Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi

Tools for Reliable Geometry

by

Federico Sichetti

Theses Series **DIBRIS-TH-2026-XX**

DIBRIS, Università di Genova
Via Opera Pia, 13 16145 Genova, Italy http://www.dibris.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica, Bioingegneria,

Robotica ed Ingegneria dei Sistemi

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum

Tools for Reliable Geometry

by

Federico Sichetti

February, 2026

Dottorato di Ricerca in Informatica ed Ingegneria dei Sistemi
Indirizzo Informatica
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università degli Studi di Genova

DIBRIS, Univ. di Genova
Via Opera Pia, 13
I-16145 Genova, Italy
http://www.dibris.unige.it/

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum
(S.S.D. INF/01)

Submitted by Federico Sichetti
DIBRIS, Università di Genova
Date of submission: February 2026

Title: Tools for Reliable Geometry

Advisor: Prof. Enrico Puppo
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università di Genova, Italy

Co-advisor: Prof. Daniele Panozzo
Courant Institute School of Mathematics, Computing, and Data Science
New York University, USA

External Reviewers:

Prof. Dr. Marcel Campen
Computer Science Department
Paderborn University, Germany

Dr. Gianmarco Cherchi
Dipartimento di Matematica e Informatica
Università di Cagliari, Italy

Abstract

Reliability in geometry processing and physical simulation is a critical requirement for scientific research, manufacturing, and safety-critical engineering applications. Typical implementations of geometric queries rely on floating-point arithmetic and discrete sampling of quantities, which can lead to undetected geometric degeneracies, non-physical states, and solver failures. These problems are even more severe when working with high-order curved geometry, which is often preferred to piecewise linear representations thanks to its better fidelity. This thesis presents a comprehensive stack of tools designed to ensure provably conservative results for geometric computations without sacrificing computational efficiency.

The first contribution is an algorithm for continuously checking and enforcing the geometrical validity of high-order finite elements as they deform over time. This method employs a robust branch-and-bound approach to approximate the earliest time of element inversion. When integrated into an elastodynamic simulation framework, the method effectively prevents non-physical states and ensures the stability of the solver under extreme deformations.

The second contribution, MiSo (Minimize+Solve), is a domain-specific language (DSL) and compiler that automates the generation of robust, interval-based solvers for a wide class of geometric queries, including collision detection and primitive intersections. MiSo abstracts the mathematical definition of a query from its numerical implementation, using a hybrid approach that combines Natural Interval Extensions with Bézier-based inclusion functions to achieve high performance. The generated solvers provide strict guarantees of correctness and are competitive with, or faster than, hand-optimized implementations for several geometric queries.

The final contribution is TIGHT, a C++ library for fast and correctly rounded interval arithmetic. TIGHT provides the foundational numerical primitives required for reliable geometric computation, ensuring bit-by-bit reproducibility and supporting transcendental functions with minimal interval width. Together, these tools facilitate the development of simulation pipelines where geometric reliability is a hard constraint rather than a heuristic compromise.

Acknowledgements

I would like to thank my supervisor, Prof. Enrico Puppo, for his invaluable guidance and sincere commitment to my success. By encouraging me to explore problems and develop my own solutions, he has undoubtedly made me a better researcher.

I am indebted to Prof. Daniele Panozzo for believing in my capabilities and for all the opportunities he has given me. I look forward to continuing our work together at NYU.

This work would not have been possible without my collaborators, Dr. Zizhou Huang, Dr. Marco Attene, and Prof. Denis Zorin, from whom I have learned a great deal. My gratitude also extends to my reviewers, Prof. Dr. Marcel Campen and Dr. Gianmarco Cherchi, for their careful reading of my thesis and their valuable feedback.

Finally, to the people I love most, my parents Sandra and Nicola, my sister Francesca and my girlfriend Silvia: thank you for always being in my corner.

Wer es in kleinen Dingen mit der Wahrheit nicht ernst nimmt, dem kann man auch in großen Dingen nicht vertrauen.

Whoever is careless with the truth in small matters cannot be trusted with important matters. (A. Einstein)

Contents

Chapter 1 Introduction and Motivation **6**

1.1 Physical simulations 6

1.1.1 The Finite Element Method and volumetric meshes 7

1.1.2 Robustness challenges in FEM 8

1.2 Writing correct programs 10

1.2.1 Numerical error 10

1.2.2 Reproducibility 11

1.2.3 Generalization and automation 11

1.3 Outline of the thesis 12

1.4 List of publications and software 12

1.4.1 Publications 12

1.4.2 Open-source software 13

Chapter 2 Related Work **14**

2.1 High-order geometry and geometrical validity 14

2.1.1 Statement of the element validity problem 16

2.1.2 Static element inversion check 17

2.1.3 Continuous element inversion check 19

2.2 Robust solution of nonlinear systems and minimization problems 19

2.2.1 Explicit root finding 19

2.2.2 Sum-of-squares polynomials . . . 20
2.2.3 Inclusion-based methods . . . 21
2.3 Robust numerical computation . . . 22
2.3.1 Rational and arbitrary-precision arithmetic . . . 22
2.3.2 Interval arithmetic . . . 22
2.3.3 Implementing IA . . . 23
2.3.4 Correct rounding . . . 25
2.4 Domain Specific Languages in graphics . . . 25

Chapter 3 High-Order Continuous Geometrical Validity 27
3.1 Overview of the method . . . 28
3.1.1 Practical example . . . 30
3.2 Preliminaries and notations . . . 31
3.2.1 Reference element domains . . . 31
3.2.2 High-order elements . . . 33
3.2.3 Minimum inclusion function . . . 34
3.3 Continuous geometrical validity . . . 35
3.3.1 Continuous validity test . . . 35
3.4 Implementation . . . 38
3.4.1 Inclusion functions for space and time . . . 39
3.4.2 Robust computation . . . 40
3.4.3 Acceleration . . . 40
3.5 Application to simulation . . . 41
3.5.1 Continuous validity in simulation . . . 42
3.5.2 Invalidity-aware quadrature rules . . . 43
3.6 Results . . . 44
3.6.1 Benchmark of filtered queries . . . 46
3.6.2 Comparisons . . . 49

3.6.3 Elastodynamic simulation . . . 53
3.7 Discussion . . . 55

Chapter 4 MiSo: Automatic Generation of Conservative Solvers 57

4.1 Overview . . . 59
4.1.1 Problem statement . . . 59
4.1.2 Discussion of the ε, δ-MINIMIZE problem when $\varepsilon > 0$ . . . 61
4.1.3 Precision thresholds . . . 63
4.1.4 Discontinuous functions . . . 64
4.1.5 How MiSo works . . . 64
4.1.6 Didactic example: line-surface intersection . . . 65
4.2 Algorithms . . . 68
4.2.1 Problem domain and decomposition . . . 69
4.2.2 Pseudocode . . . 69
4.3 Implementation . . . 72
4.3.1 Symbols . . . 74
4.3.2 Expressions . . . 74
4.3.3 Evaluating inclusion functions . . . 76
4.3.4 Domain decomposition . . . 76
4.3.5 The MiSo compiler . . . 77
4.4 Applications . . . 78
4.4.1 Static objects intersections . . . 79
4.4.2 Dynamic objects intersections . . . 81
4.4.3 Collapsing the expression . . . 85
4.4.4 Implicit booleans . . . 86
4.4.5 Minimal distance . . . 87
4.4.6 Geometrical validity check . . . 89
4.4.7 Compilation time . . . 91

4.5 Discussion . . . 91

Chapter 5 A C++ Library for Fast and Correctly Rounded Interval Arithmetic 92

5.1 Implementing elementary functions . . . 93

5.1.1 Interval extension . . . 94

5.2 Results and comparison . . . 102

5.2.1 Benchmark comparison . . . 102

5.2.2 Comparison with Filib++ within MiSo . . . 104

5.2.3 Pathological cases . . . 106

5.3 Discussion . . . 107

Chapter 6 Conclusion 109

6.1 Summary of contributions . . . 109

6.2 Discussion and impact . . . 110

6.3 Limitations . . . 110

6.4 Future work . . . 111

6.4.1 Parallelization and Hardware Acceleration . . . 111

6.4.2 Material models under extreme deformation . . . 111

6.4.3 Expanded scope for MiSo . . . 112

6.4.4 Standardization of interval arithmetic . . . 112

6.4.5 Improved support for CR functions . . . 112

6.5 Closing remarks . . . 112

Appendix A Appendix 113

A.1 Lagrange and Bézier forms of $|J_x|$ . . . 113

A.2 Subdivision rules . . . 116

A.3 Failure case for floating point algorithm . . . 121

A.4 Polyhedral bounding boxes . . . 122

Chapter 1

Introduction and Motivation

Geometry processing is fundamental to a wide range of digital applications, serving as the bridge between abstract mathematical models and their practical realization, and enabling the creation of tools used by millions of people in different fields such as medicine, entertainment, manufacturing, engineering, and science. Over the years, many different solutions have been developed for intuitively simple tasks such as computing the distance to a surface, testing if a point lies inside or outside a bounded region, deforming a 3D model, or ensuring that two models do not intersect. One of the reasons for this diversity is that the requirements for these fundamental algorithms diverge significantly depending on the intended use case.

In the realm of computer graphics and entertainment, the most important aspects of an algorithm are often performance and visual plausibility. Algorithms are designed to generate results that look convincing to the human eye, often sacrificing strict mathematical correctness for the sake of speed and interactivity. In this context, geometric errors are often considered acceptable artifacts as long as they do not disrupt the visual experience. Conversely, in scientific computing and engineering, geometry processing serves as the foundation for rigorous analysis that is necessary for manufacturing, structural analysis, aerospace, robotics, biomechanics, and more. Here, accuracy and reliability are paramount; a single inverted element or a non-watertight mesh can invalidate an entire pipeline or lead to a defective physical part, in ways that are often subtle and difficult to detect. For these applications, correctness is not just a desirable quality, but a hard constraint.

1.1 Physical simulations

The applications of simulating physical phenomena include real-time physics in virtual worlds, complex visual effects in graphics and entertainment, and high-fidelity computations essential

for engineering and scientific research. Recently, simulation methods have also proven to be a viable option to train machine learning models in tasks for which data is scarce or difficult to obtain, with some models being trained entirely on synthetic data.

Simulations are used to test designs safely, at a low cost, and with complete control over the environment; this allows a domain expert to iterate on the design, or even optimize it automatically, without the need for real-world experiments until much later in the pipeline. This paradigm, often referred to as "digital twin", relies heavily on the assumption that the virtual representation behaves consistently with physical laws; if the underlying geometric or numerical kernels fail, the validity of the entire process is compromised.

1.1.1 The Finite Element Method and volumetric meshes

The Finite Element Method (FEM) is a classic way to solve partial differential equations numerically, and many physical solvers rely on this approach. The core idea of FEM is to discretize a continuous domain into a finite number of small subdomains, known as elements. Instead of solving the complex differential equations over the entire domain at once, the method approximates the solution within each element using simple basis functions, typically polynomials. These local contributions are then assembled into a global system of equations that can be numerically solved to approximate the behavior of the entire physical system.

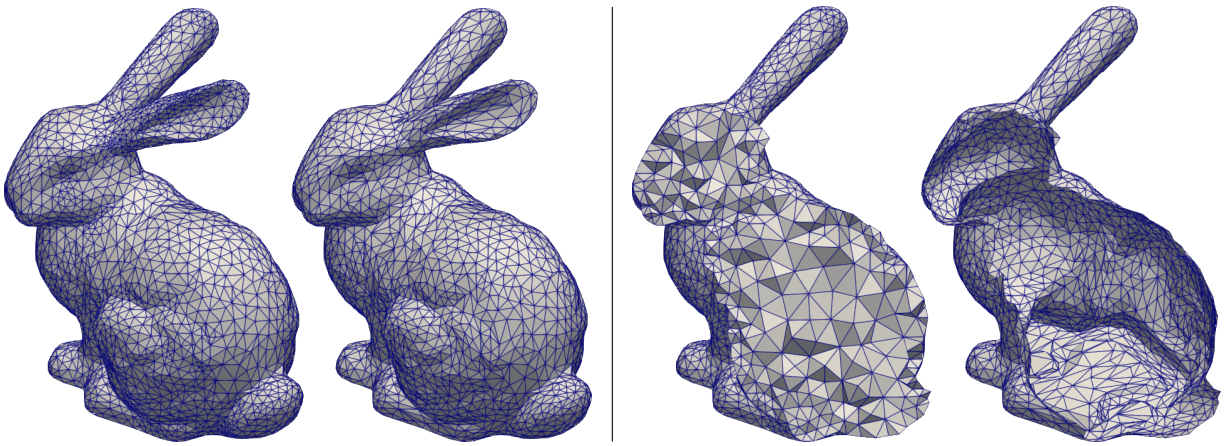


Figure 1.1: A surface mesh and a volumetric mesh. From the outside (top), the two appear almost identical; cross-sections (bottom) reveal that the volumetric mesh fills the interior with tetrahedra, whereas the surface mesh consists only of the triangulated boundary shell.

The first step of a FEM pipeline is creating a volumetric mesh of the simulated objects, that is, a discrete model of the entire space occupied by the object. This differs from the surface meshes typically used in computer graphics, which only represent the object boundary (see Figure 1.1).

Just like a 3D surface mesh can be made of triangles or quadrilaterals, a 3D volume mesh is usually composed of tetrahedra and hexahedra. More exotic element types, such as triangular prisms and pyramids are sometimes used in meshes with elements of mixed type. Some FEM software also supports codimensional objects, for example to model 1D fiber or 2D cloth in 3D space.

Another crucial difference with graphics applications is the use of high-order basis functions to represent the geometry, physical quantities and solution space. Instead of having a piecewise linear geometry, where each element linearly interpolates the positions of its vertices, a high order mesh uses additional control points to describe how the geometry curves in space. Nonlinear bases, especially polynomials, are often preferred in simulations due to the lower element count that they require to approximate an object, and the fact that they can avoid spurious stresses induced by linear approximations (known as the locking phenomenon) [43, 117]. While linear geometry relies on higher resolutions to improve the solution accuracy (h-refinement), using polynomial elements it is possible to reduce error by increasing the order (p-refinement); a sweet spot is usually found using a combination of the two.

High-order elements violate many of the assumptions that hold for linear geometry, and many geometric queries require a fundamentally different approach. Despite being a superior alternative to linear meshes, the additional hurdles one needs to go through when working with curved geometry limits the widespread adoption of high-order geometry representations in physical simulations.

Generating volumetric meshes is a vast subject on its own; while algorithms for generating linear tetrahedral meshes are relatively mature [122, 63], creating hexahedral meshes or high-order curved meshes remains a significant challenge and an active area of research. We refer to Pietroni et al. [114] for an overview of the state of the art for hex-meshing, and Geuzaine et al. [49] and Jiang et al. [67] for an overview of the state of the art for high-order meshing.

1.1.2 Robustness challenges in FEM

Once a mesh is available, the next step is specifying the physical properties of the materials, the forces at play and the constraints; then, a finite element solver is used to obtain results. While simulation algorithms are mathematically proven to approximate the real solution to some accuracy, many of the guarantees that these methods provide can fall apart when implemented naively on a real machine.

First, the input model may have subtly degenerate geometry such as self-intersections and inverted elements. In the context of simulation, these represent configurations that are not physical, and thus may cause the simulated system to behave in ways that are not physical. Even worse, for problems in which objects are moving, a clean input does not mean that degeneracies cannot arise during the deformation, and recovering from an invalid state may not

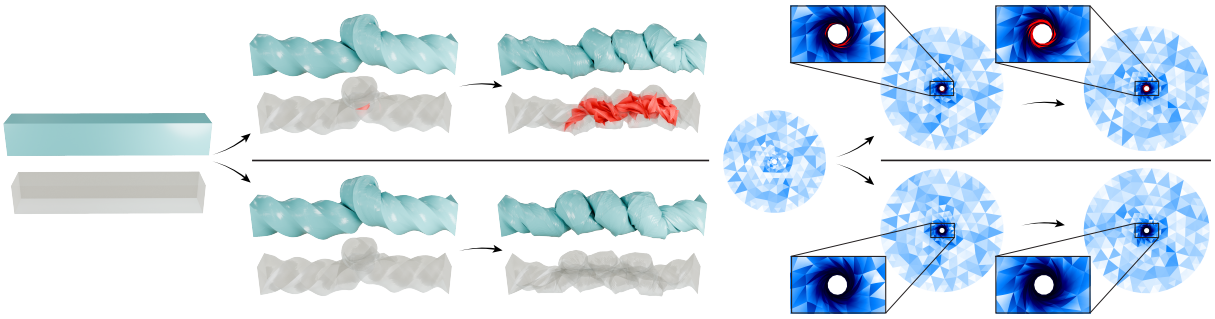


Figure 1.2: 3D (left) and 2D (right) simulations of elastic objects undergoing deformations. In the top row, some elements become inverted during the simulation, causing local self-intersection and violating the assumptions of the Finite Element Method. In the bottom row, our robust check presented in Chapter 3 ensures that the all elements remain valid throughout.

be an option (Figure 1.2). For example, if a simulator does not implement a proper Continuous Collision Detection (CCD) routine, it is very likely that the simulated objects will penetrate each other when they come into contact. Therefore, one needs ways to verify that the state of the discretized system remains valid throughout the entire process (Figure 1.3). This verification consists of purely geometric problems distinct from the physical simulation itself, such as CCD. Another example that is relevant to this thesis comes from elastodynamics, where a geometric validity test is used to ensure that the map from the reference domain to physical space preserves orientation (in other words, preventing degenerate elements for which elastic energy would become infinite).

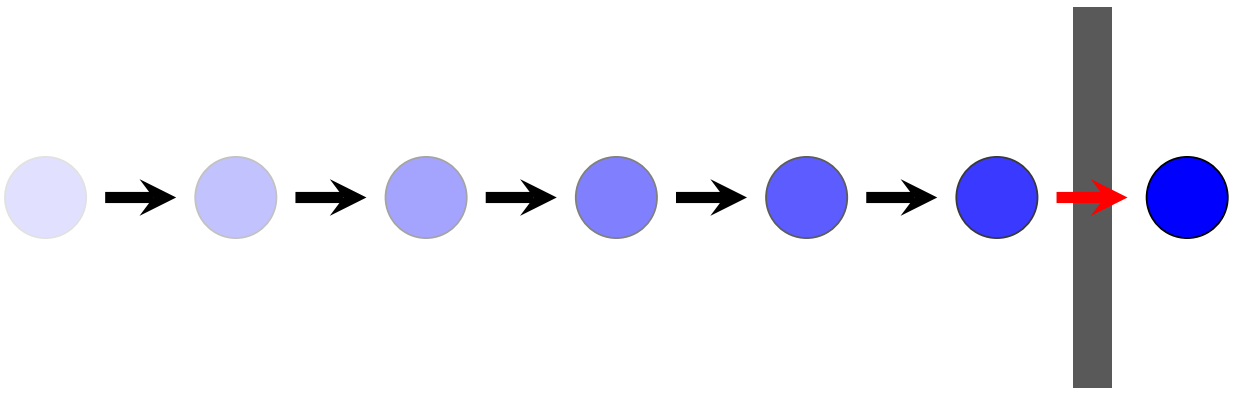


Figure 1.3: A fast-moving object (blue) can tunnel through a thin wall when collision detection is performed only at discrete time steps. Transparent frames show the object's positions at each step; none of them intersects the wall, thus the object is free to cross to the other side.

Developers must often relax the validity constraints on their data in order to get acceptable execution times, for example by implementing relaxed versions of the aforementioned continuous checks that only test a predicate on a large number of samples. While this approach can make the algorithm pass even extensive testing, it does not guarantee that the program will be able to correctly handle all valid data. Considering the increasing application of FEM in safety-critical scenarios such as aerospace and architecture, the sheer size of models that are processed, and the fact that in many applications data often comes from non-technical users, correctly handling pathological cases is more important than ever before: the consequences of an incorrect result from apparently correct input data can range from annoying to debug to outright dangerous.

These compromises may also be attributed to a lack of methods for guaranteed geometric queries on high-order geometry with acceptable speed and accuracy. The goal of this thesis is to contribute to the development of such methods.

1.2 Writing correct programs

1.2.1 Numerical error

Computers implement floating point arithmetic as a proxy for real arithmetic. Therefore every operation can accumulate error, and not all real values can be represented exactly. Double-precision floating point numbers are the predominant representation for real numbers on machines, thanks to hardware support that makes them extremely efficient to work with [50]. The fact that floating point operations introduce approximation error is common knowledge, as is the belief that these errors are too small to influence the results in a meaningful way. However, roundoff error can create inconsistent states in a program. The algorithm may use the result of a boolean predicate to make assumptions, for instance to select a branch of execution; without the guarantee that the predicate will evaluate correctly in the presence of roundoff error, there is no certainty that the program will select the correct path, leading to unexpected failures or nonsensical results on perfectly valid data. The fact that floating point error is indeed negligible in most cases makes this kind of bug difficult to detect and fix.

This work builds on the foundation of interval arithmetic as the core component to obtain algorithms with guarantees of correctness. Unlike standard arithmetic, which approximates a real value with a single floating-point number, interval arithmetic represents numbers as real intervals that contain the true value, and maintains this guaranteed inclusion property through all operations.

Historically, interval methods have been viewed as too slow for interactive computer graphics or large-scale simulation. This perception stems from two factors: the inherent computational

overhead of tracking bounds, and the "dependency problem" where naive interval evaluations yield overly pessimistic bounds, requiring deep subdivision trees to resolve queries [109]. To mitigate these issues, our code generation system MiSo enables users to quickly test different solutions for computing conservative bounds for the same problem, to find the most efficient one. Moreover, it is based on a C++ library for interval arithmetic called TIGHT we developed, which is faster than the state-of-the-art library.

1.2.2 Reproducibility

Reproducibility has been an especially hot topic in the scientific community. Changing a single bit in a floating point number can change the result of a program for ill-conditioned data, so it is important that results can be replicated on different machines even in the presence of approximation error. If a simulation of a chaotic system diverges on two different architectures due to how some function is implemented in the mathematical libraries, verifying the correctness of the results becomes an arduous task. While the IEEE 754 standard [2] gives some consistency guarantees for basic floating point arithmetic operations, this does not extend to standard math library operations such as sines, logarithms, and exponentials, which are ubiquitous in geometric calculations. Even on the same machine, different compilers or math libraries can produce subtly different results. TIGHT guarantees minimal interval width and bit-by-bit reproducibility via correct rounding, and constitues another original contribution of this thesis.

1.2.3 Generalization and automation

Reliability is, of course, also related to human error. Finite element software often requires specializing code for several combinations of element type, order, and dimension. For instance, implementing a continuous collision detection routine for 3D objects involves distinct logic for vertex-face and edge-edge interactions, which further branches based on the type of geometry (linear, quadratic, or higher-order polynomial), and trajectory (linear or screw-motion). Covering all cases by hand while writing guaranteed-correct code can be tedious and error prone.

An alternative way to address this issue is to develop generic, robust algorithms that enable programmatic generation of efficient, safe code. By abstracting the mathematical definition of a geometric query from its numerical implementation, we can design Domain-Specific Languages (DSL) to automate the production of robust solvers for geometric predicates. This approach not only reduces the surface area for human error but also eases development by enabling optimizations to be applied automatically across a large codebase, for a given class of problems. One of the main contributions of this thesis is the development of such a tool for generating robust C++ solvers of geometric queries, called MiSo.

By automating the generation of robust solvers and providing the underlying arithmetic primi-

tives, we aim to facilitate access to reliable geometric queries, with the goal of making physical simulations and geometry processing provably correct without sacrificing efficiency.

1.3 Outline of the thesis

The thesis is structured as follows:

- Chapter 2 is dedicated to a literature review of related work.
- Chapter 3 details the first original contribution of the thesis, an algorithm to continuously test geometrical validity of finite elements as they deform in time during an elastodynamic simulation, and a method to enforce it in the Incremental Potential Contact (IPC) framework using invalidity-informed adaptive quadrature.
- Chapter 4 generalizes the concepts of the previous chapter, and introduces a tool called MiSo for the automatic generation of robust interval-based solvers for a variety of problems, starting from a high-level user specification.
- Chapter 5 presents the final contribution of the thesis: a fast interval arithmetic library with correct rounding called TIGHT, which enables MiSo to reliably handle problems with transcendental functions.
- Finally, Chapter 6 contains a discussion of the obtained results and plans for future work.

1.4 List of publications and software

1.4.1 Publications

Below is a list of publications on which this thesis is based. A chapter is dedicated to each paper.

- *High-Order Continuous Geometrical Validity.* Federico Sichetti, Zizhou Huang, Marco Attene, Denis Zorin, Enrico Puppo, Daniele Panozzo. ACM Transactions on Graphics, 2025. Chapter 3 is based on this paper.
- *MiSo: A DSL for robust and efficient Solve and Minimize problems.* Federico Sichetti, Enrico Puppo, Zizhou Huang, Marco Attene, Denis Zorin, Daniele Panozzo. ACM Transactions on Graphics, 2025. Presented at SIGGRAPH 2025. Chapter 4 is based on this paper.

- *TIGHT Intervals for provably correct geometric computation.* Federico Sichetti, Marco Attene, Enrico Puppo. Proceedings of STAG: Smart Tools and Applications in Graphics 2025. Chapter 5 is based on this paper.

1.4.2 Open-source software

This thesis presents tools to design algorithms that are provably correct when implemented on a machine and thus can be fully trusted, as well as providing efficient open-source implementations that are ready to be integrated in existing software.

The following open-source projects stemmed from this line of work:

- `https://gitlab.com/fsichetti/hocgv`: a reference implementation of the high-order continuous validity test presented in Chapter 3, as well as the dataset used in the paper.
- `https://gitlab.com/fsichetti/miso`: the MiSo C++ code generator (Chapter 4).
- `https://gitlab.com/fsichetti/tight`: the TIGHT interval arithmetic library (Chapter 5).

Beyond standalone releases, the tools are being integrated into existing simulation software. The high-order continuous validity check has been integrated into PolyFEM [116], where it optionally replaces the previous point-wise check to prevent element inversions in elastodynamic simulations. MiSo is currently undergoing integration into PolyFEM as well, to provide conservative CCD for high-order elements. TIGHT is being adopted as the interval arithmetic backend for MiSo, replacing the previous NFG-based implementation.

Chapter 2

Related Work

This chapter introduces the basic notions and related work which the rest of the chapters build upon.

Section 2.1 provides a brief overview of high-order polynomial bases and their use in graphics, methods for checking static and continuous validity of elements, and discusses their application in FEA and meshing. It concludes with an overview of predicate evaluation techniques for the specific problem of geometrical validity. In Section 2.2, we survey more general methods for solving non-linear systems and non-linear minimization problems. Section 2.3 delves deeper into interval arithmetic, which is the basis of this thesis's contributions, and its effective computer implementation. Lastly, we include a small discussion on domain-specific languages (DSL) in Section 2.4, as it is relevant to Chapter 4.

2.1 High-order geometry and geometrical validity

In the context of finite element physical simulation, objects are modeled using meshes composed of simple elements, such as tetrahedra, hexahedra, and prisms. Each element is associated with two maps: (1) a geometric map that defines the element's shape; and (2) a basis map that extends quantities (such as displacement or velocities) defined at the element's nodes into its interior.

For rendering, linear functions (specifically, hat functions) are commonly used for both maps. However, linear basis functions are often a sub-optimal choice in many contexts. For example, Schneider et al. [117] advocate for the use of high-order bases for elliptic PDEs; Bargteil and Cohen [14], Mezger et al. [107], Suwelack et al. [130], Ushakova [135] use high-order elements for animation; and Mandad and Campen [99] propose to use a high-order basis for parametrization. A related, but distinct, concept is the use of high-order geometry, where the geometry of

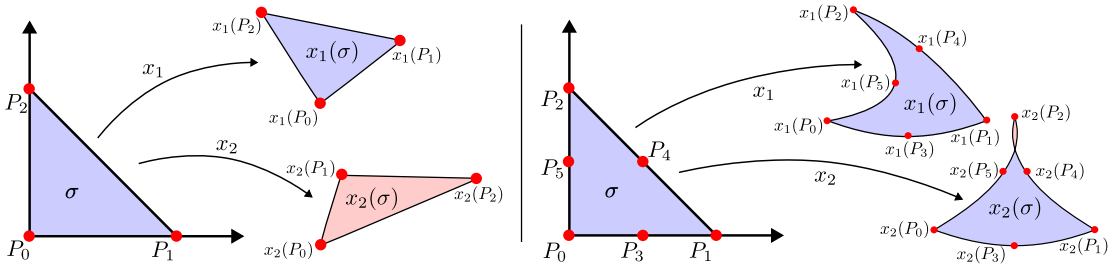


Figure 2.1: A reference domain σ is mapped to linear elements (Left) and quadratic elements (Right). Quadratic elements can be given either by an initial quadratic geometry or by combining an initial linear geometry with a quadratic displacement. Geometric maps on a Lagrangian basis are specified by mapping the domain points P_i to control points $x_j(P_i)$. Blue and red areas denote a positive and negative determinant of the Jacobian, respectively. In both cases, element $x_2(\sigma)$ is invalid. In the linear case, the whole element is inverted, while in the quadratic case, only a small portion of $x_2(\sigma)$ is inverted.

an element is represented using a high-order polynomial. More commonly, C^0 [67] or C^k geometric maps (IGA) [28] are used. The latter option is popular in mechanical engineering, where IGA envisions the use of the same representation for both interpolating the physical quantities and for representing the geometry.

When the basis map is used to interpolate a *displacement*, the element's geometry is derived by combining the initial geometric map with the basis map, yielding a polynomial whose order corresponds to the highest of the two. A typical example in graphics is the use of second-order elements in finite element (FE) simulations for fabrication [113], which generates quadratic curved elements even if the initial geometric map is piecewise linear. In the following, we will overload the term *geometry map* to describe the final geometry of an object, which can thus also incorporate the displacement basis map.

In mathematical notation, this is defined with a polynomial function:

$$x : \sigma \longrightarrow \mathbb{R}^n,$$

where σ represents a reference element domain (such as a standard simplex or unit hypercube), and n denotes the dimension of the element. See Figure 2.1 for examples.

During animation or simulation, objects are deformed by modifying the geometry map, the basis map, or both: these modifications are usually performed by applying linear transformations to the map coefficients. Note that, for a piecewise linear geometric map, this reduces to the usual interpolation of the coordinates of the mesh vertices. Similarly, for curved meshing, it is also typical to start with a piecewise linear mesh and then curve its elements to reduce the approximation error [134].

Since these meshes are used to represent physical objects, a basic, yet elusive, requirement is

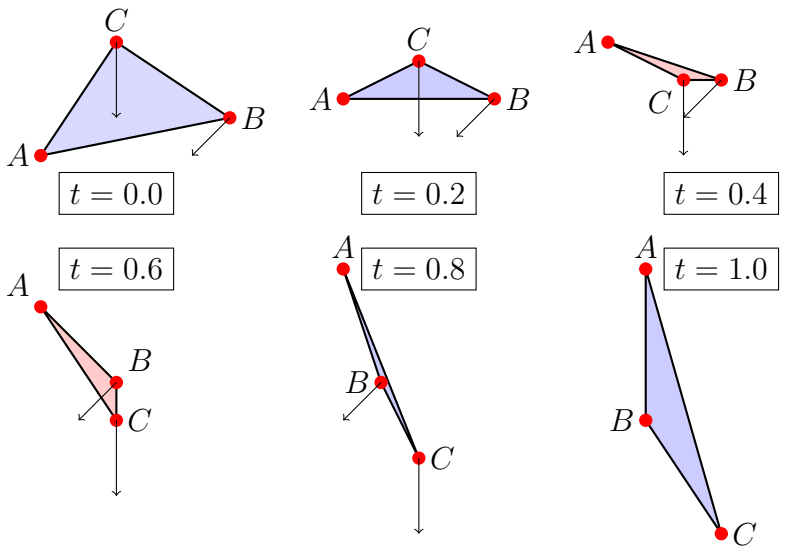


Figure 2.2: A dynamic linear element with linear trajectories that flips twice in a time interval, resulting in an element that is valid at both time steps, but invalid in the transition. Assuming point A remains fixed, the arrows represent the velocities of points B and C to reach the final position. The color of each element indicates the sign of the determinant at time t: blue if positive and red if negative. The dynamic element's Jacobian determinant is a univariate polynomial in t of degree 2, with two distinct roots in $[0, 1]$.

that the geometry of the object does not self-intersect at any time during deformation. This condition can be violated in two ways: (1) by a global intersection or (2) by a local lack of injectivity of the geometric map. The first problem has been studied thoroughly in the parametrization and simulation literature, at least for linear elements [124, 139]. The second condition has been considered mostly in the context of mesh parametrization, and, even in that case, no robust algorithm exists. In Chapter 3, we focus on the latter.

2.1.1 Statement of the element validity problem

Let ξ be the vector of coordinates in the reference element, and t time. Denote by $\overline{x}(\xi, t)$ a geometric map that is dynamically changing over time. Checking its injectivity reduces to studying the sign of the determinant of its Jacobian $|J_{\overline{x}}|$ to assess the continuous validity of the element, i.e., if, where and when $|J_{\overline{x}}|$ becomes negative. We note that this is a subtly challenging problem even for the simple case of a linear triangle (Figure 2.2): an element that is valid in its initial and final configurations might cross invalid states along its trajectory.

A requirement for robust simulations is that they employ *conservative* algorithms: that is, a program that only reports results whose correctness cannot be influenced by approximation

errors. Thus, a Boolean predicate (such as asking whether an element is valid or not) is required to return a third "undecided" value if the approximation error is large enough to make the naive result unreliable. Conversely, if the algorithm returns an answer, that answer must be provably correct. In the context of continuous validity, we wish to find the maximum time step t^* such that the element is provably valid at all times in $[0, t^*)$: a conservative answer can be any value $t \in [0, t^*)$. Obviously, in practice, it is also desirable that t be as close to t^* as possible.

A common approach among practitioners is testing the validity of elements by just computing the value of $|J_x|$ at quadrature points at every time step, as done, e.g., by Dey et al. [36]. This might fail to detect invalidity even in static tests, let alone the continuous case. More sophisticated tests are proposed by Johnen et al. [69] for static high-order elements, and by Smith and Schaefer [124] for the time-dependent problem in the specific case of 2D linear triangles. These are described in more detail in the following sections.

2.1.2 Static element inversion check

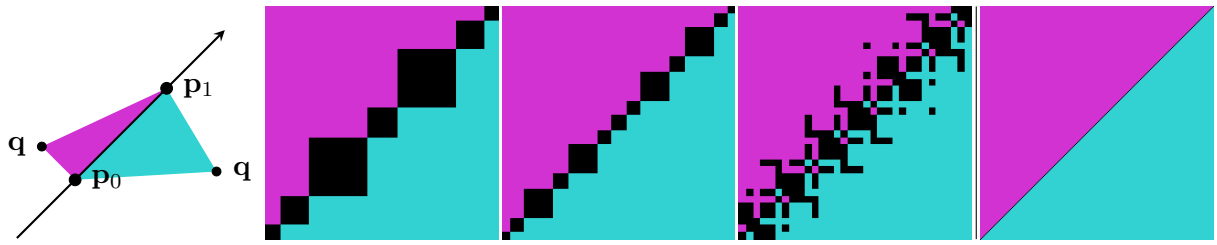


Figure 2.3: The effects of numerical error on the ORIENT2D predicate. Three collinear points $\mathbf{p}_0 = (0.5, 0.5)$, $\mathbf{p}_1 = (12, 12)$, $\mathbf{p}_2 = (24, 24)$ are fixed. A 256×256 grid of query points $\mathbf{q}_i$ is constructed around $\mathbf{p}_0$ by advancing one ULP at a time in each coordinate, probing consecutive floating-point values in a neighbourhood of radius $\sim 10^{-14}$. Each pixel is colored by the sign of the orientation predicate evaluated at $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{q}_i$ (cyan: negative, magenta: positive, black: zero), as shown in the first image. The next three images show the results of the predicate implemented with floating-point arithmetic, using different cyclic permutations of the arguments. Despite being mathematically equivalent, the predicates incorrectly return 0 in some configurations when the arguments are nearly collinear, and the third one even shows the predicate returning the opposite sign for some values. The rightmost shows the correct, adaptive-precision ORIENT2D as a reference. Images generated using the code by [118].

The special case of checking the geometric validity of a linear triangle/tetrahedron in a static setting has been solved in a seminal paper by Shewchuk [120], where robust predicates called ORIENT2D and ORIENT3D are introduced. ORIENT2D$(\mathbf{a}, \mathbf{b}, \mathbf{c})$ returns a positive value if the points $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^2$ are in counter-clockwise order, negative if clockwise, and zero if collinear. ORIENT3D$(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$ returns a positive value if $\mathbf{d} \in \mathbb{R}^3$ lies below the plane through $\mathbf{a}, \mathbf{b}, \mathbf{c}$ (with

counter-clockwise orientation when viewed from above), negative if above, and zero if coplanar. Both predicates are equivalent to the sign of a small determinant of the input coordinates, and checking the sign of a triangle's or tetrahedron's Jacobian determinant reduces to one of these predicates. The challenge is that naïve floating-point evaluation of these determinants is unreliable near zero, where catastrophic cancellation can flip or zero out the result, as illustrated in Figure 2.3. Shewchuk [120] addresses this with adaptive-precision arithmetic: the computation starts with fast floating-point arithmetic and only falls back to exact arithmetic when the result is too close to zero to trust. This paper revolutionized mesh generation and simulation, providing a reliable, yet efficient, solution to one of the basic primitives used by meshing and simulation algorithms. To the best of our knowledge, this approach has not been extended to elements with higher order; such extension is challenging because an element may flip at certain points while remaining valid at others, as in the example shown on the right side of Figure 2.1.

For high-order simplicial elements, a common approach to test for element inversion consists in testing only their quadrature points [116, 98, 46, 36]: while effective at avoiding NaNs in the integration of certain diverging elastic potentials, this approach is not conservative, leading to incorrect stresses (Figure 3.10). Earlier work had already observed the relevance of Bézier coefficient positivity as a validity criterion: Dey et al. [35] and Luo et al. [97] encountered this idea in the context of p-version mesh generation, and Hernandez-Mederos et al. [58] studied local injectivity of triangular cubic Bézier maps in 2D. Building on these observations, an efficient and general method was introduced by Johnen et al. [68, 69, 71], where the Jacobian determinant of the element is represented in Bézier form, and the inversion check reduces to testing the positivity of the Bézier coefficients as they undergo adaptive refinement. Their implementation relies on floating point arithmetic, and therefore it is not conservative and can miss inversions (we provide a numerical example in Section A.3).

For the special case of hexahedral elements, which are commonly used in commercial finite element analysis software, this problem has been extensively studied by Ushakova [135]. Unfortunately, these tests are insufficient to guarantee validity but are used nonetheless due to their efficiency. Vavasis [136] proposes a sufficient condition but does not provide a conservative algorithm that takes advantage of it. Johnen et al. [70] propose an optimized version of their previous method [69] specific to linear hexahedral elements, however, the approach still suffers from the same floating point issues as the original test. George and Borouchaki [48] similarly propose a recursive subdivision-based method for static tensor product elements; we are not aware of a publicly available implementation of this method.

A radically different approach is taken by Marschner et al. [101], where a Sum-of-Squares (SOS) relaxation is used to compute the minimum Jacobian determinant, reducing the problem to solving a sequence of small semidefinite programming problems of increasing complexity. The method offers generality and an elegant formulation, but is relatively expensive computationally (multiple SDP solves per element), and only guarantees injectivity up to numerical precision

of an iterative convex solver, which might result in invalid elements.

All the methods described above are designed to handle static checks exclusively, which is insufficient for validating deforming elements over time.

