MiSo: A DSL for Robust and Efficient Solve and Minimize Problems

FEDERICO SICHETTI, Università di Genova, Italy
ENRICO PUPPO, Università di Genova, Italy
ZIZHOU HUANG, New York University, United States
MARCO ATTENE, CNR IMATI, Italy
DENIS ZORIN, New York University, United States
DANIELE PANOZZO, New York University, United States

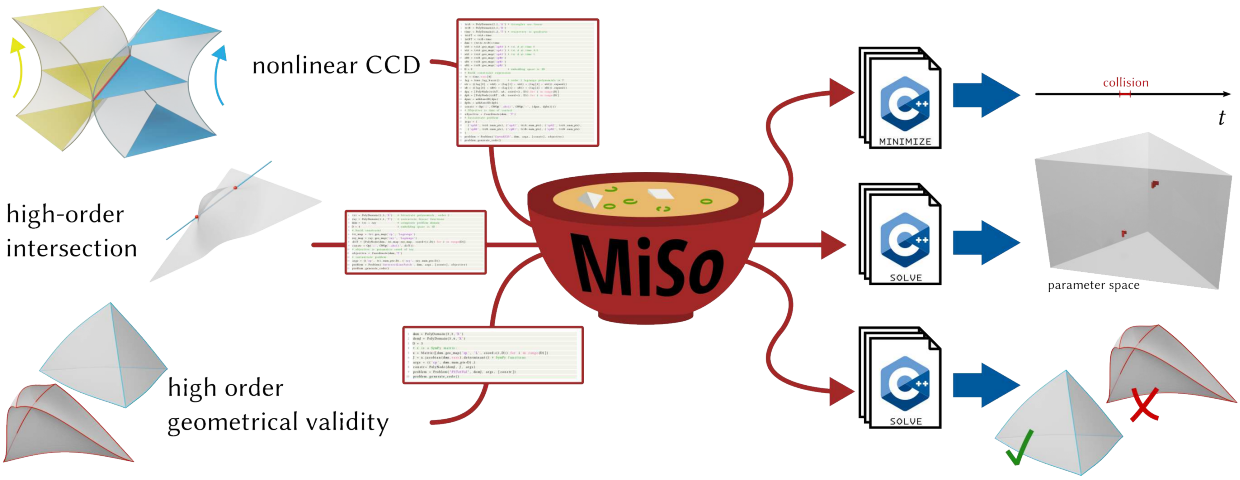

Fig. 1. A MiSo specification is a Python script that translates a Solve or Minimize problem described in terms of its constraints and objective function. The MiSo compiler generates optimized C++ code for resolving the specific problem, relying on generic solvers based on interval analysis that provide conservative results. We apply MiSo to several geometric problems, obtaining competitive performance compared to hand-optimized code.

Many problems in computer graphics can be formulated as finding the global minimum of a function subject to a set of non-linear constraints (Minimize), or finding all solutions of a system of non-linear constraints (Solve). We introduce MiSo, a domain-specific language and compiler for generating efficient C++ code for low-dimensional Minimize and Solve problems, that uses interval methods to guarantee conservative results while using floating point arithmetic. We demonstrate that MiSo-generated code shows competitive performance compared to hand-optimized codes for several computer graphics problems, including high-order collision detection with non-linear trajectories, surface-surface intersection, and geometrical validity checks for finite element simulation.

Authors' Contact Information: Federico Sichetti, Università di Genova, Italy, federico.sichetti@edu.unige.it; Enrico Puppo, Università di Genova, Italy, enrico.puppo@unige.it; Zizhou Huang, New York University, United States, zizhou@nyu.edu; Marco Attene, CNR IMATI, Italy, jaiko@ge.imati.cnr.it; Denis Zorin, New York University, United States, dzorin@cs.nyu.edu; Daniele Panozzo, New York University, United States, panozzo@nyu.edu.

ACM 1557-7368/2025/8-ART120
https://doi.org/10.1145/3731207

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; **Source code generation**; • **Theory of computation** → *Nonconvex optimization*; • **Mathematics of computing** → Interval arithmetic; *Nonlinear equations*; • **Computing methodologies** → **Mesh geometry models**; Physical simulation.

ACM Reference Format:
Federico Sichetti, Enrico Puppo, Zizhou Huang, Marco Attene, Denis Zorin, and Daniele Panozzo. 2025. MiSo: A DSL for Robust and Efficient Solve and Minimize Problems. *ACM Trans. Graph.* 44, 4, Article 120 (August 2025), 18 pages. https://doi.org/10.1145/3731207

1 Introduction

Non-linear constraint solving is fundamental to graphics and scientific computing, with applications ranging from collision detection and minimal distance computation to element inversion and Boolean operations. A vast literature addresses this topic (Section 2). For example, [Akenine-Möller et al. 2018; Akenine-Möller et al. 2024] 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 [Snyder 1992]. Testing the correctness and ensuring the efficient, accurate implementation of these algorithms is a major challenge [Wang et al. 2021]. 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 [1992] 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 [Chen et al. 2024; Wang et al. 2021] 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 4.2) that bound function ranges over domains, by composition of interval operators (Section C). 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 [Johnen et al. 2013; Lengagne et al. 2020; Stahl 1995]. 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). 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 6).

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 computer graphics problems, including linear and high-order continuous collision detection, and finite element validity checks.

MiSo is available as an open-source project at https://gitlab.com/fsichetti/miso.

2 Related Work

We survey general methods for solving non-linear constraint systems, deferring application-specific comparisons to Section 6. We conclude with a survey of domain-specific languages for other graphics applications.

2.1 Explicit Root Finding

Algebraic. Computer Algebra Systems [Meurer et al. 2017; Monagan et al. 2005; Wolfram Research, Inc. 2023] 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. [2021] 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 [Smith and Schaefer 2015] and for collision detection [Hadap et al. 2004]. Evaluating the closed-form expression with floating point arithmetic, while efficient, is unstable and might lead to incorrect roots [Wang et al. 2021].

