Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings

Ábrahám, E., Davenport, J. H., England, M. & Kremer, G.

Published PDF deposited in Coventry University's Repository

Original citation:
Ábrahám, E, Davenport, JH, England, M & Kremer, G 2021, 'Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings', Journal of Logical and Algebraic Methods in Programming, vol. 119, 100633.
https://dx.doi.org/10.1016/j.jlamp.2020.100633

DOI 10.1016/j.jlamp.2020.100633
ESSN 2352-2208

Publisher: Elsevier

This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Journal of Logical and Algebraic Methods in Programming 119 (2021) 100633

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings

Erika Ábrahám [a,*], James H. Davenport [b], Matthew England [c], Gereon Kremer [a]

[a] RWTH Aachen University, Germany
[b] University of Bath, UK
[c] Coventry University, UK

ARTICLE INFO

Article history:
Received 15 April 2019
Received in revised form 22 November 2020
Accepted 23 November 2020
Available online 27 November 2020

Keywords:
Satisfiability modulo theories
Non-linear real arithmetic
Cylindrical algebraic decomposition
Real polynomial systems

ABSTRACT

We present a new algorithm for determining the satisfiability of conjunctions of non-linear polynomial constraints over the reals, which can be used as a theory solver for satisfiability modulo theory (SMT) solving for non-linear real arithmetic. The algorithm is a variant of Cylindrical Algebraic Decomposition (CAD) adapted for satisfiability, where solution candidates (sample points) are constructed incrementally, either until a satisfying sample is found or sufficient samples have been sampled to conclude unsatisfiability. The choice of samples is guided by the input constraints and previous conflicts.

The key idea behind our new approach is to start with a partial sample; demonstrate that it cannot be extended to a full sample; and from the reasons for that rule out a larger space around the partial sample, which build up incrementally into a cylindrical algebraic covering of the space. There are similarities with the incremental variant of CAD, the NLSAT method of Jovanović and de Moura, and the NuCAD algorithm of Brown; but we present worked examples and experimental results on a preliminary implementation to demonstrate the differences to these, and the benefits of the new approach.

© 2020 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

1.1. Real algebra

Formulae in *Real Algebra* are Boolean combinations of polynomial constraints with rational coefficients over real-valued variables, possibly quantified. Real algebra is a powerful logic suitable to express a wide variety of problems throughout science and engineering. The 2006 survey [1] gives an overview of the scope here. Recent new applications include bio-chemical network analysis [2], economic reasoning [3], and artificial intelligence [4]. Having methods to analyse real algebraic formulae allows us to better understand those problems, for example, to find one solution, or symbolically describe all possible solutions for them.

In this paper we restrict ourselves to formulae in which every variable is existentially quantified. This falls into the field of *Satisfiability Modulo Theories* (*SMT*) solving, which grew from the study of the Boolean SAT problem to encompass other

[*] Corresponding author.
E-mail addresses: abraham@cs.rwth-aachen.de (E. Ábrahám), J.H.Davenport@bath.ac.uk (J.H. Davenport), Matthew.England@coventry.ac.uk (M. England), gereon.kremer@cs.rwth-aachen.de (G. Kremer).

https://doi.org/10.1016/j.jlamp.2020.100633
2352-2208/© 2020 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

domains for logical formulae. In traditional SMT solving, the search for a solution follows two parallel threads: a *Boolean* search tries to satisfy the Boolean structure of φ, accompanied by a *theory* search that tries to satisfy the polynomial constraints that are assumed to be `True` in the current Boolean search. To implement such a technique, we need a decision procedure for the theory search that is able to check the *satisfiability of conjunctions of polynomial constraints*, in other words, the *consistency of polynomial constraint sets*. The development of such methods has been highly challenging.

Tarski showed that the Quantifier Elimination (QE) problem is decidable for real algebra [5]. That means, for each real-algebraic formula $\forall x.\varphi$ or $\exists x.\varphi$ it is possible to construct another, semantically equivalent formula using the same variables as used to express φ but without referring to x. For conjunctions of polynomial constraints it means that it is possible to decide their satisfiability, and for any satisfiable instance provide satisfying variable values. Tarski's results were ground breaking, but his constructive solution was non-elementary (with a time complexity greater than all finite towers of powers of 2) and thus not applicable.

1.2. Cylindrical algebraic decomposition

An alternative solution was proposed by Collins in 1975 [6]. Since its invention, Collins' *Cylindrical Algebraic Decomposition* (*CAD*) method was the target of numerous improvements and has been implemented in many Computer Algebra Systems. Its theoretical complexity is doubly exponential.[1] The fragment of our interest, which excludes quantifier alternation, has lower theoretical complexity of singly exponential time (see for example [11]), however, currently no algorithms are implemented that realise this bound in general (see [12] for an analysis as to why). Current alternatives to the CAD method are efficient but incomplete methods using, e.g., linearisation [13,14], interval constraint propagation [15,16], virtual substitution [17,18], subtropical satisfiability [19] and Gröbner bases [20].

Given a formula in real algebra, a CAD may be produced which decomposes real space into a finite number of disjoint cells so that the truth of the formula is invariant within each cell. Collin's CAD achieved this via a decomposition on which each polynomial in the formula has constant sign. Querying one sample point from each cell can then allow us to determine satisfiability, or perform quantifier elimination.

A full decomposition is often not required and thus savings can be made by adapting the algorithm to terminate early. For example, once a single satisfying sample point is found we can conclude satisfiability of the formulae.[2] This was first proposed as part of the *partial CAD* method for QE [22]. The natural implementation of CAD for SMT performs the decomposition incrementally by polynomial, refining CAD cells by subdivision and querying a sample from each new unsampled cell before performing the next subdivision.

Even if we want to prove unsatisfiability, the decomposition of the state space into sign-invariant cells is not necessary. There has been a long development in how simplifications can be made to achieve truth-invariance without going as far as sign-invariance such as [23], [24], [25].

1.3. New real algebra methods inspired by CAD

The present paper takes this idea of reducing the work performed by CAD further, proposing that the decomposition need not even be *disjoint*, and neither sign- nor truth-invariant as a whole for the set of constraints. We will produce cells incrementally, each time starting with a sample point, which if unsatisfying we generalise to a larger cylindrical cell using CAD technology. We continue, selecting new samples from outside the existing cells until we find either a satisfying sample, or the entire space is covered by our collection of overlapping cells which we call a *Cylindrical Algebraic Covering*.

Our method shares and combines ideas from two other recent CAD inspired methods: (1) the NLSAT approach by Jovanović and de Moura, an instance of the *model constructing satisfiability calculus* (mcSAT) framework [26]; and (2) the *Non-uniform* CAD (NuCAD) approach of Brown [27], which is a decomposition of the state space into disjoint cells that are *truth*-invariant for the *conjunction* of a set of polynomial constraints, but with a weaker relationship between cells (the decomposition is not cylindrical).

We demonstrate later with worked examples how our new approach outperforms a traditional CAD while still differing from the two methods above: with more effective learning from conflicts than NuCAD and, unlike NLSAT, an SMT-compliant approach which keeps theory reasoning separate from the SAT solver.

1.4. Paper structure

We continue in Section 2 with preliminary definitions and descriptions of the alternative approaches. We then present our new algorithm, first the intuition in Section 3 and then formally in Section 4. Section 5 contains illustrative worked

[1] Doubly exponential in the number of variables (quantified or free). The double exponent does reduce by the number of equational constraints in the input [7], [8], [9]. However, the doubly-exponential behaviour is intrinsic: in the sense that classes of examples have been found where the solution requires a doubly exponential number of symbols to write down [10].

[2] There are other approaches to avoiding a full decomposition, for example, [21] suggests how segments of the decomposition of interest can be computed alone based on dimension or presence of a variety.

examples while Section 6 describes how our implementation performed on a large dataset from the SMT-LIB [28]. We conclude and discuss further research directions in Section 7.

2. Preliminaries

2.1. Formulae in real algebra

The general problem of solving, in the sense of eliminating quantifiers from, a quantified logical expression which involves polynomial equalities and inequalities with real-valued variables is an old one.

Definition 1. Consider a quantified proposition in *prenex normal form*[3]:

$$Q_1x_1 \dots Q_mx_mF(x_1, \dots, x_m, x_{m+1}, \dots, x_n), \tag{1}$$

where each Q_i is either $\exists$ or $\forall$ and F is a semi-algebraic proposition, i.e. a Boolean combination of constraints

$$p_j(x_1, \dots, x_m, x_{m+1}, \dots, x_n)\sigma_j 0,$$

where p_j are polynomials with rational coefficients and each σ_j is an element of $\{<, \leq, >, \geq, \neq, =\}$.

The *Quantifier Elimination (QE) Problem* is to determine an equivalent quantifier-free semi-algebraic proposition $G(x_{m+1}, \dots, x_n)$.

Example 1. The formula $\exists y. x \cdot y > 0$ is equivalent to $x \neq 0$; the formula $\exists y. x \cdot y^2 > 0$ is equivalent to $x > 0$; whereas the formula $\forall y. x \cdot y^2 > 0$ is equivalent to `False`.

The existence of a quantifier-free equivalent is known as the *Tarski–Seidenberg Principle* [29,5]. The first constructive solution was given by Tarski [5], but the complexity of his solution was indescribable (in the sense that no elementary function could describe it).

2.2. Cylindrical algebraic decomposition

A better (although doubly exponential) solution had to await the concept of Cylindrical Algebraic Decomposition (CAD) in [6]. We start by defining what is meant by an algebraic decomposition here.

Definition 2.

1. A *cell* from $\mathbb{R}^n$ is a non-empty connected subset of $\mathbb{R}^n$.
2. A *decomposition* of $\mathbb{R}^n$ is a collection $D = \{C_1, \dots, C_h\}$ of finitely many pairwise-disjoint cells from $\mathbb{R}^n$ with $\mathbb{R}^n = \cup_{i=1}^h C_i$.
3. A cell C from $\mathbb{R}^n$ is *algebraic* if it can be described as the solution set of a formula of the form
$$p_1(x_1, \dots, x_n)\sigma_1 0 \ \wedge \ \cdots \ \wedge \ p_k(x_1, \dots, x_n)\sigma_k 0, \tag{2}$$
where the p_i are polynomials with rational coefficients and variables from $x_1, \dots, x_n$, and where the σ_i are taken from $\{=, >, <\}$.[4] Equation (2) is a *defining formula* of C, denoted as $\mathrm{Def}(C)$.
4. A decomposition of $\mathbb{R}^n$ is algebraic if each of its cells is algebraic.
5. A decomposition D of $\mathbb{R}^n$ is *sampled* if it is equipped with a function assigning an explicit point $\mathrm{Sample}(C) \in C$ to each cell $C \in D$.

Example 2.

- $D = \{(-\infty, -1), [-1, -1], (-1, 1), [1, 1], (1, \infty)\}$ is a decomposition of $\mathbb{R}$.
- This D is also algebraic because the cells can be described by $x_1 < -1$, $x_1 = -1$, $x_1 > -1 \wedge x_1 < 1$, $x_1 = 1$ and $x_1 > 1$, respectively, in their order of listing above.
- To get a sampled algebraic decomposition, we additionally provide -2, -1, $\frac{1}{2}$, 1 and 3, respectively.

We are interested in decompositions with certain important properties relative to a set of polynomials as formalised in the next definition.

[3] Any proposition with quantified variables can be converted into this form — see any standard logic text.

[4] Since the constraints in (2) need not be equations a more accurate name would be semi-algebraic. We can avoid $\leq$ and $\geq$, since e.g. $p \leq 0$ is equivalent to $-p > 0$. Avoiding $\neq$ is a more fundamental requirement.