2.1.3 Continuous element inversion check

To the best of our knowledge, the only paper explicitly addressing the problem of checking the validity of an element that deforms over time is by Smith and Schaefer [124]. They propose an algorithm to estimate the safest step before an inversion for 2D linear elements by using the closed form of the roots of a degree 2 polynomial. Their approach is however not conservative due to floating point rounding, as we show in Section 3.6. Furthermore, the method does not scale to linear 3D elements – where the polynomial is of order 3 and robust root finding is not trivial – nor higher order elements – where the polynomial is multivariate and closed forms for the roots may not even be available.

The problem is also discussed by Anderson et al. [9] for high-order element remapping and by Dobrev et al. [37] for high-order meshing, however no algorithm for the validity check is proposed.

2.2 Robust solution of nonlinear systems and minimization problems

As mentioned in the previous section, the geometrical validity problem can be viewed from a more general point of view as solving a non-linear system of inequalities (in the static case, where one jointly solves for $\xi \in \sigma$ and $|J_x(\xi)| \leq 0$) or globally minimizing a function subject to a non-linear system (in the time-dependent case, where one seeks a minimum of $F(\xi, t) = t$ such that $\xi \in \overline{\sigma}$ and $|J_{\overline{x}}(\xi, t)| \leq 0$). Geometrical validity is not the only family of problems that fits this characterization: other common geometric queries, such as continuous collision detection, surface-surface intersection, minimal distance computation, and Boolean operations can be formulated in this way. While approximate methods are typically preferred due to their efficiency, we instead focus on guaranteed-correct methods.

2.2.1 Explicit root finding

Computer Algebra Systems [141, 108, 105] provide solvers that use algebraic manipulation to find explicit expressions for the roots of non-linear systems. Unfortunately, they are computationally expensive and limited to small systems: as an example, Wang et al. [139] use them

for computing a ground truth result for the problem of CCD and reports a running time in the order of seconds per query.

One case of particular interest is the problem of finding the roots of univariate quadratic or cubic polynomials. In these cases, a closed-form expression for the roots is known, and it has been used for dynamic inversion tests by Smith and Schaefer [124], and for collision detection by Hadap et al. [55]. Evaluating the closed-form expression with floating point arithmetic, while efficient, is unstable and might lead to incorrect roots [139].

A popular option is to use numerical methods for constrained optimization (Minimize) and root finding (Solve) [111]. These methods are very popular in graphics [55] due to their efficiency and ease of control of accuracy. A major limitation of many approaches in this class is that they must be implemented in floating-point arithmetic, thus possibly computing an incorrect result.

While it is common to use "small" numerical tolerances to mitigate this issue [24, 89, 5], this is a heuristic that does not guarantee correctness. Although forward error analysis can derive conservative tolerances [140], these often lead to excessive false positives [139].

Robust interval versions of Newton's method [129, 56] can bound zero loci for continuously differentiable functions, but they cannot handle non-differentiable operators such as abs, max, and min.

Brochu et al. [21] reduce root parity counting to a geometric intersection problem solvable with custom numerical predicates [138]. This approach, however, is limited to determining root parity (even or odd) and requires deriving a specific predicate for each constraint set. Tang et al. [131] proposed a Bernstein sign classification method for CCD. However, conservative implementation is challenging, and [139] presented a counterexample showing its failure to detect a collision.

2.2.2 Sum-of-squares polynomials

Marschner et al. [102] proposed reducing non-linear constraint solving to semidefinite programs, an approach also used for high-order patch collision detection [142] and hexahedral mesh inversion repair [100]. Sum-of-squares programming replaces non-linear polynomial positivity constraints with a more tractable representation consisting of sums of squares of polynomials of bounded degree d. This relaxation turns the original non-convex problem into a sequence of convex semidefinite programs that are guaranteed to recover the global optimum as d increases. Despite the fact that the method can generate guaranteed solutions on paper, it is computationally costly and provides no clear way of controlling the propagation of numerical error.

2.2.3 Inclusion-based methods

A robust and generic solution has been pioneered for geometric modeling applications by Snyder in his PhD thesis [127, 125, 126]. The idea is to adaptively partition the domain using an inclusion function for guidance. An inclusion function for a function f defined on a domain D returns a set enclosing the range of f on D. Inclusion functions can be used to check if the domain contains a root, does not contain a root, or might contain a root: in the latter case, refining and evaluating the inclusions on its subdomains provides an effective algorithm to detect and isolate roots. These approaches are widely used in computational geometry [60, 61], in path planning [143], and for more generally isolating polynomial roots [27].

Snyder [125] proposes using interval arithmetic (IA) to build inclusion functions in a generic and automatic way. However, his construction often results in overly pessimistic inclusions that increase the overall computational cost of the search algorithms. Applications include surface intersection, closest point queries, and other operations needed in a geometric modeling kernel. Interval methods have also found use in robotics and control [104, 66]. The high computational cost is likely why these approaches have not found wide application in graphics until recently: Wang et al. [139] benchmarked this approach against a numerical root finder for collision detection and found it on average ~ 4 orders of magnitude slower.

Ad-hoc inclusion functions can be built without the use of IA [129, 47]. Johnen et al. [69] and Chen et al. [24] propose to build inclusion functions directly from the control points of Bezier polynomials for checking element validity and for high-order continuous collision detection, respectively. In both cases, the algorithms are very efficient and comparable to the numerical root-finding algorithms, but, unfortunately, they both suffer from numerical rounding issues.

For the special case of minimizing polynomial functions over a standard simplex, there are theoretical bounds for the approximation error of a rational grid search [33, 34, 32]. These results could be used to construct inclusion functions for this special case, but we are not aware of any algorithm using them.

Wang et al. [139] propose a custom inclusion function for the continuous collision detection between triangles over linear trajectories. Similarly to the predicates proposed in [120], an exact predicate to check if the range of the inclusion function contains a zero is introduced, thus proposing an efficient and conservative algorithm. Deriving such a predicate is, however, problem-dependent: while there are tools that automate the most tedious part of deriving a floating point filter [106, 87, 11], this is still an error-prone and labor-intensive task.

2.3 Robust numerical computation

Now that we have presented at a high level the existing techniques for conservative global optimization and solution of systems, we introduce the basics of interval arithmetic and overview the state of the art in implementations, which serves as the numerical backbone for the robust solvers proposed in the thesis.

2.3.1 Rational and arbitrary-precision arithmetic

Rational arithmetic represents numbers exactly as ratios of two arbitrary-precision integers, enabling error-free computation for algebraic operations involving rational inputs. While this guarantees correctness by avoiding rounding errors entirely, the bit size of numerators and denominators quickly explodes as computations are performed, requiring simplification, and slowing down the program. The GNU Multiple Precision Arithmetic Library (GMP) [52] is the de facto standard for performing such computations efficiently, yet its cost remains prohibitive for real-time applications or long iterative processes.

Arbitrary-precision floating-point arithmetic offers a middle ground by allowing the user to define the precision (in bits) of the significand, thus reducing rounding errors to an arbitrarily small magnitude. Unlike rational arithmetic, it is not exact, but it allows for controlling the accuracy-performance trade-off dynamically. The MPFR library [45] builds upon GMP to provide correctly rounded floating-point arithmetic with arbitrary precision, ensuring that results are well-defined regardless of the chosen precision level. However, using these software-implemented types is still orders of magnitude slower than hardware-supported floating point numbers.

2.3.2 Interval arithmetic

Interval arithmetic [59] extends arithmetic on the real numbers to the set of closed real intervals $\mathbb{I} := \{[\underline{x}, \overline{x}] | \underline{x} <= \overline{x} \in \mathbb{R}\}$. Given a n-nary operator $\star : \mathbb{R}^n \to \mathbb{R}$, an interval extension of $\star$ must satisfy the *inclusion property*, that is, if $x_i \in [\underline{x_i}, \overline{x_i}] \in \mathbb{I}$ for $i \in \{1, \ldots, n\}$, then $\star(x_1, \ldots, x_n) \in \star([\underline{x_1}, \overline{x_1}], \ldots, [\underline{x_n}, \overline{x_n}])$, where we overload the notation of $\star : \mathbb{I}^n \to \mathbb{I}$ to the interval-valued version. In essence, interval arithmetic defines inclusion functions for common arithmetic operators, but these compute the exact range of the function, rather than a conservative overestimation.

Some basic examples of interval operations include:

$$[\underline{x}, \overline{x}] + [\underline{y}, \overline{y}] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$$
$$[\underline{x}, \overline{x}][\underline{y}, \overline{y}] = [\min(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}), \max(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y})]$$
$$-[\underline{x}, \overline{x}] = [-\overline{x}, -\underline{x}]$$
$$\min([\underline{x}, \overline{x}], [\underline{y}, \overline{y}]) = [\min(\underline{x}, \underline{y}), \min(\overline{x}, \overline{y})]$$
$$\max([\underline{x}, \overline{x}], [\underline{y}, \overline{y}]) = [\max(\underline{x}, \underline{y}), \max(\overline{x}, \overline{y})]$$

by the same logic it is also possible to define relational operators on intervals, such as:

$$[\underline{x}, \overline{x}] = [\underline{y}, \overline{y}] \Leftrightarrow \underline{x} = \underline{y} \wedge \overline{x} = \overline{y}$$
$$[\underline{x}, \overline{x}] > [\underline{y}, \overline{y}] \Leftrightarrow \underline{x} > \overline{y}.$$

Interval arithmetic is useful for two related, but orthogonal tasks. First, it can be used to compute inclusion functions: the interval extension of an operator is in fact an inclusion function, and inclusions for more complicated functions can be computed by composition. This first aspect of IA is independent from how interval endpoints are represented (be it floating point numbers, fractions, or even symbolic expressions), since it relies only on the mathematical properties of the operators. Second, it serves as a proxy for floating-point numbers that bounds numerical error, substituting approximated quantities for "confidence regions", which requires taking care of implementation details such as rounding modes. This use of IA is unrelated with the computation of inclusion functions, and it can be used to perform forward error analysis. The next section and Chapter 5 explore this second topic in greater depth. In the rest of the thesis, we use IA for both: bound the range of functions while taking numerical error into account.

2.3.3 Implementing IA

When working with IA on a computer, one can only represent intervals whose lower and upper bounds can be represented by IEEE 754 floating point (FP) numbers. This has deep implications on the implementation of interval algorithms.

From now on, denote the set of representable FP numbers as $\mathbb{F}$. When x and y are in $\mathbb{F}$, the result $r = x \star y$ may not be in $\mathbb{F}$, hence not representable. In that case, the tightest representable interval containing r is $[\mathrm{FP}^-(r), \mathrm{FP}^+(r)]$, where $\mathrm{FP}^-(r) = max(z : z \in \mathbb{F}, z \leq r)$ and $\mathrm{FP}^+(r) = min(z : z \in \mathbb{F}, z \geq r)$. As mentioned, IEEE 754 requires that when $\star$ is an algebraic operation (or the square root) an implementation of $\star$ must round the theoretically exact result r to either $\mathrm{FP}^-(r)$ or $\mathrm{FP}^+(r)$, depending on the current *rounding mode*. In most modern architectures, a particular register within the CPU controls the rounding mode, and specific system functions exist to set it. Therefore, a trivial approach to create a tight interval for the operation $x \star y$ is:

1. set the rounding towards $+\infty$ and evaluate the expression for the interval's upper bound;
2. set the rounding towards $-\infty$ and evaluate the expression for the interval's lower bound;
3. reset the rounding mode.

Since setting the rounding mode is typically slower than executing arithmetic operations, a more efficient approach involves keeping the rounding mode fixed during the operation, performing a computation that returns the result with opposite sign, and finally changing it to obtain the opposite rounding. For instance, computing a sum $[\underline{x}, \overline{x}] + [\underline{y}, \overline{y}]$ becomes:

1. set the rounding towards $+\infty$ (if no other parts of the program require a different rounding mode, this operation can be done only once at the beginning);
2. execute $\overline{x} + \overline{y}$ to determine the interval's upper bound;
3. execute $-\underline{x}-\underline{y}$, then flip its sign, to determine the interval's lower bound rounded towards $-\infty$.

This approach is used by existing interval arithmetic libraries such as Boost [22] and CGAL [115]. Note that this approach relies on the properties of arithmetic operators, and does not extend to general functions.

Another possibility is to deconstruct the binary representation of the result r to directly modify the mantissa, exponent, and sign, and produce a reasonably small interval around r. This approach is used by Filib [85, 86]. Alternatively, the error propagation can be analyzed to derive a bound ϵ on the rounding so that the interval $i = [r - \epsilon, r + \epsilon]$ is guaranteed to contain r. This is how libraries such as BIAS [78] or GAOL [51] work. The library Filib++ [85, 86] offers several modes that implement the various strategies; according to the authors, the fastest is the *native_onesided_global* mode, which adopts the strategy based on a fixed rounding mode and change of sign described above.

The aforementioned existing libraries were comprehensively compared by Tang et al. [133] who evaluated diverse aspects, including their correctness, efficiency and precision (in terms of interval tightness). They conclude that only Filib and Filib++ are always correct when transcendental functions are involved, although the intervals they produce might be larger than necessary. In contrast, Boost and BIAS may produce intervals that do not contain the exact result. Also, Tang's evaluation could verify that libraries that change the rounding mode produce tighter intervals.

Except for rather old architectures, most existing CPUs provide SIMD registers and instructions that proved useful to accelerate interval arithmetic libraries [79]. The basic idea is to store both bounds in a single 128bit-wide register and perform operations on them in parallel. This and

other optimizations exploiting more recent AVX architectures were included in the NFG library by Attene [12] that, to the best of our knowledge, represents the fastest existing library at the time of writing. Since NFG exploits the rounding mode, it is also guaranteed to produce as-tight-as-possible intervals for all the algebraic operations and the square root.

2.3.4 Correct rounding

Correct rounding (CR) refers to the property that an implementation of a mathematical function f has if, for any x that is representable and contained in the domain of f, it returns the same results one would get by rounding the exact result $f(x)$ to the target representation.

Producing efficient CR implementations of functions is a difficult task that has been actively researched for many years. The classical method involves performing a fast approximation of the function with a known error bound, followed by a correctness check, and a slower but higher-accuracy approximation if the check fails. Interestingly, implementing CR double-precision functions is more difficult than single-precision, because one can check correct rounding exhaustively on 32-bit floats, which is infeasible in the 64-bit case. For double precision, correctness must either be proved formally or tested on known hard-to-round cases.

The most notable libraries for CR mathematical functions are CRlibm [31] and RLibm [96]. CRlibm implements several transcendental functions of one double-precision argument, correctly rounded in the four rounding modes $towards+\infty$, $towards-\infty$, *towards zero*, *to nearest*. However, instead of rounding according to the CPU setting, it provides separate functions for each rounding mode and assumes that the CPU is set to the default *to nearest* mode with ties-to-even (whereas interval arithmetic uses directed rounding). To the best of our knowledge, the project is no longer actively maintained. RLibm is a more recent project that proposes a new approach to correct rounding by polynomial approximants, but it is currently limited to single-precision inputs.

The CORE-MATH Project [123] is an ongoing effort to build a complete collection of correctly rounded C implementations of mathematical functions to foster integration into existing mathematical libraries. CORE-MATH is actively developed and provides efficient CR routines for univariate and bivariate functions with double-precision arguments. For a thorough account of correct rounding we refer the interested reader to a recent survey by Brisebarre et al. [20].

2.4 Domain Specific Languages in graphics

Because Chapter 4 introduces a DSL for the general class of problems outlined in Section 2.2, we conclude this chapter on related works by mentioning other DSL used in graphics for different purposes, to show how these tools can facilitate research and effectively contribute to software

development.

Multiple domain specific languages have been introduced in graphics, which can be broadly divided into two categories. A DSL is introduced to automate code generation for repetitive yet challenging tasks that are tedious and difficult to perform correctly by hand. A prominent example is CVX [29, 53], providing an effective way to solve convex optimization problems and automating the generation of problem-specific code from a high-level specification. Li et al. [94] propose a method to automatically generate LaTeX and source code for linear algebra. Li et al. [95] introduce a DSL for mesh processing, allowing a direct and intuitive description of geometry processing algorithms.

Meyer and Pion [106] and Lévy [87] propose DSL which analyze expression trees to generate the C++ code of corresponding filtered predicates.

Another family of DSL is introduced to improve performance of core algorithmic blocks. These include DSL for sparse linear algebra such as SIMIT [76], TACO [77], EGGS [132, 57] and for optimizing entire simulation algorithms [16], for example TAICHI [62].

Chapter 3

High-Order Continuous Geometrical Validity

This chapter introduces the first generic formulation and algorithm for the continuous validity test of elements, supporting the most common types (such as triangles, quadrilaterals, tetrahedra, prisms, and hexahedra) and extending to high-order basis and geometric maps. Our algorithm is provably conservative when implemented using floating-point arithmetic, meaning that if an element is detected to be valid, it is guaranteed to remain valid throughout the entire specified time interval. This level of robustness, crucial for algorithmic reliability, has not been achieved by any previous method.

Physical simulation frameworks such as PolyFEM [116] deform the mesh at each time step by minimizing some energy via a Newton-style optimization. At each iteration, after computing a descent direction Δx, a line search first finds the largest step along Δx such that the energy remains well-defined throughout the trajectory, before then searching for a step that sufficiently decreases the objective. For elastic energies, well-definedness requires that no element invert at any point along the trajectory: standard hyperelastic potentials diverge as the Jacobian determinant approaches zero, so if an element inverts mid-step the energy is undefined at that point, the trajectory crosses a discontinuity, and the subsequent search for an energy-decreasing step may fail and cause the solver to stall. Our check provides the means to compute a safe step before this discontinuity is reached, in the same way that continuous collision detection (CCD) is used to prevent intersections for contact energies.

A line search over discrete time samples is insufficient for this purpose: for high-order elements, the Jacobian determinant is a polynomial of high degree in time and can change sign transiently within a single step, rendering the element momentarily invalid even if it appears valid at both endpoints. Checking only at fixed spatial quadrature points is equally insufficient, since a local inversion can occur anywhere in the element's interior, not just at sampled points.

One could in principle increase the quadrature order arbitrarily to make violations less likely to be missed, but this does not restore the theoretical guarantees of the FEM formulation, and violations become harder to detect a posteriori. A subdivision-based approach, on the other hand, naturally focuses effort on sub-element regions at higher risk of inversion, while quickly certifying validity in simple regions, without requiring dense quadrature.

If an element becomes invalid at any point, our algorithm provides a conservative estimate of the inversion time and the spatial location of the first inversion, which is used to construct an invalidity-aware quadrature rule (Section 3.5).

While designed for the dynamic case, our algorithm can also be used for the static case, with minor modifications: in this setting, our algorithm is the first to provide a conservative static geometrical validity test for high-order elements without relying on variable bit-length arithmetic.

Our algorithm is designed and implemented for high performance, as its use-case is within optimization loops requiring the testing of large datasets: on static checks, we demonstrate that our test is competitive in terms of runtime with current non-conservative methods, being slightly slower while guaranteeing a conservative answer. To quantitatively evaluate the correctness and efficiency of our approach and compare it with more specialized alternatives, we construct a dataset of 2D and 3D time-dependent queries whose ground truth is computed using (extremely expensive) symbolic root finding.

Having access to a conservative check we discovered that it is very common for high-order FE simulations to contain invalid elements in their solution; we show examples in PolyFEM in Figures 3.9 and 5.1. This seems to be a common problem with high-order FE codes: for example, FEBio [98] also uses a static check only at quadrature points. This issue is rarely mentioned in the literature [9, 37] and we are not aware of other papers proposing a solution. We believe that the presence of invalidity is due to the use of insufficiently accurate quadrature to capture the infinite elastic potential inside some of the most distorted elements. This is a major source of both the numerical fragility of this software and inaccuracy in the solution as physically invalid configurations are reported as the simulation result. By replacing the validity check and the quadrature in PolyFEM with our approach, we show that these issues disappear and the impact on performance is moderate.

3.1 Overview of the method

Given a dynamic element $\overline{x}(\sigma)$ deforming linearly over the time interval $[0, 1]$, we study the determinant of its Jacobian $|J_{\overline{x}}|$ (Sections 3.3 and 3.4). The polynomial $|J_{\overline{x}}|$ can have a high order in both its spatial and temporal variables (Section A.1). To ensure a conservative answer, we employ a custom bisection root-finding method controlled by an accuracy parameter $\delta > 0$.

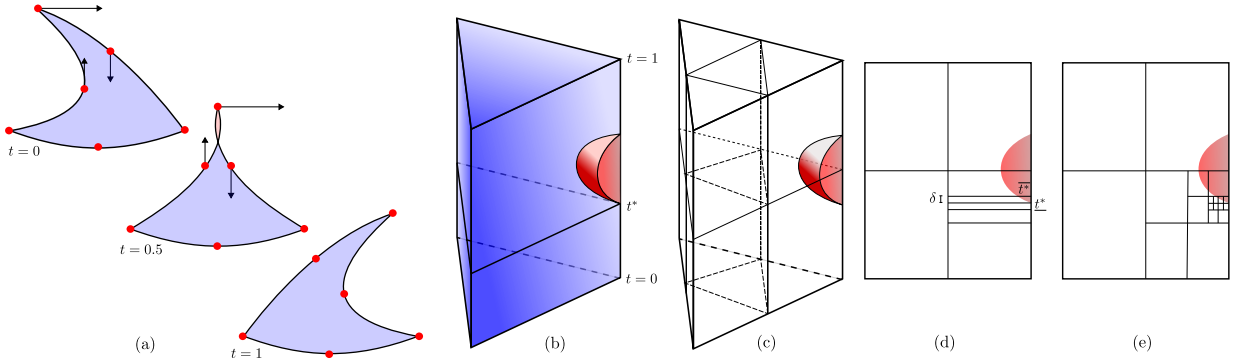


Figure 3.1: (a): A 2D quadratic element with linear trajectories is valid at the start and end positions but locally inverted during the transition. (b): Its parametric domain is an upright triangular prism, where the vertical axis represents time and each horizontal slice represents the space domain at a given time. The red region denotes the portion of the domain in which the determinant $|J_{\overline{x}}|$ is not positive. The dynamic element's Jacobian determinant is a trivariate polynomial in ξ_1, ξ_2, t of order 2 in its space variables and order 2 in the time variable. (c): The root finding algorithm bisects the time domain and quadrisects the spatial domain to isolate a thin slice $[\underline{t^*}, \underline{t^*} + \delta]$ containing the critical time of inversion t^*. In the first step, both space and time dimensions are refined, generating eight sub-domains, which are pushed into a priority queue. (d): For clarity we show a side view for the following steps. Only the time dimension is bisected: the analysis of sub-domains that do not intersect the invalid region increases the value of $\underline{t^*}$ while the analysis of sub-domains that do intersect the invalid region decreases the value of $\overline{t^*}$ until convergence. (e): If all dimensions were refined at all steps, many more sub-domains would be generated, thus killing performance.

Let t^* denote the earliest time at which $\overline{x}(\sigma)$ becomes invalid (i.e., $|J_{\overline{x}}|$ turns negative). We return a time $\underline{t^*} > t^* - \delta$ and a point $\underline{P} \in \sigma$ such that $|J_{\overline{x}}|$ is positive everywhere for $t \leq \underline{t^*}$, and $|J_{\overline{x}}(\underline{P}, t)|$ becomes negative for some $t \leq \underline{t^*} + \delta$. This guarantees that the element can safely deform up to time $\underline{t^*}$ while the point $\underline{P}$ is used to adaptively refine the quadrature rule, steering the simulation away from invalid configurations (Section 3.5). If the element remains valid throughout, we simply return $\underline{t^*} = 1$.

The parameter domain of a dynamic element has both space and time dimensions. Our test proceeds by bisecting the time dimension: it maintains lower $\underline{t^*}$ and upper $\overline{t^*}$ bounds for the critical inversion time, and terminates when the difference between these bounds is smaller than δ, or when the element is confirmed to be valid throughout the interval. The algorithm employs a priority queue of sub-domains, which are created by recursively splitting the initial space-time domain.

For a given sub-domain S, we compute a *minimum inclusion function* that returns an interval

I, which is guaranteed to contain the minimum value of $|J_{\overline{x}}|(S)$: if I is strictly positive, the element is valid in S; if I is strictly negative, the element is invalid somewhere within S (but it is not necessarily invalid everywhere in S); if I contains zero, nothing can be said and further refinement of S is necessary. The minimum inclusion function is a crucial component of our method; details on its definition and computation are provided in Sections 3.2.3 and 3.4.1, respectively.

Another key aspect of the algorithm is the decoupling of refinements in the spatial and temporal dimensions. While the time dimension may require refinement until the interval between $\underline{t^*}$ and $\overline{t^*}$ is lower than δ, the spatial dimensions usually need less refinement. In essence, bisection along the time axis is primarily driven by the need to narrow the bounds of t^*, while subdivision of the spatial domain is employed to resolve indeterminate configurations. Details and pseudocode are given in Section 3.3.1.

To account for numerical errors, we employ interval arithmetic. The value of parameter δ allows for a trade-off between computation time and the accuracy of the estimation; however, regardless of the parameter choice, the algorithm always provides a conservative estimation.

3.1.1 Practical example

Consider the quadratic element in Figure 3.1(a). It is valid at $t = 0$, becomes invalid at some intermediate time $t^* < 0.5$, and then returns to a valid state at a later time, remaining valid until $t = 1$.

The parameter domain for this dynamic element can be visualized as in Figure 3.1(b) with an upright triangular prism, where each horizontal slice of the prism represents the spatial domain at a specific time, and time progresses along the vertical axis from 0 to 1. Notably, the element inversion occurs within a localized wedge (red volume in the figure) during the times when the element is invalid. In general, inversions can occur anywhere within the domain, including regions away from vertices.

The minimum inclusion function is evaluated first for the whole prism. The result is an interval that contains zero, thus the domain is subdivided as in Figure 3.1(c) along both the time and the space dimensions, and eight sub-prisms are pushed to the priority queue. While processing the four prisms corresponding to the time interval $[0, 0.5]$, the minimum inclusion function returns a strictly positive interval for three of them, which are discarded from the queue; and it returns a strictly negative interval for the fourth one, which intersects the red wedge: this domain is bisected only in the time dimension and its two children are pushed onto the queue. Sub-domains spanning earlier times are processed first.

In the subsequent refinements (Figure 3.1(d), side view), the minimum inclusion function will always be strictly negative, hence only bisection in the time dimension will occur: the analysis

of valid intervals that do not intersect the red wedge contributes to increasing the value of $\underline{t^*}$ while the analysis of intervals that do intersect the red wedge contributes to decreasing the value of $\overline{t^*}$, until convergence. Note that, if the domain were always subdivided along space and time, as in Figure 3.1(e), many more subdomains would be generated and the algorithm could become much slower. For the 3D order 3 armadillo dataset (Table 3.2, Figure 3.6), with 10% target error, the naive approach has an average processing time of $\sim 7ms$ per element, whereas our approach takes $\sim 230\mu s$ per element on average. For higher element orders, or queries with higher precision, the maximum number of subdivisions used in the naive approach must be limited (otherwise the check will consume unreasonable amounts of memory), and the test can fail to produce an estimate within precision on some elements.

3.2 Preliminaries and notations

We address n-dimensional ($n = 2, 3$) high-order meshes consisting of elements of various types. The geometry of every element is defined by a polynomial map. We will refer to the *order* p of a polynomial as the maximum exponent of a single variable (as opposed to the usual notion of degree). Table 3.1 gives a summary of symbols defined in the following and used throughout.

3.2.1 Reference element domains

For each type of element, we define a common *reference domain* (or *reference element*) $\sigma_s^n \subset [0,1]^n$ to use as the coordinate domain.

Definition 1 (Reference Domain)**.** *Let $n, s \in \mathbb{N}$, $1 \leq s \leq n$. The n-dimensional reference domain $\sigma_s^n \subset [0,1]^n$ is the locus of points with coordinates $(\xi_1, \ldots, \xi_n)$ that satisfy the system of inequalities:*

$$\begin{gathered} \xi_i \geq 0 \quad \forall i \in \{1, \ldots, n\} \\ 1 - \sum_{i=1}^{s} \xi_i \geq 0 \\ \xi_i \leq 1 \quad \forall i \in \{n-s+1, \ldots, n\} \end{gathered} \tag{3.1}$$

With this notation, we have a general parameter space that works for all the most commonly used FEM elements. The element σ_1^1 is the unit segment; σ_1^2 is the unit square; σ_2^2 is the standard triangle; σ_1^3 is the unit cube; σ_2^3 is the unit triangular prism; σ_3^3 is the standard tetrahedron. More generally, σ_s^n is the tensor product of a standard s-simplex with a standard $(n-s)$-hypercube[1].

[1]Note that, the unit segment can be seen both as a 1-simplex and as a 1-hypercube; to avoid any ambiguity, we always treat it as a simplex, so that, e.g., the unit square is regarded as the tensor product of a 1-simplex with a 1-hypercube, rather than as a 2-hypercube.

Table 3.1: Symbols used in the text and where they are defined.

Symbol	Meaning	Def.
n	dimension of element and embedding space	3.2
s	dimension of simplicial part of element	3.2.1
σ, σ_s^n	static reference element (with dimensions)	3.2.1
$\xi_1, \ldots \xi_n$	spatial coordinates	3.2.1
p	order of an element / polynomial	3.2.2
x	geometric map of an element	3.2.2
$\lvert J_x \rvert$	Jacobian determinant of x	3.2.2
$\mathbb{I}$	intervals on the real line	3.2.3
$\square f$	inclusion function for f	3.2.3
$\square_{\min} f$	minimum inclusion function for f	3.2.3
t	time coordinate	3.3
$\overline{\sigma}, \overline{\sigma}_s^n$	dynamic reference element (with dim.)	3.3
$\overline{x}$	dynamic geometric map of an element	3.3
t^*	minimum time at which $\lvert J_{\overline{x}} \rvert$ vanishes	3.3
$\underline{t^*}, \overline{t^*}$	lower and upper bounds to t^*	3.3
δ	user-specified accuracy	3.3
$l_{\max}$	maximum level of recursion	3.3
ψ^-, ψ^+	time-only subdivision maps	3.3
ψ^q	q-th subdivision map	3.3
$\mathcal{C}_\sigma^p$	subset of indices of corners of element σ	3.4
$\mathbf{T}^{\pm}_{\mathcal{B}\to\mathcal{B}}$	time-only subdivision matrix	3.4
$\mathbf{T}^{q}_{\mathcal{B}\to\mathcal{B}}$	q-th subdivision matrix	3.4
Γ_σ^p	set of domain points of order p on σ	A.1
γ_i	domain point	A.1
$\mathcal{I}_\sigma^p$	set of indices of points of Γ_σ^p	A.1
$\mathcal{L}_i^m$	i-th Lagrange polynomial of order m	A.1
$\mathcal{B}_i^m$	i-th Bernstein polynomial of order m	A.1
$f^{\mathcal{L}}$	vector of coefficients in Lagrange form	A.1
$f^{\mathcal{B}}$	vector of coefficients in Bézier form	A.1
$\mathbf{T}_{\mathcal{L}\to\mathcal{B}}$	transition matrix from Lagrange to Bézier	A.1

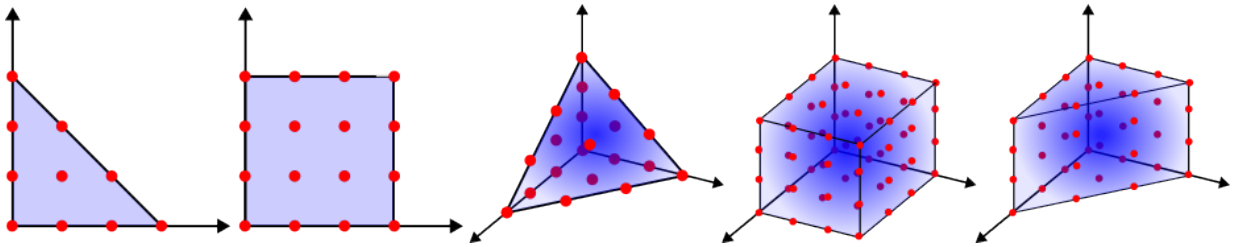


Figure 3.2: Reference elements with domain points of order 3: triangle, square, tetrahedron, hexahedron, prism.

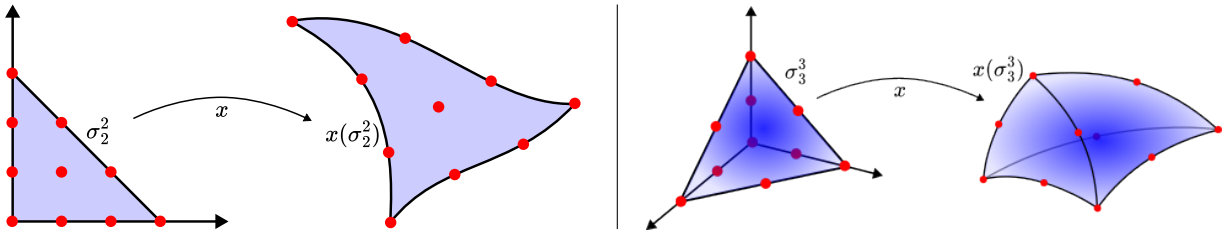


Figure 3.3: Above: A domain σ_2^2 with its domain points of order 3 and the geometric map x to the physical element $x(\sigma_2^2)$ with Lagrange control points. Below: likewise for an element σ_3^3 of order 2.

3.2.2 High-order elements

A generic polynomial $f : \sigma \longrightarrow \mathbb{R}$ can be defined with a basis of polynomials: we consider here the Lagrange basis – which is most common in FEM – and the Bernstein basis that gives the Bézier form. Both representations define f as a linear combination of the basis functions with *control coefficients* associated with *domain points* that form a regular grid over the reference element; the number of domain points sets the *order* p of the polynomial (Figure 3.2).

For a given order and dimension, pre-computed conversion matrices allow us to convert between these two representations in both directions. The Lagrange and Bézier forms and the conversion matrices are detailed in Section A.1.

The geometric map $x : \sigma_s^n \longrightarrow \mathbb{R}^n$ that maps a reference element σ_s^n into the physical element $x(\sigma_s^n)$ is represented by specifying its set of control points, where each control point is a n-dimensional point (Figure 3.3). Each coordinate of x is expressed with a multivariate polynomial, which is the tensor product of a s-variate polynomial of degree p with $(n - s)$ univariate polynomials of degree p, each in a different variable. It follows that the map x has order p in all its variables (even though its total degree may be higher).

In the following, we study the determinant of the Jacobian J_x of the geometric map x, denoted

$|J_x|$, which is also a multivariate polynomial in the same variables as x, but of a different order. In particular, the number of terms of $|J_x|$ rapidly increases with the dimension and order of the element. See Section A.1 for details.

3.2.3 Minimum inclusion function

Intuitively, given a real function f and a domain D, an *inclusion function* for f over D returns an interval that bounds the range of values of f in D. Inclusion functions are widely used in root finding and in the evaluation of robust predicates [125]. In our case, we are rather interested in an interval that just contains the *minimum* value of f.

Let $\mathbb{I}$ be the space of intervals on the real line. For $a = [\underline{a}, \overline{a}] \in \mathbb{I}$, let us define $w(a) = \overline{a} - \underline{a}$ the *width* of interval a. Let $A = a_1 \times \cdots \times a_n \in \mathbb{I}^n$ be a n-dimensional interval; we extend the definition of width as $w(A) = \max_{j=1}^{n} w(a_j)$. Given $D \subseteq \mathbb{R}^n$ compact, we further extend the definition of width as $w(D) = \min_{A \supseteq D} w(A)$.

Definition 2 (Inclusion function)**.** *Let $\Omega \subseteq \mathbb{R}^n$ be a compact domain, let f be a real function defined on Ω, and let us denote $\mathcal{P}(\Omega)$ the subsets of Ω. Given a function $f : \Omega \longrightarrow \mathbb{R}$, an* inclusion function *for f is a function $\square f : \mathcal{P}\Omega \longrightarrow \mathbb{I}$ such that, for any $D \subseteq \Omega$ we have*

$$\forall \xi \in D \quad f(\xi) \in \square f(D).$$

Definition 3 (Convergent inclusion function)**.** *We say $\square f$ to be* convergent *if for any $D \subseteq \Sigma$*

$$w(D) \to 0 \Rightarrow w(\square f(D)) \to 0.$$

In particular, if D shrinks about ξ, then $\square f(D)$ shrinks about $f(\xi)$.

Definition 4 (Minimum inclusion function)**.** *A minimum inclusion function for the function f is a function $\square_{\min} f : \mathcal{P}(\Omega) \longrightarrow \mathbb{I}$ such that, for any $D \subseteq \Omega$ we have*

$$\min_{\xi \in D} f(\xi) \in \square_{\min} f(D).$$

If the lower end of $\square_{\min} f(D)$ is positive, we know that f is everywhere positive in D; if the upper end of $\square_{\min} f(D)$ is negative, we know that f has at least one negative value in D; otherwise, nothing can be said about the sign of f in D. Compare this with a classical inclusion function $\square f(D)$, for which a negative upper bound implies f is everywhere negative in D.

A convergent inclusion function can be used to find a root of a function f by subdividing the initial domain Ω until it becomes sufficiently small. Likewise, one can use a convergent minimum inclusion function to find the portions of Ω where f is positive, by recursively subdividing the domain. The type of subdivision used to perform refinement depends on the shape of Ω. For instance, while bisection can be used for a multi-interval domain, simplicial domains may require less trivial subdivision rules (Section A.2).

3.3 Continuous geometrical validity

In a dynamic simulation, the elements of the mesh move and deform over time. Like space, time is discretized into time steps, which are typically regular. We assume that the control points move along straight-line trajectories at each time step, and, without loss of generality, we can assume each transition occurs between time $t = 0$ and time $t = 1$. Following Definition 1 we have:

Definition 5 (Dynamic reference element)**.** *Let σ_s^n be a reference element. The dynamic element $\overline{\sigma}_s^n$ of σ_s^n is the $(n+1)$-dimensional reference element $\sigma_s^{n+1} = \sigma_s^n \times [0, 1]$.*

Assuming linear trajectories, the *dynamic geometric map* $\overline{x} : \mathbb{R}^{n+1} \to \mathbb{R}^n$ of order p for $\overline{\sigma}_s^n$ is expressed by linear interpolation of the n-dimensional geometric maps $x^0(\xi)$ and $x^1(\xi)$ of the static element at the two consecutive time steps:

$$\overline{x}(\xi, t) = x^0(\xi) + t(x^1(\xi) - x^0(\xi)). \tag{3.2}$$

This map is of order p in the ξ variables and linear in t.

3.3.1 Continuous validity test

Assuming that the input element is valid at time $t = 0$, the *continuous validity test* consists of determining whether or not the Jacobian determinant $|J_{\overline{x}}(\xi)|$ is everywhere greater than 0 on $\bar{\sigma}$ at all times in $[0, 1]$. If this is not the case, the algorithm should find the earliest time $t^* \in (0, 1]$ in which the element becomes invalid – hence, it is valid within $[0, t^*)$.

3.3.1.1 Target accuracy

Since finding an exact solution is not necessary and very expensive, we instead settle for a conservative estimate $\underline{t^*}$ that is close to t^* up to a given user-provided threshold δ, i.e. $t^* \in [\underline{t^*}, \underline{t^*} + \delta]$. Moreover, since we assumed that $t^* > 0$, we require that $\underline{t^*}$ be strictly positive as well, regardless of the value of δ. This threshold is used to trade off accuracy and time performance.

3.3.1.2 Early termination

An intrinsic challenge of both the static and dynamic problems is that certifying the positivity of a polynomial can be arbitrarily hard. Therefore, there is no upper bound on the number of subdivisions required to assess the validity of an element.

To prevent the algorithm from taking unreasonable time, we employ a termination criterion that triggers when refinement becomes excessive. In these extreme cases, we halt and provide an estimate $\underline{t^*}$ that may not be within δ of the true value t^*, but is nevertheless conservative.

