $(\mathbf{x}, C, \mathfrak{P}, \mathcal{N}, M)$. Here, $\mathbf{x} \in \mathbb{IR}^n$ is an interval vector, $C \in \mathfrak{TT}.\mathcal{C}$ is the color of the node, $\mathfrak{P}$ is the parent of this node, $\mathcal{N}$ contains the n' children of the node or is empty and $M \in \mathfrak{TT}.\mathcal{M}$ the meta-information of the node. If $\mathfrak{TT}.\mathcal{M}$ is the empty set no meta-information is stored.*

The area associated with a node is determined by the interpretation mapping ψ of the tree type $\mathfrak{TT}$. It depends on the actual interpretation mapping whether we call a tree regular:

Definition 23 (Regular interval tree) *We call an interval tree regular if the following conditions are fulfilled:*

1. $\psi(\mathfrak{T}_i) = \bigcup_{\mathcal{T}_j \in \text{ch}(\mathcal{T}_i)} \psi(\mathfrak{T}_j)$;
2. $\mathring{\psi}(\mathfrak{T}_j) \cap \mathring{\psi}(\mathfrak{T}_k) = \emptyset$ *for all* $\mathcal{T}_j, \mathcal{T}_k \in \text{ch}(\mathcal{T}_i), j \neq k$ *where* $\mathring{\psi}(\mathfrak{T})$ *the interior of* $\psi(\mathfrak{T})$;
3. $\psi(\mathfrak{T}_j)$ *is a compact set for* $\forall \mathcal{T}_j \in \text{ch}(\mathcal{T}_i)$.

Standard interval trees

Now, we are ready to define the type $\mathfrak{TT}_{def}$ of standard interval trees (which are similar to the commonly used quad- and octrees)

2 Beginning from here, we use n'-tree instead of n-tree, because the identifier n is used for the dimension of geometric objects in this context.

and their interval tree extensions, by Dyllong and Grimm [DG07d; DG08]. The main difference between their and our trees is that we do not split each component of a box at every level. Instead, similarly to subpavings by Jaulin et al. [Jau+01, pp. 48-51], we only use one bisection per level. We define the default interpretation mapping $\psi_{def} : \mathbb{T}_{def} \to \mathbb{S} \subset \mathbb{R}^n$ for our standard trees as:

$$\psi_{def}(\mathcal{T}_i) := \mathfrak{T}_i.\boldsymbol{x} \ . \tag{39}$$

This means that the area associated with an interval tree node is the box $\boldsymbol{x}$ stored inside this node. Furthermore, the default color set is $\mathcal{C}_{def} := \{\text{black}, \text{gray}, \text{white}\}$. Initially, any node in the tree is colored *gray*, that is, we have no knowledge about how the area associated with a node and the object $\mathcal{O}$ are related. The meaning of black and white node colors are defined below:

Definition 24 (Standard node colors) *Let $\mathfrak{N}$ be a tree node of the tree $\mathfrak{T}$. Then setting the color of $\mathfrak{N}$ to*

WHITE *implies that* $\mathfrak{T}.\mathfrak{TT}.\psi(\mathfrak{N}) \cap \mathfrak{T}.\mathcal{O} = \emptyset$;

BLACK *implies that* $\mathfrak{T}.\mathfrak{TT}.\psi(\mathfrak{N}) \subseteq \mathfrak{T}.\mathcal{O}$;

GRAY *has no implication.*

That is, the area associated with a white node has no intersection with $\mathcal{O}$. The area associated with a black node is a subset of $\mathcal{O}$. One way to check whether a node with the associated area $\boldsymbol{x}$ can be colored black or white is to use the interval inclusion function F for the in/out function f of the geometric object $\mathcal{O}$ provided by the FRO $\mathfrak{O}_{f,n,1}$:

$$\text{color}(\boldsymbol{x}) := \begin{cases} \text{black} \ , & \text{if } \overline{F(\boldsymbol{x})} \leqslant 0 \ , \\ \text{white} \ , & \text{if } \underline{F(\boldsymbol{x})} > 0 \ , \\ \text{gray} \ , & \text{otherwise} \ . \end{cases} \tag{40}$$

Owing to overestimation during evaluation of F, the color function could label nodes gray even if they are in reality black or white. This does not introduce wrong information into the tree structure, but enlarges uncertain areas where a decision needs to be made at a deeper subdivision level. Next, we define the default split operation, which corresponds to the simple bisection algorithm shown in Alg. 1. It subdivides $\boldsymbol{x}$ in the middle of its widest component into two parts and returns a set of nodes covering both parts. To obtain the color of the new nodes, the function (40) is used. To conclude, we define the type of standard interval trees as $\mathfrak{TT}_{def} := (\mathcal{C}_{def}, \psi_{def}, \text{split}_{def}, \emptyset)$ where split_{def} is given by Alg. 1 and the set of meta-information is empty.

Other standard trees

In the above defined standard interval trees, we used IA to bound the range of the in/out function of the underlying geometric object

Algorithm 1: Split operation for a standard interval tree node.

Data: Tree $\mathfrak{IT}$, Interval tree node $\mathfrak{N} = (\mathbf{x}, C, \mathfrak{P}, \mathcal{N})$ of $\mathfrak{IT}$
Result: Set of nodes $\mathcal{R}$
$\mathbf{x} \coloneqq \mathfrak{N}.\mathbf{x}$; Determine the widest component i of $\mathbf{x}$;
$\mathbf{x}^{(0)} \coloneqq (\mathbf{x}_1, \ldots, [\underline{x_i}, \operatorname{mid} \mathbf{x}_i], \ldots, \mathbf{x}_n)$;
$\mathbf{x}^{(1)} \coloneqq (\mathbf{x}_1, \ldots, [\operatorname{mid} \mathbf{x}_i, \overline{x_i}], \ldots, \mathbf{x}_n)$;
$\mathcal{R} \coloneqq \left\{ \left(\mathbf{x}^{(0)}, \texttt{color}(\mathbf{x}^{(0)}), \mathfrak{N}, \emptyset\right) \cup \left(\mathbf{x}^{(1)}, \texttt{color}(\mathbf{x}^{(0)}), \mathfrak{N}, \emptyset\right) \right\}$;
$\mathfrak{N}.\mathcal{N} \coloneqq \mathcal{R}$; **return** $\mathcal{R}$;

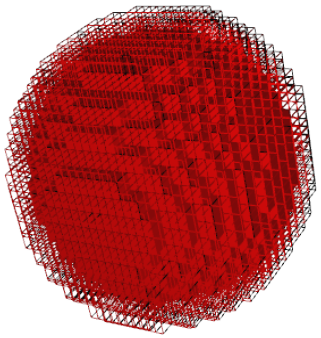


Figure 26: Decomposition of a sphere using a binary tree. The uncertain area is shown by the grid boxes.

in the `color` function (40). Sometimes the use of more sophisticated range bounding techniques, such as AA, might improve the quality of the decomposition, that is, reduce the number of gray leaf nodes. We use AA straightforwardly in our trees by first converting the box $\mathbf{x}$ associated with the current node to an affine form using (11). Then, the node color can be obtained by using an affine inclusion function of the in/out function instead of an interval inclusion function. This affine extension of the function is readily available because the function is represented by an FRO. We call this tree type *standard affine tree*. Similarly, the *standard Taylor model tree* and the *standard mean-value form tree* can be obtained. All standard trees conform with Def. 23.

6.1.2 *Contracting Trees*

Uncertain area

As explained previously, an interval tree $\mathfrak{T}$ with the associated geometric object $\mathcal{O} \subset \mathbb{R}^n$ can be interpreted both as its inner and outer approximation $\mathcal{P}_1 \subseteq \mathcal{O} \subseteq \mathcal{P}_2$. In theory, it is possible to approximate $\mathcal{O}$ arbitrarily precisely [Jau+01, p. 50] using axis-aligned boxes. However, in practice, the set $\mathcal{G} \coloneqq \mathcal{P}_2 \setminus \mathcal{P}_1$ is not infinitesimally small because we cannot use arbitrarily small boxes for approximation due to memory and runtime constraints (cf. Fig. 26). $\mathcal{G}$ is the union of the areas associated with the gray leaf nodes of $\mathfrak{T}$. We call it *uncertain area*, because we cannot decide which parts of it actually belong to $\mathcal{O}$.

Overestimation

There are several further sources of uncertainty aside from the use

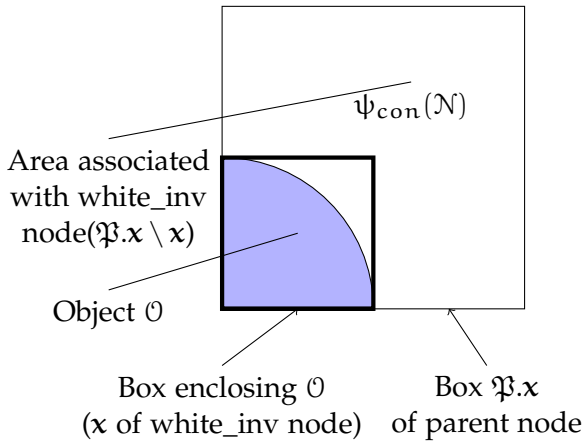


Figure 27: After pruning the uncertainty with the contractor, we proved that the area not covered by the box $\mathbf{x}$ (bold rectangle) does not belong to $\mathcal{O}$. This area is associated with the white_inv node $\mathfrak{N}$.

of axis-aligned boxes. For example, if a real-world object is captured with a 3D scanner, the resulting point-cloud can be encoded with an octree leading to inner (black), outer (white), and in-between (uncertain) regions on the object's boundary. In this case, the uncertain areas are caused not only by the octree discretization but also by scanner errors. Their width depends on various factors, such as the distribution and magnitude of these errors. Another way to cope with such point clouds is to fit them with analytic descriptions (e.g., splines, SQs). In UniVerMeC, we assume that such an analytic description of a geometric object is available. If the object is described, for example, by an in/out function f using the `color` function (40) to set the node colors, additional *evaluation uncertainty* is introduced because the true range of f is overestimated during the interval evaluation. The goal of this subsection is to describe techniques to reduce *this kind of uncertainty*. One way is to replace IA by more sophisticated range enclosure techniques, such as AA, TMs, or mean-value forms as discussed in the previous subsection. Alternatively, it is possible to increase the subdivision depth, which in turn also reduces the uncertainty induced by the use of axis-aligned boxes.

Contraction

The contracting tree tries to reduce the overestimation even further by incorporating contractors into trees. It is a refinement of the *LIETree* structure that we presented in [Kie12a]. In contrast to the LIETree, which used only ILIEs (cf. Sect. 4.3.4) for contraction, the contracting tree can employ an arbitrary number of contractors and enclosure techniques during the subdivision process. If a geometric object $\mathcal{O}$ is characterized by an in/out function $f : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}$, each point $x \in \mathcal{O}$ has to satisfy $f(x) \leqslant 0$. If $\mathbf{x} \in \mathbb{IR}^n$ is the area associated with the current node $\mathfrak{N}$, we can apply a contractor A on f and $\mathbf{x}$ to obtain the contracted box $\mathbf{x}^{(1)} \subseteq \mathbf{x}$. Owing to the definition of A (cf. Def. 13) the set, $\mathcal{S} := \mathbf{x} \setminus \mathbf{x}^{(1)}$ is disjoint with $\mathcal{O}$. However, if $\mathfrak{N}$ is split,

we cannot create a new standard white node covering $\mathbb{S}$ under the default interpretation mapping (39) because $\mathbb{S}$ is not necessarily a box anymore.

Inversion nodes

We use an extended set of colors $\mathcal{C}_{con} \coloneqq \mathcal{C}_{def} \cup \{\text{white\_inv}\}$ for trees of the contracting tree type $\mathfrak{TT}_{con}$ to cover $\mathbb{S}$. If a node has the color white_inv, it implies the same conditions as with a standard white node:

Definition 25 (Node color white_inv) *Let $\mathfrak{N}$ be a node of the interval tree $\mathfrak{T}$. Then, $\mathfrak{N}.C = \text{white\_inv}$ means that $\mathfrak{T}.\mathfrak{TT}.\psi(\mathfrak{N}.\mathbf{x}) \cap \mathfrak{T}.\mathcal{O} = \emptyset$ holds.*

The area associated with a node with the color white_inv can be computed using the following interpretation mapping:

$$\psi_{con}(\mathcal{T}_i) \coloneqq \begin{cases} \psi_{def}(\mathcal{T}_i) \ , & \text{if } \mathfrak{T}_i.C \in \mathcal{C}_{def} \ , \\ \psi_{con}(\mathfrak{T}_i.\mathfrak{P}.\mathbf{x}) \setminus \mathbf{x} \ , & \text{otherwise} \end{cases}$$

where $\mathfrak{T}_i.\mathfrak{P}.\mathbf{x}$ is the box associated with the parent[3] of $\mathfrak{T}_i.\mathfrak{P}$. We call nodes with the color white_inv *inversion nodes* because they cover the area associated with their parent node that is *not* included in their own box and, therefore, *invert* $\mathbf{x}$ with respect to the parent's area (cf. Fig. 27). Inversion nodes can cover sets obtained using interval contractors completely.

Measuring progress

To construct a contracting tree, we need a split operation, that is, an operation that subdivides a box into subboxes. Our split operation does not add an inversion node each time the set $\mathbb{S}$ is nonempty. Instead, we wait until $\mathbb{S}$ is sufficiently large, which means until the contraction process leads to some progress. To determine whether an inversion node is necessary, we use a heuristical formula from global optimization proposed by Hansen and Walster [HW04, pp. 255-256]:

$$d \coloneqq 0.25 \operatorname{wid}(\mathbf{x}) - \max_{1 \leqslant i \leqslant n} (\operatorname{wid} \mathbf{x}_i - \operatorname{wid} \mathbf{h}_i) \leqslant 0 \ , \tag{41}$$

where $\mathbf{h} \subseteq \mathbf{x}$ is the contracted box. If $d \leqslant 0$, the progress was sufficient, and we create an inversion node to cover $\mathbb{S}$. Otherwise, we store $\mathbf{h}$ for further use during the subdivision process inside the meta-information of the node. Therefore, we set $\mathfrak{TT}.\mathcal{M} \coloneqq \mathbb{IR}^n$, that is, a contracting tree node is a 5-tuple $(\mathbf{x}, C, \mathfrak{P}, \mathcal{N}, \mathbf{h})$ with $\mathbf{h} \subseteq \mathbf{x}$ where $\mathbf{x} \setminus \mathbf{h} \cap \mathcal{O} = \emptyset$. This ensures that information obtained during the contraction is not lost during the subdivision procedure, but can be stored in $\mathbf{h}$.

Split algorithm

The splitting procedure for the contracting tree nodes is carried out according to Alg. 2. It takes the node to be split, a list of inclusion functions $\mathcal{LI}$ for the in/out function of the object $\mathcal{O}$, a list of contractors $\mathcal{LA}$, and the in/out function[4] of $\mathcal{O}$ as arguments. First we try

3 Note that we can omit the interpretation mapping for the parent because the parent has to be a gray node.

4 In practice, this is one of the inclusion functions in $\mathcal{LI}$.

Algorithm 2: Splitting operation for a contracting tree node.

Data: Node $\mathfrak{N} = (\mathbf{x}, C, \mathfrak{P}, \mathcal{N}, \mathbf{h})$, List of contractors $\mathcal{LA}$, List of inclusion functions $\mathcal{LI}$, in/out function f of $\mathcal{O}$
Result: Set of nodes $\mathcal{R}$
$\mathcal{R} = \emptyset$; $\boldsymbol{e} \coloneqq (-\infty, \infty)$;
2 **forall the** $e \in \mathcal{LI}$ **do**
  $\boldsymbol{e} \coloneqq e(\mathfrak{N}.\mathbf{h}) \cap \boldsymbol{e}$;
4 **if** $\overline{e} > 0$ **then** $\mathfrak{N}.C \coloneqq$ white;
5 **else if** $\underline{e} \leqslant 0$ **then**
    if $\mathfrak{N}.\mathbf{x} == \mathfrak{N}.\mathbf{h}$ **then** $\mathfrak{N}.C \coloneqq$ black;
    else
     $\mathcal{R} \coloneqq \{(\mathfrak{N}.\mathbf{h}, \text{white\_inv}, \mathfrak{N}, \emptyset, \mathfrak{N}.\mathbf{h})\}$;
     $\mathcal{R} \coloneqq \mathcal{R} \cup \{(\mathfrak{N}.\mathbf{h}, \text{black}, \mathfrak{N}, \emptyset, \mathfrak{N}.\mathbf{h})\}$;
  if $\mathfrak{N}.C \neq \text{gray} \vee \mathcal{R} \neq \emptyset$ **then** $\mathfrak{N}.\mathcal{N} \coloneqq \mathcal{R}$; **return** $\mathcal{R}$;
forall the $A \in \mathcal{LA}$ **do**
12 Apply A on f and $\mathfrak{N}.\mathbf{h}$ to obtain $\mathbf{h}^{(c)}$; $\mathfrak{N}.\mathbf{h} \coloneqq \mathbf{h}^{(c)}$;
13 **if** $\mathfrak{N}.\mathbf{h} == \emptyset$ **then** $\mathfrak{N}.C \coloneqq$ white;
  else
    Calculate d using (41);
16 **if** $d \leqslant 0$ **then**
     $\mathcal{R} \coloneqq \{(\mathfrak{N}.\mathbf{h}, \text{white\_inv}, \mathfrak{N}, \emptyset, \mathfrak{N}.\mathbf{h})\}$;
     $\mathcal{R} \coloneqq \mathcal{R} \cup \{(\mathfrak{N}.\mathbf{h}, \text{gray}, \mathfrak{N}, \emptyset, \mathfrak{N}.\mathbf{h})\}$;
  if $\mathfrak{N}.C \neq \text{gray} \vee \mathcal{R} \neq \emptyset$ **then** $\mathfrak{N}.\mathcal{N} \coloneqq \mathcal{R}$; **return** $\mathcal{R}$;
20 Determine the widest component i of $\mathfrak{N}.\mathbf{h}$;
Bisect $\mathfrak{N}.\mathbf{x}$ using the component i and mid $\mathbf{h}_i$ into $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}$;
Bisect $\mathfrak{N}.\mathbf{h}$ using the component i and mid $\mathbf{h}_i$ into $\mathbf{h}^{(0)}, \mathbf{h}^{(1)}$;
$\mathcal{R} \coloneqq \mathcal{R} \cup \left\{ \left(\mathbf{x}^{(0)}, \text{gray}, \mathfrak{N}, \emptyset, \mathbf{h}^{(0)}\right), \left(\mathbf{x}^{(1)}, \text{gray}, \mathfrak{N}, \emptyset, \mathbf{h}^{(1)}\right) \right\}$;
$\mathfrak{N}.\mathcal{N} \coloneqq \mathcal{R}$; **return** $\mathcal{R}$;

to determine the node color of the node to be split by using each of the inclusion functions in $\mathcal{LI}$ in a loop (line 2). If a node color other than gray can be determined, it is assigned to $\mathfrak{N}$ and the algorithm terminates (lines 4; 5). Otherwise, each contractor in $\mathcal{LA}$ is applied to shrink $\mathbf{h}$ (line 12). If the resulting $\mathbf{h}$ is empty, the whole node $\mathfrak{N}$ is disjoint with $\mathcal{O}$ and can be colored white (line 13). If this is not the case, we check the criterion (41) to determine whether the progress was sufficient. In this case, a new inversion node is created (line 16). Finally, if no contractor leads to sufficient progress, a normal bisection is performed (line 20). To summarize, the tree type of a contracting tree is $\mathfrak{TT}_{con} \coloneqq \left(\mathcal{C}_{con}, \psi_{con}, \text{split}_{con}, \mathbb{IR}^n\right)$ where split_{con} is defined by Alg. 2. Trees of type $\mathfrak{TT}_{con}$ are regular.

Improvements

The outlined splitting procedure can be improved in some ways. For example, in our implementation, it is possible to assign priorities to the used enclosures and contractors. Another way to improve the contracting tree is to extend the idea of white inversion nodes to

Algorithm 3: Conversion of a white inversion node to a set of standard white nodes.

Data: white_inv node $\mathfrak{N}$ of tree $\mathfrak{T}$, Subpaving $\mathcal{P}$ covering $\mathfrak{T}.\mathfrak{TT}.\psi(\mathfrak{N})$
Result: New parent node of $\mathfrak{N}$ without white_inv children
$\mathfrak{P} \coloneqq \mathfrak{N}.\mathfrak{P}$; $\mathfrak{R} \coloneqq \mathfrak{P}$;
$\mathfrak{O} \coloneqq \mathfrak{P}.\mathcal{N} \setminus \mathfrak{N}$;
while $\mathcal{P} \neq \emptyset$ **do**
4 | **while** $\neg$`is_box`$(\mathfrak{P}.\mathbf{x} \setminus$ `head`$(\mathcal{P}))$ **do**
| | $\mathcal{P} \coloneqq$ `concat(tail(`$\mathcal{P}$`), head(`$\mathcal{P}$`))`;
6 | $\mathfrak{P}_{new} \coloneqq$ (`head(`$\mathcal{P}$`)`, white, $\mathfrak{P}, \emptyset$);
| $\mathfrak{P}.\mathcal{N} \coloneqq \{\mathfrak{P}_{new}\}$;
8 | **if** `tail(`$\mathcal{P}$`)` $\neq \emptyset$ **then**
| | $\mathfrak{P}.\mathcal{N} \coloneqq \mathfrak{P}.\mathcal{N} \cup \{(\mathfrak{P}.\mathbf{x} \setminus$ `head(`$\mathcal{P}$`)`, gray, $\mathfrak{P}, \emptyset)\}$;
10 | **else** $\mathfrak{P}.\mathcal{N} \coloneqq \mathfrak{P}.\mathcal{N} \cup \{\mathfrak{O}\}$;
| $\mathcal{P} \coloneqq$ `tail(`$\mathcal{P}$`)`;
| $\mathfrak{P} \coloneqq \mathfrak{P}_{new}$;
return $\mathfrak{R}$;

black ones. In this case, we have to invert the relation and apply a contractor on $f(x) > 0$. A practical problem here is that this inequality has to be rewritten as the equation $f(x) = (0, \infty)$ and the open interval arising cannot be handled straightforwardly in our framework. A possible implementation is to apply a contractor A on $\mathbf{x}$ and $f(x) = [\epsilon_M, \infty)$ to obtain $\mathbf{x}^{(1)}$ where $\epsilon_M \in \mathbb{F}$ is the smallest positive number in the FP system. In this case, we have proven that the set $\mathbb{S}' \coloneqq \mathbf{x} \setminus \mathbf{x}^{(1)}$ is a subset of $\mathcal{O}$. We can associate $\mathbb{S}'$ with a node colored black_inv, which can be defined analogously to white inversion nodes.

Conversion to standard trees

An important feature of the contracting tree is that it is possible to convert an inversion node to a set of standard nodes. After construction we can transform a contracting tree into a standard tree without the new color types, if this is required by an algorithm using the tree structure. Algorithm 3 describes this procedure for a white inversion node. The algorithm takes the inversion node $\mathfrak{N}$ and a subpaving $\mathcal{P}$ covering the area associated with $\mathfrak{N}$ by a minimum number of boxes. It can be obtained by a box inversion algorithm, such as that given in [Kea96, pp. 154-155]. In line 4, the `is_box` predicate checks whether a set is a box. This is necessary to ensure that the parts of the space not associated with the new standard white node (line 6) can be covered by a standard gray node in line 8. After processing the parts of the space associated with the node $\mathfrak{N}$, we add its sibling to the tree (line 10). The returned new root node $\mathfrak{R}$ replaces the parent of $\mathfrak{N}$.

Algorithm 4: Split operation for a parametric tree node.

Data: Parametric tree node $\mathfrak{N} = (\mathbf{x}, C, \mathfrak{P}, \mathcal{N}, \mathbf{u})$, Inclusion function P of $p : \mathbb{R}^{n-1} \to \mathbb{R}^n$
Result: Set of nodes $\mathcal{R}$
$\mathbf{u} \coloneqq \mathfrak{N}.\mathbf{u}$; Determine longest axis i of $\mathbf{u}$;
$\mathbf{u}^{(1)} \coloneqq (\mathbf{u}_1, \ldots, [\underline{u_i}, \operatorname{mid} \mathbf{u}_i], \ldots, \mathbf{u}_{n-1})$;
$\mathbf{u}^{(2)} \coloneqq (\mathbf{u}_1, \ldots, [\operatorname{mid} \mathbf{u}_i, \overline{u_i}], \ldots, \mathbf{u}_{n-1})$;
$\mathcal{R} \coloneqq \left\{ \left(P(\mathbf{u}^{(1)}), \text{not\_white}, N, \emptyset, \mathbf{u}^{(1)} \right) \right\}$;
$\mathcal{R} \coloneqq \mathcal{R} \cup \left\{ \left(P(\mathbf{u}^{(2)}), \text{not\_white}, N, \emptyset, \mathbf{u}^{(2)} \right) \right\}$;
$\mathfrak{N}.\mathcal{N} \coloneqq \mathcal{R}$; **return** $\mathcal{R}$;

6.1.3 *Parametric Tree*

Parametric surfaces

Both standard trees and contracting trees discussed previously employed an in/out function for the geometric object. From the geometric objects that the framework supports (cf. Def. 16), these trees are suited best for implicit objects because they are described by an in/out function. Especially for parametric objects, the use of in/out functions for characterization is not optimal. Therefore, we introduce the parametric tree, which is capable of handling parametric objects directly. Moreover, this tree type is non-regular.

Parametric tree

A parametric tree $\mathfrak{T}$ has an associated parametric object $\mathcal{O}$. Let $\mathcal{O} \coloneqq \left\{ (p_1(u), \ldots, p_n(u)) \;\middle|\; u \in \mathbf{u}^{(0)} \right\} \subset \mathbb{R}^n$ be described by the parametric function $p : \mathbb{R}^{n-1} \supset \mathbf{u}^{(0)} \to \mathbb{R}^n$ over its parametric domain $\mathbf{u}^{(0)} \in \mathbb{IR}^{n-1}$. Each parametric tree node is a 5-tuple $(\mathbf{x}, C, \mathfrak{P}, \mathcal{N}, \mathbf{u})$ where the meta-information entry is from the set $\mathbb{IR}^{n-1}$ and stores the part of the parametric domain that is associated with the current node. For each parametric tree node, $\mathbf{x} \supseteq \left\{ p(u) \;\middle|\; u \in \mathbf{u} \right\}$ has to hold. Therefore, the associated area $\mathbf{x}$ has a non-empty intersection with the geometric object $\mathcal{O}$ for each node $\mathfrak{N}$. To take into account this knowledge, we introduce a new node color:

Definition 26 (Node color not_white) *A node $\mathfrak{N}$ of an interval tree $\mathfrak{T}$ can be classified as not_white if $\mathfrak{T}.\mathfrak{TT}.\psi(\mathfrak{N}.\mathbf{x}) \cap \mathfrak{T}.\mathcal{O} \neq \emptyset$ holds.*

Because we can classify each node in a parametric tree as not_white, the set of colors is $\mathcal{C}_{par} \coloneqq \{\text{not\_white}\}$. Furthermore, the parametric tree uses the standard interpretation mapping ψ_{def}. The splitting operation is similar to that of the standard trees from Sect. 6.1.1: a bisection at the midpoint. In contrast to the standard trees, the parametric tree performs the bisection on the parametric domain $\mathbf{u}$ and not on $\mathbf{x}$ (cf. Alg. 4). This is also the reason why the parametric tree is non-regular. To summarize, the tree type of a parametric tree is $\mathfrak{TT}_{par} \coloneqq \left(\mathcal{C}_{par}, \psi_{def}, \text{split}_{par}, \mathbb{IR}^{n-1} \right)$ where split_{par} is defined by Alg. 4.

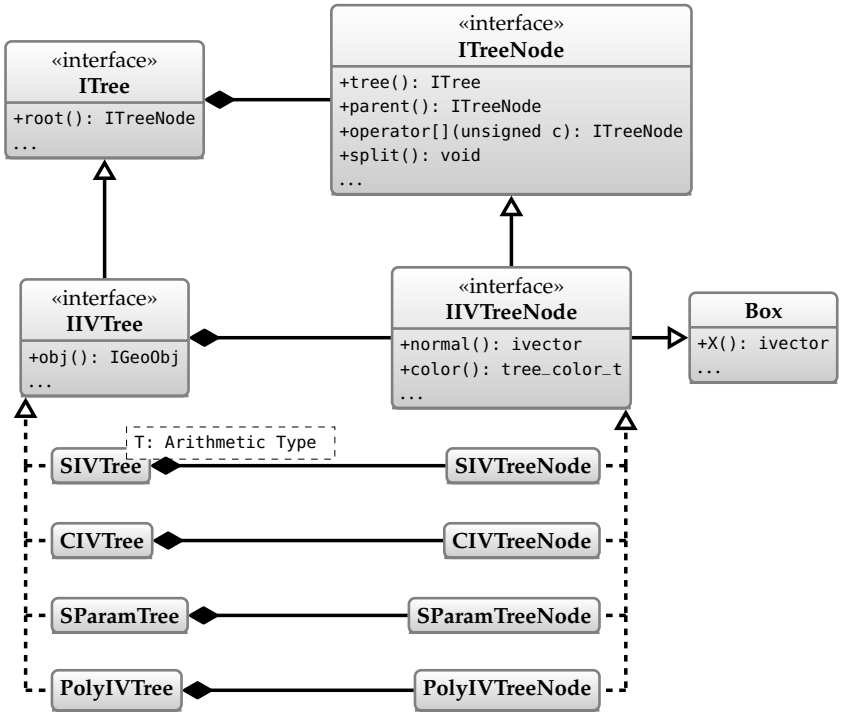


Figure 28: Overview of tree decomposition layer in UNIVERMEC (simplified).

6.1.4 *Realization in* UNIVERMEC

Implementation

The basic structure of the tree decomposition layer in UNIVERMEC is shown in Fig. 28. A tree is represented by an `ITree` interface and, following Def. 20, by a collection of `ITreeNode` instances. Interval trees (cf. Def. 22) are integrated into the overall structure by the `IIVTree` interface and interval tree nodes correspond to `IIVTreeNode`. Interval trees allow us to access the uniform representation of the associated geometric object provided by the `IGeoObj` interface (cf. Sect. 5.1). Using the `IIVTreeNode` interface, the members of the tuple defining an interval tree node can be retrieved. As an additional feature, the in-

Table 14: The tree decomposition structures implemented in UNIVERMEC and their theoretical basis.

	Colors	Mapping	Arithmetic	Regular	Type
`BinIVTree`	$\mathcal{C}_{def}$	ψ_{def}	IA	✓	$\mathfrak{TT}_{def}$
`BinAATree`	$\mathcal{C}_{def}$	ψ_{def}	AA	✓	$\mathfrak{TT}_{def}$
`BinAATree`	$\mathcal{C}_{def}$	ψ_{def}	TM	✓	$\mathfrak{TT}_{def}$
`CIVTree`	$\mathcal{C}_{con}$	ψ_{con}	*various*	✓	$\mathfrak{TT}_{con}$
`SParamIVTree`	$\mathcal{C}_{par}$	ψ_{def}	IA	×	$\mathfrak{TT}_{par}$
`PolyIVTree`	$\mathcal{C}_{def}$	ψ_{def}	IA	✓	$\mathfrak{TT}_{def}$

terface can compute the normal vectors of the geometric object over a box, if normals are available. Each of the tree types discussed above is provided as a separate class implementing the `IIVTree` interface. An overview of the classes and their connections with the theoretical basis discussed in the previous parts of this section is given in Tab. 14. The standard trees `BinIVTree`, `BinAATree`, and `BinTMTree` are realized as instatiations of the class template `SIVTree<T>` that provides a uniform basis for them. The `PolyIVTree` is a standard tree, which is optimized for polyhedral objects [0].

Purpose from the software perspective

From the software perspective, the tree decomposition layer has two main purposes: First, it provides an implementation of the interval tree concept, which is *reusable* from different algorithms. Second, it pursues the *encapsulation* and *abstraction* ideas from the earlier layers. The tree decomposition layer allows us to hide which of the techniques defined earlier is actually used inside a tree. An algorithm working with the `IIVTree` interface does not need to know which modeling type or range enclosure technique is employed or whether contractors are incorporated in the tree. Instead, it only needs to be able to interpret the colors of the tree nodes. Algorithms built on top of UniVerMeC working solely on the tree decompositions can be implemented in a uniform manner without taking the different characteristics of range-enclosure techniques or contractors into account. In this way, UniVerMeC not only enables fair comparisons between the employed techniques but makes the algorithms easily extendable.

6.2 GENERAL MULTISECTION

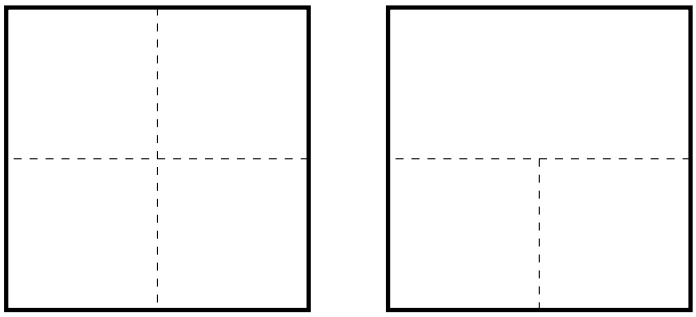
Basic branch and bound principle

As already mentioned in the introduction, algorithms with automatic result verification (e.g., for global optimization) often employ an interval branch and bound pattern. Usually, this class of algorithms stores parts of the search space in a sorted working list $\mathcal{L} := (\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(q)})$ in the form of interval boxes $\mathbf{x}^{(i)}$. The basic principle of these algorithms is to take the first box from $\mathcal{L}$, process it, and eventually create new boxes by multisection[5]. Good criteria for sorting $\mathcal{L}$ and for the multisection are crucial for improving the runtime of a branch and bound algorithm, which has an exponential worst-case complexity. These criteria have an inherently heuristical nature.

Sorting $\mathcal{L}$

This fact stipulated much research on developing and evaluating heuristics. Berner [Ber95a, pp. 29-34] examined the three strategies commonly used for sorting $\mathcal{L}$: *oldest-first*, *depth-first*, and *best-first*. The first two resemble the well-known first in, first out (FIFO) and last in, first out (LIFO) principles that can be implemented easily by using a queue or a stack for $\mathcal{L}$. The best-first strategy in global optimization sorts the list in such a way that $\underline{\Phi}\left(\mathbf{x}^{(i)}\right) \leqslant \underline{\Phi}\left(\mathbf{x}^{(j)}\right)$ holds for all $1 \leqslant i, j \leqslant q, i < j$ and an inclusion function Φ of the objec-

5 We consider the bisection a special case of multisection.



(a) The naive multisection schemes generates 2^n new boxes.

(b) The Ratz multisection scheme generates $n + 1$ new boxes.

Figure 29: Multisection schemes for subdividing the box $\mathbf{x} \in \mathbb{IR}^n$ (bold rectangle) in each coordinate.

tive function φ. Berner [Ber95a, Satz 2.2] proves that the best-first strategy does not behave worse in the normal case, compared to the other two strategies in the best case[6]. A problem of the best-first strategy is that its quality depends on the quality of the range enclosure obtained for φ over $\mathbf{x}$. Therefore, other criteria for sorting $\mathcal{L}$ were proposed. An interesting idea that has been studied recently is the *reject-index* [Cse01; Mar+06], [Kah05, pp. 34-38]. In its basic version, it tries to measure the difference between the currently known lower bound on the global minimum φ^* and $\underline{\Phi(\mathbf{x})}$ scaled by $\operatorname{wid}\mathbf{x}$.