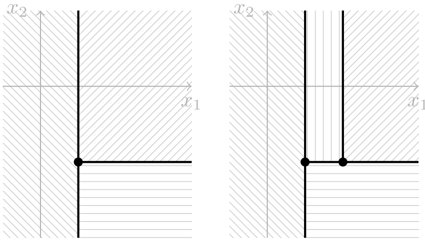


Fig. 1. Illustrations of cylindricity.

Definition 3. A cell C from $\mathbb{R}^n$ is said to be *sign-invariant* for a polynomial $p(x_1, \dots, x_n)$ if and only if precisely one of the following is `True`:

$$(1)\quad \forall \mathbf{x} \in C.\, p(\mathbf{x}) > 0;\ \text{or} \qquad (2)\quad \forall \mathbf{x} \in C.\, p(\mathbf{x}) < 0;\ \text{or} \qquad (3)\quad \forall \mathbf{x} \in C.\, p(\mathbf{x}) = 0.$$

A cell C is sign-invariant for a set of polynomials if and only if it is sign-invariant for each polynomial in the set individually. A decomposition of $\mathbb{R}^n$ is sign-invariant for a polynomial (a set of polynomials) if each of its cells is sign-invariant for the polynomial (the set of polynomials).

For example, the decomposition in Example 2 is sign-invariant for $x_1^2 - 1$.

All of the techniques discussed in this paper to produce decompositions are defined with respect to an ordering on the variables in the formulae.

Definition 4. For positive natural numbers $m < n$, we see $\mathbb{R}^n$ as an extension of $\mathbb{R}^m$ by further dimensions, and denote the coordinates of $\mathbb{R}^m$ as $x_1, \dots, x_m$ and the coordinates of $\mathbb{R}^n$ as $x_1, \dots, x_n$. Unless specified otherwise we assume polynomials in this paper are defined with these variables under the ordering corresponding to their labels, i.e., $x_1 \prec x_2 \prec \cdots \prec x_n$.

We define the *main variable* of a polynomial as the highest variable in the ordering that is present in the polynomial. If we say that a set of polynomials are in $\mathbb{R}^i$ then we mean that they are defined with variables $x_1, \dots, x_i$ and at least one such polynomial has main variable x_i.

We can now define the structure of the cells in our decomposition.

Definition 5. Let $m < n$ be positive natural numbers. A decomposition D of $\mathbb{R}^n$ is said to be *cylindrical* over a decomposition D' of $\mathbb{R}^m$ if the projection onto $\mathbb{R}^m$ of every cell of D is a cell of D'. I.e. the projections of any pair of cells from D are either identical (if the same cell in D') or disjoint (if different ones).

The cells of D which project to $C \in D'$ are said to form the *cylinder* over C. For a sampled decomposition, we also insist that the sample point in C be the projection of the sample points of all the cells in the cylinder over C. A (sampled) decomposition D of $\mathbb{R}^n$ is cylindrical if and only if for each positive natural number $m < n$ there exists a (sampled) decomposition D' of $\mathbb{R}^m$ over which D is cylindrical.

So combining the above definitions we have a (Sampled) Cylindrical Algebraic Decomposition (CAD). Note that cylindricity implies the following structure on a decomposition cell.

Definition 6. A cell from $\mathbb{R}^n$ is *locally cylindrical* if it can be described by conditions $c_1(x_1)$, $c_2(x_1, x_2)$, $\dots$, $c_n(x_1, \dots, x_n)$ where each c_i is one of:

$$\begin{aligned}
\ell_i(x_1, \dots, x_{i-1}) &< x_i, \\
\ell_i(x_1, \dots, x_{i-1}) &< x_i < u_i(x_1, \dots, x_{i-1}), \\
& \phantom{<}\, x_i < u_i(x_1, \dots, x_{i-1}), \\
& \phantom{<}\, x_i = s_i(x_1, \dots, x_{i-1}).
\end{aligned}$$

Here ℓ_i, u_i, s_i are functions in $i - 1$ variables (constants when $i = 1$). These functions can be rational polynomials or indexed root expressions (whose values for given $x_i, \dots, x_n$ might be algebraic numbers).

Example 3. Fig. 1 shows two decompositions of $\mathbb{R}^2$ in which, each dot, each line segment, and each hatched region are cells. The decomposition of $\mathbb{R}^2$ on the left of the figure is cylindrical over $\mathbb{R}^1$ (horizontal axis), but the decomposition on the right is not (two cells have overlapping non-identical projections onto x_1). All cells are locally cylindrical. Note that we are assuming $x_1 \prec x_2$; for $x_2 \prec x_1$ neither of the decompositions would be cylindrical, but still each cell would be locally cylindrical.

Collins' solution to the QE problem in Definition 1 was an algorithm to produce a CAD [6] of $\mathbb{R}^n$, sign-invariant for all the p_j, and thus truth-invariant for F. Hence it is then only necessary to consider the truth or falsity of F at the finite number of sample points and query their algebraic definitions to form G. Furthermore, if Q_i is $\forall$, we require F to be `True` at all sample points, whereas $\exists$ requires the truth of F for at least one sample point. As has been pointed out [23,25], such a CAD is finer than required for QE since as well as answering the QE question asked, it could answer another that involved the same p_j; but possibly different σ_j, and even different Q_i as long as the variables are quantified in the same order.

2.3. Our setting: SMT for real algebra

In this paper we are interested in the subset of QE problems coming from *Satisfiability Modulo Theories* (SMT) solving [30], namely sentences (formulae whose variables are all bound by quantifiers) which use only the existential quantifier. I.e. (1) with $m = n$ and all $Q_i = \exists$.

Traditional SMT-solving aims to decide the truth of such formulae by searching for solutions satisfying the Boolean structure of the problem using a SAT-solver, and concurrently checking the consistency of the corresponding theory constraints. It additionally restricts to the case that F is a pure conjunction of the $p_j \sigma_j 0$; and makes the following additional requirements on any solution:

1. If the answer is `True` (generally referred to as SATisfiable), we need only an explicit point at which it is `True` (rather than a description of all such points);
2. If the answer is `False` (generally referred to as UNSATisfiable), we want some kind of justification that can be used to guide the SAT-solver's search.

So, in SMT solving we are less interested in complete algebraic structures but rather in deciding satisfiability and computing solutions if they exist. Reflecting this interest, Real Algebra is often called Non-Linear Real Arithmetic (NRA) in the SMT community. The use of CAD (and computer algebra tools more generally) in SMT has seen increased interest lately [31,32] and several SMT-solvers make use of these [33,34].

2.4. Relevant prior work

Our contribution is a new approach to adapt CAD technology and theory for this particular problem class. There are three previous works that have also sought to adapt CAD ideas to the SMT context:

2.4.1. Incremental CAD

Incremental CAD is an adaptation of traditional CAD so that it works incrementally, checking the consistency of an increasing number of polynomial constraints, as implemented in the SMT-RAT solver [33,35]. Traditional CAD as formulated by Collins and implemented in Computer Algebra Systems consists of two sequentially combined phases (projection and lifting): it will first perform all projection computations to produce algebraic data, and then use this to construct all the cells (with sample points). The incremental adaptation instead processes one projection operation at a time, and then uses that to derive any additional cells (with their additional samples), before making the next projection computation. For the task of satisfiability checking, this allows for early termination not only for satisfiable problems (if we find a sample that satisfies all sign conditions then we do not need to compute any remaining projections) but also for unsatisfiable ones (if the CAD for a subset of the input constraints is computed and no sample from its cells satisfies all those sign conditions). Although the implementation is technically involved, the underlying theory is traditional CAD and in the case of UNSAT that is exactly what is produced.

2.4.2. NLSAT

The NLSAT algorithm by Jovanović and de Moura [36] lifts the theory search to the level of the Boolean search, where the search at the Boolean and theory levels are mutually guided by each other away from unsatisfiable regions when it can be determined by some kind of propagation or lookahead. Partial solution candidates for the Boolean structure and for the corresponding theory constraints are constructed incrementally in parallel until either a full solution is found or a conflict is detected, meaning that the candidates cannot be extended to full solutions. Boolean conflicts are generalised using propositional resolution. For theory conflicts, CAD technology is used to generalise the conflict to an unsatisfying region (a CAD cell). These generalisations are learnt by adding new clauses that exclude similar situations from further search by the above-mentioned propagation mechanisms. In the case of a theory conflict, the incremental construction of theory solution candidates allows to identify a minimal set of constraints that are inconsistent under the current partial assignment. This minimal constraint set induces a coarser state space decomposition and thus in general larger cells that can be used to generalise conflicts and exclude from further search by learning. The exclusion of such a cell is learnt by adding a new clause expressing that the constraints are not compatible with the (algebraic description of the) unsatisfying cell. This clause will lead the search away from the given cell, not only locally but for the whole future search when the constraints are all `True`.

2.4.3. Non-uniform CAD

The non-uniform CAD (NuCAD) algorithm by Brown [27] takes as input a set of polynomial constraints and computes a decomposition whose cells are *truth-invariant* for the *conjunction* of all input constraints. It starts with the coarsest decomposition, having the whole space as one cell, which does not guarantee any truth invariance yet. We split it to smaller cells that are sign-invariant for the polynomial of one of the input constraints, and mark all refined cells that violate the chosen constraint as final: they violate the conjunction and are thus truth-invariant for it. Each refinement is made by choosing a sample in a non-final cell and generalising to a locally cylindrical cell. At any time all cells are locally cylindrical, but there is no global cylindricity condition. The algorithm proceeds with iterative refinement, until all cells are marked as final. Each refinement will be made with respect to one constraint for which the given cell is not yet sign-invariant. There are two kinds of cells in a final NuCAD: cells that violate one of the input constraints (but these cells are not necessarily sign- nor truth-invariant for all other constraints), and cells that satisfy all constraints (and are sign-invariant for all of them). The resulting decomposition is in general coarser (i.e. it has less cells) than a regular CAD. Its cells are neither sign- nor truth-invariant for individual constraints, but they are truth-invariant for their conjunction and that is sufficient for consistency checking.

3. Intuitive idea behind our new algorithm

3.1. From sample to cell in increasing dimensions

We want to check the consistency of a set of input polynomial sign constraints, i.e., the satisfiability of their conjunction. Traditional CAD first generates algebraic information on the formula we study (the projection polynomials) and then uses these to construct a set of sample points, each of which represents a cell in the decomposition.[5] Our new approach works the other way around: we will select (or guess) a sample point by fixing it dimension-wise, starting with the lowest dimension and iteratively moving up in order to extend lower-dimensional samples to higher-dimensional ones. Thus we start with a zero-dimensional sample and iteratively explore new dimensions by recursively executing the following:

- Given an $(i-1)$-dimensional sample $s = (s_1, \dots, s_{i-1})$ that does not evaluate any input constraint to `False`, we extend it to a sample $(s_1, \dots, s_{i-1}, s_i)$, which we denote as $s \times s_i$.
- If this i-dimensional sample can be extended to a satisfying solution, either by recursing or because it already has full dimension, then we terminate, reporting consistency (and this witness point).
- Otherwise we take note of the reason (data on the conflicting requirements that explains why the sample can not be extended) and exclude from further search not just this particular sample $s \times s_i$, but all extensions of s into the ith dimension with any value from a (hopefully large) interval I around s_i which is unsatisfiable for the same reason.

This means that for future exploration, the algorithm is guided to look somewhere away from the reasons of previous conflicts. This should allow us to find a satisfying sample quicker, or in the case of UNSAT build a covering of all $\mathbb{R}^n$ such that we can conclude UNSAT everywhere with fewer cells than a traditional CAD. In the last item above, the generalisation of an unsatisfying sample $s \times s_i$ to an unsatisfying interval $s \times I$ will use a constraint $p(x_1, \dots, x_i)\sigma 0$ that is violated by the sample, i.e., $p(s \times s_i)\sigma 0$ does not hold. It might be the case that s_i is a real zero of $p(s)$ and we then have $I = [s_i, s_i]$. Otherwise we get an interval $I = (\ell, u)$ with $\ell < s_i < u$ where ℓ is either the largest real root of p below s_i or $-\infty$ if no such root exists (and analogously u is either the smallest real root above s_i or ∞). Thus we have $p(s \times r) \neq 0$ for all $r \in (\ell, u)$. Since $p(s \times s_i)\sigma 0$ is `False` and the sign of a polynomial does not change between neighbouring zeros, we can safely conclude that $p(x_1, \dots, x_i)\sigma 0$ is violated by all samples $(s \times r)$ with $r \in I$.