Numerical. A popular option is to use numerical methods for constrained optimization (Minimize) and root finding (Solve). [Nocedal and Wright 2006]. These methods are very popular in graphics [Hadap et al. 2004] 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 many use "small" numerical tolerances to mitigate this issue [Akenine-Möller et al. 2018; Chen et al. 2024; Li et al. 2021b], this heuristic does not guarantee correctness. Although forward error analysis can derive conservative tolerances [Wang 2014], these often lead to excessive false positives [Wang et al. 2021].

Robust interval versions of Newton's method [Hansen and Greenberg 1983; Stahl 1995] can bound zero loci for continuously differentiable functions, but they cannot handle non-differentiable operators such as abs, max, and min.

Counting. Brochu et al. [2012] reduce root parity counting to a geometric intersection problem solvable with custom numerical predicates [Wang et al. 2022]. 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. [2014] proposed a Bernstein sign classification method for CCD. However,

conservative implementation is challenging, and [Wang et al. 2021] presented a counterexample showing its failure to detect a collision.

2.2 Sum Of Squares Polynomials

Marschner et al. [2021] proposed reducing non-linear constraint solving to semidefinite programs, an approach also used for high-order patch collision detection [Zhang et al. 2023] and hexahedral mesh inversion repair [Marschner et al. 2020a]. However, this method is computationally costly and, like other numerical root-finders, provides only approximate solutions due to numerical precision. We compare against an SOS algorithm for collision detection in Section 6.

2.3 Inclusion-Based

A robust and generic solution has been pioneered for geometric modeling applications by John Snyder in his PhD thesis [Snyder 1991, 1992; Snyder et al. 1993]. 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. An inclusion function 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 [Hormann et al. 2021, 2023], in path planning [Zhang et al. 2024], and for more generally isolating polynomial roots [Collins and Akritas 1976].

Interval Arithmetic. Snyder [Snyder 1992] proposes using interval arithmetic 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 [Jaulin et al. 2002; Merlet 2007]. The high computational cost is likely why these approaches have not found wide application in graphics until recently: Wang et al. [2021] benchmarked this approach against a numerical root finder for collision detection and found it on average ~4 orders of magnitude slower.

Floating Point. Ad-hoc inclusion functions can be built without the use of interval arithmetic [Garloff et al. 2003; Stahl 1995]. Johnen et al. [2014] and Chen et al. [2024] 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. We use a similar approach to build our inclusion functions (Section 5.3).

Rational Grid Search. 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 [De Klerk et al. 2006, 2014, 2017]. These results could be used to construct inclusion functions for this special case, but we are not aware of any algorithm using them.

Predicates. Wang et al. [2021] propose a custom inclusion function for the continuous collision detection between triangles over linear trajectories. Similarly to the predicates proposed in [Shewchuk 1997], 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 [Attene 2020; Lévy 2016; Meyer and Pion 2008], this is still an error-prone and labor-intensive task.

2.4 DSL in Graphics

Multiple domain specific languages have been introduced in graphics, which can be broadly divided into two categories.

Convenience and Correctness. 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 [CVX Research, Inc. 2012; Grant and Boyd 2008], 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. [2021a] propose a method to automatically generate LaTeX and source code for linear algebra. Li et al. [2024] introduce a DSL for mesh processing, allowing a direct and intuitive description of geometry processing algorithms.

The two works closer to our contribution are [Meyer and Pion 2008] and [Lévy 2016], which analyze expression trees to generate the C++ code of corresponding filtered predicates.

Efficiency. Another family of DSL is introduced to improve performance of core algorithmic blocks. These include DSL for sparse linear algebra such as SIMIT [Kjolstad et al. 2016], TACO [Kjolstad et al. 2017], EGGS [Herholz et al. 2022; Tang et al. 2020] and for optimizing entire simulation algorithms [Bernstein and Kjolstad 2016], for example TAICHI [Hu et al. 2019].

Our DSL. MiSo is mainly targeting the first goal: automatically convert a high-level specification of Solve or Minimize into a reliable and conservative C++ code. However, our code generator unrolls many computations, favoring compiler optimizations. We observed that our generated code has comparable performance to hand-tuned implementations for specific instances commonly found in graphics (Section 6).

3 Overview

We begin by formally characterizing the class of addressable problems, followed by a broad description of MiSo's operation and a working example. Technical details are provided in Sections 4 and 5.

3.1 Problem statement

We transition from a continuous formulation of the problems to their discrete numerical counterparts, discussing the choices and limitations inherent in this process.

Definition 1 (Solve Problem). *A* Solve Problem *is defined by:*

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

Table 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. 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)
Geometric Validity	Solve	σ	ξ	-	$\lvert J_x(\xi)\rvert$
Continuous Collision Detection	Minimize	$\sigma_1 \times \sigma_2 \times [0,1]$	(ξ, η, t)	t	$d(\overline{x_1}(\xi,t), \overline{x_2}(\eta,t))$
Parametric Primitive Intersection	Solve	$\sigma_1 \times \sigma_2$	(ξ, η)	-	$d(x_1(\xi), x_2(\eta))$
Minimal Distance	Minimize	$\sigma_1 \times \sigma_2$	(ξ, η)	$d(x_1(\xi), x_2(\eta))$	-
Diameter of Primitive	Minimize	$\sigma \times \sigma$	(ξ_1, ξ_2)	$-d(x(\xi_1), x(\xi_2))$	-
Boolean Intersection	Solve	$[0,1]^n$	ξ	-	$\max(D_1(\xi), D_2(\xi))$
Boolean Union	Solve	$[0,1]^n$	ξ	-	$\min(D_1(\xi), D_2(\xi))$
Boolean Difference	Solve	$[0,1]^n$	ξ	-	$\max(D_1(\xi), -D_2(\xi))$

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 2 (Minimize Problem). *A* Minimize Problem *is defined by:*

a) *and* b) *as in Definition 1;*
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 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 4.1 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 3 (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.*

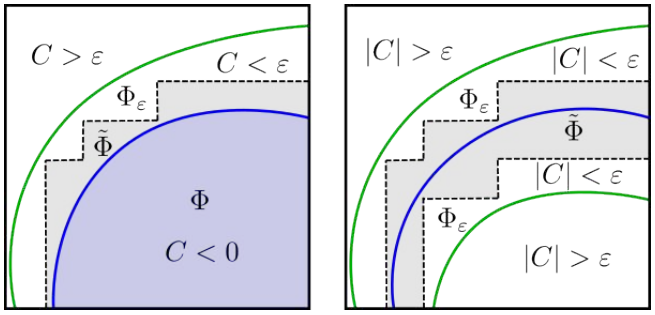

Fig. 2. Regions defining a solution for ε-Solve Problems on a domain $[0,1] \times [0,1]$ with an inequality constraint (Left) and an equality constraint (Right). Left: the blue-shaded region below the blue line is the solution Φ of the corresponding Solve Problem in the continuum; the region Φ_ε extends 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.