Multisection strategies

It is important to choose not only a good order for storing boxes in $\mathcal{L}$ but also a good generation scheme for them. Such schemes are denoted as multisection, which means that an axis-aligned box $\mathbf{x} \in \mathbb{IR}^n$ is subdivided into $j \geqslant 2$ disjoint subboxes. Because the resulting boxes have to be axis-aligned again, the subdivision is always performed in parallel to a coordinate axis. Two possible schemes subdividing $\mathbf{x}$ along every coordinate axis are depicted in Fig. 29 for the two dimensional case. The naive scheme (cf. Fig. 29a) suffers from an exponential growth of boxes in each step, which is not suitable even for moderate values of n. An alternative schema was proposed by Ratz [Rat92, pp. 60-62] in the scope of the interval Gauss-Seidel method. The number of boxes grows linearly if this scheme is used (cf. Fig. 29b). Here, only the *most important coordinate* is bisected, whereas the subdivision in the other directions is performed for only one of the boxes created during this bisection. In the naive scheme, too, the growth can be limited if instead only the most important coordinates are considered for bisection.

Coordinate weights

Determining the importance of a coordinate direction is another crucial topic in this scope. In [RC95], Ratz and Csendes compared the commonly used rules (cf. Tab. 15). The maximum width rule

6 The best case is that the global minimum φ^* is known apriori and can be used to apply the midpoint test (53) (cf. Sect. 7.2.1).

Table 15: Common rules for determining the weight W_i of a coordinate direction according to Ratz and Csendes [RC95].

RULE	WEIGHT W_i
Hansen/Walster 1992	$W_i = \operatorname{wid}\left(\frac{\partial}{\partial x_i}\Phi(\mathbf{x})\right)\operatorname{wid}\mathbf{x}_i$
Machine Representation	$W_i = \begin{cases} \operatorname{wid}\mathbf{x}_i\ , & \text{if } 0 \in \mathbf{x}_i\ , \\ \frac{\operatorname{wid}\mathbf{x}_i}{\min_{x_i \in \mathbf{x}_i}(\lvert x_i \rvert)}\ , & \text{otherwise} \end{cases}$
Maximum width	$W_i = \operatorname{wid}\mathbf{x}_i$
Ratz 1992	$W_i = \operatorname{wid}\left(\frac{\partial}{\partial x_i}\Phi(\mathbf{x})(\mathbf{x}_i - \operatorname{mid}\mathbf{x}_i)\right)$

that bisects the longest side of the box is probably the most straightforward idea. However, it is not necessarily the best choice as the authors state. They conclude, that the rule Ratz 1992 delivers the best performance overall, followed by the Hansen/Walster 1992 rule. Both rules try to determine the importance of a variable by incorporating information about first-order derivatives. Another important conclusion drawn by the authors is that the selection of an appropriate rule depending on the actual problem can lead to a significantly better performance. Csendes, Klatte, and Ratz [CKR00] defined a theoretical basis using special posteriori selection rules to compare apriori rules from Tab. 15 and obtained results similar to the previous ones. Many other rules for selection of coordinate directions were proposed (e.g., [Rat92, pp. 97-98], [HW04, pp. 258-259] employ second-order information). Another interesting idea is to create a hybrid strategy from the different rules. It was proposed by, for example, Beelitz [Bee06, pp. 60-62], because he was not able to determine a best rule even with extensive tests. For further details, we refer to the treatment from both a theoretical and practical point of view in [MCC00a; MCC00b].

Split point selection

Another parameter that can be altered during the multisection is *where* to split along a chosen coordinate direction. A common choice is the midpoint of the coordinate, that is, $\operatorname{mid}\mathbf{x}_i$ (e.g., in [Kea96, pp. 157-159]). If the box $\mathbf{x}$ has gaps, they are another natural choice as a splitting point (e.g., [HW04, pp. 261-262]). A gap can be created, for example, by the extended interval division in the interval Newton procedure.

Implementation

As seen in the above discussion, there is a large number of different techniques and ideas for managing and generating boxes in branch and bound algorithms available, which makes it almost impossible to provide implementations for all of them. Therefore, in UniVerMeC we provide interfaces with which the different techniques can be integrated easily into our framework. The `IBoxListMgr` interface is responsible for the working list $\mathcal{L}$ of boxes of type `T` (cf. Fig. 30). Basically, it provides the same functions as a *priority queue*. The actual

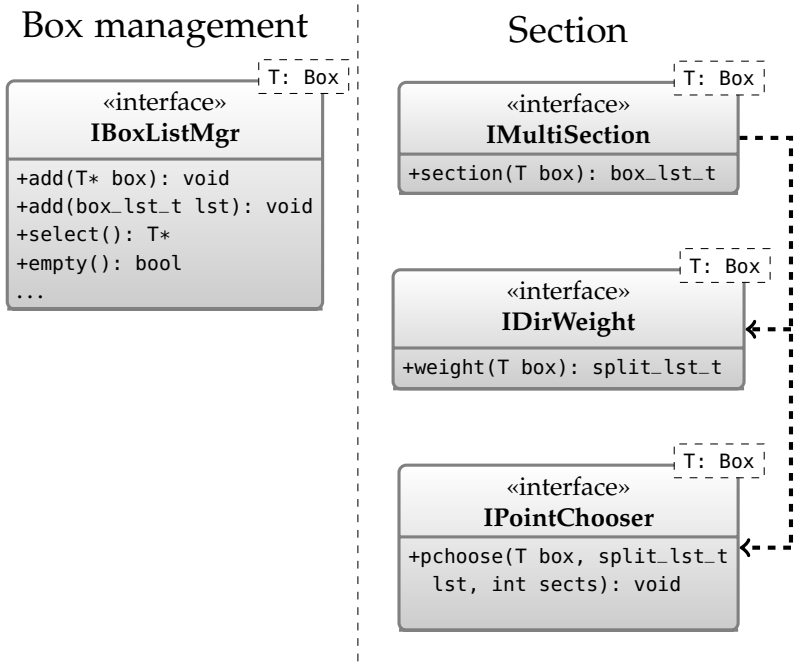


Figure 30: Interfaces of the multisection layer of UNIVERMEC.

multisection part in UNIVERMEC is split up onto three interfaces that correspond to three parameters of a multisection strategy, which have been identified above: `IMultiSection` (basic scheme), `IDirWeight` (coordinate weights), `IPointChooser` (split point selection). The actual multisection is performed by an `IMultiSection` instance. A concrete realization of a multisection scheme through `IMultiSection` can be parametrized with the other two interfaces, so that it can employ different coordinate weights and split point selection easily. Currently, we provide the Ratz section scheme, the coordinate weight rules maximum width, and Ratz 1992.

6.3 CONCLUSIONS

Formal framework

In this chapter, we presented our refinements for the interval tree approach by Dyllong and Grimm, and we discussed how our technique relates to the subpavings introduced by Jaulin et al. An important new aspect of our work is that we presented our trees in scope of a formal framework, which is also able to cover the standard interval trees and the CSG trees introduced in [DG07d; DG08].

Standard trees

Our standard IA, AA and TM trees are a direct application of the already known interval tree idea. Their uniform implementation can be carried out in a flexible framework, such as UNIVERMEC, in a straightforward manner. The use of more sophisticated arithmetics inside the trees cannot only improve the range-enclosure quality but also clear the way for a comprehensive and fair comparison of the different arithmetics in the scope of geometric computations. This is even more

important as, in contrast to the work of Dyllong and Grimm, the focus of our work does not lie on CSG objects, with quadric primitives, but on more complex implicit objects which often are described by much more complicated expressions. Therefore, they are plagued to an increased degree by overestimation problems, such as the dependency problem, which sometimes can be reduced by applying more sophisticated arithmetics.

Contracting tree

A novel structure we presented in this chapter was the contracting tree. It combines the well-known concepts of contractors and interval trees while retaining the important regularity notion. Using contractors during the subdivision can lead to a significant reduction of the uncertain areas and, thus, increase the quality of the decomposition. Additionally, the contracting tree fits very well into the software framework that we provide because it employs the different arithmetics, range-enclosure techniques, and contractors offered by UniVerMeC in a uniform manner. Furthermore, we showed that it can be converted losslessly to the standard tree types, which means that algorithms do not need any extra adaption to the tree.

Parametric tree

The second novel tree structure closes a conceptual gap with respect to supporting different geometric modeling types. All other tree types discussed in the thesis are best used in conjunction with implicit objects. With the parametric tree, we provide a new tree type optimized for objects described parametrically. This extends the scope of where our tree structures can be applied practically and can be seen mainly as a software addition. It is important for the decoupling purpose of the decomposition layer from a software engineering perspective.

Multisection strategies

The second section of the chapter gave an overview of techniques for multisection strategies and sorting working lists. This discussion led to a breakdown of a multisection strategy into three distinct components: multisection scheme, coordinate weights, and split point. We explained how the identified components are mapped onto software interfaces, which allow users to integrate new multisection techniques discussed in the literature more easily. Furthermore, the uniform components can be reused from different algorithms.

7 ALGORITHMS

Categories for algorithms in UniVerMeC

The purpose of this chapter is to describe algorithms that can be implemented in or made accessible by our framework. In general, if a given task should be solved by an algorithm, we can distinguish three different cases:

- A new algorithm was designed for this task and a complete implementation is necessary.
- An existing algorithm is suitable but a reimplementation is necessary.
- An existing algorithm provided by a third party can be used. It needs to be interfaced to the existing problem descriptions.

In this chapter, we give examples for the categories and show how the use of UniVerMeC can facilitate each of these tasks. For each category a different depth of knowledge is necessary. While the first category requires a deep understanding and the second category still a good amount of knowledge of the topic, the third category only demands using the third party library's specific interface. The level of detail in which we discuss the different algorithms in this chapter is adjusted to these needs.

Algorithms available in UniVerMeC

As an example of a new algorithm, we give an in-depth discussion of ϵ-*distance*. Developed in scope of the thesis, this approach computes verified bounds on the distance between two geometric objects. The implementation of the algorithm is made much easier by the use of the framework, because techniques that ϵ-distance employs, namely those for describing geometric objects (cf. Sect. 5.1) and for their hierarchical decompositions (cf. Sect. 6.1) are already available.

Global optimization is our example for the second category. We give an overview of important parts of the theory of *interval global optimization* and show how state-of-art algorithms from the area can be adapted and implemented in our framework in a straightforward manner. Moreover, we demonstrate how the implementation benefits from the already existing support for different range-arithmetics (cf. Chap. 3), functions, derivatives, contractors (cf. Chap. 4), and multisections (cf. Sect. 6.2).

As an example of the last category, we *interface already existing third party libraries* for solving IVPs. Because we treat these libraries as black boxes, we completely omit a theory discussion in this section. Although, the existing libraries themselves do not benefit from UniVerMeC in this case, users can take advantage of the uniform model descriptions and the ability to combine different algorithms or solvers.

Chapter organization

The chapter is structured as follows. The new distance computation algorithm is described in Sect. 7.1, followed by the global optimization algorithm in Sect. 7.2. After that we describe the interfacing of VALENCIA-IVP and VNODE-LP (two verified solvers for IVPs) before finally summarizing the material in Sect. 7.4.

7.1 DISTANCE COMPUTATION

Standard algorithms

Distance computation plays an important role in many application domains (e.g., robotics or biomechanics). For convex polyhedral models, standard algorithms, such as GJK [GJK88] or V-clip [Mir98], are readily available. Whereas the latter uses the Voronoi regions of features to compute the distance, the former is an iterative approach that calculates the Minkowski difference, between two polyhedrons[1], constructs a simplex inside the difference and computes the distance between the simplex and the origin. In each iteration, the simplex is modified to reduce the distance. A more complete overview for non-verified state-of-the-art procedures can be found in [Koc+09; LG98].

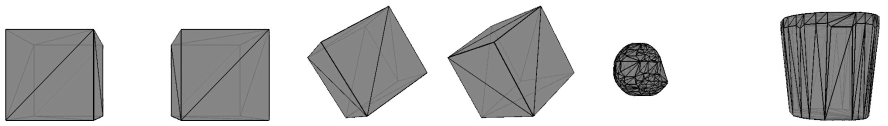
Handling other modeling types

The distance computation between modeling types other than polyhedral objects has attracted some attention in recent work (e.g., [LS02; UY07]). A very interesting class of objects are SQs, which can be described both by an implicit equation or parametrically. They were introduced by Barr [Bar81] in the context of computer graphics. Because the model depends only on 5 parameters, SQs are widely applied for surface reconstruction [JLS00]. Recent research showed that they can be used in biomechanics for bone modeling [Cuy11]. An approximation of the distance between SQs can be obtained using general purpose optimization methods as shown in [Cha+08; PSD09]. However, rigorous bounds are of great interest in biomechanics. In his recent master thesis [Chu11], the main results of which were published in [Aue+11], Chuev investigated how to obtain such bounds for convex SQs. He developed an a posteriori verification method based on an FP approximation (e.g., from GJK or interior point general purpose optimization). Compared to verified a apriori methods for convex objects, such as the interval GJK variant [DL04b] presented by Dyllong and Luther, which could be adapted to convex SQs as well, the approach of Chuev has the advantage that well-tested and fast FP implementations can be used to obtain the approximate distance. The use of the rather slow IA is limited to the verification step. However, in contrast to purely interval methods, his approach cannot handle model uncertainty in a straightforward way.

A posteriori verification

From the few algorithms available for verified distance computation, we outline the approach of Chuev [Chu11] because it has im-

1 The applicability of the GJK algorithm is not limited to polyhedral objects. It is suitable for convex objects in general if a so-called *support mapping* [Ber04, pp. 130-131] is available.



(a) Two cubes with parallel sides.

(b) Two rotated cubes.

(c) Two objects described by over 700 hyperplanes.

Figure 31: Three test scenarios for which the interval global optimization approach required many iterations before delivering a narrow distance enclosure.

portant usage in scope of the biomechanical application discussed in Sect. 8.1.2 and is different compared to the interval tree methods described in detail later. It computes a rigorous bound on the distance between two convex SQs or a convex SQ and a point. For simplicity reasons, we will outline the method for the latter case only. Let a convex SQ $\mathcal{O}$ be given by its implicit function $f : \mathbb{R}^3 \to \mathbb{R}$ and a point $p \in \mathbb{R}^3$. The FP solver returns an approximate solution point $q \in \mathbb{R}^3$ for the minimum distance between the SQ and p. The a posteriori verification of the approximation is performed in two separate steps. In the first step, we compute a point q^- on the line $l(s) = p + s \cdot (q - p)$, so that $f(q^-) \leqslant 0$ holds. A rigorous upper bound on the minimum distance between p and $\mathcal{O}$ is obtained by $\|p - q^-\|$. Assuming that q is a good approximation, the quality of the rigorous upper bound increases if q^- approaches q. The second step starts by searching a point q^+ on the line $l(s)$, so that, $f(q^+) > 0$. Then the interval hyperplane $h(x) = \nabla F(\square(q^-, q^+)) \cdot (x - \square(q^-, q^+))$ is constructed where ∇F is an inclusion function for the gradient of f and $\square$ the interval hull operator. Because $h(x)$ contains a support plane of $\mathcal{O}$, which is simultaneously a separating plane between $\mathcal{O}$ and p, calculating the minimum distance between $h(x)$ and p yields a rigorous lower bound on the distance between $\mathcal{O}$ and p. The algorithm can be implemented on a computer easily because all operations can be carried out with standard IA.

Using general purpose optimization

A disadvantage of this a posteriori verification approach is that it is applicable only for convex objects. This is also true for criteria from general purpose optimization. For example, the KKT conditions are necessary and sufficient for the global optimum in the case of convex optimization problems [BV09, p. 244]. For non-convex problems, it is more difficult to find criteria for verifying a given approximate solution a posteriori. However, non-convex geometric objects arise naturally in modeling real-world scenarios (e.g., bent SQs for bone modeling in *total hip replacement* cf. Sect. 8.1.2). One way to compute the distance is to apply a general-purpose interval global optimization algorithm (e.g., [Bee06; HW04; Kea96]), which can cope with both convex and non-convex problems. Consider two geomet-

ric objects $\mathcal{O}_1, \mathcal{O}_2 \subset \mathbb{R}^n$ described by the in/out functions f_1, f_2 with $f_i : \mathbb{R}^n \to \mathbb{R}$. To compute the minimum distance between them, the following optimization problem can be solved:

$$\begin{aligned} \min_{x \in \mathbb{R}^{2n}} & \left\| x^{(1)} - x^{(2)} \right\|^2 , \\ & f_1(x^{(1)}) \leqslant 0 , \\ & f_2(x^{(2)}) \leqslant 0 . \end{aligned} \tag{42}$$

where $x = \left(x^{(1)}, x^{(2)}\right)^T$ with $x^{(i)} \in \mathbb{R}^n$. We tested this approach in [DK10] only for convex polyhedra, which can be described by unions of halfspaces. These unions translate directly into inequality constraints for an optimization problem. The tests were carried out using an implementation of the state-of-art interval global optimization algorithm for inequality constrained problems by Hansen and Walster [HW04, pp. 343-379] (cf. Sect. 7.2). We performed only slight adaptions of the general purpose approach; in particular, we employed parallelization to improve the runtime. While it was possible to derive the distances in this way (cf. [DK10] for details), the solving process was slow, especially in some geometric degenerated scenarios (cf. Fig. 31). For our purposes, a more important problem is that, while in theory these kinds of algorithms work without derivatives, the actucal implementations usually expect a differentiable problem. This is not guaranteed if the objects $\mathcal{O}_i$ are non-smooth (e.g., non-convex polyhedra) or CSG objects where special treatment of the set-theoretic operations (e.g., with R-functions [Sha07; Sha91]) would be necessary to ensure differentiability (cf. Sect. 5.1). Also, the exploitation of parametric descriptions is difficult with standard global optimization methods.

7.1.1 *A Basic Distance Computation Algorithm for Interval Trees*

Distance computation between trees

Another approach to calculate the distance between the two non-convex objects $\mathcal{O}_1, \mathcal{O}_2$ is to construct their hierarchical decompositions (e.g., as an octree) and carry out the distance computation between them. The algorithm for rigorous distance computation proposed in this thesis follows this approach and derives a rigorous bound that also holds for the original objects based on the trees. Our algorithm is a refinement of the algorithm in [BDL04] (Dist. 1), [DL04a] (Dist. 2), and [DG07a; DG07b] (Dist. 3/4). The algorithm Dist. 1 computes the distance between two objects encoded in one octree. Thus, both objects share a common coordinate system. The algorithm Dist. 2 is developed for the case of two objects encoded by octrees that are represented in different coordinate systems. Both algorithms work on octrees with a fixed subdivision depth. These algorithms do not consider the underlying objects and are limited by the maximum sub-

division depth of the octrees which they take as input. The last algorithms Dist. 3/4 assume again that the trees are not represented in a common coordinate system. Additionally, they offer a dynamic splitting behavior. To achieve this, an analytic model is assumed to lie behind the octree data structure. Thus, the subdivision depth of a node can be dynamically increased if necessary. However, both algorithms return only a lower bound on the distance.

ϵ - distance algorithm

Our own algorithm *ϵ-distance* [DK12; KLD13] can be seen as a hybrid version of the above algorithms. It assumes that the interval trees decomposing $\mathcal{O}_1, \mathcal{O}_2$ share a common coordinate system. Therefore, the algorithm uses an improved version of the node-node distance computation mechanism from [BDL04]. Because our interval trees are always connected with an analytic model of the underlying geometric object (cf. Def. 22), the algorithm can make use of an adaptive mechanism for increasing the subdivision depth if necessary. In contrast to the Dist. 3/4 approach, we compute a fully rigorous enclosure $\boldsymbol{\Upsilon}$ on the distance between $\mathcal{O}_1, \mathcal{O}_2$, so that, $\upsilon = \min_{x \in \mathcal{O}_1, y \in \mathcal{O}_2} \|x - y\| \in \boldsymbol{\Upsilon}$ holds with $\operatorname{wid} \boldsymbol{\Upsilon} \leqslant \epsilon$ where $\epsilon > 0$ is the user-defined accuracy. We will restrict the basic discussion of the algorithm to the tree types with color sets $\mathcal{C} \subseteq \{\text{black}, \text{gray}, \text{white}, \text{not\_white}\}$. Inversion nodes can be converted to these colors if necessary (cf. Alg. 3).

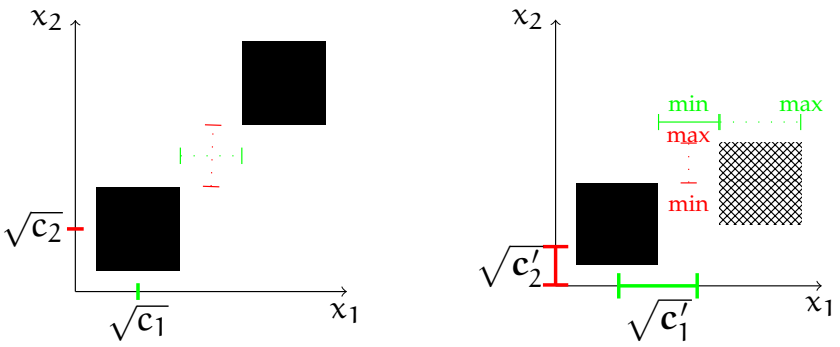
Used data structures

The input of the algorithm are the trees $\mathfrak{IT}_1$ and $\mathfrak{IT}_2$ associated with the geometric objects $\mathcal{O}_1$ and $\mathcal{O}_2$, respectively. Additionally, it is assumed that $\mathfrak{IT}_i.\mathfrak{TT}.\psi(\mathfrak{N})$ maps into $\mathbb{IR}^n$ for all $\mathfrak{N} \in \mathbb{T}_{\mathfrak{IT}_i.\mathfrak{TT}}$, $i = 1, 2$. Throughout the rest of this section, we denote the interval box (e.g., $\mathfrak{IT}_i.\mathfrak{TT}.\psi(\mathfrak{N})$) associated with a node (e.g., $\mathfrak{N}$) by a bold letter ($\mathbf{n}$) to simplify the notation. The algorithm ϵ-distance uses a working list $\mathcal{L}$ with candidate tuples $(\mathfrak{X}, \mathfrak{Y}, \mathbf{d})$. Here $\mathfrak{X}$ and $\mathfrak{Y}$ denote nodes from the first and second subdivision tree, whereas $\mathbf{d}$ is an enclosure of the square of the Euclidean distance $\|\mathbf{x} - \mathbf{y}\|^2$ between them. Further variables of the algorithm are the enclosure $\boldsymbol{\Upsilon}$ of the minimum distance υ between $\mathcal{O}_1$ and $\mathcal{O}_2$ and the final list $\mathcal{L}_{\mathtt{final}}$. A user-provided threshold ϵ is employed as a termination criterion, that is, the algorithm stops if $\operatorname{wid} \boldsymbol{\Upsilon} \leqslant \epsilon$.

Case selector for distance queries between interval tree nodes

The basic idea of ϵ-distance is as follows. The algorithm takes the first tuple $(\mathfrak{X}, \mathfrak{Y}, \mathbf{d})$ from $\mathcal{L}$ and splits the node associated interval vector of which has the smaller width. If we assume that $\mathfrak{X}$ was split and denote the resulting set of child nodes by $\mathcal{N}$, the ϵ-distance generates a new tuple $(\mathfrak{N}, \mathfrak{Y}, \mathrm{d}(\mathfrak{N}, \mathfrak{Y}))$ for each node $\mathfrak{N} \in \mathcal{N}$ that is not white. We can discard tuples with white[2] nodes because they do not belong to the object. The function $\mathrm{d}(\mathfrak{N}, \mathfrak{Y})$ returns an enclosure of the square of the Euclidean distance between the object parts of $\mathcal{O}_1$ and $\mathcal{O}_2$ contained in the area covered by the nodes $\mathfrak{N}, \mathfrak{Y}$. The enclosure

2 or white inversion



(a) Two black nodes

(b) A black and a gray node

Figure 32: Graphical representation of case selectors. The lines visualize the distances between the two depicted areas associated with nodes. On the axes, the $\sqrt{c_i}$ ($\sqrt{\mathbf{c}'_i}$) are shown. They represent (or enclose) the projection of distance between these axis-aligned boxes on each axis.

$\mathbf{d}$ is computed by the function $d(\mathfrak{X}, \mathfrak{Y})$ defined in (43), which uses a slightly extended version (45) of the case selector (44) from [BDL04]:

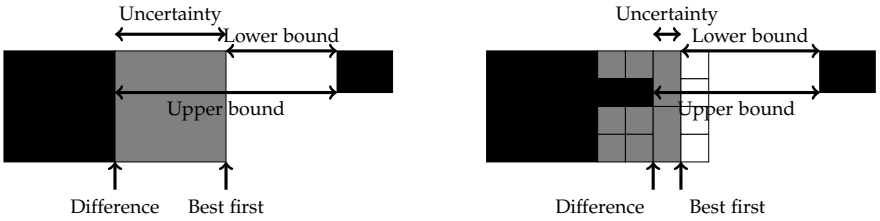
$$d(\mathfrak{X}, \mathfrak{Y})^2 := \begin{cases} \sum_{i=1}^{n} c_i \ , & \mathfrak{X}.C = \mathfrak{Y}.C = \text{black} \ , \\ \sum_{i=1}^{n} \mathbf{c}'_i \ , & \mathfrak{X}.C = \text{black} \neq \mathfrak{Y}.C \ , \\ \|\mathbf{x} - \mathbf{y}\|^2 \ , & \text{otherwise} \ ; \end{cases} \tag{43}$$

$$c_i := \begin{cases} (\underline{y_i} - \overline{x_i})^2 \ , & \underline{y_i} > \overline{x_i} \ , \\ (\underline{x_i} - \overline{y_i})^2 \ , & \underline{x_i} > \overline{y_i} \ , \\ 0 \ , & \text{otherwise} \ ; \end{cases} \tag{44}$$

$$\mathbf{c}'_i := \begin{cases} (\mathbf{y}_i - \overline{x_i})^2 \ , & \underline{y_i} > \overline{x_i} \ , \\ (\underline{x_i} - \mathbf{y}_i)^2 \ , & \underline{x_i} > \overline{y_i} \ , \\ [0, \max(h'_i, h''_i)] \ , & \text{otherwise} \ ; \end{cases} \tag{45}$$

$$h'_i := \begin{cases} (\overline{y_i} - \overline{x_i})^2 \ , & \overline{y_i} > \overline{x_i} \ , \\ 0 \ , & \text{otherwise} \ ; \end{cases}$$

$$h''_i := \begin{cases} (\underline{y_i} - \underline{x_i})^2 \ , & \underline{y_i} > \underline{x_i} \ , \\ 0 \ , & \text{otherwise} \ . \end{cases}$$



(a) Start situation: The width of the distance enclosure is dominated by the uncertain area's width.

(b) Situation after applying the criteria: The width of the uncertain area has been reduced. Thus, producing a better distance enclosure.

Figure 33: The algorithm switches between two criteria for list sorting: *difference* and *best-first*. The former tries to produce black nodes and the latter white nodes. After applying both criteria we should be able to reduce the uncertain region from both sides.

The new selector $\mathbf{c}_i'$ provides tighter bounds if only one node is black[3]. Both selectors, the old and the improved one, are depicted in Fig. 32. If $\mathfrak{N}$ and $\mathfrak{Y}$ are both black, we discard them and update $\boldsymbol{\Upsilon}$ as follows: Since $\overline{d(\mathfrak{N}, \mathfrak{Y})}$ establishes an upper bound for the actual distance between the objects, $\overline{\Upsilon}$ chosen as the minimum between its current value and $\overline{d(\mathfrak{N}, \mathfrak{Y})}$:

$$\overline{\Upsilon} \coloneqq \min\left(\overline{\Upsilon}, \overline{d(\mathfrak{N}, \mathfrak{Y})}\right) \quad . \tag{46}$$

The lower bound $\underline{\Upsilon}$ is updated by $\min\left(\underline{\Upsilon}, \underline{d(\mathfrak{N}, \mathfrak{Y})}\right)$.

Handling gray and not_white nodes

If at least one node is gray, we have an uncertain area (cf. Fig. 33a) where it is unknown if a part of the object lies there. However, $\underline{d(\mathfrak{N}, \mathfrak{Y})}$ is a verified lower bound for the minimum distance between $\mathfrak{N}.\mathfrak{x}$ and $\mathfrak{y}$ for all points they possibly contain. The tuple can be discarded if

$$\underline{d(\mathfrak{N}, \mathfrak{Y})} > \overline{\Upsilon} \quad . \tag{47}$$

This criterion is equivalent to the well-known *midpoint test* in interval global optimization. Because we stop if $\operatorname{wid} \boldsymbol{\Upsilon} \leqslant \epsilon$, we do not need to process a tuple further if $\underline{d(\mathfrak{N}, \mathfrak{Y})} - \overline{\Upsilon} \leqslant \epsilon$ holds, that is, the tuple fulfills the termination criterion. In this case, $\underline{\Upsilon}$ is updated with $\underline{\Upsilon} = \min\left(\underline{\Upsilon}, \underline{d(\mathfrak{N}, \mathfrak{Y})}\right)$ and the tuple is discarded, otherwise it is inserted into $\mathcal{L}$ for further processing. Nodes with the color not_white can be handled similarly to gray nodes. The main difference is that if both nodes are not white, we know that the upper bound on their distance establishes a rigorous upper bound the distance between $\mathcal{O}_1, \mathcal{O}_2$. Therefore, we apply (46) to update $\overline{\Upsilon}$ in this case.

Sorting $\mathcal{L}$

Similarly to interval global optimization, the sorting of $\mathcal{L}$ has a

3 Likewise, if $\mathfrak{Y}$ is black and $\mathfrak{X}$ is gray, the second case of (43) applies.

great impact on the algorithm's performance. The sorting criteria are mostly of heuristic nature. The intuitive idea is to use a *best-first* criterion that puts tuples with small lower bounds first. That is, if

$$\underline{d_1} < \underline{d_2} \qquad \text{(best-first)} \tag{48}$$

holds for two tuples with the associated distance enclosures $\mathbf{d}_1, \mathbf{d}_2$, the tuple with the enclosure $\mathbf{d}_1$ should be processed first. This strategy usually leads to a faster subdivision of the gray leaf nodes bordering on white nodes. Another possibility is to sort the list according to the minimum difference of the tuple's associated distance and the current verified upper bound on the minimum distance:

$$\underline{d_1} - \overline{\Upsilon} < \underline{d_2} - \overline{\Upsilon} \qquad \text{(difference)} \ . \tag{49}$$

This criterion is similar to the *reject-index* [Cse01] (cf. Sect. 6.2) without scaling and often leads to a heavier subdivision of the gray leaf nodes bordering on black nodes. Our algorithm alternates both criteria reducing the uncertain area from both sides (cf. Fig. 33b).

Algorithm

Our procedure to compute a verified enclosure between the distance of two geometric objects $\mathcal{O}_1, \mathcal{O}_2$ is given in Alg. 5. As input, the algorithm takes the accuracy threshold ϵ and two interval trees $\mathcal{IT}_1, \mathcal{IT}_2$ representing the objects. Therefore, the implementation is independent of the actual underlying geometric modeling type. As mentioned above, we assume that the range of $\mathcal{IT}_i.\mathfrak{T}\mathfrak{T}.\psi$ is a box for every node in $\mathbb{T}_{\mathcal{IT}_i.\mathfrak{T}\mathfrak{T}}$, $i = 1, 2$. We allow the use of nodes with the color white_inv as an exception from this rule, because the algorithm does not need to consider the area associated with nodes that do not intersect with the geometric objects. To allow for easy extension with further colors several predicates are used to check whether a node of a tree for the geometric object $\mathcal{O}$ has certain properties instead of checking for concrete colors. In the following list, the predicates, their conditions and the corresponding node colors are given:

`is_solid` The node cannot be subdivided further (*black*, *white*, *white_inv*).

`is_divis` The node can be subdivided further (*gray*, *not_white*).

`has_point` The associated area of the node intersects $\mathcal{O}$ (*black*, *not_white*).

`is_empty` The associated area of the node does not intersect $\mathcal{O}$ (*white*, *white_inv*).

After beginning, the algorithm initializes its working list $\mathcal{L}$ with the two root nodes of the trees in line 2. Here, we assume that `is_empty` evaluates to false for both root nodes. After that, the threshold ϵ_2 is

Algorithm 5: Calculation of a distance enclosure between two interval trees

Data: Trees $\mathfrak{IT}_1, \mathfrak{IT}_2$ for objects $\mathcal{O}_1, \mathcal{O}_2$; Accuracy ϵ
Result: Enclosure $\boldsymbol{\Upsilon}$ of the minimum distance between $\mathcal{O}_1, \mathcal{O}_2$
$\mathcal{L}_{final} \coloneqq \emptyset$; $\boldsymbol{\Upsilon} \coloneqq [\infty, \infty]$;
$\mathfrak{R}_1 \coloneqq \text{root}(\mathfrak{IT}_1.\mathcal{T})$; $\mathfrak{R}_2 \coloneqq \text{root}(\mathfrak{IT}_2.\mathcal{T})$;
$\mathcal{L} \coloneqq \{(\mathfrak{R}_1, \mathfrak{R}_2, \mathrm{d}(\mathfrak{R}_1, \mathfrak{R}_2))\}$;
$\epsilon_2 \leftarrow \max(\text{wid}\,\mathbf{r}_1, \text{wid}\,\mathbf{r}_2)/2$;
START: **while** $\mathcal{L} \neq \emptyset$ **do**
  $(\mathfrak{X}, \mathfrak{Y}, \mathbf{d}) \coloneqq$ `head(`$\mathcal{L}$`)`; $\mathcal{L} \coloneqq$ `tail(`$\mathcal{L}$`)`;
  if $\underline{\mathrm{d}} > \overline{\Upsilon}$ **then continue**;
  if $((\text{wid}\,\mathbf{x} < \text{wid}\,\mathbf{y}) \wedge$ `is_divisible(`$\mathfrak{Y}$`)`$) \vee$ `is_solid(`$\mathfrak{X}$`)`
  then
    `swap(`$\mathfrak{X}$, $\mathfrak{Y}$`)`;
  $\mathcal{C} \coloneqq$ `split(`$\mathfrak{X}$`)`;
  foreach $\mathfrak{C}$ *in* $\mathcal{C}$ **do**
    if `is_empty(`$\mathfrak{C}$`)` **then continue**;
    `// swap(`$\mathfrak{C}$,$\mathfrak{Y}$`) if necessary`
    $(\mathfrak{X}', \mathfrak{Y}', \mathbf{d}') \coloneqq (\mathfrak{C}, \mathfrak{Y}, \mathrm{d}(\mathfrak{C}, \mathfrak{Y}))$;
    if $\underline{\mathrm{d}'} > \overline{\Upsilon}$ **then continue**;
    if `has_point(`$\mathfrak{X}'$`)` $\wedge$ `has_point(`$\mathfrak{Y}'$`)` **then** $\overline{\Upsilon} \coloneqq \min\left(\overline{\Upsilon}, \overline{\mathrm{d}'}\right)$;
    if `is_solid(`$\mathfrak{X}'$`)` $\wedge$ `is_solid(`$\mathfrak{Y}'$`)` **then** $\underline{\Upsilon} \coloneqq \min\left(\underline{\Upsilon}, \underline{\mathrm{d}'}\right)$;
    else if $\overline{\Upsilon} - \underline{\mathrm{d}'} \leqslant \epsilon$ **then** $\underline{\Upsilon} \coloneqq \min\left(\underline{\Upsilon}, \underline{\mathrm{d}'}\right)$;
    else if $(\text{wid}\,\mathbf{x}' < \epsilon_2 \vee$ `is_solid(`$\mathfrak{X}'$`)`$) \wedge$
        $(\text{wid}\,\mathbf{y}' < \epsilon_2 \vee$ `is_solid(`$\mathfrak{Y}'$`)`$)$ **then**
      $\mathcal{L}_{final} \coloneqq \mathcal{L}_{final} \cup \{(\mathfrak{X}', \mathfrak{Y}', \mathbf{d}')\}$;
    else $\mathcal{L} \coloneqq \mathcal{L} \cup \left\{(\mathfrak{X}', \mathfrak{Y}', \mathbf{d}')\right\}$;
if $\overline{\Upsilon} < \infty$ **then** Switch sorting criterion for $\mathcal{L}$;
foreach $(\mathfrak{X}, \mathfrak{Y}, \mathbf{d})$ *in* $\mathcal{L}_{final}$ **do**
  if $\underline{\mathrm{d}} > \overline{\Upsilon}$ **then continue**;
  else if $\overline{\Upsilon} - \underline{\mathrm{d}} \leqslant \epsilon$ **then** $\underline{\Upsilon} \coloneqq \min\left(\underline{\Upsilon}, \underline{\mathrm{d}}\right)$;
  else $\mathcal{L} \coloneqq \mathcal{L} \cup \{(\mathfrak{X}, \mathfrak{Y}, \mathbf{d})\}$;
$\epsilon_2 \coloneqq 0.9\epsilon_2$; $\mathcal{L}_{final} \coloneqq \emptyset$;
if $\mathcal{L} \neq \emptyset$ **then goto** *START*;
return $\boldsymbol{\Upsilon}$;

initialized in line 4. If the area $\mathbf{x}$ associated with a node $\mathfrak{X}$ falls under the threshold ϵ_2, that is, if

$$\operatorname{wid}\mathbf{x} \leqslant \epsilon_2 \ ,$$

it is temporarily suspended from the subdivision process. If both nodes of a tuple fall below the threshold, the tuple is moved to an intermediate list $\mathcal{L}_{\mathtt{final}}$. Basically, this approach combines the *oldest-first* sorting strategy (cf. Sect. 6.2) with the best-first and difference criteria, which are used for sorting $\mathcal{L}$. This ensures that the search area is subdivided more uniformly, while still preferring parts that are likely to contain the desired minimum distance. Another advantage is that the oldest-first strategy often reduces the number of elements in $\mathcal{L}$ [Kah05, p. 34]. However, whether this also is the case if the oldest-first strategy is combined with other strategies needs a more thorough experimental analysis. After entering the main loop (line 5), the algorithm takes the first tuple from $\mathcal{L}$ and tries to discard it (line 7) using (47). If this is not possible, the node with the larger associated area is split. After that, we iterate over its children. New tuples are generated (line 13) using the created child nodes if they are not proven to have an empty intersection with $\mathcal{O}$. Further, if the `has_point` predicate evaluates true for both nodes, we can update the upper bound $\overline{\Upsilon}$ (line 15). If both nodes are black, the minimum $\underline{\Upsilon}$ is updated and the tuple is discarded (line 16). The same steps are carried out if a tuple delivers the desired accuracy (line 17). As explained above, a tuple falling under the threshold ϵ_2 is temporarily moved to $\mathcal{L}_{\mathtt{final}}$ and ignored (line 18). If $\mathcal{L}$ is empty, we exchange the sorting criterion if possible[4] (line 22), move all elements from $\mathcal{L}_{\mathtt{final}}$ to $\mathcal{L}$ so that $\mathcal{L}_{\mathtt{final}}$ is empty (line 26), decrease ϵ_2, and finally restart the main loop (line 28). The above algorithm does not use higher order information, such as offered by normals, in order to be applicable to a wide range of models.