We continue and check further extensions of $s = (s_1, \dots, s_{i-1})$, until either we find a solution or the ith dimension is fully covered by excluding intervals. In the latter case, we take the collection of intervals produced and use CAD projection theory to rule out not just the original sample in $\mathbb{R}^{i-1}$ but an interval around it within dimension $(i-1)$, i.e. the same procedure in the lower dimension. Intuitively, each sample $s \times s_i$ violating a constraint with polynomial p can be generalised to a cell in a p-sign-invariant CAD. So when all extensions of s have been excluded (the ith dimension is fully covered by excluding intervals) then we project all the covering cells to dimension $i-1$ and exclude their intersection from further search.

3.2. Restoring cylindricity

If we were to generalise violating samples to cells in a CAD that is sign-invariant for *all* input constraints then in the case of unsatisfiability we would explore a CAD structure. But instead, by *guessing* samples in not yet explored areas and identifying violated constraints individually per sample, we are able to generalise samples to larger cells that can be excluded from further search: like in the NLSAT approach.

Unlike NLSAT, we then build the *intersection* creating a cylindrical arrangement at the cost of making the generalisations smaller (but as large as possible while still ordered cylindrically). What is the advantage gained from a cylindrical ordering?

[5] An incremental CAD approach would incrementally create new projection polynomials and new sample points.

In NLSAT the excluded cells are not cylindrically ordered; and so SMT-mechanisms are used in the Boolean solver (like clause learning and propagation) to lead the search away from previously excluded cells. In contrast, our aim is to make this book-keeping remain inside the algebraic procedure, which can be done in a depth-first-search approach when the cells are cylindrically ordered.

3.3. Cylindrical algebraic coverings

So, we maintain cylindricity from CAD, but we relax the disjointness condition on cells in a decomposition, allowing our cells to *overlap* as long as their cylindrical ordering is still maintained. Instead of decomposition we will use the name *covering* for these structures.

Definition 7.

1. A *covering* of $\mathbb{R}^n$ is a collection $D = \{C_1, \dots, C_h\}$ of finitely many (not necessarily pairwise-disjoint) cells from $\mathbb{R}^n$ with $\mathbb{R}^n = \cup_{i=1}^h C_i$.
2. A covering of $\mathbb{R}^n$ is *algebraic* if each of its cells is algebraic.
3. A covering D of $\mathbb{R}^n$ is *sampled* if it is equipped with a function assigning an explicit point $\mathrm{Sample}(C) \in C$ to each cell $C \in D$.
4. A cell C from $\mathbb{R}^n$ is *UNSAT* for a polynomial constraint (set of constraints) if and only if all points in C evaluate the constraint (at least one of the constraints) to `False`. A covering of $\mathbb{R}^n$ is UNSAT for a constraint (set) if each of its cells is UNSAT for the constraint (at least one from the set).
5. A covering D of $\mathbb{R}^n$ is said to be *cylindrical* over a covering D' of $\mathbb{R}^m$ if the projection onto $\mathbb{R}^m$ of every cell of D is a cell of D'. The cells of D which project to $C \in D'$ form the *cylinder* over C. For a sampled covering, the sample point in C needs to be the projection of the sample points of all the cells in the cylinder over C. A (sampled) covering D of $\mathbb{R}^n$ is cylindrical if and only if for each $0 < m < n$ there exists a (sampled) covering D' of $\mathbb{R}^m$ over which D is cylindrical.

The coverings produced in our algorithm are all UNSAT coverings for constraint sets (i.e. at least one constraint is unsatisfied on every cell).

3.4. Differences to existing methods

Our new method shares and combines ideas from the related work in Section 2.4 but differs from each.

- A version of the CAD method which proceeds incrementally by constraint or projection factor is implemented in the SMT-RAT solver. Our new method differs as even in the case of unsatisfiability it will not need to compute a full CAD but rather a smaller number of potentially overlapping cells.
- While its learning mechanism made NLSAT the currently most successful SMT solution for Real Algebra, it brings new scalability problems by the large number of learnt clauses. Our method is conflict-driven like NLSAT, but instead of learning clauses, we embed the learning in the algebraic procedure. Learning clauses allows to exclude unsatisfiable CAD cells for certain combinations of polynomial constraints for the whole future search, but it also brings additional costs for maintaining the truth of these clauses. Our approach remembers the unsatisfying nature of cells only for the current search path, and learns at the Boolean level only unsatisfiable combinations of constraints. To unite the advantages from both worlds, our approach could be extended to a hybrid setting where we learn at both levels (by returning information on selected cells to be learned as clauses).
- Like NuCAD, our algorithm can compute refinements according to different polynomials in different areas of the state space and to stop the refinement if any constraint is violated, however we retain the global cylindricity of the decomposition. Furthermore, driven by model construction, we can identify minimal sets of relevant constraints that we use for cell refinement, instead of the arbitrary choice of these polynomials used by NuCAD. Our expectation is thus that on average this will lead to coarser decompositions, and certainly rule out some unnecessary worst case decompositions.

4. The new algorithm

4.1. Interface, I/O, and data structure

Our main algorithm, `get_unsat_cover`, is presented as Algorithm 2, while Algorithm 1 provides the user interface to it. A user is expected to define the set of constraints whose satisfiability we want to study globally, and then make a call to Algorithm 1 with no input. This will then call the main algorithm with an empty tuple as the initial sample s; the main algorithm is recursive and will later call itself with non-empty input. In these recursive calls the input is a partial sample point $s \in \mathbb{R}^{i-1}$ which does not evaluate any global constraint defined over $\mathbb{R}^{i-1}$ to `False`, and for which we want to explore dimension i and above.

Algorithm 1: `user_call()`.

```
   Data: Global set $C$ of polynomial constraints defined over $\mathbb{R}^n$.
   Input : None
   Output: Either (SAT, $S$) where $S \in \mathbb{R}^n$ is a full-dimensional satisfying witness;
           or (UNSAT, $\overline{C}$) when no such $S$ exists, where $\overline{C} \subset C$ is also unsatisfiable.
1  ($flag$, $data$) := get_unsat_cover( () )                                 // Algorithm 2
2  if $flag$ = SAT then
3      return ($flag$, $data$)
4  else
5      return ($flag$, infeasible_subset($data$))
```

Algorithm 2: `get_unsat_cover(s)`.

```
    Data: Global set $C$ of polynomial constraints defined over $\mathbb{R}^n$.
    Input : Sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$ such that no global constraint evaluated at $s$ is False. Note that if $s = ()$ then $i = 1$ (i.e. we start
            from the first dimension).
    Output: Either (SAT, $S$) where $S \in \mathbb{R}^n$ is a full-dimensional satisfying witness;
            or (UNSAT, $\mathbb{I}$) when $s$ cannot be extended to a satisfying sample in $\mathbb{R}^n$. $\mathbb{I}$ represents a set
            of intervals which cover $s \times \mathbb{R}$ and come with algebraic information (see Section 4.1).
1   $\mathbb{I}$ := get_unsat_intervals(s)                                      // Algorithm 3
2   while $\bigcup_{I \in \mathbb{I}} I \neq \mathbb{R}$ do
3       $s_i$ := sample_outside($\mathbb{I}$)
4       if $i = n$ then
5           return (SAT, $(s_1, \ldots, s_{i-1}, s_i)$)
6       $(f, O)$ := get_unsat_cover($(s_1, \ldots, s_{i-1}, s_i)$)                    // recursive call
7       if $f$ = SAT then                                              // then $O$ is a satisfying sample
8           return (SAT, $O$)                                          // pass answer up to main call
9       else if $f$ = UNSAT then                                       // then $O$ is an unsat covering
10          $R$ := construct_characterization($(s_1, \ldots, s_{i-1}, s_i)$, $O$)       // Algorithm 4
11          $I$ := interval_from_characterization($(s_1, \ldots, s_{i-1})$, $s_i$, $R$)  // Algorithm 5
12          $\mathbb{I} := \mathbb{I} \cup \{I\}$
13  return (UNSAT, $\mathbb{I}$)
```

The main algorithm provides two outputs, starting with a flag. When the flag is SAT then the partial sample s was extended to a full sample from $\mathbb{R}^n$ (the second output) which satisfies the global constraints. When the flag is UNSAT then the method has explored the higher dimensions and determined that the sample cannot be extended to a satisfying solution. It does this by computing an *UNSAT cylindrical algebraic covering* for the constraints with the partial sample s substituted for the first $i-1$ variables. Information on the covering, and the projections of these cells, are all stored in the second output in this case.

More formally, the output $\mathbb{I}$ is a set of objects I each of which represent an interval of the real line (specifically $s \times \mathbb{R}$). We use I later to mean both the interval, and our data structure encoding it which carries additional algebraic information. In total such a data structure I has six attributes, starting with:

- the lower bound ℓ;
- the upper bound u;
- a set of polynomials L that defined ℓ;
- a set of polynomials U that defined u.

The bounds are constant, but potentially algebraic, numbers. The polynomials define them in that they are multivariate polynomials which when evaluated at s became univariate with the bound as a real root. The final two attributes are also sets of polynomials:

- a set of polynomials P_i (multivariate with main variable x_i);
- a set of polynomials $P_\perp$ (multivariate with main variable smaller than x_i).

These polynomials have the property that allows for generalisation of s to an interval: the property is that perturbations of s which do not change the signs of these polynomials will result in the interval I (whose numerical bounds may have also perturbed but will still be defined by the same ordered real roots of the same polynomials) remaining a region of unsatisfiability.

Within a covering there must also be special intervals which run to ∞ and $-\infty$. For intervals with these bounds we store the polynomials from the constraints which allowed us to conclude the infinite interval.

In the case of UNSAT, the user algorithm will have to process the data $\mathbb{I}$ into an *infeasible subset* (Algorithm 1 Line 5), i.e. a subset of the constraints that are still unsatisfiable. Ideally this would be minimal (a minimal infeasible subset) although

Algorithm 3: `get_unsat_intervals(s)`.

```
    Data: Global set $C$ of polynomial constraints defined over $\mathbb{R}^n$.
    Input : Sample point $s$ of dimension $i-1$ with $i$ a positive integer ($s$ is empty for $i=1$).
    Output: $\mathbb{I}$ which represents a set of unsatisfiable intervals over $s \times I$ and comes with some algebraic information (see Section 4.1).
1   $\mathbb{I} := \emptyset$
2   $C_i :=$ set of all constraints from $C$ with main variable $x_i$
3   foreach constraint $c = p\sigma 0$ in $C_i$ do                    // I.e. $p$ is the defining polynomial of $c$
4       $c' := c(s)$                                                  // Substitute $s$ into $c$ to leave univariate
5       if $c' = False$ then
6           Define $I$ with $I_\ell = -\infty, I_u = \infty, I_L = \emptyset, I_U = \emptyset, I_{P_i} = \{p\}, I_{P_\perp} = \emptyset$
7           return $\{I\}$
8       if $c' = True$ then
9           continue to next constraint
        // at this point $c'$ is univariate in the $i$th variable
10      $Z :=$ real_roots$(p, s)$                                     // $Z = [z_1, \ldots, z_k]$
11      $Regions := \{(-\infty, z_1), [z_1, z_1], (z_1, z_2), \ldots, (z_k, \infty)\}$   // $\{(-\infty, +\infty)\}$ if $Z$ was empty
12      foreach $I \in Regions$ do
13          Let $\ell$ and $u$ be the lower and upper bounds of $I$
14          Pick $r \in I$                                            // e.g. $r := \ell + (u-\ell)/2$
15          if $c'(r) = False$ then
16              Set $L, U$ each to $\emptyset$
17              if $\ell \neq -\infty$ then $L := \{p\}$
18              if $u \neq \infty$ then $U := \{p\}$
19              Define new interval $I$ with $I_\ell = \ell, I_u = u, I_L = L, I_U = U, I_{P_i} = \{p\}, I_{P_\perp} = \emptyset$
                // Polynomials stored here undergo simplification first − see Section 4.4.3
20              Add $I$ to $\mathbb{I}$
21  return $\mathbb{I}$
```

any reduction of the full set would carry benefits. We discuss how we implement this later in Section 4.6. We note that the correctness of Algorithm 1 follows directly from the correctness of its sub-algorithms.