Let the *depth* of a subdomain S be the number of subdivisions required to obtain S from the initial domain. In our implementation we stop when the depth of a subdomain S exceeds a threshold $l_{\max}$. How often this condition is triggered in practice depends on the dataset and element type. In our benchmark, choosing a value of $l_{\max} = 7$ was enough to reach the desired precision of 10^{-2} on all but very few 3D tensor product elements. We refer to Table 3.2 and Section 3.7 for details.

3.3.1.3 Minimum inclusion function

We rely on a convergent inclusion function, which comes together with a procedure to decompose an element into sub-elements. For the sake of clarity and generality, we first describe our algorithm avoiding the details on how we define our inclusion function and domain subdivision strategy, which we detail in Section 3.4. We also use $J = |J_{\overline{x}}|$ as a short-hand notation for the determinant of the Jacobian of the dynamic element at hand.

Given a generic inclusion function $\square J$ on the domain $\overline{\sigma}_s^n$, we define a minimum inclusion function as follows: let $\square J(D) = [\underline{J_D}, \overline{J_D}]$ then

$$\square_{\min} J(D) = [\underline{J_D}, \min_{\overline{\xi}_i \in \mathcal{S}_D} J(\overline{\xi}_i)],$$

where $\mathcal{S}_D$ is a small set of samples in D. In practice, we sample J at these points to bound the minimum of J from above. Note that, if J is negative at any of those samples, we know that the element becomes invalid in D.

3.3.1.4 Subdivision maps

Given a reference domain $\sigma = \sigma_s^n$, we define a set of Q linear maps $\{\psi^q : \sigma \longrightarrow \sigma\}_q$ called the *subdivision maps* of σ, and we call $\psi^q(\sigma)$ a *subdomain* of σ; we require that the union of all subdomains is σ, and the intersection of any two subdomains is either empty or has dimension less than n. We define the standard subdivision maps for $n \in \{2, 3\}$ that we use in our implementation in Section A.2. In the following, we always assume that $Q = 2^n$.

For a time dependent reference domain $\overline{\sigma} = \overline{\sigma}_s^n$, its subdivision maps are the same as σ_s^{n+1}, but we also define two additional *time subdivision maps*, denoted ψ^- and ψ^+, such that $\psi^-(\overline{\sigma})$ and $\psi^+(\overline{\sigma})$ are respectively the lower and upper half of $\overline{\sigma}$ when bisected in the time dimension only.

Algorithm 1 Maximum valid time step with inclusion functions

function MaxValidStep($J, \delta, l_{\max}$)
  $P \leftarrow$PriorityQueue($\prec$) ▷ *priority queue for subdomains*
  $\overline{t^*} \leftarrow 1$ ▷ *initialize upper bound of* t^*
  $\underline{t^*} \leftarrow 0$ ▷ *initialize lower bound of* t^*
  Push(P, σ)
  $F \leftarrow$ false ▷ *flag of whether an invalidity has been found*
  $l \leftarrow 0$ ▷ *maximum subdivision depth reached so far*
  while true **do**
    if $F \wedge (\overline{t^*} - \underline{t^*} \leq \delta) \wedge (\underline{t^*} > 0)$ **then** ▷ *reached accuracy*
      return $\underline{t^*}$ ▷ *conservative estimate of* t^*
    if IsEmpty(P) **then**
      return 1
    $S \leftarrow$Pop(P) ▷ *get the next subdomain from* P
    $l \leftarrow \max\{l,$ Depth(S)$\}$ ▷ *update maximum depth*
    if $l > l_{\max}$ **then** ▷ *maximum level reached: give up*
      return $\underline{t^*}$ ▷ *conservative estimate of* t^*
    $\underline{t^*} \leftarrow$ StartTime(S) ▷ *everything before this time is valid*
    $I \leftarrow \square_{\min} J(S)$ ▷ *check minimum inclusion*
    if High(I) ≤ 0 **then** ▷ *there is an invalidity in* S
      if EndTime(S)$< \overline{t^*}$ **then**
        $F \leftarrow$ true
        $\overline{t^*} \leftarrow$ EndTime(S)
      Push($P, \psi^-(S)$) ▷ *bisect on the* t *axis only*
      Push($P, \psi^+(S)$) ▷ *bisect on the* t *axis only*
    else if $\neg$(Low(I) > 0) **then**
      for $q \in \{1, \ldots, Q\}$ **do**
        Push($P, \psi^q(S)$) ▷ *subdivide on* ξ *and bisect on* t
function $\prec(S_0, S_1)$ ▷ *priority function*
  if StartTime(S_0) $\neq$ StartTime(S_1) **then** ▷ *lower time first*
    return StartTime(S_0)$<$StartTime(S_1)
  else ▷ *for ties, prioritize boxes most likely to be invalid*
    return High($\square_{\min} J(S_0)$)$<$High($\square_{\min} J(S_1)$)

3.3.1.5 Pseudocode

The pseudo-code of the algorithm is given in Algorithm 1. The algorithm takes in input a polynomial $J(\xi, t)$ defined on domain $\overline{\sigma}$ and the thresholds δ and l_{max} and returns a time $\underline{t^*}$ without invalid configurations, and within a time δ of an invalid configuration. The algorithm keeps internal current lower and upper bounds for t^*, initializing them to 0 and 1, respectively. Given

a subdomain $S \subset \overline{\sigma}$, pseudocode functions StartTime(S) and EndTime(S) return respectively the minimum and maximum values of the time coordinate for points in S.

The algorithm uses a priority queue P (line 2) of subdomains of σ, with a related priority function $\prec$ (line 28) giving higher priority to sub-domains that span intervals of time with an earlier start point (that is, a lower minimum t). The initial domain σ is pushed into the priority queue P (line 5); then elements are popped from the P one by one (line 13), and if their minimum inclusion function does not guarantee their validity, they are subdivided and their subdomains are pushed to P. See the next subsubsection for the subdivision strategy. This continues until the queue becomes empty or an early exit condition is met. By construction, the priority function $\prec$ guarantees that when we pop an element S from the queue, then J is positive at all times before StartTime(S), and $\underline{t^*}$ can then be updated accordingly.

Early exits occur if the required accuracy δ is achieved (line 9), meaning that the difference between $\overline{t^*}$ and $\underline{t^*}$ is less than δ, or the maximum depth $l_{\max}$ has been reached (line 15), meaning that we pop from the queue an interval that comes from a sequence of $l_{\max}$ subdivisions.

3.3.1.6 Subdivision strategy

If the interval I returned by $\square_{\min} J(S)$ is completely negative – meaning that S contains negative values of J (line 19) – then the upper bound $\overline{t^*}$ is updated, and element S is bisected along the time dimension only, with the two resulting subdomains being pushed into P. Conversely, if interval I contains zero, the element S is split along all its dimensions (line 25), including time, according to the subdivision scheme of reference element $\overline{\sigma}$; again, the resulting elements are pushed into P. Finally, no subdivision is necessary if I only contains positive values, and the space-time region occupied by S will not be considered again for the remainder of the algorithm.

Note that, by bisecting only the time dimension (line 19), we postpone any refinement of the spatial dimensions until we find a time interval in which J may potentially be positive everywhere. This strategy allows us to avoid many unnecessary refinements in the space dimensions, and to decouple the subdivision on time (controlled by accuracy δ) from the subdivision in space (which does not have an accuracy requirement).

3.4 Implementation

The implementation of our method requires designing a minimum inclusion function $\square_{\min} f$ and a corresponding subdivision strategy that uses robust computations while keeping the runtime sufficiently low to enable its use within a simulation loop.

3.4.1 Inclusion functions for space and time

Interval arithmetic provides a universal way to design inclusion functions. Given a polynomial $f : \Omega \longrightarrow \mathbb{R}$ and $D \subseteq \Omega$, let A_D be the smallest multi-interval containing D. We could define

$$\square f(D) = f(A_D),$$

where the evaluation of f on the right side is intended with interval arithmetic, and thus returns an interval. Any strategy subdividing D and reducing A_D (e.g., bisection along all coordinates) provides a convergent inclusion function.

We tried this approach, but the inclusion functions may be very loose about f and require many refinement steps to converge, or even get stuck on nearly invalid elements due to the numerical error accumulating too fast for the inclusion function to keep up with. We compare the time performance for the static case only in Table 3.3. We instead follow the approach proposed by Johnen et al. [69] for the static validity test and extend it to our continuous setting.

3.4.1.1 Overview of Bézier refinement

Let f be the order m polynomial of which we want to find the minimum on $\overline{\sigma}$ (in our case, $f = |J_{\overline{x}}|$).

Our inclusion function is based on the Bézier representation of f and a recursive decomposition of $\overline{\sigma}$. The reason why we want to represent our polynomial in the Bézier basis is the convex hull property [41], by which the values of f on $\overline{\sigma}$ are bounded by the minimum and maximum coefficients of f when expressed in the Bézier basis.

To obtain the vector of Bézier coefficients $f^{\mathcal{B}}$ of f, we first compute its vector of Lagrange coefficients $f^{\mathcal{L}}$, which can be obtained by simply evaluating f at the domain points; then we premultiply $f^{\mathcal{L}}$ with a change of basis matrix $\mathbf{T}_{\mathcal{L}\to\mathcal{B}}$ that we shall call *transformation matrix*, which is described in detail in Section A.1.

Let $\overline{\mathcal{I}}_\sigma^m$ be the set of indices of the control points of f and $\overline{\mathcal{C}}_\sigma^m \subset \overline{\mathcal{I}}_\sigma^m$ be the set of indices corresponding to the corners of $\overline{\sigma}$ at time 1. Since the Bézier basis is interpolating at the corners of the domain (i.e. $\beta_j = f(\gamma_j)$ for all $j \in \overline{\mathcal{C}}_\sigma^m$), we define the minimum inclusion function as

$$\square_{\min} f(\overline{\sigma}) = [\min_{i \in \overline{\mathcal{I}}_\sigma^m} \beta_i, \min_{j \in \overline{\mathcal{C}}_\sigma^m} \beta_j]. \tag{3.3}$$

Therefore, if all entries of $f^{\mathcal{B}}$ are positive we know the element is valid everywhere, and if any of the corner entries is non-positive we know that the element is invalid at the end time. Otherwise, the interval returned by the inclusion function contains zero, and we need to refine the search by subdividing $\overline{\sigma}$.

The subdivision of $\overline{\sigma}$ is performed via another set of change of basis matrices, dubbed *subdivision matrices*, $\mathbf{T}^q_{\mathcal{B}\to\mathcal{B}}$, for $q = 1, \ldots, Q$. Premultiplication of $f^{\mathcal{B}}$ by these matrices gives a Bézier representation of f on a smaller portion of the domain, which can be used to compute tighter bounds local to each subdomain. These matrices are defined in Section A.1.

Bisection in the time dimension only is performed analogously by multiplication of $f^{\mathcal{B}}$ with two *time subdivision matrices* $\mathbf{T}^-_{\mathcal{B}\to\mathcal{B}}$ and $\mathbf{T}^+_{\mathcal{B}\to\mathcal{B}}$, also described in Section A.1.

3.4.2 Robust computation

All the transformation and subdivision matrices $\mathbf{T}_{\mathcal{L}\to\mathcal{B}}$, $\mathbf{T}^q_{\mathcal{B}\to\mathcal{B}}$, and $\mathbf{T}^{\pm}_{\mathcal{B}\to\mathcal{B}}$ are only dependent on element type (tetrahedron, hexahedron, etc.) and order, and as such can be precomputed offline. Since we want to minimize the accumulation of error in our computations, and all entries of these matrices are rational, we construct these matrices using exact rational arithmetic for each element type and order. The outcome is a rational matrix, which we convert to intervals by rounding the two endpoints outward if the exact value cannot be represented as a floating point number. The resulting interval is guaranteed to contain the exact value of the fraction while being as tight as possible.

The input to our check is the set of Lagrange control points for the elements of a mesh, represented in floating point. Each floating point coordinate is converted to a singleton interval (i.e. an interval with zero width) and all subsequent operations are performed in interval arithmetic with conservative rounding.

The combination of rational precomputation and interval arithmetic ensures that our algorithm is conservative while maintaining a low computational cost (Section 3.6): the rational precomputation is performed only once offline and does not affect runtime, while the use of interval arithmetic adds a minor ($\sim 2\times$) overhead over a direct floating point implementation.

3.4.3 Acceleration

3.4.3.1 Global queries

In practical applications, one is often interested in the maximum time for which *all* elements are valid. We refer to this as a *global* dynamic query and give a strategy to accelerate it.

After an invalid element has been found (with estimated valid time step $\underline{t^*}$), it becomes unnecessary to validate the other elements at later times: $\underline{t^*}$ will anyhow be the maximal allowed step. It might be tempting to terminate early as soon as an inverted element has been found. However, this is not conservative as some other elements may still have a lower $\underline{t^*}$. We thus keep track of the smallest value of $\underline{t^*}$ found in previous checks and leverage the fact that $\underline{t^*}$ can

never decrease in Algorithm 1 to stop computation on an element as soon as its estimate for t^* exceeds the running minimum.

The order in which elements are processed matters for global queries: it is beneficial to process elements that are most likely invalid first, as it will provide higher opportunities for this pruning strategy to be effective. For this reason, we first sort the polynomials according to their constant term, in ascending order. For the order 3 Armadillo mesh in Figure 3.6, this strategy improves the total running time by about 60% over running the queries individually; whereas the speedup is less relevant for the order 2 mesh (about 12%).

3.4.3.2 Parallelization

Validity checks for meshes are trivially parallelized by processing elements in batches. To avoid any synchronization between different threads, every batch of queries assigned to a thread i does an independent sorting and keeps its own running minimum $\underline{t^*}_i$ to use as an early termination condition, as explained in the previous subsubsection.

3.4.3.3 Precomputation of Jacobian determinant

The input to the subdivision procedure is a Lagrange representation of the Jacobian determinant polynomial of the element. This only depends on the shape and order of the element, as well as its control points.

For each element type and order combination, we symbolically compute the expression of each Lagrange coefficient in terms of the control point coordinates, and remove common subexpressions with CSE [110] (we use the implementation in SymPy [105]). This approach increases the compilation time but provides dramatic runtime performance boosts: for the order 3 Armadillo dataset, we get a speedup of about $20\times$.

3.5 Application to simulation

Incremental potential time-stepping [73] is becoming popular in graphics [88] and biomechanics [103] due to its robustness to extreme deformation and contact[89, 80, 39, 42, 81, 82, 91, 25, 43, 92, 83, 65, 64, 119, 93, 40]. We briefly summarize the approach here, without contact handling, as it is relevant to motivate the need for a continuous dynamic positivity check in physical simulation: as part of this overview, we will show that the check alone is insufficient, as IPC also requires a *consistent* invalidity-aware quadrature rules, which we introduce in Section 3.5.2.

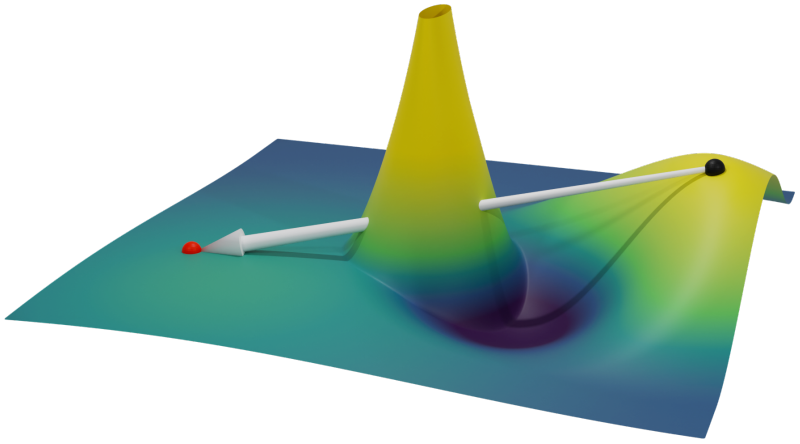


Figure 3.4: Potential landscape and one descent step from the black point. The descent step to the red point is invalid since it crosses an invalid region with infinite potential.

3.5.1 Continuous validity in simulation

The updated displacement u^{t+1} of an object at the next time step is computed solving an *unconstrained* non-linear energy minimization:

$$u^{t+1} = \arg\min_{u} \; E(u, u^t, v^t), \tag{3.4}$$

where u^t is the displacement at the step t, v^t is velocity, and $E(u, u^t, v^t)$ is a time-stepping Incremental Potential [73]. We refer to Li et al. [88] for more details.

For common non-linear material models, this potential is infinite when an element has a negative Jacobian, as the Jacobian determinant appears in the denominator of the expression. A physically valid trajectory cannot reach a state with infinite potential: however, this is a challenging condition to enforce in practice.

3.5.1.1 Line search

The potential E is minimized with a descent algorithm (gradient descent or Newton), which computes a local approximation of a descent direction: this approximation might, for a finite step length, cross a region with infinite potential (Figure 3.4). This is a typical challenge in collision detection [139], but is rarely considered for the elastic potential – the only work we know that considers this problem, in 2D only, is Smith and Schaefer [124]. This challenge can be solved using a continuous inversion check within the line search, which is the focus of our work. To the best of our knowledge, state-of-the-art IPC solvers Li et al. [88] and Schneider

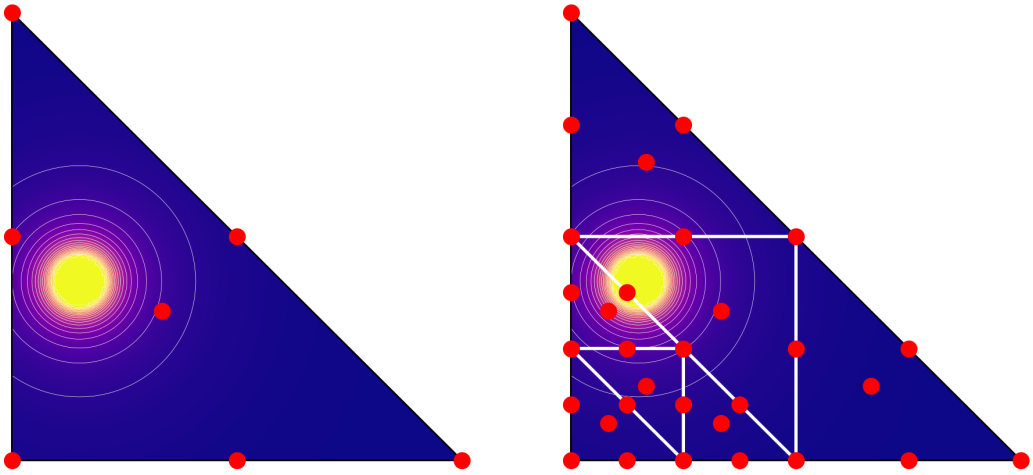


Figure 3.5: The function is infinite in the bright yellow region, and its integral over the whole triangle is also infinite. Numerical integration using a fixed quadrature rule (left) erroneously produces a finite value. Our adaptive quadrature technique (right) puts quadrature points in the infinite-valued region and correctly captures the behavior of the function.

et al. [116] use a static check instead of a continuous one, which cannot guarantee trajectory validity.

3.5.1.2 Quadrature

Non-linear elastic potentials cannot be integrated exactly with numerical quadrature (as they are not polynomials), leading to unbounded errors for diverging potentials. We show in Figure 3.5 an example of an element with an infinite potential integrated with both a standard fixed quadrature rule and the adaptive quadrature derived by our algorithm: only in the second case does the numerical integration correctly diverge. The use of a fixed quadrature leads to solver failures as the direction computed using quadrature is not a descent direction and might thus block the progress of the solver.

3.5.2 Invalidity-aware quadrature rules

Our algorithm can output additional information to generate adaptive quadrature schemes: The goal is to produce a set of quadrature points for the static element σ such that at least one of the points would end up in an invalid region at time t^*, correctly making the integral computed using quadrature diverge at that time.

3.5.2.1 Tracking subdivisions

In Algorithm 1, each subdomain keeps track of the sequence of subdivisions of $\overline{\sigma}$ that were taken to produce it. However, since we only require a quadrature rule for σ and not $\overline{\sigma}$, it is sufficient to keep track of spatial subdivisions: this means that when an element is subdivided in time only using ψ^+ and ψ^-, the sequence of the children will match that of the parent; and when an element is subdivided in all dimensions using the subdivision scheme ψ^q for $\overline{\sigma}$, pairs of subdomains that span the same region of space at different times will share the same sequence.

To produce the adaptive quadrature rule, our algorithm uses the sequence of spatial subdivisions of the element whose minimum inclusion function reduced $\overline{t^*}$ for the last time, or equivalently, the element that found the earliest invalidity. Unless the early exit condition for maximum depth is triggered, such invalidity is guaranteed to be in $[\underline{t^*}, \underline{t^*} + \delta]$. In the very rare cases when the algorithm fails to find an invalid point and gives up earlier, we instead return the subdivision sequence of the element with the deepest hierarchy, which is likely to be very close to an invalid region. In this case, the optimization step will still be guaranteed to be valid for its duration.

3.5.2.2 Adaptive quadrature

This information is then used to partition the static element by recursively subdividing it using the very same sequence of subdivisions, as in Figure 3.5 (right). A standard quadrature rule is applied to every subelement of σ, and the integral is evaluated as the sum of integrals on all subdomains. However, in order to guarantee that the newly placed quadrature points would indeed intersect an invalid region in the full time step, it is required that the quadrature rule contain all points in the sampling set $\mathcal{S}_D$ used to compute the minimum inclusion function, projected onto σ; for example, if $\mathcal{S}_D$ is the set of corners of $\overline{\sigma}$, the selected scheme must place quadrature points at the corners of σ.

3.6 Results

Our algorithm is implemented in C++, using PolyFEM [116] for finite element (FE) system construction, IPC Toolkit [44] for evaluating IPC potentials and collision detection, Pardiso [7, 18, 17] for the large linear systems in our global Newton solves, [10, 11] for interval computation, GMP [52] for rational computation, and OpenMP [30] for parallelization.

The simulation experiments are run on a cluster node with an Intel Cascade Lake Platinum 8268 processor limited to 16 threads and 32Gb of memory. The benchmark experiments are run single-threaded on a laptop computer with an AMD Ryzen 7 4000 processor and 16GB of memory.

Table 3.2: Results of our continuous validity test on 2D and 3D datasets element types and orders. We report the number of elements (total, valid on the whole interval, invalid at some time) and processing time in microseconds per element (average, average for valid elements, average for invalid elements, median, maximum, standard deviation). On some datasets, marked with an asterisk (*), the algorithm "gives up" on very few elements and returns a conservative answer (these are 57 hexahedra and 26 prisms for which the algorithm exceeds the available memory on the personal machine where the benchmark was run, and must be stopped early via the $l_{\max}$ parameter, set to a maximum subdivision depth of 7; altering the subdivision strategy mitigates this issue and allows the algorithm to converge on the whole dataset). In all other cases, the algorithm reaches the target precision of 1% on all elements.

Dataset	Type	n	s	p	Element count			Time per element (μs)					
					tot	val	inv	avg	avg val	avg inv	med	max	std
Kangaroo	Tris	2	2	1	172800	146248	26552	1.60	1.32	3.10	1.40	205.40	0.85
				2	172800	145602	27198	2.14	1.69	4.54	1.89	42.53	1.10
				3	172800	145623	27177	3.83	3.10	7.75	3.28	67.26	1.86
				4	172800	145613	27187	9.28	7.94	16.45	7.96	309.19	4.03
Bar 2D	Quads	2	1	1	112887	14816	98071	3.93	2.95	4.07	3.84	10203.74	34.23
				2	83653	75085	8568	74.67	7.89	659.96	2.93	2046480.70	10296.53
Armadillo	Tets	3	3	1	54985	51605	3380	1.56	1.44	3.43	1.47	21.58	0.53
				2	54985	48599	6386	9.90	7.34	29.41	5.31	6726.90	40.37
				3	54985	47988	6997	362.79	130.92	1953.05	44.00	78411.99	1593.81
Bunny	Tets	3	3	1	19800	19561	239	1.47	1.45	3.51	1.40	49.38	0.63
				2	19800	19058	742	7.27	5.91	42.22	5.31	1729.90	24.56
				3	19800	18954	846	175.41	65.97	2627.34	43.02	60907.39	1294.73
Bar 3D	Hexes	3	1	1	56031	24918	31113*	325.33	11.10	576.99	16.55	6650064.32	39487.14
	Prisms	3	2	1	82403	56913	25490*	246.64	8.22	778.96	2.38	2407415.47	16401.57

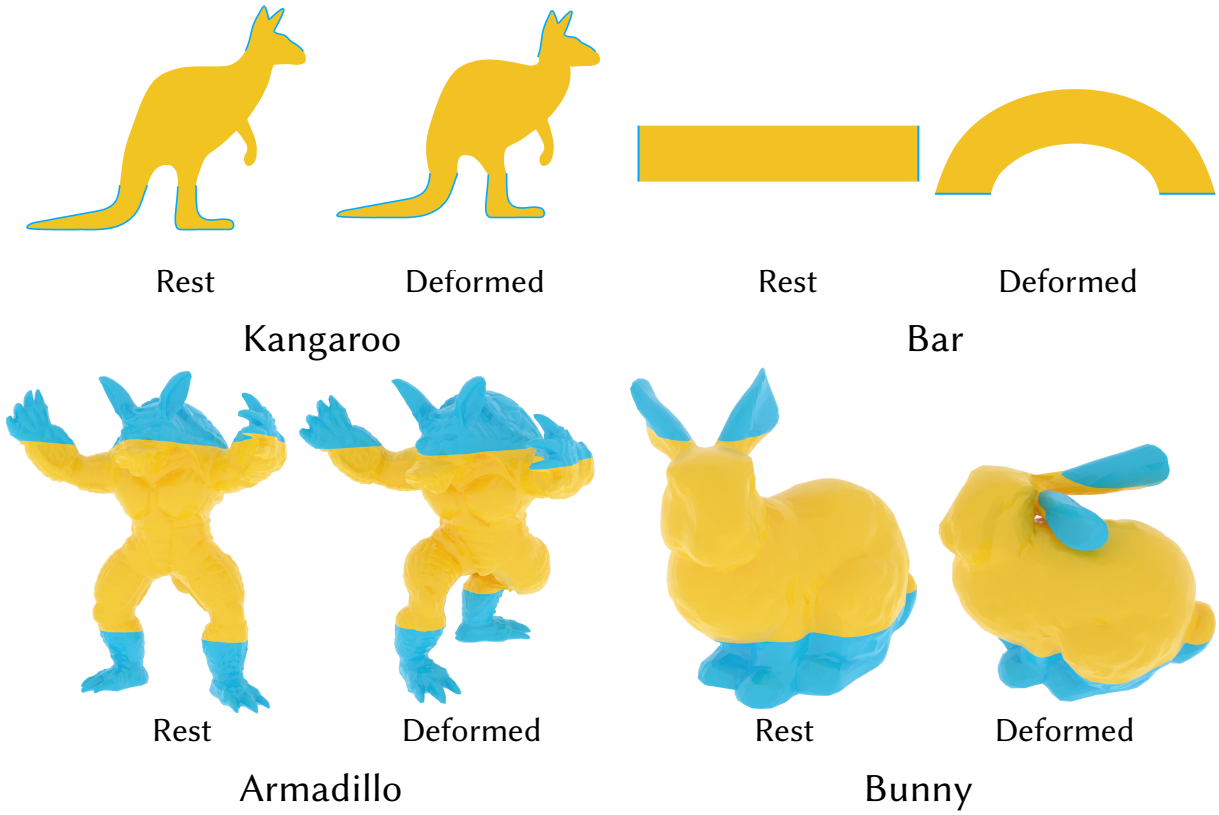


Figure 3.6: Initial and final frames of simulations from which queries are exported. The parts in cyan are used as handles to twist and compress the model while the deformation occurs in the parts in yellow.

3.6.1 Benchmark of filtered queries

We collect queries of element inversion checks from elastodynamic simulation data using Incremental Potential Contact (IPC) [88] and the Neo-Hookean elasticity model. We pick two 2D models and two 3D models shown in Figure 3.6. We bend the bar and compress the kangaroo in 2D, whereas for 3D models we twist them by 90° while compressing them by 20%. Note the queries included in the benchmark are a subset of the entire simulation, since the vast majority of elements are valid. For collection, we discard all queries where the initial configuration is invalid, which might happen as the positivity check in IPC [88] is not conservative; in each line search in the nonlinear solves, we evaluate the Jacobian at every quadrature point J_i for $i = 1, 2, \ldots, n$ and collect elements with elements with $\min_i J_i \leq 0$ or $\min_i J_i / \max_i J_i < 0.2$.

Our benchmark contains 172000 2D queries from the Kangaroo model (orders 1 through 4),

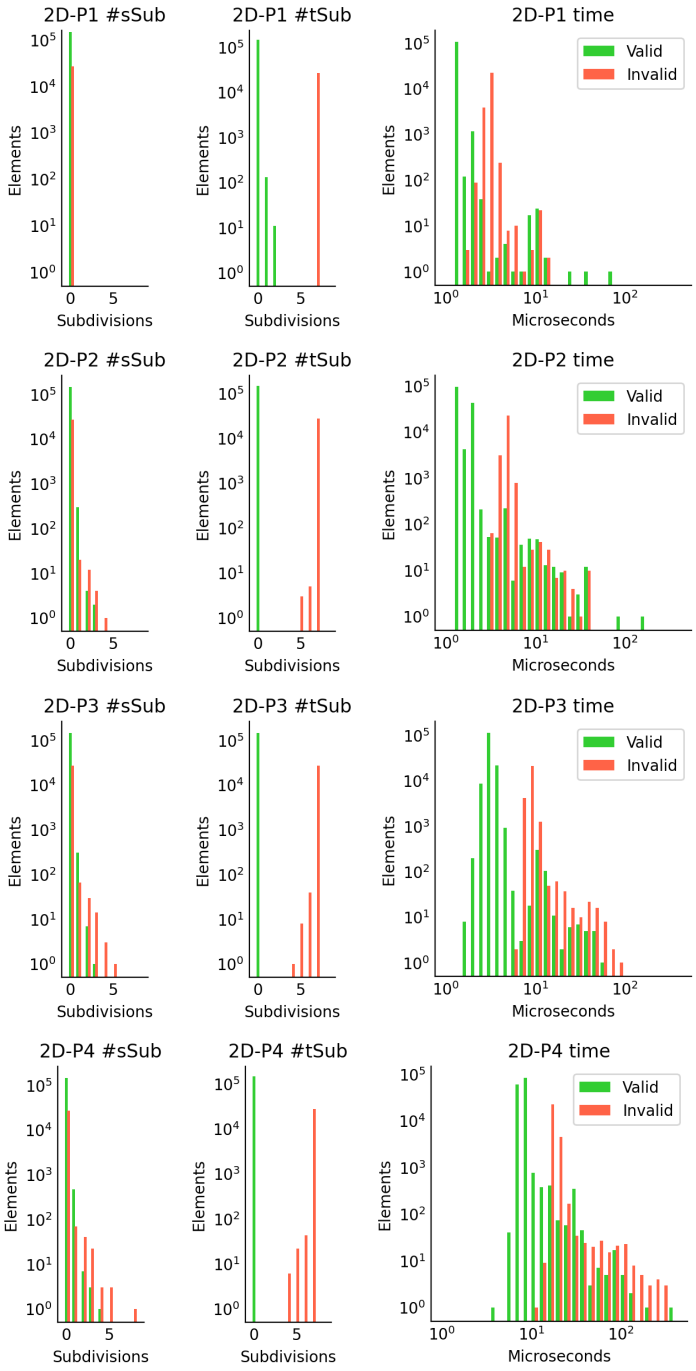


Figure 3.7: Statistics for the Kangaroo datasets. Top to bottom: elements of order 1, 2, 3, 4; Left to right: number of space subdivisions (#ssub), time subdivisions (#tsub), and time to test an element. Green valid elements; red invalid ($t^* < 1$) elements.

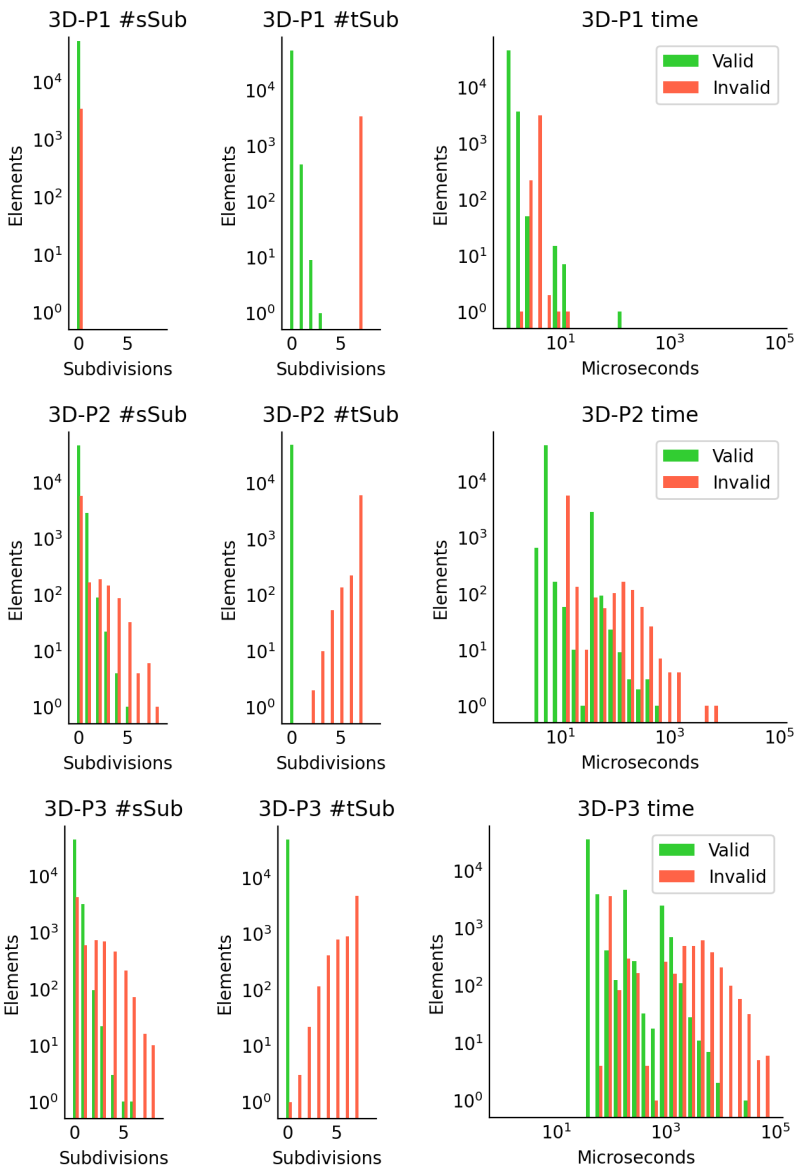


Figure 3.8: Statistics for the Armadillo datasets. Top to bottom: elements of order 1, 2, 3; Left to right: number of space subdivisions (#ssub), time subdivisions (#tsub), and time to test an element. Green valid elements; red invalid ($t^* < 1$) elements.

54985 3D queries from the Armadillo model (orders 1 through 3), 19800 3D queries from the Bunny model (orders 1 through 3), 130291 queries for the Bar model of order 1, and 31879 queries for the Bar model of order 2 (Table 3.2).

To validate correctness, we designed a Wolfram Mathematica [141] script that symbolically computes the root of $|J_{\overline{x}}|$ with minimum t for a given element, and released it as part of our open-source code. The script receives in input the type, dimension n, and order p of an element together with the control points describing its geometric map at times $t = 0$ and $t = 1$; computes the symbolic expression of $|J_{\overline{x}}|$, which is a polynomial in $n + 1$ variables; finds its roots inside the domain $\overline{\sigma}$; and returns the minimum value of t in which a root is found.

We restrict the ground truth to order 1 and 2 for triangles, and order 1 for quads and tetrahedra, due to the limitations of symbolic solvers: for a higher-order basis, Mathematica could not return a result within 6 hours on some of the elements. To the best of our knowledge, ours is the first dataset containing conservative times of inversion. Our algorithm correctly detects all invalid elements and returns conservative answers for the inversion times.

The average per-element cost of our algorithm increases with the degree, from around 1.5μs for order 1 (both in 2D and 3D) to around 10μs for order 4 in 2D and 300μs for order 3 in 3D. Invalid elements are more expensive to process on average, as they require subdivisions until precision is reached, whereas most valid elements can be resolved in a single iteration if all their Bézier coefficients are positive. See Table 3.2 and Figures 3.7 and 3.8 for details.

When restricted to the static case (i.e., check the validity of an element at a given time), our algorithm becomes considerably faster than the dynamic one (by a factor of about $50\times$ on the order 3 Armadillo), and it has a running time slightly faster ($2\times$) than the non-conservative static baseline used in PolyFEM and FEBio, which consists of checking the sign of $|J_x|$ obtained with (inexact) floating point computations only at quadrature points thanks to our optimizations. See Table 3.3 for details.

3.6.2 Comparisons

We are not aware of any other algorithm that provides a continuous validity check for elements of arbitrary order, so we compare it with algorithms that solve only a subset of the problem.

3.6.2.1 Linear continuous

For the special case of triangular elements with linear basis, Smith and Schaefer [124] propose to use a symbolic solver to find the roots of $|J_f|$. While extremely efficient (1μs on average), this approach can fail to produce correct results due to numerical errors: on the 26552 invalid elements in the linear Kangaroo dataset (Figure 3.6), their method fails to detect inversions

Table 3.3: Comparison of methods for static validity checks, performed at $t = 1$ on the order 3 Armadillo dataset. We list the number of detected valid and invalid elements, the number of elements for which the test was undecided, and the average and median computation times per element. Sampling at quadrature points is fast, but incorrectly classifies several invalid elements as valid; bisection with a robust interval-based inclusion function is fast on "easy" elements, but struggles a lot with nearly-inverted elements and fails to classify several of them; our implementation of the Bézier refinement based inclusion check by Johnen et al. [69] and our conservative method give the same results; our method is guaranteed correct at a slight performance cost. Our precomputations (last two rows) decrease the computation time by at least an order of magnitude.

Static algorithm	#val	#inv	#und	avg μs	med μs
Quadrature Points	48214	6771	-	16	16
Interval Bisection	46281	6561	2143	1400	23
FP Bézier (no optim.)	48050	6935	0	71	78
Ours (no optim.)	48050	6935	0	86	95
FP Bézier	48050	6935	0	5	5
Ours	48050	6935	0	8	8

13324 times, producing a time step larger than the ground truth maximum. Attempting to be conservative with a "large" numerical threshold of 10^{-5} still fails in 1563 tests ($\sim 5\%$).

This method is limited to linear triangles, and cannot be extended to other elements or degrees due to its reliance on closed-form expressions for the roots.

3.6.2.2 Static high-order

For the special case of static validity check for elements of arbitrary type and order, Johnen et al. [69] introduce a method based on adaptive subdivision. In the static case, our solution implements the same algorithm, but with robust arithmetic and additional pre-computations for the Jacobian and transformation matrices. When implemented with double precision floating point arithmetic, this method is more efficient than our static approach (on the 3D Armadillo model of order 3 our method takes $3\mu s$ longer per element on average, see Table 3.3), however it is not robust.