7.1.2 *Using Normals for Distance Computation*

Clustering effects

Depending on the geometric configuration, the basic algorithm described in the previous subsection might suffer from massive clustering effects, which can be reduced sometimes by incorporating higher order information into the computation process [DK94]. Therefore, our algorithm can optionally make use of information provided by normals.

Non-collinearity test

Denote enclosures of the normal vectors of $\mathcal{O}_1, \mathcal{O}_2$ over $\mathbf{x}, \mathbf{y}$ for a tuple $(\mathfrak{X}, \mathfrak{Y}, \mathbf{d})$ by $\mathbf{n}_x$ and $\mathbf{n}_y$. The tuple can be safely discarded if

$$0 \notin \mathbf{n}_x \times \mathbf{n}_y \ . \tag{50}$$

4 The algorithm starts with the best-first criterion because the difference criterion is only suitable if a meaningful $\overline{\Upsilon}$ is known.

The criterion is based on the observation that the two normal vectors have to be collinear to each other at a point of minimum distance between two objects. It has been used by Snyder et al. [Sny+93] in a rigorous algorithm for collision detection.

Normal cone test

Another test is to check whether the vectors $\mathbf{y}-\mathbf{x}$ and $\mathbf{x}-\mathbf{y}$ lie in the normal cones with the generating vectors $\mathbf{n}_x$ and $\mathbf{n}_y$, respectively. We can discard a tuple if at least one of the intersections

$$\frac{\mathbf{y}-\mathbf{x}}{\|\mathbf{y}-\mathbf{x}\|} \cap \frac{\mathbf{n}_x}{\|\mathbf{n}_x\|} \cap -\frac{\mathbf{n}_y}{\|\mathbf{n}_y\|} \;,\; \frac{\mathbf{x}-\mathbf{y}}{\|\mathbf{x}-\mathbf{y}\|} \cap \frac{\mathbf{n}_y}{\|\mathbf{n}_y\|} \cap -\frac{\mathbf{n}_x}{\|\mathbf{n}_x\|} \tag{51}$$

is empty. As a rule, this condition serves its purpose only if no interval vector contains the zero.

Aquiring normals in UniVerMeC

In case of an analytic model description (e.g., implicit functions) normal vectors can be acquired by differentiation of the corresponding expressions. In UniVerMeC, derivatives are available through the `IDerivative` interface (cf. Sect. 4.4.2). The information is gathered inside the interval trees decomposing the geometric object and made accessible through the `normal()` member function in `IIVTreeNode` (cf. Sect. 6.1.4). Because ϵ-distance relies solely on this function to obtain normals, the algorithm can use them independently of the actual geometric modeling types describing $\mathcal{O}_1, \mathcal{O}_2$.

7.1.3 *Improvements of ϵ-Distance Algorithm Using Floating-Point Methods*

Initialize $\overline{\Upsilon}$

The runtime of the algorithm depends heavily on its ability to prune parts of the search space quickly. Therefore, we can accelerate it by providing a good initial value for $\overline{\Upsilon}$. It can be derived by the means of FP methods not necessarily returning the best but still a good value for $\overline{\Upsilon}$. Such a value can be obtained using, for example, FP optimization algorithms.

Using approximate solutions

Let $f_1, f_2 : \mathbb{R}^n \to \mathbb{R}$ the in/out functions describing $\mathcal{O}_1, \mathcal{O}_2 \subset \mathbb{R}^n$ and $x^{*'} = (x^{(1)*'}, x^{(2)*'})^T \in \mathbb{R}^{2n}, x^{(i)*'} \in \mathbb{R}^n$ be the approximate solution point of (42) returned by an FP solver. In order to initialize the interval algorithm with these values, we have to make sure that the points are feasible by checking $f_1\left(x^{(1)*'}\right) \leqslant 0$ and $f_2\left(x^{(2)*'}\right) \leqslant 0$. In finite arithmetic, we can use, for example, IA to perform this check in a rigorous manner. If both points are feasible, we can initialize $\overline{\Upsilon}$ by

$$\overline{\Upsilon} := \left\| x^{(1)*'} - x^{(2)*'} \right\|^2 \quad .$$

If the point is not feasible or its feasibility cannot be proved with standard IA, we can try to move it inside the feasible region along the object's negative normal vector. Alternatively, we can apply an a posteriori verification technique, for example, ϵ-inflation in combination with a fixed-point theorem (cf. for details [Ham+97, p. 299]).

Obtaining approximate solutions

To obtain an approximate solution, any FP solver interfaced to UNIVERMEC can be used. Because our solver interfaces work on the same uniform function representation as that used by the geometric layer (cf. Sect. 5.1) for the in/out functions, it can be applied in a straightforward manner to (42). Currently, an interface to the interior point optimization algorithm provided by the IPOPT library [WB06] (cf. Sect. 7.3.3) is part of the framework. Note that, in the case of interior point algorithms, we can ensure that the returned approximate solution $x^{*'}$ is feasible by modifying the termination criterion, so that the approximate solution lies slightly in the interior of the object and its feasibility can be proven with naive IA in most cases directly. We used this approach in [DK10].

7.1.4 *Further Improvements*

Exploiting hull consistency

The distance computation algorithm described in the previous section allows us to compute the distance between geometric objects represented by interval trees. As such, its performance depends not only on the geometric configuration and the accuracy ϵ requested by the user but also on the quality of the hierarchical decompositions. Because the algorithm works with our very general interval tree notion (cf. Def. 22), it can make use of more sophisticated structures, for example, the contracting tree (cf. Sect. 6.1.2), to improve the decomposition quality. However, a tree $\mathfrak{IT}$ remains associated to a geometric object $\mathcal{O}$ in every case, that is, the node colors reflect knowledge about $\mathcal{O}$. This makes the introduction of contractors that prune the areas associated with tree nodes complicated if these contractors exploit domain knowledge of the algorithm (e.g., $\overline{\Upsilon}$). To allow the use of such contractors, the interval trees can be extended with the possibility to store temporary nodes that are removed after the main algorithm finishes. As an example of such an exploitation, consider the replacement of the simple midpoint test (47) by hull consistency[5] (cf. Sect. 4.3.3). If $\overline{\Upsilon}$ is the upper bound on the minimum distance and $\mathbf{x}, \mathbf{y}$ are two boxes, then

$$\|x^* - y^*\|^2 \leqslant \overline{\Upsilon}$$

holds for all candidates $x^*, y^* \in \mathbb{R}^n$ for the minimum distance, where $x^* \in \mathbf{x}, y^* \in \mathbf{y}$. If we solve this inequality formally for the i-th component of x, we obtain the following condition:

$$x_i^* \in \mathbf{y}_i \pm \sqrt{\left([0, \overline{\Upsilon}] - \sum_{\substack{j=1 \\ j \neq i}}^{n} (\mathbf{x}_j - \mathbf{y}_j)^2 \right) \cap [0, \infty)} \ .$$

5 Replacing the midpoint-test by hull consistency is also employed in modern algorithms for global optimization (e.g. [HW04, pp. 293-294])

That is, x^* is a candidate for the minimum distance between the boxes $\mathbf{x}$ and $\mathbf{y}$ only if this condition holds for each of its coordinates $i = 1 \dots n$.

Extended proximity queries

Another interesting approach for improving the algorithm is to use more sophisticated distance computation primitives in addition to (43). For example, it might be possible to construct an ILIE (cf. Sect. 4.3.4), which encloses the boundary of the geometric object over a box $\mathbf{x}$ and then to use this hyperplane to compute the distance. Moreover, it would be interesting to identify convex subregions of the geometric objects. Special techniques (e.g., interior point optimizers, GJK) can be used to find local solutions in these convex areas much faster than using our general-purpose method. A further approach was proposed by Dyllong and Grimm [DG07c] for proximity queries between CSG models. The authors use a special interval octree structure [DG07d] that simplifies the CSG tree by employing a technique from Duff [Duf92]: If a box associated with a node contains only a single CSG primitive, it is not subdivided further and the node is tagged *non-terminal gray*. Special distance procedures are then implemented for all primitives (e.g., box-sphere, sphere-sphere). This method is only suitable if the number of primitives is limited and they are simple enough so that explicit techniques for them exist. Therefore, Dyllong and Grimm only considered quadric primitives.

7.2 GLOBAL OPTIMIZATION

The second algorithm which is implemented directly in the UNIVERMEC framework is an interval-based global optimization method. Usually, modern global optimization algorithms resemble the historical algorithms[6], such as the *Moore-Skelboe* [Ske74] or *Ichijda-Fujii* [IF79] method, with a basic branch and bound pattern. The improvements are techniques, such as new heuristics for multisection (cf. Sect. 6.2) or new accelerating devices. Modern global optimization algorithms are described, for example, in [Bee06; HW04; Kea03]. Because the effectiveness of the new techniques depends to a significant degree on the actual optimization problem and most algorithms for global optimization are structured in a rather monolithic way, the need for more flexible approaches arose and lead to the development of such techniques as contractor programming [CJ09].

Design goals

In this chapter the flexible interval global optimization algorithm supplied with the UNIVERMEC framework is described. The algorithm itself is based on the state-of-art method described by Hansen and Walster [HW04]. This implementation is an adapted version [AKR12; KAR13] of the original one discussed in [DK10]. Three main objectives guided the development and the adaption of the algorithm: First, it should fit well into the existing UNIVERMEC environment and

6 cf. [RR88] for a thorough treatment of these

allow us to reuse techniques and methods provided by previous layers in the framework. Second, similarly to contracting programming, the set of accelerating devices should be configurable. Third, the algorithm should allow the use of modern techniques for parallelization on the CPU and the GPU, which can lead to a significant speed up on modern hardware.

Employing UniVerMeC

The first objective can be fulfilled in a straightforward manner by carefully considering the design concepts on which UniVerMeC is based and using them in the new algorithm where necessary. A corner stone of our approach is to describe objective and constraint functions using the homogeneous data-type independent function representation (cf. Sect. 5.3). This automatically allows us to employ more sophisticated arithmetics for bounding the range of a function or to take advantage of information on derivatives. Moreover, contractors provided with the framework in form of `IContractor` instances (cf. Sect. 4.4.2) can be easily integrated in the algorithm using this function representation. This meets our second objective. To fulfill the third objective parallelization, we employ OpenMP [DM98] for shared-memory CPU parallelization. Currently, GPU parallelization can be applied only in form of multiple parallel evaluations of the objective function, if the `IFunction` instance realizing the FRO supports the `IGPUEval` feature.

Related work

As already mentioned above, our algorithm is based on the algorithm for inequality constrained problems by Hansen and Walster [HW04, pp. 343-378]. In our variant, the original algorithm is subdivided into several phases which can be configured by different strategies depending on the actual problem. This flexible structure allows for application of our software to cases where monolithic general-purpose implementations run into difficulties (cf. Sect. 8.2). This configurability makes our algorithm resemble the contractor programming approach by Chabert and Jaulin [CJ09]. However, current implementations (e.g. IBEX) of this approach exhibit certain limitations. For example, they are restricted to the use of IA whereas UniVerMeC can employ more sophisticated techniques, such as TMs. While the use of TMs is in itself not new in global optimization (e.g., [MB05a]), the interoperable and interchangeable employment of *different* arithmetics and their combinations requires a solid software foundation, which is provided by UniVerMeC on the arithmetic (cf. Chap. 3) and function (cf. Chap. 4) levels. For a more detailed comparison between IBEX and UniVerMeC, see Sect. 1.3 and Sect. 4.

Previous parallelization approaches

The parallelization of interval global optimization has been considered by several authors. For example, Berner [Ber95a] investigated how such an algorithm can be run on a computer cluster with distributed memory. She laid special emphasis on how to distribute the work among the different processor nodes. A similar approach is chosen by Beelitz [Bee06] in his thesis in the scope of the recent solver

SONIC. The author considered two means of parallelization: Aside from the use of the message-passing interface (MPI) [Mpi] for systems with distributed memory, he also investigated how the multiple cores of modern processors can be utilized with the help of shared-memory parallelization in OpenMP.

The CPU parallelization in UniVerMeC is also realized in OpenMP. The approach is therefore similar to that approach of Beelitz. In contrast to his software, the algorithm in UniVerMeC does not support the use of MPI for distributed memory parallelization. Instead, we employ the GPU to perform multiple evaluations of the objective function in parallel in combination with the shared-memory parallelization. Compared to the distributed-memory parallelization, the GPU approach is currently more limited because computationally expensive steps, such as interval Newton, can not be carried out using the GPU. However, recent progress in interval computations on the GPU makes it likely that more and more steps from the optimization algorithm can be transferred to the GPU. For example, Beck and Nehmeier [BN13] investigated how an interval Newton algorithm can be implemented on the GPU and Kozikowski and Kubica [KK13] considered the application of IA and AD in conjunction with OpenCL. Both publications are important building blocks which could be used in the future for transferring further parts of interval global optimization to the GPU.

Section structure

The rest of this section is structured as follows. After describing the basic branch and bound pattern for global optimization and commonly used accelerating devices in Sect. 7.2.1, we discuss our own configurable algorithm in Sect. 7.2.2. Details on the parallelization are given in Sect. 7.2.3. Finally, a brief description of the implementation of accelerating devices and possible future improvements are given in Sect. 7.2.4.

7.2.1 *Basic Algorithm*

Abstract branch and bound pattern

The interval global optimization algorithm computes an enclosure $\boldsymbol{\varphi}^*$ for the global minimum φ^* of the optimization problem (38) with the objective function $\varphi : \mathbb{R}^n \supset \mathcal{D} \to \mathbb{R}$ and inequality constraints $g_i(x) \leqslant 0, i = 1, \ldots, m$. Similarly to other interval global optimization algorithms, our approach can be described by an *abstract branch and bound pattern* (e.g., [Kea96, p. 171]). In Alg. 6, we adapt this pattern formally for our concrete algorithm. The input is the initial search region $\mathbf{x}^{(0)} \in \mathbb{IR}^n$ and the problem (38). If $\varphi^* \in \mathbf{x}^{(0)}$, the output list of boxes $\mathcal{L}_{final}$ contains at least one box $\mathbf{x}^{(i)}$ with $\varphi^* \in \left\{\varphi(x^{(i)}) \mid x^{(i)} \in \mathbf{x}^{(i)}\right\}$. Some authors extend the above pattern slightly. For example, the original version of Kearfott uses an additional list with boxes guaranteed to contain unique minimizers.

Algorithm 6: Abstract pattern for branch and bound algorithms based on [Kea96].

Data: Search space $\mathbf{x}^{(0)}$, Optimization problem (38)
Result: List of boxes containing candidates for global optimizers
$\mathcal{L} \coloneqq \{\mathbf{x}^{(0)}\}$; $\mathcal{L}_{\mathtt{final}} \coloneqq \emptyset$;
while $\mathcal{L} \neq \emptyset$ **do**
  $\mathbf{x} \coloneqq \mathtt{head}(\mathcal{L})$; $\mathcal{L} \coloneqq \mathtt{tail}(\mathcal{L})$;
4 Try to discard $\mathbf{x}$;
5 Try to contract $\mathbf{x}$;
6 **if** $\mathbf{x}$ *fulfills the termination criterion* **then**
    $\mathcal{L}_{\mathtt{final}} \coloneqq \mathcal{L}_{\mathtt{final}} \cup \{\mathbf{x}\}$;
  else
    Perform multisection on $\mathbf{x}$ and add resulting boxes to $\mathcal{L}$;
return $\mathcal{L}_{\mathtt{final}}$;

Such a proof can be carried out with the interval Newton operator (cf. Sect. 4.3.1, 4.3.2) or other existence tests.

Feasbility

To create an actually useful algorithm, the rather abstract pattern in Alg. 6 needs to be refined. Basically, we have to replace the lines 4-6 by concrete techniques. However, before outlining the methods for this, we need to discuss under which circumstances a box or parts of it can be discarded safely due to the constraints. Safely means that we do not want to loose the guarantee that the computed global minimum is rigorous. In a constrained minimization problem such as (38), we can discard a point x from the solution set if it is infeasible, that is, $g_i(x) > 0$ for at least one $i \in \{1, \ldots, m\}$. Using range arithmetics (e.g., IA), we can test the feasibility of whole boxes. A box $\mathbf{x}$ is called infeasible if $\underline{G_i(\mathbf{x})} > 0$ for at least one $i \in \{1, \ldots, m\}$ and feasible if $\overline{G_i(\mathbf{x})} \leqslant 0$ for $i = 1, \ldots, m$. Here, G_i has to be an inclusion function of g_i. Additionally, we define the feasible region $\mathcal{F}$ of (38) by

$$\mathcal{F} \coloneqq \left\{ x \in \mathcal{D} \;\middle|\; g_i(x) \leqslant 0 \text{ for } i = 1, \ldots, m \right\}$$

and the strictly feasible region $\mathcal{F}_S$ by

$$\mathcal{F}_S \coloneqq \left\{ x \in \mathcal{D} \;\middle|\; g_i(x) < 0 \text{ for } i = 1, \ldots, m \right\} \quad .$$

Furthermore, we assume that

$$\mathring{\mathbf{x}}^{(0)} \supseteq \mathcal{F}_S \tag{52}$$

holds, that is, $\mathcal{F}_S$ lies in the interior of the starting box. This condition can be always fulfilled by adding additional inequality constraints to the problem if necessary.

Midpoint, monotonicity and non-convexity tests

Besides these basic means to discard a box, modern optimization algorithms use several further techniques. The midpoint test is used

in most interval global optimization algorithms and was proposed by Ichida and Fujii [IF79]. A box $\mathbf{x}$ is discarded if

$$\underline{\Phi(\mathbf{x})} > \overline{\varphi^*} \tag{53}$$

where $\overline{\varphi^*}$ is the currently known best upper bound on φ^*. We can update $\overline{\varphi^*}$ by

$$\overline{\varphi^*} \coloneqq \min\left(\overline{\varphi^*}, \varphi(\operatorname{mid}\mathbf{x})\right) \ , \tag{54}$$

if a box $\mathbf{x}$ is feasible. Additionally, most modern algorithms employ higher-order information to discard boxes. Two commonly used tests are the *monotonicity* and the *non-convexity* (e.g., [Rat92, pp. 43-53],[Wie97, pp. 72-73][Kah05, pp. 50-51]). The monotonicity test exploits the fact that a (local) minimum can only occur either at the boundary of $\mathcal{F}$ or at a stationary point of φ. If we assume that $\varphi \in \mathcal{C}^1(\mathcal{D})$, IA can be used to discard a box $\mathbf{x} \in \mathbb{IR}^n, \mathbf{x} \subseteq \mathcal{F}_S$, if

$$0 \notin (\nabla\Phi)_i(\mathbf{x}) \ , \qquad i = 1, \ldots, n \tag{55}$$

holds, where $(\nabla\Phi)_i$ is an inclusion function for the i-th component of the gradient of φ. In this case, φ is monotone for at least one coordinate direction, and, thus, $\mathbf{x}$ cannot contain a stationary point. For the non-convexity test second-order information is required. Therefore, we assume here that $\varphi \in \mathcal{C}^2(\mathcal{D})$. The test exploits the fact that a necessary condition for a (local) minimum $\varphi^{*'}$ inside $\mathcal{F}_S$ is, that $\varphi^{*'}$ has a neighborhood where φ is (locally) convex. Denote by $\nabla^2\Phi$ an inclusion function for the Hessian matrix of φ. A box $\mathbf{x} \in \mathbb{IR}^n, \mathbf{x} \subseteq \mathcal{F}_S$ can be discarded if

$$\overline{(\nabla^2\Phi)_{ii}(\mathbf{x})} < 0 \ , \tag{56}$$

because then the Hessian matrix cannot be semi-positive definite over $\mathbf{x}$. This implies that φ is non-convex in this box.

Fritz John conditions

Another important part of modern interval global optimization algorithms is the employment of the interval Newton method (cf. Sect. 4.3.1, 4.3.2). Usually, the Newton step is performed to check the KKT or the closely related Fritz-John conditions (e.g., [Kea96, pp. 195-198], [Bee06, pp. 104-106]). In both cases, we have to assume that $\varphi, g_i \in \mathcal{C}^2(\mathcal{D}), i = 1, \ldots, m$. Following Hansen and Walster [HW04, pp. 347-348], we use the Fritz-John conditions in our algorithm:

$$\begin{aligned} \lambda_0 \nabla\varphi(x) + \sum_{i=1}^{m} \lambda_i \nabla g_i(x) = 0 \ , & \\ \lambda_i g_i = 0 \ , \qquad & \text{for } i = 1, \ldots, m \ , \\ \lambda_i \geqslant 0 \ , \qquad & \text{for } i = 0, \ldots, m \ . \end{aligned} \tag{57}$$

In contrast to the KKT conditions, (57) introduces an additional Lagrange multiplier[7] λ_0. This leads to $m+n+1$ variables but only $m+n$ equations. Therefore, Hansen and Walster [HW04, pp. 326-330] introduce a normalization equation

$$\sum_{i=0}^{m} \lambda_i - 1 = 0 \tag{58}$$

for the case of an inequality constrained problem. Additionally, the normalization allows for deriving initial bounds on the Lagrange multiplier:

$$\lambda_i \in [0,1] \ , \qquad \text{for } i = 0,\ldots,m \ .$$

These bounds allow us to carry out the multidimensional interval Newton step using the Gauss-Seidel iteration (cf. Sect. 4.3.2) to solve the nonlinear system of equations consisting of (57) and (58).

Termination criterion

The last ingredients required to complete Alg. 6 are the multisection procedure, the sorting for $\mathcal{L}$, and a termination criterion. Any of multisection strategies and sorting criteria discussed in Sect. 6.2 is a viable choice. As for the termination criteria one simple possibility is to use a single threshold $\epsilon_x > 0$ for the multisection, and move a box $\mathbf{x}$ to $\mathcal{L}_{\mathtt{final}}$ if $\operatorname{wid}\mathbf{x} \leqslant \epsilon_x$. In more sophisticated methods, a box $\mathbf{x}$ has to fulfill more conditions. For example, three thresholds $\epsilon_x, \epsilon_\varphi, \epsilon_g > 0$ are used in the algorithm of Hansen and Walster [HW04, pp. 369-371]:

$$\operatorname{wid}\mathbf{x} \leqslant \epsilon_x \ , \tag{59a}$$

$$\operatorname{wid}\Phi(\mathbf{x}) \leqslant \epsilon_\varphi \ , \tag{59b}$$

$$\overline{G_i(\mathbf{x})} \leqslant \epsilon_g \ , \qquad \text{for } i = 1,\ldots,m \ . \tag{59c}$$

The first criterion (59a) ensures that boxes in $\mathcal{L}_{\mathtt{final}}$ are not too large, whereas the second (59b) tries to limit the width of the objective function enclosure for possible solution candidates. This is likely to improve the final bound on the global minimum. While theoretically, we have to require $\overline{G_i(\mathbf{x})} \leqslant 0$, we cannot prove this in every case because of overestimation. If a (nearly) optimal value occurs close to or on the boundary of $\mathcal{F}$, we often cannot discard it (e.g., with (53)). The third criterion (59c) guarantees that the algorithm terminates in this case nonetheless, without altering the correctness of the returned enclosure of the global minimum.

Determining $\boldsymbol{\varphi}^*$

Finally, after termination of Alg. 6, we need to determine an enclosure $\boldsymbol{\varphi}^*$ for the global minimum φ^*. This is done by processing all

7 Because the Lagrange multipliers are the variables in the *dual problem* to (38), they are sometimes called *dual variables*. The dual problem can be used to find a lower bound on φ^* of (38). A thorough treatment of duality can be found in [BV09, pp. 215-272].

boxes on the solution list $\mathcal{L}_{\mathtt{final}} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(l)}\}$ in the following way:

$$\boldsymbol{\varphi}^* \coloneqq \left[\min_{i=1,\ldots,l} \underline{\Phi(\mathbf{x}^{(i)})}, \min\left(\overline{\varphi^*}, \max_{i=1,\ldots,l} \overline{\Phi\left(\mathbf{x}^{(i)}\right)} \right) \right] \quad .$$

Note that we can improve the upper bound of $\boldsymbol{\varphi}^*$ by taking into account the already known bound $\overline{\varphi^*}$ obtained by the midpoint test. Furthermore, the returned result is a rigorous bound on φ^* even if the criterion (59c) lead to the acceptance of boxes $\mathbf{x}^{(i)}$ with $\mathbf{x}^{(i)} \setminus \mathcal{F} \neq \emptyset$. Based on the contents of the list $\mathcal{L}_{\mathtt{final}}$, we can determine if the problem is feasible. If we found at least one feasible box for (54), we know that the problem is feasible. If $\mathcal{L}_{\mathtt{final}}$ is empty after termination, it is infeasible. Otherwise, the feasibility is unknown.

Further improvements

The algorithm of Hansen and Walster, on which our own implementation is based, uses the techniques described above but in a more sophisticated manner. For example, the tests (53)-(56) are replaced by hull and box consistency (cf. Sect. 4.3.3). Additionally, several further techniques that we do not implement are suggested. Examples are a line search method [HW04, pp. 349-352] or methods that perform a first or second order Taylor expansion of φ [HW04, pp. 296-299] and formally solve the inequality

$$\Phi(\mathbf{x}) \leqslant \overline{\varphi^*} \tag{60}$$

in order to prune a box $\mathbf{x}$.

7.2.2 *A Configurable Algorithm*

Algorithm phases

After presenting the general techniques of interval global optimization, we describe the flexible algorithm that is available in UniVerMeC. Careful analysis of the basic algorithm showed that we have to distinguish three kinds of boxes[8]: Feasible boxes $\mathbf{x} \subseteq \mathcal{F}$ where (54) can be updated, strictly feasible boxes $\mathbf{x} \subseteq \mathcal{F}_S$ where techniques from unconstrained optimization such as (55) or (56) can be applied, and boxes with unknown feasibility. In the last case, techniques such as hull or box consistency (cf. Sect. 4.3.3) can be applied to the constraints $g_i(x)$ to discard infeasible parts. Our algorithm accounts for these different kinds of treatment by supplying corresponding strategy sets, which are only called with the appropriate boxes:

STRAT_POS_INFEAS Applied to boxes with unknown feasibility

STRAT_FEAS Applied to feasible boxes $\mathbf{x} \subseteq \mathcal{F}$

STRAT_STRICT_FEAS Applied to strictly feasible boxes $\mathbf{x} \subseteq \mathcal{F}_S$

STRAT_SPLIT Applied to new boxes generated during multisection

8 which are treated differently inside the algorithm

```
struct opt_worker_state_t
{
  // Upper bound
  algorithm::utils::UpperBound ubound;
  // Current box
  OptBox* cbox;
  // Objective function
  const functions::IFunction &org_obj;
  // Inequality constraints
  const FLst org_ineq_constraints;
  // Enclosure of objective function
  const functions::IFunction *obj;
  // Enclosure of inequality constraints
  const FLst* ineq_constraints;
  // Statistical information
  std::map<std::string, unsigned> discards;
};
```

Listing 4: Excerpt from the `opt_worker_state_t` structure passed to strategy elements.

STRAT_FINAL Applied to all boxes before termination

STRAT_{A,B,C,D} Called in between for all boxes

Additionally, STRAT_TMP is supplied, which is called during configuration changes on all boxes. The STRAT_SPLIT phase is, for example, used to compute bounds on the objective function, which are, depending on the actual criterion, necessary for sorting $\mathcal{L}$. STRAT_FINAL can be used to improve the bounds on φ^* during the termination (e.g., by applying hull consistency on (60)). The strategy sets STRAT_{A,B,C,D} can be used for strategy elements that need to be called in between. A *configuration change* means that the strategies are altered during runtime.

Configuration change

Currently, a configuration change can be triggered by reaching either a specific number of iterations or a specific minimal box width (for all boxes). Denote the latter threshold by $\epsilon_t > 0$. If $\operatorname{wid}\mathbf{x} \leqslant \epsilon_t$ holds for a box $\mathbf{x}$, it is no longer subdivided but stored in a temporary list $\mathcal{L}_{tmp}$. When a configuration change occurs, the strategies from STRAT_TMP are applied to all boxes of $\mathcal{L}_{tmp}$. After that, all boxes from $\mathcal{L}_{tmp}$ are moved back into $\mathcal{L}$ and ϵ_t is decreased. This kind of behavior is similar to our sub-routine in the ϵ-distance algorithm (cf. Sect. 7.1.1). We introduce LIFO-like elements into the sorting of $\mathcal{L}$ also in this case

Strategy elements

Up to this point, we did not explain the components of which strategy sets consisted. Each set can either be empty or consist of one or more *strategy elements*. When our solver processes a phase, it calls the strategy elements in the order preconfigured by the user. A strategy element works on the local *worker state* of the solver. The relevant

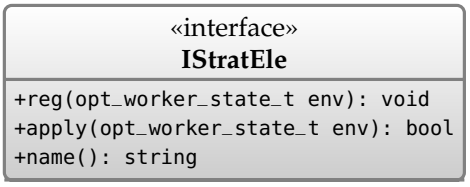


Figure 34: The strategy element interface `IStratEle` is used to apply members of the strategy sets to the current box.

parts of the C++ structure defining the worker state[9] are given in listing 4. As shown in the listing, the strategy element can alter the `OptBox` class[10] storing the actual box together with some additional information (e.g., feasibility status, active constraints, Lagrange multipliers). Further, the known upper bound $\overline{\varphi^*}$ and some statistical information can be also changed. Note that the structure depicted in the listing contains two representations for both the objective function and the inequality constraints. While the entries `org_obj` and `org_ineq_constraints` store the original functions supplied by the user, additional enclosures (cf. Sect. 4.4.2) are stored in `obj` and `ineq_constraints`, respectively. This allows us, for example, to use mean-value forms instead of natural extensions in order to obtain narrower range bounds. A strategy element has to implement the interface `IStratEle` depicted in Fig. 34. The main method of which is called `apply`. It can alter the local worker state (e.g., prune the current box). The return value of `apply` determines whether the current box can be discarded completely.

Algorithm configuration

Before outlining the final algorithm, we explain how to configure it. An excerpt from the C++ structure defining the algorithm configuration is given in listing 5. The structure is provided by users with the help of an instance of the `IOptStrategy` interface. Basically, the `phase_config_t` structure consists of nine ordered lists containing the concrete strategy elements to be applied, the current multisection strategy, a termination criterion and information about the time of the next configuration change. The last item is specified using the `min_width` value corresponding to ϵ_t and the variable `max_it` denoting the maximum number of iterations allowed with the current configuration. Multisection strategies are described through the `IMultiSection` interface (cf. Fig. 30, Sect. 6.2). Similarly to Hansen and Walster [HW04, pp.. 360-371 (step 9)], we call the termination criterion after *each* strategy element.

This criterion is made available by a function pointer[11], which takes the local worker state as parameter and returns a value from the enu-

9 We call this structure worker state because if the algorithm is run in parallel in multiple threads, each thread has its own state (cf. Sect. 7.2.3).

10 It is derived from the `core::GappedBox` class described earlier (cf. Sect. 4.4.2).

11 Note that we avoid using member function pointers for stylistic reasons. Instead we employ an ordinary C function pointer, that is, `term_crit` needs to point either to a global function or a static member.

```
enum TERM_RESULT { NONE, SOLUTION, REDUCED, TMP_REMOVED };
typedef TERM_RESULT (*f_term_t)(const
    opt_worker_state_t&);
struct phase_config_t
{
  // strategies for the splitting step
  strat_ele_lst_t psplit;
  // strategies for PHASE_A
  strat_ele_lst_t pa;
  // strategy elements for PHASE_POS_INFEAS
  strat_ele_lst_t pinfeas;
  // ...
  // strategy elements for PHASE_FINAL
  strat_ele_lst_t sol_strats;
  // GPU strategy elements
  gpu_strat_ele_lst_t gpu_psplit;
  // multisection strategy
  decomp::IMultiSection<OptBox>* section;
  // pointer to termination criterion
  f_term_t term_crit;
  // minimum width for subdivision in this phase
  core::arith::mreal min_width;
  // maximum number of iterations in this phase
  unsigned max_it;
};
```

Listing 5: Excerpt from the `phase_config_t` structure responsible for configuring the optimization algorithm.

meration `TERM_RESULT`. This returned value determines further course of action taken by the main algorithm in the following way:

- NONE: No action needed.
- SOLUTION: Box is moved to $\mathcal{L}_{\mathrm{final}}$.
- TMP_REMOVED: Box is moved to $\mathcal{L}_{\mathrm{tmp}}$.
- REDUCED: Box is moved back into $\mathcal{L}$.

In the last three cases, the main loop of the algorithm is restarted. Our supplied standard strategy uses (59) to check whether a final solution has already been found. In addition, it uses the ϵ_t threshold to decide whether the box should be moved to $\mathcal{L}_{\mathrm{tmp}}$. The last return value REDUCED indicates whether the strategy elements applied upto the current point led to *sufficient progress*. If this the case, the (reduced) box is added to $\mathcal{L}$ without a multisection and the main loop is restarted. The advantage of this approach is that we apply computationally cheap strategy elements first, and if they lead to a sufficient progress, we can reapply them on the reduced box and avoid expensive steps until they are really required. To check sufficient progress, Hansen and Walster proposed the criterion (41) that we already used in the scope of contracting trees (cf. Sect. 6.1.2).