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

Figure 2 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.

Definition 4 (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 3 shows an example of an ε, δ-Minimize problem in 2D: the (first) intersection between an oriented line segment and a curve.

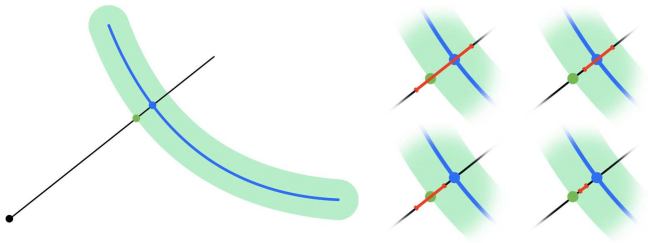

Fig. 3. An ε, δ-Minimize Problem: the first intersection between the black segment and the blue line is sought. The segment starts at the black bullet. The blue bullet corresponds to solution F^* of the corresponding Minimization Problem in the continuum. The green bullet corresponds to the solution 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.

Remark 1. 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 to guarantee that a subregion $\sigma \subset \Sigma$ contains 0, 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 2 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.

Remark 2. 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 5.5, 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. We refer to Section A for a more thorough discussion of this choice.

Remark 3. The algorithms presented in Section 4 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. See Section B for a more thorough discussion.

3.2 How MiSo Works

MiSo is used to specify a problem in the classes given in Definitions 3 and 4 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 [Snyder 1992]: 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 5.5).

3.3 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 6.

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 $C(u, v, t) \leq 0$.

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 $\varepsilon > 0$ and $\delta > 0$.

Figure 4 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 5 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), 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.

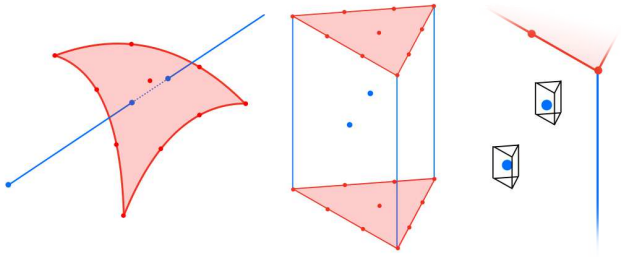

Fig. 4. Intersection between a straight-line segment and a cubic triangular patch. Left: in physical space, the origin of the segment is to the bottom left; the red bullets are the Lagrange control points defining the patch; the Minimize Problem seeks the first (leftmost) of the two intersections (blue 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.

```
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
```

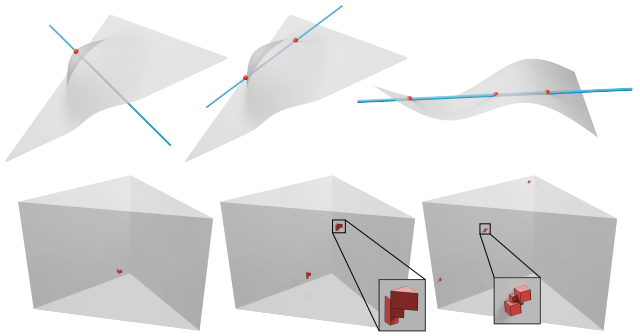

Fig. 5. Cubic triangle - linear segment intersection. From the top: MiSo specification of the problem; Intersecting geometries for 1, 2, 3 intersecting points; Corresponding solutions of the Solve problem in parametric space are clusters of prisms. The value of ε is much larger than the value used in the experiments in Section 6 to produce visible results.

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 $\tilde{\Phi}$.

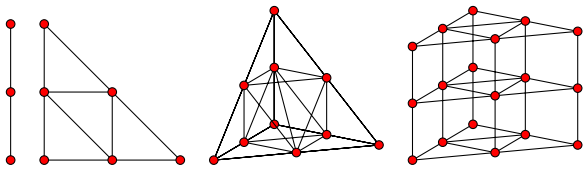

Fig. 6. Subdivision rules for simplices of dimension 1, 2, 3, and for a composite space $\triangle^2 \times \triangle^1$.

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 Algorithms

The generic algorithms Solve and Minimize are based on interval analysis and inspired by those originally described by Snyder [1992]. 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.

4.1 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 6). 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 6.

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.

4.2 Inclusion Functions

Given a function $f : \Omega \longrightarrow Y$, an *inclusion function* for f is any function

$$\square f : \mathcal{P}\Omega \longrightarrow \mathcal{P}Y$$

such that for any $D \subseteq \Omega$ we have $f(D) \subseteq \square f(D)$, where $\mathcal{P}$ denotes the power set. Here, we consider the specific case of scalar functions and inclusion functions that return an interval.

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 $w(D) = \min_{A \supseteq D} w(A)$ with $A \in \mathbb{I}^n$.

Let $f : \Sigma \subseteq \mathbb{R}^n \longrightarrow \mathbb{R}$ be a real function. We define an (interval) *inclusion function* for f as $\square f : \mathcal{P}\Sigma \longrightarrow \mathbb{I}$ such that, for any $D \subseteq \Sigma$ we have $\forall \xi \in D \quad f(\xi) \in \square f(D)$.

Definition 5 (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 $f(\xi)$.

A convergent inclusion function can be used to find a root of a function f by subdividing the initial domain Σ until it becomes sufficiently small [Snyder 1992].

Given an expression of a function f, our system automatically generates an inclusion function for f using interval arithmetic [Attene 2020]. Refer to Section 5.2 for a list of currently supported expressions.