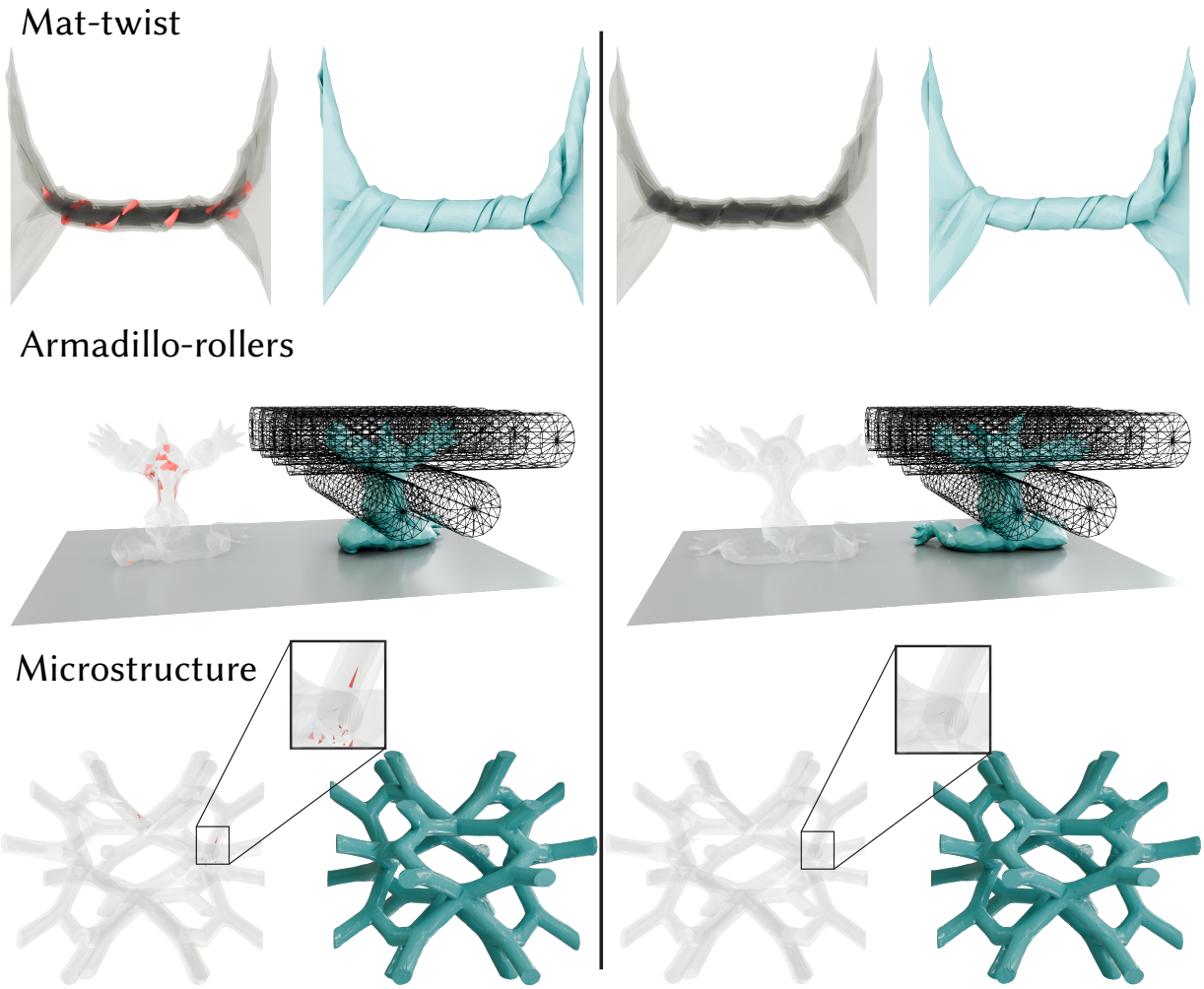


Figure 3.9: Simulation examples in [43] with (right) and without (left) our inversion check. The high-order elements with non-positive Jacobian points are shown in red. On the left, the numbers of flipped elements are 20, 45, and 24 from top to bottom. Our method guarantees the positivity of Jacobian.

Table 3.4: Simulation Statistics. Columns from left to right: simulation dimension, geometric and solution bases orders, number of cells, number of time steps, peak memory usage, the average time of the simulation per time step, the average time of our check per time step, the average time of each check over the entire mesh, and average time of the simulation per time step using the quadrature point check in PolyFEM instead of ours (BL=baseline)

		Order				Mem	Timing (sec)			
	dim	geom	soln	# Cells	# Steps	(GB)	tot/step	check/step	check/query	tot/step (BL)
Beam-twist	3	1	2	1740	400	1.4	16	5.7	0.067	7.6
Ring-twist	2	1	2	1136	100	0.7	0.72	0.16	0.011	0.15
Mat-twist	3	1	2	2166	250	1.3	10	2.9	0.069	6.9
Armadillo-rollers	3	1	2	5978	230	3.3	114	27	0.19	111
Microstructure	3	4	2	6414	25	2.9	531	472	5.6	52

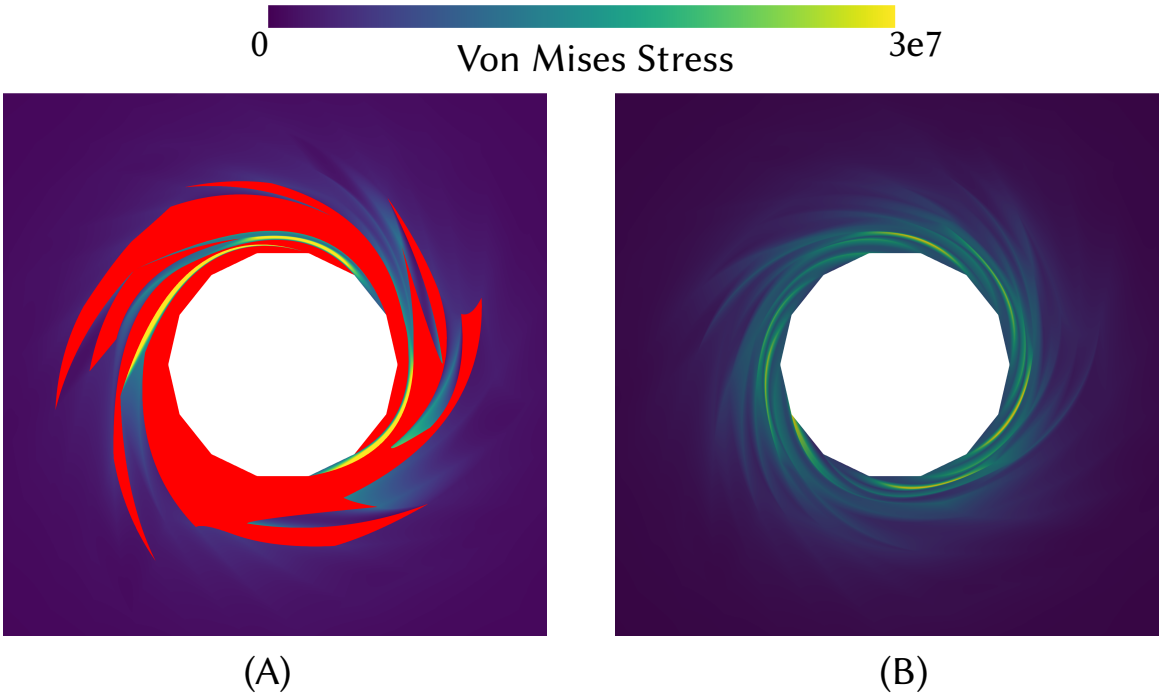


Figure 3.10: Von Mises stress distribution of Ring-twist in Figure 5.1. The NaN is shown in red. (A) Without our method, NaN appears on elements with flipped points. (B) With our Jacobian check and adaptive quadrature, the stress is everywhere finite.

3.6.3 Elastodynamic simulation

We integrate our check within the PolyFEM software [116] and use it to reproduce a selection of the bundled elastodynamic simulations. We report our findings as we integrated the check, since they highlight some fundamental issues with existing high-order FE solvers and the non-linear material models commonly used in graphics and engineering.

3.6.3.1 Potential formulation

In the simulation, we minimize the Neo-Hookean energy with Newton's method. The Neo-Hookean energy density has the form of

$$w_e(F) := \frac{\mu}{2}(\text{Tr}[FF^T] - 2 - 2\log(\det F)) + \frac{\lambda}{2}\log^2(\det F), \tag{3.5}$$

where F is the deformation gradient matrix (2×2 or 3×3), λ and μ are Lamé parameters. Call J_d and J_r the Jacobians of the deformed element and rest element respectively, the deformation

gradient is

$$F = J_d(J_r)^{-1}.$$

Suppose the rest element is valid, i.e., $\det J_r > 0$; then the positivity of $\det F = \det J_d / \det J_r$ depends solely on $\det J_d$, i.e. the Jacobian determinant that we are checking. As long as $\det J_d > 0$, Equation (3.5) is valid.

3.6.3.2 Baseline simulator

Our baseline simulator is PolyFEM [116], using the convergent IPC formulation [90] on tetrahedral meshes, quadratic Lagrangian bases and the Neo-Hookean material model [112]. PolyFEM uses only a static inversion check on the quadrature points: in Figures 3.9 and 5.1 we show that the final result contains many invalid elements leading to NaN in stress (Figure 3.10A). These invalid elements are not detected using the check on quadrature points alone, but are correctly identified by our conservative check (Figure 3.10B).

3.6.3.3 Conservative line search only

Replacing the static sampling invalidity check with our conservative and continuous check without adaptive quadrature leads to convergence issues in the solver. When the elements become close to inversion, the negative gradient direction is not a descending direction. This happens due to the error in the numerical integration of the potential, that can be solved by using our adaptive quadrature method. All the simulations in our experiments suffer from this issue: the simulation halts after several steps (twist-beam fails at time step 145, armadillo-rollers at 33, mat-twist at 141, ring-twist at 28, and the microstructure at 7).

3.6.3.4 Conservative line search + adaptive quadrature

This solution works but dramatically increases the number of non-linear iterations needed in the Newton method. The remaining issue is that, while the Neo-Hookean energy density is infinite at the point where the Jacobian determinant is zero, it grows at a slow rate. Mathematically, when the Jacobian determinant is exactly zero at one point and positive everywhere else, the integral of the Neo-Hookean energy still may be finite: consider the integral of a 1D function $f(x) = -\log(|x|)$ on the interval $[-1, 1]$, although it has a singularity at $x = 0$, the integral is

$$\int_{-1}^{1} -\log(|x|)dx = -2\int_{0}^{1} \log(x)dx = 2 < \infty$$

This is undesired since it leads to NaN in Equation (3.5) and stress evaluation, causing failure of the simulation. This problem can be mitigated by adding an additional barrier term $\frac{\mu}{\det F}$ to

Equation (3.5). While this solution formally fixes the problem (and also practically fixes it in our experiments), it makes the potential harder to minimize while changing the material model: we are not aware of any analysis of this problem, and we believe it is an exciting avenue for future work.

With these modifications, PolyFEM produces results without invalid elements. We note that given that the material models are infinite for invalid elements, the existing non-conservative approaches often create solutions that are non-physical. A more detailed study of the effect of these errors could be an interesting venue for future work.

A second observation is that the issue with the slow growth of the Neo-Hookean potential (and others, such as Mooney-Rivlin) is likely due to the misuse of these models for deformations that are outside of the regime they are designed to handle. It would be interesting to carefully study experimentally how accurate these material models are under extreme deformations, and see if it is possible to design other material models which are numerically more suitable for interior point optimization.

3.6.3.5 High-order IPC

We reproduce simulation examples that use high-order finite elements in Ferguson et al. [43], including Mat-twist, Armadillo-rollers, and Microstructure (Figure 3.9). In Figure 5.1, we include two more examples: Beam-twist and Ring-twist. In the Beam-twist, we apply Dirichlet boundary conditions on the two sides of the beam, rotate one side, and keep the other side fixed; in the Ring-twist, we apply Dirichlet boundary conditions to rotate the inner circle of the ring with constant speed and allow the outer circle free to move. In Figure 3.9, the simulation results in Ferguson et al. [43] have flipped elements since the simulator only checks Jacobian at quadrature points in each element, while with our conservative check and adaptive quadrature, there is no flipped element. We report the statistics in Table 3.4. For quadratic elements, the runtime of our method is at worst comparable to the solve time and can be as fast as 23% of the solve time; for quartic geometric elements and quadratic solutions, our method is much slower than the solve time, since the solve is on quadratic elements while the Jacobian check should be performed on the quartic elements.

3.7 Discussion

We introduced a formulation for continuous inversion test and a corresponding conservative and efficient algorithm. Our solution addresses an open problem in existing finite element solver and parametrization algorithms, increasing robustness and providing, for the first time, a guarantee for interior point solvers to stay within the space of valid elements. While the issue does appear for linear elements, invalid elements are more commonly present in existing algorithms

that use high-order bases. We believe our algorithm, and its reference implementation, will be a drop-in replacement for existing non-conservative checks used in many graphics algorithms that will increase robustness for a limited performance cost.

Chapter 4

MiSo: Automatic Generation of Conservative Solvers

In this chapter, we discuss the generalization and automation of the conservative approach we used to address the geometrical validity problem, extending it to other related problems, namely approximating the solution set of systems of non-linear constraints and bounding the constrained global minimum of a non-linear function. As previously mentioned, there is an abundance of geometric queries that can be formulated in these terms, with countless variations and a vast literature: for example, [5, 6] summarize methods for static collision detection between proxies, referencing over 100 algorithms tailored to different primitive pairs and accuracy/efficiency trade-offs. Similar per-primitive-pair specialization is required for minimal distance queries and, likewise, each finite element (FE) type and order necessitates custom code for positive Jacobian checks. This complexity doubles when considering time-dependent scenarios.

While real-time applications often restrict primitives to boxes due to limited computational resources, high-fidelity simulations may require conservative, high-accuracy predicates [125]. Testing the correctness and ensuring the efficient, accurate implementation of these algorithms is a major challenge [137]. The difficulty of generalizing theoretical improvements across different cases hinders progress in this pervasive and crucial family of algorithms, essential to modern computing.

In contrast to algorithm specialization, Snyder [125] proposed a general framework, based on interval analysis, for conservative solutions to high-order constrained optimization. This framework offers two algorithms: Solve, which finds all solutions to a non-linear constraint system, and Minimize, which finds the constrained global minimum of a function. For Solve, the conservative algorithm returns a region guaranteed to contain all solutions (if any), potentially including points near the feasible domain. For Minimize, it returns a value less than or

equal to the true minimum and within a bounded distance of it. In both cases, this conservativeness accounts for numerical rounding errors.

Although often considered slower than methods like Newton's minimization, recent work [137, 24] demonstrates the effectiveness and relative efficiency of this conservative approach, particularly when seeking guaranteed solutions.

Snyder's approach uses Natural Interval Extensions (NIE) to compute *inclusion functions* (Section 3.2.3) that bound function ranges over domains, by composition of interval operators. Although general, NIE's convergence to the true range via domain decomposition can be slow. For the common case of polynomials, tighter bounds are achievable via their Bézier representation [84, 129, 68]. However, Bézier representation can be computationally expensive for polynomials with many terms.

We employ a hybrid approach, blending Bézier inclusion functions and NIE. Decomposing polynomial expressions into simpler forms (fewer variables or lower degree) allows us to construct a spectrum of inclusion functions that ranges from fully NIE-based (expanded expressions) to fully Bézier-based (collapsed expressions). This decomposition is performed manually by the user via the `collapse` method in the MiSo specification (see Sections 4.3.2 and 4.4.2); the DSL makes it convenient to explore different hybrid decompositions, but does not automate the choice. Hybrid solutions can dramatically improve efficiency. For example, our hybrid solver for continuous collision detection between high-order polynomial patches is orders of magnitude faster than purely NIE-based and purely Bézier-based solutions (Section 4.4).

We developed MiSo on top of such a hybrid approach. MiSo is a Python-based domain-specific language (DSL) for the specification of Solve and Minimize problems. MiSo enables the user to quickly explore possible hybrid approaches by changing a few lines of code. From a simple specification, the MiSo compiler produces a numerically robust C++ solver for the given problem, automatically generating all the necessary representations of the functions involved, the related transformations required for domain subdivision, and the evaluation of inclusion functions.

Domain decomposition and interval arithmetic are used to guarantee conservative results. Setting a compile-time flag switches to a faster, non-conservative computation mode based on standard floating-point arithmetic. A known limitation of subdivision-based methods is that they suffer from a curse of dimensionality; hence, our method may become impractical for problems in many dimensions. However, we show that we are able to achieve competitive performance for a number of fundamental geometric problems, especially those involving high-order geometry.

We demonstrate competitive performance against hand-optimized code for key geometric problems, including linear and high-order continuous collision detection, and finite element validity checks.

Table 4.1: A list of common problems that can be addressed with MiSo. From the left: name of the problem; domain; coordinates on the domain; objective function; constraint. Legend: GV: (static) geometrical validity; CGV: continuous geometrical validity; CCD: continuous collision detection; PP$\cap$: parametric primitive intersection; MD: minimal distance; DP: diameter of primitive; B$\cap$: boolean intersection; B$\cup$: boolean union; B$\setminus$: boolean difference. The symbol σ represents a generic Cartesian product of standard simplices, which is the parametric space of a geometric primitive; the domains of different elements are denoted with subscripts; a repeated symbol refers to the same domain considered more than once. The symbols ξ and η represent tuples of coordinates referring to the related factors in the Cartesian product; t is a scalar coordinate; the symbols ξ_1, ξ_2 refer to tuples of coordinates representing two distinct points in the same domain. The symbol x, possibly with subscripts, represents a geometric map from a parametric space into physical space; J_x is its Jacobian; likewise, $\overline{x}$ represents a time-dependent geometric map. The symbols D_i represent the SDF of an implicitly defined primitive.

Problem	Algorithm	Domain Σ	Coordinates	Objective F	Constraint C (≤ 0)
GV	Solve	σ	ξ	-	$\lvert J_x(\xi)\rvert$
CGV	Minimize	$\sigma \times [0,1]$	(ξ, t)	t	$\lvert J_{\overline{x}}(\xi, t)\rvert$
CCD	Minimize	$\sigma_1 \times \sigma_2 \times [0,1]$	(ξ, η, t)	t	$d(\overline{x_1}(\xi, t), \overline{x_2}(\eta, t))$
PP$\cap$	Solve	$\sigma_1 \times \sigma_2$	(ξ, η)	-	$d(x_1(\xi), x_2(\eta))$
MD	Minimize	$\sigma_1 \times \sigma_2$	(ξ, η)	$d(x_1(\xi), x_2(\eta))$	-
DP	Minimize	$\sigma \times \sigma$	(ξ_1, ξ_2)	$-d(x(\xi_1), x(\xi_2))$	-
B$\cap$	Solve	$[0,1]^n$	ξ	-	$\max(D_1(\xi), D_2(\xi))$
B$\cup$	Solve	$[0,1]^n$	ξ	-	$\min(D_1(\xi), D_2(\xi))$
B$\setminus$	Solve	$[0,1]^n$	ξ	-	$\max(D_1(\xi), -D_2(\xi))$

4.1 Overview

We begin by characterizing the class of addressed problems, and follow with a broad description of MiSo's operation and a working example, deferring details to Sections 4.2 and 4.3.

4.1.1 Problem statement

We now formulate the Solve and Minimize problems in the continuum, and then transition to their discrete numerical counterparts, discussing the choices and limitations inherent in this process.

Definition 6 (Solve Problem)**.** *A* Solve Problem *is defined by:*

a) A domain Σ *that is the Cartesian product of standard simplices, possibly of different dimensions;*

b) A system of constraints*, consisting of inequalities of the type $C_i(\xi) \leq 0$ with $\xi \in \Sigma$ for $i = 1, \ldots, k$, where all C_i's are continuous scalar functions. Any equality constraint $C_i(\xi) = 0$ is represented as $|C_i(\xi)| \leq 0$.*

The problem asks to find the region $\Phi \subseteq \Sigma$ satisfying the constraints, called the feasible region*.*

Definition 7 (MINIMIZE Problem)**.** *A* MINIMIZE Problem *is defined by:*

a) and b) *as in Definition 6;*

c) A continuous scalar objective function *$F(\xi)$ defined on Σ.*

The problem asks to find the minimum F^* of F within the feasible region *Φ.*

Table 4.1 shows a few relevant geometric problems that fit our framework. Some MINIMIZE problems are unconstrained, meaning their feasible region is the whole domain Σ.

The domain Σ is chosen to facilitate modeling geometric problems. Although products of unit intervals provide compact subsets of Cartesian space, higher-dimensional simplicial domains are better suited for representing geometric elements. While compact domains theoretically limit support for unbounded primitives (e.g., rays or planes), this is rarely a practical concern, as bounded approximations suffice. See Section 4.2.1 for details.

We address numerical versions of the above problems and seek a conservative solution. This means that, even when an exact solution cannot be found numerically, we return a solution, which is guaranteed to be within a certain threshold from the ground truth. This is formally stated in the following definitions.

Definition 8 (ε-SOLVE Problem)**.** *An ε-*SOLVE Problem *is defined by the same elements of a* SOLVE *Problem, plus:*

c) An array of thresholds $\varepsilon_i > 0$ for $i = 1, \ldots, k$.

Let $\Phi_\varepsilon \subseteq \Sigma$ be the region satisfying $C_i(\xi) \leq \varepsilon_i$ for all $i = 1, \ldots, k$, called the buffer region*. A solution to the problem is any region $\tilde{\Phi}$ such that*

$$\Phi \subseteq \tilde{\Phi} \subseteq \Phi_\varepsilon$$

where Φ is the solution to the corresponding SOLVE *Problem.*

In the following, most examples involve just one constraint equation: in that case, we drop the subscript on C and ε, for simplicity.

Figure 4.1 depicts the relation between regions Φ, $\tilde{\Phi}$ and Φ_ε. The region $\tilde{\Phi}$ found by our solver consists of subdomains of the same shape as domain Σ, obtained by domain subdivision, hence the staircase shape in the figure.

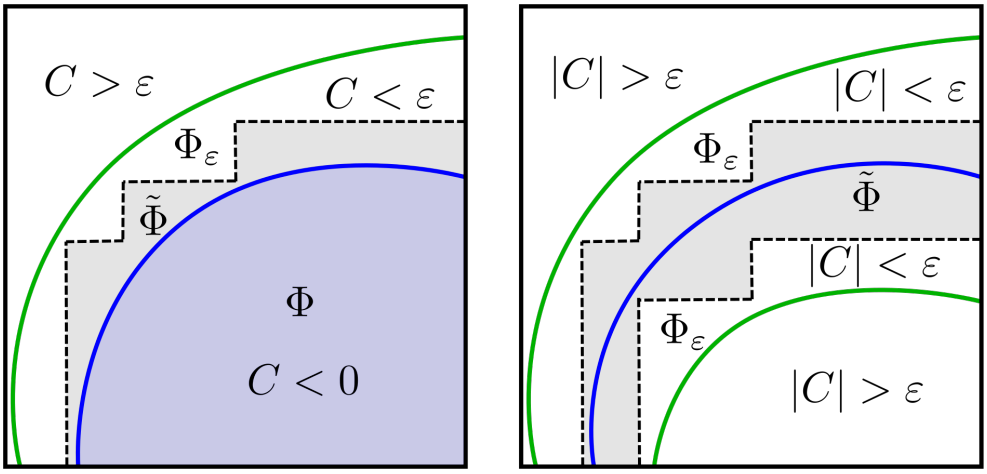


Figure 4.1: Regions defining a solution for ε-Solve Problems on a domain $[0,1] \times [0,1]$ with an inequality constraint (Left) and an equality constraint (Right). Left: the blue-shaded region below the blue line is the solution Φ of the corresponding Solve Problem in the continuum; the region Φ_ε extends between the blue and green lines providing a buffer in which the numerical solution is sought; the grey-shaded region below the black polyline is a possible solution $\tilde{\Phi}$. Right: Φ consists just of the blue line; a solution $\tilde{\Phi}$ is the grey-shaded region enclosed between the two black polylines.

Definition 9 (ε, δ-MINIMIZE Problem)**.** *An* ε, δ-MINIMIZE Problem *is defined by the same elements of a* MINIMIZE *Problem, plus:*

d) Thresholds $\varepsilon \geq 0$ *for* $i = 1, \ldots, k$ *and* $\delta > 0$.

A solution to the problem is any interval $\tilde{F}^*$ *with a width* $\leq \delta$ *such that:*

- *the lower end of* $\tilde{F}^*$ *is less or equal to the minimum* F^* *of* F *on the feasible region* Φ*;*
- *the upper end of* $\tilde{F}^*$ *is greater or equal to the minimum* F^*_ε *of* F *on the buffer region* Φ_ε*.*

Figure 4.2 shows an example of an ε, δ-MINIMIZE problem in 2D: the (first) intersection between an oriented line segment and a curve.

4.1.2 Discussion of the ε, δ-MINIMIZE problem when $\varepsilon > 0$

We note that the solution of the ε, δ-MINIMIZE problem might not contain the solution of the corresponding MINIMIZE problem when $\varepsilon > 0$. In our case, we approach the problem numerically, so an intrinsic difficulty arises when considering equality constraints - it is impossible

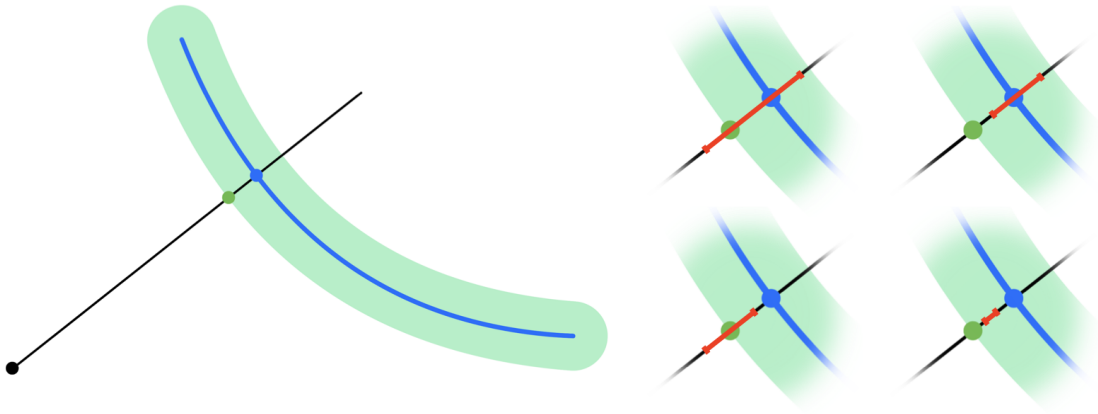


Figure 4.2: An ε, δ-Minimize Problem: the first intersection between the black segment and the blue line is sought. The segment starts at the black bullet. The blue bullet corresponds to solution F^* of the corresponding Minimization Problem in the continuum. The green bullet corresponds to the solution for region Φ_ε, whose image is the green area. The red segments in the blow-ups to the right correspond to possible intervals $\tilde{F}^*$ returned as numerical solutions. Their lower ends are guaranteed to lie below the blue bullet and their upper ends are guaranteed to lie beyond the green bullet. The different configurations arise from the interplay between the values of ε and δ: as ε tends to zero, the upper left configuration becomes more frequent; as δ tends to zero, the lower right configuration becomes more frequent.

to guarantee that a subregion $\sigma \subset \Sigma$ contains 0 without additional assumptions, because we cannot compute the exact range of F on σ in general. Furthermore, with finite precision arithmetic, it is impossible to verify that a single point P satisfies the equality constraint in general. Therefore, the Minimize problem of Definition 7 does not always admit a numerical solution. However, it is possible to certify, via conservative computations, that σ does *not* intersect Φ, and that P does not lie in Φ.

For these reasons, we are always able to guarantee that the lower bound of $\tilde{F}^*$ bounds F^* from below, but we can only bound it from above when Φ does not have zero measure in Σ. Setting $\varepsilon > 0$ in such cases constitutes a meaningful approximation of the equality-constrained problem.

The common practice in the state of the art is to simply solve the relaxed problem without equality constraints, i.e., with $\Phi \equiv \Phi_\varepsilon$. Compared to this approach, we offer the same bound from above and a tighter bound from below.

4.1.3 Precision thresholds

We prescribe thresholds on precision in the *range* of the functions C_i and F, rather than the more common practice of setting thresholds in their *domain* Σ. While slightly more complex, this approach allows for greater flexibility and improved error control. As detailed in Section 4.3.5, our implementations of SOLVE and MINIMIZE are readily adaptable to specific application requirements, and managing precision in parameter space only requires modifying a few lines of code.

4.1.3.1 Precision in the domain

In Definition 8 we could define Φ_ε as a region containing Φ with all the points within distance ε of Φ. Similarly, in Definition 9 (assuming $\delta = \varepsilon$) the solution could be defined as a volume $\tilde{\sigma}^* \subset \Sigma$ of size at most ε in all its coordinates, containing an inverse image of F^* (multiple minima with value F^* may exist in Φ). However, $F(\tilde{\sigma}^*)$ must be evaluated to explicitly approximate F^*.

Although this approach simplifies the definitions and algorithms, it provides limited information about the accuracy of the solutions relative to the true feasible region and minimum. Requiring only minor code modifications, we employ it in Section 4.4.2 solely for comparison with [24] to comply with their choice.

4.1.3.2 Precision in the range

As stated in Definition 8, the buffer region Φ_ε depends on the C_i's and contains Φ but, other than that, its shape is not directly related to the feasible region Φ. Instead, we impose an explicit bound on how much the C_i's may violate the constraints.

In Definition 9, having distinct thresholds ε, δ is necessary because in general neither the C_i's nor F can be evaluated exactly, and their ranges are unrelated. If $\varepsilon_i > 0$ for at least one index i, then we have $F^*_\varepsilon \leq F^*$; the solution $\tilde{F}^*$ may span either or them, or both, or lie between them, possibly shrinking to a singleton (see Figure 4.2). If $\varepsilon_i = 0$ at all i, then $F^*_\varepsilon = F^*$ hence $F^* \in \tilde{F}^*$.

If Φ has volume in Σ, the bounds on F^* can be tightened as the search shrinks around Φ even when $\varepsilon = 0$, and the solver can stop as soon as the interval about F^* is tight enough. On the other hand, if Φ has measure zero in Σ (as in Figure 4.1 Right and Figure 4.3), which is usually the case in the presence of equality constraints, it may be impossible to decrease the bound on F^* from above with $\varepsilon = 0$: the solver would indefinitely shrink the search about Φ without converging, and only an additional termination condition can provide an answer [125].

If all the C_i's and F are Lipschitz continuous, domain and range precision are equivalent, differing only by their Lipschitz constants. However, determining these constants and balancing domain tolerances can be challenging.

4.1.4 Discontinuous functions

The algorithms presented in Section 4.2 to resolve the numerical problems produce conservative results even if the constraint functions C_i and the objective function F are discontinuous, but they may fail to provide a result within the required thresholds ε, δ. The treatment of non-continuous functions is a subtle issue in floating-point: since a function f can be only sampled at finitely many points and evaluated up to a finite precision, all discontinuities between adjacent samples can be filled with steep ramps, making the function virtually continuous. The concept of a *convergent inclusion function* can be extended to discontinuous functions. For a multi-interval $A \in \mathbb{I}^n$, a function f (possibly discontinuous) defined on A, and a point $p \in A$, an inclusion function $\square f$ for f is convergent if

$$\lim_{A \to \{p\}} \square f(A) = [\liminf_{\xi \to p} f(\xi), \limsup_{\xi \to p} f(\xi)],$$

If f is continuous at p, this definition is consistent with Definition 3. However, if p is a discontinuity, $\square f(A)$ cannot shrink beyond the jump of f at p.

If a discontinuity point p is significant – e.g., F has its minimum at p, or a constraint C_i has a zero-spanning jump at p – and the jump is larger than the given thresholds ε, δ, our algorithms will halt after $K_{\max}$ domain subdivisions around p without reaching the required precision. The returned result is still conservative, though. This can also occur with continuous functions if ε and δ are too small compared to the gradient of the constraint/objective functions. In these cases, too many (possibly infinite) subdivisions would be necessary to converge, and our algorithms would undergo an early exit after a given maximum number of subdivisions $K_{\max}$. Indeed, due to the inherent discretization of domain and range in floating-point arithmetic, distinguishing between functions with extremely high gradients and discontinuous functions becomes practically impossible.

As discussed in Section 4.1.3.1, we can alternatively define the precision thresholds in the domain, rather than in the range. In this case, the algorithms always converge, and the maximum level of domain decomposition is defined by the thresholds themselves.

4.1.5 How MiSo works

MiSo is used to specify a problem in the classes given in Definitions 8 and 9 and automatically generate a solver for such a problem.

All solvers produced by MiSo are instances of two generic solvers, which rely on interval analysis [125]: they evaluate the inclusion functions of the constraint and objective functions over sub-domains of Σ, and perform space decomposition to converge to the solution.

A MiSo specification consists of a Python script that translates the mathematical specification of an ε-Solve or ε, δ-Minimize Problem into C++ functions. These functions take problem-specific parameters – $(\mathbf{C}, \varepsilon)$ for ε-Solve and $(\mathbf{C}, F, \varepsilon, \delta)$ for ε, δ-Minimize, where $\mathbf{C}$ denotes the collection of C_i's – and return the problem solution. The solution to an ε-Solve problem is a collection of non-intersecting sub-domains of Σ, resulting from recursive subdivisions, whose union forms the region $\tilde{\Phi}$. The output of an ε, δ-Minimize problem is an interval $\tilde{F}^*$ defined by two floating-point values.

We provide two backends for our code generator: (1) one prioritizing correctness, employing rounded interval arithmetic to guarantee a conservative answer, and (2) one prioritizing efficiency, which performs computations with standard floating point arithmetic whenever possible. The former is 2-3 times slower on average but provides provably conservative results, which is a useful feature for certain problems such as collision detection. The choice of the backend is a compilation flag, and the specifications need not be modified (Section 4.3.5).

4.1.6 Didactic example: line-surface intersection

We consider the problem of finding the intersections between a cubic triangle patch and a segment, both given in parametric form and show how it can be solved using our method. Other instances of this problem, involving other types of surfaces and/or curved trajectories, will be discussed in Section 4.4.

The mathematical specification of the problem can be given as follows:

a) The domain is $\Sigma = \triangle^2 \times \triangle^1$ where: $\triangle^2$ is the two-dimensional standard simplex, used as the parametric domain of a cubic triangle; and $\triangle^1$ is the unit interval, used as the parametric domain of a straight-line segment. Σ is a triangular prism and its coordinates are (u, v, t).

b) Let $S_{2,3} : \triangle^2 \longrightarrow \mathbb{R}^3$ and $r_{1,1} : \triangle^1 \longrightarrow \mathbb{R}^3$ be the parametrizations of a generic cubic triangular patch and a generic segment, respectively. We define

$$C(u, v, t) = \|S_{2,3}(u, v) - r_{1,1}(t)\|_2^2$$

the L_2 distance between a generic point of the triangular patch and a generic point of the segment. The problem's constraint is defined by $C(u, v, t) \leq 0$; this is equivalent to $C(u, v, t) = 0$, but we will relax this to $C(u, v, t) \leq \epsilon$ to avoid having a measure-zero feasible region, as discussed in Section 4.1.2.

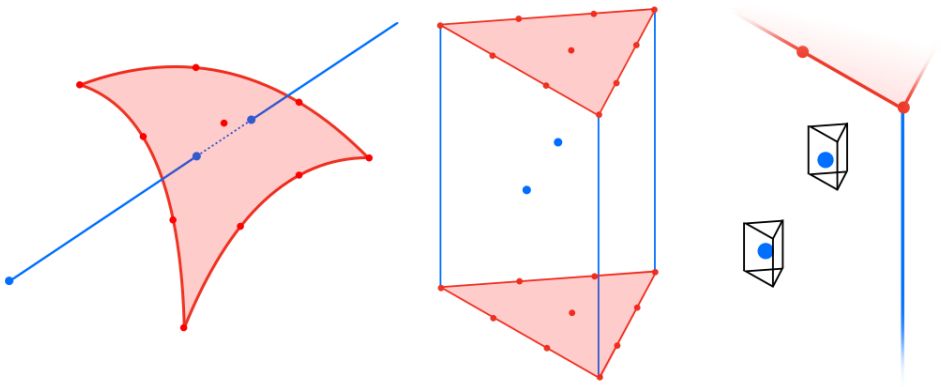


Figure 4.3: Intersection between a straight-line segment and a cubic triangular patch. Left: in physical space, the origin of the segment is to the bottom left; the red bullets are the Lagrange control points defining the patch; the Minimize Problem seeks the first (leftmost) of the two intersections (blue bullets). Center: the problem domain Σ is an upright triangular prism; the blue bullets are the inverse images of the intersections in the left image and give the solution of the Solve Problem; the T (blue) coordinate of the lowest of the two points is the solution of the Minimize Problem. Right: the ε-Solve Problem returns two tiny volumes enclosing the blue bullets; the ε, δ-Minimize Problem returns an interval spanning the T coordinate of the lower-left blue bullet.

c) The objective function is $F(u, v, t) = t$, whose minimum t^* gives the first point $r_{1,1}(t^*)$ along the oriented segment to intersect the patch.

d) The precision thresholds for the numerical problem are values $\varepsilon > 0$ (tolerance for the inequality constraint on distance) and $\delta > 0$ (required precision for time).

Figure 4.3 depicts the setting of the problem in physical space, the domain Σ, and its solution in the continuum. In the numerical version, each possible intersection point consists of a region of physical space containing that point; likewise, its inverse image in the prism will be a volume and the sought solution will be an interval (upright segment in the prism) stabbing such a volume.

Figure 4.4 contains a MiSo specification of the problem together with some results of the corresponding Solve problem.

MiSo is implemented as a Python library called `pymiso`. All MiSo code must be placed within a `pymiso.Context` object, created at line 1 using Python's `with` construct and assigned a name; all subsequent MiSo calls are done from this object.

In Line 2, the two parameters from the triangle's parametric domain are created and assigned to `X`. Because they are part of the same simplex (thus their sum is constrained to be less than

```
1 with pymiso.Context() as miso:
2     X = miso.variables(2) #declare variables
3     T = miso.variables(1)
4     line = miso.poly_space((T, 1)).geo_map(miso.bases.LAGRANGE, 3) #geometric maps
5     patch = miso.poly_space((X, 3)).geo_map(miso.bases.LAGRANGE, 3)
6     dist = ((line-patch)**2).sum() #compute the distance function
7     miso.generate('./src/generated', 'RayPatch', dist, objective=T) #generate code
```

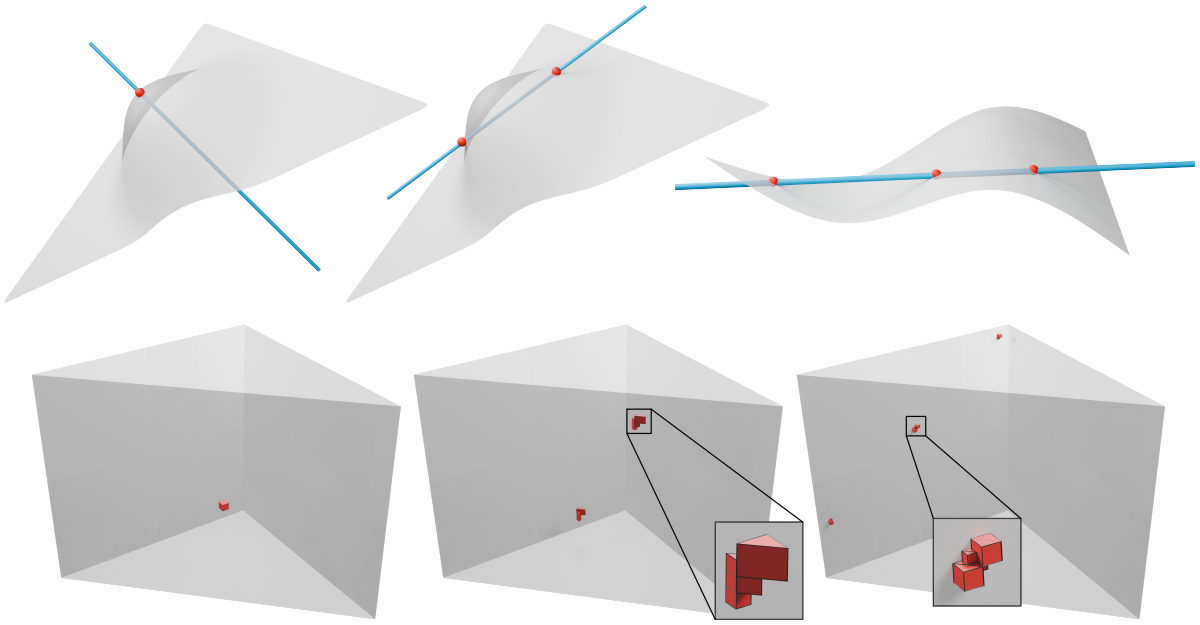


Figure 4.4: Cubic triangle - linear segment intersection. From the top: MiSo specification of the problem; Intersecting geometries for 1, 2, 3 intersecting points; Corresponding solutions of the SOLVE problem in parametric space are clusters of prisms. The value of ε is much larger than the value used in the experiments in Section 4.4 to produce visible results.