Algorithm 7: Sequential version of the configurable interval global optimization algorithm in UniVerMeC.

Input: Search region $\mathbf{x}^{(0)}$, Optimization problem (38), Strategy configuration
Output: Enclosure $\boldsymbol{\varphi}^*$ of minimum φ^*, List of candidate boxes

```
Get initial configuration;
2  L := {x^(0)};
   L_tmp := ∅;
   S := RUNNING;
5  φ̄* := ∞;
6  while S == RUNNING do
7      if L = ∅ then
           Get configuration for next phase;
           forall the x' in L_tmp do
               apply_strat_eles(STRAT_TMP, x');
               L := L ∪ {x'};
12         L_tmp := ∅;
13         if L = ∅ then S := FINISHED; continue; ;
14     x := head(L); L := tail(L);
       apply_strat_eles(STRAT_A, x);
       if ¬feasible(x) then
           apply_strat_eles(STRAT_POS_INFEAS, x);
       apply_strat_eles(STRAT_B, x);
       if feasible(x) then apply_strat_eles(STRAT_FEAS, x) ;
       apply_strat_eles(STRAT_C, x);
       if strictly_feasible(x) then
           apply_strat_eles(STRAT_STRICT_FEAS, x);
23     apply_strat_eles(STRAT_D, x);
24     L_split := split(x);
       forall the x' in L_split do
           apply_strat_eles(STRAT_SPLIT, x');
27         L := L ∪ {x'};
28 forall the x' in L_final do
       apply_strat_eles(STRAT_FINAL, x');
   φ̲* := min_{x∈L_final} Φ(x);
   φ̄* := min(φ̄*, max_{x∈L_final} Φ̄(x));
32 return (φ*, L_final)
```

Algorithm description

In Alg. 7, the sequential version of our approach is shown. It requires the search region $\mathbf{x}^{(0)}$, the optimization problem (38), and the strategy configuration[12] as input. The algorithm returns an enclosure $\boldsymbol{\varphi}^*$ of the global minimum φ^* and a list of candidate global minimizers. For checking whether a box can be guaranteed to be feasible or strictly feasible, the predicates[13] `feasible` and `strictly_feasible` are used. The main parts of the algorithm are encapsulated into the `apply_strat_eles` function. This function takes the current strategy set and the current box as arguments. Every strategy element from the set is applied to the current box. After each application of a strategy element, a user-defined termination criterion is used to determine whether the box can be discarded, moved to $\mathcal{L}_{\mathtt{tmp}}$ or $\mathcal{L}_{\mathtt{final}}$, or sufficient progress was made. If any of these is the case, the appropriate action is taken, and the current loop is restarted.

The algorithm starts with initializing $\mathcal{L}$, $\mathcal{L}_{\mathtt{tmp}}$, $\overline{\varphi^*}$ and the status variable S in lines 2-5. The status variable can take the values RUNNING or FINISHED which indicate whether the algorithm is still working in the main loop or finished its work there. The main loop is entered in line 6. If there is no box in $\mathcal{L}$ the configuration is changed, the strategy for the boxes in $\mathcal{L}_{\mathtt{tmp}}$ is carried out, and the boxes are moved back into $\mathcal{L}$ (lines 7-12). If $\mathcal{L}$ is still empty, the algorithm leaves the main loop, applies the strategies for the final phase, and returns the global minimum (lines 28-32) before terminating. Otherwise, the algorithm takes the first box $\mathbf{x}$ from $\mathcal{L}$ and applies the strategy elements depending on the feasibility of $\mathbf{x}$ (lines 14-23). After that, the multisection is performed and the new boxes are attached to $\mathcal{L}$ (lines 24-27).

7.2.3 *Parallelization of the Algorithm*

Reentrancy

The basic techniques discussed in Sect. 7.2.1 can be carried out in parallel on different subdivision boxes. For example, two interval Newton steps to solve (57), (58) for the boxes $\mathbf{x}^{(i)}, \mathbf{x}^{(j)}, \mathbf{x}^{(i)} \neq \mathbf{x}^{(j)}$ do not have any data dependency. Therefore, our parallelization approach basically consists in launching multiple threads that execute the main loop of Alg. 7 independently of each other. This approach is only suitable if the strategy elements called through the `apply_strat_eles` function are *reentrant*, that is, it is safe to call them from multiple threads in parallel on different `opt_worker_state_t` instances.

The implementation of reentrant strategy elements is encouraged by the architecture of UniVerMeC. Note that the `apply` method in the `IStratEle` interface (cf. Fig. 34) is declared `const` in our implementation. This ensures that a call to this method can not modifiy any

12 In the implementation, this is an instance of the `IOptStrategy` interface.

13 Note that if the predicates return *false*, this does not imply that a box is *not* (strictly) feasible but only that the (strict) feasibility could not be proven.

Table 16: Handling of variables in the parallel version of the interval global optimization algorithm with regard to thread synchronization.

VAR.	SHARED / COPY	SYNC. METHOD	SYNC. TIME
$\mathcal{L}$	shared	critical section	
$\mathcal{L}_{tmp}$	private copy	critical section	config. change
$\mathcal{L}_{final}$	private copy	critical section	finalization phase
$\overline{\varphi^*}$	private copy	critical section	each iteration
IT	shared	atomic instr.	
S	shared	atomic instr.	

data stored in the `IStratEle` instance[14]. In general, we assume that a method declared `const` performs only *read* operations on shared data, and is, therefore, reentrant from multiple threads. Thus, a strategy element should modify only the passed `opt_worker_state_t` structure. To allow the straightforward and easy implementation of `const` correct functions most methods in interfaces provided by UniVerMeC are also declared to be `const`[15]. In the rest of our work, we assume that all strategy elements used are reentrant.

Shared variables

While we can safely postulate that the strategy elements called by different threads share no data, certain other data structures used in Alg. 7 need to be shared. Usually, the required *wall clock time* to solve a problem decreases with additional processors[16] (i.e., parallelization yields good results), if each of these processors performs a *meaningful* task and the *synchronization overhead* is not too high [Ber95a, p. 74]. Often both conditions are in conflict with each other if it is not possible to subdivide the main task a priori into equal subtasks that can be assigned to each processor and require no or very little communication with each other.

In global optimization, we do not know at the beginning of the algorithm how the work is distributed inside the search region $\mathbf{x}^{(0)}$[17]. Therefore, an a priori subdivision of the search region might lead to the case in which one processor has to carry out all the work, suppressing any potential speed up through parallelization. To avoid this, we have to perform a dynamical work sharing during runtime. While this is a non-trivial task in the case of parallelization with distributed memory (e.g., [Bee06, pp. 125-139]), it is possible to share $\mathcal{L}$ between all threads in our case. The advantage is that each thread

14 If we assume that keywords such as `mutable` are avoided during the implementation of the actual instances of these interfaces. The use of such keywords might undermine the guarantees made by declaring a member function `const`.

15 For example, all methods provided by the interfaces `IVFunction`, and `IContractor`.

16 In the optimal case, the decrease is linear. However, the maximum gain depends according to *Amdahl's law* on how much of the program needs to be carried out sequentially [CJP08, pp. 33-34].

17 This depends, for example, on where global or local optimizers are situated in $\mathbf{x}^{(0)}$.

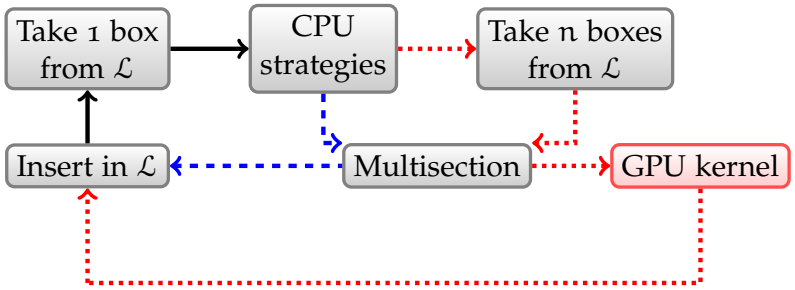


Figure 35: Integration of GPU strategy elements into the interval global optimization algorithm. The black paths are followed by all threads, dashed blue paths only by the CPU threads and dotted red paths by the CPU thread feeding the GPU with data.

gets the box most likely to contain the global minimum[18]. However, the access to $\mathcal{L}$ has to be synchronized (e.g., using a critical region). Therefore, this approach is only suitable for a low number of processors. Our tests in [DK10] showed that the *speedup*[19] started to decline beginning with 4 processors.

Similarly to $\mathcal{L}$, the lists $\mathcal{L}_{\texttt{tmp}}$ and $\mathcal{L}_{\texttt{final}}$ can be accessed from different threads. For these lists, a local copy can be employed for each thread. These copies are only merged if necessary, that is, during a configuration change or after leaving the main loop. Additionally, it is crucial to share $\overline{\varphi^*}$ in order to allow fast pruning of the search region by (53). To avoid a large number of critical sections, each thread has a local copy of $\overline{\varphi^*}$, which is synchronized with the global one only once in each iteration. On the one hand, this limits the overhead to one synchronization per iteration. On the other hand, it ensures that improved values for the crucial bound $\overline{\varphi^*}$ are distributed relatively fast among all threads. Other data such as the iteration counter IT or the status variable S can be shared by employing *atomic* operations. The above discussion is summarized in Tab. 16.

GPU integration

As a second possibility for parallelization of our algorithm, we allow for the employment of the GPU in UniVerMeC. This has already been indicated in listing 4, where a list of strategy elements to be run on the GPU (`gpu_psplit`) can be defined. The use of GPU strategy elements is restricted currently to the `STRAT_SPLIT` phase. Due to the lack of universal libraries for verified computations on the GPU, our main use-case is the evaluation of computationally expensive objective functions over those boxes generated in the split phase. Such a strategy element can be implemented straightforwardly by exploiting the `IGPUEval` interface provided at the function layer (cf. Sect. 4.4.2). Note that the `apply_strat_eles` function has a slightly modified semantics for GPU strategy elements. Instead of a single box $\mathbf{x}'$, it de-

18 depending on the used sorting heuristic

19 Basically, the parallel speedup $S(p) = \frac{T_1}{T_p}$ is the quotient of the required wall clock time T_1 on one processor and the time T_p on p processors [CJP08, pp. 33-34].

Table 17: Strategy elements used in the default strategy of the global optimization algorithm.

NAME	DESCRIPTION
`bounder_t`	Bound on the objective function using IA
`midpoint_t`	Midpoint test (53)
`feas_t`	Feasibility test based on IA
`ilie_cbounder_t`	Feasibility update and pruning of infeasible parts using an ILIE
`bc_const_t`	Pruning infeasible parts using box consistency
`u_ubound_t`	Updating $\overline{\varphi^*}$ with the help of (54)
`ilie_bounder_t`	Updating the bound on the objective function and pruning of the box using (60) with an ILIE
`bc_ubound_t`	Prune box using (60) with box consistency
`bc_grad_t`	Apply box consistency on gradient checking (55)

livers multiple boxes at once. In fact, additional multisections are performed before starting the GPU computation in the current implementation. This ensures that the GPU's computational resources are utilized better. The integration of the GPU steps into the parallel CPU version is shown in Fig. 35. One CPU thread is responsible for transferring data from and to the GPU. Its execution path is denoted by the dotted red line. Notice that this thread also performs the normal activities of the other CPU threads. If non-blocking operations[20] are used by GPU strategy elements, a better overall utilization of the available computational resources is achieved.

7.2.4 *Provided Strategy Elements and Possible Enhancements*

Implementing strategy elements

As a usage example for the algorithm, we provide a default strategy (`DefaultStrategy`) implementing the strategy interface `IOptStrategy` and a number of strategy elements (cf. Tab. 17). They make use of the techniques provided by the framework to implement variants of the interval global optimization methods described in Sect. 7.2.1. These predefined strategy elements rely heavily on the already existing infrastructure of UniVerMeC. Thus, they are comparatively easy to implement.

Consider, for example, the implementation of `ilie_bounder_t` given in listing 6. This strategy element tries to prune the current box by formally solving the equation (60) using an ILIE. The advantages of using a uniform framework for implementation are clearly visible in the listing: To construct an ILIE, we need to evaluate φ over $\mathbf{x}$ using

20 For example, the `IGPUFuture` approach described in Sect. 4.4.2.

```
struct ilie_bounder_t : public IStratEle
{
  virtual bool apply(opt_worker_state_t &env) const
    {
      using namespace core::arith;
      functions::encl::ILIEEncl encl(*env.obj);
      std::unique_ptr<functions::IFunction>
          lobj(encl.encl(env.cbox->ref_X()));
      if(lobj) {
        interval bound((*lobj)(env.cbox->ref_X()));
        env.cbox->bound() &= bound;
        if(!is_infinity(env.ubound()))
          return lobj->features().custom_consistency()->
            get(0).prune(*env.cbox,
                interval(-infinity(),env.ubound()));
      }
      return false;
    }

  virtual std::string name() const
    {
      return "ILIE Bounder";
    }
};
```

Listing 6: Strategy element for prunig a box by formally solving (60) using an ILIE.

AA. In the listing, this step is performed implicitly by constructing an enclosure object that returns a new function representation. It delivers new bounds on φ over $\mathbf{x}$ in AA. Additionally, this new function object provides a custom contractor that formally solves (60) to prune $\mathbf{x}$. Basically, all enclosure and contractor techniques supplied inside UniVerMeC can be used in a similar manner without too much effort, allowing for the maximum flexibility of the actual algorithm configuration used in global optimization.

Relationship to state-of-the-art solvers

The algorithm we consider in this section is not intended to be a complete replacement for state-of-the-art solvers such as COCONUT [Sch04], GlobSol [Kea03] or SONIC [Bee+04]. Instead, it can serve well as a platform for testing different techniques in an easy way. Here, UniVerMeC ensures flexibility with respect to employed techniques thanks to its uniform treatment of different range arithmetics. This feature is not provided by the other solvers. Additionally, there are several special cases in which choosing UniVerMeC over the other solvers is advantageous. For example, the GPU parallelization is currently supported only by UniVerMeC. Another possibility is for situations where other algorithms available within UniVerMeC should be applied on the same model description. Such a situation is discussed in Sect. 8.2. Here, the optimization algorithm is combined with external IVP solvers to validate the solution. An example for an even deeper integration is investigated in an ongoing master

thesis [Pus13], where the value of the objective function is calculated using verified IVP solvers, and, thus, the IVP solvers are called from within the optimization algorithm. Note that no modification of our algorithm is necessary because our function representation allows us to treat even such cases as black boxes.

Future enhancements

Since our implementation is not intended to be a replacement for state-of-the-art-solvers (asides from special cases) but a test environment for different and new techniques, further improvements we discuss in the following concentrate on points relevant for this goal. The extension of the GPU support is very interesting because it is currently an outstanding feature. Using the GPU throughout major parts of the algorithm might allow us to solve complicated problems, which can be dealt with currently only by using expensive computer clusters[21], even with state-of-the-art solvers, on much less expensive systems. Furthermore, making better use of the more sophisticated arithmetics might improve the algorithm results significantly.

However, it is difficult to give general directions about when to use a certain arithmetic or range-enclosure technique to obtain better results. For example, Kearfott and Arazyan [KA00] conclude their examination of TMs in the scope of global optimization by stating that their usage "is sometimes helpful and sometimes not helpful". Kearfott and Walster [KW02] highlight several cases where TMs lead to better results in interval global optimization and investigate whether there is a good heuristic to determine when to use them. One investigated heuristic consists in randomly choosing some boxes and evaluating them with the help of both IA and TMs to judge whether the increased computational effort of using TMs is worthwhile. Implementing such a sampling procedure during the strategy phase changes of our global optimization algorithm might be an interesting extension for the future. Additionally, the use of techniques developed especially for arithmetics other than IA (e.g., ILIEs or affine reformulation [NMH10]) should be investigated further. Combining different possibilities might provide insight into which techniques result in improvements and are, as such, interesting for incorporation into the state-of-the-art solvers.

7.3 INTERFACING OF EXTERNAL SOLVERS

Relation between UniVerMeC *and external solvers*

Besides acting as platform for the direct implementation of algorithms as shown in the two previous sections, UniVerMeC makes it possible to access already existing external solvers. In this case, the goal is to allow third party solvers to work on UniVerMeC's internal model and problem descriptions discussed in Chap. 5. Therefore, an external solver interfaced to the framework does not employ the whole

21 See, for example, the description of the computer systems employed to solve some realistic test problems with SONIC [Bee06, p. 146].

infrastructure but as a rule, uses only the components contained in the problem description.

Difficulty factors for employing external solvers

The main task while implementing an interface to an external solver is centered on software analysis. It is necessary to examine the problem input format required by the external library to be interfaced and make a connection between those components and their counterparts in UNIVERMEC. An often employed design pattern in this scope is the *adapter pattern* [Gam+95, pp. 139-150]. In our experience, the difficulty of implementing such an interface for numerical solvers depends mostly on two factors[22]:

1. Whether the external solver is supplied with a well-defined interface that can be utilized by UNIVERMEC to provide the problem description.

2. The availability of an appropriate (in the best case standardized) format for exchanging numerical quantities.

In general, the first point is the responsibility of the developer of third party software. An important aspect that should have been considered during the design process of the external software is, from our point of view, the reduction of external dependencies in the software interface. For example, quantities such as matrices and vectors should be represented by built-in types of the chosen programming language (e.g., by `std::vector` in C++ or C-style arrays). The reason is that using special class libraries (e.g., MTL [GL]) in the public interface would break encapsulation and put the additional burden on the user of learning them and possibly dealing with incompatibilities between different class libraries in order to use the third party library.

This aspect is connected with the second factor mentioned above. Custom approaches to representing numerical quantities should be abandoned in favor of the standardized ways. For example, built-in types for the standardized IEEE 754-2008 FP representations are available in many of modern programming languages. These should be used wherever possible. A serious problem in the case of a verified solver employing range arithmetics is that a standardized format for exchanging verified data is not available at the moment. At least for IA, the adoption of the upcoming standard IEEE P1788 would make things easier in this regard. Until then, exchanging intervals by their endpoints using IEEE 754-2008 types is an alternative.

Related work

In general, the interfacing or combination of different tools does not seem to receive much attention in the verified context. An exception is the environment VERICOMP [AR12] (cf. Sect. 1.3), which allows users to compare different verified IVP solvers. One reason for the lack of interoperable tools might be that "[...] solvers have disparate interfaces, which makes developing a unified comparison platform a

22 There are also other factors leading to difficulties. For example, if a program can be compiled only with outdated compilers, written in an obscure language, and so on.

challenge" as stated in [AR12]. VERICOMP is used to run the integrated IVP solvers either with a set of predefined problems or with problems provided by the user. In the second case, VERICOMP tries to recommend optimal settings based on the already solved problems under certain optimatility conditions derived by the authors. As already explained in Sect. 1.3, UNIVERMEC follows a more general approach and, thus, can be used to implement applications such as VERICOMP with less effort. Other examples interfacing different verified solvers are is the comparison platform for global optimization tools described in [DFS11] (cf. Sect. 1.3). Furthermore, the COCONUT environment [Sch04] is capable of calling external solvers. Similarly to VERICOMP both follow a more specific approach compared to UNIVERMEC.

In contrast to the integration of different verified solvers, the combination of rigorous solvers with non-rigorous implementations received more attention. For example, several FP solvers are applied in scope of the global optimization package GLOBSOL to find points that lie near global or local solutions, and can later be verified in a rigorous manner with interval methods [Kea+04]. In UNIVERMEC, we want to support the employment of both rigorous and non-rigorous external solvers. However, in contrast to existing software packages, we want to supply building blocks that can be combined to solve specific problems. Examples for such combinations are given in Sect. 7.1.3, where FP solvers are used to accelerate the distance computation algorithm or in Sect. 8.2 where parameters of mathematical models are identified using an external FP solver and then validated with the help of an external rigorous IVP solver. In the latter case as well, the interval global optimization method from Sect. 7.2 can be applied to identify the parameters, which are validated then once again by an external IVP solver.

7.3.1 *ValEncIA-IVP*

Required input

VALENCIA-IVP [RA11] is a verified solver for obtaining verified enclosures on the solution of IVPs such as the ones defined in Sect. 5.2 (cf. Def. 18). It computes an enclosure of the exact solution to an IVP over a certain time interval by an algorithm derived from the Picard iteration. Basically, it relies only on enclosures of the range of the right-hand side of the problem at points of time computed using a constant step size and over intervals between them. Additionally, it requires bounds on the Jacobian matrix of the right side over the same points. The Hessian matrix is needed only for the sensitivity analysis, which we do not support currently. All three types of information can be provided using the function layer (cf. Sect. 4.4).

Interfacing a standalone solver

VALENCIA-IVP was designed to work as a standalone application and not as a library which can be accessed from external programs.

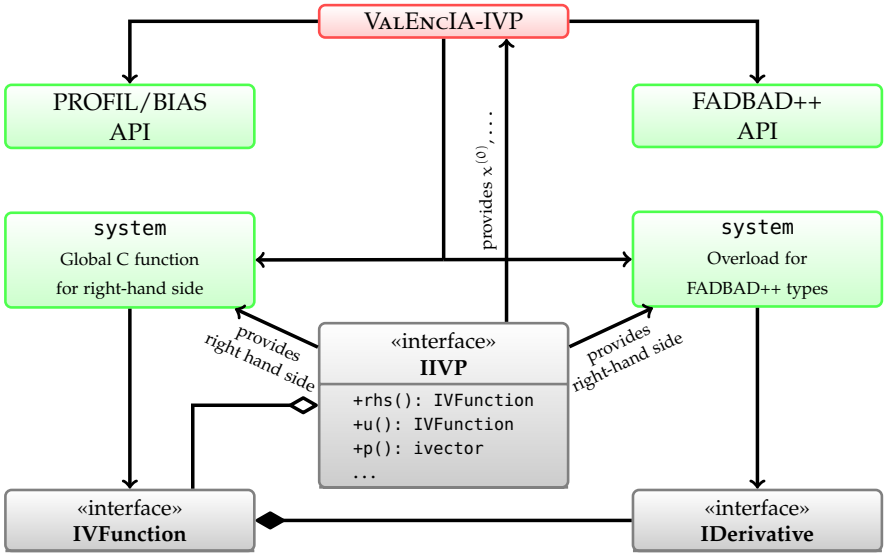


Figure 36: ValEncIA-IVP is intended for standalone usage. The green compability layer was added which mapped the functions used by the solver to their UniVerMeC counterparts. Furthermore, the specification of an IVP through the `IIVP` interface is mapped to ValEncIA-IVP's internal representation.

Users have to specify their problem by adjusting global functions representing the right-hand side of the problem and by setting the initial values and parameters in the `main` function. Besides, the solver is permanently coupled with the libraries PROFIL/BIAS [Knü94] for IA and Fadbad++ [SB] for AD. This makes altering the code of the solver inevitable if we are to employ it in the context of UniVerMeC, a dynamic environment where problems to solve can be exchanged at runtime.

Compatibility layers

A full decoupling of ValEncIA-IVP from its underlying libraries would result in a reimplementation of major code portions. To avoid this and to enable the easy integration of new versions, we implement a compability layer which maps the APIs used by ValEncIA-IVP to those provided by UniVerMeC. As shown in Fig. 36, the compability layer consists of four components. The PROFIL/BIAS and FADBAD++ components provide the parts of the API of the respective packages used by ValEncIA-IVP. In case of PROFIL/BIAS, it is sufficient to write small wrappers which map its functions and types to their respective counterparts of the `core` layer of UniVerMeC. The actual library used to carry out interval computations is then the one employed by UniVerMeC. In turn, the FADBAD++ emulation consists only of a `fadbad::F` type which allows us to store values for the derivatives and to return them through the usual FADBAD++ API.

Specifying the right-hand side

In the original implementation, the right-hand side of the IVP to be solved by ValEncIA-IVP is specified by the global templated function `system`. It is called either with FP, PROFIL/BIAS IA, or FADBAD++ types to evaluate the right-hand side or its Jacobian matrix. Our layer

provides two overloads for these functions, which return either the value of the right-hand side or our `fadbad::F` types containing the values of the derivatives. The actual computations of the values are carried out by the `IVFunction` instance representing the right-hand side of the IVP. In contrast to the original implementation, the adjusted solver can now work with IVPs, the derivatives of the right sides of which are either described analytically by a closed-form expression or derived by AD. Furthermore, the IVP can be changed at runtime and need not be specified at compile time.

Changes to VALENCIA-IVP

Up to this point it was not necessary to change the VALENCIA-IVP code itself. Still the following slight alterations of the code are unavoidable:

1. Header references to the original PROFIL/BIAS and FADBAD++ headers are removed and the new headers for the compatibility API are added.
2. The initialization functions for the initial state values, parameters and so on are moved from the `main` method to an external one, the functions are altered in such a way, that they read these values from the current `IIVP` instance.
3. The `main` method is renamed.

Fortunately, none of these requires major changes in VALENCIA-IVP itself. Thus, there is a high probability that a new version of the solver can be added to UNIVERMEC in a straightforward manner.

Usage

After adding the compatibility layer, we can solve an IVP encoded by the `IIVP` instance `problem` by calling the function:

```
extras::interfaces::solve_ivp_valencia(problem, stop_time,
    step_size, filename);
```

The values `stop_time` and `step_size` determine the integration interval and the step size of the solver. Results are written by VALENCIA-IVP in the file specified by `filename`.

Further improvements

Note that there is room for improvements in our way of incorporating VALENCIA-IVP. It might be an interesting idea for the future to make the state enclosure at `stop_time` and, if possible, the enclosures of the intermediate integration steps accessible directly inside the framework without needing to read them from the file. However, this would require a more thorough adaption of the solver's code base. Another possibility for improvements lies in the fact that neither our interface nor VALENCIA-IVP itself are thread safe at the moment since several global variables have to be used. To be able to use them in a multithreaded mode, it is necessary to revise the whole code thoroughly.

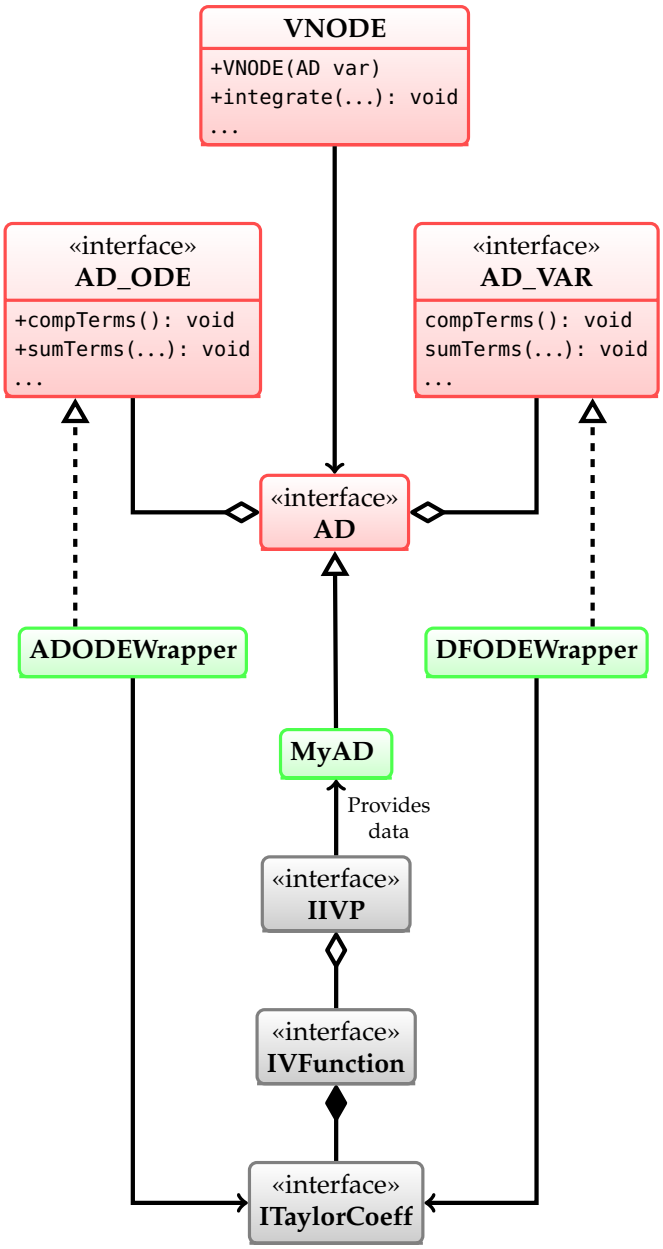


Figure 37: The red interfaces are used by VNODE-LP to access the actual IVP. To use VNODE-LP inside UniVerMeC we provide the green classes implementing these interfaces. The necessary information is obtained by them from the underlying IVP represented by the `IIVP` interface.

7.3.2 *VNODE-LP*

Literate programming

VNODE-LP [Nedo6] is a verified solver for obtaining rigorous bounds on the solutions to IVPs. A distinctive feature of VNODE-LP is that it was developed using the *literate programming* approach [Knu84]. This approach allows to combine code and documentation. In the case of a verified numerical program, this has the advantage that there is a clear correspondence between the underlying mathematical theory and its actual code. In this way, finding of errors in the implementation is facilitated.

Providing Taylor coefficients

VNODE-LP is based on the Taylor expansion principle and computes the Taylor coefficients of the solution from the right-hand side of the IVP. Therefore, the solver requires these coefficients together with those of the corresponding *variational equation*[23]. As shown in Fig. 37, two abstract interfaces are used by VNODE-LP to access the IVP to be solved. The first, `AD_ODE`, provides the Taylor coefficients of the solution, and the second `AD_VAR`, those of the variational equation. Both interfaces provide these data using VNODE-LP's internal interval type[24]. These design principles of VNODE-LP allow for its use as an external library in UNIVERMEC. Basically, we only have to provide implementations for the interfaces `AD_ODE` and `AD_VAR`. These implementations obtain the necessary coefficients through the `ITaylorCoeff` interface from the function layer (cf. Sect. 4.4.2) and convert the data from UNIVERMEC's internal interval type to VNODE-LP's one. A lossless conversion between the types is possible since both represent intervals by their `double` endpoints. In the same way, the initial values, the starting time, and so on can be communicated through the solver's public interface.

Usage

VNODE-LP can be called from inside UNIVERMEC by the following function:

```
core::arith::ivector solve_ivp_vnodelp(const ivp::IIVP
    &ivp, const core::arith::mreal &stop,
    std::vector<core::arith::ivector> &sresults,
    std::ostream &os = std::cout, core::arith::mreal
    sub_step = -1.0, const ivp_callback_t *cb = 0, bool
    cparams = false);
```

The first three parameters specify the problem in the form of an `IIVP` instance, the integration end time and a vector to store (intermediate) results. For convenience, a fourth argument is an `ostream` instance which is used to display intermediate results and the solution (e.g., for plotting with GNUPLOT). The `sub_step` (s_{step}) variable is used to subdivide the integration interval $[t_0, t_{end}]$ into subintervals $[t_0 + i \cdot s_{step}, t_0 + (i+1) \cdot s_{step}], i = 0, \ldots, k$ where $t_0 + (k+1) \cdot s_{step} = t_{end}$ for which intermediate results are generated. Furthermore, users can

23 Basically, these are the coefficients of the derivative (Jacobian matrix) of the right-hand side.

24 The interval type is provided by an external library used by VNODE-LP.

Table 18: Overview of the external solvers interfaced with UniVerMeC and of information required by them. The rigorous solvers are given in the first part of the table and the non-rigorous in the second.

SOLVER	TYPE	REQUIRED INFORMATION
ValEncIA-IVP	IVP	Initial values, FP and interval bounds of the right-hand side and its Jacobian matrix
VNODE-LP	IVP	Initial values, interval Taylor coefficients of the solution and the variational equation
ODE	IVP	Initial values, FP values of the right-hand side (and optionally its Jacobian matrix)
VODE	IVP	Initial values, FP values of the right-hand side (and optionally its Jacobian matrix)
IpOpt	Opti. (38)	Starting point, FP values for the objective function and constraints/their derivatives, FP values for Hessian matrix of Lagrangian

specify a callback function `cb`, which is called after the integration over each subinterval is completed. It can be used to manipulate intermediate results, parameters, or solver settings. The last parameter `cparams` indicates whether parameters of the IVP to be solved change during the integration. In particular, this is the case, if the function $u(t)$ is piecewise constant. Note that a parameter change is only allowed in between subintervals specified by `sub_step`. Furthermore, a small value for `sub_step` may lead to a slowdown of VNODE-LP's integration process because its automatic stepsize control is only applicable inside the subintervals. Our side of the VNODE-LP implementation is thread safe.

7.3.3 *Other Solvers*

Interfacing FP solvers

Besides ValEncIA-IVP and VNODE-LP, several further solvers were interfaced with UniVerMeC. A complete overview of the available solvers is given in Tab. 18. Among these are the two non-rigorous IVP solvers ODE [SG75] and VODE [BBH89]. Additionally, the interior point optimization algorithm IpOpt [WB06] is supported. Here, we do not describe the interfacing of non-rigorous solvers in detail because this process is straightforward in most aspects (cf. [KAR14] for details on the VODE interface). The reason is that a commonly acknowledged standardized format for data exchange is available with

IEEE 754-2008 and that, additionally, less information was needed by the examined non-rigorous IVP solvers compared to the rigorous ones. The examined packages required only values for the right-hand side (and optionally the Jacobian matrix).

7.4 CONCLUSIONS

Ways of adding algorithms

In this chapter, we discussed how algorithms can be implemented inside of UNIVERMEC and how UNIVERMEC can be used to employ algorithms implemented by third party developers. In the former case, the newly implemented algorithm can be tightly connected to the underlying framework to benefit from already implemented data-structures and methods. In the latter case, the third party library is likely to make use only of the uniform model description provided by the framework through adapters. Each way has its advantages depending on the users' goals. If the goal is to analyze a problem described using UNIVERMEC's modeling layer by combining different solver libraries, it is sufficient to provide the corresponding interfaces to them. Alternatively, the framework can be used to implement new algorithms for verified computations. In this case, the building blocks provided at the different layers of the framework can be reused in order to speed up implementation of new algorithms. We provided examples for both use-cases in this chapter.

Distance computation

The first discussed algorithm was ϵ-distance (cf. Sect. 7.1). This is a novel distance computation approach for deriving a rigorous enclosure on the distance between two possibly non-convex geometric objects. It was developed in scope of this thesis and its implementation makes heavy use of techniques supplied by UNIVERMEC. Besides using the modeling layer (cf. Sect. 5.1) to describe the geometric objects, the algorithm works on hierarchical space decompositions constructed by the techniques outlined in Sect. 6.1. To speed up the computations, the algorithm can make use of normals or FP solvers. Both are provided inside the integrated environment of UNIVERMEC, making the implementation fairly straightforward.

Interval global optimization

In Sect. 7.2, an interval global optimization algorithm was discussed as the second example. This algorithm benefits greatly from our homogeneous data type independent function representation (cf. Sect. 4.4). Moreover, it can employ such techniques as contractors, which are readily available inside the framework. Besides showing how standard methods from global optimization can be implemented using the already existing UNIVERMEC building blocks, we discussed how the monolithic global optimization approach of Hansen and Walster [HW04] can be modified to fit into the dynamic and modular environment. Additionally, we explained how the support for GPU computations in UNIVERMEC can be exploited inside this algorithm.

Third party solvers

The final Sect. 7.3 discussed ways to interface third party imple-

mentations of solvers. Interfaced solvers do not make use of major portions of the framework. Usually, they employ only the uniform model description (cf. Sect. 5), which is accessed with the help of an adapter to be provided by the users. The main focus of presentation was on verified solvers because they play an important role in our actual applications in the next chapter, and their integration was much more complicated than non-rigorous ones. The reason is that there is currently no standardized data exchange format such as IEEE 754-2008 for non-rigorous solvers. Usually, a certain interval library is employed by each verified solver. These libraries are often not interoperable, which makes interfacing the solvers harder. Additionally, verified IVP solvers tend to need more information than their FP counterparts. We discussed the interfacing of the verified IVP solvers VALENCIA-IVP and VNODE-LP and concluded the section with a brief overview of interfaced FP solvers.