4.2. Initial constraint processing

The first task in Algorithm 2 is to see what effect the partial sample s has on the global constraints. This is described as Algorithm 3, which will produce those intervals $I \subseteq \mathbb{R}$ such that $s \times I$ is conflicting with some input constraints (a partial covering). This method resembles a CAD lifting phase where we substitute a sample point into a polynomial to render it univariate, compute the real roots, and decompose the real line into sign-invariant regions. Here we do the same for the truth of our input constraints.

Algorithm 3 Lines 5–9 deal with the case where after substitution the truth value of the constraint may be immediately determined (e.g. the defining polynomial evaluated to a constant). The constraint either provides the entire line as an UNSAT interval, or no portion of it. The rest of the code deals with the case where the substitution rendered a constraint univariate in the ith variable. We use `real_roots(p, s)` to return all real roots of a polynomial p over a partial sample point s that sets all but one of its variables. We do not specify the details of the real root isolation algorithm here[6] but note that it will need to handle potentially algebraic coefficients. We assume roots are returned in ascending numerical order with any multiple roots represented as a single list entry.

The inner for loop queries a sample point in each region of the corresponding decomposition of the line to determine any infeasible regions for the constraint, storing them in the output data structure $\mathbb{I}$. $\mathbb{I}$ represents a set of intervals I such that $s \times I$ conflicts with some input constraint. The intervals from $\mathbb{I}$ may be overlapping, and some may be redundant (i.e. fully contained in others). We discuss this issue of redundancy further in Section 4.5.

It is clear that Algorithm 3 will meet its specification, in that it will define intervals on which constraints are unsatisfiable: the falsity of a constraint caused the inclusion of an interval in the output while the bounds of the interval were defined to ensure that the polynomial defining the failing constraint would not change sign inside. The role and property of the stored algebraic information will be discussed in Section 4.4.

The call to Algorithm 3 initialises $\mathbb{I}$ in Algorithm 2 in which we will build our UNSAT covering. It is unlikely but possible that $\mathbb{I}$ already covers $\mathbb{R}$ after the call to Algorithm 3: it could even contain $(-\infty, \infty)$. If $\mathbb{I}$ is already a covering then we would skip the main loop of Algorithm 2 and directly return it. For example, this would be triggered by either of the constraints $y^2x < 0$ or $y^2 + x < 0$ at sample $s = (x \mapsto 0)$. The former would have been returned early by Line 7 of Algorithm 3 while the latter would have required real root isolation and have been returned in Line 21 of Algorithm 3.

[6] Our implementation in SMT-RAT uses bisection with Descartes' rule of signs.

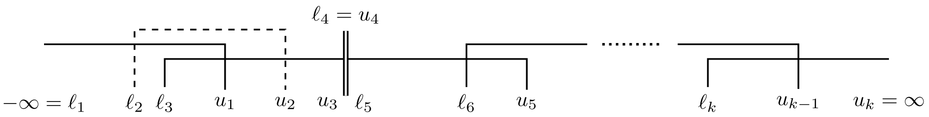


Fig. 2. An example of an UNSAT covering.

4.3. The main loop of Algorithm 2

We will iterate through Lines 3–12 of Algorithm 2 until the set of intervals represented by $\mathbb{I}$ cover all $\mathbb{R}$. In each iteration we collect additional intervals for our UNSAT covering.

To do this we first generate a sample point s_i from $\mathbb{R} \setminus (\cup_{I\in\mathbb{I}} I)$ using a subroutine `sample_outside(`$\mathbb{I}$`)` which is left unspecified. This could simply pick the mid-point in any current gap, or perhaps something more sophisticated (a common strategy would be to prefer integers, or at least rationals).

Note that $s \times s_i$ necessarily satisfies all those constraints with main variable x_i, otherwise we would have generated an interval excluding s_i at Line 1, as well as all constraints with smaller main variables (from the input specification on s). This means that: (a) if $s \times s_i$ has full dimension, we have found a satisfying sample point for the whole set of constraints and can return this along with the SAT flag in Line 5; and (b) if not full dimension then we will meet the input specification for the recursive call on Line 6. The recursion means we will explore $s \times s_i$ in the next dimension up. The previous check on dimension acts as the base case and thus the recursion is clearly bounded in depth, ensuring termination if the main loop always terminates.

If the result of the recursive call is SAT we simply pass on the result in Line 8. Otherwise we have an UNSAT covering for $s \times s_i$ and our next task is to see whether we can generalise this knowledge to rule out not just s_i, but an interval around it. We do this in two steps.

- First in Line 10 we call Algorithm 4 to construct a *characterisation* from the UNSAT covering: a set of polynomials which were used to determine unsatisfiability and with the property that on the region around $s \times s_i$ where none of them change sign the reasons for unsatisfiability generalise. I.e., while the exact bounds of the intervals in the coverings may vary: (a) they are still defined by the same ordered roots of the same polynomials (over the sample); and (b) they do not move to the extent that the line is no longer covered.
- Second in Line 11 we call Algorithm 5 to find the interval in dimension i over s in which those characterisation polynomials are sign-invariant.

We describe these two sub-algorithms in detail in the next subsection.

The interval produced by Algorithm 5 may be $(-\infty, \infty)$. In this case all other intervals in $\mathbb{I}$ are now redundant and the main loop of Algorithm 2 stops. Otherwise we continue to iterate.

The loop will terminate because although the search space is infinite the combinations of constraints is not. Each generalisation rules out a portion of space defined by a set of polynomials all having invariant sign on it. The number of polynomials computed is finite and their changes in sign are finite. Thus eventually we must either cover all the line or sample in a satisfying region. This termination argument is very similar to traditional CAD but here we aim to compute fewer, larger overlapping regions.

The correctness of Algorithm 2 thus depends on the correctness of these two crucial sub-algorithms.

4.4. Generalising the UNSAT covering from the sample

It remains to examine the details of how the UNSAT covering is generalised from the sample s to an interval around it (the calls to Algorithms 4 and 5 on Lines 10 and 11 of Algorithm 2).

4.4.1. Ordering within a covering

The input to Algorithm 4 is an UNSAT covering $\mathbb{I}$ whose elements define intervals I which together cover $\mathbb{R}$. For an example of such a covering see Fig. 2. There may be some redundancy here, like the second interval (from ℓ_2 to u_2) in Fig. 2 which is completely covered by the first and third intervals already. To ensure soundness of our approach we need to remove at least those intervals which are included within a single other interval. We discuss more details on dealing with redundancies in coverings in Section 4.5.

For now we simply make the reasonable assumption of the existence of an algorithm `compute_cover` which computes such a *good covering* of the real line as a subset of an existing covering $\mathbb{I}$. Since it is not crucial we will not specify the algorithm here, but we note that the ideas in [37] may be useful.

We assume that `compute_cover` orders the intervals in its output according to the following total ordering:

$$(\ell_1, u_1) \leq (\ell_2, u_2) \qquad \Leftrightarrow \qquad \ell_1 \leq \ell_2 \wedge (\ell_1 < \ell_2 \vee u_1 \leq u_2)\,, \qquad (3)$$

i.e. ordered on ℓ_i with ties broken by u_i. We will always have $\ell_1 = -\infty$ and $u_k = \infty$ with the remaining bounds defined as algebraic numbers (possibly not rational). We further require that

$$(\ell_1, u_1) \leq (\ell_2, u_2) \qquad \Leftrightarrow \qquad \ell_1 \leq \ell_2 \wedge u_1 \leq u_2\,, \tag{4}$$

only possible if we exclude the cases where one interval is a subset of another. Note that this is not just an optimisation but is actually crucial for the correctness of this approach, as explained in Section 4.5.

The intervals we consider are either open (if $\ell \neq u$) or point intervals (if $\ell = u$). Note that it might make sense to extend the presented algorithm to allow for closed (or half-open) intervals as well, for example when built from weak inequalities. This could help to avoid some work on individual sample points like ℓ_4 in Fig. 2 and the point intervals we deal with in the examples in Section 5. Such changes are straight-forward to implement and so we do not discuss them here to simplify the presentation.

4.4.2. Constructing the characterisation

The first line of Algorithm 4 ensures the ordering specified above, while the remainder uses CAD projection ideas to collect everything we need to ensure that the UNSAT covering $\mathbb{I}$ stays valid when we generalise the underlying sample point s_i later on. We include polynomials for a variety of reasons.

- First in Line 5 we pass along any polynomials with a lower main variable that had already been stored in the data structure. These were essentially produced by the following steps at any earlier recursion, by an act of projection that skipped a dimension. For example, the projection of any polynomial $f(x_1, x_3)$ into (x_1, x_2)-space will actually give a univariate polynomial in x_1.
- Next in Lines 6 and 7 we identify polynomials that will ensure the existence of the current lower and upper bounds. We add discriminants, whose zeros indicate where the original polynomial has multiple roots, and leading coefficients (with respect to the main variable), whose zeros indicate asymptotes of the original polynomial. We may also require additional coefficients, as discussed in Section 4.4.5. If we ensure these polynomials do not change sign then we know that the algebraic varieties that defined ℓ and u continue to exist (and no other varieties are spawned).
- In Lines 8 and 9 we generate polynomials whose sign-invariance ensures that ℓ and u stay the *closest* bounds. I.e. we avoid the situation where they are undercut by those coming from some other variety. We need only concern ourselves with those coming from the direction of the bound. For example, when protecting an upper bound we need only worry about roots that are above it and take resultants accordingly on Line 9. This is because any root coming from below would need to first pass through the lower bound and the resultant from Line 8 would block generalisation past that point.
- In Lines 10–11 we finally derive resultants to ensure that the overlapping lower and upper bounds of adjacent intervals do not cross, which would *disconnect* the UNSAT covering (leaving it not covering some portion of the line). The correctness of this step requires an ordering with lack of redundancy, as specified above and discussed in detail in Section 4.5. This step also has the effect of ensuring the intervals do not overlap further to the extent that one then becomes redundant.

We note that the projection polynomials we collect are a subset of those collected by the McCallum projection operator for a full CAD [38]. That operator would take the leading coefficient and discriminant of every polynomial involved, and all of the possible cross resultants. Here we take only those relevant to the reasons for unsatisfiability. Note that we have not taken the resultant of the polynomials that define the lower bound of an interval with those defining the upper bound of the same interval. We explain why these are not necessary in Section 4.5.3 after discussing in detail the issue of interval redundancy.

Recall that we may have satisfiability over s refuted by a single constraint in Algorithm 3 (i.e. the defining polynomial cannot change sign over s). In that case, after Line 1 of Algorithm 4 the data structure $\mathbb{I}$ contains a single interval $(-\infty, \infty)$ and we would have no resultants to compute.