4.3 Pseudocode

We describe the main structure of our generic solvers Solve and Minimize. Their programming interfaces follow from the problem statements of Definitions 3 and 4.

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.

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 1. 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 4.1).

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.

Algorithm 1 Solve

Input: Initial domain Σ, constraint expression C, acceptance threshold ε, boolean FindOne, iteration limit $K_{\max}$
Output: Solution set S

```
1:  L ← Queue
2:  S ← ∅
3:  K ← 0                                              ▷ iteration counter
4:  ProcessRegion(Σ)
5:  loop
6:      if IsEmpty(L) then
7:          return S
8:      if K ≥ K_max then
9:          InsertAll(S, L)                            ▷ ensures we are conservative
10:         return S
11:     σ ← Pop(L)
12:     K ← K + 1
13:     for all q ∈ {1, ..., Q} do
14:         ProcessRegion(ψ^q(σ))                      ▷ subdivide σ
15:         if FindOne ∧ ¬IsEmpty(S) then
16:             return S
17: function ProcessRegion(ρ)
18:     if [□C(ρ)]_lo ≤ 0 then
19:         if FindOne then ▷ look for a point in the feasible region
20:             V ← Sample(ρ)
21:             for all v ∈ V do
22:                 if [□C(v)]_lo ≤ 0 ∧ [□C(v)]_hi ≤ ε then
23:                     Insert(S, v)                   ▷ v ∈ Φ̃
24:                     return
25:         if [□C(ρ)]_hi ≤ ε then
26:             Insert(S, ρ)                           ▷ ρ ⊂ Φ̃
27:         else
28:             Enqueue(L, ρ)                          ▷ ρ will be subdivided
```

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 than F^* and its upper end is higher than F^*_ε. The pseudo-code is given in Algorithm 2.

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 $> \delta$.

Algorithm 2 Minimize

Input: Initial domain Σ, constraint expression C, constraint and objective acceptance thresholds ε and δ, iteration limit $K_{\max}$
Output: Interval $\tilde{F}^*$

```
1:  l ← ∞                                              ▷ lower bound of F̃*
2:  u ← ∞                                              ▷ upper bound of F̃*
3:  P ← PriorityQueue
4:  K ← 0
5:  ProcessRegion(Σ)
6:  loop
7:      if u − l ≤ δ then
8:          return [l, max{l, u}]                      ▷ reached the desired precision
9:      if IsEmpty(P) then
10:         return [∞, ∞]                              ▷ feasible region is empty
11:     if K ≥ K_max then
12:         return [l, max{l, u}]                      ▷ return current approximation
13:     σ ← Pop(P)
14:     K ← K + 1
15:     l ← [□F(σ)]_lo                                 ▷ update lower bound
16:     for all q ∈ {1, ..., Q} do                     ▷ subdivide σ
17:         ProcessRegion(ψ^q(σ))
18: function ProcessRegion(ρ)
19:     if [□C(ρ)]_lo ≤ 0 ∧ [□F(ρ)]_lo ≤ u then
20:         V ← Sample(ρ)                              ▷ look for a point in the buffer region
21:         for all v ∈ V do
22:             if [□C(v)]_hi ≤ ε then
23:                 u ← min{u, [□F(v)]_hi}             ▷ update upper bound
24:         Enqueue(P, ρ, Priority(ρ))                 ▷ ρ will be subdivided
25: function Priority(ρ)
26:     return −[□F(ρ)]_lo                             ▷ smallest lower bound on □F goes first
```

5 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 5.2). The related inclusion function is assembled by analyzing this tree and applying specific rules for the leaves and the inner operators (Section 5.3). The inclusion functions for polynomials exploit the convex hull property of their Bézier representation while rules for the inner operators stem from interval analysis (Section C).

5.1 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, 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`.

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.

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 calling `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.

5.2 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)

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.

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: these 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.

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 6.

5.3 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 combinations of intervals at inner nodes of the tree follow the rules summarized in Section C. 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 [Aanjaneya et al. 2022, 2024; Lim and Nagarakatte 2022].

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 6).

5.4 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 polynomials 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 [Johnen et al. 2014], which we summarize here. 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 $T_{\mathcal{B}\leftarrow\mathcal{L}} = T^{-1}_{\mathcal{L}\leftarrow\mathcal{B}}$ 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 (Section 5.4).

5.5 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 upon;
- 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 3.3, 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.

In both cases, our C++ code has the NFG library [Attene 2020, 2025] as the only dependency, for interval arithmetic.

Extensions. By default, the subdivision rules in the generated code are those explained in Section 5.4, 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 6.2, we use a customized version of Minimize in a comparison against [Chen et al. 2024] to comply with their termination conditions.

Extending MiSo to support a wider class of expressions, e.g., including transcendental functions and/or rational polynomials is more delicate, since it requires modifying MiSo-core, and will be the subject of future work.

6 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.

6.1 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 [Akenine-Möller et al. 2018] for an overview of fast methods used in real-time physics engines and to [Wang et al. 2021] for an overview of conservative methods used for offline simulation.

Explicit Cubic Triangle-Linear Segment Intersection. We consider the problem used in Section 3.3 as a didactic example. First we formulate this problem as a Solve and execute the query on 3 cases (Figure 5). 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.

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 great care (see Section C). 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 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 7. One query finds an intersection with 74181 regions in 4D parametric space in 338ms to precision 10^{-2}.

6.2 Dynamic Objects Intersections

Static collision detection might miss collisions in dynamic simulation when objects move quickly. 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 [Chen et al. 2024; Wang et al. 2021] or nonlinear [Ferguson et al. 2021; Zhang et al. 2023] trajectory.

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

```
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)
```

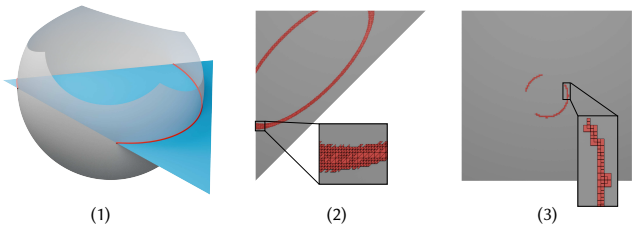

Fig. 7. Intersection of a triangle with a hollow sphere. (1) Both primitives are expressed in parametric form, the sphere requiring rational polynomials. They intersect at two disjoint arcs (in red). (2) Our solutions in the parametric domain of the triangle. (3) Our solutions in the parametric domain 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.