1), they must be declared together. Similarly, Line 3 declares a third variable that is assigned to `T`, representing the ray's parameter. This describes the optimization domain's triangular prism shape.

In Line 4, the geometric map of the ray is defined. The function `miso.poly_space` creates an object that represents the space of polynomials of degree 1 in `T`. The parameters of this function are enclosed in a Python `tuple`, as `poly_space` can accept multiple variable-order pairs. Then the class method `geo_map` defines the geometric map of the corresponding element in the given basis (`LAGRANGE` in this case) and embedding dimension (3 in this case, so the codomain of this function is $\mathbb{R}^3$), implicitly declaring the arguments for the control points. Line 5 analogously defines the geometric map of the two-dimensional triangle patch.

Line 6 defines the squared distance between the two primitives. Because `line` and `patch` are

vectors, the difference and power operators are automatically applied component-wise. The final `sum` operation gathers the elements of the vector by summing them. The result is the only constraint function for this problem. Note that the threshold ε is not given in the specification, but rather set later by the user when calling the solver.

Finally, Line 7 generates the C++ code in the specified folder as a class named `RayPatch` that contains all the machinery to evaluate the inclusion functions of the constraints and the objective, and to perform domain subdivision.

This class is used to instantiate the specific solver from a generic templated solver `minimize`

```
RealInterval minimize<T>(T cps, vector<double> eps, double delta);
```

which is called

```
F_star = minimize<RayPatch>({Rx, Ry, Rz, Tx, Ty, Tz}, {e}, d);
```

where `Rx, Ry, Rz` contains the coordinates of the two control points defining the ray, `Tx, Ty, Tz` contain the coordinates of each of the 10 control points defining the triangular patch, and `e` and `d` specify the precision thresholds. The corresponding ε-Solve problem, which seeks all the intersections between ray and surface patch, is easily obtained by instantiating `solve` instead of `minimize`, requiring no changes to the specification code:

```
RealInterval solve<T>(T cps, vector<double> eps);
```

which is called

```
F_star = solve<RayPatch>({Rx, Ry, Rz, Tx, Ty, Tz}, {e});
```

where parameters have the same meaning as above. The output of this function is a collection of domains of the same type of Σ, whose union forms $\tilde{\Phi}$.

The class `RayPatch` provides the data to compute the spatial extent of each such (sub)domain – a convex polyhedron – and the code to evaluate the constraint and objective functions inside it.

4.2 Algorithms

The generic algorithms Solve and Minimize are based on interval analysis and inspired by those originally described by Snyder [125]. They need to evaluate the inclusion functions of their input functions and subdivide their domain to narrow the search. We describe the class of domains we address and their subdivision; next we define the inclusion functions in general; and finally, we present the algorithms.

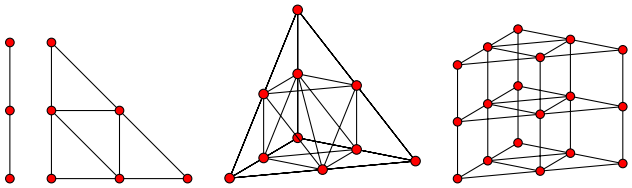


Figure 4.5: Subdivision rules for simplices of dimension 1, 2, 3, and for a triangular prism $\triangle^2 \times \triangle^1$.

4.2.1 Problem domain and decomposition

The problem domain is $\Sigma = \triangle^{n_1} \times \cdots \times \triangle^{n_s}$ the Cartesian product of n_i dimensional standard simplices. The *signature* of Σ is $(n_1, \ldots, n_s)$.

A *decomposition scheme* for a domain $\sigma \subseteq \Sigma$ is a list of affine transformations

$$\psi^q : \sigma \to \sigma \quad q = 1, \ldots, Q$$

that determine how σ is subdivided, where Q is the number of subdomains after decomposition. In practice, each ψ^q maps σ onto a subset that has the same shape, and we have $\bigcup_{q=1}^{Q} \psi^q(\sigma) = \sigma$ and $\psi^q(\sigma) \cap \psi^{q'}(\sigma)$ has measure zero in σ for all $q \neq q'$, i.e., two sub-domains can at most share boundaries.

We currently support bisection for the unit interval $\triangle^1$, quadrisection for the triangle $\triangle^2$, and decomposition of a tetrahedron $\triangle^3$ into eight tetrahedra (Figure 4.5). The decomposition of a composite space $\Sigma = \triangle^{n_1} \times \cdots \times \triangle^{n_s}$ is performed by applying the related decomposition schemes to all the factors in the Cartesian product.

For instance: if $n_i = 1$ at all i, i.e., Σ is a hypercube, then the composite scheme is the standard bisection along all coordinates; the scheme for a prism $\triangle^2 \times \triangle^1$ is depicted in Figure 4.5.

Given Σ with signature $(n_1, \ldots, n_s)$, where $n_i \leq 3$ for all i, the related subdivision scheme is generated automatically by our system.

Given an expression of a function f, our system automatically generates an inclusion function for f using interval arithmetic [11]. Refer to Section 4.3.2 for a list of currently supported expressions.

4.2.2 Pseudocode

We describe the main structure of our generic solvers Solve and Minimize. Their programming interfaces follow from the problem statements of Definitions 8 and 9.

For conciseness, we provide descriptions and pseudo-code by assuming a single constraint function C and threshold ε. In the real version dealing with systems of constraints, it is implicitly assumed that any comparison of C against ε is substituted with the Boolean $\wedge$ conjunction of each C_i against ε_i.

4.2.2.1 Solve

Algorithm Solve takes as input the problem domain Σ, the constraint function C, the precision value ε, plus two optional parameters: a Boolean FindOne to indicate whether just one feasible point or the whole feasible region is sought, and an integer $K_{\max}$ to manage early termination. If FindOne is false (default), it returns a list of non-overlapping regions that cover the entire feasible region Φ and is contained in the buffer region Φ_ε; otherwise, it returns the first point/region that satisfies the constraint.

The pseudo-code is given in Algorithm 2. The algorithm uses a queue L of subdomains, initially empty, to store subdomains that potentially intersect the boundary of the buffer region Φ_ε, and require further subdivision. The algorithm starts by processing the whole domain Σ with the function ProcessRegion. This function distinguishes three cases, depending on the relation between the inclusion function of C over the considered region ρ and the buffer region Φ_ε: if it lies completely outside Φ, then ρ is discarded; ρ *potentially* intersects Φ and lies *completely* inside Φ_ε, then ρ is added to the output; otherwise, ρ is added to L for further decomposition.

If only one solution is sought, then ρ is sampled, looking for points that may belong to the feasible region: if one such point is found, it is returned as output.

After initialization, the algorithm enters a loop that stops when L becomes empty. At each iteration, one domain σ is popped from the queue, it is subdivided into subdomains $\psi^q(\sigma)$, each of which is processed with the function ProcessRegion. The number Q of subdomains and the subdivision rules ψ^q depend on the shape of the domain Σ and are generated automatically by the compiler that processes the problem specification (Section 4.2.1).

If the number of iterations exceeds the maximum number (default $K_{\max} = \infty$) then a conservative approximate solution is returned, which consists of all regions already classified to be part of $\tilde{\Phi}$, plus all regions still in the queue S when the algorithm stops. In this case, the output is guaranteed to contain Φ but it may exceed Φ_ε.

4.2.2.2 Minimize

Algorithm Minimize takes in input the problem domain Σ, the constraint function C, the objective function F, the precision values ε and δ, plus an optional parameter $K_{\max}$, as in the previous algorithm. It returns an interval $\tilde{F}^*$ not larger than δ such that its lower end is lower

Algorithm 2 SOLVE

Input: Initial domain Σ, constraint expression C, acceptance threshold ε, boolean FINDONE, iteration limit $K_{\max}$

Output: Solution set S

$L \leftarrow$QUEUE
$S \leftarrow \varnothing$
$K \leftarrow 0$ ▷ *iteration counter*
PROCESSREGION(Σ)
loop
  if ISEMPTY(L) **then**
    return S
  if $K \geq K_{\max}$ **then**
    INSERTALL(S, L) ▷ *ensures we are conservative*
    return S
  $\sigma \leftarrow$POP(L)
  $K \leftarrow K + 1$
  for all $q \in \{1, \ldots, Q\}$ **do**
    PROCESSREGION($\psi^q(\sigma)$) ▷ *subdivide* σ
    if FINDONE $\wedge$ $\neg$ISEMPTY(S) **then**
      return S
function PROCESSREGION(ρ)
  if $\square C(\rho) \leq 0$ **then**
    if FINDONE **then** ▷ *look for a point in the feasible region*
      $V \leftarrow$SAMPLE(ρ)
      for all $v \in V$ **do**
        if $\square C(v) \leq 0 \wedge \overline{\square C(v)} \leq \varepsilon$ **then**
          $\overline{\text{INSERT}}(S, v)$ ▷ $v \in \tilde{\Phi}$
          return
    if $\overline{\square C(\rho)} \leq \varepsilon$ **then**
      INSERT(S, ρ) ▷ $\rho \subset \tilde{\Phi}$
    else
      ENQUEUE(L, ρ) ▷ ρ *will be subdivided*

than F^* and its upper end is higher than F^*_ε. The pseudo-code is given in Algorithm 3.

The algorithm uses two local variables l, u to maintain the lower and upper bounds of the output interval, which are initially set to infinity. In this case, a priority queue of regions P is used, where priority depends on the lower bound of the inclusion function of the region: a region with the smallest lower bound has the highest priority (function Priority). The queue is initialized by processing the whole domain Σ. Function ProcessRegion takes in input a region ρ and adds it to the queue if its inclusion function intersects the feasible region. It also samples ρ looking for a point v that belongs to the buffer region Φ_ε, and if one is found it uses the value of $F(v)$ to update the upper bound u. After initialization, the algorithm enters a loop that has three termination conditions: if the desired precision is met, or the maximum number of iterations is reached, then the current interval $[l, u]$ is returned; if P becomes empty without converging, then it means that the feasible region is empty, hence an infinite interval is returned. Note that, since we test the lower bound against F^* and the upper bound against F^*_ε, and $\Phi \subseteq \Phi_\varepsilon$, we may have $F^*_\varepsilon < F^*$ hence potentially $u < l$. For this reason, we set the upper bound of $\tilde{F}^*$ to $\max\{l, u\}$. In the loop, a region σ is extracted from the queue, and the lower bound of the solution is updated, because the priority warrants that the lower bound of F on all regions in the queue is greater than or equal to that on σ. The σ is subdivided and its sub-regions are processed.

As in the previous case, if the algorithm exits because it exceeds the maximum number of iterations, the solution is conservative but approximate: it is still true that the lower end of $\tilde{F}^*$ is lower than F^* and its upper end is larger than F^*_ε, but its width is $> \delta$.

4.3 Implementation

The main purpose of the MiSo compiler is to produce the C++ code to evaluate the inclusion functions for the functions defined in the problem specification, and to implement domain subdivision.

Each function defined for the specific problem is first represented with an expression tree, whose leaves can be constants, variables, or polynomials represented symbolically in SymPy, while inner nodes are operators (Section 4.3.2). The related inclusion function is assembled by analyzing this tree and applying specific rules for the leaves and the inner operators (Section 4.3.3). The inclusion functions for polynomials exploit the convex hull property of their Bézier representation while rules for the inner operators stem from interval analysis.

Algorithm 3 MINIMIZE

Input: Initial domain Σ, constraint expression C, constraint and objective acceptance thresholds ε and δ, iteration limit $K_{\max}$

Output: Interval $\tilde{F}^*$

$l \leftarrow \infty$ ▷ *lower bound of* $\tilde{F}^*$
$u \leftarrow \infty$ ▷ *upper bound of* $\tilde{F}^*$
$P \leftarrow$ PRIORITYQUEUE
$K \leftarrow 0$
PROCESSREGION(Σ)
loop
    if $u - l \leq \delta$ **then**
        return $[l, \max\{l, u\}]$ ▷ *reached the desired precision*
    if ISEMPTY(P) **then**
        return $[\infty, \infty]$ ▷ *feasible region is empty*
    if $K \geq K_{\max}$ **then**
        return $[l, \max\{l, u\}]$ ▷ *return current approximation*
    $\sigma \leftarrow$ POP(P)
    $K \leftarrow K + 1$
    $l \leftarrow \underline{\Box F(\sigma)}$ ▷ *update lower bound*
    for all $q \in \{1, \ldots, Q\}$ **do** ▷ *subdivide* σ
        PROCESSREGION($\psi^q(\sigma)$)
function PROCESSREGION(ρ)
    if $\overline{\Box C(\rho)} \leq 0 \wedge \overline{\Box F(\rho)} \leq u$ **then**
        $V \leftarrow$ SAMPLE(ρ) ▷ *look for a point in the buffer region*
        for all $v \in V$ **do**
            if $\overline{\Box C(v)} \leq \varepsilon$ **then**
                $u \leftarrow \min\{u, \overline{\Box F(v)}\}$ ▷ *update upper bound*
        ENQUEUE(P, ρ, PRIORITY(ρ)) ▷ ρ *will be subdivided*
function PRIORITY(ρ)
    return $-\underline{\Box F(\rho)}$ ▷ *smallest lower bound on* $\Box F$ *goes first*

4.3.1 Symbols

Before writing the expressions for the constraints and/or objective, the user must define the symbols that will appear in the expressions. MiSo has two types of symbols: variables and arguments. Both are defined through methods of the `Context` object that appears in the code in Figure 4.3, which keeps track of what symbols have been created. MiSo uses SymPy to store and manipulate expressions, but disallows the use of `Symbol` objects in expressions if they are not known to the `Context`.

4.3.1.1 Variables

These symbols describe the parameter space of the problem. They are created through the `variables` method of the `Context` object on a per-simplex basis: `variables(n)` declares `n` variables corresponding to the dimensions of a standard `n`-simplex. Variables are automatically assigned names by their `Context`, but the user can also specify a custom name: this has no practical effect on the generated C++ code but can be useful, e.g., to debug a faulty specification by printing expressions with descriptive names.

4.3.1.2 Arguments

These symbols represent data the user must pass to the program. They are created through the `arguments` method of the `Context` object on a per-vector basis, meaning that `arguments(n)` declares `n` arguments that will be passed in as a single iterable. Just like variables, arguments are automatically assigned names by their `Context`, with the possibility to specify a custom name to make the generated C++ class have meaningful parameter names in its constructor.

While all symbols must be declared, the utility method `geo_map` can implicitly declare arguments (that represent the control points of the element in the generated expressions) if the user does not supply their own.

4.3.2 Expressions

MiSo represents functions associated with a problem as expression trees made of objects of type `Node`. This base class has derived classes corresponding to the types of objects that make up a MiSo expression tree:

- `OpNode` (Operator nodes)
- `PolyNode` (Polynomial nodes)

- `VectorNode` (Vector nodes)

4.3.2.1 Operator nodes

`OpNodes` represent the operators of expressions. Every `OpNode` has a list of child nodes that can be `PolyNodes` as well as other `OpNodes`. Both unary and binary operators are supported, as well as n-ary versions of associative operators (e.g., the sum or product of n expressions). Rather than being created directly, `OpNodes` are returned by overloaded arithmetic operators of the `Node` parent class.

4.3.2.2 Polynomial nodes

The leaves of MiSo expression trees are of type `PolyNode`. This object represents any polynomial expression $P_{\mathbf{K}}(\mathbf{x})$ in the (previously declared) variables $\mathbf{x}$ and with coefficients possibly depending on the (previously declared) arguments $\mathbf{K}$. The following entities are all represented as `PolyNodes` in MiSo expression trees:

- numeric constants, as polynomials of degree 0;
- arguments, as polynomials of degree 0;
- variables, as polynomials of degree 1;
- polynomial expressions involving the above, where the degree of the polynomial in each variable is automatically computed.

The `PolyNode` object overloads several polynomial and non-polynomial Python operators: they do not return a new `PolyNode` but rather an `OpNode` with the operands as child nodes. If the user wants to combine part of an expression tree into a single polynomial node, calling the `collapse` method on the root of the subtree will return an equivalent `PolyNode` object, or raise an exception if non-polynomial operators are present in the subtree.

4.3.2.3 Vector nodes

Like operator nodes, vector nodes store a list of child nodes, but in this case they represent the components of a vector. The main use of the `VectorNode` class is to automatically cast operations with single nodes and other vectors to component-wise operations. The class also exposes "reduction" methods that return a single `OpNode` with the components as children. A vector's components can be vectors themselves, allowing the user to represent matrices such as the Jacobian matrix in Table 4.6.

4.3.3 Evaluating inclusion functions

The evaluation of an inclusion function for a given expression boils down to evaluating the inclusion functions for the leaves of the tree (from their Bézier representation) and then combining them up the tree with interval arithmetic, according to the operators in the inner nodes (as natural interval extensions).

The result of evaluating a node is always an interval. In the case of a constant, this interval can be of zero width.

For a polynomial P, we need to compute an interval that contains $P(\sigma)$. We always use the Bézier form as internal representation of polynomials: a specific polynomial is translated to this form as soon as it is taken in input, being represented as a set of coefficients with respect to the Bézier basis of its corresponding space. Because of the convex hull property of the Bézier representation, we know that the range of $P(\sigma)$ is contained in the interval spanned by its lowest and highest Bézier coefficients.

The types of operators in the inner nodes can be extended easily as long as (robust) inclusion functions are made available for them. For instance, trigonometric operators could be included in MiSo by relying on the RLIBM library [3, 96, 4].

Note that, our choice of treating the polynomials as a special class of functions is crucial to the efficiency of our solvers, since it allows us to trade-off between NIE-based and Bézier-based inclusion functions. Given the polynomial nature of many practical computer graphics problems, our generic, automatically generated code often demonstrates performance on par with, or even surpassing, hand-crafted solutions tailored to specific problems (Section 4.4).

4.3.4 Domain decomposition

The decompositions of a domain σ and a polynomial defined on σ use the same functions. Indeed, an affine function that maps σ onto a subset of itself is a (vector) linear polynomial in all its variables.

Each decomposition function ψ^q computes the Bézier control coefficients of a polynomial defined on σ mapping it to its restriction to subdomain $\psi^q(\sigma)$. These coefficients are computed by applying the De Casteljau decomposition on all the simplices that define σ and combining them by tensor product. This is possible because the basis of the tensor product space of polynomials over a Cartesian product $\triangle^{n_1} \times \cdots \times \triangle^{n_s}$ with orders $p_1, \ldots, p_s$ is the tensor product of the bases of the single spaces of polynomials defined on each $\triangle^{n_i}$ with order p_i. This property allows us to compile the conversion matrices that implement all the ψ^q mappings for the domain and the polynomials defined on it.

The code for domain decomposition, and for the corresponding decomposition of polynomi-

als defined on the problem domain, is generated automatically by considering the spaces of polynomials corresponding to the leaves of the tree and generating subdivision matrices with a contruction similar to [69], which was already presented in Section 3.4.1.1, but is briefly summarized here for clarity.

Given a polynomial, we can compute its Lagrange representation by sampling it on the set of *Lagrange points* $\mathbf{L}$ appropriate for its degree. In particular, this can be done for the polynomials of the Bézier basis to generate a change of basis matrix $T_{\mathcal{L}\leftarrow\mathcal{B}}$ to convert a polynomial in Bézier form to its Lagrange form. By inverting this matrix, we obtain the opposite transformation $T_{\mathcal{B}\leftarrow\mathcal{L}} = T_{\mathcal{L}\leftarrow\mathcal{B}}^{-1}$ to compute the Bézier coefficients of a polynomial given its sampled values. To compute the Bézier coefficients on a subdomain, we instead sample the Bézier basis polynomials at the transformed Lagrange points $\psi^q(\mathbf{L})$, which gives another transformation from Bézier to Lagrange form $T^q_{\mathcal{L}\leftarrow\mathcal{B}}$; by premultiplying this by the Lagrange-to-Bézier matrix we computed earlier we get the desired transformation matrix

$$T^q_{\mathcal{B}\leftarrow\mathcal{B}} = T_{\mathcal{B}\leftarrow\mathcal{L}} T^q_{\mathcal{L}\leftarrow\mathcal{B}}$$

Each of these linear transformations M is assembled by the MiSo compiler, then their product $M\mathbf{v}$ with an unknown vector $\mathbf{v}$ is expanded, and the resulting expression vector is simplified with SymPy's common sub-expression elimination (CSE) tools, and finally encoded as a C++ function specific to that domain. Since the matrices depend only on the polynomial degree and the domain type, the computation occurs entirely at compile time: the generated C++ functions contain only scalar arithmetic, with no runtime matrix allocation or multiplication.

4.3.5 The MiSo compiler

We provide a package consisting of three distinct pieces:

- PyMiSo: a Python library that relies on SymPy to generate C++ code;
- MiSo-core: a C++ library on which the generated code relies;
- MiSo-algorithms: implementations of generic Solve and Minimize that rely on the MiSo-core library and are instantiated with the code generated by running a problem specification.

As seen in Section 4.1.6, a problem specification in MiSo consists of a Python script using functions from PyMiSo. The code generated by running such a script is wrapped in a C++ class, which is used to instantiate a specific solver by plugging it into a template function from MiSo-algorithms.

All the operations necessary to convert polynomials between different bases and perform domain decomposition are treated symbolically in PyMiSo to generate the related C++ code that computes such operations numerically, again leveraging CSE to simplify the expressions and assist the compiler.

We provide two backends that differ only in the base type to represent real numbers; the backend is selected when compiling the C++ code, without changing the specification:

- Floating-point: in this case, all computations involving real values are done with standard floating-point operations, while interval arithmetic is used only to perform interval analysis (essentially, in evaluating the inclusion functions). The code will be faster, but the result will not be conservative.
- Interval: in this case, interval arithmetic is used throughout, for both interval analysis and numerical computations. The result will be conservative, at some additional cost.

At the time of writing, MiSo also supports a wide class of operators, including transcendental functions, thanks to the adoption of the TIGHT library for interval arithmetic presented in Chapter 5. For a detailed description and empirical evaluation, refer to that chapter.

By default, the subdivision rules in the generated code are those explained in Section 4.3.4, i.e., domain Σ is subdivided along all cartesian axes. MiSo enables customizing these rules, and in some of our experiments with moving objects we also use a subdivision in the time dimension only.

MiSo can be easily customized in two ways:

1. By adding new functions to PyMiSo one can, for instance: add new subdivision rules; and add conversions of polynomials from other bases.
2. By customizing the generic algorithms in MiSo-algorithms: these can be simply cloned, modified, and instantiated with the class generated by running the specification. In Section 4.4.2, we use a customized version of Minimize in a comparison against [24] to comply with their termination conditions.

4.4 Applications

We now explore the uses of our DSL in a disparate set of graphics and geometric modeling applications. For each application, we briefly introduce the state of the art, select a representative problem, and present a solution with a MiSo program.

For all experiments, we used a laptop with an AMD Ryzen 7 4000 series processor, 16 GB of RAM, and compiled with GCC 12.2.0 on Debian Linux.

4.4.1 Static objects intersections

Computing the intersection between parametric primitives is a staple in physical simulation to detect and respond to contact. Many methods have been proposed, with a wide range of primitives supported and accuracy targets. We refer to [5] for an overview of fast methods used in real-time physics engines and to [139] for an overview of conservative methods used for offline simulation.

4.4.1.1 Explicit cubic triangle-linear segment intersection

We consider the problem used in Section 4.1.6 as a didactic example. First we formulate this problem as a SOLVE and execute the query on 3 cases (Figure 4.4). We used $\varepsilon = 10^{-4}$, and the three queries return the intersections in all 3 cases, with a runtime of $52\mu s$, $163\mu s$, and $215\mu s$, respectively. We can treat the (oriented) segment as a ray and ask to find the first time of collision with the curved patch. To do so, we solve the corresponding MINIMIZE problem with the single parameter of the segment as the objective function, representing time. The same queries as before are solved with $\delta = \varepsilon = 10^{-4}$ in $124\mu s$, $191\mu s$, and $148\mu s$ respectively, with results $[0.333251, 0.333312]$, $[0.146423, 0.146423]$, $[0.294250, 0.294281]$. For the first query, we know that the true time of impact is $1/3$. Note that the upper bound does not contain this point: the algorithm verified that, at that time, the two objects were closer than ε.

4.4.1.2 Explicit linear triangle-sphere intersection

We compute the intersection between a linear triangle and a hollow sphere. Both primitives are represented in parametric form: the sphere is incomplete, being parametrized on a square via the stereographic projection, which is a rational polynomial map.

Note that this requires divisions between intervals, which is supported in our framework, but is risky and should be handled with care. The division between intervals is permitted, but it should be avoided whenever possible, as there is a risk of encountering a division by 0. Divisions are implemented as product of the dividend with the reciprocal of the divisor; if the divisor contains 0, the computation of the inverse will raise an exception by default. Because of the conservative nature of our implementation, it is possible for this exception to be encountered even when the true function does not include zero, due to the overestimation of function ranges and accumulated roundoff error. If this happens, it generally means that the operation was unsafe to begin with, as it is nearly singular. However, we provide a compilation flag that allows the user to accept the risk and proceed with computations. When this flag is enabled, the operation to compute the reciprocal of an interval containing 0 will return $[0, \infty]$ if the left extreme is 0, $[-\infty, 0]$ if the right extreme is 0, and $[-\infty, \infty]$ otherwise. Floating point operations involving ∞ are handled by the NFG library as normal floating point operations, but the

```python
with pymiso.Context() as miso:
    U = miso.variables(1, 'U') #declare variables with names (optional)
    V = miso.variables(1, 'V')
    X = miso.variables(2, 'X')
    center = miso.arguments(3, 'c') #declare arguments explicitly (sphere center)
    # Create the sphere's geometric map via stereographic projection
    uv_scale = 4
    Uc = ((2*U-1) * uv_scale).collapse()
    Vc = ((2*V-1) * uv_scale).collapse()
    U2V2 = (Uc**2 + Vc**2)#.collapse()
    sphere = (miso.vector(2*Uc, 2*Vc, U2V2-1) / (U2V2+1)) + center
    pb = miso.bases.LAGRANGE
    triangle = miso.poly_space((X,1)).geo_map(pb, 3) #triangle geometric map
    dist = ((triangle-sphere)**2).sum() #distance
    miso.generate('./src/generated', 'SphereTriangleSSI', dist)
```

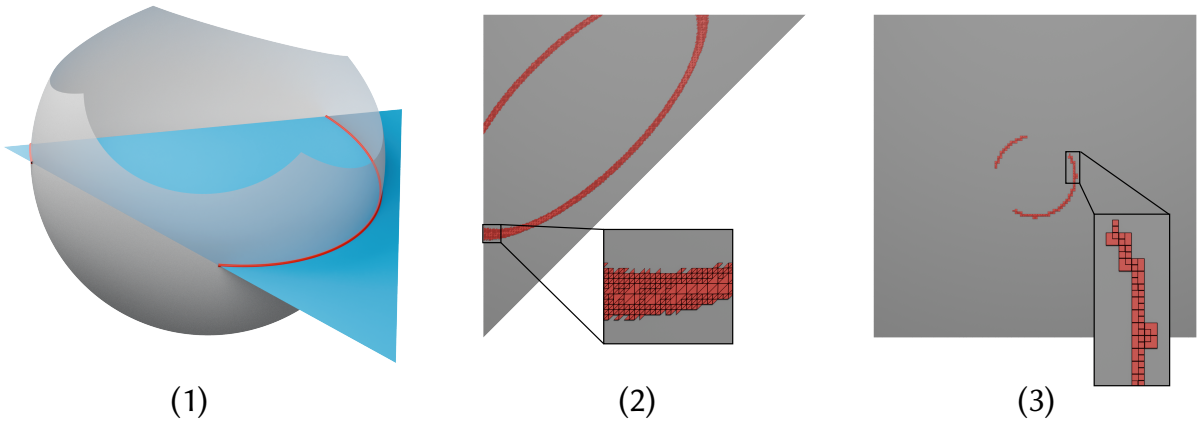


Figure 4.6: Intersection of a triangle with a hollow sphere. (1) Both primitives are expressed in parametric form, the sphere requiring rational polynomials. They intersect at two disjoint arcs (in red). (2) Our solutions in the parametric domain of the triangle. (3) Our solutions in the parametric domain of the sphere. Note that the shown solutions are obtained with reduced accuracy to make them visible, and contain overlapping regions since they are projections of a 4D solution set (with no overlaps) onto 2D spaces.

results may contain NaNs for some combinations of operands that contain infinite values and/or zeroes. While we keep this formulation as an example of a problem involving rational polynomials, the issue above can be avoided for this specific problem by multiplying the coordinates of both primitives by the denominator of the stereographic projection map. The division of an interval by an exact number different from zero is also available, and it is a safe operation.

The query should be called twice to find the intersection with the full sphere. We express this as a Solve problem, using the program in Figure 4.6. One query finds an intersection with 74181 regions in 4D parametric space in 338ms to precision 10^{-2}.

4.4.2 Dynamic objects intersections

Static collision detection might miss collisions in dynamic simulation when objects move fast. Continuous collision detection addresses this challenge by finding the time at which the first collision appears, assuming that the primitives are moving through time using a linear [137, 24] or nonlinear [42, 142] trajectory.

4.4.2.1 Alternative subdivision strategies

In the following MiSo specifications we also show how the user can specify different subdivision strategies. The `generate` function has an optional paramter `strategies` for this purpose; the user can supply an iterable of subdivision objects (created with `subdiv_strategy`) and the compiler will generate the necessary code to subdivide the domain according to these strategies. It is assumed that at least one strategy is provided, and if the user does not specify one, the default of subdividing on all axes will be used automatically.

Strategies are assigned an index, starting from 0, ordered as they are passed to `generate`. Each region pushed onto the queue holds an integer specifying the index of the strategy to use to subdivide it, with strategy 0 being used when a strategy with the specified index does not exist. The conditions under which each strategy is used depends on the implementation of the algorithms. The provided Solve algorithm always uses strategy 0, the default. In the provided Minimize algorithm, strategy 0 is the default, whereas strategy 1 is used to subdivide regions where a feasible point has been found.

The reasoning behind this is the following: consider a region of space-time that spans times $[\underline{t}, \overline{t}]$. If the algorithm verifies that the system is in an invalid state at time $\overline{t}$, it may be enough to split in the time variable only, pushing 2 subdomains onto the priority queue as opposed to e.g. 32 for the CCD problem for surface elements.

If the user wants to implement other strategies (e.g. alternating subdivisions on the two elements in a CCD query), it is easy to add them to the specification code, and modify the base algorithm so that it uses the desired strategy under certain conditions.

4.4.2.2 Linear triangle - quadratic trajectories

The program in Figure 4.7 computes, conservatively, the first intersection between two triangles whose vertices are moving on a quadratic trajectory. The Minimize problem is solved to precision $\delta = 10^{-4}$ with tolerance $\varepsilon = 10^{-4}$. The first query reports a collision interval of $[0.499939, 0.499969]$ in $744\mu s$. The true time of collision is $1/2$: as in Section 4.4.1, this is a conservative solution to the Minimize problem. In the second case, our algorithm correctly classifies the trajectory as collision-free in $34\mu s$, returning $[\infty, \infty]$.

```
with pymiso.Context() as miso:
    X = miso.variables(2)
    Y = miso.variables(2)
    T = miso.variables(1)
    pb = miso.bases.LAGRANGE
    elem_a = miso.poly_space((X, 1))
    elem_b = miso.poly_space((Y, 1))
    xa0 = elem_a.geo_map(pb, 3) #static geometric maps
    xa1 = elem_a.geo_map(pb, 3)
    xa2 = elem_a.geo_map(pb, 3)
    xb0 = elem_b.geo_map(pb, 3)
    xb1 = elem_b.geo_map(pb, 3)
    xb2 = elem_b.geo_map(pb, 3)
    timebasis = miso.poly_space((T,2)).basis(pb) #Lagrange basis in T
    xav = miso.vector(xa0, xa1, xa2) #dynamic geometric maps
    xa = (xav * timebasis).sum().collapse()
    xbv = miso.vector(xb0, xb1, xb2)
    xb = (xbv * timebasis).sum().collapse()
    dist = ((xb-xa)**2).sum()
    sd = [miso.subdiv_strategy(), miso.subdiv_strategy(T)]
    miso.generate('./src/generated', 'CurvedCCD', dist, objective=T, strategies=sd)
```

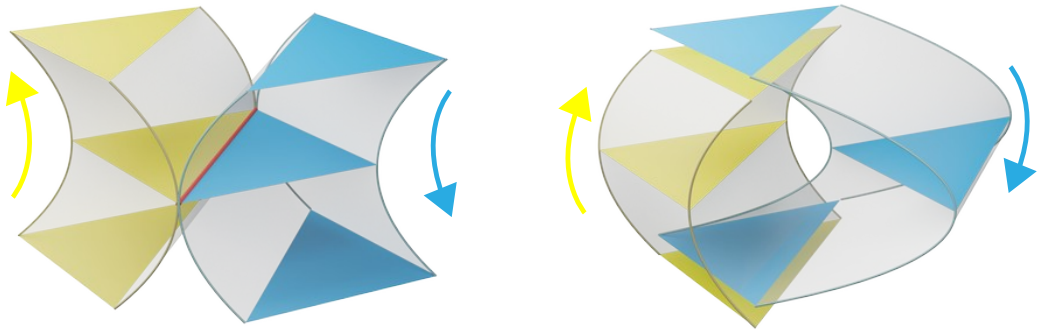


Figure 4.7: Linear triangle CCD on quadratic trajectories. The direction of trajectories is marked with arrows. Left: Collision occurs and is detected on a whole edge (red) at a single time; primitives are disjoint at all other times. Right: no collision at all.

4.4.2.3 Comparison with [139]

We consider the problem of continuous collision detection of a pair of edges whose vertices are moving on linear trajectories. Figure 4.8 shows the specification and an example query. We compare the result of MiSo with the heavily optimized algorithm in [139] in Table 4.2. Our algorithm is conservative and around an order of magnitude faster on hard queries, whereas it is one order of magnitude slower on easy queries.

```
with pymiso.Context() as miso:
    X = miso.variables() #declare variables
    Y = miso.variables()
    T = miso.variables()
    elem_a = miso.poly_space((X, 1)) #create static elements
    elem_b = miso.poly_space((Y, 1))
    pb = miso.bases.BEZIER
    xa0 = elem_a.geo_map(pb, 3) #compute static geometric maps
    xa1 = elem_a.geo_map(pb, 3)
    xb0 = elem_b.geo_map(pb, 3)
    xb1 = elem_b.geo_map(pb, 3)
    xa = ((xa0 * (1-T)) + (xa1 * T)).collapse() #compute dynamic geometric maps
    xb = ((xb0 * (1-T)) + (xb1 * T)).collapse()
    L22 = ((xb-xa)**2).sum() #compute squared distance
    sd = [miso.subdiv_strategy(), miso.subdiv_strategy(T)] #subdiv. strategies
    miso.generate('./src/generated', 'EECCD', L22, objective=T, strategies=sd)
```

Figure 4.8: Linear segment CCD on linear trajectories. The code, used for comparisons with [139], is similar to the one in Figure 4.7.

Table 4.2: EECCD on the handcrafted (15K queries) and simulation (41M queries) datasets from [139], 10^6 max iterations. We show average time per query in microseconds, number of false positives (#FP) and false negatives (#FN, i.e. missed collisions).

Dataset	Handcrafted			Simulation		
Method	Ours		Theirs	Ours		Theirs
$\varepsilon = \delta$	10^{-4}	10^{-6}	10^{-6}	10^{-4}	10^{-6}	10^{-6}
μs/query	102	243	3029	3.5	4.3	0.78
#FP	222	52	214	11375	42	17
#FN	0	0	0	0	0	0

Table 4.3: Cubic triangle CCD in 3D. Runtime averaged on 1000 random queries, 10^{-4} precision. MINIMIZE has been modified for this test to use the same termination condition and subdivision strategy as [24]'s method.

Method	ms/query
[24] TDI + OBB	33
[24] "traditional" + OBB	241
[142] (SOSP)	7528
MiSo w/ interval arithmetic backend	482
MiSo w/ floating point backend	176

4.4.2.4 Comparison with [142] and [24]

Adapting our algorithm to higher-order collision detection requires minimal changes to our MiSo program. In Figure 4.9, we show a MiSo program to compute the cubic triangle to cubic

```python
with pymiso.Context() as miso:
    X = miso.variables(2)
    Y = miso.variables(2)
    T = miso.variables(1)
    elem_a = miso.poly_space((X, 3))
    elem_b = miso.poly_space((Y, 3))
    pb = miso.bases.LAGRANGE
    xa0 = elem_a.geo_map(pb, 3)
    xa1 = elem_a.geo_map(pb, 3)
    xb0 = elem_b.geo_map(pb, 3)
    xb1 = elem_b.geo_map(pb, 3)
    xa = ((xa0 * (1-T)) + (xa1 * T)).collapse()
    xb = ((xb0 * (1-T)) + (xb1 * T)).collapse()
    dist = ((xb-xa)**2).sum()
    sd = [miso.subdiv_strategy(), miso.subdiv_strategy(T)]
    miso.generate('./src/generated', 'CubicTriCCD', dist, objective=T, strategies=sd)
```

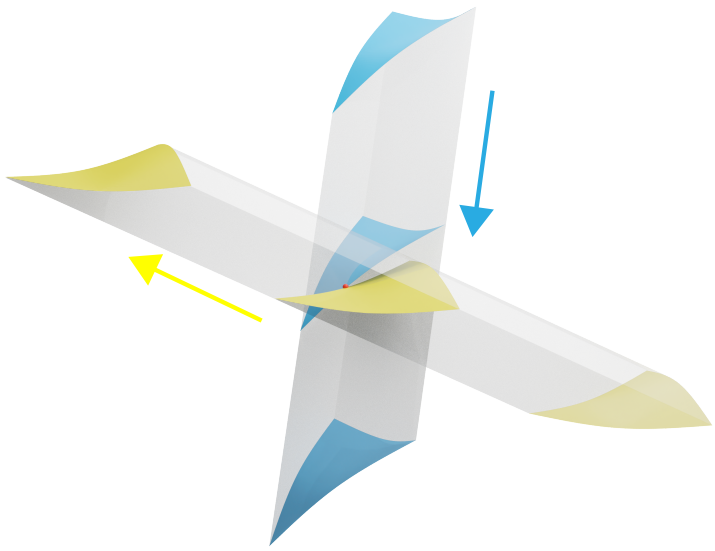


Figure 4.9: Cubic triangles CCD on linear trajectories. The code is similar to the one in Figure 4.7. The result at the bottom is an example of the queries used for comparisons with [142, 24].

triangle continuous collision problem introduced in [142]. We compare our method with the SOS-based solver in [142] and the inclusion-based solver in [24] on randomly generated queries.