4.4.3. Simplification and bases of polynomials

Algorithm 4 finishes in Line 12 with some standard CAD simplifications to the polynomials. These all stem from the fact that polynomials matter only so much as where they vanish. E.g.:

- Remove any constants, or other polynomials than can easily be concluded to never equal zero.
- Normalise the remaining polynomials to avoid storing multiple polynomials which define the same varieties. E.g. multiply each polynomial by the constant needed to make it monic (i.e. divide by leading coefficient so for example polynomial $2x - 1$ becomes $x - \frac{1}{2}$). Other normal forms include the primitive positive integral with respect to the main variable.
- Store a square free basis for the factors rather than the polynomials themselves (this much is necessary, else later resultants/discriminants will vanish); or even fully factorising (optional, but generally favoured for the efficiency gains it can bring).

Algorithm 4: `construct_characterization(`$s, \mathbb{I}$`)`.

```
   Input  : Sample point $s \in \mathbb{R}^i$ and data structure $\mathbb{I}$ describing UNSAT covering over $s$ in dim. $i+1$.
   Output: A set of polynomials $R \subseteq \mathbb{Q}[x_1, \ldots, x_i]$ that characterises a region around $s$ that is already unsatisfiable for the same reasons.
 1 $\mathbb{I}$ := compute_cover($\mathbb{I}$)                                         // See Section 4.4.1
 2 $R := \emptyset$
 3 foreach $I \in \mathbb{I}$ do
 4     Extract $\ell = I_\ell, u = I_u, L = I_L, U = I_U, P_{i+1} = I_{P_{i+1}}, P_\perp = I_{P_\perp}$
 5     $R := R \cup P_\perp$
 6     $R := R \cup \mathtt{disc}(P_{i+1})$
 7     $R := R \cup \{\mathtt{required\_coefficients}(p) \mid p \in P_{i+1}\}$
 8     $R := R \cup \{\mathtt{res}(p, q) \mid p \in L, q \in P_{i+1}, q(s \times \alpha) = 0 \text{ for some } \alpha \leq l\}$
 9     $R := R \cup \{\mathtt{res}(p, q) \mid p \in U, q \in P_{i+1}, q(s \times \alpha) = 0 \text{ for some } \alpha \geq u\}$
10 for $j \in \{1, \ldots, |\mathbb{I}| - 1\}$ do
11     $R := R \cup \{\mathtt{res}(p, q) \mid p \in U_j, q \in L_{j+1}\}$
12 Perform standard CAD simplifications to $R$
13 return $R$
```

We note that the original constraints are stored as presented for their analysis by Algorithm 3 but that when we store their defining polynomials in $\mathbb{I}$, we are actually storing the simplified bases of these polynomials described here. Thus line 19 in Algorithm 3 from earlier is actually simplification as well as storage.

4.4.4. *Constructing the generalisation*

Now let us discuss how this characterisation (set of polynomials) is used to expand the sample to an interval by Algorithm 5. We first separate the polynomials into P_i and $P_\perp$ where P_i are those polynomials that contain x_i and $P_\perp$ the rest. We then identify the crucial points over s beyond which our covering may cease to be. This step essentially evaluates the polynomials with main variable x_i over the sample in $\mathbb{R}^{i-1}$ and calculates real roots of the resulting univariate polynomial. There is some additional work within this sub-algorithm call on Line 3 which we discuss in Section 4.4.5.

We then construct the interval around s_i from the closest roots ℓ and u and collect the polynomials that vanish in ℓ and u in the sets L and U, respectively. We supplemented the real roots with $\pm\infty$ to ensure that ℓ and u always exist, i.e. we can have $(-\infty, u)$ or (ℓ, ∞). In this case the corresponding set L or U is empty.

Algorithm 5: `interval_from_characterization(`s, s_i, P`)`.

```
   Input  : Sample point $s \in \mathbb{R}^{i-1}$; an extension $s_i$ to the sample; and set of polynomials $P$ in $\mathbb{Q}[x_1, \ldots, x_i]$ that characterise why $s \times s_i$ cannot be
            extended to a satisfying witness.
   Output: An interval $I$ around $s_i$ such that on $s \times I$ the constraints are unsatisfiable for the same reasons as on $s \times s_i$.
 1 $P_\perp := \{p \in P \mid p \in \mathbb{Q}[x_1, \ldots, x_{i-1}]\}$
 2 $P_i := P \setminus P_\perp$
 3 $Z := \{-\infty\} \cup$ real_roots_with_check($P_i, s$) $\cup \{\infty\}$
 4 $l := \max\{z \in Z \mid z \leq s_i\}$
 5 $u := \min\{z \in Z \mid z \geq s_i\}$
 6 $L := \{p \in P_i \mid p(s \times l) = 0\}$
 7 $U := \{p \in P_i \mid p(s \times u) = 0\}$
 8 Define new interval $I$ with $I_\ell = \ell, I_u = u, I_L = L, I_U = U, I_{P_i} = P_i, I_{P_\perp} = P_\perp$
 9 return $I$
```

In the case where P_i has no real roots at all over s then the UNSAT covering is valid unconditionally over s, and the interval $(-\infty, \infty)$ is formed and passed back to the main algorithm (where it could be taken to form the next covering in its entirety).

Let us briefly consider some simple examples for Algorithm 5. Suppose we have variable ordering $x \prec y$, the partial sample $(x \mapsto 0)$ and that P contains only the polynomial whose graph defines the unit circle: $x^2 + y^2 - 1$. Then Line 3 forms the set $\{-\infty, -1, 1, \infty\}$. If s_i had been the extension $(y \mapsto 0)$ then we would select $\ell = -1, u = 1$, i.e. generalise to the y-axis inside the circle. Similarly, if the extension had been $(y \mapsto 2)$ we would select $\ell = 1, u = \infty$, i.e. generalise to the whole y axis above the circle. Finally, consider what would happen if s_i had been the extension $(y \mapsto 1)$. In that case, since s_i is one of the roots in Z it would be selected for both ℓ and u. I.e. in that case no generalisation of s_i is possible.

4.4.5. *Correctness of the generalisation*

The correctness of Algorithms 4 and 5 relies on CAD theory via McCallum projection, with Algorithm 4 an analogue of projection and Algorithm 5 an analogue of lifting. However, there are a number of subtleties.

First, regarding exactly which coefficients are computed in Algorithm 4 Line 7. As explained above, the leading coefficient is essential to include as its vanishing would indicate an asymptote. However, we must also consider what happens within regions of such vanishing: there the polynomial has reduced in degree, and so a lesser coefficient is now leading and must be recorded for similar reasons. We should take successive coefficients until one is taken that can be easily concluded not to vanish (e.g. it is constant) or if it may be concluded that the included coefficients cannot vanish simultaneously. In our context this is simpler: so long as a coefficient evaluates at the sample point to something non-zero then we need not include any subsequent coefficients (since by including that coefficient in the characterisation we ensure we do not generalise to where it is zero). We do need to include preceding coefficients that were zero as we must ensure that the generalisation does not cause them to reappear. This is described by Algorithm 6.

Algorithm 6: `required_coefficients`(s, P_{i+1}).

```
Input  : Sample point $s \in \mathbb{R}^i$ and set of polynomials $P_{i+1}$ each with main variable $x_{i+1}$.
Output: A set of polynomials in $\mathbb{Q}[x_1, \dots, x_i]$ consisting of those coefficients of $p$ required for the characterisation around $s$.
1 $C = \emptyset$
2 foreach $p \in P_{i+1}$ do
3     while $p \neq 0$ do
4         $lc :=$ lcoeff$(p)$
5         $C := C \cup \{lc\}$
6         if $lc$ evaluated at $s$ is non zero then
7             Break the while loop
8         $p := p -$ lterm$(p)$
9 return $C$
```

With such calculation of coefficients our characterisations are then subsets of the McCallum projection [38]: we ensure that individual polynomials are delineable but we cannot claim that for the characterisation as a set since we differ from [38] in which resultants are computed. The inclusion of resultants in CAD projection is to ensure a constant structure of intersections between varieties in each cell. McCallum projection takes all possible resultants between polynomials involved. Instead, we take only those needed to maintain the structure of our covering, as detailed by the arguments in Section 4.4.2.

4.4.6. *Completeness of the generalisation*

McCallum projection is not *complete*, i.e. there are some (statistically rare) cases where its use is known to be invalid, and these could be inherited in our algorithm. The problem can occur when a polynomial vanishes at a sample of lower dimension (known as *nullification*) potentially losing critical information. For example, consider a polynomial $zy - x$ which vanishes at the lower dimensional sample $(x, y) = (0, 0)$. The polynomials' behaviour clearly changes with z but that information would be lost.

Such nullifications can be easily identified when they occur. We assume that the sub-algorithm used in Algorithm 5 Line 3 will inform the user of such nullification. What should be done in this case? An extreme option would be to recompute the characterisation to include entire subresultant sequences as in Collins' projection [6], or to use the operator of Hong [39]. A recent breakthrough in CAD theory could offer a better option: [40] proved that Lazard projection [41] is complete.[7] The Lazard operator includes leading and trailing coefficients, and requires more nuanced lifting computations. Instead of evaluating polynomials at a sample and calculating real roots of the resulting univariate polynomials, we must instead perform a Lazard evaluation [40] of the polynomial at the point. This will substitute the sample coordinate by coordinate, and in the event of nullification divide out the vanishing factor to allow the substitution to continue. Thus no roots are lost through nullification.

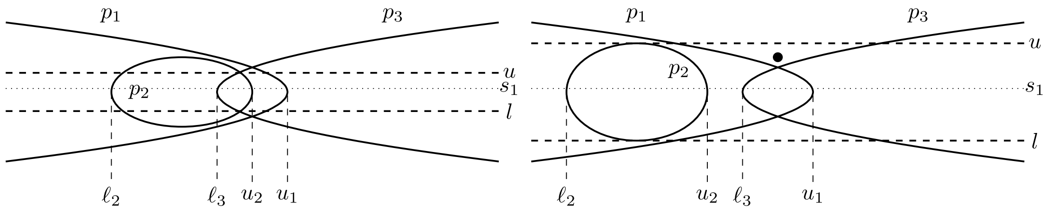
We have not yet adapted our algorithm to Lazard theory: although SMT-RAT already has an implementation of Lazard evaluation, we are not yet clear on how the polynomial identification in Algorithm 5 should be adapted in cases of nullification. Also, it is not trivial to argue the safe exclusion of trailing coefficients in cases where the leading coefficient is constant, as it is with McCallum, since the underlying Lazard delineability relies heavily on properties of the trailing coefficient. Our current implementation is hence technically incomplete, however, we can produce warnings for such cases. The experiments on the SMT-LIB detailed in Section 6 show that these nullifications are a very rare occurrence.

4.5. *Redundancy of intervals*

We discussed in Section 4.4.1 that the set of intervals in a covering may contain redundancies. We distinguish between two possibilities for how an interval I can be redundant:

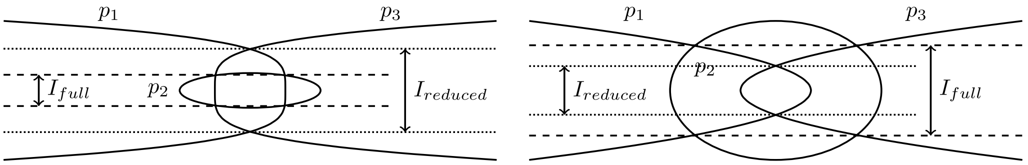
1. I is covered by a single other interval entirely, as interval (ℓ_2, u_2) is by interval (ℓ_1, u_1) in Fig. 3.
2. I is covered through multiple other intervals, as the interval defined by the bounds of p_2 would be by those from p_1 and p_3 in Fig. 4.

[7] The proofs in Lazard's original 1994 paper were found to be flawed and so the safe use of this operator comes only with the 2019 work of McCallum, Parusiński and Paunescu [40].



(a) Redundancy that still yields resultants. (b) Redundancy that yields no resultants.

Fig. 3. Possible situations for redundant intervals of the first kind.



(a) Beneficial reduction. (b) Detrimental reduction.

Fig. 4. Possible situations for redundant intervals of the second kind.