`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 $[[t]_{\text{lo}}, [t]_{\text{hi}}]$. If the algorithm verifies that the system is in an invalid state at time $[t]_{\text{hi}}$, 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.

```python
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)
```

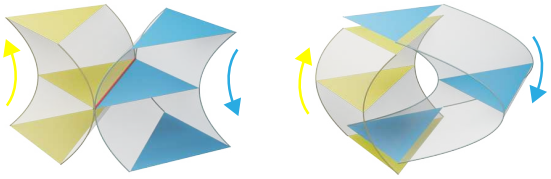

Fig. 8. Linear triangle CCD on quadratic trajectories. The direction of trajectories is marked with arrows. Left: Collision occurs and is detected on a whole edge (red) at a single time; primitives are disjoint at all other times. Right: no collision at all.

```python
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)
```

Fig. 9. Linear segment CCD on linear trajectories. The code, used for comparisons with [Wang et al. 2021], is similar to the one in Figure 8.

Linear Triangle - Quadratic Trajectories. The program in Figure 8 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 744μs. The true time of collision is 1/2: as in Section 6.1, this is a conservative solution to the MINIMIZE problem. In the second case, our algorithm correctly classifies the trajectory as collision-free in 34μs, returning $[\infty, \infty]$.

Comparison with [Wang et al. 2021]. We consider the problem of continuous collision detection of a pair of edges whose vertices are moving on linear trajectories. Figure 9 shows the specification and an example query. We compare the result of MiSo with the heavily optimized algorithm in [Wang et al. 2021] in Table 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.

```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)
```

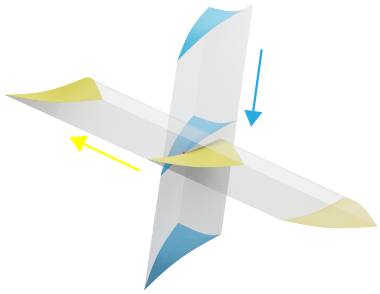

Fig. 10. Cubic triangles CCD on linear trajectories. The code is similar to the one in Figure 8. The result at the bottom is an example of the queries used for comparisons with [Chen et al. 2024; Zhang et al. 2023].

Table 2. EECCD on the handcrafted (15K queries) and simulation (41M queries) datasets from [Wang et al. 2021], 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

Comparison with [Zhang et al. 2023] and [Chen et al. 2024]. Adapting our algorithm to higher-order collision detection requires minimal changes to our MiSo program. In Figure 10, we show a MiSo program to compute the cubic triangle to cubic triangle continuous collision problem introduced in [Zhang et al. 2023]. We compare our method with the SOS-based solver in [Zhang et al. 2023] and the inclusion-based solver in [Chen et al. 2024] on randomly generated queries.

For a fair comparison, we employ a slightly modified version of MINIMIZE that uses the same termination condition as [Chen et al. 2024], as discussed in Section A. We also substitute the squared L_2 distance with the L_∞ distance as in [Chen et al. 2024]. Their method

Table 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 [Chen et al. 2024]'s method.

Method	*ms*/query
[Chen et al. 2024] TDI + OBB	33
[Chen et al. 2024] "traditional" + OBB	241
[Zhang et al. 2023] (SOSP)	7528
MiSo w/ interval arithmetic backend	482
MiSo w/ floating point backend	176

```
1  def QuadPatchCCD(variant=0):
2    with pymiso.Context() as miso:
3      X1 = miso.variables()
4      X2 = miso.variables()
5      Y1 = miso.variables()
6      Y2 = miso.variables()
7      T = miso.variables()
8      elem_a = miso.poly_space((X1, 1), (X2, 1))
9      elem_b = miso.poly_space((Y1, 1), (Y2, 1))
10     pb = miso.bases.BEZIER
11     xa0 = elem_a.geo_map(pb, 3)
12     xa1 = elem_a.geo_map(pb, 3)
13     xb0 = elem_b.geo_map(pb, 3)
14     xb1 = elem_b.geo_map(pb, 3)
15     xa = ((xa0 * (1-T)) + (xa1 * T)).collapse()
16     xb = ((xb0 * (1-T)) + (xb1 * T)).collapse()
17     dist = ((xb-xa)**2).sum() #hybrid version
18     if variant == 'Bezier':
19       dist = dist.collapse() #Bezier-based version
20     elif variant == 'NIE':
21       dist = dist.expand() #NIE-based version
22     sd = [miso.subdiv_strategy(), miso.subdiv_strategy(T)]
23     miso.generate('./src/generated', 'QuadCCD', dist, objective=T, strategies=sd)
```

Fig. 11. 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 $[0, 1]^5$).

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 D.

Our approach is more efficient than SOS, while being guaranteed conservative. Compared to [Chen et al. 2024], 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 [Chen et al. 2024] (See Table 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 [Chen et al. 2024] relies on a numerical tolerance.

6.3 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.

Table 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

First, we consider three possible specifications of the CCD problem for bilinear quad patches, shown in Figure 11. We run the experiment on 100 random collision pairs generated as in [Chen et al. 2024], 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 10:

(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.

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 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, 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.

```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)
```

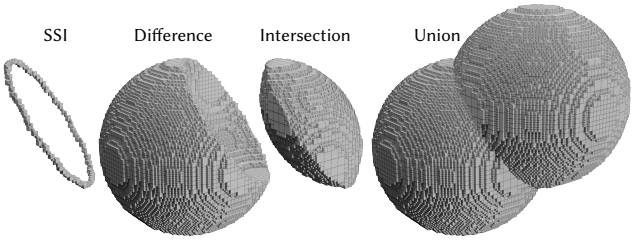

Fig. 12. Boolean and SSI operations between two implicit spheres. Top: MiSo specification is parametrized on the type of Boolean operation. Bottom, from left to right: results of SSI, boolean difference, intersection, and union with $\varepsilon = 10^{-2}$.