For a fair comparison, we employ a slightly modified version of MINIMIZE that uses the same termination condition as [24], as discussed in Section 4.1.3.1. We also substitute the squared L_2 distance with the L_∞ distance as in [24]. Their method also uses Oriented Bounding Boxes (OBB) to verify the separation between patches, which is difficult to do robustly, so we instead use Polyhedral Bounding Boxes (PBB) in our version. A complete description of the modified specification and the use of PBBs is in Section A.4.

Our approach is more efficient than SOS, while being guaranteed conservative. Compared to [24], our program has comparable performance without using their time-dependent inclusion approach (TDI), and becomes one order of magnitude slower when TDI is used for [24] (See Table 4.3). It would be interesting to generalize the TDI idea to our DSL and make it usable in additional problems beyond continuous collision detection. Note that, our solver is conservative, while [24] relies on a numerical tolerance.

4.4.3 Collapsing the expression

To demonstrate the performance gains that can be achieved with a DSL that allows for quick experimentation of different solutions, we compare several equivalent MiSo specifications to CCD queries in terms of efficiency.

First, we consider three possible specifications of the CCD problem for bilinear quad patches, shown in Figure 4.10. We run the experiment on 100 random collision pairs generated as in [24], using the squared Euclidean distance, with parameters $\varepsilon = 10^{-6}$, $\delta = 10^{-4}$, and the numerically robust backend. The NIE-based approach takes an average of 2 seconds per query, the Bézier-based approach takes 1 second per query, and our hybrid approach takes 52 milliseconds per query.

Next, we consider three variants of the specification shown in Figure 4.9:

1. one without modifications, where we collapse the expressions for the time-dependent geometric maps;
2. one "slightly expanded" version without the two `collapse` statements at lines 12 and 13, meaning that the inclusion functions of the static geometric maps will be computed separately with Bézier inclusions and then combined by natural interval extension;
3. one "slightly collapsed" with a single `collapse` statement on the `(xb-xa)` expression in line 14, meaning that the difference of each coordinate of the two time-dependent geometric maps will be considered a single polynomial in 5 variables.

```
def QuadPatchCCD(variant=0):
  with pymiso.Context() as miso:
    X1 = miso.variables()
    X2 = miso.variables()
    Y1 = miso.variables()
    Y2 = miso.variables()
    T = miso.variables()
    elem_a = miso.poly_space((X1, 1), (X2, 1))
    elem_b = miso.poly_space((Y1, 1), (Y2, 1))
    pb = miso.bases.BEZIER
    xa0 = elem_a.geo_map(pb, 3)
    xa1 = elem_a.geo_map(pb, 3)
    xb0 = elem_b.geo_map(pb, 3)
    xb1 = elem_b.geo_map(pb, 3)
    xa = ((xa0 * (1-T)) + (xa1 * T)).collapse()
    xb = ((xb0 * (1-T)) + (xb1 * T)).collapse()
    dist = ((xb-xa)**2).sum() #hybrid version
    if variant == 'Bezier':
      dist = dist.collapse() #Bezier-based version
    elif variant == 'NIE':
      dist = dist.expand() #NIE-based version
    sd = [miso.subdiv_strategy(), miso.subdiv_strategy(T)]
    miso.generate('./src/generated', 'QuadCCD', dist, objective=T, strategies=sd)
```

Figure 4.10: MiSo specification for CCD of linear quadrilateral patches. Notice that the variables are declared separately as they are not part of the same simplex (the parameter space of the problem is $[0, 1]^5$).

In addition, one could fully expand the expression (i.e., compute inclusions with interval arithmetic only) or fully collapse it into one node (since the squared Euclidean distance is polynomial). The fully expanded version produces inclusion functions too large to converge in a reasonable number of iterations and in general is slower by multiple orders of magnitude; whereas the code generation phase of the second one used too many resources to complete on our setup, as it had to pre-compute transformation matrices of a degree 6 polynomial in 5 variables.

The three equivalent versions were tested on the same dataset as the experiment in Table 4.3, but in this case without modifications in the Minimize algorithm, with parameters $\varepsilon = 10^{-6}$, $\delta = 10^{-4}$, and the numerically robust backend. The results are presented in Table 4.4, and show that there is a sweet spot between fully NIE-based and fully Bézier-based inclusions that can be easily found by changing a few lines in the MiSo specification.

In terms of generation time, the collapsed version took 15 minutes, whereas the expanded and unmodified versions took approximately 10 seconds each.

4.4.4 Implicit booleans

We take inspiration from [127] and cast Boolean operations between two implicit spheres as a Solve problem (Figure 4.11). We report statistics in Table 4.5: changing the primitive types

Table 4.4: Comparison of several equivalent specifications of the order 3 triangle CCD problem, using MINIMIZE with parameters $\varepsilon = 10^{-6}, \delta = 10^{-4}$, and the numerically robust backend. Fully expanded and fully collapsed versions of this problem are not included, as the former does not converge in a reasonable time, and the latter uses too much memory to generate the code.

Average	ms/query	# iterations	μs/iteration
V1 (unchanged)	203	6520	31
V2 (expanded)	271	9740	28
V3 (collapsed)	2267	6340	358
Median	ms/query	# iterations	μs/iteration
V1 (unchanged)	38	1156	32
V2 (expanded)	54	1798	30
V3 (collapsed)	413	1082	382

Table 4.5: SSI and booleans between two implicit spheres - full solution with tolerances $\varepsilon = 10^{-2}$ and $\varepsilon = 10^{-3}$. We list the total runtime of SOLVE and the number of regions in the output.

	$\varepsilon = 10^{-2}$		$\varepsilon = 10^{-3}$	
Query	Total ms	#regions	Total ms	#regions
SSI	1	282	5	3390
Solid $\cap$	4	4570	260	410954
Solid $\setminus$	8	10082	595	1002961
Solid $\cup$	13	16564	1015	1602996

and operations are minor changes in the specifications, making our compiler a powerful tool to compute conservative Boolean solutions between implicit and explicit primitives.

4.4.5 Minimal distance

Computing distances between linear meshes is a classical problem that has been extensively studied [13, 15, 23, 26, 54, 72, 74, 144] and for which robust and efficient algorithms exist. Its curved version is more challenging [8, 75, 128].

4.4.5.1 Quadratic Bézier patch - segment

With MiSo, it is simple to compute minimal distances between curved primitives: we show an example with a quadratic Bézier patch and a linear segment in Figure 4.12. With $\delta = 10^{-2}$ this query takes around 2ms and returns the interval $[0.750706, 0.760696]$ as estimation of the squared L_2 distance between the primitives; with $\delta = 10^{-4}$ the timing increases to 223ms and

```python
def ImplicitSphereBooleans(operation):
  with pymiso.Context() as miso:
    X = miso.variables()
    Y = miso.variables()
    Z = miso.variables()
    centerA = miso.arguments(3, 'ca')
    radiusA = miso.arguments(1, 'ra')
    centerB = miso.arguments(3, 'cb')
    radiusB = miso.arguments(1, 'rb')
    xyz = miso.vector(X,Y,Z)
    d2a = ((xyz-centerA)**2).sum() - radiusA**2
    d2b = ((xyz-centerB)**2).sum() - radiusB**2
    if operation == 'SSI': constr = abs(d2a) | abs(d2b)
    elif operation == 'Union': constr = d2a & d2b
    elif operation == 'Intersection': constr = d2a | d2b
    elif operation == 'Difference': constr = d2a | -d2b
    else: raise ValueError('Unknown operation')
    miso.generate('./src/generated', f'Sphere{operation}', constr)
```

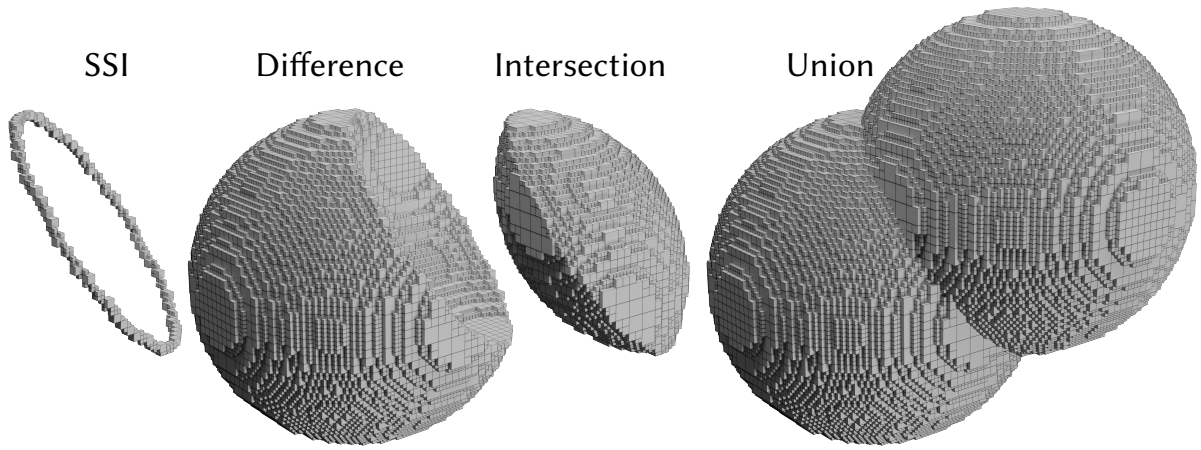


Figure 4.11: Boolean and SSI operations between two implicit spheres. Top: MiSo specification is parametrized on the type of Boolean operation. Bottom, from left to right: results of SSI, boolean difference, intersection, and union with $\varepsilon = 10^{-2}$.

```
with pymiso.Context() as miso:
    U = miso.variables()
    V = miso.variables()
    R = miso.variables()
    pb = miso.bases.LAGRANGE
    segment = miso.poly_space((R,1)).geo_map(pb, 3)
    patch = miso.poly_space((U,2),(V,2)).geo_map(pb, 3)
    dist = ((segment-patch)**2).sum()
    miso.generate('./src/generated', 'ClosestPoint', objective=dist)
```

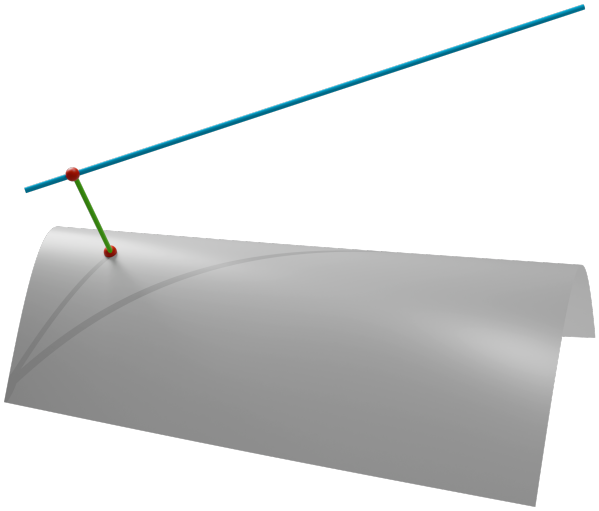


Figure 4.12: Distance between quadratic Bézier patch (gray) and segment (blue). The direction (green) of the closest point pairs (red) is shown.

returns $[0.760585, 0.760685]$.

4.4.6 Geometrical validity check

MiSo can generate solvers for the geometrical validity problem presented in Chapter 3. Here we compare against [69] on the static high-order validity check. In Table 4.6 we report results on a mesh consisting of 48050 valid tetrahedra and 6935 invalid ones (Figure 4.13 bottom right). Since this is a boolean test, we use the SOLVE algorithm with the FINDONE flag activated, to early terminate the search as soon as we find that the element contains an invalidity. Our version with interval arithmetic is slightly slower on average ($7\mu s$ vs $5\mu s$) than the algorithm proposed in [69], which uses numerical tolerances and is thus not provably conservative. The same program, compiled with our floating point backend, leads to a faster (3μs) average runtime.

Table 4.6: Geometrical validity of order 3 tetrahedra. We show the per-query runtime averaged on 54985 queries with 6935 inverted elements.

Method	μs/query
[69]	5
MiSo w/ interval arithmetic backend	7
MiSo w/ floating point backend	3

```
def SimplexValidity(D, P):
    with pymiso.Context() as miso:
        X = miso.variables(D)
        geo = miso.poly_space((X,P)).geo_map(miso.bases.LAGRANGE, D)
        jd = geo.jacobian().det().collapse() #Jacobian determinant
        miso.generate('./src/generated', f'ValidityD{D}P{P}', jd)
```

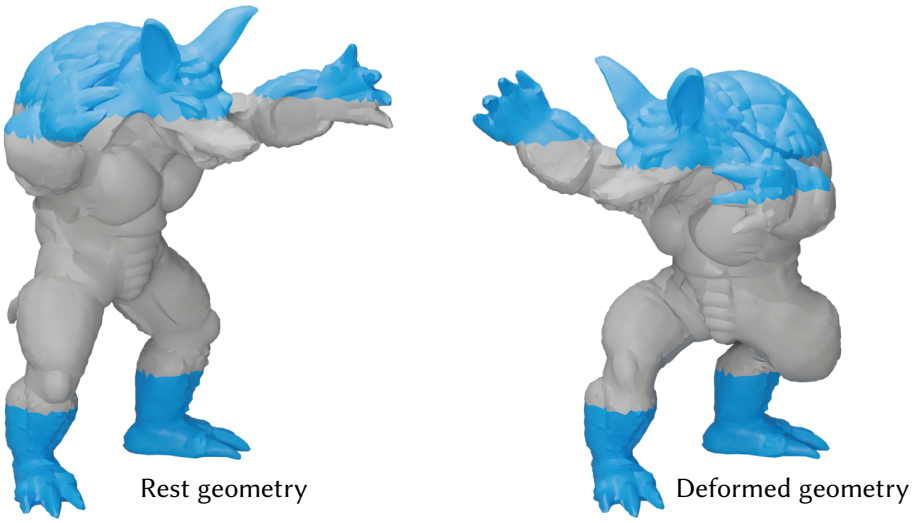


Figure 4.13: Geometrical validity check for cubic tetrahedral elements. The shape to the left consists of all valid tetrahedra; we test the deformed model to the right, which contains 6935 inverted elements.

4.4.7 Compilation time

The efficiency of our automatically generated code is partially due to unrolling all numerical computations, thus allowing for code optimization at compile time. We perform common subexpression elimination (CSE) on SymPy expressions generated by our specification before generating the C++ code, thus providing the C++ compiler with pre-optimized expressions. Compiling a specification for the queries presented in the previous section requires, on average, about one minute for running both the Python script and the C++ compiler on a single core.

Without the SymPy CSE stage, the C++ compiler can do the optimization, with little effect on the efficiency of the final code, but compilation time can increase dramatically. For instance, for the problem of geometrical validity of tetrahedra of order three (which involves large polynomials), our total generation+compilation time on a single core is $\sim 130s$, but increases to about 20 minutes if SymPy's CSE is disabled.

4.5 Discussion

We introduced MiSo, a domain-specific language and compiler to generate efficient and robust C++ code for solving a plethora of problems in graphics and geometry processing.

We believe our tool will benefit the graphics community in several ways: (1) it produces robust and provably conservative solvers with minimal user effort, (2) it can be used as a reasonably efficient conservative ground truth to validate ad-hoc accelerated algorithms, (3) it enable quick prototyping of Solve and Minimize algorithms variants, extending the benefits to other classes of problems.

With our approach, the historically cumbersome task of inventing, developing, and testing ad-hoc collision detection algorithm for each pair of primitives [6] can be automated: we hope with future improvements of MiSo (for example using the strategy proposed in [24] for time-dependent problems) that the runtime could become competitive or even surpass manually implemented codes.

Chapter 5

A C++ Library for Fast and Correctly Rounded Interval Arithmetic

In the previous chapters we presented conservative geometric predicates based on interval arithmetic, and tools to implement them effectively on a machine. In this chapter, we zoom in on the implementation of interval arithmetic (IA), and introduce a novel C++ library for fast and accurate IA.

As the width of an interval grows during computation, it is crucial to minimize this propagation of uncertainty to maintain precision. This is achieved through *correctly rounded* (CR) operations, where each elementary operation returns the tightest possible floating-point interval that contains the exact result. Apart from CR operations yielding improved precision, correct rounding can ensure *bit-by-bit reproducibility*: because the result to be returned is well-defined, an expression evaluated with correct rounding will return the same result on any machine, and with any correctly-rounded implementations. Without this property, brittle code is subject to unexpected bugs that are hard to replicate, and results that are not consistent over time due to improvements to the underlying mathematical libraries.

The 2019 revision of the IEEE 754 standard for floating point arithmetic requires a compliant implementation of a function to round correctly for all inputs [2]. Indeed, required operations such as summation, subtraction, multiplication, division, and square roots produce the same, correctly rounded results on any IEEE 754-compliant machine. However, this is not true for *recommended* functions like $\sin$ or $\log$: because they are not mandatory, mathematical libraries are allowed to implement fast, non-CR routines that are not IEEE 754-conforming, but the language implementation as a whole will be conforming as long as the mandatory operations are CR. As a result, most operations in existing mathematical libraries are not correctly rounded. We are not aware of any existing libraries that guarantee the creation of as-tight-as-possible intervals when the expressions involve this kind of operations. See Chapter 2 for further discussion.

In this chapter, we describe the design principles of our TIGHT library for correctly rounded interval arithmetic, which always produces as-tight-as-possible intervals, is faster than any existing interval library, and supports transcendental functions. Our original contributions include:

1. We extend the NFG library [12] – which provides the most efficient implementation of interval arithmetic to date, but is limited to algebraic operations – with transcendental operations, based on the CORE-MATH floating-point correctly rounded implementation [123]. In Section 5.1, we discuss the challenges involved in extending transcendental operators to intervals while guaranteeing correct rounding.
2. We integrate our library with the Domain Specific Language MiSo presented in Chapter 4, which supports the fast prototyping of non-linear constraint solving and optimization. Extension of the language with transcendental functions largely broadens its spectrum of applicability.
3. We demonstrate the effectiveness and efficiency of our library by implementing surface-surface intersection between non-algebraic surfaces, and continuous collision detection between geometric primitives undergoing roto-translational motion. In Section 5.2, we also compare our library against the popular Filib++ library [85, 86], achieving a faster performance.

5.1 Implementing elementary functions

TIGHT's interval class wraps the NFG interval library [12], which efficiently supports CR interval computation, limited to the four basic arithmetic operations, the square, and the square root, i.e., those floating-point operations for which the IEEE 754 standard prescribes correct rounding. We extend the scope of the library to also support transcendental functions, exploiting the results of the CORE-MATH Project [123], which provides CR floating-point implementation of the most common transcendental functions.

Our major contribution consists of implementing CR *interval* functions also for those transcendental functions (Figure 5.1), and providing a whole support to both polynomial and transcendental interval computation within a unified context. The functions currently supported by TIGHT intervals are:

- basic arithmetic: $x + y$, $x - y$, xy, x/y, $-x$, $|x|$, $\max(x, y)$, $\min(x, y)$;
- power functions: x^2, $\sqrt{x}$, $\sqrt[3]{x}$, $1/\sqrt{x}$, and the generic x^y;
- trigonometric functions and their inverses: $\sin(x)$, $\cos(x)$, $\tan(x)$, $\arcsin(x)$, $\arccos(x)$, $\arctan(x)$;

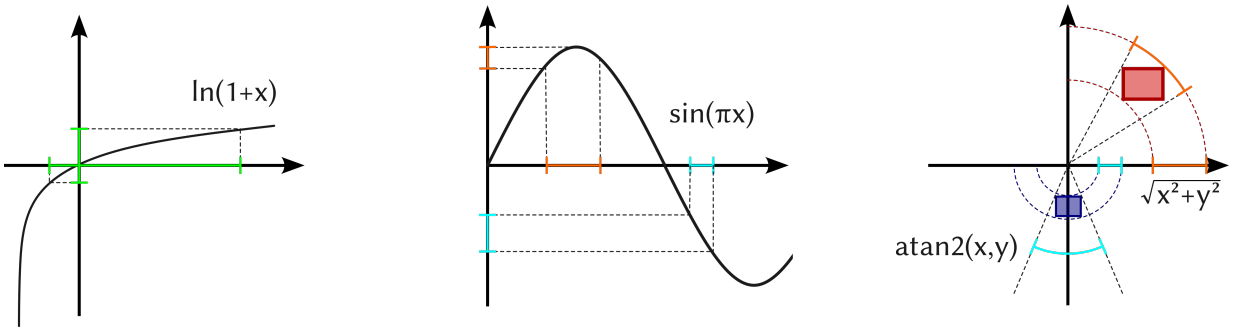


Figure 5.1: Conservative evaluation of transcendental functions on a floating-point interval: (left) for a monotonic function, it is sufficient to properly round – one up and the other down – the values computed at the two endpoints of the interval; (center) for a non-monotonic function, we need to know the values of the maxima and minima within the interval (center); we also support common functions with two arguments, such as the distance of a 2D point from the origin, and the atan2 function (right). All floating-point evaluations are correctly rounded to warrant the tightest interval about the real value.

- trigonometric functions with scaled argument: $\sin(\pi x)$, $\cos(\pi x)$, $\tan(\pi x)$, $\arcsin(x)/\pi$, $\arccos(x)/\pi$, $\arctan(x)/\pi$;
- hyperbolic functions and their inverses: $\sinh(x)$, $\cosh(x)$, $\tanh(x)$, $\operatorname{arcsinh}(x)$, $\operatorname{arccosh}(x)$, $\operatorname{arctanh}(x)$;
- exponentials in base e, 2 and 10: e^x, 2^x, 10^x, $e^x - 1$, $2^x - 1$, $10^x - 1$;
- logarithms in base e, 2 and 10: $\log(x)$, $\log_2(x)$, $\log_{10}(x)$, $\log(1+x)$, $\log_2(1+x)$, $\log_{10}(1+x)$;
- functions to convert to polar coordinates: $\sqrt{x^2 + y^2}$, $\operatorname{arctan2}(y, x)$;
- the error functions $\operatorname{erf}(x)$, $\operatorname{erfc}(x)$.

5.1.1 Interval extension

Assuming the availability of a correctly rounded function f, we want to obtain a correctly rounded inclusion function for f, that is, an interval-valued function $\square f$ such that the endpoints of the resulting interval are correctly rounded outward.

If an input contains NaNs, infinity, or points that are outside the domain of f, we return a NaN interval. Other libraries opt to provide some extended definition of operators to handle

such inputs; we postpone this to future work. In the following, we limit our discussion to valid inputs.

The challenge of extending a function to intervals with correct rounding is twofold. First, we need an expression for the range of the function; this is obtained by enumerating the possible cases for a given function. Then, these expressions must be instantiated on the input datum and rounded correctly - downward for the lower bound and upward for the upper one.

5.1.1.1 Computing extensions

When f is monotonically increasing on $[\underline{x}, \overline{x}]$, the range of the function on an interval is easily obtained as $\square f([\underline{x}, \overline{x}]) = [f(\underline{x}), f(\overline{x})]$ (Figure 5.2 right); if it is monotonically decreasing, the two endpoints are swapped. If f is not monotonic on $[\underline{x}, \overline{x}]$, we need to know where f attains its extrema on the interval. As we will see for some functions, even deciding that $[\underline{x}, \overline{x}]$ lies in a part of the domain where f is monotonic is tricky in floating-point arithmetic.

Once we know how to compute the range of the function in exact arithmetic, correct rounding amounts to rounding the left endpoint down and the right endpoint up. Given that we are operating in round-upward mode, the latter is free. To round down we can always change the rounding mode and reset it after the operation. However, changing rounding modes flushes the CPU pipeline, thus it is a relatively expensive operation. Fortunately, it can be avoided in most cases:

1. If f is odd, the result of $f(x)$ rounded down can be computed in upward rounding mode as $-f(-x)$, since negation is always exact (it only changes the sign bit). This is the same technique used by NFG for the basic arithmetic operations.
2. For non-odd functions, if we know the values of x for which $f(x)$ is representable in floating-point, we can check *a priori* if (upward) rounding happens, and if it does, we take the next smaller floating-point value.
3. Only for the remaining functions, we do change the rounding mode to downwards rounding, and reset it to upwards after computing the lower bound.

5.1.1.2 Computing $\sqrt{x}$

The square root is supported natively by NFG, since IEEE 754 mandates correct rounding for it, and so no additional implementation is required on our part.

5.1.1.3 Computing $\arcsin(x)$, $\arctan(x)$, $\arcsin(x)/\pi$, $\arctan(x)/\pi$, $\sinh(x)$, $\tanh(x)$, $\operatorname{arcsinh}(x)$, $\operatorname{arctanh}(x)$, $\sqrt[3]{x}$, and $\operatorname{erf}(x)$

All these functions are odd and monotonic. For an odd, monotonically increasing function f for which we have access to a CR implementation, the correctly rounded $\square f$ is $\square f([\underline{x}, \overline{x}]) = [-f(-\underline{x}), f(\overline{x})]$.

5.1.1.4 Computing $\tan(x)$ and $\tan(\pi x)$

Because the tangent is an odd function, rounding down is not a problem. The only difficulty in this case is that x may be an interval that crosses one of the vertical asymptotes of the function, so additional checks are needed. Our logic works as follows (Figure 5.2):

1. if x has a width of at least a period, then it must contain a singularity and we stop (Figure 5.2 left);
2. otherwise, we compute the function at the endpoints, and because the two points differ by less than a period, x can only contain a singularity if $\tan(\underline{x}) > \tan(\overline{x})$, in which case we stop (Figure 5.2 center);
3. if the previous tests passed, we return the intervals with the endpoints we already computed (Figure 5.2 right).

The same technique applies to the scaled version.

5.1.1.5 Computing $\arccos(x)$, $\arccos(x)/\pi$, $\operatorname{arccosh}(x)$, e^x, $e^x - 1$, 2^x, $2^x - 1$, 10^x, $10^x - 1$, $\log(x)$, $\log(1 + x)$, $\log_2(x)$, $\log_2(1 + x)$, $\log_{10}(x)$, $\log_{10}(1 + x)$ and $1/\sqrt{x}$

For these monotonic functions we can devise a fast test that checks whether the image of x is exactly representable. Denote $R \subset \mathrm{Dom} f$ the set of FP values such that $f(R) \subseteq \mathbb{F}$. The lower and upper bound are both computed with upward correct rounding; if $x \in R$, the lower bound is left unchanged, otherwise it is changed to the next smaller representable value.

To get the next smaller representable value, we could use the `nextafter` function offered by the standard library, which operates directly on the bit representation of the number. However, because we are using upward rounding across our program, it is slightly faster to compute the floating point number immediately before y as $-(\epsilon - y)$ where ϵ is the smallest positive representable number.

These representable sets can be determined analytically for each function:

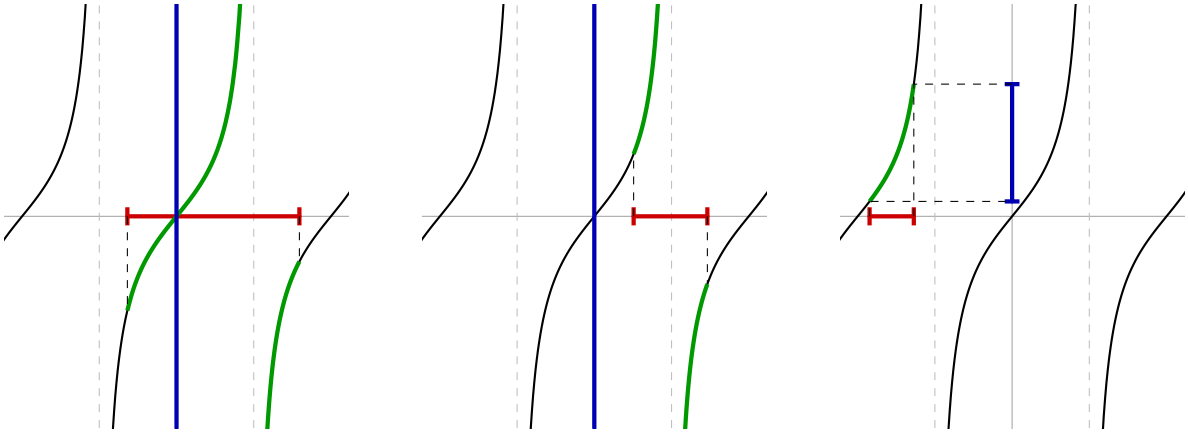


Figure 5.2: The three cases for $\tan(x)$: (left) the interval spans at least a full period, guaranteeing a singularity; (center) the interval straddles an asymptote, detected when $\tan(\underline{x}) > \tan(\overline{x})$; (right) the interval lies within a single monotone branch, yielding a bounded result. In the first two cases the result is $(-\infty, +\infty)$.

- For e^x, $e^x - 1$, and $\log(1+x)$, the only element of R is 0. By the Lindemann-Weierstrass theorem, e^x is transcendental for any non-zero algebraic x; since all FP numbers are rational, hence algebraic, no non-zero FP input can produce a FP output for e^x. For $e^x - 1$: if $e^x - 1$ were FP for some $x \neq 0$, then e^x would be rational, contradicting the above. For $\log(1+x)$: if $\log(1+x) = y$ were a non-zero FP number, then $1 + x = e^y$ would be transcendental, contradicting x being FP.

- For $\arccos(x)$, $\operatorname{arccosh}(x)$, and $\log(x)$, only $x = 1$ belongs to R: if the output y were a non-zero FP number, the respective inverse relations $x = \cos(y)$, $x = \cosh(y)$, $x = e^y$ would make x transcendental by Lindemann-Weierstrass, contradicting x being FP; the only remaining case is $y = 0$, giving $x = 1$.

- For $\arccos(x)/\pi$, $R = \{-1, 0, 1\}$: by Niven's theorem, $\arccos(x)/\pi$ is rational only for $x \in \{-1, -\frac{1}{2}, 0, \frac{1}{2}, 1\}$, giving output values $\{1, \frac{2}{3}, \frac{1}{2}, \frac{1}{3}, 0\}$; of these, only $\{0, \frac{1}{2}, 1\}$ are exactly representable in floating point, so only the corresponding inputs $\{-1, 0, 1\}$ belong to R.

- For 2^x, $R = \{k \in \mathbb{Z} : -1074 \leq k \leq 1023\}$: 2^k is exactly representable for $k \in [-1022, 1023]$ as a normalized double, and also for $k \in [-1074, -1023]$ as a subnormal, since $2^k = 2^{k+1074} \cdot 2^{-1074}$, where 2^{k+1074} is an integer with a single bit set at position $k + 1074 \in [0, 51]$, fitting exactly in the 52-bit mantissa.

- For $2^x - 1$, $R = \{k \in \mathbb{Z} : -53 \leq k \leq 53\}$: $2^k - 1$ ceases to be representable below $k = -53$ (since $1 - 2^{-54}$ rounds to 1) and above $k = 53$ (since $2^{54} - 1$ is odd and falls in

the spacing-2 region $[2^{53}, 2^{54})$, rounding to 2^{54}); $2^{53}-1$ is still representable as it uses all 53 significant bits.

- For 10^x, $R = \{k \in \mathbb{Z} : 0 \le k \le 22\}$: $10^k = 2^k \cdot 5^k$ is exactly representable only when 5^k fits in the 52-bit mantissa, which fails at $k = 23$; negative powers 10^{-k} for $k > 0$ are also not exactly representable.
- For $10^x - 1$, $R = \{k \in \mathbb{Z} : 0 \le k \le 15\}$: once the spacing between adjacent doubles near 10^k exceeds 1 (which happens at $k = 16$), the value $10^k - 1$ is no longer distinguishable from 10^k in floating point.
- For $\log_{10}(x)$, $R = \{10^k : k \in \mathbb{Z},\ 0 \le k \le 22\}$, by the same reasoning as 10^x.
- For $\log_2(x)$, $R = \{2^k : k \in \mathbb{Z},\ -1074 \le k \le 1023\}$: $\log_2(2^k) = k$ is an integer, hence representable, and 2^k covers all representable powers of 2 in double precision.
- For $\log_2(1+x)$, $R = \{2^k - 1 : k \in \mathbb{Z},\ 0 \le k \le 53\}$: the result equals the integer k when $1 + x = 2^k$, i.e. $x = 2^k - 1$; above $k = 53$, the value $2^k - 1$ is odd but falls in a region where doubles are spaced by at least 2, so it rounds to 2^k and is not representable.
- For $\log_{10}(1+x)$, $R = \{10^k - 1 : k \in \mathbb{Z},\ 0 \le k \le 15\}$: the result equals k when $x = 10^k - 1$, and the same spacing argument as for $10^x - 1$ applies.
- For $1/\sqrt{x}$, $R = \{2^{2k} : k \in \mathbb{Z},\ -537 \le k \le 511\}$: $1/\sqrt{2^{2k}} = 2^{-k}$ is exactly representable, and 2^{2k} is representable when $2k \in [-1074, 1023]$.

5.1.1.6 Computing $\cosh(x)$ and $\sqrt{x^2+y^2}$

The hyperbolic cosine is the only single-argument U-shaped function in our set (except x^2, which is supported by NFG already). To compute it, we take the absolute value of x (defined as the interval of all possible values of $|y|$ for all $y \in x$) and see if it contains 0. If it does, the lower bound is 1; otherwise it is $\cosh(\underline{|x|})$ rounded down as in the previous section (unconditionally, since $0 \notin |x|$ implies the lower bound is not representable). The upper bound is simply $\cosh(\overline{|x|})$.

In terms of how it is handled, the Pythagorean sum (also known as `hypot`) is similar to $\cosh(x)$, but in two variables. It returns the distance of the closest and farthest point in the 2D box from the origin, rounded down and up respectively (Figure 5.1 right). We compute $|x|$ and $|y|$ and test if any of them contains zero; in such case, the minimum distance is simply $\max(\underline{|x|}, \underline{|y|})$, where at least one of these numbers is zero. Otherwise, since we have no fast and reliable way of knowing whether the distance is a representable number, we resort to changing the rounding mode to compute the lower bound. The upper bound is easily computed in any case with a call to the CORE-MATH function.

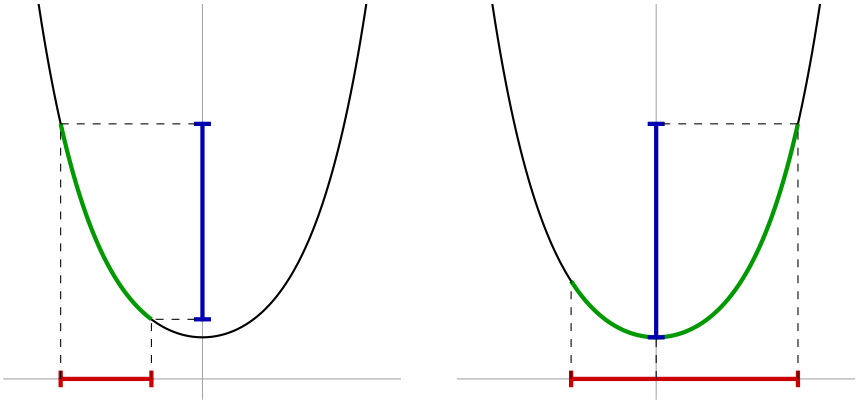


Figure 5.3: The hyperbolic cosine $\cosh(x)$. (left) the interval is in a monotone region; (right) the interval contains 0, therefore the lower bound is 1.

5.1.1.7 Computing $\sin(x)$, $\cos(x)$, $\sin(\pi x)$ and $\cos(\pi x)$

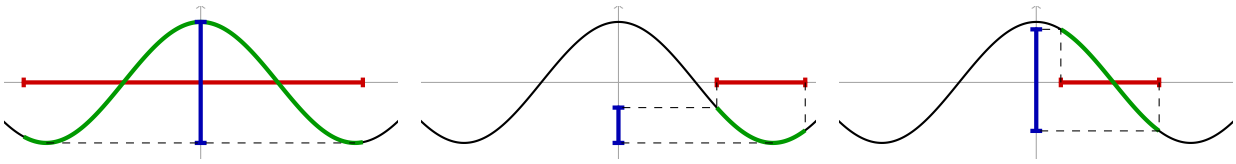


Figure 5.4: Three cases for $\cos(x)$: (left) the interval spans more than one period, so the result is $[-1, 1]$; (center) the interval contains a minimum, giving lower bound -1; (right) the interval is in a monotone region.

Perhaps surprisingly, sines and cosines are the hardest functions to implement.

Let us discuss rounding issues first. $\sin(x)$ and $\sin(\pi x)$ are odd and rounding down is trivial. We know that the only value of x for which $\cos(x)$ is a FP number is 0, so we can round down as in Section 5.1.1.5. For $\cos(\pi x)$, the set R consists of rational numbers with denominator 2, so we check if $2x$ is integer (multiplication by 2 is exact) and round down as before.

To compute the interval inclusion, we first check if x is larger than a period of f, in which case we know it must contain at least two consecutive extrema, and we can return $[-1, 1]$ (Figure 5.4 left). Otherwise, we need to know the sign of f' at the two endpoints of x. If the signs are equal, then x contains either 0 or 2 critical points; specifically, it contains 2 critical points if its width is at least π (and again we return $[-1, 1]$), otherwise it contains none. If the signs are different, then x contains a single critical point. This leaves us with four cases to consider:

- x lies entirely within part of the domain where f is monotonically increasing, so we return $[f(\underline{x})_-, f(\overline{x})]$;

- x lies entirely within part of the domain where f is monotonically decreasing, so we return $[f(\overline{x})_-, f(\underline{x})]$ (Figure 5.4 right);
- x contains a minimum, so we return $[-1, \max(f(\underline{x}), f(\overline{x}))]$ (Figure 5.4 center);
- x contains a maximum, so we return $[\max(f(\underline{x})_-, f(\overline{x})_-), 1]$.

The cases where x has an endpoint at an extremum do not require special handling. This leaves us with the issue of how to efficiently compute the sign of f' at the endpoints of x.

For $\sin(\pi x)$ and $\cos(\pi x)$, the derivative changes sign at integer multiples of $1/2$, so we can check the value of $\lceil 2\underline{x} \rceil \bmod 4$ (and likewise for $\overline{x}$, respectively); depending on whether we are computing the sine or cosine, two of the possible values correspond to a positive derivative and the other two to a negative one. To correctly get this result, we compute the ceiling operator by converting the endpoints, multiplied by 2, to 64-bit integers; overflows are not a problem, since past the value of 2^{54} the distance between adjacent double-precision FP numbers is at least as large as a full period of the function, so non-singleton intervals will return $[-1, 1]$ anyway and can be handled as a special case.

For the regular $\sin$ and $\cos$, we cannot apply the same strategy. One would like to know the consecutive integer multiples of $\pi/2$ that contain a floating point value, but since π is irrational we would need a correctly rounded function to compute x/π, or at least πx. While such methods have been researched in previous work [19], they are not part of CORE-MATH and are beyond the scope of this work. Instead, we compare $\underline{x}$ and $\overline{x}$ with precomputed, correctly rounded multiples of $\pi/2$ in the range $[-2\pi, 2\pi]$, and if an endpoint is beyond this limit, we call the correctly rounded function corresponding to the derivative of $\cos$ or $\sin$, which is more expensive but gives correct results. For this reason, TIGHT's current implementation of $\sin$ and $\cos$ is relatively cheap if x lies in $[-2\pi, 2\pi]$ and somewhat slower otherwise.

5.1.1.8 Computing $\mathrm{erfc}(x)$

For erfc, we are not aware of an easy way to check if the image of a number is representable. The function is monotonically decreasing, so we implement it with a single change in rounding mode.

5.1.1.9 Computing x^y

We provide two versions of the power function: one where x and y are both intervals, but x is not allowed to have negative values, and one where x can be any interval but y is an unsigned integer.

For the integer power function, we have a special case for $y = 2$ that calls NFG's fast square function. For odd exponents, x^n is an odd function and the lower bound is computed by the negation trick. For even exponents, the lower bound is computed by temporarily switching the rounding mode.