We now explain why we need to remove redundancies of the first kind, but the second could be kept.

4.5.1. Redundancy of the first kind

The resultants we produce in Lines 10–11 of Algorithm 4 are meant to ensure that consecutive intervals (within the ordering) continue to overlap on the whole region we exclude. Thus we must guarantee that they overlap on our sample point in the first place.

In the examples of Fig. 3 we assume constraints which are satisfied only in the regions outside of the graphed curves, and so the point marked in Fig. 3b is a satisfying witness. The two examples differ only by a small change to the polynomial p_2: in Fig. 3a p_2 intersects p_3 while in Fig. 3b it does not. For both examples the numbering of the intervals means we will calculate the resultants $\text{res}(p_1, p_2)$, $\text{res}(p_2, p_3)$ but not $\text{res}(p_1, p_3)$. In each case the bounds obtained are indicated by the dashed lines: in Fig. 3a they come from the roots of $\text{res}(p_2, p_3)$ while in Fig. 3b this resultant has no real roots and the closest bounds instead come from the discriminant of p_2.

So, in Fig. 3a the excluded region is all unsatisfiable, i.e. correct, but only by luck! The resultant that bounds the excluded regions has no relation to the actual bound (the intersection of p_1 and p_3). In Fig. 3b we exclude too much and erroneously exclude the dot which actually marks a satisfying sample.

We do not see an easy way to distinguish the two situations presented in Fig. 3 and we therefore excluded all redundancies of this kind in Section 4.4.1, by requiring that we have the stronger ordering (4) rather than just the weaker version (3).

4.5.2. Redundancy of the second kind

The error described above of excluding a satisfiable point because of a redundancy of the first kind, could not occur in the presence of a redundant interval of the second kind. The algorithm assumes that every adjacently numbered interval overlaps, which is not the case for a redundancy of the first kind but is the case for one of the second kind.

However, should we still remove redundant intervals of the second kind for efficiency purposes? Removal would mean less intervals in the covering, and thus less projection in Algorithm 4 and less root isolation in Algorithm 5. However, it does not mean the generalisation has to be bigger.

Intuitively, reducing such a redundancy makes the overlap between adjacent intervals smaller. But this may also mean that the interval we can exclude in the lower dimension is smaller than it would have been if we kept the redundant interval, retaining a larger overlap. Consider the two examples from Fig. 4: the polynomials differ but the geometric situation and position of intervals is similar. In both cases an interval defined by the bounds of p_2 will be redundant when combined with those from the bounds of p_1 and p_3. If we do not reduce the covering by the redundant interval then we exclude I_{full} but if we do we exclude $I_{reduced}$. The excluded region would grow from reduction in Fig. 4a but shrink in Fig. 4b.

We do not see an easy way to check which situation we have (other than completely calculating both and taking the better one). The decision of whether to reduce could be taken heuristically by an implementation. Further investigation into this would be an interesting topic for future work.

4.5.3. Upper and lower bounds of the same interval crossing

Recall that at the end of Section 4.4.2 we noted that our characterisation does not include the resultant of polynomials defining the upper bound of an interval with those defining the lower bound of the same interval. Now we have discussed

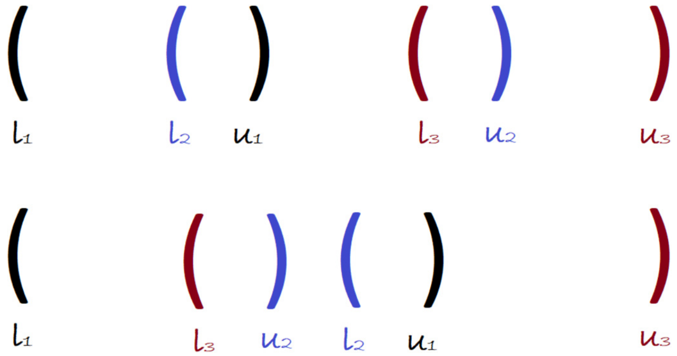


Fig. 5. An example to show that if an interval's upper and lower bounds cross then the interval is no longer required.

redundancy we can explain why these are not required. Consider for example the triple of intervals in the top of Fig. 5 and the possibility that ℓ_2 and u_2 may swap order (which would have been blocked by taking the resultant of defining polynomials in the characterisation). Now, since the characterisation did include the resultants of polynomials defining upper bounds with those defining lower bounds of the next interval it is not possible for ℓ_2 to pass to the right of u_1, or for u_2 to pass to the left of ℓ_3. Thus the only way that ℓ_2 and u_2 could pass in a generalisation is if their neighbours moved with them, as in the bottom of Fig. 5. We can now observe that the second interval has become redundant, i.e. the portion of the line that it used to cover is now fully covered by the other two intervals. The bounds for the second interval must now be fully contained by both the first *and* second interval. Thus at all times in this situation the first and third interval must overlap. Hence this redundancy is of the second kind and thus safe for the correctness of the algorithm.

4.6. Embedding as theory solver

Our algorithm can be used as a standalone solver for a set of real arithmetic constraints, but our main motivation is to use it as a theory solver within an SMT solver. Such theory solvers should be *SMT compliant*:

- They should work *incrementally*, i.e allow for the user to add constraints and solve the resulting problem with minimal recomputation.
- They should similarly allow for *backtracking*, i.e. the removal of constraints by the user.
- They should construct reasons for unsatisfiability, which usually refers to a subset of the constraints that are already unsatisfiable, often referred to as *infeasible subsets*.

For infeasible subsets we store, for every interval, a set of constraints that contributed to it, in a similar way to what was called *origins* in [35]. For intervals created in Algorithm 3 we simply use the respective constraint c as their origin. For intervals that are constructed in Algorithm 5 the set of constraints is computed as the union of all origins used in the corresponding covering in Algorithm 4. The constraints are then gathered together as the final step before returning an UNSAT result in Algorithm 1 (Line 5).

We have yet to implement incrementality and backtracking, but neither pose a theoretical problem. Though possibly involved in the implementation, the core idea for incrementality is straight-forward: after we found a satisfying sample we retain the already computed (partial) coverings for every variable and extend them with more intervals from the newly added constraints. For backtracking, we need to remove intervals from these data structures based on the input constraints they stem from. As we already store these to construct infeasible subsets, this is merely a small addition.

5. Worked examples

5.1. Simple SAT example in 2D

We start with a simple two-dimensional example to show how the algorithm proceeds in general. Our aim is to determine the satisfiability for the conjunction of the following three constraints:

$$c_1: \quad 4 \cdot y < x^2 - 4, \qquad c_2: \quad 4 \cdot y > 4 - (x-1)^2, \qquad c_3: \quad 4 \cdot y > x + 2. \tag{5}$$

The three defining polynomials are graphed in Fig. 6, with the unsatisfiable regions for each adding a layer of shading (so the white regions are where all three constraints are satisfied).

We now simulate how our algorithm would process these constraints, under variable ordering $x \prec y$, to find a point in one of those satisfying regions. The user procedure (Algorithm 1) starts by calling the main algorithm `get_unsat_cover()` (Algorithm 2) with an empty tuple as the sample.

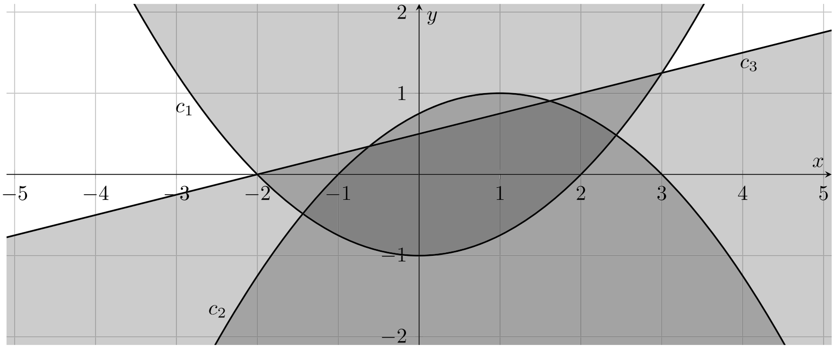


Fig. 6. Graphs of the polynomials defining the three constraints in the simple example (5).

`get_unsat_cover(`$s = ()$`)`. Since all the constraints are bivariate, the call to `get_unsat_intervals()` cannot ascertain any intervals to rule out, initialising $\mathbb{I} = \emptyset$. We thus enter the main loop of Algorithm 2 and sample $s_1 = 0$ in Line 3. This sample does not have full dimension and we perform the recursive call in Line 6.

`get_unsat_cover(`$s = (x \mapsto 0)$`)`. This time, the call of `get_unsat_intervals(`$s = (x \mapsto 0)$`)` sees all three constraints rendered univariate by the sample. Each univariate polynomial has one real root and thus decomposes the real line into three intervals. After analysing each of these 9 regions we identify 6 for which a constraint is infeasible. Thus $\mathbb{I}$ is initialised

Constraint	Real Roots	Intervals
$c_1: \quad 4 \cdot y < x^2 - 4$	$\{-1\}$	$(-\infty, -1)$, $[-1, -1]$, $(-1, \infty)$
$c_2: \quad 4 \cdot y > 4 - (x-1)^2$	$\{\frac{3}{4}\}$	$(-\infty, \frac{3}{4})$, $[\frac{3}{4}, \frac{3}{4}]$, $(\frac{3}{4}, \infty)$
$c_3: \quad 4 \cdot y > x + 2$	$\{\frac{1}{2}\}$	$(-\infty, \frac{1}{2})$, $[\frac{1}{2}, \frac{1}{2}]$, $(\frac{1}{2}, \infty)$

as a set of six intervals as below (where p_i refers to the defining polynomial for constraint c_i).

$$\begin{array}{lllllll}
I_1: & \ell = -1, & u = -1, & L = \{p_1\}, & U = \{p_1\}, & P_2 = \{p_1\}, & P_\perp = \emptyset, \\
I_2: & \ell = -1, & u = \infty, & L = \{p_1\}, & U = \emptyset, & P_2 = \{p_1\}, & P_\perp = \emptyset, \\
I_3: & \ell = -\infty, & u = \frac{3}{4}, & L = \emptyset, & U = \{p_2\}, & P_2 = \{p_2\}, & P_\perp = \emptyset, \\
I_4: & \ell = \frac{3}{4}, & u = \frac{3}{4}, & L = \{p_2\}, & U = \{p_2\}, & P_2 = \{p_2\}, & P_\perp = \emptyset, \\
I_5: & \ell = -\infty, & u = \frac{1}{2}, & L = \emptyset, & U = \{p_3\}, & P_2 = \{p_3\}, & P_\perp = \emptyset, \\
I_6: & \ell = \frac{1}{2}, & u = \frac{1}{2}, & L = \{p_3\}, & U = \{p_3\}, & P_2 = \{p_3\}, & P_\perp = \emptyset.
\end{array}$$

Although $(-\infty, \infty)$ is not an interval of $\mathbb{I}$ we observe that $\mathbb{R}$ is already covered by the two intervals $(-\infty, \frac{1}{2})$ and $(-1, \infty)$. Note how the second of the three constraints is not part of the conflict which simplifies the following calculations.

Since the real line is already covered we do not enter the main loop of Algorithm 2 in this call. Instead we immediately proceed to the final line where we return (UNSAT, $\mathbb{I}$) to the call of `get_unsat_cover(`$s = ()$`)`. Since the flag is UNSAT the next step in that call is constructing a characterisation in Line 10.