Table 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

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

6.4 Implicit Booleans

We take inspiration from [Snyder 1991] and cast Boolean operations between two implicit spheres as a Solve problem (Figure 12). We report statistics in Table 5: changing the primitive types and operations are minor changes in the specifications, making our compiler a powerful tool to compute conservative Boolean solutions between implicit and explicit primitives.

6.5 Minimal Distance

Computing distances between linear meshes is a classical problem that has been extensively studied [Baerentzen and Aanaes 2005; Bartoň et al. 2010; Carretero and Nahon 2005; Cignoni et al. 1998; Guezlec 2001; Jones et al. 2006; Kang et al. 2019; Zheng et al. 2022]

```python
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)
```

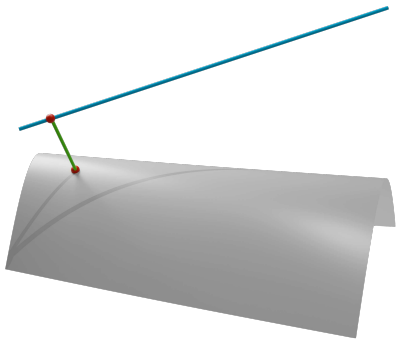

Fig. 13. Distance between quadratic Bézier patch (gray) and segment (blue). The direction (green) of the closest point pairs (red) is shown.

and for which robust and efficient algorithms exist. Its curved version is more challenging [Alt and Scharf 2008; Kim et al. 2013; Son et al. 2021].

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 13. 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 returns [0.760585,0.760685].

6.6 Geometrical Validity Check

In finite element simulation, it is usually assumed that the geometric map of each element is injective or, equivalently, that the Jacobian of the geometric map is positive everywhere on the element domain. If this property holds, a finite element is said to be *valid*. For linear elements, validity can be tested using the *Orient2D* and *Orient3D* predicates introduced in [Shewchuk 1997]. For high-order elements – including hexahedral meshes, which are at least trilinear – the problem is more challenging, as the Jacobian is not constant in the element domain [Marschner et al. 2020b]. Johnen et al. [2014] propose an inclusion-based method, which we compare with our MiSo implementation (Figure 14).

In Table 6 we report results on a mesh consisting of 48050 valid tetrahedra and 6935 invalid ones (Figure 14 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 implementation in [Johnen et al. 2014], which uses numerical tolerances and is thus not provably conservative. The same program, compiled with our floating point backend, leads to a faster ($3\mu s$) average runtime.

Table 6. Geometrical validity of order 3 tetrahedra. We show the per-query runtime averaged on 54985 queries.