Extension with further solvers

In the future, the framework can be extended by interfacing further solvers. Possible interesting candidates are, on the one hand, verified solvers for optimization problems such as GLOBSOL [Kea03] and, on the other hand, non-rigorous optimizers. Here, specialized solvers for linear optimization problems are important additions, because they are necessary not only for computing optimal preconditioners for the interval Newton procedure (cf. Sect. 4.3.2) but also for solving linear relaxed problems (e.g., obtained using affine reformulations [NMH10]).

Extension with further algorithms

Aside from third party solvers, new algorithms can be implemented inside UNIVERMEC directly. One algorithm class not represented currently is verified path planning for which an approach was presented by Jaulin [Jau01]. In a master thesis [Sch11], it was investigated how this algorithm can be implemented inside UNIVERMEC. A more sophisticated and possibly faster approach for verified path planning based on rapidly exploring random trees [LaV06, pp. 228-237] was presented in [Gri08]. An interesting future goal would be to implement this approach to investigate whether the verified feasibility tests used in [Gri08] can be improved by employing more sophisticated arithmetics . A further interesting point here would be to study how the algorithm behaves inside scenes described by complex models provided by the modeling layer of UNIVERMEC.

8 APPLICATIONS

Application examples

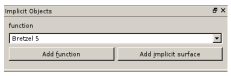
In the previous chapter, we presented several algorithms that make use of various features of UniVerMeC. In particular, we use its uniform handling of different verified techniques. In this chapter, we describe applications in which these algorithms and UniVerMeC were successfully used. Additionally, we give numerical results for them. We consider two different areas in order to demonstrate that our framework is universally applicable: distance computation between geometric models and parameter identification for SOFC models.

In the scope of distance computation, we show that the ϵ-distance algorithm introduced in Sect. 7.1.1 derives rigorous bounds on the distance between geometric objects given by various modeling types. These modeling types and forms appear in the scope of an automatic assistance system for THR surgery [Cuy11], where distance computation plays an important role. Our results can be seen as a stepping stone towards the full verification of the medical process. As a second example from the same area, we consider the comparison of different range bounding techniques using the ϵ-distance algorithm. This example shows that UniVerMeC can act as a fair comparison platform, which was one of our design goals (cf. Sect. 2.1).

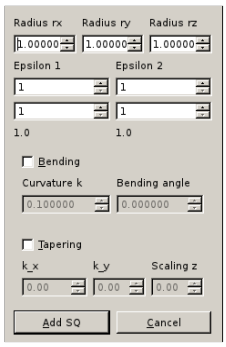
In the second application area (the parameter identification of SOFC models), our focus lies not only on the interval optimization algorithm with GPU acceleration described in Sect. 7.2 but also on the capability of UniVerMeC to apply internal and external interfaced solvers to the same problem description. Additionally, this last example demonstrates how the results of non-verified algorithms can be validated using verified solvers if a process cannot be carried out completely in a rigorous way. Here, we have to cope with both IVPs (cf. Sect. 5.2) and optimization problems (cf. Sect. 5.3) because the objective function of the parameter identification problem depends on the SOFC models, which are described by the means of IVPs.

Chapter structure

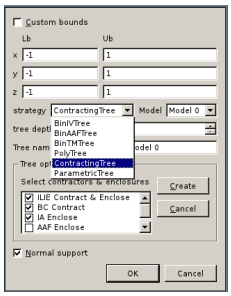
This chapter is structured as follows: We start in Sect. 8.1 with a short introduction into the TreeVis program, which is a GUI for the geometric part of UniVerMeC. After that, we present the test cases for the comparison of the different arithmetics along with the obtained results in Sect. 8.1.1 and consider the distance computation in scope of the THR assistance system in Sect. 8.1.2. In Sect. 8.2, we discuss the usage of UniVerMeC in the context of SOFCs. We present results of parameter identification and validate the obtained sets using a rigorous IVP solver. Conclusions are in Sect. 8.3.



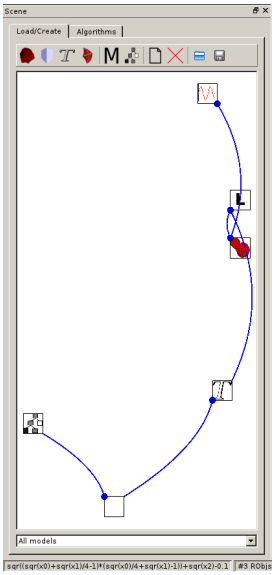
(a) An implicit surface can be specified by choosing a predefined one or through a formula.



(b) SQs can be configured using a dialog frame.



(c) Hierarchical decompositions are configured and attached to a model with the help of a dialog window.



(d) All surfaces, transformations, etc. are stored and managed in a resource graph.

Figure 38: TreeVis allows users to manage a geometric scene in an interactive manner.

8.1 TREEVIS

Interactive scene configuration

TreeVis is an interactive GUI supplied with UniVerMeC. It provides an easy access to the geometric modeling layer (cf. Sect. 5.1), the interval tree decompositions (cf. Sect. 6.1) and the ϵ-distance algorithm (cf. Sect. 7.1.1). With TreeVis, users can create a geometric scene, which consists of one or several geometric objects. These objects can be added to or removed from the scene interactively. Implicit objects are defined either by entering the formula for the implicit function directly or by choosing a predefined surface (cf. Fig. 38a). Polyhedra are loaded from ASCII files describing them (e.g., in the object file format [Off]). SQs are configured through a specialized dialog (cf. Fig. 38b), which allows for altering the model parameters. In this way, both implicit and parametric descriptions of an SQ are generated. Additionally, TreeVis allows users to apply transformations such as rotation[1] or bending, to a geometric object. These transformations are realized inside UniVerMeC by arranging them and the associated geometric objects in a tree (similar to CSG trees, cf. Sect. 5.1). After users finished configuring the transformation tree, they instantiate it and create what we call a TreeVis *model*. Finally, users can add inter-

1 In its current implementation, TreeVis always uses FP matrices to represent rotations. This is a GUI limitation. In the underlying framework UniVerMeC both FP and interval matrices can be applied.

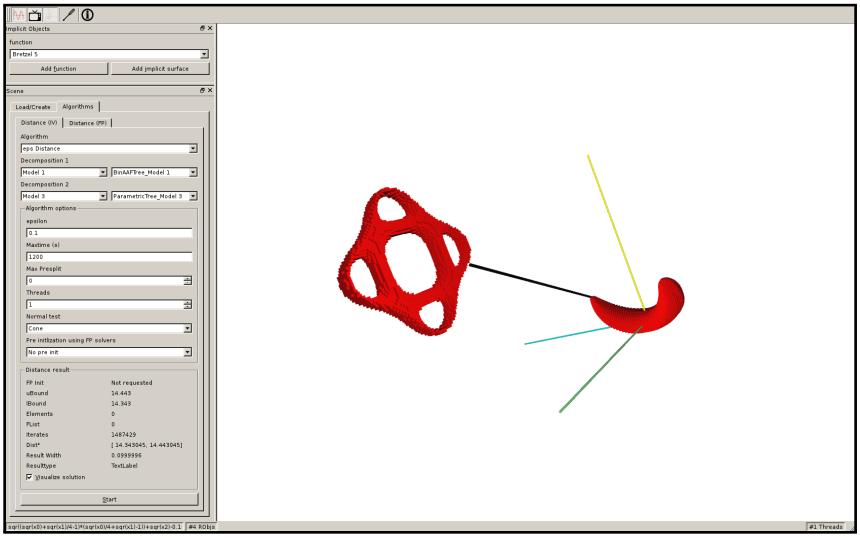


Figure 39: The TREEVIS user interface allows for interactive visualization and verified distance computation between geometric objects.

val tree decompositions to a TREEVIS model. Interval tree options can be set with the help of a dialog window (cf. Fig. 38c). The trees make it possible to visualize the model (cf. Fig. 39). Additionally, users can call the ϵ-distance algorithm (cf. Sect. 7.1.1) from TREEVIS to derive a rigorous bound on the distance between two models.

Resource graph

Internally, TREEVIS employs a *resource graph* to manage the scene. The graph stores the instances of the underlying UNIVERMEC classes and the information about them. Additionally, it tracks dependencies between these classes. The purpose of the resource graph is to allow for a leak-free memory management and for loading or storing scene configurations. In this context, we mean by configuration the information necessary to instantiate all classes in the graph again. The resource graph is implemented using the BOOST GRAPH LIBRARY [SLL02]. Configurations are stored using the GRAPHML format [Gra], which is recognized by this library.

For user convenience, we visualize the graph inside TREEVIS to provide insight into the underlying structure of the UNIVERMEC classes, which act as building blocks for the final geometric models. As an illustration, consider the graph in Fig. 38d. To create it, users should press the "Add implicit surface" button first. The new geometric model depends on three classes: `IFunction` as a representation of the underlying implicit function itself; `ImplSurface`, which depends on `IFunction` and connects it with a geometric object; `ICSGLeaf`, which depends on `ImplSurface` and allows us to fit the geometric object into a CSG tree[2]. Having defined the implicit surface, we can add a transformation node moving it away from the coordinate system's

2 Note that the actual CSG operations are carried out using UNIVERMEC and, thus, are limited to the ones discussed in Sect. 5.1.

origin. Finally, the terminal model instance can be created. A tree decomposing the model can be associated with this instance.

8.1.1 *Comparisons Between Range Arithmetics*

Considered object types

The first practical application of UniVerMeC is the comparison of different range arithmetics in the scope of verified distance computation. For this purpose, we use the ϵ-distance algorithm to obtain rigorous enclosures of the distances between various implicit surfaces. Therefore, this comparison also serves as a practical test for our distance computation approach. We focus on models described by implicit functions because they are represented analytically, which allows us to compare the range arithmetics from the point of view of their impact on the algorithm's performance. Polyhedral objects would not be well suited for this task because their description by simple halfplanes would diminish the impact of the used range arithmetic. Parametric descriptions would be also suited for such a comparison. We did not employ them because the only class of surfaces described parametrically inside UniVerMeC were SQs.

Quality of range bounds vs. computation times

Distinctive features of the ϵ-distance algorithm are that it computes distances up to a user-specified accuracy and performs an adaptive subdivision on the hierarchical structures decomposing the geometric models. In our comparison, we measure the time that the algorithm requires to obtain a rigorous distance bound up to the required tolerance with a specific arithmetic. This criterion is more realistic than, for example, studying only the quality of the range bounds. To understand this point, consider a test function for which AA always returns tighter bounds than IA. However, we cannot guarantee that using AA leads to a better algorithm performance because, in general, operations on affine forms are more expensive than those on intervals. That is, it might be faster to subdivide the trees deeper and perform cheap interval operations instead of using a smaller subdivision depth and carrying out fewer expensive AA operations. In our study, we avoid this pitfall by considering only the time that is required to obtain the final result. We think that, in practice, users are also likely to be interested only in the end result. Aside from the runtime, the required memory usage[3] is also of interest. In general, the correlation between techniques computing tight range bounds and the overall required memory is more obvious: cheaper techniques lead to a more extensive subdivision and, in this way, generate more nodes, which in turn increases the memory use.

3 In [AR10], several further comparison criteria for set-based IVPs are given. Based on them, the authors define a recommender system (cf. Sect. 1.3, [AR12]) for verified IVP solvers. The criterion used in our comparison can be seen as a variant of the C_5 criterion from this paper ("user CPU time wrt. resulting interval width"). Since the criteria from [AR10] are designed specifically for IVPs, not all of them are applicable in our context.

Table 19: Implicit surfaces used for comparing range-bounding methods.

OBJECT	FORMULA
Bretzel 2	$(x^2(1-x^2)-y^2)^2+0.5y^2-0.025(1+(x^2+y^2+z^2))$
Citrus	$x^2+z^2-4y^3(1-0.5y)^3$
Cylinder	x^2+y^2-1
Dedocube	$((x^2+y^2-0.64)^2+(z^2-1)^2)((y^2+z^2-0.64)^2+(x^2-1)^2)((x^2+z^2-0.64)^2+(y^2-1)^2)0.02$
Dodeca	$2-\cos(x+\alpha y)+\cos(x-\alpha y)+\cos(y+\alpha z)+\cos(y-\alpha z)+\cos(z+\alpha x)+\cos(z-\alpha x)$ with $\alpha=1.61803$
Ellipsoid	$0.5x^2+0.25y^2+z^2-1$
Heart	$(x^2+2.25y^2+z^2)^3-x^2z^3-0.1125y^2z^3$
Klein's bottle	$(x^2+y^2+z^2+2y-1)((x^2+y^2+z^2-2y-1)^2-8z^2)+16xz(x^2+y^2+z^2-2y-1)$
Plop	$x^2+(z+y^2)^3$
Pseudocube	$x^{500}+y^{500}+z^{500}-1$
Sphere	$x^2+y^2+z^2-r^2$
Sphere hole	$(y-x^2-y^2+1)^4+(x^2+y^2+z^2)^4-1$
Stretched sphere	$x^2+y^2+z^4-1$
Tetrahedral	$(x^2+y^2+z^2)^2+8xyz-10(x^2+y^2+z^2)+25$
Trigo	$(1-x^2+y^2)^2+\sin(z)^3-0.125$

Adequacy of ϵ-distance

Besides its methodological adequacy, the implementation of ϵ-distance inside UniVerMeC ensures that the comparison is fair. The reason is that the implementation of ϵ-distance itself neither needs nor has any knowledge of the arithmetic used to bound the range of the expression describing the implicit function. That is, the implementation of the algorithm is the same regardless of the employed arithmetic. Furthermore, the homogeneous data type independent FRO provided by the `IFunction` interface (cf. Sect. 4.4.2) and the usage of a function parser to create the implicit functions ensure, in combination with the arithmetic layer (cf. Sect. 3.6), that the overhead for using different arithmetics inside the framework is always the same. Finally, the affine transformations (e.g., rotation) are carried out uniformly inside the geometric modeling layer (cf. Sect. 5.1). The only component we exchange is the interval tree which uses specific methods for range bounding to determine the node colors.

Test cases

To compare the different range-bounding methods, we use 16 different test cases, which are based on 15 well-known (e.g., [Kno+09]) implicit surfaces (cf. Tab. 19). A test case consists of the two implicit objects for which we want to derive a distance bound and, possibly, affine transformations describing their position in space. Affine transformations are applied according to equation (36) where the rotation matrix and translation vector are converted to the used arithmetic beforehand. Test cases for which the expressions describing the implicit surfaces do not suffer from the dependency problem are called *simple* (*non-simple* otherwise). Further, we make a distinction between

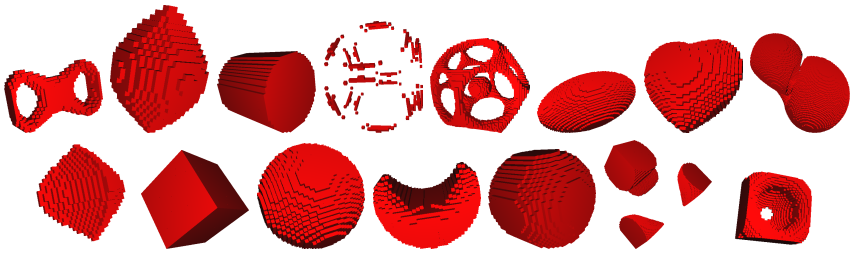


Figure 40: Plot of all implicit surfaces produced by an interval tree. The subdivision process was stopped if the volume of a node was less than 10^{-4}. In the first line Bretzel 2 to Kleins bottle and in the second line Plop to Trigo, ordered according to Tab. 19.

Table 20: Configurations for the 16 test cases. The surfaces are given in Tab. 19. Rotation is described by the axis and the angle in degrees. The starting box is used for the root nodes of the tree structures for the hierarchical decomposition. In the last column, the intersection of all derived enclosures is given.

SCENE	OBJECT	TRANSLATION	ROTATION	START BOX	MINIMUM ENCLOSURE
1	Trigo Sphere ($r = 0.5$)	$(0,0,1)$		$([-4,4],[-4,4],[-4,4])$ $([-1,1],[-1,1],[-1,1])$	$[0.338, 0.340]$
2	Sphere Pseudocube	$(-3,1,0)$		$([-1,1],[-1,1],[-1,1])$ $([-4,-2],[0,2],[-1,1])$	$[1.000, 1.001]$
3	Ellipsoid Plop	$(0,0,-2)$		$([-2,2],[-2,2],[-2,2])$ $([-1,1],[-1,1],[-3,-1])$	$[0.999, 1.001]$
4	Cylinder Stret. Sphere	$(3,1,0)$		$([-1,1],[-1,1],[-1,1])$ $([2,4],[0,2],[-1,1])$	$[1.162, 1.163]$
5	Sphere Pseudocube	$(3,0,0)$	$\vec{r} = e_x, \alpha = 45$	$([-1,1],[-1,1],[-1,1])$ $([2,4],[-1,1],[-1,1])$	$[0.999, 1.001]$
6	Cylinder Stret. Sphere	$(3,1,0)$	$\vec{r} = e_x, \alpha = 45$	$([-1,1],[-1,1],[-1,1])$ $([2,4],[0,2],[-1,1])$	$[1.111, 1.112]$
7	Trigo Sphere	$(1.6,0,0)$	$\vec{r} = e_y, \alpha = 90$	$([-4,4],[-4,4],[-4,4])$ $([0.6,2.6],[-1,1],[-1,1])$	$[0.400, 0.401]$
8	Ellipsoid Plop	$(0,0.7,-1.5)$	$\vec{r} = e_x, \alpha = -70$	$([-2,2],[-2,2],[-2,2])$ $([-1,1]^3 + \text{Trans.})$	$[0.000, 0.000]$
9	Klein's bottle Dodeca	$(4,4,4)$		$([1,8],[1,8],[1,8])$ $([-4.5,4.5]^3)$	$[0.316, 0.318]$
10	Heart Bretzel 2	$(0.5,0,2)$		$([-0.5,1.5],[-1,1],[1,3])$ $([-1,1],[-1,1],[-1,1])$	$[0.811, 0.813]$
11	Sphere hole Citrus	$(0,3,0)$ $(0,3,0)$		$([-2,2],[1,5],[-2,2])$ $([-1,1],[2,4],[-1,1])$	$[0.0000, 0.012]$
12	Tetrahedral Dedocube			$([-3,3],[-3,3],[-3,3])$ $([-1,1],[-1,1],[-1,1])$	$[0.281, 0.283]$
13	Klein's bottle Dodeca	$(4,4,4)$	$\vec{r} = e_z, \alpha = 45$	$([1,8],[1,8],[1,8])$ $([-4.5,4.5]^3)$	$[0.131, 0.134]$
14	Bretzel 2 Heart	$(0,1.5,0)$	$\vec{r} = e_x, \alpha = -90$	$([-1,1],[-1,1],[-1,1])$ $([-2,2],[-0.5,3.5],[-2,2])$	$[0.086, 0.088]$
15	Tetrhedral Dedocube		$\vec{r} = e_y, \alpha = 45$	$([-3,3],[-3,3],[-3,3])$ $([-2,2],[-2,2],[-2,2])$	$[0.217, 0.220]$
16	Sphere hole Citrus	$(0,-3,0)$	$\vec{r} = e_x, \alpha = 270$	$([-1,1],[-1,1],[-1,1])$ $([-2,2],[-5,1],[-2,2])$	$[0.156, 0.159]$

```
#!/usr/bin/python

import univermecext
import sys

scene = univermecext.SceneGraph()
stream = univermecext.open_stream(sys.argv[1])
scene.load_scene(stream)
for h in scene.handles():
    t = scene.property(h).type()
    if t == univermecext.vertex_prop_t.MODEL:
        print "Found model root:  " + str(scene.model_name(h))
```

Listing 7: Python script using an extension module of UNIVERMEC to read a resource graph file.

rotated and *non-rotated* cases. In this way, we obtain four groups of test cases (cf. Tab. 20): simple (1-4), simple rotated (5-8), non-simple (9-12), non-simple rotated (13-16). We have already used the same groups in [DK12] to compare the different arithmetics. However, our new results are not directly comparable to the ones in [DK12] because we use newer versions of the libraries for range enclosures. Moreover, we exchanged the non-verified AA library LIBAFFA against YALAA. Additionally, the results were computed with a newer version of the ϵ-distance algorithm, which employs the improved case selector (44), whereas the original version of the case-selector from [BDL04] was used in [DK12]. This test rerun revealed several bugs in the original implementation from [DK12] that led to erroneous results for the minimum enclosures. The corrected[4] versions are given in Tab. 20.

Test method

To compare the techniques, we tried to compute enclosures for the distance for each scenario with the accuracies $\epsilon \in \{0.1, 0.01, 0.001\}$ and measured the required CPU time with and without the normal cone test (51). The maximum allowed CPU time was set to 1200 seconds. Every scenario was tested with the standard trees using IA, AA and TMs for range enclosure. Additionally, we used the contracting tree in two configurations: In the first, we employed the mean-value form for the range enclosure and no contraction so that the tree behaved like a standard one. In the second configuration, AA was used for range bounding and box consistency (cf. Sect. 4.3.3) for contraction. The underlying libraries were C-XSC [HK04] (v. 2.5.2), YALAA [Kie12b] (v. 0.91), RIOT [Ebl06] and FADBAD++ [SB] (v. 2.1). Our test system consisted of an Intel Xeon CPU E5-2680 with 8 cores and 64 GB RAM. All tests were carried out under Linux. UNIVERMEC was compiled using the GCC compiler (v. 4.7.2) with `-O3` optimizations. To minimize the impact of measurement errors, each test was performed three times, and the average of the required time was taken.

4 The table also corrects minor typing errors for the given geometric configurations.

Table 21: Results for the test cases from Tab. 20 for different accuracies and tree structures without the use normal vectors. For each test case, the CPU time (t) in seconds, the number of iterations (i) in ten thousands and the number of tree nodes (n) in thousands are given. The best result for each category is marked in green and if a test case could not be solved in the preset time of 1200s it is marked with a dash (–). Additionally, the averages for each group ($\varnothing_1$) and the averages over all test cases ($\varnothing_2$) are shown.

		BinIVTree (IA)			BinAATree (AA)			BinTMTree (TMs)			CIVTree (mean-value form (20))			CIVTree (AA; box consistency)		
	ε	0.1	0.01	0.001	0.1	0.01	0.001	0.1	0.01	0.001	0.1	0.01	0.001	0.1	0.01	0.001
1	t	0.31	0.79	101.13	0.71	1.46	108.02	2.86	4.53	135.22	1.20	4.63	107.11	0.84	1.55	12.73
	i	32.72	82.59	5602.15	43.96	97.62	5638.82	43.84	95.22	5615.23	96.84	342.40	5792.83	7.01	22.76	509.46
	n	18.68	30.86	311.13	23.51	35.90	316.98	23.13	35.39	315.70	32.74	78.86	329.92	8.37	18.01	79.35
2	t	0.59	11.25	1045.95	0.01	0.05	0.41	5.31	31.47	–	0.78	12.29	1067.38	0.05	0.27	1.89
	i	57.14	761.08	48533.36	0.04	0.30	2.25	57.37	762.18	–	60.56	775.74	49099.88	0.04	0.27	1.83
	n	13.10	51.75	398.66	0.87	5.90	44.09	13.15	51.82	–	13.83	53.04	402.61	0.82	4.98	33.68
3	t	90.32	–	–	352.24	–	–	338.37	–	–	518.74	–	–	145.16	–	–
	i	4971.02	–	–	16816.67	–	–	15274.49	–	–	24747.24	–	–	899.49	–	–
	n	189.63	–	–	327.12	–	–	305.57	–	–	414.64	–	–	96.98	–	–
4	t	0.86	57.53	–	1.20	60.35	–	2.57	74.17	–	1.20	61.56	–	1.07	40.56	–
	i	89.10	3413.81	–	90.87	3421.80	–	89.35	3414.40	–	101.15	3538.45	–	36.86	1736.02	–
	n	22.85	190.51	–	23.13	191.02	–	22.93	190.60	–	24.56	195.74	–	14.90	121.93	–
$\varnothing_1$	t	23.02	23.19	573.54	88.54	20.62	54.21	87.28	36.72	135.22	130.48	26.16	587.24	36.78	14.12	7.31
	i	1287.49	1419.16	27067.76	4237.89	1173.24	2820.54	3866.26	1423.93	5615.23	6251.45	1552.19	27446.36	235.85	586.35	255.65
	n	61.06	91.04	354.89	93.66	77.60	180.54	91.19	92.60	315.70	121.44	109.21	366.26	30.27	48.31	56.51
5	t	1.41	25.58	–	0.02	0.13	0.96	9.42	58.17	–	1.69	26.51	–	0.04	0.26	1.76
	i	121.62	1581.85	–	0.05	0.33	2.39	113.74	1518.67	–	124.38	1563.53	–	0.05	0.36	2.56
	n	23.28	92.90	–	1.00	6.52	47.71	22.09	89.70	–	23.70	92.56	–	0.93	6.53	46.85
6	t	0.19	15.13	–	0.26	11.45	848.97	0.95	17.13	913.02	0.29	11.69	851.55	0.64	11.61	875.28
	i	19.81	1018.39	–	16.08	749.93	40141.16	15.59	744.83	40117.81	21.72	787.65	40453.46	18.40	567.50	30913.57
	n	8.89	72.08	–	7.93	59.60	466.32	7.81	59.22	465.82	9.34	62.73	470.08	9.05	55.95	390.87
7	t	0.29	1.61	154.59	1.16	3.41	166.02	6.85	13.24	215.01	2.41	5.02	166.60	0.59	1.46	22.59
	i	29.50	161.19	8334.47	61.78	209.21	8427.92	66.29	206.40	8384.54	165.92	360.81	8730.41	11.09	35.76	1047.16
	n	18.23	60.64	406.06	30.03	74.64	421.41	30.67	74.09	419.79	55.54	104.87	454.80	9.12	22.62	110.65
8	t	0.02	0.03	0.03	0.15	0.19	0.18	0.51	0.54	0.55	0.31	0.38	0.38	4.00	4.47	6.23
	i	2.29	3.08	3.11	7.61	9.88	10.10	6.13	7.17	7.24	31.03	36.51	37.01	2.93	3.48	3.83
	n	1.28	2.55	2.55	4.56	5.57	5.63	4.00	4.33	4.35	6.93	9.35	9.47	2.33	2.55	3.00
$\varnothing_1$	t	0.48	10.59	77.31	0.40	3.80	254.03	4.43	22.27	376.19	1.18	10.90	339.51	1.32	4.45	226.47
	i	43.30	691.13	4168.79	21.38	242.34	12145.39	50.44	619.27	16169.87	85.76	687.13	16406.96	8.12	151.77	7991.78
	n	12.92	57.04	204.31	10.88	36.58	235.27	16.14	56.84	296.65	23.88	67.37	311.45	5.36	21.91	137.84
9	t	77.77	–	–	3.70	4.86	116.74	8.80	16.31	197.36	31.98	39.08	153.39	4.95	7.58	86.81
	i	4266.56	–	–	258.58	299.71	5115.12	96.01	144.92	4933.42	1961.16	2362.96	7710.40	242.11	300.93	3254.29
	n	237.12	–	–	19.89	34.98	207.89	18.41	35.47	205.61	65.06	109.71	263.54	19.35	38.92	175.47
10	t	0.79	4.45	95.23	2.43	9.25	107.32	8.29	28.98	324.66	12.10	21.45	74.08	2.72	7.75	38.54
	i	78.59	344.16	4378.33	167.13	432.04	2624.42	151.86	387.19	2530.37	759.20	1239.60	3022.19	50.77	120.67	283.94
	n	24.73	158.56	2901.05	25.84	143.87	1761.48	28.83	138.96	1752.79	106.60	189.86	1293.81	20.14	66.02	349.13
11	t	0.11	0.49	2.26	0.25	0.43	0.53	2.24	3.48	4.35	0.99	1.70	1.98	3.78	3.92	4.72
	i	9.89	41.64	169.32	8.21	17.39	19.07	6.65	11.54	13.43	65.96	121.50	136.21	5.90	8.82	10.75
	n	5.92	23.37	90.38	6.48	9.85	12.74	5.92	9.37	12.69	21.82	31.86	39.91	7.90	8.52	11.27
12	t	176.20	183.47	–	6.77	11.78	148.46	13.28	47.23	327.64	11.41	29.36	694.38	7.36	28.05	239.53
	i	8878.33	10004.04	–	279.57	460.38	5536.01	88.82	325.64	4227.08	713.94	1615.26	27552.03	147.33	407.24	4086.06
	n	1949.55	1561.80	–	141.67	204.84	1329.13	44.18	179.86	1158.10	85.04	344.28	3848.62	44.02	193.04	1335.38
$\varnothing_1$	t	63.72	62.80	48.74	3.29	6.58	93.26	8.15	24.00	213.50	14.12	22.90	230.96	4.70	11.82	92.40
	i	3308.34	3463.28	2273.82	178.37	302.38	3323.65	85.83	217.32	2926.08	875.06	1334.83	9605.21	111.53	209.42	1908.76
	n	554.33	581.24	1495.72	48.47	98.39	827.81	24.34	90.91	782.30	69.63	168.93	1361.47	22.85	76.62	467.81
13	t	223.40	–	–	3.82	4.60	11.84	8.99	14.06	49.95	46.69	47.72	70.19	4.92	6.69	17.92
	i	10711.36	–	–	262.77	277.54	616.49	96.41	109.72	496.66	2658.49	2711.29	3805.67	250.67	266.94	604.16
	n	386.34	–	–	17.55	30.42	90.63	17.58	27.88	92.80	70.68	83.71	204.42	17.14	31.04	90.27
14	t	0.23	0.32	1.26	3.41	4.52	19.03	7.98	12.42	24.81	7.60	10.53	20.08	32.01	40.06	70.20
	i	23.96	33.62	110.14	225.65	291.17	865.60	185.62	241.62	396.80	496.97	692.31	1232.31	86.53	109.15	362.95
	n	10.58	17.45	97.00	27.79	41.72	231.54	16.87	34.17	99.44	45.93	65.99	183.68	19.09	27.22	190.56
15	t	90.40	158.27	–	33.75	51.04	434.67	76.52	152.91	805.67	–	–	–	53.24	95.16	324.23
	i	5286.66	8950.85	–	1407.15	2126.73	18680.19	922.07	1571.90	17068.31	–	–	–	1983.32	2988.31	9895.31
	n	273.01	715.85	–	205.51	314.42	1134.47	103.16	253.62	1050.39	–	–	–	141.85	366.44	912.75
16	t	0.14	0.31	1.38	0.39	2.57	100.52	2.38	28.61	785.37	1.28	2.77	–	0.36	0.61	3.00
	i	15.72	32.43	117.97	23.13	91.96	3187.48	13.85	74.98	3076.38	113.32	224.52	–	4.08	7.72	34.48
	n	5.29	14.73	72.99	7.74	60.57	1393.89	6.88	56.74	1388.70	19.24	54.18	–	3.16	6.48	31.96
$\varnothing_1$	t	78.54	52.96	1.32	10.34	15.68	141.52	23.97	52.00	416.45	18.52	20.34	45.13	22.63	35.63	103.84
	i	4009.42	3005.63	114.06	479.67	696.85	5837.44	304.49	499.56	5259.54	1089.59	1209.38	2518.99	581.15	843.03	2724.23
	n	168.80	249.34	84.99	64.65	111.78	712.63	36.12	93.10	657.83	45.28	67.96	194.05	45.31	107.80	306.38
$\varnothing_2$	t	41.44	35.32	175.23	25.64	11.07	147.40	30.96	33.55	315.30	42.58	19.62	291.56	16.36	16.67	121.82
	i	2162.14	2032.98	8406.11	1229.33	565.73	6490.50	1076.76	641.09	7238.94	2141.19	1169.47	13415.67	234.16	438.39	3643.60
	n	199.28	230.23	534.98	54.41	81.32	533.14	41.95	82.75	580.51	66.38	105.48	681.89	25.95	64.68	268.65

Table 22: Results for the test cases from Tab. 20 for different accuracies and tree structures under the employment of the normal cone test (51). For each test case, the CPU time (t) in seconds, the number of iterations (i) in ten thousands and the number of tree nodes (n) in thousands are given. The best result for each category is marked in green and test cases which could not be solved in the preset time of 1200s with a dash (–). Additionally, the averages for each group ($\varnothing_1$) and the averages over all test cases ($\varnothing_2$) are shown.