`construct_characterisation(`$s = (x \mapsto 0), \mathbb{I}$`)`. As already observed, the intervals $(-\infty, \frac{1}{2})$ and $(-1, \infty)$ cover $\mathbb{R}$ and thus the call to `compute_cover()` at the start should simplify $\mathbb{I}$ to the following:

$$\begin{array}{llllll}
I_1: \ell = -\infty, & u = \frac{1}{2}, & L = \emptyset, & U = \{4 \cdot y - x - 2\}, & P_2 = \{4 \cdot y - x - 2\}, & P_\perp = \emptyset, \\
I_2: \ell = -1, & u = \infty, & L = \{4 \cdot y - x^2 + 4\}, & U = \emptyset, & P_2 = \{4 \cdot y - x^2 + 4\}, & P_\perp = \emptyset.
\end{array}$$

As P_i only contains a single polynomial in each interval there are no resultants to calculate in the first loop, and only a single one from the second:

$$\mathtt{res}_y(4 \cdot y - x - 2, 4 \cdot y - x^2 + 4) = -4 \cdot x^2 + 4 \cdot x + 24.$$

Further, $P_\perp = \emptyset$ and the discriminants and leading coefficients all evaluate to constants:

$$\texttt{disc}_y(4\cdot y - x - 2) = 1, \qquad \texttt{lcoeff}_y(4\cdot y - x - 2) = 4,$$
$$\texttt{disc}_y(4\cdot y - x^2 + 4) = 1, \qquad \texttt{lcoeff}_y(4\cdot y - x^2 + 4) = 4.$$

So the output set from construct_characterisation$(s = (x \mapsto 0), \mathbb{I})$ consists of a single polynomial: $R = \{x^2 - x - 6\}$. Then in the original call to get_unsat_cover$(s = ())$ we continue working in Line 11 with the construction of an interval from the characterisation.

interval_from_characterisation$(s = (), s_i = (x \mapsto 0), P = \{x^2 - x - 6\})$. We see that $P_\perp = \emptyset$ and $P_i = P$ and obtain the real roots $\{-2, 3\}$. Consequently we obtain $l = -2$, $u = 3$ with both L and U as P_i. We thus return the unsatisfiable interval $(-2, 3, P_i, P_i, P_i, \emptyset)$.

Back in our initial call to Algorithm 2 we add this interval to $\mathbb{I}$ and continue with a second iteration of the main loop. This time we must sample some value of x outside of $(-2, 3)$, for example $x = -3$ or $x = 4$, both of which can be extended to a full satisfying sample: $(x \mapsto -3, y \mapsto 0)$ and $(x \mapsto -4, y \mapsto 2)$ respectively. In fact, any sample for x outside of the interval $(-2, 3)$ can be extended to a full assignment and that would be always discovered during the next recursive call to Algorithm 2.

get_unsat_cover$(s = (x \mapsto \hat{x})\})$. Here $\hat{x}$ is the sample for x outside $(-2, 3)$ chosen above. For any such sample first Algorithm 5 will rule out any extension for y that is infeasible and then sample_outside would pick a satisfying value of y in the first iteration of the main loop. This would form a full dimensional sample which is returned along with the flag SAT by Line 5.

Back in our initial call to Algorithm 2 we then pass the tuple of flag and satisfying sample back to the user function in the return on Line 8.

5.1.1. Comparing to incremental CAD

Let us consider how this would compare with the incremental version of traditional CAD. Recall that this performs one projection step at a time, and then refines the decomposition with respect to the output (producing extra samples). Thus the computation path depends on the order in which projections are performed. The discriminants and coefficients of the input do not contribute anything meaningful to the projections, so it all depends in which order the resultants are computed. If the implementation were unlucky and picked the least helpful resultant it will end up performing more decomposition than is required. For this particular example the effect is mitigated a little by the implementation's preference for picking integer sample points allowing it to find a satisfying sample earlier than guaranteed by the theory: i.e. the cell being formed by the decomposition is not truth invariant for all constraints, but by luck the sample picked is SAT and so the algorithm can terminate early. So for this example our incremental CAD performs no more work than the new algorithm, but that is through luck rather than guidance. In contrast, the superiority of the new algorithm over incremental CAD is certain for the next example.

5.1.2. Comparison to NLSAT

We may also compare to the NLSAT method [36] which also seeks a single satisfying sample point for all the constraints. Like our new method, NLSAT is model driven starting with a partial sample; it *explains* inconsistent models locally using CAD techniques; and thus exploits the locality to exclude larger regions; combining multiple of these explanations to determine unsatisfiability. The difference is that the conflicts are then converted into a new lemma for the Boolean skeleton and passed to a separate SAT solver to generate a new model. The implementation of NLSAT in SMT-RAT (see Section 6.4) performs the following steps for the first example:

Theory model	Explanation clause	Excluded interval
$x \mapsto 0$	$(c_1 \wedge c_2) \to (x \le \underline{-1.44} \vee \underline{2.44} \le x)$	$(\underline{-1.44}, \underline{2.44})$
$x \mapsto -2$	$(c_1 \wedge c_3) \to (x \ne -2)$	$[-2, -2]$
$x \mapsto 3$	$(c_1 \wedge c_3) \to (x \ne 3)$	$[3, 3]$
$x \mapsto -3$	$y \mapsto 0$	

The boundaries of the first excluded interval are the two real roots of the polynomial $2x^2 - 2x - 7$ which is the resultant of the defining polynomials for c_1 and c_2. Rather than give these as surds or algebraic numbers we use the decimal approximation simply to shorten the presentation in the paper. Throughout, a decimal underlined refers to a full algebraic number that we have computed but choose not to display for brevity.

For this example NLSAT took three conflicts to find a satisfying model (compared to two for our new algorithm). However, the difference is due to luck. In the first iteration NLSAT was unlucky to select a subset of constraints that rules out a smaller UNSAT region. It could have instead chosen c_1 and c_3 in the first iteration, essentially yielding the very same computation as our method. Conversely our algorithm could also have chosen c_1 and c_2 as a cover and then have needed another iteration.

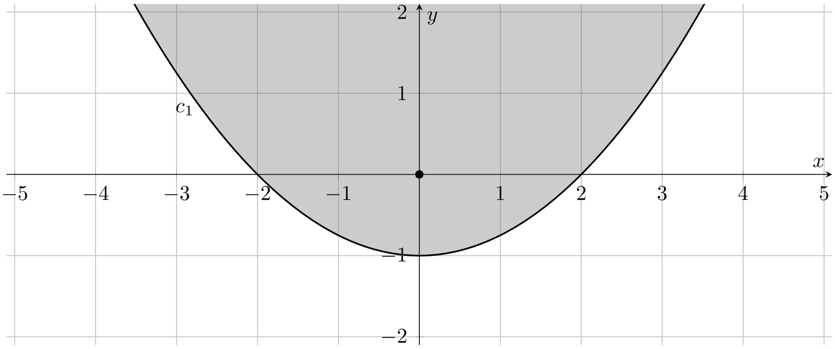


Fig. 7. First NuCAD cell for $(0, 0)$.

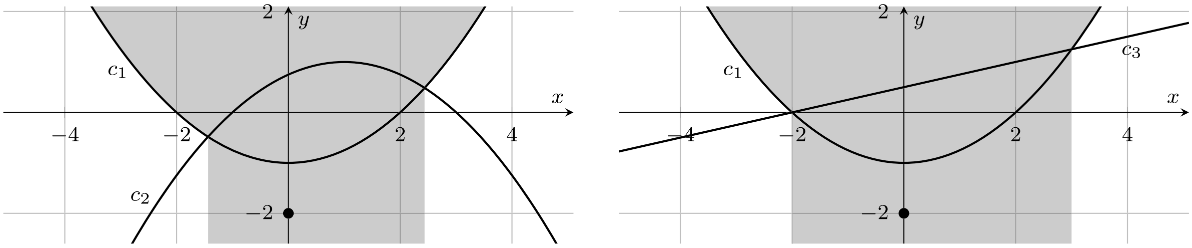


Fig. 8. Possible continuations for NuCAD after Fig. 7.

5.1.3. Comparison to NuCAD

Our algorithm as stated is designed to solve satisfiability problems, and thus terminates as soon as a satisfying sample is found. Thus it does not compare directly with NuCAD which builds an entire decomposition of $\mathbb{R}^n$ relative to the truth of a quantifier free logical formula which can then be used to solve more general QE problems stated about that formula.[8] NuCAD constructs the decomposition of one cell (with known truth-value) in $\mathbb{R}^n$ at a time, so there is a natural variant of the algorithm where we construct cells only until we find one in which the input formula is `True`. Consider how NuCAD might proceed for the simple example. A natural starting point would be to consider the origin first. NuCAD would recognise that constraints are not satisfied here and choose one of them to process. Let us assume NuCAD chooses in the order the constraints are labelled (i.e. pick c_1); then like us it would generalise this knowledge beyond the sample and decompose the plane as in Fig. 7. The shaded cell is known to be UNSAT while the white region is a cell whose truth value is as yet unknown.

Now NuCAD must pick a new sample outside of the shaded cell. A natural choice would be keep one coordinate zero (i.e. move along the axis). If it were to move left along the x axis and pick the first integer outside the shaded cell, i.e. sample $(-3, 0)$ then it would find a satisfying point and terminate, but if it were to move right to $(3, 0)$ it would have to decompose further. Another natural strategy may be to keep the x value fixed and try other y values, which allows a direct comparison to our method. A preference for the next integer leads to the sample $(0, -2)$ where both c_2 and c_3 are violated. The two possible resulting cells are shown in Fig. 8 where picking c_2 leads to the decomposition on the left, and picking c_3 to the decomposition on the right (similar to the two possible steps our algorithm had).

It is possible under a natural strategy for NuCAD to find a satisfying point on its second sample, but this is not guaranteed by the theory. In comparison, our algorithm is more guided, if starting at the origin then it could not take more than three iterations for this example.

5.2. More involved UNSAT example in 2D

In the first example above we saw that the new algorithm was able to learn from conflicts to guide itself towards a satisfying sample. However, due to luck and sensible implementation choices the alternatives could process the example with a similar amount of work to the new algorithm. So we next present a more involved example that will make clearer the savings offered by the new approach. This example will ultimately turn out to be unsatisfiable and thus the incremental CAD of SMT-RAT will in the end perform all projection and produce a regular CAD, as for example would be produced by a traditional CAD implementation like QEPCAD. The advantage of our algorithm in this case is that we can avoid some parts of the projection that are the most expensive.

[8] Although this does require some additional computation as outlined in [42].

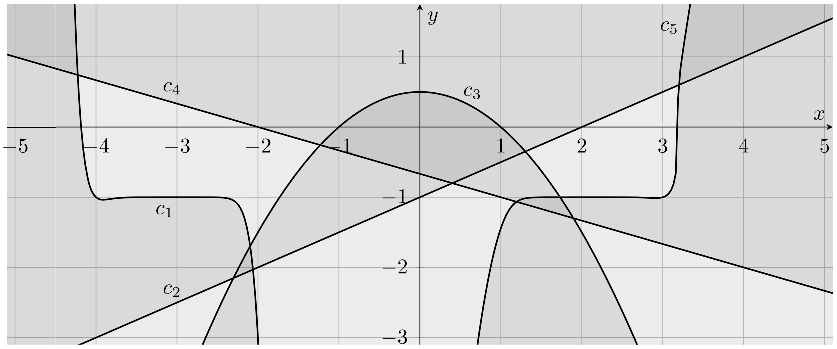


Fig. 9. Graphs of the polynomials defining the constraints in the more involved example (6).

Our aim for this example is to determine the satisfiability for the conjunction of the following five constraints:

$$\begin{array}{lll} c_1 : y > (-x-3)^{11} - (-x-3)^{10} - 1 & c_2 : 2 \cdot y < x - 2 & c_3 : 2 \cdot y > 1 - x^2 \\ c_4 : 3 \cdot y < -x - 2 & c_5 : y^3 > (x-2)^{11} - (x-2)^{10} - 1 & \end{array} \tag{6}$$

They are depicted graphically in Fig. 9, with each constraint again adding a layer of shading where it is unsatisfiable. This time we see there is no white space and so together the constraints are unsatisfiable.