For the general power function, we distinguish nine cases based on the position of the 2D box (x, y) on the $x \geq 0$ half-plane: whether x is above, below or contains 1, and whether y is above, below or contains 0. For eight out of nine cases, we only require two calls to CORE-MATH's power function to compute the result; in the worst case, i.e., when (x, y) contains $(1, 0)$, we need four. In all cases the lower bound is computed by temporarily switching the rounding mode, since determining whether x^y is representable for interval-valued x and y is not tractable in general.

5.1.1.10 Computing $\mathrm{arctan2}(y, x)$

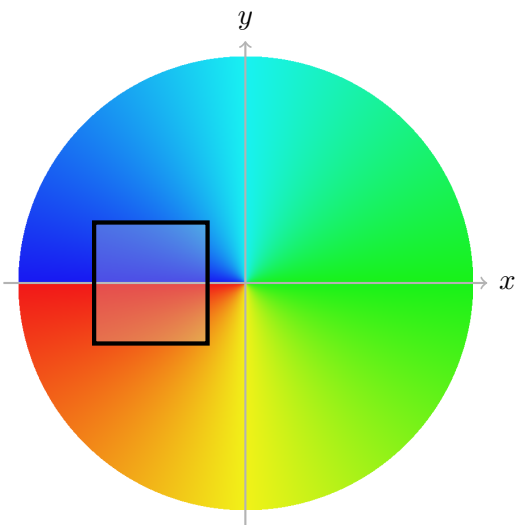


Figure 5.5: The $\mathrm{arctan2}(y, x)$ function maps each direction from the origin to an angle in $(-\pi, \pi]$, visualized as a color gradient. The seam on the negative x-axis is where the function is discontinuous: the box shows the case where $x < 0$ and y straddles 0, causing the range of angles to cross the discontinuity.

The two-argument arctangent gives the range of angles between the positive horizontal semi-axis and the line connecting the origin with points (x, y) in the plane[1], with values in $(-\pi, \pi]$ (Figure 5.1 right). It requires a similar distinction of nine cases; this time x can be negative, so we check whether each interval is above, below, or contains 0. Since $\mathrm{arctan2}(y, x)$ is odd

[1]By convention, arctan2 takes the y coordinate first and x second.

in its first argument, the lower bound is computed as $-\operatorname{arctan2}(-\underline{y}, \underline{x})$ (or the corresponding endpoint depending on the case), which is exact under upward rounding. Again, we do this to limit the number of calls to the mathematical library, and in eight out of nine cases, we perform two calls. The ninth case, where (x, y) contains $(0, 0)$, is degenerate, and we return NaN. However, one of the non-degenerate cases is very problematic, namely when x is in the negative half-space and y contains 0 (Figure 5.5). In this case, the range of angles contains the points of angle π, and the function is discontinuous. Mathematically, one should return the whole range $[-\pi, \pi]$, but the meaningful result would be the disjoint union $[-\pi, a] \cup [b, \pi]$ for some values a, b. To give meaningful results, we opt for a different solution: TIGHT's $\operatorname{arctan2}$ function returns both an interval and a boolean flag that signals the pathological case, and when the flag is on, we instead return the *complement* of the range of angles of the box, i.e., the ones that are not spanned by points in the box, with consequently inverted rounding. In the previous notation, we return $[a, b]$ with a rounded up and b rounded down. A caller that expects points in this range should then check the flag and decide how to use this result.

5.2 Results and comparison

We evaluate our library in several scenarios. Furthermore, because Filib/Filib++ is the only library which is always correct when transcendental functions are involved [133], we compare TIGHT with this representative of the state of the art.

All experiments were timed single-threaded on a server equipped with Intel Xeon Gold 6430 CPUs and 64GB of RAM. The compiler used is GCC version 13.3.0 on Ubuntu.

5.2.1 Benchmark comparison

We start by comparing the execution times and average interval width of single functions in TIGHT and Filib++. To compute the width, for each operation we select one or two singleton input intervals that lie inside the function domain, and compute the number of floating point values between the lower and upper ends of the computed interval (plus one); a width of 1 ULP (or 0, in the case of an exactly representable output) corresponds to a correctly rounded result. To get timings, we call each function one million times; to prevent the compiler from optimizing calls, and to evaluate the timing on non-singleton intervals, we perturb the input interval at each iteration by enlarging it by an ULP on both sides, and sum the upper and lower bound of each result to a dummy accumulator. Note that some operations are omitted for Filib++ as they are not supported in the library.

Estimated times for executing one interval operation with the two libraries are reported in Table 5.1. Times are in nanoseconds. Note that, for the simplest operations that cost about or even

Table 5.1: Benchmark for single interval operations: times to execute a single operation (in nanoseconds) with TIGHT and Filib++ and the related width in ULPs of the interval returned when the operands are singleton intervals of the type $[x, x]$.

Op.	Avg. ns		width in ULPs		Op.	Avg. ns		width in ULPs	
	TIGHT	Filib++	TIGHT	Filib++		TIGHT	Filib++	TIGHT	Filib++
+	0.28	1.08	1	1	acosh	31.79	15.58	1	35
-	0.34	0.71	0	0	atanh	22.39	25.84	1	27
*	0.66	1.13	1	1	exp	14.84	17.68	1	9
/	0.89	1.30	1	1	exp2	17.90	18.59	1	9
min	0.30	-	0	-	exp10	19.61	18.28	1	10
max	0.30	-	0	-	expm1	15.06	10.51	1	14
abs	1.45	2.11	0	0	exp2m1	66.32	-	1	-
sin	98.20	17.34	1	23	exp10m1	78.21	-	1	-
cos	98.48	17.50	1	28	log	39.91	10.89	1	9
tan	128.41	18.59	1	52	log2	15.81	11.14	1	50
asin	36.49	13.99	1	39	log10	58.41	13.76	1	44
acos	20.15	13.44	1	36	log1p	15.36	14.59	1	12
atan	38.41	10.33	1	28	log2p1	61.70	-	1	-
sinpi	30.12	-	1	-	log10p1	69.79	-	1	-
cospi	31.37	-	1	-	sqrt	2.18	3.78	1	2
tanpi	29.37	-	1	-	cbrt	18.78	-	1	-
asinpi	40.24	-	1	-	rsqrt	21.32	-	1	-
acospi	44.97	-	1	-	erf	60.70	-	1	-
atanpi	41.97	-	1	-	erfc	62.67	-	1	-
sinh	10.84	11.19	1	20	pow	136.80	36.31	1	15
cosh	12.94	34.28	1	11	hypot	39.20	-	1	-
tanh	20.08	10.90	1	23	atan2	28.07	-	1	
asinh	8.33	32.40	1	19					

less than one nanosecond, estimates are not fully reliable and may change on different runs. The values reported in the table for those operations are averages over different runs. However, we consistently had at all runs faster times with TIGHT, for all those operations that are already supported in NFG. For the newly implemented transcendental functions, times are comparable, but sometimes they are slower with TIGHT. This is the cost of all additional operations that we undergo to guarantee correct rounding, which is not guaranteed by Filib++.

The latter issue is evident by looking at the two rightmost columns in Table 5.1, which report the width of the resulting interval in ULPs (number of contiguous floating-point values), when the input consists of a singleton $[x, x]$. While TIGHT always produces intervals with a width of either zero or one ULP, hence correctly rounded, Filib++ produces quite large intervals in many cases (it is still correctly rounded, but slower, for algebraic operations).

Figure 5.6: Test problem: CCD of a triangle moving rigidly along a screw trajectory, and another deforming triangle. The triangle on top rotates and moves along an axis until the two primitives come into contact.

5.2.2 Comparison with Filib++ within MiSo

To assess the practical effectiveness of our library, we integrate it into MiSo (presented in Chapter 4), which makes intensive use of interval arithmetic. Since MiSo's architecture allows for easily switching the numeric backend, we were able to perform a side-by-side comparison of our library and Filib++ for several problems, while extending the software's support to transcendental operators.

The problems presented in the following involve heavy use of trigonometric functions that, as outlined above, are more expensive to compute in TIGHT than in Filib++. However, as we shall see, TIGHT's result in faster solution times overall, thanks to faster handling of basic arithmetic and its smaller intervals.

5.2.2.1 CCD along non-algebraic trajectories

We consider the classical problem of continuous collision detection (CCD): given two primitives in space with some prescribed trajectory, we seek the first time in $[0, 1]$ for which the primitives first come into contact, and call it t^*. This type of test is essential in the simulation field to guarantee that physical objects do not interpenetrate. In practice, finding the exact value of t^*

may be infeasible and often unnecessary. With interval methods, we instead seek an interval T^* that is guaranteed to contain t^* and is smaller than a user-specified precision $\delta > 0$. Then the lower bound of T^* tells us a moment in time until which we can safely move the objects without collisions, and the upper bound gives a moment when a collision has surely happened already. The difficulty of the query depends on the type of primitives, the type of trajectory, and the precision required.

Within MiSo, this is formulated as a MINIMIZE problem, that is, a constrained global optimization. In this case the optimization variables are $\{\mathbf{u}, \mathbf{v}, t\}$, where $\mathbf{u} = (u_0, u_1)$ and $\mathbf{v} = (v_0, v_1)$ are the parametric coordinates of the two primitives, and t is time; the constraints are the domain constraints $(u_0, u_1, v_0, v_1, t) \in [0, 1]^5, u_0 + u_1 \leq 1, v_0 + v_1 \leq 1$ (which are all implicit in MiSo), and the collision constraint $d(x, y) < \epsilon$ with a small but positive ϵ – i.e., we only consider pairs of points for which collision happens. The objective function is t – i.e., we want to find the collision that happens earliest.

In our test, we consider two moving linear (i.e., flat) triangles, where one is linearly deforming (i.e., each vertex follows a linear trajectory independent of the others), while the other vertex is undergoing a rigid roto-translation, following a spiral motion (Figure 5.6). More precisely, the center of the second triangle follows the spiral, and its normal remains aligned with the spiral's tangent. Of course, computing the roto-translation requires trigonometric functions, while all other computations involved are algebraic.

We test the same problem with different sets of parameters, changing the speed of the rotation and the position of the other triangle. The queries implemented with TIGHT take 27ms, 133ms and 2.5s respectively, versus the 127ms, 756ms and 13.2s of Filib++, for a speedup of approximately $5\times$.

5.2.2.2 Intersection of two parametric tori

Computing the intersection of parametric surfaces, also known as surface-surface intersection (SSI), is an important operation in CAD. In our example, we want to describe the intersection locus of two tori expressed parametrically (Figure 5.7). Again, the parametric representation of each torus requires computing trigonometric functions.

SSI can be formulated in MiSo as a SOLVE problem, that is, the problem of covering the set of all solutions of a constraint system within a certain tolerance. Being a conservative method, the algorithm returns a region that is larger than the true solution, but is guaranteed to contain every point of it.

Similarly to the previous example, the optimization variables are $\{\mathbf{u}, \mathbf{v}\}$, where $\mathbf{u} = (u_0, u_1)$ and $\mathbf{v} = (v_0, v_1)$ are the parametric coordinates of the two tori; the constraints are the domain constraints $(u_0, u_1, v_0, v_1) \in [0, 1]^4$ (implicit in MiSo), and the intersection constraint $d(x, y) < \epsilon$. This formulation gives solutions in *parameter* space rather than physical space. To be precise,

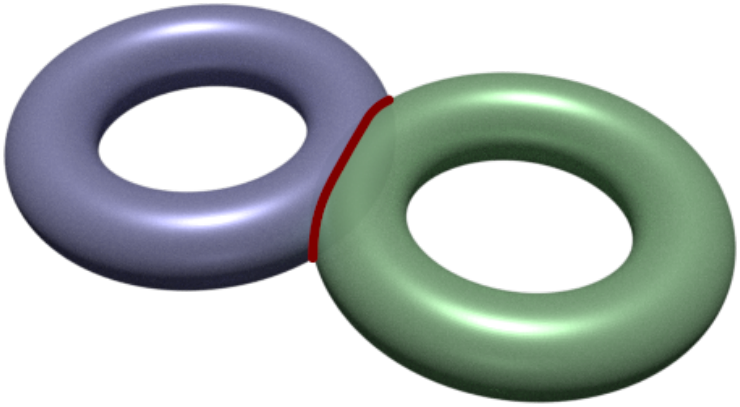


Figure 5.7: Test problem: surface-surface intersection (SSI) of two parametric tori of inner radius 0.3 and outer radius 1, each shifted by ± 1.1 along the x axis.

this formulation solves a more general version of the problem: since the parameter space is 4-dimensional, the solution is a collection of 4D boxes that describe pairs of contacting regions, up to the specified tolerance.

In both cases, the solver performs 11105 iterations and found a solution composed of 6976 4D boxes. However, the result took 136ms to compute with TIGHT and 2186ms with Filib++, a speedup of about $16\times$.

5.2.3 Pathological cases

Not producing tight enough intervals can have unpredictable results for seemingly easy problems. We provide a very simple example where returning a slightly larger interval results in the program being unable to compute the result.

Suppose we have a function $f(x, y)$ defined on the plane that is inversely proportional to the distance of point (x, y) to a circle centered in the origin with radius 5 (Figure 5.8). Thus, we are evaluating $f(x, y) = 1/(\sqrt{x^2 + y^2} - 5)$. This type of function resembles contact potentials used in IPC physical simulations [88]. We want to evaluate f with interval arithmetic at point $P(3+10^{-15}, 4+10^{-15})$, so that P lies outside the sphere. Remember that, because we are using interval arithmetic, we need not fear that adding a small value leads to cancellation for very small ϵ: in those cases we simply get an interval that contains the true value. We compute this function in three ways:

- with TIGHT intervals, and using the function `hypot` to compute the Euclidean distance of P from the origin (which is absent in Filib++);
- with TIGHT intervals, using only operators which are also supported by Filib++ (all al-

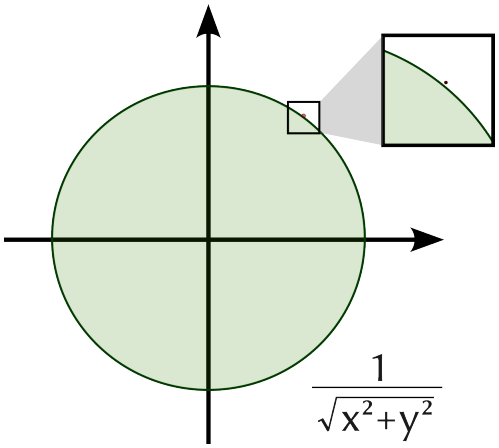


Figure 5.8: An example where tighter intervals of CR functions prevent errors: when the inputs to the problem are a few ULPs away from producing pathological situations, CR minimizes error and is often able to avoid failure. Moreover, its results are machine-independent; a non-CR implementation may complete successfully on one machine but fail on another.

gebraic in this case);

- with Filib++ intervals, using the same mode as the rest of the chapter.

In the first two cases, the denominator is correctly computed as positive with TIGHT and the operation returns a real interval (albeit larger in the second case, due to multiple operations being involved). When using Filib++, the square root produces an interval with lower bound equal to the radius, resulting in a division by zero.

While this example involves only a few operations, for more complex expressions, the propagation of error can be even more dramatic. Correct rounding does not eliminate the issue, but it reduces propagation to a minimum and makes it predictable, since each operation can introduce at most 1 ULP of error on each side.

5.3 Discussion

We have introduced TIGHT, an efficient C++ library for correctly rounded interval arithmetic built on top of the state-of-the art in both correctly rounded computations and interval arithmetic. Thanks to its correctly rounded nature, it can be used to ensure that applications yield consistent results across machines, and to facilitate reproducibility in academia. TIGHT is designed to be future-proof: if correctly rounded mathematical routines become the standard in the coming years, or more performant CR methodologies are developed, switching the underlying library requires minimal changes. Crucially, since the result of a CR operation is well

defined, TIGHT will continue to return the same results *forever* on all machines, even in the event of a library change.

We hope that our library will become a useful tool for researchers and developers not only in robust geometry processing, but also in other fields where correct and consistent predicates are a priority.

Chapter 6

Conclusion

6.1 Summary of contributions

The main argument of this thesis is that reliability in geometric computing should not be a luxury reserved for niche applications, nor should it require prohibitive trade-offs in terms of performance or development effort. We have presented a vertical stack of tools designed to address key robustness challenges in geometry processing and physical simulation, moving from the fundamental arithmetic level up to high-level algorithmic abstractions and specific applications in finite element analysis.

Our first contribution, presented in Chapter 3, addressed the specific problem of continuous geometrical validity for high-order finite elements. As modern simulation pipelines increasingly adopt high-order bases to improve accuracy, the lack of robust validity checks has become a significant source of instability. Existing methods, often relying on point-wise checks at quadrature locations, fail to detect local inversions that occur between samples or between time steps. We introduced the first algorithm capable of conservatively verifying the validity of deforming curved elements. By coupling this check with an invalidity-aware adaptive quadrature scheme, we demonstrated that it is possible to eliminate element inversion artifacts in elastodynamic simulations, thereby preventing non-physical states and solver divergence.

Recognizing that manually implementing such robust predicates is error-prone and tedious, our second contribution, MiSo (Chapter 4), generalized this approach. MiSo is a Domain Specific Language and compiler that automates the generation of conservative solvers for a broad class of Solve and Minimize problems. By abstracting the mathematical definition of a geometric query from its numerical implementation, MiSo allows researchers and engineers to prototype and deploy robust algorithms for collision detection, primitive intersection, and distance computation without needing to manage the complexities of interval analysis or domain decomposition manually. We demonstrated that solvers generated by MiSo are competitive

with, and in some cases faster than, hand-optimized code, while providing strict guarantees of correctness.

Finally, in Chapter 5, we addressed the foundational need for accurate numerical primitives with the TIGHT library. While interval arithmetic is a cornerstone of conservative computing, its practical application has often been hampered by slow performance and a lack of support for transcendental functions. TIGHT provides a high-performance, correctly rounded interval arithmetic library that extends the state of the art to include a vast set of functions, and serves as the numerical backend for MiSo.

6.2 Discussion and impact

The tools presented in this thesis collectively argue for a shift in how we approach geometric robustness. Rather than relying on the ad-hoc adjustment of tolerance thresholds, we advocate for the use of guaranteed conservative bounds.

The continuous geometrical validity check highlighted the previously overlooked issue that high-order finite element solvers can produce invalid intermediate states that are simply missed by standard checks. This discovery has significant implications for the reliability of simulation software for elastodynamics. By providing a method to detect and prevent these states reliably, we enable the development of safer simulation tools. A recent paper by Du et al. [38] proposes an acceleration strategy for our validity check, showing a growing interest from the geometry processing and simulation communities in the development of robust solutions.

The goal of MiSo is to facilitate access to robust geometric algorithms via automatic code generation. This allows domain experts to focus on the mathematical description of their problems, while the tool ensures the numerical robustness of the implementation and enables them to rapidly experiment with different interval-based solutions. MiSo also makes it possible to easily transfer future improvements to these interval-based algorithms to a wide range of problems.

Finally, we believe that our TIGHT library will be useful tool to push the adoption and development of interval methods in scientific computing. Because of its correctly-rounded properties, the approximations produced by the same sequence of calculations will not change over time, simplifying the reproducibility of experiments and guaranteeing consistency; and thanks to its speed, we hope to see more researchers and practitioners using it to write truly reliable code.

6.3 Limitations

While our methods provide strong guarantees, these come at a price. A primary constraint of the subdivision-based approaches used in both High Order Continuous Geometrical Valid-

ity and MiSo is the curse of dimensionality. The computational cost of domain decomposition grows exponentially with the number of variables. While this is acceptable for low-dimensional geometric primitives (e.g., 3D elements, surface patches), it renders the current approach impractical for high-dimensional configuration spaces unless significant heuristics or pruning strategies are employed. Additionally, for many of the geometric queries presented, including the continuous geometrical validity check, the current method works well for relatively low element orders (e.g., linear to cubic or quartic), but the cost increases with the polynomial degree. For very high-order elements, the subdivision depth required to resolve roots may become prohibitive.

Finally, while the use of IA makes the results robust to roundoff error, the overhead of IA compared to standard FP operations is still non-negligible. Although we have minimized this cost, conservative algorithms will inevitably be slower than their non-robust counterparts. This performance gap is the price of correctness. However, as shown in our benchmarks, algorithmic optimizations such as the hybrid Bézier/NIE approach in MiSo can often mitigate this overhead, making conservative methods a viable option for offline applications, or limited use in real-time applications.

6.4 Future work

There are several avenues for future research that could improve and expand the presented works.

6.4.1 Parallelization and Hardware Acceleration

The algorithms described in this thesis are largely inherently parallelizable, as the domain decomposition process generates independent sub-problems that can be processed concurrently. The reference implementation of the validity check already provides a CPU parallel version; a promising direction is to adapt MiSo and the validity checks for execution on massively parallel architectures like GPUs. This could dramatically reduce the runtime of conservative queries.

6.4.2 Material models under extreme deformation

Our investigation into element validity revealed numerical instabilities in standard material models (like Neo-Hookean) under extreme compression, even when elements are technically valid. The discovery that these potentials can remain finite even as the Jacobian approaches zero suggests a need to re-evaluate constitutive models for robust use in interior-point solvers.

6.4.3 Expanded scope for MiSo

Currently, MiSo works best for solving low-dimensional geometric queries. Integrating time-dependent inclusion strategies, such as those proposed by Chen et al. [24], could improve performance for dynamic problems.

6.4.4 Standardization of interval arithmetic

The TIGHT library, while efficient, does not yet fully conform to the IEEE 1788-2015 standard for interval arithmetic [1]. Future development should focus on full compliance, including rigorous handling of exceptions and special values (infinity, NaN), to facilitate its adoption in the broader scientific computing community.

6.4.5 Improved support for CR functions

NFG's arithmetic operations owe their speed to vectorization. Vectorized mathematical libraries exist [121] but we are not aware of a correctly-rounded solution. A vectorized CR implementation would further increase TIGHT's efficiency for those functions. Furthermore, as mentioned in Section 5.1.1.7, integrating CR functions for multiplication with an irrational constant value as in [19] would bring a more consistent speed for $\sin$ and $\cos$ on arbitrary arguments.

6.5 Closing remarks

As simulation in safety-critical scenarios becomes widespread, the cost of failure rises dramatically. A result that looks realistic but violates physical laws is no longer just a visual artifact: it is a liability. This thesis provides a foundation upon which to build the necessary infrastructure to trust these computations, enabling scientists and engineers to simulate reality in a way that is not just visually plausible, but grounded in mathematical truth.

Appendix A

Appendix

A.1 Lagrange and Bézier forms of $|J_x|$

Let $f : \sigma_s^n \longrightarrow \mathbb{R}$ be a polynomial of order p defined on a standard element according to Definition 1. We detail the representation of f in Lagrange and Bézier forms and the conversions between these two representations.

A.1.0.1 Lagrange form

Let $\Gamma_\sigma^p = (1/p)\mathbb{Z}^n \cap \sigma$ be a grid of uniformly distributed domain points of σ, and $\mathcal{I}_\sigma^p$ be its set of indices (Figure 3.2). The Lagrange basis of order p consists of $|\Gamma_\sigma^p|$ order p polynomials such that for each point $\gamma_i \in \Gamma_\sigma^p$, $\mathcal{L}_i^p(\gamma_j) = \delta_{ij}$. A function f is represented in the Lagrange basis as:

$$f(\xi) = \sum_{i \in \mathcal{I}_\sigma^p} y_i \mathcal{L}_i^p(\xi), \tag{A.1}$$

where $y_i = f(\gamma_i)$ for $i \in \mathcal{I}_\sigma^p$.

A.1.0.2 Bézier form

aThe same function f can be expressed equivalently in Bézier form by using a Bernstein basis on the same set Γ_σ^p of domain points:

$$f(\xi) = \sum_{i \in \mathcal{I}_\sigma^p} \beta_i \mathcal{B}_i^p(\xi), \tag{A.2}$$

where the $\mathcal{B}_i^p$ are Bernstein polynomials of order p and the β_i are their corresponding control coefficients. Unlike the Lagrange case, β_i equals $f(\gamma_i)$ only at the corners of σ. However, the graph of f is contained in the convex hull of points (γ_i, β_i), providing a simple way to bound f from below and above at all points of σ [41].

A.1.0.3 Transformation matrix

Let us denote $f^{\mathcal{L}}$ the vector consisting of all the y_i, and likewise, $f^{\mathcal{B}}$ the vector consisting of all the β_i, for $i \in \mathcal{I}_\sigma^p$. We can convert between the two representations through transformation matrices [69]:

$$f^{\mathcal{L}} = \mathbf{T}_{\mathcal{B}\to\mathcal{L}} f^{\mathcal{B}} \qquad f^{\mathcal{B}} = \mathbf{T}_{\mathcal{L}\to\mathcal{B}} f^{\mathcal{L}}. \tag{A.3}$$

Such matrices depend only on the reference element σ_s^n and the order p thus can be computed once for each element type and order. Matrix $\mathbf{T}_{\mathcal{B}\to\mathcal{L}}$ is easily computed by evaluation of Bernstein polynomials on Γ_σ^p:

$$(\mathbf{T}_{\mathcal{B}\to\mathcal{L}})_{ij} = \mathcal{B}_j^p(\gamma_i) \quad \forall i, j \in \mathcal{I}_\sigma^p \tag{A.4}$$

and $\mathbf{T}_{\mathcal{L}\to\mathcal{B}}$ is the inverse of $\mathbf{T}_{\mathcal{B}\to\mathcal{L}}$.

A.1.0.4 Subdivision matrices

For each q, we first build a transformation matrix from the Bézier basis to the Lagrange basis that interpolates the domain points $\psi^q(\Gamma(m))$ of the q-th subdomain. This is analogous to building matrix $\mathbf{T}_{\mathcal{B}\to\mathcal{L}}$, by sampling the Bézier basis on $\psi^q(\Gamma(m))$ instead of $\Gamma(m)$:

$$(\mathbf{T}^q_{\mathcal{B}\to\mathcal{L}})_{ij} = \mathcal{B}_j^m(\psi^q(\gamma_i)) \quad \forall i, j \in \mathcal{I}_\sigma(m). \tag{A.5}$$

Then we multiply it with the Lagrange-to-Bézier matrix to build

$$\mathbf{T}^q_{\mathcal{B}\to\mathcal{B}} = \mathbf{T}_{\mathcal{L}\to\mathcal{B}} \mathbf{T}^q_{\mathcal{B}\to\mathcal{L}}, \tag{A.6}$$

which allows us to go directly from the Bézier coefficients on the domain to the Bézier coefficients on each subdomain.

A.1.0.5 Time subdivision matrices

Time subdivision matrices defined by plugging the time subdivision maps

$$\psi^-(\xi, t) = (\xi, t/2), \qquad \psi^+(\xi, t) = (\xi, (t+1)/2) \tag{A.7}$$

in Equation (A.5) and using Equation (A.6) as above.

A.1.0.6 Representations of the Jacobian determinant

Given a standard element σ_s^n as above, let us define $m_\triangle = np - s$ and $m_\square = np - 1$. Let us denote $i_0 = m_\triangle - (\sum_{j=1}^s i_j)$ and $\xi_0 = 1 - (\sum_{j=1}^s \xi_j)$.

The Lagrange basis polynomials used to represent the Jacobian determinant $|J_x(\xi)|$ on reference element $\sigma = \sigma_s^n$ are:

$$\mathcal{L}^\sigma_{i_1,\ldots,i_n}(\xi_1,\ldots,\xi_n) = \left(\prod_{j=0}^{s} \ell_{i_j}^{i_j,m_\triangle}(\xi_{i_j})\right)\left(\prod_{j=s+1}^{n} \ell_{i_j}^{m_\square,m_\square}(\xi_{i_j})\right) \tag{A.8}$$

$$\ell_j^{q,m}(\zeta) = \prod_{k\in\{0,\ldots,q\}\setminus\{j\}} \frac{m\zeta - k}{j-k} \tag{A.9}$$

The Bézier basis polynomials used to represent the Jacobian determinant $|J_x(\xi)|$ on reference element $\sigma = \sigma_s^n$ are:

$$\mathcal{B}^\sigma_{i_1,\ldots,i_n}(\xi_1,\ldots,\xi_n) = \left(\binom{m_\triangle}{i_0,\ldots,i_s}\xi_0^{i_0}\ldots\xi_s^{i_s}\right)\prod_{j=s+1}^{n} b_{i_j}^{m_\square}(\xi_{i_j}) \tag{A.10}$$

$$b_j^q(\zeta) = \binom{q}{j}\zeta^j(1-\zeta)^{q-j} \tag{A.11}$$

where $\mathcal{B}^\sigma_{i_1,\ldots,i_n}$ is the product of an order $m_\triangle$ Bernstein polynomial on the s-simplex basis, in the variables $\xi_1,\ldots,\xi_s$, with an order $m_\square$ Bernstein polynomial on the $(n-s)$-tensor product basis in the variables $\xi_{s+1},\ldots,\xi_n$, which is itself a product of $(n-s)$ univariate Bernstein polynomials of order $m_\square$. It is easy to see that

- the order of $|J_x(\xi)|$ in $\{\xi_1,\ldots,\xi_s\}$ is $m_\triangle = np - s$;
- the order of $|J_x(\xi)|$ in $\{\xi_{s+1},\ldots,\xi_n\}$ is $m_\square = np - 1$.

Notice that on n-simplices ($s = n$) and n-hypercubes ($s = 1$), $|J_x|$ has the same order in every variable.

Extension to Dynamic

For the dynamic case, we add a term for time to the Bézier basis construction, an order n univariate Bernstein polynomial in t:

$$\overline{\mathcal{B}}^\sigma_{i_1,\ldots,i_n}(\xi_1,\ldots,\xi_n,t) = \mathcal{B}^\sigma_{i_1,\ldots,i_n,i_t}(\xi_1,\ldots,\xi_n)b^n_{i_t}(t) \tag{A.12}$$

Element name	n	s	p	$\xi_{1\ldots s}$	$\xi_{s+1\ldots n}$	t	$\overline{N}$
Linear triangle	2	2	1	0	-	2	3
Quadratic triangle	2	2	2	2	-	2	18
Cubic triangle	2	2	3	4	-	2	45
Quartic triangle	2	2	4	6	-	2	84
Quintic triangle	2	2	5	8	-	2	135
Bilinear quadrangle	2	1	1	1	1	2	12
Biquadratic quadrangle	2	1	2	3	3	2	48
Bicubic quadrangle	2	1	3	5	5	2	108
Linear tetrahedron	3	3	1	0	-	3	4
Quadratic tetrahedron	3	3	2	3	-	3	80
Cubic tetrahedron	3	3	3	6	-	3	336
Quartic tetrahedron	3	3	4	9	-	3	880
Bilinear tri. prism	3	2	1	1	2	3	36
Biquadratic tri. prism	3	2	2	4	5	3	360
Trilinear hexahedron	3	1	1	2	2	3	108
Triquadratic hexahedron	3	1	2	5	5	3	864

Table A.1: Order of $|J_{\overline{x}}(\xi)|$ in the spatial variables ξ and the time variable t, and the number $\overline{N}$ of its terms for a generic order p map on $\overline{\sigma}^n_s$.

It is easy to see that the order of $|J_{\overline{x}}|$ remains the same as the static case in its spatial variables, while it is n in its time variable. The number of polynomials in the Bernstein basis, which is equal to the number of terms of the given polynomial, increases combinatorially with the dimension n and order p and is given by

$$N = \binom{np}{s}(np)^{n-s}, \qquad \overline{N} = N(n+1), \tag{A.13}$$

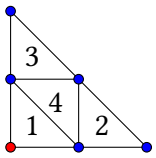
for the static and the dynamic case, respectively. Note that, this combinatorial growth poses intrinsic limitations to scaling the problem up in degree and dimension. Nevertheless, it remains tractable for the most common cases, as exemplified in Table A.1.

A.2 Subdivision rules

The subdivision functions ψ^q for the various types of elements are listed below. Their related subdomains are shown in the insets (red bullet = origin). The dynamic elements for $n = 3$ require 4D hypercubes (for tensor product) and hyper-prisms (for simplicial and mixed).

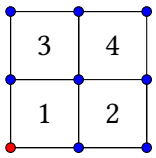
A.2.0.1 Triangle

$$
\begin{aligned}
\psi^1(\xi) &= (\xi_1/2, \xi_2/2) \\
\psi^2(\xi) &= ((\xi_1+1)/2, \xi_2/2) \\
\psi^3(\xi) &= (\xi_1/2, (\xi_2+1)/2) \\
\psi^4(\xi) &= ((1-\xi_1)/2, (1-\xi_2)/2)
\end{aligned}
$$



A.2.0.2 Quad

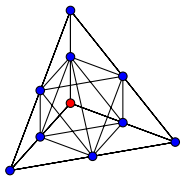
$$
\begin{aligned}
\psi^1(\xi) &= (\xi_1/2, \xi_2/2) \\
\psi^2(\xi) &= ((\xi_1+1)/2, \xi_2/2) \\
\psi^3(\xi) &= (\xi_1/2, (\xi_2+1)/2) \\
\psi^4(\xi) &= ((\xi_1+1)/2, (\xi_2+1)/2)
\end{aligned}
$$



A.2.0.3 Tetrahedron

The subdivision of the tetrahedron is non-trivial: it is split into four tetrahedra incident at the corners of the domain (corresponding to subdomains 1–4) and a central octahedron, which is further split into four tetrahedra (domains 5–8).

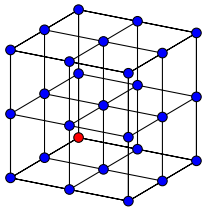
$$
\begin{aligned}
\psi^1(\xi) &= (\xi_1/2, \xi_2/2, \xi_3/2) \\
\psi^2(\xi) &= ((\xi_1+1)/2, \xi_2/2, \xi_3/2) \\
\psi^3(\xi) &= (\xi_1/2, (\xi_2+1)/2, \xi_3/2) \\
\psi^4(\xi) &= (\xi_1/2, \xi_2/2, (\xi_3+1)/2) \\
\psi^5(\xi) &= ((1-\xi_2-\xi_3)/2, \xi_2/2, (\xi_1+\xi_2+\xi_3)/2) \\
\psi^6(\xi) &= ((1-\xi_2)/2, (\xi_1+\xi_2)/2, (\xi_2+\xi_3)/2) \\
\psi^7(\xi) &= ((\xi_1+\xi_2)/2, (1-\xi_1)/2, \xi_3/2) \\
\psi^8(\xi) &= (\xi_1/2, (\xi_2+\xi_3)/2, (1-\xi_1+\xi_2)/2)
\end{aligned}
$$



A.2.0.4 Hexahedron

The subdivision of a hexahedron works both for a static hexahedral domain and a dynamic quad domain. In the latter case, the third coordinate is time.