		BinIVTree (IA)			BinAATree (AA)			BinTMTree (TMs)			CIVTree (mean-value form (20))			CIVTree (AA; box consistency)		
	ϵ	0.1	0.01	0.001	0.1	0.01	0.001	0.1	0.01	0.001	0.1	0.01	0.001	0.1	0.01	0.001
1	t	0.15	0.16	0.19	0.49	0.55	0.64	2.59	2.81	3.20	0.20	0.23	0.27	0.43	0.49	0.55
	i	3.81	4.26	4.97	5.45	6.03	6.67	5.16	5.70	6.33	3.81	4.26	4.97	1.61	1.99	2.33
	n	6.28	6.97	8.66	9.60	10.85	12.62	9.58	10.76	12.55	6.28	6.97	8.66	5.03	5.84	6.74
2	t	0.02	0.03	0.04	0.01	0.10	0.54	9.22	30.18	37.67	0.02	0.04	0.06	0.04	0.17	1.06
	i	0.17	0.28	0.38	0.03	0.19	1.26	3.54	13.61	20.61	0.17	0.28	0.38	0.03	0.17	1.06
	n	0.77	1.45	1.91	0.67	3.73	25.15	11.45	35.70	46.03	0.77	1.45	1.91	0.59	3.05	18.96
3	t	0.20	0.67	3.45	1.27	4.28	22.96	5.64	19.27	99.44	0.30	1.01	5.40	0.84	9.53	141.81
	i	4.39	15.99	80.74	10.28	33.22	164.07	8.88	30.18	146.69	4.39	15.99	80.74	4.19	38.74	411.71
	n	12.20	41.72	218.62	23.89	79.34	413.36	22.09	72.83	369.23	12.20	41.72	218.62	13.64	142.71	1954.10
4	t	0.01	0.03	0.08	0.06	0.12	0.37	0.10	0.23	0.67	0.02	0.05	0.13	0.05	0.11	0.22
	i	0.19	0.39	0.85	0.33	0.72	1.62	0.19	0.39	0.85	0.19	0.39	0.85	0.14	0.33	0.59
	n	0.82	1.94	7.25	1.26	3.22	13.63	0.82	1.94	7.25	0.82	1.94	7.25	0.68	1.79	3.61
$\varnothing_1$	t	0.09	0.22	0.94	0.46	1.26	6.13	4.39	13.12	35.24	0.14	0.33	1.47	0.34	2.58	35.91
	i	2.14	5.23	21.74	4.02	10.04	43.41	4.44	12.47	43.62	2.14	5.23	21.74	1.49	10.31	103.92
	n	5.02	13.02	59.11	8.85	24.28	116.19	10.98	30.31	108.77	5.02	13.02	59.11	4.99	38.35	495.85
5	t	0.02	0.04	0.07	0.05	0.32	2.36	15.85	51.74	64.82	0.03	0.07	0.11	0.04	0.27	1.80
	i	0.30	0.54	0.81	0.05	0.33	2.39	6.67	26.56	40.86	0.30	0.54	0.81	0.05	0.36	2.55
	n	0.99	2.12	3.39	1.00	6.52	47.71	21.14	67.97	89.48	0.99	2.12	3.39	0.91	6.49	46.81
6	t	0.02	0.03	0.05	0.03	0.07	0.13	0.14	0.24	0.44	0.03	0.04	0.09	0.06	0.12	0.24
	i	0.22	0.42	0.65	0.18	0.31	0.54	0.16	0.29	0.52	0.21	0.41	0.64	0.23	0.46	0.75
	n	0.84	1.64	2.71	0.80	2.02	4.22	0.74	1.94	4.14	0.85	1.95	3.77	0.89	1.79	3.86
7	t	0.24	0.31	0.37	1.55	1.81	1.95	10.96	12.04	12.67	0.36	0.46	0.55	0.42	0.51	0.59
	i	5.95	7.59	8.92	10.71	12.83	14.04	10.41	12.37	13.54	5.95	7.60	8.92	2.53	3.27	3.81
	n	11.12	14.23	16.63	20.37	24.22	26.76	20.55	24.14	26.62	11.12	14.23	16.63	5.78	7.33	8.43
8	t	0.01	0.03	0.03	0.22	0.26	0.27	0.85	0.93	0.94	0.02	0.05	0.05	0.04	0.06	0.08
	i	0.41	0.77	0.77	1.61	2.03	2.05	1.40	1.57	1.57	0.41	0.76	0.76	0.21	0.35	0.43
	n	1.01	2.13	2.13	3.72	4.56	4.58	3.29	3.58	3.59	1.01	2.09	2.09	0.56	0.98	1.24
$\varnothing_1$	t	0.07	0.10	0.13	0.46	0.61	1.18	6.95	16.24	19.71	0.11	0.16	0.20	0.14	0.24	0.68
	i	1.72	2.33	2.79	3.14	3.87	4.75	4.66	10.20	14.12	1.72	2.33	2.78	0.75	1.11	1.89
	n	3.49	5.03	6.22	6.47	9.33	20.82	11.43	24.41	30.95	3.49	5.10	6.47	2.04	4.15	15.09
9	t	19.84	32.91	49.34	5.03	5.90	7.54	25.06	32.41	48.00	6.79	7.60	8.55	4.99	6.25	8.51
	i	633.81	1000.72	1361.98	144.66	149.01	154.50	64.13	68.48	73.75	252.68	272.10	287.97	149.79	161.51	179.30
	n	109.88	222.72	437.17	16.37	20.98	30.69	15.57	20.15	29.79	36.83	45.61	57.86	16.14	25.59	39.73
10	t	0.61	1.05	1.32	3.03	3.89	4.31	15.51	19.61	22.20	0.82	1.25	1.56	0.85	2.15	2.42
	i	17.62	29.08	38.22	56.68	67.49	70.86	54.71	63.16	66.46	17.51	26.34	33.44	5.66	14.21	17.75
	n	15.60	26.58	32.76	17.37	24.88	29.25	17.23	22.90	27.16	15.34	22.79	29.71	7.44	17.52	19.37
11	t	0.14	0.47	1.92	0.62	0.97	1.18	6.37	9.01	11.11	0.19	0.51	0.80	0.27	0.29	0.43
	i	3.87	12.69	48.81	5.13	9.38	10.81	3.82	6.50	8.22	3.76	9.57	14.75	1.38	1.68	2.21
	n	5.11	16.44	65.85	6.41	9.51	11.95	5.76	8.66	11.65	4.87	11.86	18.76	3.32	3.54	5.28
12	t	40.42	52.71	71.31	9.91	12.00	30.54	44.61	74.20	200.62	4.25	12.47	15.94	5.16	17.27	52.16
	i	1164.03	1524.21	1767.95	107.34	136.57	175.15	43.80	83.63	119.62	135.52	303.08	354.50	65.81	154.92	225.67
	n	654.45	572.97	1585.80	80.05	95.68	360.21	38.65	77.79	396.16	53.38	167.75	233.07	33.20	112.92	399.66
$\varnothing_1$	t	15.25	21.78	30.97	4.65	5.69	10.89	22.89	33.81	70.48	3.01	5.46	6.71	2.82	6.49	15.88
	i	454.83	641.68	804.24	78.45	90.62	102.83	41.61	55.44	67.01	102.37	152.77	172.66	55.66	83.08	106.23
	n	196.26	209.68	530.39	30.05	37.76	108.03	19.30	32.38	116.19	27.60	62.00	84.85	15.02	39.89	116.01
13	t	35.05	54.26	67.17	5.54	6.31	7.11	25.94	30.54	38.56	9.80	10.22	11.58	5.51	7.00	9.16
	i	1101.17	1617.60	1912.78	146.10	149.04	151.37	65.78	67.97	70.49	376.57	393.58	420.28	179.90	187.47	202.92
	n	180.68	322.90	426.51	15.44	19.51	23.85	15.04	17.45	22.10	37.87	47.71	68.30	15.70	25.90	39.32
14	t	0.23	0.27	0.32	3.87	4.43	4.73	17.63	23.97	26.37	0.32	0.37	0.45	0.32	0.48	0.72
	i	8.05	9.19	10.20	69.47	78.41	80.95	60.11	71.53	73.77	8.05	9.17	10.17	2.89	3.64	4.38
	n	6.38	7.44	9.00	20.92	23.85	26.02	14.51	21.99	24.11	6.38	7.40	8.96	2.67	4.02	5.92
15	t	46.71	81.93	334.83	46.98	65.64	112.18	235.82	426.10	730.47	7.10	16.25	62.13	8.35	35.58	84.04
	i	1559.28	2515.35	7543.52	384.84	559.85	923.60	248.66	437.89	768.69	239.80	446.20	1346.22	197.65	429.02	887.54
	n	194.13	571.61	4919.94	185.16	269.84	495.74	88.73	208.58	427.55	57.46	174.13	559.07	34.94	208.88	441.56
16	t	0.10	0.14	0.18	0.70	0.98	1.21	4.51	6.51	8.06	0.15	0.20	0.26	0.31	0.38	0.50
	i	2.71	3.75	4.69	6.33	8.48	10.41	3.91	5.55	7.33	2.72	3.74	4.54	1.22	1.72	2.55
	n	3.65	5.04	6.30	7.09	9.65	11.63	5.21	7.06	8.89	3.64	5.00	6.11	2.28	3.08	4.57
$\varnothing_1$	t	20.52	34.15	100.62	14.27	19.34	31.31	70.97	121.78	200.86	4.35	6.76	18.60	3.62	10.86	23.61
	i	667.80	1036.47	2367.80	151.68	198.94	291.58	94.61	145.74	230.07	156.78	213.17	445.30	95.42	155.46	274.35
	n	96.21	226.75	1340.44	57.15	80.71	139.31	30.88	63.77	120.67	26.34	58.56	160.61	13.90	60.47	122.84
$\varnothing_2$	t	8.99	14.07	33.17	4.96	6.73	12.38	26.30	46.24	81.58	1.90	3.18	6.75	1.73	5.04	19.02
	i	281.62	421.43	799.14	59.32	75.87	110.64	36.33	55.96	88.71	65.75	93.37	160.62	38.33	62.49	121.60
	n	75.24	113.62	484.04	25.63	38.02	96.09	18.15	37.72	94.14	15.61	34.67	77.76	8.99	35.71	187.45

We did not perform the tests using the graphical TreeVis program because the visualization could distort the results. Instead, we opted for automatic processing of all tests with the help of the Python interface offered by UniVerMeC. The necessary extension module for the Python scripting language is generated with the help of SWIG[5] [Swi]. Using this module, we can read the resource graph file format of TreeVis. For example, the program in listing 7 reads the resource graph file passed as an argument and prints the names of all model roots defined there. In this way, TreeVis is used to graphically configure the scene and the actual tedious test runs are handled automatically by a script, which additionally minimizes input/output errors.

Test results

The obtained results without the normal cone test are given in Tab. 21. In the table, the required CPU time (t), total number of algorithm iterations (i) and the number of used tree nodes (n) for each accuracy, scenario and tree are listed. Additionally, the averages for each subgroup are given. Besides the CPU time, the total number of tree nodes might be of interest because the memory required to complete a scenario is directly proportional to this number. For the standard trees, the numbers confirm our result from [DK12] which indicates that AA is a good overall choice. For the simple scenarios, standard IA performs comparatively well. However, it fails for two scenarios in the highest accuracy, whereas AA is successful for them. The tight range enclosures obtained with TMs seem not to make up for their high computational cost. However, better enclosure widths achieved using TMs make the algorithm employ fewer tree nodes in several scenarios compared to the other techniques. Besides the standard trees, we also tested our contracting tree (column 6). It delivered the best performance overall in respect both to computational time and number of tree nodes.

In Tab. 22, the results for the test series using normal vectors are given. In general, the normal cone test improves the performance for all tree structures and allows us to carry out the scenarios with the highest accuracy without premature termination. Especially for the simple test cases not suffering from the dependency problem, standard IA delivered the best overall performance (cf. Fig. 41) if the additional information provided by normal vectors is used. If the more complex scenarios 9-16 are taken into account, the mean-value form performs best with respect to the CPU time. The use of the contracting tree keeps the number of tree nodes low in some scenarios but fails to do so in others. Its performance with respect to the CPU time is average[6]. It might be an interesting topic for future research to test the contracting tree with ILIEs because we obtained promising results for them in the context of the LIETree in [Kie12a]. Addition-

5 Small Interface and Wrapper Generator

6 Here, one reason might be an implementation detail of the `CIVTree` class that enforces additional function evaluations in order to obtain the enclosure for the normal vectors.

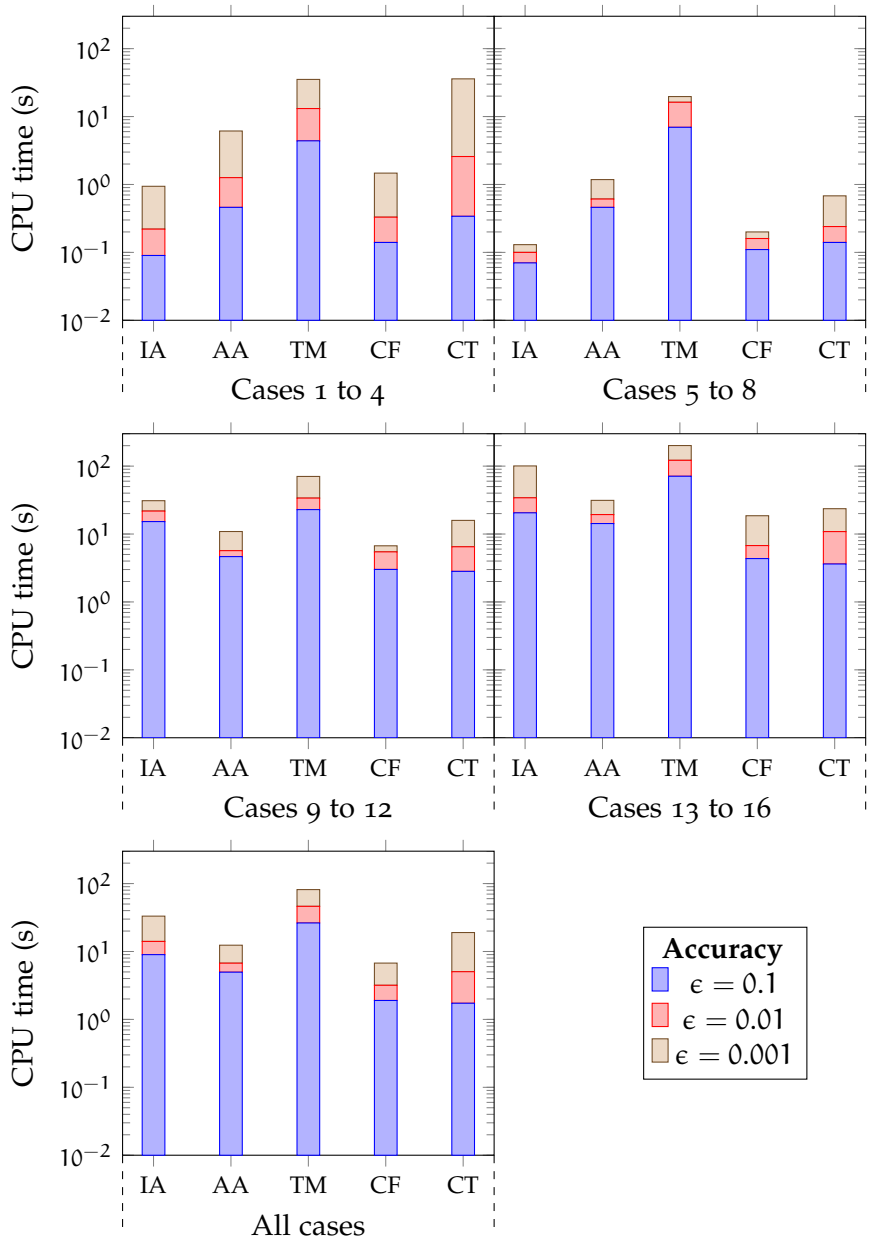


Figure 41: Average CPU time for the ϵ-distance algorithm with the normal cone test on a logarithmic scale for IA, AA, TM, the mean-value form (20) (MF) and the contracting tree (CT). The times for the higher accuracies need to be added to the lower ones to obtain the total time for the accuracy.

ally, the extension of the contracting tree with black inversion nodes (cf. Sect. 6.1.2, which is currently not implemented) might improve the overall performance of this tree structure. While still good AA is not the best choice in this test series. TMs suffer from their high computational cost[7].

8.1.2 *Verification of Distances for Total Hip Replacement*

Automatic system for THR

In the last subsection, we showed how the ϵ-distance algoritm can be used to compare different range-bounding methods and to derive rigorous enclosure of distances for the test cases. We focus on this because verified distance computation plays a major role in several application areas where obtaining rigorous bounds is of high importance. As an example for such an area, we consider a system developed by Cuypers for automated selection of a hip implant [Cuy11] in a THR surgery, which appeared in the course of the project PROREOP [Pro]. The system takes the CT image of a rooted femoral shaft as input and constructs an SQ model from it. Furthermore, it uses SQs to describe the potential implants. For the automatic selection of an appropriate implant, the THR assistance system simulates the insertion process of the implant into the femoral shaft, which requires several distance computations, in particular, between two SQs and an SQ and a polyhedron. All these geometric models can be non-convex.

Numerical verification and validation

The original system of Cuypers was validated in the study [Cuy11, pp. 124-141] against the state of the art technique of manual implant selection. However, questions regarding the *numerical* verification and validation were not considered in this original work. In [ALC13], the whole system was considered from the numerical point of view. The authors examined important subprocesses with regard to the numerical verification taxonomy introduced in [ALo9] (cf. Sect. 2.1.1). Additionally, a fully verified approach for the distance computation subprocess was presented in [Aue+11; Chu11] for the case of convex SQs modeling the bones. In this thesis, we describe our results from [KLD13], where we considered the computation of rigorous bounds between non-convex SQs and between a non-convex SQ and a non-convex polyhedron using UniVerMeC.

SQ models

As mentioned previously, an SQ [Bar81] can be described by the implicit function (19). The model depends on five parameters, where a_1, a_2, a_3 determine the scaling at the coordinate axes and ϵ_1, ϵ_2 de-

7 It would be interesting to test whether this high computational costs appear because of using Riot [Eblo6] and whether TMs perform better if the reference implementation COSY [BMo6] is employed

Table 23: Parameters of the SQ model and their ranges from [ALC13].

NAME	MEANING	CONDITION	DEFAULT
the standard SQ model			
τ	stability parameter	$\tau > 0$	10^{-5}
a_1, a_2, a_3	scaling of the SQ	$a_1, a_2, a_3 \geqslant \tau$	1.0
ϵ_1, ϵ_2	roundness of the SQ	$\tau \leqslant \epsilon_1, \epsilon_2 \leqslant 2-\tau,$	1.0
deformation parameters			
k_{x_1}, k_{x_2}	tapering factors	$-1 \leqslant k_{x_1}, k_{x_2} \leqslant 1$	0.0
k	curvature	$k \geqslant \tau$	0.1
α	bending angle	$-\pi \leqslant \alpha \leqslant \pi$	0.0

fine the roundness of the shape. Aside from its implicit representation (19), the SQ possesses a parametric description:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} a_1 \cos^{\epsilon_1} \eta \cos^{\epsilon_2} \omega \\ a_2 \cos^{\epsilon_1} \eta \sin^{\epsilon_2} \omega \\ a_3 \sin^{\epsilon_1} \eta \end{bmatrix}, \qquad \begin{matrix} -\pi/2 \leqslant \eta \leqslant \pi/2 \ , \\ -\pi \leqslant \omega \leqslant \pi \end{matrix} \tag{61}$$

Here, we assume that the signed power function is defined as $x^y = \operatorname{sign}(x)|x|^y$. To improve the quality of fitness, the standard model can be extended by global deformations. In the THR procedure, tapering and bending are used. With tapering, the SQ can be broadened or flattened along a given axis. This transformation introduces the two tapering factors k_{x_1}, k_{x_2}. Bending transforms the area along x_3-axis of the original SQ into a circular section. It is described by the curvature coefficient k and the bending angle α. Bending might lead to a non-convex model. Formulas for both transformations are to be found in [JLS00, pp. 43-48]. Note that the bending formulation given there is not defined on the plane $x_1 = 0$. Therefore, we use the alternative bending formulation derived in [KLD13]. To ensure model stability the range of several parameters is restricted to certain intervals given in Tab. 23 (cf. [ALC13]).

Test cases and configuration

To demonstrate that UNIVERMEC and the introduced ϵ-distance algorithm can be used to verify the important distance computation subprocess of the THR procedure, we used different test cases shown in Tab. 24. We limited these tests to non-convex scenarios because specialized procedures are available for the convex ones (cf. [Aue+11; Chu11], Sect. 7.1). The geometric configurations are visualized in Fig. 42. Similarly to our comparison in Sect. 8.1.1, we tried to compute bounds for each test case with the different accuracies $\epsilon \in \{0.1, 0.01, 0.001\}$. All tests were carried out on an Intel i7-860 2,8 GHz system with 8 GB main memory running under Linux. We used the

Table 24: Configurations of the four test cases. The SQ parameters are provided in the order $(a_1, a_2, a_2, \epsilon_1, \epsilon_2, k, \alpha, k_{x_1}, k_{x_2})$. The last column contains the transformation of the second SQ in the form $(t_{x_1}, t_{x_2}, t_{x_3}, r_{x_1}, r_{x_2}, r_{x_3}, \alpha_r)$ where $t_{x_1}, t_{x_2}, t_{x_3}$ is the translation, $r_{x_1}, r_{x_2}, r_{x_3}$ the rotation axis and α_r the rotation angle.

SCENE	SQ 1	SQ 2	TRANSFORMATION SQ 2
1	$(0.25, 0.75, 0.25, \frac{50}{37}, \frac{10}{3}, 0, 0, 0, 0)$	$(0.25, 0.75, 0.25, \frac{50}{37}, \frac{10}{3}, 0, 0, 0, 0)$	$(4, 4, 0, 1, 0, 0, 70)$
2	$(0.50, 0.25, 2, \frac{19}{18}, \frac{19}{25}, 0.9, 45, 0, 0)$	$(0.25, 0.25, 3, 1, 1, 0.5, 70, 0, 0)$	$(0, 2.5, -0.75, 1, 0, 0, 90)$
3	*Polyhedron*	$(1, 1, 4, \frac{18}{5}, 1, 0, 3, 30, 0, 0)$	$(-2, 4, -1.5, 0, 0.7, -0.3, 90)$
4	$(0.25, 1, 4, 1, 1, 0.3, 120, 1.5, 2.7)$	$(0.35, 0.75, 0.9, \frac{3}{2}, 1, 0, 0, 1.8, 0.4)$	$(4, 0, 0, 1, 0, 0, 0)$

Table 25: Test results for the scenarios from Tab. 24 with and without the normal cone test (51). The tree configuration column indicates which tree type (standard interval tree or parametric tree) was used. The qualifier FP init indicates that the algorithm was initialized using results from an FP solver. The total CPU time in seconds is given. A dash indicates that the scenario could not be finished in the predefined time. A cross shows that the configuration is not supported by our system.

	TREE CONFIGURATION	WITHOUT NORMALS			WITH NORMALS			RESULT
		0.1	0.01	0.001	0.1	0.01	0.001	
1	`SParamIVTree / SParamIVTree`	10.3	35.2	235.0	10.5	35.1	231.7	$[4.935, 4.937]$
	`BinIVTree / BinIVTree`	–	–	–	–	–	–	
2	`SParamIVTree / SParamIVTree`	27.8	295.5	–	24.3	153.9	–	$[0.368, 0.375]$
	`SParamIVTree / SParamIVTree` (FP init)	18.0	283.1	–	13.9	148.4	–	
3	`SParamIVTree / PolyIVTree`	266.3	376.9	-	×	×	×	$[0.410, 0.422]$
4	`SParamIVTree / SParamIVTree`	104.0	-	-	25.2	353.5	–	$[2.670, 2.671]$

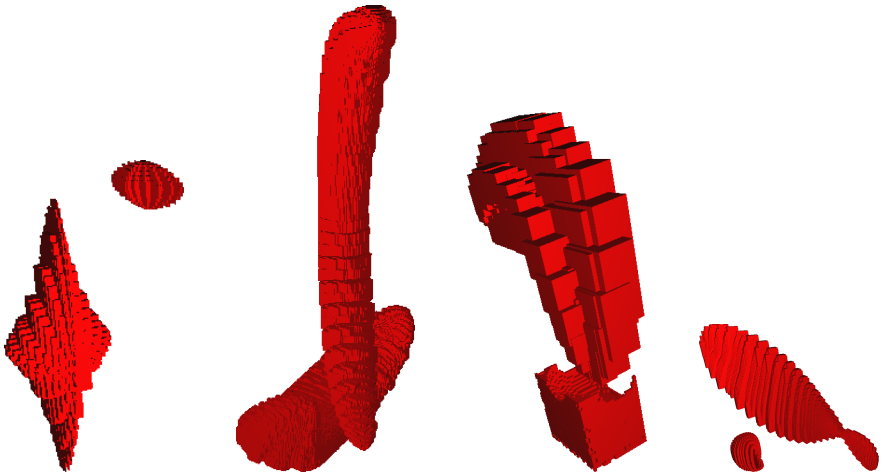


Figure 42: The test cases 1-4 from Tab. 24 (from left to right).

GCC (v. 4.6) compiler, `-O3` optimizations with C-XSC [HK04] as the underlying interval library and FADBAD++ for AD. The maximum CPU time was limited to 600 seconds.

Numerical results

As mentioned above, SQs can be described both by an implicit function or by an explicit parametrization. Because UNIVERMEC supports both possibilities for geometric modeling (cf. Sect. 5.1) and ϵ-distance is capable of deriving rigorous bounds on the distance regardless of the modeling type, we tried to solve the first scenario with both modeling kinds using parametric trees (cf. Sect. 6.1.3) and standard interval trees (cf. Sect. 6.1.1). For the implicit case, it was not possible to complete the scenario in time and after 600 seconds a distance enclosure with the width ≈ 0.35 was obtained for $\epsilon = 0.1$. Because it turned out that the parametric representation is much better suited for distance computation, we used it in the subsequent cases 2-4. Its suitability might originate from the fact that the parametric tree subdivides the parametric domain, which is one dimension lower than the object space subdivided in the implicit model. It is interesting that the use of normals did not make a difference in this test case.

In scenario 2, we considered two non-convex nearly intersecting bent SQs. The use of normal vectors decreased the runtime for $\epsilon = 0.01$ by almost 50 percent. However, even with the normal support, it was not possible to reach a result for $\epsilon = 0.001$ in time.

The third test case consisted of a bent SQ and a non-convex polyhedron. We used the `PolyIVTree` for subdividing the polyhedron (cf. Sect. 6.1.4), which employs a ray intersection test to determine the node colors. This tree structure does not support normals at the moment. So that, the test could be run only without them. Despite these limitations, an enclosure with the width of $\epsilon = 0.01$ could be obtained in the predefined time.

The fourth test case consisted of a bent/tapered SQ and a tapered SQ. Using normal vectors, it was possible to improve the distance enclosure by a large margin and to obtain results for $\epsilon = 0.01$ in time.

As discussed in Sect. 7.1.3, an FP solver can be used to accelerate the ϵ-distance algorithm. This is demonstrated in the second test case where we used the interior point optimizer IpOpt [WB06] which is interfaced to UniVerMeC[8] (cf. Sect. 7.3.3). Note that this optimizer expects the problem in the form (42) (for implicit functions). Thus, the overall procedure benefits from the fact that both implicit and parametric descriptions are supported by UniVerMeC. The implicit description is used for obtaining the initial value using the FP solver whereas the ϵ-distance algorithm works with the parametric one.

Summary

The results of this section show that we can provide a full numerical verification of an important subprocess of the automatic THR surgery assistant system developed by Cuypers [Cuy11] with the help of UniVerMeC. In this scope, we benefit from the fact that UniVerMeC can employ both implicit and parametric descriptions. The obtained bounds on the distance are at least one magnitude smaller than the modeling errors. That is, numerical errors are not the dominating ones. Although our approach is rather slow, it can work also with non-convex objects. It can be carried out offline because the implant selection is performed before the surgery. Currently, the integration of our verified subprocedure into the software program developed by Cuypers [Cuy11] is a future task. During the implementation, the considerations with regard to the other subprocesses made in [ALC13] need to be taken into account additionally.

8.2 VERICELL

VeriCell environment

In this section, we describe the usage of UniVerMeC inside the program VeriCell which was developed by Auer and Pusch [KAR14; Pus13]. VeriCell is a graphical interface for handling SOFC models. An SOFC is a device that converts chemical energy into electricity and is currently being developed as an important building block for decentralized energy supplies. One challenge is robust and reliable control of such devices. To develop such control strategies simplified, but accurate mathematical models for SOFCs are required [Dö+13]. The goal of the VeriCell program is to provide a flexible environment that can be used to work with and test different control-oriented SOFC models. Important tasks to be performed with SOFC models are parameter identification, validation of parameters, and simulation.

Employment of UniVerMeC

Currently, VeriCell considers only the thermal behavior of SOFCs, which can be described by a system of partial differential equations

8 We have to choose a feasible starting point manually in such a way that IpOpt does not converge to the origin (in the local coordinate system) of an SQ because the derivatives required by this solver are not defined there.

(PDEs). Using a spatial semi-discretization, it is possible to derive a set of nonlinear ODEs [Dö+13]. VERICELL works exclusively with these ODEs. The uniform function representation of UNIVERMEC (cf. Sect. 4.4) is used for the right-hand sides of the ODEs. Furthermore, the modeling layer (cf. Sect. 5.2) is employed for the IVPs for the ODEs if a cell is simulated. Additionally, the optimization problem for the parameter identification of the SOFC models can be defined with the help of UNIVERMEC. The simulation of SOFCs is then performed by one of the IVP solvers (cf. Sect. 7.3) interfaced to UNIVERMEC, which can be graphically configured and called from within the VERICELL environment. Besides, VERICELL allows for using the integrated interval global optimization algorithm of UNIVERMEC (cf. Sect. 7.2) or the interior point solver IPOPT interfaced to the framework (cf. Sect. 7.3.3) for parameter identification.

SOFC models

In the rest of this section, we present the results for parameter identification and simulation of SOFC models that were obtained using UNIVERMEC and the interfaced solvers. We work with the control-oriented SOFC models derived in the joint project VERIPC-SOFC between the universities of Rostock and Duisburg-Essen [Dö+13]. Depending on the resolution of the semi-discretization grid, which was used to derive the ODEs, these models have one ($1 \times 1 \times 1$), three ($1 \times 3 \times 1$) or nine states ($3 \times 3 \times 1$). In the following, we consider the first two of these basic models. The first one gives no information about the temperature inside the cell stack, which is partially provided by the second one. The $1 \times 3 \times 1$ models consists of more complex expressions and is more complicated from the numerical point of view. The additional temperature values inside the stack simulate states that cannot be measured.

The simple $1 \times 1 \times 1$ model can be described by

$$\begin{aligned}\dot{\theta}_{FC} = {} & \dot{m}_{H_2} \cdot \left(p_{\Delta H,2} \cdot \theta_{FC}^2 + p_{\Delta H,1} \cdot \theta_{FC} + p_{\Delta H,0}\right) \\ & + 6 \cdot p_A \cdot (\theta_A - \theta_{FC}) + (\theta_{AG} - \theta_{FC}) \\ & \cdot \left(\dot{m}_{H_2} \cdot (p_{H_2,2} \cdot \theta_{FC}^2 + p_{H_2,1} \cdot \theta_{FC} + p_{H_2,0}) \right. \\ & + \dot{m}_{H_2O} \cdot (p_{H_2O,2} \cdot \theta_{FC}^2 + p_{H_2O,1} \cdot \theta_{FC} + p_{H_2O,0}) \\ & \left. + \dot{m}_{N_2} \cdot (p_{N_2,A,2} \cdot \theta_{FC}^2 + p_{N_2,A,1} \cdot \theta_{FC} + p_{N_2,A,0})\right) \\ & + I_{FC} \cdot p_{el} - \dot{m}_A \cdot (\theta_{FC} - \theta_{CG}) \cdot \left(77 \cdot p_{N_2,C,0}/100\right. \\ & + 11 \cdot p_{O_2,0}/50 + 77 \cdot p_{N_2,C,1} \cdot \theta_{FC}/100 + 11 \cdot p_{O_2,1} \cdot \theta_{FC}/50 \\ & \left. + 77 \cdot p_{N_2,C,2} \cdot \theta_{FC}^2/100 + 11 \cdot p_{O_2,2} \cdot \theta_{FC}^2/50\right)\end{aligned} \tag{62}$$

where $\dot{\theta}_{FC}$ is the state-variable and $\theta_{FC}(0) = 299.7053$ K the initial condition. The model depends on 20 parameters, for example, on the coefficients of second-order polynomials that approximate the heat capacities of hydrogen ($p_{H_2,i}$), nitrogen ($p_{N_2,A,i}$) or air ($p_{O_2,i}$) and, additionally, on 8 time dependent control-input variables. In our IVP representation (cf. Def. 18), these time-dependent parameters are de-

scribed by a piecewise-constant control input function $u : \mathbb{R} \to \mathbb{R}^8$. A detailed description of the parameters and their initializations can be found in [AKR12]. The initial values were obtained by the FP optimizer FMINSEARCH, which is part of the MATLAB package and is an implementation of the Nelder-Mead simplex method [NM65].

Problem statement

The identification of the parameter values of the SOFC models is an important task because good choices of the parameters enable us to adapt the model closely to the existing real system. Usually, the parameter identification is performed by solving a least-squares optimization problem. In our case, the objective function φ of the optimization problem (38) is given by

$$\varphi(p) := \sum_{i=1}^{l} \sum_{j=1}^{s} \left(y_j(t_i, p) - y_{j,meas}(t_i) \right)^2 \tag{63}$$

where $p \in \mathbb{R}^n$ is the vector containing the parameters to be identified[9], $y(t_i, p)$ the simulated temperature at time t_i, $y_{meas}(t_i)$ the measured temperature, s the number of measurable states, and l the number of measurements. In our case, there are $l = 19964$ equidistant measurements (every second) and one ($1 \times 1 \times 1$) or two ($1 \times 3 \times 1$) measurable states. We assume without loss of generality in the formula (63) that states with numbers 1 through s are measurable. The large number of measurements does not only make evaluating φ expensive computationally but also increases the overestimation due to the dependency problem if evaluated with IA. Additionally, least-square problems such as (63) often suffer from cancellation effects in general, which increases the overestimation even more [Kie+11].

Solving the IVP

In our case, we have no analytical solution for (62) (and for the higher-dimensional $1 \times 3 \times 1$ model). Therefore, to obtain fully rigorous results on the minimum of φ, we have to solve the IVP numerically using a verified solver such as VNODE-LP [Ned06]. Although solving this problem with VNODE-LP is possible in UNIVERMEC, this time-consuming process slows down the global optimization algorithm. Therefore, we followed the approach of Rauh et al. [Rau+12] and used Euler's method to approximate the IVP:

$$y^{(k)} := y^{(k-1)} + h \cdot f(y^{(k-1)}, p) \ . \tag{64}$$

Here, f is the right-hand side of (62), $y^{(k-1)}$ is the value obtained at the previous time step and h is the step size. Note that we can obtain a *verified approximation* by a simple interval extension of (64). This verified approximation accounts for the rounding errors but neglects the discretization ones. In Fig. 43, the maximum absolute deviation between the results of (64) and those obtained using the fully verified solver VNODE-LP are shown. For a step size of $h = 1s$, which is also

9 Note that we do not necessarily try to identify all parameters but select a subset of them for identification.

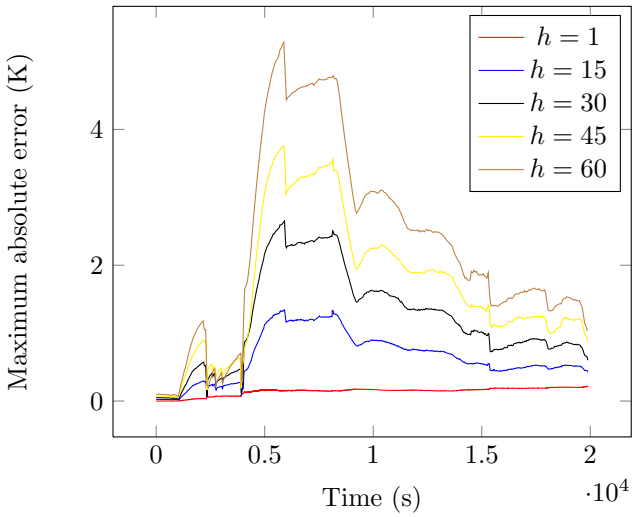


Figure 43: The maximum absolute deviation of the verified enclosure obtained by VNODE-LP and different step sizes h for Euler's method for the $1 \times 1 \times 1$ model.

the sampling rate of the measurements, this deviation is less than 0.25K at each time step. This is acceptable considering that a worst-case measurement error can reach $\pm$15K. The reason for this rather good performance of Euler's method is our small step size in this context [Dö+13]. However, Fig. 43 also shows that we cannot increase the step size to decrease our computational effort because this might lead to a much greater error. Thus, we stick to $h = 1s$ in the rest of our discussion.

Advantages of Euler's method

Using Euler's method, we can derive an explicit expression for the objective function φ:

$$\varphi(p) := \sum_{i=1}^{l} \sum_{j=1}^{s} \left(y_j^{(i-1)}(p) + f(y_j^{(i-1)}(p), p) - y_{j,meas}(t_i) \right)^2 \quad . \quad (65)$$

With this explicit expression, it is possible to compute the derivatives of φ straightforwardly by AD. Note that if we apply a numerical solver to find the solution to the IVP, computing proper enclosures of the solution derivatives require solving an additional system of ODEs for the sensitivities of the original problem [Kie+11]. Obviously, this process is very expensive computationally. Using the verified approximation approach, we can speed up the function evaluations of φ additionally by employing the GPU. The reason is that the objective function (65) requires only operations which are present in our GPU algebra (16).

Model evaluation on the GPU

To enable model evaluation on the GPU, the CPU implementation of (65) in UniVerMeC is accompanied by a corresponding CUDA kernel, which by using allows the function to be evaluated with both FPA and IA. The kernel is accessed through the interface `IGPUEval` (cf.

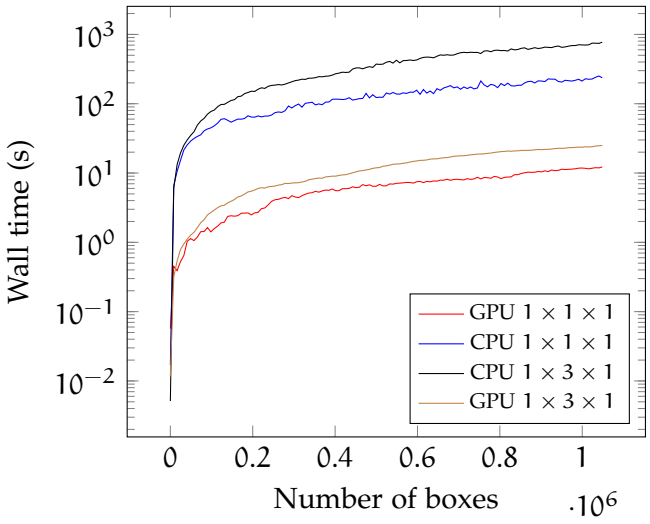


Figure 44: Benchmarks for the evaluation of the objective function on the GPU and the CPU. The given time is the wall time on a logarithmic scale. For the GPU, it includes the necessary memory transfers.