Method	μs/query
[Johnen et al. 2014]	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 = x.jacobian().det().collapse() #Jacobian determinant
        miso.generate('./src/generated', f'ValidityD{D}P{P}', jd)
```

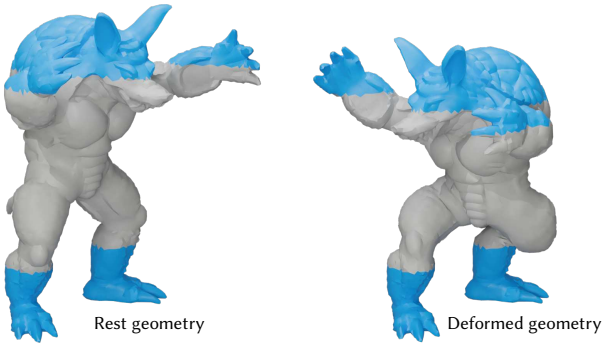

Fig. 14. Geometrical validity check for cubic tetrahedral elements. The shape to the left consists of all valid tetrahedra; we test the deformed model to the right, which contains geometrically inverted elements.

6.7 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.

7 Conclusions

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 [Akenine-Möller et al. 2024] can be automated: we hope with future improvements of MiSo (for example using the strategy proposed in [Chen et al. 2024] for time-dependent problems) that the runtime could become competitive or even surpass manually implemented codes.

Limitations. The main limitation of our compiler is that it is inherently limited to problems with a small number of dimensions: our subdivision approach becomes impractical otherwise. However, as we demonstrated in the paper, there are many important low-dimensional problems of this kind in graphics. A second addressable limitation is that our compiler slows down with high-order polynomials, as it relies on symbolic computation. Hybrid strategies mixing symbolic and numerical evaluation might ameliorate this problem, which we only encountered when writing queries involving elements of high polynomial degree (over 5).

Future work. MiSo is a first step toward the compilation of robust and efficient solvers for Solve and Minimize problems. There are many avenues of future work including: (1) adding support for transcendental functions using [Aanjaneya et al. 2022, 2024], (2) parallelization of the search on shared memory multi-core processors, (3) automatic, possibly data-driven, optimization of the subdivision strategy and inclusion function computation, (4) adoption and generalization of the strategy in [Chen et al. 2024] for time-dependent problems.

Reproducibility. We will release an open-source implementation of our compiler to make our results reproducible and to foster the adoption of MiSo in the graphics community.

Acknowledgments

This work was supported in part through the NYU IT High Performance Computing resources, services, and staff expertise. This work was also partially supported by the NSF grants OAC-2411349 and IIS-2313156, a gift from Adobe Research, and by the MUR-PRIN Project N. 2022YB4NRS "FabDesign".

References

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

M. Aanjaneya, J.P. Lim, and S. Nagarakatte. 2024. The RLIBM Project. https://github.com/rutgers-apl/The-RLIBM-Project

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

T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire. 2024. Real-Time Rendering - Ray Tracing Resources Page. https://www.realtimerendering.com/intersections.html

H. Alt and L. Scharf. 2008. Computing the Hhausdorff Distance between Curved Objects. *International Journal of Computational Geometry & Applications* 18, 04 (2008), 307–320.

M. Attene. 2020. Indirect predicates for Geometric Constructions. *Computer-Aided Design* 126 (2020), 102856.

M. Attene. 2025. NFG - Numbers For Geometry. https://github.com/MarcoAttene/NFG

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

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

G. Louis Bernstein and F. Kjolstad. 2016. Why New Programming Languages for Simulation? *ACM Trans. Graph.* 35, 2, Article 20e (May 2016), 3 pages.

T. Brochu, E. Edwards, and R. Bridson. 2012. Efficient geometrically exact continuous collision detection. *ACM Trans. Graph.* 31, 4, Article 96 (July 2012), 7 pages.

J.A. Carretero and M.A. Nahon. 2005. 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 (2005), 1144–1155.

X. Chen, C. Yu, X. Ni, M. Chu, B. Wang, and B. Chen. 2024. A Time-Dependent Inclusion-Based Method for Continuous Collision Detection between Parametric Surfaces. *ACM SIGGRAPH Computer Graphics* 43, 6 (2024).

P. Cignoni, C. Rocchini, and R. Scopigno. 1998. Metro: Measuring Error on Simplified Surfaces. *Computer Graphics Forum* 17, 2 (1998), 167–174.

G.E. Collins and A.G. Akritas. 1976. 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)*. ACM Press, 272–275.

CVX Research, Inc. 2012. CVX: Matlab Software for Disciplined Convex Programming, version 2.0. https://cvxr.com/cvx.

Etienne De Klerk, Monique Laurent, and Pablo A. Parrilo. 2006. A PTAS for the minimization of polynomials of fixed degree over the simplex. *Theoretical Computer Science* 361, 2-3 (Sept. 2006), 210–225. doi:10.1016/j.tcs.2006.05.011

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

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

Z. Ferguson, M. Li, T. Schneider, F. Gil-Ureta, T. Langlois, C. Jiang, D. Zorin, D.M. Kaufman, and D. Panozzo. 2021. Intersection-free Rigid Body Dynamics. *ACM Transactions on Graphics (SIGGRAPH)* 40, 4, Article 183 (2021).

J. Garloff, C. Jansson, and A.P. Smith. 2003. Lower bound functions for polynomials. *J. Comput. Appl. Math.* 157, 1 (2003), 207–225.

M. Grant and S. Boyd. 2008. Graph implementations for nonsmooth convex programs. In *Recent Advances in Learning and Control*, V. Blondel, S. Boyd, and H. Kimura (Eds.). Springer-Verlag Limited, 95–110.

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

S. Hadap, D. Eberle, P. Volino, M.C. Lin, S. Redon, and C. Ericson. 2004. Collision detection and proximity queries. In *ACM SIGGRAPH 2004 Course Notes* (Los Angeles, CA) *(SIGGRAPH '04)*. ACM, New York, NY, USA, 15–es.

E.R. Hansen and R.I. Greenberg. 1983. An interval Newton method. *Appl. Math. Comput.* 12, 2-3 (May 1983), 89–98.

P. Herholz, X. Tang, T. Schneider, S. Kamil, D. Panozzo, and O. Sorkine-Hornung. 2022. Sparsity-Specific Code Optimization using Expression Trees. *ACM Transactions on Graphics* 41, 5 (2022), 175:1–19.

K. Hormann, L. Kania, and C. Yap. 2021. 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* (Virtual Event, Russian Federation) *(ISSAC '21)*. ACM, New York, NY, USA, 193–200.

K. Hormann, C. Yap, and Y.S. Zhang. 2023. Range Functions of Any Convergence Order and Their Amortized Complexity Analysis. In *Computer Algebra in Scientific Computing: 25th International Workshop, CASC 2023* (Havana, Cuba). Springer-Verlag, Berlin, Heidelberg, 162–182.

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

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

A. Johnen, J.-F. Remacle, and C. Geuzaine. 2013. Geometrical validity of curvilinear finite elements. *J. Comput. Phys.* 233 (2013), 359–372.

A. Johnen, J. F. Remacle, and C. Geuzaine. 2014. Geometrical Validity of High-Order Triangular Finite Elements. *Eng. with Comput.* 30, 3 (2014), 375–382.

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

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

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

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

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. 2016. Simit: A Language for Physical Simulation. *ACM Trans. Graph.* 35, 2, Article 20 (May 2016), 21 pages.

S. Lengagne, R. Kalawoun, F. Bouchon, and Y. Mezouar. 2020. Reducing pessimism in Interval Analysis using B-splines Properties: Application to Robotics. *Reliable Computing* 27 (2020), 63–87.

B. Lévy. 2016. Robustness and efficiency of geometric programs: The Predicate Construction Kit (PCK). *Computer-Aided Design* 72 (2016), 3–12.

M. Li, D.M. Kaufman, and C. Jiang. 2021b. Codimensional incremental potential contact. *ACM Trans. Graph.* 40, 4, Article 170 (July 2021), 24 pages.