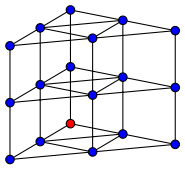
$$
\begin{aligned}
\psi^1(\xi) &= (\xi_1/2, \xi_2/2, \xi_3/2) \\
\psi^2(\xi) &= ((\xi_1+1)/2, \xi_2/2, \xi_3/2) \\
\psi^3(\xi) &= (\xi_1/2, (\xi_2+1)/2, \xi_3/2) \\
\psi^4(\xi) &= ((\xi_1+1)/2, (\xi_2+1)/2, \xi_3/2) \\
\psi^5(\xi) &= (\xi_1/2, \xi_2/2, (\xi_3+1)/2) \\
\psi^6(\xi) &= ((\xi_1+1)/2, \xi_2/2, (\xi_3+1)/2) \\
\psi^7(\xi) &= (\xi_1/2, (\xi_2+1)/2, (\xi_3+1)/2) \\
\psi^8(\xi) &= ((\xi_1+1)/2, (\xi_2+1)/2, (\xi_3+1)/2)
\end{aligned}
$$



A.2.0.5 Prism

The subdivision of a prism works both for a static prism domain and a dynamic triangle domain. In the latter case, the third coordinate is time.

$$
\begin{aligned}
\psi^1(\xi) &= (\xi_1/2, \xi_2/2, \xi_3/2) \\
\psi^2(\xi) &= ((\xi_1+1)/2, \xi_2/2, \xi_3/2) \\
\psi^3(\xi) &= (\xi_1/2, (\xi_2+1)/2, \xi_3/2) \\
\psi^4(\xi) &= ((1-\xi_1)/2, (1-\xi_2)/2, \xi_3/2) \\
\psi^5(\xi) &= (\xi_1/2, \xi_2/2, (\xi_3+1)/2) \\
\psi^6(\xi) &= ((\xi_1+1)/2, \xi_2/2, (\xi_3+1)/2) \\
\psi^7(\xi) &= (\xi_1/2, (\xi_2+1)/2, (\xi_3+1)/2) \\
\psi^8(\xi) &= ((1-\xi_1)/2, (1-\xi_2)/2, (\xi_3+1)/2)
\end{aligned}
$$



A.2.0.6 Hypercube

This is a 4-hypercube that works as a domain for a dynamic hexahedral element.

$$
\begin{aligned}
\psi^1(\xi,t) &= (\xi_1/2, \xi_2/2, \xi_3/2, t/2, t/2)\\
\psi^2(\xi,t) &= ((\xi_1+1)/2, \xi_2/2, \xi_3/2, t/2, t/2)\\
\psi^3(\xi,t) &= (\xi_1/2, (\xi_2+1)/2, \xi_3/2, t/2)\\
\psi^4(\xi,t) &= ((\xi_1+1)/2, (\xi_2+1)/2, \xi_3/2, t/2)\\
\psi^5(\xi,t) &= (\xi_1/2, \xi_2/2, (\xi_3+1)/2, t/2)\\
\psi^6(\xi,t) &= ((\xi_1+1)/2, \xi_2/2, (\xi_3+1)/2, t/2)\\
\psi^7(\xi,t) &= (\xi_1/2, (\xi_2+1)/2, (\xi_3+1)/2, t/2)\\
\psi^8(\xi,t) &= ((\xi_1+1)/2, (\xi_2+1)/2, (\xi_3+1)/2, t/2)\\
\psi^9(\xi,t) &= (\xi_1/2, \xi_2/2, \xi_3/2, t/2, (t+1)/2)\\
\psi^{10}(\xi,t) &= ((\xi_1+1)/2, \xi_2/2, \xi_3/2, t/2, (t+1)/2)\\
\psi^{11}(\xi,t) &= (\xi_1/2, (\xi_2+1)/2, \xi_3/2, (t+1)/2)\\
\psi^{12}(\xi,t) &= ((\xi_1+1)/2, (\xi_2+1)/2, \xi_3/2, (t+1)/2)\\
\psi^{13}(\xi,t) &= (\xi_1/2, \xi_2/2, (\xi_3+1)/2, (t+1)/2)\\
\psi^{14}(\xi,t) &= ((\xi_1+1)/2, \xi_2/2, (\xi_3+1)/2, (t+1)/2)\\
\psi^{15}(\xi,t) &= (\xi_1/2, (\xi_2+1)/2, (\xi_3+1)/2, (t+1)/2)\\
\psi^{16}(\xi,t) &= ((\xi_1+1)/2, (\xi_2+1)/2, (\xi_3+1)/2, (t+1)/2)
\end{aligned}
$$

A.2.0.7 Hyperprism 1

This is the tensor product of a tetrahedron with an interval that works as a domain for a dynamic tetrahedral element.

$$\psi^1(\xi,t) = (\xi_1/2, \xi_2/2, \xi_3/2, t/2)$$
$$\psi^2(\xi,t) = ((\xi_1+1)/2, \xi_2/2, \xi_3/2, t/2)$$
$$\psi^3(\xi,t) = (\xi_1/2, (\xi_2+1)/2, \xi_3/2, t/2)$$
$$\psi^4(\xi,t) = (\xi_1/2, \xi_2/2, (\xi_3+1)/2, t/2)$$
$$\psi^5(\xi,t) = ((1-\xi_2-\xi_3)/2, \xi_2/2, (\xi_1+\xi_2+\xi_3)/2, t/2)$$
$$\psi^6(\xi,t) = ((1-\xi_2)/2, (\xi_1+\xi_2)/2, (\xi_2+\xi_3)/2, t/2)$$
$$\psi^7(\xi,t) = ((\xi_1+\xi_2)/2, (1-\xi_1)/2, \xi_3/2, t/2)$$
$$\psi^8(\xi,t) = (\xi_1/2, (\xi_2+\xi_3)/2, (1-\xi_1+\xi_2)/2, t/2)$$
$$\psi^9(\xi,t) = (\xi_1/2, \xi_2/2, \xi_3/2, (t+1)/2)$$
$$\psi^{10}(\xi,t) = ((\xi_1+1)/2, \xi_2/2, \xi_3/2, (t+1)/2)$$
$$\psi^{11}(\xi,t) = (\xi_1/2, (\xi_2+1)/2, \xi_3/2, (t+1)/2)$$
$$\psi^{12}(\xi,t) = (\xi_1/2, \xi_2/2, (\xi_3+1)/2, (t+1)/2)$$
$$\psi^{13}(\xi,t) = ((1-\xi_2-\xi_3)/2, \xi_2/2, (\xi_1+\xi_2+\xi_3)/2, (t+1)/2)$$
$$\psi^{14}(\xi,t) = ((1-\xi_2)/2, (\xi_1+\xi_2)/2, (\xi_2+\xi_3)/2, (t+1)/2)$$
$$\psi^{15}(\xi,t) = ((\xi_1+\xi_2)/2, (1-\xi_1)/2, \xi_3/2, (t+1)/2)$$
$$\psi^{16}(\xi,t) = (\xi_1/2, (\xi_2+\xi_3)/2, (1-\xi_1+\xi_2)/2, (t+1)/2)$$

A.2.0.8 Hyperprism 2

This is the tensor product of a prism with an interval that works as a domain for a dynamic prism element.

$$
\begin{aligned}
\psi^1(\xi,t) &= (\xi_1/2, \xi_2/2, \xi_3/2, t/2)\\
\psi^2(\xi,t) &= ((\xi_1+1)/2, \xi_2/2, \xi_3/2, t/2)\\
\psi^3(\xi,t) &= (\xi_1/2, (\xi_2+1)/2, \xi_3/2, t/2)\\
\psi^4(\xi,t) &= ((1-\xi_1)/2, (1-\xi_2)/2, \xi_3/2, t/2)\\
\psi^5(\xi,t) &= (\xi_1/2, \xi_2/2, (\xi_3+1)/2, t/2)\\
\psi^6(\xi,t) &= ((\xi_1+1)/2, \xi_2/2, (\xi_3+1)/2, t/2)\\
\psi^7(\xi,t) &= (\xi_1/2, (\xi_2+1)/2, (\xi_3+1)/2, t/2)\\
\psi^8(\xi,t) &= ((1-\xi_1)/2, (1-\xi_2)/2, (\xi_3+1)/2, t/2)\\
\psi^9(\xi,t) &= (\xi_1/2, \xi_2/2, \xi_3/2, (t+1)/2)\\
\psi^{10}(\xi,t) &= ((\xi_1+1)/2, \xi_2/2, \xi_3/2, (t+1)/2)\\
\psi^{11}(\xi,t) &= (\xi_1/2, (\xi_2+1)/2, \xi_3/2, (t+1)/2)\\
\psi^{12}(\xi,t) &= ((1-\xi_1)/2, (1-\xi_2)/2, \xi_3/2, (t+1)/2)\\
\psi^{13}(\xi,t) &= (\xi_1/2, \xi_2/2, (\xi_3+1)/2, (t+1)/2)\\
\psi^{14}(\xi,t) &= ((\xi_1+1)/2, \xi_2/2, (\xi_3+1)/2, (t+1)/2)\\
\psi^{15}(\xi,t) &= (\xi_1/2, (\xi_2+1)/2, (\xi_3+1)/2, (t+1)/2)\\
\psi^{16}(\xi,t) &= ((1-\xi_1)/2, (1-\xi_2)/2, (\xi_3+1)/2, (t+1)/2)
\end{aligned}
$$

A.3 Failure case for floating point algorithm

We present an artificial example where floating point error causes an invalid element to be detected as valid by the non-robust implementation of the static validity check. Consider the static quadratic triangle with control points:

$$
\begin{aligned}
\mathbf{v}_{00} &= (\texttt{0.0}, \texttt{-1.0})\\
\mathbf{v}_{20} &= (\texttt{0.9999999999999997}, \texttt{-1.0})\\
\mathbf{v}_{02} &= (\texttt{-1.1093356479670479e-16}, \texttt{-3.3306690738754696e-16})\\
\mathbf{v}_{01} &= (\texttt{0.5}, \texttt{-0.75})\\
\mathbf{v}_{11} &= (\texttt{0.75}, \texttt{-0.25})\\
\mathbf{v}_{10} &= (\texttt{0.25000000000000017}, \texttt{-0.5})
\end{aligned}
$$

control points $\mathbf{v}_{00}$, $\mathbf{v}_{20}$, and $\mathbf{v}_{02}$ are the three vertices of the triangle, and $\mathbf{v}_{01}$, $\mathbf{v}_{11}$, $\mathbf{v}_{10}$ are the edge midpoints. The value of the Jacobian determinant is positive at all control points, except $\mathbf{v}_{00}$. At this

point, the Jacobian is

$$\begin{aligned} J_{00} =&(4\mathbf{v}_{01}^x - 3\mathbf{v}_{00}^x - \mathbf{v}_{20}^x)(4\mathbf{v}_{10}^y - 3\mathbf{v}_{00}^y - \mathbf{v}_{02}^y) - \\ &(4\mathbf{v}_{10}^x - 3\mathbf{v}_{00}^x - \mathbf{v}_{02}^x)(4\mathbf{v}_{01}^y - 3\mathbf{v}_{00}^y - \mathbf{v}_{20}^y) \\ =&\texttt{-1.1093356479670467e-16} < 0 \end{aligned}$$

when computed with exact rational arithmetic. However, when tested with the floating point implementation, the Jacobian determinant at this vertex evaluates to `2.220446049250313e-16` > 0; because all other Bézier coefficients are positive as well, the element is classified as valid.

Our conservative implementation takes representation error into account and does not converge for this element, meaning that the algorithm will stop after $l_{\max}$ subdivisions leaving the element undecided, and thus it will be treated as invalid.

A.4 Polyhedral bounding boxes

The following program, used in Section 4.4.2, relies on the hyperplane separation theorem: two convex objects are not intersecting if a plane separating them exists, i.e., if their projections on the normal direction of such plane are disjoint. Using the three cardinal directions of the Cartesian space only is equivalent to testing for intersections between the AABB's of the objects. We rather test against 13 directions, including the various diagonals of coordinate planes and octants; this is equivalent to enclosing the primitives into tight polyhedra, each being the convex envelope of 26 planes with fixed orientations. This is a compromise between using the looser AABB's and the tighter OBB's, which are more expensive to compute and test. The projections of the distance of the two objects onto the normal directions of the faces of such polyhedra are computed with the function `PBB3D` defined below.

This function accepts a 3D polynomial map as a list of `PolyNodes`. The result is the projection of this map on the 13 axes

$$\begin{gathered} (1,0,0),(0,1,0),(0,0,1), \\ (1,1,0),(1,0,1),(0,1,1),(1,-1,0),(1,0,-1),(0,1,-1), \\ (1,1,1),(1,1,-1),(1,-1,1),(1,-1,-1). \end{gathered}$$

that is, a list of 13 expressions such that the inclusion function of each of them encloses the projection of the geometric object on one of the chosen axes.

```
with pymiso.Context() as miso:
    def PBB3D(x):
      v = miso.vector(
        x[0],
        x[1],
        x[2],
        x[0] + x[1],
        x[0] + x[2],
        x[1] + x[2],
        x[0] - x[1],
        x[0] - x[2],
```

```
        x[1] - x[2],
        x[0] + x[1] + x[2],
        x[0] + x[1] - x[2],
        x[0] - x[1] + x[2],
        x[0] - x[1] - x[2])
        return abs(v).max()
    X = miso.variables(2)
    Y = miso.variables(2)
    T = miso.variables(1)
    elem_a = miso.poly_space((X, 3))
    elem_b = miso.poly_space((Y, 3))
    pb = miso.bases.LAGRANGE
    xa0 = elem_a.geo_map(pb, 3)
    xa1 = elem_a.geo_map(pb, 3)
    xb0 = elem_b.geo_map(pb, 3)
    xb1 = elem_b.geo_map(pb, 3)
    xa = ((xa0 * (1-T)) + (xa1 * T)).collapse()
    xb = ((xb0 * (1-T)) + (xb1 * T)).collapse()
    dist = PBB(xb-xa)
    sd = [miso.subdiv_strategy(), miso.subdiv_strategy(T)]
    miso.generate('./src/generated', 'CubicTriCCD', dist, objective=T, strategies=sd)
```

Bibliography

[1] IEEE standard for interval arithmetic, 2015. doi:10.1109/IEEESTD.2015.7107164.

[2] IEEE standard for floating-point arithmetic, 2019. doi:10.1109/IEEESTD.2019.8766229.

[3] M. Aanjaneya, J.P. Lim, and S. Nagarakatte. Progressive polynomial approximations for fast correctly rounded math libraries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 552–565, New York, NY, USA, 2022. ACM.

[4] M. Aanjaneya, J.P. Lim, and S. Nagarakatte. The rlibm project, 2024. URL `https://github.com/rutgers-apl/The-RLIBM-Project`.

[5] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire. *Real-Time Rendering 4th Edition.* A.K. Peters/CRC Press, Boca Raton, FL, USA, 2018.

[6] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire. Real-time rendering - ray tracing resources page, 2024. URL `https://www.realtimerendering.com/intersections.html`.

[7] Christie Alappat, Achim Basermann, Alan R. Bishop, Holger Fehske, Georg Hager, Olaf Schenk, Jonas Thies, and Gerhard Wellein. A recursive algebraic coloring technique for hardware-efficient symmetric sparse matrix-vector multiplication. *ACM Transactions on Parallel Computing*, 7(3), June 2020. ISSN 2329-4949.

[8] H. Alt and L. Scharf. Computing the hausdorff distance between curved objects. *International Journal of Computational Geometry & Applications*, 18(04):307–320, 2008.

[9] R. W. Anderson, V. A. Dobrev, Tz. V. Kolev, and R. N. Rieben. Monotonicity in high-order curvilinear finite element arbitrary lagrangian–eulerian remap. *International Journal for Numerical Methods in Fluids*, 77(5):249–273, October 2014. ISSN 1097-0363. doi: 10.1002/fld.3965.

[10] M. Attene. Indirect predicates library. `https://github.com/MarcoAttene/Indirect_Predicates`, 2019.

[11] M. Attene. Indirect predicates for geometric constructions. *Computer-Aided Design*, 126:102856, 2020.

[12] M. Attene. NFG - numbers for geometry, 2025. URL https://github.com/MarcoAttene/NFG.

[13] J.A. Baerentzen and H. Aanaes. Signed distance computation using the angle weighted pseudonormal. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):243–253, 2005.

[14] A. W. Bargteil and E. Cohen. Animation of deformable bodies with quadratic bézier finite elements. *ACM Trans. Graph.*, 33(3):1–10, May 2014.

[15] M. Bartoň, I. Hanniel, G. Elber, and M.-S. Kim. Precise hausdorff distance computation between polygonal meshes. *Computer Aided Geometric Design*, 27(8):580–591, 2010. URL https://www.sciencedirect.com/science/article/pii/S016783961000049X. Advances in Applied Geometry.

[16] G. Louis Bernstein and F. Kjolstad. Why new programming languages for simulation? *ACM Trans. Graph.*, 35(2):20e:1–20e:3, May 2016.

[17] Matthias Bollhöfer, Aryan Eftekhari, Simon Scheidegger, and Olaf Schenk. Large-scale sparse inverse covariance matrix estimation. *SIAM Journal on Scientific Computing*, 41(1):380–401, 2019.

[18] Matthias Bollhöfer, Olaf Schenk, Radim Janalik, Steve Hamm, and Kiran Gullapalli. State-of-the-art sparse direct solvers. *Parallel Algorithms in Computational Science and Engineering*, pages 3–33, 2020.

[19] N. Brisebarre and J.-M. Muller. Correctly rounded multiplication by arbitrary precision constants. In *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*, pages 13–20, 2005.

[20] N. Brisebarre, G. Hanrot, J.-M. Muller, and P. Zimmermann. Correctly-rounded evaluation of a function: Why, how, and at what cost? *ACM Comput. Surv.*, 58(1), 2025.

[21] T. Brochu, E. Edwards, and R. Bridson. Efficient geometrically exact continuous collision detection. *ACM Trans. Graph.*, 31(4), July 2012.

[22] H Brönnimann, S Melquiond, and S Pion. The design of the boost interval arithmetic library. *Theoretical Computer Science*, 351:111–118, 2006. doi: 10.1016/j.tcs.2005.09.062.

[23] J.A. Carretero and M.A. Nahon. Solving minimum distance problems with convex or concave bodies using combinatorial global optimization algorithms. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 35(6):1144–1155, 2005.

[24] X. Chen, C. Yu, X. Ni, M. Chu, B. Wang, and B. Chen. A time-dependent inclusion-based method for continuous collision detection between parametric surfaces. *ACM SIGGRAPH Computer Graphics*, 43(6), 2024.

[25] Yunuo Chen, Minchen Li, Lei Lan, Hao Su, Yin Yang, and Chenfanfu Jiang. A unified newton barrier method for multibody dynamics. *ACM Trans. Graph.*, 41(4), July 2022. ISSN 0730-0301. doi: 10.1145/3528223.3530076. URL https://doi.org/10.1145/3528223.3530076.

[26] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, 1998.

[27] G.E. Collins and A.G. Akritas. Polynomial real root isolation using descarte's rule of signs. In *Proceedings of the third ACM symposium on Symbolic and algebraic computation - SYMSAC '76*, SYMSAC '76, page 272–275. ACM Press, 1976.

[28] J. A. Cottrell, T. J. R. Hughes, and Y. Bazilevs. *Isogeometric Analysis: Toward Integration of CAD and FEA*. Wiley, 2009.

[29] CVX Research, Inc. CVX: Matlab software for disciplined convex programming, version 2.0. `https://cvxr.com/cvx`, August 2012.

[30] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, jan 1998. ISSN 1070-9924. doi: 10.1109/99.660313. URL `https://doi.org/10.1109/99.660313`.

[31] C. Daramy-Loirat, D. Defour, F. de Dinechin, and J.-M. Muller. CR-LIBM: A correctly rounded elementary function library. In *Proceedings SPIE 48th Annual Meeting - Optical Science and Technology*, San Diego (CA) – USA, 2003.

[32] E. De Klerk, M. Laurent, Z. Sun, and J.C. Vera. On the convergence rate of grid search for polynomial optimization over the simplex. *Optimization Letters*, 11(3):597–608, March 2017.

[33] Etienne de Klerk, Monique Laurent, and Pablo A. Parrilo. A PTAS for the minimization of polynomials of fixed degree over the simplex. *Theoretical Computer Science*, 361(2-3):210–225, September 2006. ISSN 03043975. doi: 10.1016/j.tcs.2006.05.011. URL `https://linkinghub.elsevier.com/retrieve/pii/S0304397506003252`.

[34] Etienne de Klerk, Monique Laurent, and Zhao Sun. An error analysis for polynomial optimization over the simplex based on the multivariate hypergeometric distribution, July 2014. URL `http://arxiv.org/abs/1407.2108`. arXiv:1407.2108 [math].

[35] S. Dey, R.M. O'Bara, and M.S. Shephard. Curvilinear mesh generation in 3D. In *Proceedings of the 8th International Meshing Roundtable*, pages 407–417, 1999.

[36] S. Dey, R.M. O'Bara, and M.S. Shephard. Towards curvilinear meshing in 3d: the case of quadratic simplices. *Computer-Aided Design*, 33(3):199–209, 2001.

[37] V. Dobrev, P. Knupp, T. Kolev, K. Mittal, and V. Tomov. The target-matrix optimization paradigm for high-order meshes. *SIAM Jou. Sci. Comp.*, 41(1):B50–B68, 2019.

[38] Wei Du, Shibo Liu, Jia-Peng Guo, Ligang Liu, and Xiao-Ming Fu. Robust and efficient preservation of high-order continuous geometric validity. *IEEE Transactions on Visualization and Computer Graphics*, 31(12):10533–10544, 2025. doi: 10.1109/TVCG.2025.3603025.

[39] Yu Fang, Minchen Li, Chenfanfu Jiang, and Danny M. Kaufman. Guaranteed globally injective 3d deformation processing. *ACM Trans. Graph.*, 40(4), July 2021. ISSN 0730-0301. doi: 10.1145/3450626.3459757. URL https://doi.org/10.1145/3450626.3459757.

[40] Yu Fang, Minchen Li, Yadi Cao, Xuan Li, Joshuah Wolper, Yin Yang, and Chenfanfu Jiang. Augmented incremental potential contact for sticky interactions. *IEEE Transactions on Visualization and Computer Graphics*, 30(8):5596–5608, 2024. doi: 10.1109/TVCG.2023.3295656.

[41] G. Farin. *Curves and Surfaces for CAGD: A Practical Guide*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2001.

[42] Z. Ferguson, M. Li, T. Schneider, F. Gil-Ureta, T. Langlois, C. Jiang, D. Zorin, D.M. Kaufman, and D. Panozzo. Intersection-free rigid body dynamics. *ACM Transactions on Graphics (SIGGRAPH)*, 40(4), 2021.

[43] Z. Ferguson, P. Jain, D. Zorin, T. Schneider, and D. Panozzo. High-order incremental potential contact for elastodynamic simulation on curved meshes. In *ACM SIGGRAPH 2023 Conference Proceedings*, SIGGRAPH '23, New York, NY, USA, 2023. Association for Computing Machinery.

[44] Zachary Ferguson et al. IPC Toolkit. https://ipc-sim.github.io/ipc-toolkit/, 2020. URL https://ipc-sim.github.io/ipc-toolkit/.

[45] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2), 2007.

[46] A. Gargallo-Peiró, X. Roca, J. Peraire, and J. Sarrate. Distortion and quality measures for validating and generating high-order tetrahedral meshes. *Engineering with Computers*, 31(3):423–437, 2015.

[47] J. Garloff, C. Jansson, and A.P. Smith. Lower bound functions for polynomials. *Journal of Computational and Applied Mathematics*, 157(1):207–225, 2003.

[48] P.L. George and H. Borouchaki. Validity of lagrange (bézier) and rational bézier quads of degree 2. 99(8):611–632, 2014. ISSN 0029-5981, 1097-0207. doi: 10.1002/nme.4696. URL https://onlinelibrary.wiley.com/doi/10.1002/nme.4696.

[49] C. Geuzaine, A. Johnen, J. Lambrechts, J. F. Remacle, and T. Toulorge. *The Generation of Valid Curvilinear Meshes*, pages 15–39. Springer International Publishing, Cham, 2015.

[50] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[51] F. Goualard. Gaol: Not just another interval library, 2016. URL https://github.com/goualard-f/GAOL.

[52] T. Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 edition, 2012. http://gmplib.org/.

[53] M. Grant and S. Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences, pages 95–110. Springer-Verlag Limited, 2008.

[54] A. Guezlec. "meshsweeper": dynamic point-to-polygonal mesh distance and applications. *IEEE Transactions on Visualization and Computer Graphics*, 7(1):47–61, 2001.

[55] S. Hadap, D. Eberle, P. Volino, M.C. Lin, S. Redon, and C. Ericson. Collision detection and proximity queries. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, page 15–es, New York, NY, USA, 2004. ACM.

[56] E.R. Hansen and R.I. Greenberg. An interval Newton method. *Applied Mathematics and Computation*, 12(2-3):89–98, May 1983.

[57] P. Herholz, X. Tang, T. Schneider, S. Kamil, D. Panozzo, and O. Sorkine-Hornung. Sparsity-specific code optimization using expression trees. *ACM Transactions on Graphics*, 41(5):175:1–19, 2022.

[58] Victoria Hernandez-Mederos, Jorge Estrada-Sarlabous, and Dionne León. On local injectivity of 2d triangular cubic bezier functions. *Revista Investigación Operacional*, 27:261–275, 01 2006.

[59] Timothy Hickey, Qun Ju, and Maarten Van Emden. Interval arithmetic: from principles to implementation. *Journal of the ACM*, 48:1038–1068, 2001.

[60] K. Hormann, L. Kania, and C. Yap. Novel range functions via taylor expansions and recursive lagrange interpolation with application to real root isolation. In *Proceedings of the 2021 International Symposium on Symbolic and Algebraic Computation*, ISSAC '21, page 193–200, New York, NY, USA, 2021. ACM.

[61] K. Hormann, C. Yap, and Y.S. Zhang. Range functions of any convergence order and their amortized complexity analysis. In *Computer Algebra in Scientific Computing: 25th International Workshop, CASC 2023*, page 162–182, Berlin, Heidelberg, 2023. Springer-Verlag.

[62] Y. Hu, T.-M. Li, L. Anderson, J. Ragan-Kelley, and F. Durand. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics*, 38 (6):1–16, November 2019.

[63] Yixin Hu, Teseo Schneider, Bolun Wang, Denis Zorin, and Daniele Panozzo. Fast tetrahedral meshing in the wild. *ACM Trans. Graph.*, 39(4), 8 2020. ISSN 0730-0301. doi: 10.1145/3386569.3392385. URL https://doi.org/10.1145/3386569.3392385.

[64] Kemeng Huang, Floyd M. Chitalu, Huancheng Lin, and Taku Komura. Gipc: Fast and stable gauss-newton optimization of ipc barrier energy. *ACM Trans. Graph.*, 43(2), March 2024. ISSN 0730-0301. doi: 10.1145/3643028. URL https://doi.org/10.1145/3643028.

[65] Zizhou Huang, Davi Colli Tozoni, Arvi Gjoka, Zachary Ferguson, Teseo Schneider, Daniele Panozzo, and Denis Zorin. Differentiable solver for time-dependent deformation problems with contact. *ACM Trans. Graph.*, 43(3), May 2024. ISSN 0730-0301. doi: 10.1145/3657648. URL https://doi.org/10.1145/3657648.

[66] L. Jaulin, I. Braems, and E. Walter. Interval methods for nonlinear identification and robust control. In *Proceedings of the 41st IEEE Conference on Decision and Control, 2002.*, volume 4, pages 4676–4681, Las Vegas, NV, USA, 2002. IEEE.

[67] Z. Jiang, Z. Zhang, Y. Hu, T. Schneider, D. Zorin, and D. Panozzo. Bijective and coarse high-order tetrahedral meshes. *ACM Trans. Graph.*, 40(4):1–16, 2021.

[68] A. Johnen, J.-F. Remacle, and C. Geuzaine. Geometrical validity of curvilinear finite elements. *Journal of Computational Physics*, 233:359–372, 2013. ISSN 0021-9991.

[69] A. Johnen, J. F. Remacle, and C. Geuzaine. Geometrical validity of high-order triangular finite elements. *Eng. with Comput.*, 30(3):375–382, 2014.

[70] A. Johnen, J.-C. Weill, and J.-F. Remacle. Robust and efficient validation of the linear hexahedral element. *Procedia Engineering*, 203:271–283, 2017.

[71] A. Johnen, C. Geuzaine, T. Toulorge, and J.-F. Remacle. Efficient computation of the minimum of shape quality measures on curvilinear finite elements. *Computer-Aided Design*, 103:24–33, 2018.

[72] M.W. Jones, J.A. Baerentzen, and M. Sramek. 3d distance fields: a survey of techniques and applications. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):581–599, 2006.

[73] C. Kane, J. E. Marsden, M. Ortiz, and M. West. Variational integrators and the newmark algorithm for conservative and dissipative mechanical systems. *International Journal for Numerical Methods in Engineering*, 49(10):1295–1325, 2000.

[74] Y. Kang, S.-H. Yoon, M.-H. Kyung, and M.-S. Kim. Fast and robust computation of the hausdorff distance between triangle mesh and quad mesh for near-zero cases. *Computers & Graphics*, 81: 61–72, 2019.

[75] Y.-J. Kim, Y.-T. Oh, S.-H. Yoon, M.-S. Kim, and G. Elber. Efficient hausdorff distance computation for freeform geometric models in close proximity. *Computer-Aided Design*, 45(2):270–276, 2013. Solid and Physical Modeling 2012.

[76] F. Kjolstad, S. Kamil, J. Ragan-Kelley, D.I.W. Levin, S. Sueda, D. Chen, E. Vouga, D.M. Kaufman, G. Kanwar, W. Matusik, and S. Amarasinghe. Simit: A language for physical simulation. *ACM Trans. Graph.*, 35(2):20:1–20:21, May 2016. ISSN 0730-0301.

[77] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[78] O. Knüppel. PROFIL/BIAS — a fast interval library. *Computing*, 53(3):277–287, 1994.

[79] B. Lambov. Interval arithmetic using sse-2. In P. Hertling, C. M. Hoffmann, W. Luther, and N. Revol, editors, *Reliable Implementation of Real Number Algorithms: Theory and Practice*, pages 102–113, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[80] Lei Lan, Yin Yang, Danny Kaufman, Junfeng Yao, Minchen Li, and Chenfanfu Jiang. Medial ipc: accelerated incremental potential contact with medial elastics. *ACM Trans. Graph.*, 40(4), July 2021. ISSN 0730-0301. doi: 10.1145/3450626.3459753. URL https://doi.org/10.1145/3450626.3459753.

[81] Lei Lan, Danny M. Kaufman, Minchen Li, Chenfanfu Jiang, and Yin Yang. Affine body dynamics: fast, stable and intersection-free simulation of stiff materials. *ACM Trans. Graph.*, 41(4), July 2022. ISSN 0730-0301. doi: 10.1145/3528223.3530064. URL https://doi.org/10.1145/3528223.3530064.

[82] Lei Lan, Guanqun Ma, Yin Yang, Changxi Zheng, Minchen Li, and Chenfanfu Jiang. Penetration-free projective dynamics on the gpu. *ACM Trans. Graph.*, 41(4), July 2022. ISSN 0730-0301. doi: 10.1145/3528223.3530069. URL https://doi.org/10.1145/3528223.3530069.

[83] Lei Lan, Minchen Li, Chenfanfu Jiang, Huamin Wang, and Yin Yang. Second-order stencil descent for interior-point hyperelasticity. *ACM Trans. Graph.*, 42(4), July 2023. ISSN 0730-0301. doi: 10.1145/3592104. URL https://doi.org/10.1145/3592104.

[84] S. Lengagne, R. Kalawoun, F. Bouchon, and Y. Mezouar. Reducing pessimism in interval analysis using b-splines properties: Application to robotics. *Reliable Computing*, 27:63–87, 2020.

[85] M. Lerch, G. Tischler, J.W.V. Gudenberg, W. Hofschuster, and W. Kramer. Filib++, a fast interval library supporting containment computations. *ACM Trans. Math. Softw*, 32(2):299–324, 2006.

[86] M. Lerch, G. Tischler, J.W.V. Gudenberg, W. Hofschuster, and W. Kramer. Fi_lib and filib++, 2011. URL https://www2.math.uni-wuppertal.de/wrswt/software/filib.html.

[87] B. Lévy. Robustness and efficiency of geometric programs: The predicate construction kit (pck). *Computer-Aided Design*, 72:3–12, 2016.

[88] M. Li, Z. Ferguson, T. Schneider, T. Langlois, D. Zorin, D. Panozzo, C. Jiang, and D. M. Kaufman. Incremental potential contact: Intersection- and inversion-free large deformation dynamics. *ACM Trans. Graph. (SIGGRAPH)*, 39(4), 2020.

[89] M. Li, D.M. Kaufman, and C. Jiang. Codimensional incremental potential contact. *ACM Trans. Graph.*, 40(4), July 2021.

[90] Minchen Li, Zachary Ferguson, Teseo Schneider, Timothy Langlois, Denis Zorin, Daniele Panozzo, Chenfanfu Jiang, and Danny M. Kaufman. Convergent incremental potential contact, 2023.

[91] Xuan Li, Yu Fang, Minchen Li, and Chenfanfu Jiang. Bfemp: Interpenetration-free mpm–fem coupling with barrier contact. *Computer Methods in Applied Mechanics and Engineering*, 390, 2 2022. doi: 10.1016/j.cma.2021.114350.

[92] Xuan Li, Yu Fang, Lei Lan, Huamin Wang, Yin Yang, Minchen Li, and Chenfanfu Jiang. Subspace-preconditioned gpu projective dynamics with contact for cloth simulation. In *SIGGRAPH Asia 2023 Conference Papers*, SA '23, New York, NY, USA, 2023. Association for Computing Machinery.

ISBN 9798400703157. doi: 10.1145/3610548.3618157. URL https://doi.org/10.1145/3610548.3618157.

[93] Xuan Li, Minchen Li, Xuchen Han, Huamin Wang, Yin Yang, and Chenfanfu Jiang. A dynamic duo of finite elements and material points. In *ACM SIGGRAPH 2024 Conference Papers*, SIGGRAPH '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705250. doi: 10.1145/3641519.3657449. URL https://doi.org/10.1145/3641519.3657449.

[94] Y. Li, S. Kamil, A. Jacobson, and Y. Gingold. I♡la: Compilable markdown for linear algebra. *ACM Transactions on Graphics (TOG)*, 40(6), 2021.

[95] Y. Li, S. Kamil, K. Crane, A. Jacobson, and Y. Gingold. I♡mesh: A dsl for mesh processing. *ACM Trans. Graph.*, 43(6), 2024.

[96] J.P. Lim and S. Nagarakatte. One polynomial approximation to produce correctly rounded results of an elementary function for multiple representations and rounding modes. *Proc. ACM Program. Lang.*, 6(POPL), January 2022.

[97] X.-J. Luo, M.S. Shephard, J.-F. Remacle, R.M. O'Bara, M.W. Beall, B.A. Szabó, and R. Actis. p-version mesh generation issues. In *Proceedings of the 11th International Meshing Roundtable*, pages 343–354, 2002.

[98] S. A. Maas, B. J. Ellis, G. A. Ateshian, and J. A. Weiss. Febio: Finite elements for biomechanics. *Jou. of Biomechanical Engineering*, 134(1), 2012.

[99] M. Mandad and M. Campen. Efficient piecewise higher-order parametrization of discrete surfaces with local and global injectivity. *Computer-Aided Design*, 127:102862, 2020.

[100] Z. Marschner, D. Palmer, P. Zhang, and J. Solomon. Hexahedral mesh repair via sum-of-squares relaxation. *Computer Graphics Forum*, 39(5):133–147, August 2020.

[101] Z. Marschner, D. Palmer, P. Zhang, and J. Solomon. Hexahedral Mesh Repair via Sum-of-Squares Relaxation. *Computer Graphics Forum*, 39(5):133–147, August 2020.

[102] Z. Marschner, P. Zhang, D. Palmer, and J. Solomon. Sum-of-squares geometry processing. *ACM Trans. Graph.*, 40(6), December 2021.

[103] L. Martin, P. Jain, Z. Ferguson, T. Gholamalizadeh, F. Moshfeghifar, K. Erleben, D. Panozzo, S. Abramowitch, and T. Schneider. A systematic comparison between febio and polyfem for biomechanical systems. *Computer Methods and Programs in Biomedicine*, 244:107938, 2024.

[104] J.P. Merlet. Interval analysis and robotics. volume 66, pages 147–156, 11 2007. ISBN 978-3-642-14742-5. doi: 10.1007/978-3-642-14743-2_13.

[105] A. Meurer, C.P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J.K. Moore, S. Singh, T. Rathnayake, S. Vig, B.E. Granger, R.P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M.J. Curry, A.R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal,

R. Cimrman, and A. Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3: e103, 2017.

[106] A. Meyer and S. Pion. Fpg: A code generator for fast and certified geometric predicates. In *Real numbers and computers*, pages 47–60, 2008.

[107] J. Mezger, B. Thomaszewski, S. Pabst, and W. Straßer. Interactive physically-based shape editing. *Computer Aided Geometric Design*, 26(6):680–694, August 2009.

[108] M.B. Monagan, K.O. Geddes, K. M. Heal, G. Labahn, S.M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 10 Programming Guide*. Maplesoft, Waterloo ON, Canada, 2005.

[109] Ramon E. Moore. *Interval Analysis*. Prentice-Hall Series in Automatic Computation. Prentice Hall, Englewood Cliffs, NJ, 1966.

[110] Steven S Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, Oxford, England, September 1997.

[111] J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer, New York, 2006.

[112] R.W. Ogden. *Non-Linear Elastic Deformations*. Dover Civil and Mechanical Engineering. Dover Publications, 2013. ISBN 9780486318714. URL https://books.google.com/books?id=52XDAgAAQBAJ.

[113] Julian Panetta, Qingnan Zhou, Luigi Malomo, Nico Pietroni, Paolo Cignoni, and Denis Zorin. Elastic textures for additive fabrication. *ACM Transactions on Graphics*, 34(4):1–12, July 2015. ISSN 1557-7368. doi: 10.1145/2766937. URL http://dx.doi.org/10.1145/2766937.

[114] Nico Pietroni, Marcel Campen, Alla Sheffer, Gianmarco Cherchi, David Bommes, Xifeng Gao, Riccardo Scateni, Franck Ledoux, Jean Remacle, and Marco Livesu. Hex-mesh generation and processing: A survey. *ACM Trans. Graph.*, 42(2), October 2022. ISSN 0730-0301. doi: 10.1145/3554920. URL https://doi.org/10.1145/3554920.

[115] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 5.3 edition, 2021. URL https://doc.cgal.org/5.3/Manual/packages.html.

[116] T. Schneider, J. Dumas, X. Gao, D. Zorin, and D. Panozzo. PolyFEM. https://polyfem.github.io/, 2019.

[117] T. Schneider, Y. Hu, X. Gao, J. Dumas, D. Zorin, and D. Panozzo. A large-scale comparison of tetrahedral and hexahedral elements for solving elliptic pdes with the finite element method. *ACM Trans. on Graph.*, 41(3):1–14, 2022.

[118] Dan Shapero. predicates: Robust geometric predicates in c. https://github.com/danshapero/predicates, 2015. A modernization of Shewchuk's adaptive-precision floating-point predicates (public domain).

[119] Xing Shen, Runyuan Cai, Mengxiao Bi, and Tangjie Lv. Preconditioned nonlinear conjugate gradient method for real-time interior-point hyperelasticity. In *ACM SIGGRAPH 2024 Conference Papers*, SIGGRAPH '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705250. doi: 10.1145/3641519.3657490. URL https://doi.org/10.1145/3641519.3657490.

[120] J. R. Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3):305–363, October 1997.

[121] Naoki Shibata and Francesco Petrogalli. Sleef: A portable vectorized library of c standard mathematical functions. *IEEE Transactions on Parallel and Distributed Systems*, 31(6):1316–1327, 2020. doi: 10.1109/TPDS.2019.2960333.

[122] Hang Si. Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Softw.*, 41(2), 2 2015. ISSN 0098-3500. doi: 10.1145/2629697. URL https://doi.org/10.1145/2629697.

[123] Alexei Sibidanov, Paul Zimmermann, and Stéphane Glondu. The CORE-MATH project. In *29th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 26–34, 2022.

[124] J. Smith and S. Schaefer. Bijective parameterization with free boundaries. *ACM Trans. Graph.*, 34 (4), 2015.

[125] J Snyder. Interval analysis for computer graphics. *ACM SIGGRAPH*, pages 121–130, 1992.

[126] J. Snyder, A.R. Woodbury, K. Fleischer, B. Currin, and A.H. Barr. Interval methods for multi-point collisions between time-dependent curved surfaces. In *ACM SIGGRAPH*. ACM, August 1993.

[127] J.M. Snyder. *Generative Modeling: An Approach to High Level Shape Design for Computer Graphics and CAD*. PhD thesis, 1991.

[128] S.-H. Son, M.-S. Kim, and G. Elber. Precise hausdorff distance computation for freeform surfaces based on computations with osculating toroidal patches. *Computer Aided Geometric Design*, 86: 101967, 2021.

[129] V. Stahl. *Interval Methods for Bounding the Range of Polynomials and Solving Systems of Nonlinear Equations*. PhD thesis, September 1995.

[130] S. Suwelack, D. Lukarski, V. Heuveline, R. Dillmann, and S. Speidel. Accurate surface embedding for higher order finite elements. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '13. ACM, 2013.

[131] M. Tang, R. Tong, Z. Wang, and D. Manocha. Fast and exact continuous collision detection with Bernstein sign classification. *ACM Transactions on Graphics*, 33(6):1–8, 2014. ISSN 0730-0301, 1557-7368. doi: 10.1145/2661229.2661237. URL https://dl.acm.org/doi/10.1145/2661229.2661237.

[132] X. Tang, T. Schneider, S. Kamil, A. Panda, J. Li, and D. Panozzo. Eggs: Sparsity-specific code generation. *Computer Graphics Forum*, 39(5):209–219, August 2020.

[133] Xuan Tang, Zachary Ferguson, Teseo Schneider, Denis Zorin, Shoaib Kamil, and Daniele Panozzo. A cross-platform benchmark for interval computation libraries. In *Parallel Processing and Applied Mathematics: 14th International Conference, Revised Selected Papers, Part II*, page 415–427, Berlin, Heidelberg, 2022. Springer-Verlag. ISBN 978-3-031-30444-6.

[134] Thomas Toulorge, Christophe Geuzaine, Jean-François Remacle, and Jonathan Lambrechts. Robust untangling of curvilinear meshes. *Journal of Computational Physics*, 254:8–26, December 2013. ISSN 0021-9991. doi: 10.1016/j.jcp.2013.07.022. URL http://dx.doi.org/10.1016/j.jcp.2013.07.022.

[135] O. V. Ushakova. Nondegeneracy tests for hexahedral cells. *Computer Methods in Applied Mechanics and Engineering*, 200(17–20):1649–1658, 2011.

[136] S. Vavasis. A bernstein-bezier sufficient condition for invertibility of polynomial mapping functions. 2003.

[137] B. Wang, Z. Ferguson, T. Schneider, X. Jiang, M. Attene, and D. Panozzo. A large-scale benchmark and an inclusion-based algorithm for continuous collision detection. *ACM Trans. Graph.*, 40(5): 1–16, 2021.

[138] B. Wang, . Ferguson, X. Jiang, M. Attene, D. Panozzo, and T. Schneider. Fast and exact root parity for continuous collision detection. *Computer Graphics Forum (Proceedings of Eurographics)*, 41(2), 2022.

[139] Bolun Wang, Zachary Ferguson, Teseo Schneider, Xin Jiang, Marco Attene, and Daniele Panozzo. A large-scale benchmark and an inclusion-based algorithm for continuous collision detection. *ACM Trans. Graph.*, 40(5), sep 2021. ISSN 0730-0301. doi: 10.1145/3460775. URL https://doi.org/10.1145/3460775.

[140] Huamin Wang. Defending continuous collision detection against errors. *ACM Transactions on Graphics*, 33(4):1–10, July 2014.

[141] Wolfram Research, Inc. Mathematica, Version 13.3, 2023. URL https://www.wolfram.com/mathematica. Champaign, IL.

[142] P. Zhang, Z. Marschner, J. Solomon, and R. Tamstorf. Sum-of-squares collision detection for curved shapes and paths. In *ACM SIGGRAPH 2023 Conference Proceedings*, New York, NY, USA, 2023. ACM.

[143] Z. Zhang, Y.-J. Chiang, and C. Yap. Theory and explicit design of a path planner for an se(3) robot, 2024. URL https://arxiv.org/abs/2407.05135.

[144] Y. Zheng, H. Sun, X. Liu, H. Bao, and J. Huang. Economic upper bound estimation in hausdorff distance computation for triangle meshes. *Computer Graphics Forum*, 41(1):46–56, 2022.