Sect. 4.4.2) at the function layer and, thus, can be integrated straightforwardly into our global optimization algorithm (cf. Sect. 7.2.3).

Benchmark results for evaluations of the objective function (65) for both models are shown in Fig. 44. The simulations were performed on an Intel Xeon CPU E5-2680 with 8 cores, 64 GB RAM and a GeForce GTX 580 with 512 CUDA cores and CUDA [NVI12] (V. 4.2) (cf. Sect. 8.1.1 for further details on the computer configuration). The benchmarks start with $n = 1$ boxes. In every step, we increase n by 8192 until $n > 1048576$. For the simple $1 \times 1 \times 1$ model, the GPU implementation has a speedup of approximately 19 compared to the parallel CPU one. In the $1 \times 3 \times 1$ case, the achieved speedup is 30. Note that the GPU kernel is not optimized heavily. Our comparison between an optimized GPU kernel of the nine-dimensional model and the non-optimized variant in [KAR13] indicated only a minor speedup. This might be due to the high arithmetic density of (65) compared to the relatively small amount of data that is required during the evaluation. In fact, the kernel requires only the input parameters selected for identification, the fixed set of constant parameters, the time-dependent control variables, and the measurements for each time step. Note that the latter three sets of data can be shared between all threads on the GPU.

Consistency of parameter sets

To classify parameters, Rauh et al. [Rau+12] introduced the notion of a *consistent parameter set*. A parameter set p is called consistent if the condition

$$y_j(t_i, p) \in \left(y_{j,meas}(t_i) + \delta\right) \tag{66}$$

is fulfilled for each time step $i = 1, \ldots, l$ and each measurable state $j = 1, \ldots, s$, where $\delta = [-15, 15]$ is the worst-case measurement error.

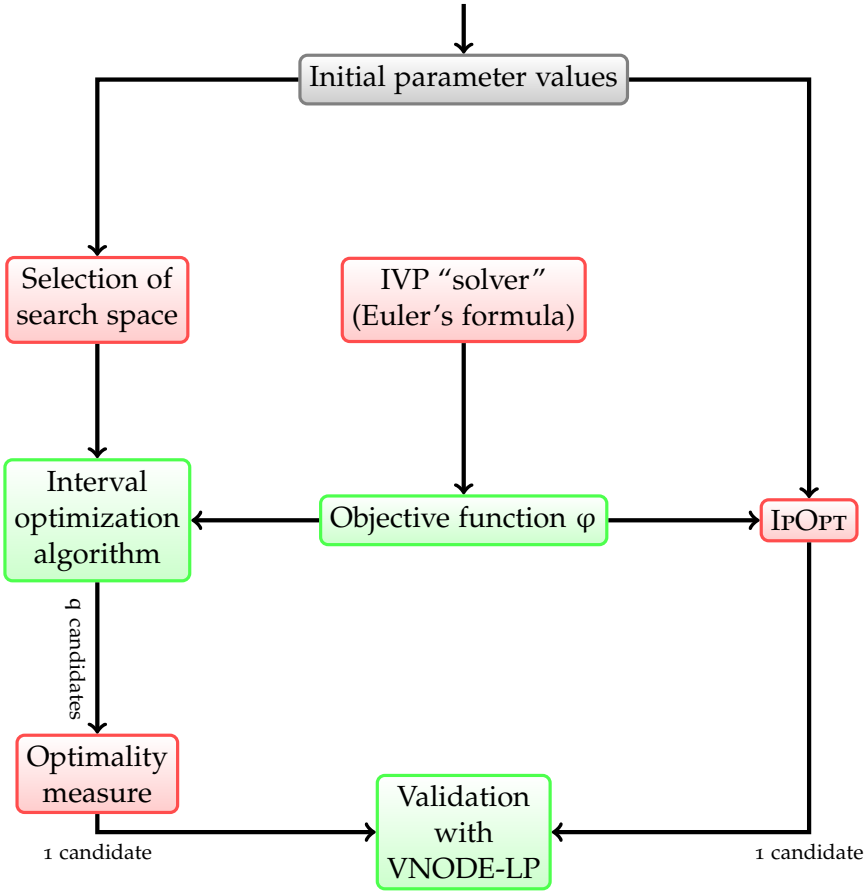


Figure 45: Overview of the procedure for identifying consistent parameter sets for SOFC models in UNIVERMEC. Steps that are verified are shown in green boxes and non-verified steps in red boxes.

In case of verified interval evaluations, $\mathbf{y}_j(t_i, \mathbf{p})$ needs to be a subset of $y_{j,meas}(t_i) + \delta$. Analogously, a parameter set is called inconsistent if $y_j(t_i, p) \notin \left(y_{j,meas}(t_i) + \delta\right)$ or, in the interval case,

$$\mathbf{y}_j(t_i, \mathbf{p}) \cap \left(y_{j,meas}(t_i) + \delta\right) = \emptyset \tag{67}$$

holds at least for one time step $i \in \{1, \ldots, l\}$ and measurable state $j \in \{1, \ldots, s\}$. In UNIVERMEC, it is possible to validate that a parameter set is consistent with respect to (66).

Obtaining consistent states

The complete procedure for identifying a consistent parameter set using UNIVERMEC is shown in Fig. 45. Currently, UNIVERMEC offers two distinct ways to find such a parameter set. Either the integrated interval optimization algorithm (cf. Sect. 7.2.2) or the external FP solver IPOPT (cf. Sect. 7.3.3) can be used. In both cases the procedure starts with providing an initial guess for the parameter values. Such guesses can be obtained, for example, from an external source (e.g., MATLAB's FMINSEARCH in [Rau+12] or setting all parameters to zero) or from a previous run of our framework. Note that using arbitrary initial values (e.g., setting all parameters to zero) is supported only

Table 26: Results of the parameter identification for the $1 \times 1 \times 1$ model. The first column identifies the used solver. If the initial value for the solver is known, it is given in the second column. The errors measure is given in the third column. In the fourth column, the wall time is specified. The last column indicates whether the identified parameter set fulfills (66).

SOLVER	p_{init}	e	TIME	CONSISTENT?
IPOPT	$(0, \ldots, 0)$	2.39 K	555.5 s	✓
FMINSEARCH	?	8.24 K	$\approx$ 21600 s	×
Interval opt.	FMINSEARCH	7.84 K	4130.86 s	×

if IPOPT is used in the next step of the procedure. The reason is that IPOPT can optimize with respect to the complete parameter set. In contrast, our interval solver considers only 6 parameters[10] at the moment and tries to improve a local solution found by another solver using mainly the test for consistent parameter sets. A search space consisting of all parameters is too large because of the branch and bound nature of the interval method. Therefore, we restrict the search space to

$$\mathbf{x}^{(0)} = p_{init} + [-1, 1]^6 \tag{68}$$

where $\mathbf{p}_{init} \in \mathbb{R}^6$ is the initial guess for the selected parameters. The remaining parameters are replaced by their FP values. Note that the algorithm itself (restricted to the search space) and the interval evaluation of φ on the CPU and the GPU are verified[11]. At termination, the interval optimization algorithm returns a list containing q candidate interval vectors for the parameter set. Following Rauh et al. [Rau+12], we identify each parameter candidate interval vector with its midpoint and calculate the following measure:

$$e = \sqrt{\frac{\sum_{i=1}^{l} \sum_{j=1}^{s} \left(y_j(t_i, \operatorname{mid} \mathbf{p}) - y_{j,meas}(t_i) \right)^2}{l}} \, .$$

For the further processing we select the candidate with the smallest value of e. This step is not necessary if IPOPT is used because this solver returns only one solution. In the last step, we check whether the parameter set is consistent. This step is again fully verified because we employ the rigorous IVP solver VNODE-LP to calculate state enclosures for each time step t_i, $i = 1, \ldots, l$ and check for each t_i whether the interval equivalent of (66) holds.

Solver configuration: IpOpt

In both considered cases ($1 \times 1 \times 1$ and $1 \times 3 \times 1$ models), the results

10 These parameters are selected based on their importance for the process [AKR12].

11 Euler's method employed inside φ to solve the IVP is not verified and if IPOPT evaluates φ FPA is used instead of IA.

for IPOPT were obtained with the test configuration described above. Other parameter sets were obtained at a test system of the University of Rostock and are described in [Dö+13; Rau+12]. In any case, the validation with VNODE-LP was carried out within UNIVERMEC. One difficulty during the usage of IPOPT is to obtain the derivative values of (65). Although the explicit expression for φ obtained by Euler's formula allowed us to employ AD, the large number of measurements and the nesting caused by the recursive nature of (64) slowed the used FADBAD++ library down significantly. Therefore, we opted not to provide the second order derivatives usually required by IPOPT but used an option of the solver that automatically approximates the required values based on the first-order information that we supply.

Solver configuration: Interval optimization

The interval global optimization algorithm was configured with the following strategy:

STRAT_SPLIT Bounding the objective function with IA and testing (67)

STRAT_A Testing feasibility of the box

STRAT_POS_INFEAS Applying box consistency on the constraints

STRAT_TMP Trying to find an upper bound $\overline{\varphi^*}$ and using the midpoint test (53).

Additionally, the maximum subdivision depth was set to 10^{-2} and the maximum number of iterations to 200000. The starting box $\mathbf{x}^{(0)}$ was obtained using (68). Based on the starting box, we added bound constraints to the problem to ensure that the condition (52) held for our problem. Note that the only test for discarding boxes besides the midpoint test (53) is the inconsistency condition (67). In particular, we refrain from the use of derivatives for two reasons: First, AD is not supported on the GPU at the moment in UNIVERMEC[12] and obtaining analytical closed form expressions of the derivatives for manual implementation on the GPU is not suitable in this case. Second, we use the criterion (67) not only to check for inconsistent states but also for limiting the overestimation during the evaluation of φ. This makes (65) non-differentiable due to the intersection operator. Our method is largely similar to the algorithm used in [Rau+12]; however, we use the Ratz section scheme (cf. Sect. 6.2) as opposed to the standard bisection used there. Moreover, our implementation is faster since we employed CPU and GPU parallelization.

$1 \times 1 \times 1$ results

The numerical results for the $1 \times 1 \times 1$ model are given in Tab. 26. Bearing in mind that the wall times are not completely comparable, it seems fair to state that IPOPT clearly outperforms FMINSEARCH in this case. Using IPOPT, we were even able to obtain a consistent parameter set where FMINSEARCH failed to provide one. We used the interval

12 Refer to Sect. 3.7 for general information on limitations of GPU computations in UNIVERMEC and to [KK13] for information on AD with IA on the GPU which might be used to extend the framework in the future.

Table 27: Results of the parameter identification for the $1 \times 3 \times 1$ model. The first column identifies the used solver. If the initial value for the solver is known, it is given in the second column. The errors measure is given in the third column. In the fourth column, the wall time is specified. The last column indicates whether the identified parameter set fulfills (66).

SOLVER	p_{init}	e	TIME	CONSISTENT?
IPOPT	$(0, \ldots, 0)$	430.55 K	2817.23 s	×
FMINSEARCH	?	37.57 K	?	×
IPOPT	FMINSEARCH	26.59 K	235.565 s	×
Interval opt.	FMINSEARCH	57.31 K	5021.06 s	×

optimization routine only with the results of FMINSEARCH because IPOPT delivered an already consistent parameter set. Compared to the original FP results, the use of the interval routine led to a decrease of the error measure. However, the returned set is still not consistent. The GPU accelerated search routine required a little more than an hour to obtain the results. This is much faster than the pure CPU method described in [Rau+12] which needed $\approx$ 10 hours[13]

$1 \times 3 \times 1$ results

The results for the higher dimensional $1 \times 3 \times 1$ model are given in Tab. 27. In this case, IPOPT failed to provide acceptable bounds when initialized with zero for all parameters. The reason might be the numerical problems originating from the approximation of the second-order information. Therefore, we used the results obtained in Rostock with the FMINSEARCH method to initialize IPOPT. In this case, the solver terminated after 139 iterations due to errors in the evaluation (63) or its derivatives. The results given in the table are for a maximum of 130 iterations, which led to a slight improvement over the initial FMINSEARCH results. Note that the comparatively low wall time is caused by the facts that the solver obtained a good initial bound and performed relatively few iterations as well as that the evaluation of derivatives is multi-threaded[14] inside UNIVERMEC.

We also tried to improve the results of FMINSEARCH using interval methods in this case. Here, the GPU interval optimization had not any advantage but led to a worse parameter set. One reason for the bad interval results might be that the more complex $1 \times 3 \times 1$ model contains non-measurable states for which we cannot use the condition (67) to limit the overestimation.

Validation results

During the validation step (cf. Fig. 45), the SOFCs are simulated using the rigorous solver VNODE-LP. In Fig. 46, the simulation re-

13 Note that the computations were carried out on different systems and slightly different versions of the SOFC models.

14 The derivatives have to be obtained by forward differentiation because FADBAD++ cannot handle the large computational graph of (65) with backward differentiation. The total CPU time required by IPOPT was 2791.14s.

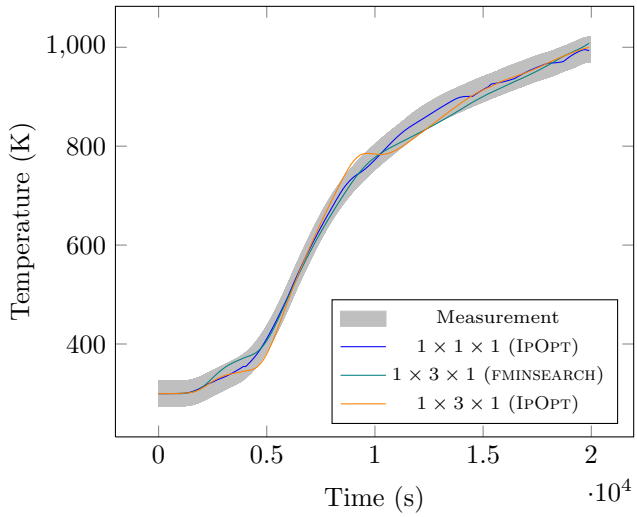


Figure 46: Simulation results for the parameter sets from Tab. 26, 27 obtained with VNODE-LP. In the plot, the rigorous upper bound on the simulated temperature and the consistent area are shown.

sults for the parameter set for the $1 \times 1 \times 1$ model obtained by IpOpt and the two parameter sets for the $1 \times 3 \times 1$ model are shown. For a complete simulation run with one test parameter set, VNODE-LP required approximately 1700 seconds of CPU time on our test system. In the plot, we show only the upper bound on the simulated temperature and the state θ_3[15] of the three dimensional model. The parameter set for the $1 \times 1 \times 1$ model clearly lies inside the gray tolerance range for the measurements. The $1 \times 3 \times 1$ parameter set obtained by IpOpt violates this area. For the other considered parameter set that was obtained by fminsearch, the violation of the consistency conditions is less noticeable. However, it is also inconsistent and has a larger deviation for the measurable state θ_0 not shown in the plot.

Future improvements

The usage of UniVerMeC allowed us to combine different verified and non-verified solvers to obtain a parameter set for the one-dimensional model and to rigorously prove that these parameters are indeed consistent. Furthermore, the objective function evaluation benchmark (cf. Fig. 44) and the comparison of the runtimes of our GPU accelerated interval global optimization method to the method described in [Rau+12] showed that the employment of GPGPU can lead to a significant speed-up in interval computations.

However, it was not possible in general to improve the parameter sets obtained by the FP solvers by the interval algorithm. While one reason is certainly that the interval optimization algorithm only performs a local search in our configuration due to memory and runtime restrictions, another reason might be that we employ standard IA without techniques to reduce the wrapping effect or the dependency problem apart from the condition (67), which in turn makes (65) non-

15 θ_3 corresponds to θ in the one-dimensional model.

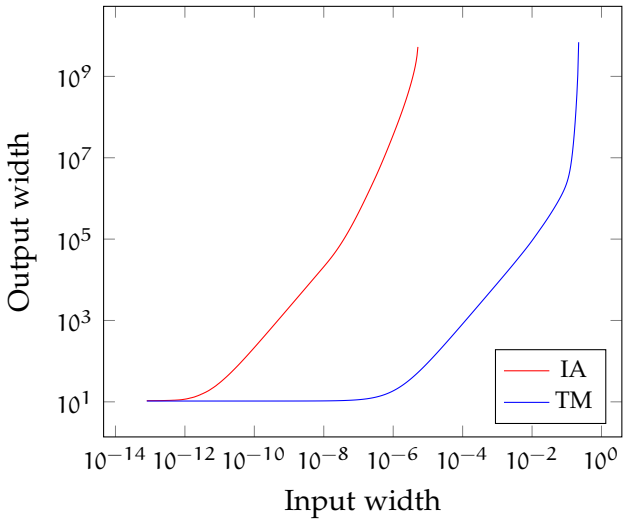


Figure 47: Width of output intervals with respect input width (sum of the widths of all components) for the objective function φ (65) and the $1 \times 1 \times 1$ model.

differentiable. To demonstrate the large overestimation, we consider the result width of (65) for the one-dimensional model in dependence on the widths of the inputs parameters. Using standard IA, the evaluation overflows if the total input uncertainty is greater than $\approx 5 \cdot 10^{-6}$ (cf. Fig. 47). With TMs, we can improve the situation significantly at the expense of an increased computational effort. Twenty evaluations of φ with IA require 0.23 seconds CPU time on our test system as opposed to TMs, which required 307.12 seconds. Additionally, it is not possible to use the employed TM library Riot in a multi-threaded environment (or on the GPU). Despite these limitations, using TMs for range enclosure in the interval global optimization algorithm of UniVerMeC is worth considering.

Another approach, as already mentioned above, aims at the complete rigorous solving of the optimization problem by replacing Euler's method by a verified IVP solver [Kie+11]. Such solvers feature specialized techniques to reduce overestimation, which might improve the range bounds on the φ. Currently, this aspect is investigated with the help of UniVerMeC in the scope of a master's thesis [Pus13].

8.3 CONCLUSIONS

Comparisons between arithmetics

In this chapter, we reported on numerical results of the usage of UniVerMeC in two practical applications. First, we considered use cases where geometric objects were described by implicit functions. Here, we used the ϵ-distance algorithm to derive rigorous bounds on the distance between those models. Moreover, we examined how different verified range arithmetics behaved in the scope of rigor-

ous distance computation. For this purpose, we constructed several test cases, grouped them according to the complexity of the expressions describing the geometric objects, and derived bounds on the distance between them using different arithmetics for enclosing the corresponding ranges. The use of UNIVERMEC ensured that the overall overhead was the same for all arithmetics and, thus, the comparison was fair. Here, we confirmed our previous results from [DK12] and found that using AA was a good overall choice without the information provided by the normal cone test (51). If (51) was used the test results indicated that the mean-value form (20) was the favorable choice for the range enclosure. Additionally, the test confirmed the usefulness of our contracting tree structure introduced in Sect. 6.1.2, of the ϵ-distance algorithm itself (cf. Sect. 7.1.1) and of its extension with normals (cf. Sect. 7.1.2).

Application in THR assistance

Our second practical use case was from the area of distance computation again. We considered an automatic assistance system for THR surgery where various distance queries between possibly non-convex SQs and polyhedra were carried out. We demonstrated that UNIVERMEC could handle these kinds of distance queries in a rigorous manner. This use case confirmed not only that the verification of this important subprocess of the THR assistance system was possible but also that UNIVERMEC could carry out distance queries between different modeling types even if one modeling type was non-smooth (e.g, a polyhedron). Additionally, we showed that the method for improving the runtime of the ϵ-distance algorithm using an FP optimizer actually had the desired effect.

Application in parameter identification and simulation of SOFCs

Our last use case was the parameter identification for control-oriented SOFC models. Here, we demonstrated how different rigorous and non-rigorous solvers could work together sharing the same problem description. Although we could not derive fully rigorous bounds on the optimal parameters, it was possible to find a parameter set using an FP solver that had a much smaller error measure than the sets identified previously. Using UNIVERMEC, it was possible to integrate this solver seamlessly into a procedure that proved with the help of a verified IVP solver that the enclosures of simulated states of the SOFC model with this new parameter set were inside the tolerance range defined by measurements on a real SOFC for every time step. Finally, we demonstrated that the evaluation of the objective function of an optimization problem on the GPU could lead to a significant speed up of our interval global optimization algorithm.

9 CONCLUSIONS

In this thesis, we presented the theoretical foundations of interoperable handling of verified techniques supplemented by a prototypical implementation UniVerMeC. A goal of our approach was to make combination of existing methods or their interchangeable use possible. The interchangeability of techniques allowed us to achieve our next goal: It ensured fair comparisons between the techniques because, using our approach, it was possible to exchange a single technique (e.g., the used range arithmetic) while keeping the environment constant (e.g., same algorithm implementation, same test scenarios). This level of abstraction during the assessment process is new in the area. In addition to providing fair comparisons, it allowed for a faster development of new methods in the geometrical context in this thesis since basic building blocks could be reused and exchanged easily. Besides this, our approach facilitated application of verified techniques to real life problems as demonstrated in this thesis for the areas of total hip replacement and SOFC control.

Arithmetics

The structure of this thesis followed roughly the layered hierarchy principle employed as the basic architectural pattern of our prototype implementation. First, we discussed and specified the necessary general requirements for a framework implementing the goals above. After that, we introduced rigorous range arithmetics, which are the most basic building blocks of verified numerical computations. In this scope, we presented a new flexible library for AA called YalAA. It handles different affine computation models and features in contrast to existing AA implementations, verified affine extensions of elementary functions that are neither convex nor concave. On the level of arithmetics, we introduced a heterogeneous algebra that defines common sets of operations for FPA, IA, AA and TMs. These sets are oriented on the upcoming IEEE P1788 standard. This algebra is extended by hierarchically arranged safe conversions between the different arithmetic types. Based on those theoretical considerations, we designed and realized a generator for semi-automatic creation of adapters so that different libraries implementing the basic arithmetic types can be easily integrated in the overall system. Additionally, we specified a refinement of this algebra for the GPU. The set of operations and supported arithmetics for the GPU is smaller as a consequence of the currently limited state of GPU support in verified computations.

Functions

The second layer we discussed concerned important auxiliary methods such as AD, function enclosures, and interval contractors, which are necessary for the algorithms we consider later. Our main con-

tribution at this level is the introduction of FROs that are theoretical constructs for characterizing a real function by a number of inclusion functions. Other important characteristics, for example, derivatives, contractors, or Taylor coefficients, can be represented and attached to an FRO. This concept is embedded into our universal framework. This theoretical basis allowed us to derive a software realization of a homogeneous function type that is independent of the actual arithmetic. In this scope, we also described the integration of function evaluations on the GPU into our implementation.

Problem modeling

Arithmetics and functions are the most important basic building blocks for model descriptions in many application areas. Therefore, after introducing uniform concepts for both, we investigated how different modeling and problem types could be described with their help inside UniVerMeC. Our examples were from the areas of geometric models, IVPs and optimization.

Hierarchical decompositions

Having described our modeling types, we introduced hierarchical decompositions that played an important role in the algorithms we were interested in. The main contributions at this level are the following: first, we introduced a flexible theoretical foundation for describing various interval trees. Second, we presented two new tree types. The first was the *contracting tree*, which made use of interval contractors to reduce uncertain regions during the decompositions. Moreover, it featured special inversion nodes for better handling of regions produced by these contractors. The other tree structure was introduced for dealing with parametric surfaces. Aside from demonstrating how those tree types could be implemented in UniVerMeC, we discussed the integration of various well-known multisection schemes from global optimization.

Algorithms

The usefulness of providing uniform access to various verified methods was further demonstrated by different algorithms supported inside the framework. In this scope, we presented a new algorithm for obtaining rigorous enclosures on the minimum distance between non-convex objects described by different geometric modeling types. The implementation of the algorithm ensured flexible access to and easy reuse of various verified methods. As a second example, we discussed the implementation of a state-of-the-art interval global optimization algorithm inside the framework. The focus of the discussion was how UniVerMeC facilitated the implementation of a highly configurable algorithm. Furthermore, we explained how the capability to evaluate certain functions on the GPU could be integrated into the optimization method. Finally, we demonstrated how various third party solvers could be interfaced with the framework. In this way, they worked with the uniform modeling descriptions provided by UniVerMeC and could be integrated into general computation scenarios if necessary.

Practical test scenarios

To demonstrate the practicability of our approach, we considered

three real life motivated test scenarios. In the first one, we computed rigorous distance enclosures for 16 different test cases that were described by implicit functions and grouped according to the complexity of their expressions. We solved each test case with the help of different techniques for range bounding and compared the required CPU times. The obtained results demonstrated not only that the new distance algorithm was capable of solving all test cases but also that UNIVERMEC was suitable for easy and fair comparison between different arithmetics. Moreover, our new tree structures turned out to be an improvement compared to standard trees for certain test cases.

As a second scenario, we considered the verification of an important subprocess of a system for automatic selection of an appropriate implant in the context of THR surgery. The considered subprocess consisted of several distance queries between two possibly non-convex SQs, or a non-convex SQ and a non-convex polyhedron. Using our new distance algorithm, it was possible to derive verified enclosures for all those scenarios. This result demonstrates that our algorithm can derive distances between objects described by different geometric modeling types. More generally, it shows that verification of this important subprocess in the overall THR assistance system is possible.

Our last scenario concerned parameter identification of control-oriented SOFC models. To identify an optimized set of parameters for different SOFC models, we applied our global optimization method and interfaced third party optimizers. In addition, we demonstrated how function evaluation on the GPU can accelerate the identification process. The computed parameter sets were examined with respect to their compliance with measurements taken using a real SOFC testrig. The consistency check was performed by means of a rigorous IVP solver interfaced to UNIVERMEC. We were able to prove that a consistent set of parameters was identified. In general, this test scenario demonstrated how the framework allowed us to use different techniques provided by internal and external solvers and to combine them in one process to tackle complex practical problems.

Future research

An important aspect for further improvements is the full incorporation of the upcoming IEEE P1788 interval standard. Wide acceptance of IEEE P1788 throughout the interval community would make certain tasks easier performed by UNIVERMEC (e.g., interfacing of third party libraries for verified computation). However to realize this point, we have to wait until IA libraries (e.g., C-XSC) and the third party libraries implementing the solvers conform to the standard. Another important goal for the whole framework would be to unify the CPU algebra (15) and the GPU algebra (16) as soon as the corresponding lower-level libraries provide the necessary support. A full unification could be expected only if the other arithmetics aside from IA were to support the GPU computations. At least for AA, preliminary work has been done in the non-rigorous case [Kno+09].

Aside from these general improvements at the basic level of the framework, more specific ones are possible at the higher layers. For example, further enclosure methods or contractors at the function layer would be a valuable addition for the algorithms working in UniVerMeC. The modeling layer can be extended by further problem types if users from further application areas feel the need to do so. At the decomposition layer, it would be interesting to test whether the black inversion nodes outlined in Sect. 6.1.2 improve the performance of contracting trees. A further candidate for improvement at this layer is the handling of polyhedra by special data structures (e.g., extended octrees [BN90]). The ϵ-distance algorithm can be improved, on the one hand, by replacing the case selector (43) with more case-specific procedures. On the other hand, the ϵ-distance algorithm would benefit greatly from parallelization. Our global optimization method could be improved by providing further state-of-the-art techniques (see [Neu04] for an overview). Furthermore, the GPU integration should be improved as soon as the library support becomes adequate. To extend the scope of UniVerMeC with respect to application areas, it would be interesting to add further algorithms (e.g., the path planning approach from [Jau01]) or interface more solvers (e.g., GlobSol [Kea03]). To further facilitate the usage of UniVerMeC we plan to make it available as open-source software in the near future.

The results for the test scenarios used to evaluate UniVerMeC were satisfactory in general. Especially for the distance computations in the THR procedure, it would be interesting to find ways to speed up the process. In our experience, the evaluation of powers in SQ expressions is one of the main difficulties in this special scenario. Therefore, aside from the techniques outlined above (e.g., parallelization), using faster implementations[1] for the power function might lead to a decrease in computation times. However, such steps should carried out by the developers of low-level IA libraries directly.

Finally, we see the parameter identification procedure as a further candidate for improvements. For example, it might be interesting to test how other FP solvers that do not require the Hessian matrix of the objective function (65) perform with the considered optimization problem because IpOpt seemed to suffer from the Hessian matrix approximation technique that we used to achieve acceptable computation times. The complete verification of the identification process is, currently, being investigated in the scope of UniVerMeC in an ongoing master's thesis [Pus13].

1 Refer to [HNW13] for a discussion of the general interval power function.

OWN PUBLICATIONS

[AKR12] E. Auer, S. Kiel, and A. Rauh. "Verified Parameter Identification for Solid Oxide Fuel Cells." In: *Proceedings of the 5th International Conference on Reliable Engineering Computing*. 2012 (cit. on pp. vii, viii, 18, 127, 170, 174).

[Aue+10] E. Auer et al. "Verification and Validation for Femur Prosthesis Surgery." In: *Computer-assisted proofs - tools, methods and applications*. Ed. by B. M. Brown et al. Dagstuhl Seminar Proceedings 09471. Schloss Dagstuhl, 2010. URL: http://drops.dagstuhl.de/opus/volltexte/2010/2513 (cit. on p. vii).

[Aue+11] E. Auer et al. "Relevance of Accurate and Verified Numerical Algorithms for Verification and Validation in Biomechanics." In: *EUROMECH Colloquium 511*. Ponta Delgada, Azores, Portugal, 2011 (cit. on pp. vii, 116, 164, 165).

[CKL09] R. Cuypers, S. Kiel, and W. Luther. "Automatic Femur Decomposition, Reconstruction, and Refinement Using Superquadric Shapes." In: *Proceedings of the IASTED International Conference*. Vol. 663. 2009, p. 59 (cit. on p. vii).

[DK10] E. Dyllong and S. Kiel. "Verified Distance Computation Between Convex Hulls of Octrees Using Interval Optimization Techniques." In: *PAMM* 10.1 (2010), pp. 651–652. ISSN: 1617-7061 (cit. on pp. vii, viii, 118, 126, 127, 140).

[DK12] E. Dyllong and S. Kiel. "A Comparison of verified distance computation between implicit objects using different arithmetics for range enclosure." In: *Computing* 94 (2 2012), pp. 281–296. ISSN: 0010-485X (cit. on pp. vii, viii, 119, 159, 162, 179).

[KAR13] S. Kiel, E. Auer, and A. Rauh. "Use of GPU Powered Interval Optimization for Parameter Identification in the Context of SO Fuel Cells." In: *Proceedings of NOLCOS 2013 - 9th IFAC Symposium on Nonlinear Control Systems*. 2013. DOI: 10.3182/20130904-3-FR-2041.00169 (cit. on pp. viii, 127, 172).

[KAR14] S. Kiel, E. Auer, and A. Rauh. "An Environment for Testing, Verification and Validation of Dynamical Models in the Context of Solid Oxide Fuel Cells." In: *Reliable Computing* 19.3 (2014), pp. 302–317 (cit. on pp. viii, 150, 168).

[Kie12a] S. Kiel. "Verified Spatial Subdivision of Implicit Objects Using Implicit Linear Interval Estimations." In: *Curves and Surfaces*. Ed. by J.-D. Boissonnat et al. Vol. 6920. Lecture Notes in Computer Science. Springer, 2012, pp. 402–415 (cit. on pp. vii, viii, 33, 104, 162).

[Kie12b] S. Kiel. "YalAA: Yet Another Library for Affine Arithmetic." In: *Reliable Computing* 16 (2012), pp. 114–129 (cit. on pp. vii, viii, 12, 31, 38, 40, 159).

[KLD13] S. Kiel, W. Luther, and E. Dyllong. "Verified distance computation between non-convex superquadrics using hierarchical space decomposition structures." In: *Soft Computing* 17.8 (2013), pp. 1367–1378. ISSN: 1432-7643 (cit. on pp. vii, viii, 69, 119, 164, 165).

REFERENCES

[Ago05a] M. K. Agoston. *Computer Graphics and Geometric Modeling - Implementation and Algorithms*. Springer, 2005 (cit. on p. 93).

[Ago05b] M. K. Agoston. *Computer Graphics and Geometric Modeling - Mathematics*. Springer, 2005 (cit. on pp. 93, 94).

[AH83] G. Alefeld and J. Herzberger. *Introduction to Interval Computations*. New York: Academic Press, 1983 (cit. on pp. 26, 67).

[AL09] E. Auer and W. Luther. "Numerical Verification Assessment in Computational Biomechanics." In: *Numerical Validation in Current Hardware Architectures*. Ed. by Annie Cuyt et al. Vol. 5492. Lecture Notes in Computer Science. Springer, 2009, pp. 145–160. ISBN: 978-3-642-01590-8 (cit. on pp. 10–12, 164).

[ALC13] E. Auer, W. Luther, and R. Cuypers. "Process-oriented verification in biomechanics." In: *Procedings of ICOSSAR 2013: 11th International Conference on Structural Safety & Reliability*. New York, 2013 (cit. on pp. 164, 165, 168).

[Ale01] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2001 (cit. on pp. 31, 41, 63).

[AR10] E. Auer and A. Rauh. "Toward Definition of Systematic Criteria for the Comparison of Verified Solvers for Initial Value Problems." In: *Parallel Processing and Applied Mathematics*. Ed. by R. Wyrzykowski et al. Vol. 6068. Lecture Notes in Computer Science. Springer, 2010, pp. 408–417. ISBN: 978-3-642-14402-8 (cit. on p. 156).

[AR12] E. Auer and A. Rauh. "VERICOMP: a system to compare and assess verified IVP solvers." English. In: *Computing* 94 (2-4 2012), pp. 163–172. ISSN: 0010-485X (cit. on pp. 5, 144, 145, 156).

[Bar81] A. H. Barr. "Superquadrics and Angle-Preserving Transformations." In: *Computer Graphics and Applications, IEEE* 1.1 (1981), pp. 11 –23. ISSN: 0272-1716 (cit. on pp. 18, 66, 116, 164).

[BBH89] P. Brown, G. Byrne, and A. Hindmarsh. "VODE: A Variable-Coefficient ODE Solver." In: *SIAM Journal on Scientific and Statistical Computing* 10.5 (1989), pp. 1038–1051 (cit. on pp. 86, 150).

[BDL04] K. Bühler, E. Dyllong, and W. Luther. "Reliable Distance and Intersection Computation Using Finite Precision Geometry." In: *Numerical Software with Result Verification*. Ed. by R. Alt et al. Vol. 2991. Lecture Notes in Computer Science. Springer, 2004, pp. 579–600. ISBN: 978-3-540-21260-7 (cit. on pp. 118–120, 159).

[Bee+04] T. Beelitz et al. *SONIC — A Framework for the Rigorous Solution of Nonlinear Problems*. Tech. rep. Online accessed on 03.06.2013 http://www-ai.math.uni-wuppertal.de/SciComp/preprints/SC0407.ps.gz. Bergische Universität Wuppertal, 2004 (cit. on p. 142).

[Bee06] T. Beelitz. "Effiziente Methoden zum Verifizierten Lösen von Optimierungsaufgaben und Nichtlinearen Gleichungssystemen." PhD thesis. Bergische Universität Wuppertal, 2006 (cit. on pp. 72, 112, 117, 127–129, 131, 139, 143).

[Ben+99] F. Benhamou et al. "Revising hull and box consistency." In: *Logic Programming - Proceedings of the 1999 International Conference on Logic Programming*. Ed. by D. de Schreye. Massachusetts Institute of Technology, 1999, pp. 230 –245 (cit. on pp. 64, 82).