Y. Li, S. Kamil, K. Crane, A. Jacobson, and Y. Gingold. 2024. I♡Mesh: A DSL for Mesh Processing. *ACM Trans. Graph.* 43, 6 (2024).

Y. Li, S. Kamil, A. Jacobson, and Y. Gingold. 2021a. I♡LA: Compilable Markdown for Linear Algebra. *ACM Transactions on Graphics (TOG)* 40, 6 (2021).

J.P. Lim and S. Nagarakatte. 2022. One polynomial approximation to produce correctly rounded results of an elementary function for multiple representations and rounding modes. *Proc. ACM Program. Lang.* 6, POPL, Article 3 (Jan. 2022), 28 pages.

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

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

Z. Marschner, P. Zhang, D. Palmer, and J. Solomon. 2021. Sum-of-squares geometry processing. *ACM Trans. Graph.* 40, 6, Article 253 (Dec. 2021), 13 pages.

J.P. Merlet. 2007. Interval Analysis and Robotics. *Robotics Research* 66, 147–156. doi:10.1007/978-3-642-14743-2_13

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. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (2017), e103.

A. Meyer and S. Pion. 2008. FPG: A code generator for fast and certified geometric predicates. In *Real numbers and computers*. 47–60.

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

J. Nocedal and S.J. Wright. 2006. *Numerical Optimization*. Springer, New York.

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

J. Smith and S. Schaefer. 2015. Bijective Parameterization with Free Boundaries. *ACM Trans. Graph.* 34, 4, Article 70 (2015), 9 pages.

J.M. Snyder. 1991. *Generative Modeling: An Approach to High Level Shape Design for Computer Graphics and CAD*. Ph. D. Dissertation.

J Snyder. 1992. Interval Analysis For Computer Graphics. *ACM SIGGRAPH* (1992), 121–130.

J. Snyder, A.R. Woodbury, K. Fleischer, B. Currin, and A.H. Barr. 1993. Interval Methods for Multi-Point Collisions between Time-Dependent Curved Surfaces. In *ACM SIGGRAPH*. ACM.

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

V. Stahl. 1995. *Interval Methods for Bounding the Range of Polynomials and Solving Systems of Nonlinear Equations*. Ph. D. Dissertation.

M. Tang, R. Tong, Z. Wang, and D. Manocha. 2014. Fast and exact continuous collision detection with Bernstein sign classification. *ACM Transactions on Graphics* 33, 6 (2014), 1–8. doi:10.1145/2661229.2661237

X. Tang, T. Schneider, S. Kamil, A. Panda, J. Li, and D. Panozzo. 2020. EGGS: Sparsity-Specific Code Generation. *Computer Graphics Forum* 39, 5 (Aug. 2020), 209–219.

B. Wang, . Ferguson, X. Jiang, M. Attene, D. Panozzo, and T. Schneider. 2022. Fast and Exact Root Parity for Continuous Collision Detection. *Computer Graphics Forum (Proceedings of Eurographics)* 41, 2 (2022), 9 pages.

B. Wang, Z. Ferguson, T. Schneider, X. Jiang, M. Attene, and D. Panozzo. 2021. A Large-scale Benchmark and an Inclusion-based Algorithm for Continuous Collision Detection. *ACM Trans. Graph.* 40, 5, Article 188 (2021), 16 pages.

Huamin Wang. 2014. Defending continuous collision detection against errors. *ACM Transactions on Graphics* 33, 4 (July 2014), 1–10.

Wolfram Research, Inc. 2023. Mathematica, Version 13.3. https://www.wolfram.com/mathematica Champaign, IL.

P. Zhang, Z. Marschner, J. Solomon, and R. Tamstorf. 2023. Sum-of-Squares Collision Detection for Curved Shapes and Paths. In *ACM SIGGRAPH 2023 Conference Proceedings* (Los Angeles, CA, USA). ACM, New York, NY, USA, Article 76, 11 pages.
Z. Zhang, Y.-J. Chiang, and C. Yap. 2024. Theory and Explicit Design of a Path Planner for an SE(3) Robot. arXiv:2407.05135 [cs.RO] https://arxiv.org/abs/2407.05135
Y. Zheng, H. Sun, X. Liu, H. Bao, and J. Huang. 2022. Economic Upper Bound Estimation in Hausdorff Distance Computation for Triangle Meshes. *Computer Graphics Forum* 41, 1 (2022), 46–56.

A Precision in the domain or in the range?

Precision in the Domain. In Definition 3 we could define Φ_ε as a region containing Φ with all the points within distance ε of Φ. Similarly, in Definition 4 (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 6.2 solely for comparison with [Chen et al. 2024] to comply with their choice.

Precision in the Range. As stated in Definition 3, 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 4, 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 3). 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 2 Right and Figure 4), 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 [Snyder 1992].

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.

B Discontinuous functions

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 5. 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 A, 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.

C Interval arithmetic

Interval arithmetic provides a set of operations on real intervals $\mathbb{I}$ such that if $x \in a = [\underline{a}, \overline{a}] \in \mathbb{I}$ and $y \in b = [\underline{b}, \overline{b}] \in \mathbb{I}$, then $x \star y \in a \star b$, where $\star$ in the right-hand side is the interval version of operation $\star$ on reals.

We employ interval arithmetic for two primary purposes:

(1) Interval Analysis: This is fundamental to our approach, utilized consistently for tasks such as evaluating inclusion functions.
(2) Rounding Error Handling: When the "conservative backend" is selected, interval arithmetic is specifically employed to account for the inherent imprecision of floating-point computations. Here, all input values are initially represented as singleton intervals (intervals with identical endpoints). Subsequently, for each interval operation, the lower bound of the result is rounded downwards, and the upper bound is rounded upwards, ensuring an enclosure of the true result.

Our implementation leverages the `Indirect_Predicates` library [Attene 2020]. The following operations and relations on intervals are supported in MiSo:

$$-[\underline{x}, \overline{x}] = [-\overline{x}, -\underline{x}]$$
$$[\underline{x}, \overline{x}] + [\underline{y}, \overline{y}] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$$
$$[\underline{x}, \overline{x}][\underline{y}, \overline{y}] = [\min(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}), \max(\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y})]$$
$$\min([\underline{x}, \overline{x}], [\underline{y}, \overline{y}]) = [\min(\underline{x}, \underline{y}), \min(\overline{x}, \overline{y})]$$
$$\max([\underline{x}, \overline{x}], [\underline{y}, \overline{y}]) = [\max(\underline{x}, \underline{y}), \max(\overline{x}, \overline{y})]$$
$$\mathrm{abs}([\underline{x}, \overline{x}]) = \begin{cases} [\underline{x}, \overline{x}] & \text{if } \underline{x} > 0 \\ [-\overline{x}, -\underline{x}] & \text{if } \overline{x} < 0 \\ [0, \max(-\underline{x}, \overline{x})] & \text{if } \underline{x} \leq 0 \leq \overline{x} \end{cases}$$

The division of an interval by an exact number different from zero is also available, and it is a safe operation. 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 results may contain NaNs for some combinations of operands that contain infinite values and/or zeroes.

While infinite values are acceptable values for our algorithms, handling potentially NaN values requires additional checks.

D Polyhedral bounding boxes

The following program, used in Section 6.2, 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

$$(1,0,0),(0,1,0),(0,0,1),$$
$$(1,1,0),(1,0,1),(0,1,1),(1,-1,0),(1,0,-1),(0,1,-1),$$
$$(1,1,1),(1,1,-1),(1,-1,1),(1,-1,-1).$$

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)
```