[Ben96] F. Benhamou. "Heterogeneous constraint solving." In: *Algebraic and Logic Programming*. Ed. by M. Hanus and M. Rodríguez-Artalejo. Vol. 1139. Lecture Notes in Computer Science. Springer, 1996, pp. 62–76. ISBN: 978-3-540-61735-8 (cit. on pp. 70, 90).

[Ber04] G. van den Bergen. *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann Publishers, 2004 (cit. on p. 116).

[Ber95a] S. Berner. *Ein paralleles Verfahren zur verifizierten globalen Optimierung*. Shaker, 1995 (cit. on pp. 110, 111, 128, 139).

[Ber95b] M. Berz. "Modern Map Methods for Charged Particle Optics." In: *Nuclear Instruments and Methods A363* (1995), pp. 100–104 (cit. on pp. 1, 44).

[BG06] F. Benhamou and L. Granvilliers. "Chapter 16 - Continuous and Interval Constraints." In: *Handbook of Constraint Programming*. Ed. by F. Rossi, P. van Beek, and T. Walsh. Vol. 2. Foundations of Artificial Intelligence. Elsevier, 2006, pp. 571 –603 (cit. on p. 74).

[BHK03] M. Bräuer, W. Hofschuster, and W. Krämer. *Steigungsarithmetiken in C-XSC*. Tech. rep. Online accessed on 17.8.2012 http://www2.math.uni-wuppertal.de/~xsc/preprints/prep_01_3.pdf. Bergische Unversität GH Wuppertal, 2003 (cit. on p. 68).

[Bil08] G. Bilotta. "Self-verified extension of affine arithmetic to arbitrary order." In: *Le Matematiche* 63.1 (2008), pp. 15–30 (cit. on p. 35).

[BJ05] B. Bastl and F. Jezek. "Comparison of Implicitization Methods." In: *Journal for Geometry and Graphics* 9.1 (2005), pp. 11–29 (cit. on p. 95).

[BL70] G. Birkhoff and J. D. Lipson. "Heterogeneous algebras." In: *Journal of Combinatorial Theory* 8.1 (1970), pp. 115 –133. ISSN: 0021-9800 (cit. on p. 48).

[BM06] M. Berz and K. Makino. *COSY INFINITY 9.0*. Tech. rep. MSUHEP 060803. Michigan State University, 2006 (cit. on pp. 46, 164).

[BMP06] H. Bronnimann, G. Melquiond, and S. Pion. "The design of the Boost interval arithmetic library." In: *Theoretical computer science* 351.1 (2006), pp. 111–118 (cit. on p. 31).

[BN13] P.-D. Beck and M. Nehmeier. "Parallel Interval Newton Method on CUDA." In: *Applied Parallel and Scientific Computing*. Ed. by Pekka Manninen and Per Öster. Vol. 7782. Lecture Notes in Computer Science. Springer, 2013, pp. 454–464. ISBN: 978-3-642-36802-8 (cit. on pp. 60, 129).

[BN90] P. Brunet and I. Navazo. "Solid representation and operation using extended octrees." In: *ACM Transactions on Graphics (TOG)* 9.2 (1990), pp. 170–197 (cit. on p. 184).

[Boo] *Boost C++ Libraries*. Online accessed on 19.06.2012 http://www.boost.org (cit. on p. 31).

[Bus+96] F. Buschmann et al. *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley, 1996 (cit. on pp. 14, 15).

[BV09] S. Boyd and L. Vandenberghe. *Convex Optimization*. Online accessed on 15.05.2013 http://www.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf. Cambridge University Press, 2009 (cit. on pp. 98, 117, 132).

[Büh02] K. Bühler. "Implicit linear interval estimations." In: *Proceedings of the 18th spring conference on Computer graphics*. ACM. 2002, p. 132 (cit. on pp. 74–76).

[C11a] *C Programing Language – ISO/IEC 9899:2011*. Geneva, Switzerland, 2011 (cit. on p. 40).

[C11b] *Information technology – Programming languages – C++ ISO/IEC 14882:2011*. Geneva, Switzerland, 2011 (cit. on pp. 79, 86, 87).

[CE00] K. Czarneski and U. W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000 (cit. on pp. 14, 41, 54).

[Cha] G. Chabert. *IBEX Homepage*. Online accessed on 07.2.2013 http://www.emn.fr/z-info/ibex/ (cit. on pp. 5, 63, 91).

[Cha+08] N. Chakraborty et al. "Proximity Queries Between Convex Objects: An Interior Point Approach for Implicit Surfaces." In: *IEEE Transactions on Robotics* 24.1 (2008), pp. 211–220 (cit. on p. 116).

[Che11] C.-Y. Chen. "Extended interval Newton method based on the precise quotient set." In: *Computing* 92 (4 2011), pp. 297–315. ISSN: 0010-485X (cit. on p. 71).

[Chu11] A. Chuev. "Akkurate Abstandsberechungen zwischen 3D Objekten dargestellt als Superquadriken." MA thesis. Universität Duisburg-Essen, 2011 (cit. on pp. 116, 164, 165).

[CJ09] G. Chabert and L. Jaulin. "Contractor programming." In: *Artificial Intelligence* 173.11 (2009), pp. 1079 –1100. ISSN: 0004-3702 (cit. on pp. 5, 63, 91, 127, 128).

[CJP08] B. Chapman, G. Jost, and R. Van der Pas. *Using OpenMP: portable shared memory parallel programming*. MIT press, 2008 (cit. on pp. 139, 140).

[CKR00] T. Csendes, R. Klatte, and D. Ratz. "A Posteriori Direction Selection Rules for Interval Optimization Methods." In: *CEJOR* 8 (2000), pp. 225–236 (cit. on p. 112).

[Cli] M. Cline. *C++ FAQ 25.10 – What does it mean to "delegate to a sister class" via virtual inheritance?* Online accessed on 29.05.2014 http://www.parashift.com/c++-faq-lite/mi-delegate-to-sister.html (cit. on p. 87).

[CS90] J.L.D. Comba and J. Stolfi. "Affine arithmetic and its applications to computer graphics." In: *Proceedings of VI SIBGRAPI (Brazilian Symposium on Computer Graphics and Image Processing)*. Citeseer. 1990, pp. 9–18 (cit. on pp. 1, 33, 34).

[Cse01] T. Csendes. "New Subinterval Selection Criteria for Interval Global Optimization." In: *Journal of Global Optimization* 19.3 (2001), pp. 307–327. ISSN: 0925-5001 (cit. on pp. 111, 122).

[Cud] *NVIDIA CUDA C Programming Guide*. 4.2. NVIDIA. 2012 (cit. on p. 89).

[Cuy11] R. Cuypers. *Geometrische Modellierung mit Superquadriken zur Optimierung skelettaler Diagnosesysteme*. Logos, 2011 (cit. on pp. 18, 116, 153, 164, 168).

[DFS11] F. Domes, M. Fuchs, and H. Schichl. *The Optimization Test Environment*. Online accessed on 06.08.2013 http://www.mat.univie.ac.at/~dferi/testenv.html. 2011 (cit. on pp. 6, 145).

[DG07a] E. Dyllong and C. Grimm. "A Modified Reliable Distance Algorithm for Octree-encoded Objects." In: *PAMM* 7.1 (2007), pp. 4010015–4010016 (cit. on p. 118).

[DG07b] E. Dyllong and C. Grimm. "An Efficient Distance Algorithm for Interval-Based Octree-Encoded CSG Objects with Time-Space Coherence Utilization." In: *PAMM* 7.1 (2007), pp. 1023007–1023008 (cit. on p. 118).

[DG07c] E. Dyllong and C. Grimm. "Proximity Queries between Interval-Based CSG Octrees." In: *AIP Conference Proceedings* 936.1 (2007). Ed. by Theodore E. Simos, George Psihoyios, and Ch. Tsitouras, pp. 162–165 (cit. on pp. 2, 127).

[DG07d] E. Dyllong and C. Grimm. "Verified Adaptive Octree Representations of Constructive Solid Geometry Objects." In: *Simulation und Visualisierung*. Citeseer. 2007, pp. 223–235 (cit. on pp. 99, 100, 102, 113, 114, 127).

[DG08] E. Dyllong and C. Grimm. "A Reliable Extended Octree Representation of CSG Objects with an Adaptive Subdivision Depth." In: *Parallel Processing and Applied Mathematics*. Ed. by R. Wyrzykowski et al. Vol. 4967. Lecture Notes in Computer Science. Springer, 2008, pp. 1341–1350 (cit. on pp. 99, 100, 102, 113).

[DK94] K. Du and R. B. Kearfott. "The cluster problem in multivariate global optimization." In: *Journal of Global Optimization* 5 (3 1994), pp. 253–265. ISSN: 0925-5001 (cit. on pp. 18, 124).

[DL04a] E. Dyllong and W. Luther. "An Accurate Distance Algorithm for Octree-Encoded Objects." In: *PAMM* 4.1 (2004), pp. 562–563. ISSN: 1617-7061 (cit. on p. 118).

[DL04b] E. Dyllong and W. Luther. "The GJK distance algorithm: an interval version for incremental motions." In: *Numerical Algorithms* 37.1 (2004), pp. 127–136. ISSN: 1017-1398 (cit. on p. 116).

[DM98] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming." In: *Computational Science Engineering, IEEE* 5.1 (1998), pp. 46 –55. ISSN: 1070-9924. DOI: 10.1109/99.660313 (cit. on p. 128).

[Dom09] F. Domes. "GloptLab - A configurable framework for the rigorous global solution of quadratic constraint satisfaction problems." In: *Optimization Methods and Software* (4-5 2009), pp. 727 –747 (cit. on p. 5).

[dS97] L. H. de Figueiredo and J. Stolfi. *Self-Validated Numerical Methods and Applications*. Rio de Janeiro: IMPA, 1997 (cit. on pp. 33–37, 40, 41, 44, 46).

[Duf92] T. Duff. "Interval arithmetic recursive subdivision for implicit functions and constructive solid geometry." In: *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '92. New York, NY, USA: ACM, 1992, pp. 131–138. ISBN: 0-89791-479-1 (cit. on pp. 96, 127).

[Dze12] T. Dzetkulič. *Rigorous Computation with Function Enclosures in Chebyshev Basis*. 2012 (cit. on p. 56).

[Dö+13] T. Dötschel et al. "Thermal behavior of high-temperature fuel cells: reliable parameter identification and interval-based sliding mode control." In: *Soft Computing* (2013), pp. 1–15. ISSN: 1432-7643 (cit. on pp. 168, 169, 171, 175).

[Eble06] I. Eble. "Über Taylormodelle." PhD thesis. Universität Karlsruhe, 2006 (cit. on pp. 44, 46, 159, 164).

[Ede95] H. Edelsbrunner. "Algebraic decomposition of non-convex polyhedra." In: *Proceedings. 36th Annual Symposium on Foundations of Computer Science*. 1995, pp. 248–257 (cit. on pp. 94, 95).

[FL05] A. Frommer and B. Lang. "Existence Tests for Solutions of Nonlinear Equations Using Borsuk's Theorem." In: *SIAM Journal on Numerical Analysis* 43.3 (2005), pp. 1348–1361 (cit. on p. 69).

[FLS04] A. Frommer, B. Lang, and M. Schnurr. "A Comparison of the Moore and Miranda Existence Tests." In: *Computing* 72.3-4 (2004), pp. 349–354. ISSN: 0010-485X (cit. on p. 69).

[FPC10] O. Fryazinov, A. Pasko, and P. Comninos. "Fast reliable interrogation of procedurally defined implicit surfaces using extended revised affine arithmetic." In: *Computers & Graphics* 34.6 (2010), pp. 708 –718. ISSN: 0097-8493 (cit. on pp. 60, 96).

[Gam+95] E. Gamma et al. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995 (cit. on pp. 14, 16, 54, 82, 144).

[GCH] O. Gay, D. Coeurjolly, and N.J. Hurst. *libaffa*. Online accessed on 15.11.2010 http://www.nongnu.org/libaffa/ (cit. on pp. 37, 40).

[GJK88] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. "A fast procedure for computing the distance between complex objects in three-dimensional space." In: *IEEE Journal of Robotics and Automation* 4.2 (Apr. 1988), pp. 193 –203. ISSN: 0882-4967 (cit. on p. 116).

[GK] J. Guzman and H. Kaiser. *Boost.Spirit Homepage*. Online accessed on 28.02.2013 http://http://boost-spirit.com/ (cit. on p. 87).

[GL] P. Gottschling and A. Lumsdaine. *The Matrix Template Library 4*. Online accessed on 18.06.2012 http://www.osl.iu.edu/research/mtl/mtl4/ (cit. on pp. 58, 89, 144).

[Gra] *GraphML Specification*. Online accessed on 10.06.2013 http://graphml.graphdrawing.org/specification.html. 2003 (cit. on p. 155).

[Gri00] A Griewank. *Evaluating Derivatives - Principles and Techniques of Algorithmic Differentiation*. SIAM, 2000 (cit. on pp. 64, 65).

[Gri08] C. Grimm. "Result Verification of RRT-based Single-Query Path Planning through Interval Analysis." In: *PAMM* 8.1 (2008), pp. 10971–10972. ISSN: 1617-7061 (cit. on p. 152).

[Gö13] D. Göttner. "Entwurf und Implementierung einer GPU-basierten Bibliothek für eine erweiterte Intervallarithmetik zur Visualisierung impliziter Objekte mit realistischen Beleuchtungsmodellen." Ongoing. MA thesis. Universität Duisburg-Essen, 2013 (cit. on p. 60).

[Ham+95] R. Hammer et al. *C++ Toolbox for Verified Computing*. SpringerVerlag, 1995 (cit. on pp. 71, 73).

[Ham+97] R. Hammer et al. *C++ Toolbox for Verified Scientific Computing I: Basic Numerical Problems*. Secaucus, NJ, USA: Springer, 1997. ISBN: 3540591109 (cit. on p. 125).

[Han] Institute of Microelectronic Systems in Hannover. *aaflib*. Online accessed on http://sourceforge.net/projects/aaflib/ (cit. on p. 40).

[Han75] E. Hansen. "A generalized interval arithmetic." In: *Interval Mathematics*. Ed. by Karl Nickel. Vol. 29. Lecture Notes in Computer Science. Springer, 1975, pp. 7–18 (cit. on p. 1).

[Hay10] N. T. Hayes. *Trits to Tretris*. P1788, Motion 18. 2010 (cit. on p. 31).

[Hij+10] Y. Hijazi et al. "CSG Operations of Arbitrary Primitives with Interval Arithmetic and Real-Time Ray Casting." In: *Scientific Visualization: Advanced Concepts*. Ed. by H. Hagen. Vol. 1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2010, pp. 78–89. ISBN: 978-3-939897-19-4 (cit. on p. 60).

[HK04] W. Hofschuster and W. Krämer. "C-XSC 2.0 – A C++ Library for Extended Scientific Computing." In: *Numerical Software with Result Verification*. Ed. by R. Alt et al. Vol. 2991. Lecture Notes in Computer Science. Springer, 2004, pp. 259–276 (cit. on pp. 28, 31, 159, 167).

[HNW13] O. Heimlich, M. Nehmeier, and J. Wolff von Gudenberg. "Variants of the general interval power function." In: *Soft Computing* 17.8 (2013), pp. 1357–1366. ISSN: 1432-7643 (cit. on pp. 29, 184).

[HS08] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier, 2008 (cit. on p. 81).

[HW04] E. Hansen and G. W. Walster. *Global Optimization Using Interval Analysis*. New York: Marcel Dekker, 2004 (cit. on pp. 2, 4, 67–70, 72–75, 105, 112, 117, 118, 126–128, 131–133, 135, 136, 151).

[Hwu11] "GPU Computing Gems Jade Edition." In: ed. by W. W. Hwu. Morgan Kaufmann, 2011. Chap. Interval Arithmetic in CUDA (cit. on p. 60).

[Iee] *IEEE Standard for Floating-Point Arithmetic - IEEE Std 754-2008 (Revision of IEEE Std 754-1985)*. 2008 (cit. on p. 24).

[IF79] K. Ichida and Y. Fujii. "An interval arithmetic method for global optimization." In: *Computing* 23.1 (1979), pp. 85–97. ISSN: 0010-485X (cit. on pp. 127, 131).

[Isoa] *Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model ISO/IEC 7498-1:1994(E)*. Geneva, Switzerland, 1996 (cit. on p. 15).

[Isob] *Information technology - Vocabulary ISO/IEC 2382-1:1993*. Geneva, Switzerland, 1993 (cit. on p. 13).

[Jau+01] L. Jaulin et al. *Applied Interval Analysis*. Springer, 2001 (cit. on pp. 28, 68, 69, 99, 102, 103, 113).

[Jau01] L. Jaulin. "Path Planning Using Intervals and Graphs." In: *Reliable Computing* 7.1 (2001), pp. 1–15 (cit. on pp. 2, 152, 184).

[JLS00] A. Jaklič, A. Leonardis, and F. Solina. *Segmentation and recovery of superquadrics*. Vol. 20. Springer, 2000 (cit. on pp. 96, 116, 165).

[Jol11] M. Joldes. "Approximations polynomiales rigoureuses et applications." Online accessed on 12.12.2012, `http://perso.ens-lyon.fr/mioara.joldes/these/theseJoldes.pdf`. PhD thesis. École Normale Supérieure de Lyon, 2011 (cit. on p. 56).

[KA00] R. B. Kearfott and A. Arazyan. "Taylor Series Models in Deterministic Global Optimization." In: *Proceedings of Automatic Differentiation 2000: From Simulation to Optimization*. 2000 (cit. on p. 143).

[Kah05] J. H. T. Kahou. "Some new Acceleration Mechanisms in Verified Global Optimization." PhD thesis. Bergische Universität Wuppertal, 2005 (cit. on pp. 111, 124, 131).

[Kea+02] R. B. Kearfott et al. *Standardized notation in interval analysis*. Online accessed on 06.11.2012 `http://radon.mat.univie.ac.at/~neum/ms/notation.pdf`. 2002 (cit. on p. xviii).

[Kea+04] R. B. Kearfott et al. "Libraries, Tools, and Interactive Systems for Verified Computations: Four Case Studies." In: *Numerical Software with Result Verification*. Ed. by R. Alt et al. Vol. 2991. LNCS. Springer, 2004, pp. 36–63 (cit. on p. 145).

[Kea03] R. B. Kearfott. "GlobSol: History, Composition, and Advice on Use." In: *Global Optimization and Constraint Satisfaction*. Ed. by C. Bliek, C. Jermann, and A. Neumaier. Vol. 2861. Lecture Notes in Computer Science. Springer, 2003, pp. 17–31. ISBN: 978-3-540-20463-3 (cit. on pp. 127, 142, 152, 184).

[Kea96] R. B. Kearfott. *Rigorous Global Search: Continuous Problems*. Kluwer Academic Publishers, 1996 (cit. on pp. xvii, 73, 107, 112, 117, 129–131).

[Kie+11] M. Kieffer et al. "Verified Global Optimization for Estimating the Parameters of Nonlinear Models." In: ed. by A. Rauh and E. Auer. Springer, 2011 (cit. on pp. 98, 170, 171, 178).

[KK13] G. Kozikowski and B. J. Kubica. "Interval Arithmetic and Automatic Differentiation on GPU Using OpenCL." In: *Applied Parallel and Scientific Computing*. Ed. by P. Manninen and P. Öster. Vol. 7782. Lecture Notes in Computer Science. Springer, 2013, pp. 489–503. ISBN: 978-3-642-36802-8 (cit. on pp. 129, 175).

[Kno+09] A. Knoll et al. "Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic." In: *Computer Graphics Forum* 28.1 (2009), pp. 26–40. ISSN: 1467-8659 (cit. on pp. 33, 60, 157, 183).

[Knu84] D. E. Knuth. "Literate Programming." In: *The Computer Journal* 27.2 (1984), pp. 97–111 (cit. on p. 149).

[Knu97] D. E. Knuth. *The Art of Computer Programming - Fundamental Algorithms*. Vol. 1. Addison Wesly, 1997 (cit. on p. 100).

[Knü94] O. Knüppel. "PROFIL/BIAS—a fast interval library." In: *Computing* 53.3 (1994), pp. 277–287 (cit. on pp. 31, 146).

[Koc+09] S. Kockara et al. "Contact Detection Algorithms." In: *Journal of Computers* 4.10 (2009), p. 1053. ISSN: 1796-203X (cit. on p. 116).

[Kol07] V. L. Kolev. "Optimal Multiplication of G-intervals." In: *Reliable Computing* 13 (5 2007), pp. 399–408. ISSN: 1385-3139 (cit. on p. 35).

[KR77] B.W. Kernighan and D.M. Ritchie. *The M4 macro processor*. Tech. rep. 1977 (cit. on p. 55).

[Kul09] U. Kulisch. *Arithmetic operations for floating-point intervals*. Motion 05.2 of IEEE P1788, Online accessed on 28.11.2012 http://grouper.ieee.org/groups/1788/PositionPapers/ArithOp2.pdf. 2009 (cit. on p. 71).

[KW02] R. B. Kearfott and G. W. Walster. "Symbolic Preconditioning with Taylor Models: Some Examples." In: *Reliable Computing* 8.6 (2002), pp. 453–468. ISSN: 1385-3139 (cit. on p. 143).

[LaV06] M. S. LaValle. *Planning Algorithms*. Online accessed on 06.06.2013 http://planning.cs.uiuc.edu/bookbig.pdf. Cambridge University Press, 2006 (cit. on p. 152).

[Ler+06] M. Lerch et al. "FILIB++, a fast interval library supporting containment computations." In: *ACM Transactions on Mathematical Software (TOMS)* 32.2 (2006), pp. 299–324 (cit. on pp. 28, 31).

[LG98] M. Lin and S. Gottschalk. "Collision Detection between Geometric Models: A Survey." In: *Proceedings of IMA Conference on Mathematics of Surfaces*. 1998 (cit. on p. 116).

[LHB02] A. Lemke, L. Hedrich, and E. Barke. "Analog circuit sizing based on formal methods using affine arithmetic." In: *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*. ACM. 2002, pp. 486–489 (cit. on p. 33).

[Lip96] S. B. Lippman. *Inside the C++ Object Model*. Addison Wesley, 1996 (cit. on p. 54).

[Loh01] R. J. Lohner. "On the Ubiquity of the Wrapping Effect in the Computation of Error Bounds." In: *Perspectives on Enclosure Methods*. Ed. by U. Kulisch, R. Lohner, and A. Facius. Springer, 2001, pp. 201–217 (cit. on pp. 32, 33).

[LS02] C. Lennerz and E. Schomer. "Efficient distance computation for quadratic curves and surfaces." In: *Geometric Modeling and Processing, 2002. Proceedings*. 2002, pp. 60 – 69 (cit. on p. 116).

[Mak98] K. Makino. "Rigorous Analysis of Nonlinear Motion in Particle Accelerators." PhD thesis. Michigan State University, 1998 (cit. on p. 45).

[Mar+06] M. C. Markót et al. "New interval methods for constrained global optimization." In: *Mathematical Programming* 106.2 (2006), pp. 287–318. ISSN: 0025-5610 (cit. on p. 111).

[MB03] K. Makino and M. Berz. "Taylor models and other validated functional inclusion methods." In: *International Journal of Pure and Applied Mathematics* 4.4 (2003), pp. 379–456 (cit. on p. 44).

[MB05a] K. Makino and M. Berz. "Verified Global Optimization with Taylor Model based Range Bounders." In: *Transactions on Computers* 4.11 (2005) (cit. on p. 128).

[MB05b] K. Makino and M. Berz. "Verified global optimization with Taylor model-based range bounders." In: *Transactions on Computers* 11.1 (2005), pp. 1611–1618 (cit. on p. 46).

[MCC00a] M. S. Markót, T. Csendes, and A. E. Csallner. "Multisection in Interval Branch-and-Bound Methods for Global Optimization II. Numerical Tests." In: *Journal of Global Optimization* 16.3 (2000), pp. 219–228. ISSN: 0925-5001 (cit. on p. 112).

[MCC00b] M. S. Markót, T. Csendes, and A. E. Csallner. "Multisection in Interval Branch-and-Bound Methods for Global Optimization – I. Theoretical Results." In: *Journal of Global Optimization* 16.4 (2000), pp. 371–392. ISSN: 0925-5001 (cit. on p. 112).

[Meso2] F. Messine. "Extentions of Affine Arithmetic: Application to Unconstrained Global Optimization." In: *Journal of Universal Computer Science* 8.11 (2002), pp. 992–1015 (cit. on pp. 33, 35).

[MHo3] J. C. Mason and D. C. Handscomb. *Chebyshev Polynomials*. CRC Press, 2003 (cit. on pp. 36, 38).

[Mir98] B. Mirtich. "V-Clip: Fast and robust polyhedral collision detection." In: *ACM Trans. Graph.* 17 (3 1998), pp. 177–208. ISSN: 0730-0301 (cit. on p. 116).

[MKCo9] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. SIAM, 2009 (cit. on p. 72).

[Moo66] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966 (cit. on pp. 1, 25).

[Moo77] R. Moore. "A Test for Existence of Solutions to Nonlinear Systems." In: *SIAM Journal on Numerical Analysis* 14.4 (1977), pp. 611–615 (cit. on p. 69).

[Mpi] *MPI: A Message-Passing Interface Standard Version 3.0*. 2012 (cit. on p. 129).

[MTo6] F. Messine and A. Touhami. "A General Reliable Quadratic Form: An Extension of Affine Arithmetic." In: *Reliable Computing* 12 (3 2006), pp. 171–192. ISSN: 1385-3139 (cit. on pp. 33, 35).

[Mul+10] J.-M. Muller et al. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010 (cit. on pp. 24, 25).

[Mulo6] J.-M. Muller. *Elementary Functions Algorithms and Implementations*. Second edition. Birkhäser, 2006 (cit. on p. 36).

[Mye95] N. C. Myers. "Traits: a new and useful template technique." In: *C++ Report* (1995). `http://www.cantrip.org/traits.html` Last access on 19.06.2012 (cit. on p. 41).

[Nau12] U. Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. SIAM, 2012 (cit. on p. 64).

[Nedo6] N.S. Nedialkov. *VNODE-LP A Validated Solver for Initial Value Problems in Ordinary Differential Equations*. Tech. rep. CAS-06-06-NN. McMaster University, 2006 (cit. on pp. 7, 21, 86, 91, 97, 149, 170).

[Neuo3] A. Neumaier. "Taylor forms use and limits." In: *Reliable Computing* 9.1 (2003), pp. 43–79 (cit. on pp. 2, 33, 44).

[Neuo4] A. Neumaier. "Complete search in continuous global optimization and constraint satisfaction." In: *Acta Numerica* 13 (Apr. 2004), pp. 271–369. ISSN: 1474-0508 (cit. on p. 184).

[Neu90] A. Neumaier. *Interval methods for systems of equations*. Cambridge University Press, 1990 (cit. on pp. 1, 67–69, 73).

[Neu93] A. Neumaier. "The wrapping effect, ellipsoid arithmetic, stability and confidence regions." In: *Computing (Suppl.)* 9 (1993), pp. 175–190 (cit. on p. 1).

[NM11] J. Ninin and F. Messine. *Reliable Affine Arithmetic*. International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN 2010). 2011 (cit. on p. 35).

[NM65] J. A. Nelder and R. Mead. "A Simplex Method for Function Minimization." In: *The Computer Journal* 7.4 (1965), pp. 308–313 (cit. on p. 170).

[NMH10] J. Ninin, F. Messine, and P. Hansen. *A Reliable Affine Relaxation Method for Global Optimization*. Tech. rep. Online accssed on 03.06.2013 ftp://ftp.irit.fr/IRIT/APO/RT-APO-10-05.pdf. IRIT and Cahiers du GERAD, 2010 (cit. on pp. 33, 143, 152).

[NVI12] NVIDIA. *NVIDIA CUDA C Programming Guide 4.2*. 2012 (cit. on pp. 59, 60, 90, 172).

[NZ14] D. Y. Nadezhin and I. Zhilin S. "JInterval Library: Principles, Development, and Perspectives." In: *Reliable Computing* 19.3 (2014), pp. 229–247 (cit. on p. 30).

[Off] *Object File Format* (cit. on p. 154).

[Ope] *The OpenCL Specification - Version 1.2 - Rev. 15*. 2011 (cit. on p. 59).

[P17] "P1788/D8.4: Draft Standard for Interval Arithmetic" (cit. on pp. 29–31, 49).

[Pro] *PROREOP Website*. http://www.uni-due.de/proreop/, Last accessed: 15.02.2013 (cit. on pp. 18, 164).

[PSD09] R. F. Portal, L. G. Sousa, and J. P. Dias. "Contact detection of convex superquadrics using optimization techniques with graphical user interface." In: *Proceedings of the 7th EUROMECH Solid Mechanics Conference*. 2009 (cit. on p. 116).

[Pus13] T. Pusch. "Umsetzung einer verifizierten Parameteridentifikation mit GUI Realisierung für Festoxidbrennstoffzellen." Ongoing. MA thesis. Universität Duisburg-Essen, 2013 (cit. on pp. 21, 143, 168, 178, 184).

[RA11] A. Rauh and E. Auer. "Verified Simulation of ODEs and DAEs in ValEncIA-IVP." In: *Reliable Computing* 5.4 (2011), pp. 370–381 (cit. on pp. 21, 86, 97, 145).

[Ral81] L. B. Rall, ed. *Automatic Differentiation: Techniques and Applications*. Springer, 1981 (cit. on p. 64).

[Rat92] D. Ratz. "Automatische Ergebnisverifikation bei globalen Optimierungsproblemen." Online accessed on 12.04.2013 http://digbib.ubka.uni-karlsruhe.de/volltexte/41092. PhD thesis. Universität Karlsruhe (TH), 1992 (cit. on pp. 111, 112, 131).

[Rau+12] A. Rauh et al. "Interval Methods for Control-Oriented Modeling of the Thermal Behavior of High-Temperature Fuel Cell Stacks." In: *In Proc. of SysID 2012*. Accepted. 2012 (cit. on pp. 170, 172–177).

[RC95] D. Ratz and T. Csendes. "On the selection of subdivision directions in interval branch-and-bound methods for global optimization." In: *Journal of Global Optimization* 7.2 (1995), pp. 183–207 (cit. on pp. 111, 112).

[RMB05] N. Revol, K. Makino, and M. Berz. "Taylor models and floating-point arithmetic: proof that arithmetic operations are validated in COSY." In: *Journal of Logic and Algebraic Programming* 64.1 (2005), pp. 135–154 (cit. on p. 46).

[Rot12] W. Rotzsche. "Mengentheoretische Operationen mit R-Funktionen auf CSG-Bäumen unter Nutzung von Intervallbibliotheken." MA thesis. Universität Duisburg-Essen, 2012 (cit. on p. 96).

[RR88] H. Ratschek and J. Rokne. *New Computer Methods for Global Optimization*. Online version, access on 13.05.2013 http://pages.cpsc.ucalgary.ca/~rokne/global_book.pdf. Ellis Horwood Ltd, 1988 (cit. on p. 127).

[Rum10] S. M. Rump. "Verification methods: Rigorous results using floating-point arithmetic." In: *Acta Numerica* 19 (2010), pp. 287–449 (cit. on pp. 11, 25).

[Sam06] H Samet. *Foundations of Multidimensional and Metric Data Structures*. San Francisco: Morgan Kaufmann, 2006 (cit. on p. 99).

[Sam90] H. Samet. *The Design and analysis of spatial data-structures*. Addison-Wesly Publishling, 1990 (cit. on p. 99).

[SB] O. Stauning and C. Bendtsen. *fadbad++ web page*. Online accessed on 18.01.2010 http://www.fadbad.com/ (cit. on pp. 88, 91, 146, 159).

[Sch04] H. Schichl. "Global Optimization in the COCONUT Project." In: *Numerical Software with Result Verification*. Ed. by R. Alt et al. Vol. 2991. Lecture Notes in Computer Science. Springer, 2004, pp. 243–249. ISBN: 978-3-540-21260-7 (cit. on pp. 142, 145).

[Sch11] A. Schamberger. "Verifizierte Pfadplanung unter Verwendung einer einheitlichen Objektrepräsentation mit Visualisierung." MA thesis. Universität Duisburg-Essen, 2011 (cit. on p. 152).

[SG75] L. Shampine and M. Gordon, eds. *Computer Solution of Ordinary Differential Equations: The Initial Value Problem*. Freeman, 1975 (cit. on p. 150).

[Sha07] V. Shapiro. "Semi-analytic geometry with R-functions." In: *ACTA numerica* 16 (2007) (cit. on pp. 96, 118).

[Sha91] V. Shapiro. *Theory of R-functions and applications: A primer*. 1991 (cit. on pp. 17, 96, 118).

[Ske74] S. Skelboe. "Computation of rational interval functions." In: *BIT Numerical Mathematics* 14.1 (1974), pp. 87–95. ISSN: 0006-3835 (cit. on p. 127).

[SLL02] J. G. Siek, L. Lee, and A. Lumsdaine. *The Boost Graph Library. User Guide and Reference Manual*. Addison-Wesley Longman, 2002 (cit. on p. 155).

[Sny+93] J. M. Snyder et al. "Interval methods for multi-point collisions between time-dependent curved surfaces." In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '93. ACM, 1993, pp. 321–334. ISBN: 0-89791-601-8 (cit. on p. 125).

[Sta95] V. Stahl. "Interval Methods for Bounding the Range of Polynomials and Solving Systems of Nonlinear Equations." PhD thesis. Johannes Kepler Universität Linz, 1995 (cit. on p. 69).

[Sta97] O. Stauning. "Automatic Validation of Numerical Solutions." PhD thesis. Technical University of Denmark, 1997 (cit. on pp. 58, 84).

[Sto] J. Stolfi. *libaa*. Online accessed on 08.06.2011 http://www.ic.unicamp.br/~stolfi/ (cit. on pp. 23, 35, 37, 40).

[Swi] *SWIG web page*. Online accessed on 21.06.2013 http://www.swig.org (cit. on p. 162).

[Ueb95] C. Ueberhuber. *Computernumerik 2*. Springer, 1995 (cit. on p. 70).

[UY07] A. Uteshev and M. Yashina. "Distance Computation from an Ellipsoid to a Linear or a Quadric Surface in R^n." In: *Computer Algebra in Scientific Computing*. Ed. by V. Ganzha, E. Mayr, and E. Vorozhtsov. Vol. 4770. Lecture Notes in Computer Science. Springer, 2007, pp. 392–401 (cit. on p. 116).

[VH12] Kreinovich V. and N. Hayes. *midprad*. P1788 Motion 37. 2012 (cit. on p. 30).

[VSHF09] X.-H. Vu, D. Sam-Haroud, and B. Faltings. "Enhancing numerical constraint propagation using multiple inclusion representations." In: *Annals of Mathematics and Artificial Intelligence* 55 (3 2009), pp. 295–354. ISSN: 1012-2443 (cit. on pp. 5, 21, 23, 24, 47, 48, 61).

[WB06] A. Wächter and L. T. Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming." In: *Math. Program.* 106.1 (2006), pp. 25–57. ISSN: 0025-5610 (cit. on pp. 7, 126, 150, 168).

[WFF11] N. Whitehead and A. Fit-Florea. *Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs*. Tech. rep. NVIDIA, 2011 (cit. on pp. 13, 60).

[Wie97] A. Wiethoff. "Verifizierte globale Optimierung auf Parallelrechnern." Online accessed on 15.05.2013 http://digbib.ubka.uni-karlsruhe.de/volltexte/118497. PhD thesis. Universität Karlsruhe, 1997 (cit. on p. 131).

[WK09] J. Wolff von Gudenberg and V. Kreinovich. *Elementary Functions*. IEEE P1788 – Motion 10.02. 2009 (cit. on p. 29).

[Yap97] C.-K. Yap. "Towards exact geometric computation." In: *Computational Geometry* 7.1-2 (1997), pp. 3 –23. ISSN: 0925-7721 (cit. on p. 4).

COLOPHON

This document was typeset using TEX and KOMA-Script. It uses the typographical look-and-feel `classicthesis` developed by André Miede. The figures were created using Ti*k*Z and PGFPLOTS.