Before we proceed let us remark on how the example was constructed. There are two constraints of high degree (c_1 and c_5) while the rest are fairly simple. The problem structure was chosen so that c_1 and c_2 conflict on the left part of the figure, c_2 and c_3 in the middle and c_4, and c_5 on the right. Thus while both c_1 and c_5 are important to determining unsatisfiability, they never need to be considered together to do this. I.e. we have no need of their resultant (with respect to y): an irreducible degree 33 polynomial in x which is dense (34 non-zero terms) with coefficients that are mostly 20 digit integers. In this example c_1 and c_5 are of degree 11 but we could generalise the example by increasing the degree of these polynomials arbitrarily while retaining the underlying problem structure which allows us to avoid working with them together. Hence the advantage over algorithms which do consider them together can be made arbitrarily large.

We now describe how our new algorithm would proceed for this example but skip some details compared to the explanation of the previous example, to avoid unnecessary repetition.

`get_unsat_cover` $(s = ())$. As all constraints contain y we get no unsatisfiable intervals in the first dimension. We enter the loop and sample $s_1 = 0$ and enter the recursive call.

`get_unsat_cover` $(s = (x \mapsto 0))$. All constraints are univariate and `get_unsat_intervals` $(s = (x \mapsto 0))$ returns $\mathbb{I}$ defining the following 10 unsatisfiable intervals:

$$\begin{array}{lll} (-\infty, -236197), & [-236197, -236197], & \text{from } c_1, \\ [-1, -1], & (-1, \infty), & \text{from } c_2, \\ (-\infty, \frac{1}{2}), & [\frac{1}{2}, \frac{1}{2}], & \text{from } c_3, \\ [-\frac{2}{3}, -\frac{2}{3}], & (-\frac{2}{3}, \infty), & \text{from } c_4, \\ (-\infty, \underline{-14.5}), & [\underline{-14.5}, \underline{-14.5}] & \text{from } c_5. \end{array}$$

We can select intervals $(-\infty, \frac{1}{2})$ and $(-1, \infty)$ to cover $\mathbb{R}$ and return this to the main call.[9]

The main call analyses the covering and constructs the characterisation $\{x^2 + x - 3\}$ leading to the exclusion of the interval $(\underline{-2.30}, \underline{1.30})$. A graphical representation is shown in the left image of Fig. 10. We then iterate through the loop again selecting a sample outside of this interval, say $s_1 = 2$, with which we enter another recursive call.

`get_unsat_cover` $(s = (x \mapsto 2))$. This call is almost identical to the one above, only we are now forced to use c_4 to build a conflict with c_3 instead of c_2. The unsatisfiable intervals that initialise $\mathbb{I}$ are now as follows:

$$\begin{array}{lll} (-\infty, -58593751), & [-58593751, -58593751], & \text{from } c_1, \\ [0, 0], & (0, \infty), & \text{from } c_2, \end{array}$$

[9] The latter interval was provided by c_2, but we could have instead taken $(-\frac{2}{3}, \infty)$ from c_4 and still covered $\mathbb{R}$. .

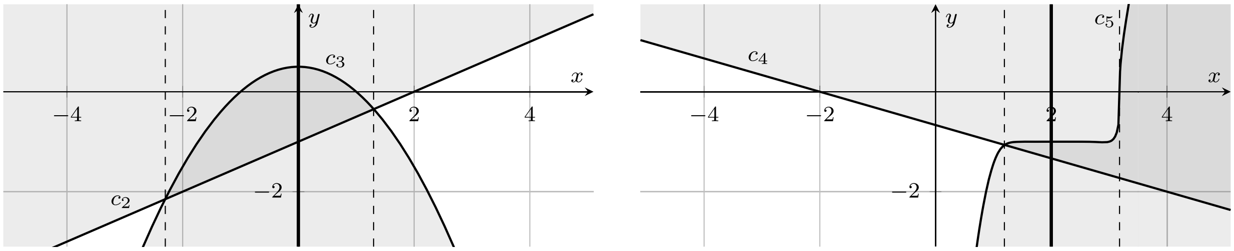


Fig. 10. Excluded regions for $(x \mapsto 0)$ on the left and for $(x \mapsto 2)$ on the right.

$(-\infty, -\frac{3}{2})$,	$[-\frac{3}{2}, -\frac{3}{2}]$,	from c_3,
$[\underline{-1.33}, \underline{-1.33}]$,	$(\underline{-1.33}, \infty)$,	from c_4,
$(-\infty, -1)$,	$[-1, -1]$	from c_5.

We obtain the (unique) minimal covering from these by selecting $(-\infty, -1)$ and $(\underline{-1.33}, \infty)$ provided by c_5 and c_4 respectively. So we skip the loop and return this to the original parent call.

In the parent call the characterisation (after some simplification) contains two polynomials of degrees nine and eleven from the discriminant for c_5 and the resultant of the pair. They each yield one real root: $\underline{1.19}$ from the resultant (corresponding to the crossing point of c_4 and c_5) and $\underline{3.18}$ from the discriminant corresponding to the point of inflection of c_5. These are visualised in the right image of Fig. 10. Note that the latter point is spurious in the sense that the point of inflection has no bearing on the satisfiability of our problem. But while we could safely ignore it, we have no algorithmic way of doing so. Thus we exclude the interval $(\underline{1.19}, \underline{3.18})$ and proceed.

We now continue iterating the main loop with first the sample point $s_1 = \underline{3.18}$ and then $s_1 = 4$, which both yield the same conflict based on c_5 and c_4 and thus the exact same characterisation, excluding $[\underline{3.18}, \underline{3.18}]$ and $(\underline{3.18}, \infty)$ in turn. Finally we select $s_1 = -3$ and recurse one last time.

get_unsat_cover $(s = (x \mapsto -3))$. Once again we compute the unsatisfiable intervals and obtain the following intervals in the initialisation of $\mathbb{I}$:

$(-\infty, -1)$,	$[-1, -1]$,	from c_1,
$[-\frac{5}{2}, -\frac{5}{2}]$,	$(-\frac{5}{2}, \infty)$,	from c_2,
$(-\infty, -4)$,	$[-4, -4]$,	from c_3,
$[\frac{1}{3}, \frac{1}{3}]$,	$(\frac{1}{3}, \infty)$,	from c_4,
$(-\infty, -388)$,	$[-388, -388]$	from c_5.

We now use the unique covering $(-\infty, -1)$ and $(-\frac{5}{2}, \infty)$ (from c_1 and c_2), yielding a characterisation consisting solely of the resultant of the polynomials defining c_1 and c_2 which has degree 11. The excluded interval is $(-\infty, \underline{-2.06})$ which covers the whole region to the left of those excluded before.

At this point we have collected the following unsatisfiable intervals for the lowest dimension x in our main function call.

$$(-\infty, \underline{-2.06}), (\underline{-2.30}, \underline{1.30}), (\underline{1.19}, \underline{3.18}), [\underline{3.18}, \underline{3.18}], (\underline{3.18}, \infty)$$

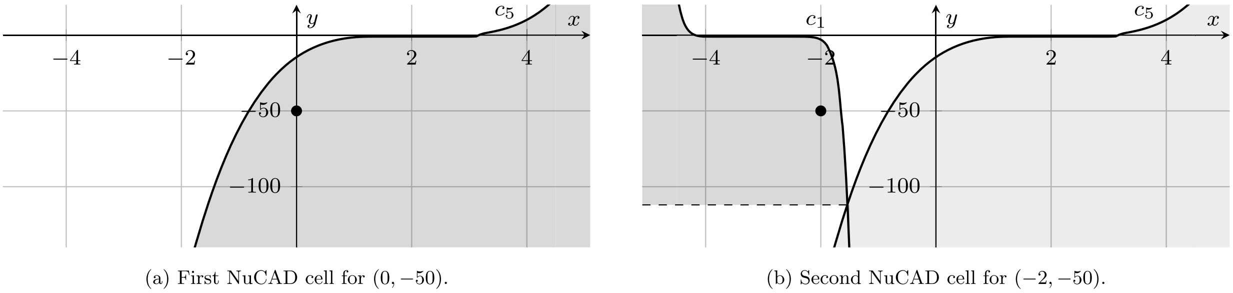
We see that we have covered $\mathbb{R}$ and thus we return a final answer of UNSAT. Note that the highest degree of any polynomial we used in the above was eleven: the degree 33 resultant of c_1 and c_5 was never used nor even computed.

It is important to recognise the general pattern here that allows our algorithm to gain an advantage over a regular CAD. We have two more complicated constraints involved in creating the conflict, but they are separated in space, or at least the regions where they are needed to construct the conflict are separated. Increasing the degrees within c_1 and c_5 would affect the algorithm only insofar that we have to perform real root isolation on larger polynomials while regular CAD has to deal with the resultant of these polynomials whose degree grows quadratically.

5.2.1. Comparing to incremental CAD

The full projection contains one discriminant and 10 resultants (plus also some coefficients depending on which projection operator is used). Since no satisfying sample will be found the incremental CAD will actually produce a full sign-invariant CAD for the polynomials in the problem, containing 273 cells.

There is scope for some optimisations here, for example, because the constraints are all strict inequalities we know that they are satisfied if and only if they are satisfied on some full dimensional cell and so we could avoid lifting over any restricted dimension cells. The CAD decomposes the real line into 27 cells and so we could optimise to lift only over the 13



(a) First NuCAD cell for $(0, -50)$.

(b) Second NuCAD cell for $(-2, -50)$.

Fig. 11. Graphs of the polynomials defining c_1 and c_5 in the more involved example (6).

intervals to consider 77 cells in the plane. This avoids some work but we still need to compute and isolate real roots for the full projection set, and perform further lifting and real root isolation over half of the cells in $\mathbb{R}^1$. This is significantly more real root isolation and even projection than computed by the new algorithm, in particular, the large resultant discussed above.

5.2.2. Comparison to NLSAT

SMT-RAT's implementation of NLSAT performs the following samples and conflicts for this example.

Theory model	Explanation clause	Excluded interval
$x \mapsto 0$	$(c_2 \wedge c_3) \to (x \le -2.3 \vee 1.3 \le x)$	$(\underline{-2.3}, \underline{1.3})$
$x \mapsto -3$	$(c_1 \wedge c_2) \to (\underline{-2.06} \le x)$	$(-\infty, \underline{-2.06})$
$x \mapsto 2$	$(c_4 \wedge c_5) \to (x \le \underline{1.19} \vee x \ge \underline{3.18})$	$(\underline{1.19}, \underline{3.184})$
$x \mapsto 5$	$(c_2 \wedge c_5) \to (x \le \underline{3.199})$	$(\underline{3.199}, \infty)$
$x \mapsto 52307/16384$	$(c_4 \wedge c_5) \to (x \le \underline{3.184})$	$(\underline{3.184}, \infty)$
$x \mapsto \underline{3.184}$	$(c_4 \wedge c_5) \to (x \ne \underline{3.184})$	$[\underline{3.184}, \underline{3.184}]$

Like the new algorithm it starts with the origin and gradually rules out the whole x-axis by generalising from a sample each time. However, NLSAT required 6 iterations to do this, compared to 5 for the new algorithm. The main reason for this is that it uses c_2 and c_5 (instead of c_4 and c_5) to explain the conflict at $x \mapsto 5$ and thus obtains a slightly smaller interval, leaving a gap between $\underline{3.184}$ and $\underline{3.199}$. However, this poorer choice could have also been made by our new algorithm.

5.2.3. Comparison to NuCAD

This time, because the example is UNSAT, NuCAD will have to complete and produce an entire decomposition of the plane. The order in which cells are constructed will be driven by implementation as the only specification in the algorithm is to choose samples outside of cells with known truth value. Since any reasonable implementation would start with the origin and prefer integer samples it is likely that NuCAD's computation path would follow similarly to our algorithm for this example. But it should be noted that this is not guaranteed. For example, if instead of the origin NuCAD were to start with the model $(0, -50)$ then here c_5 is violated and it may start by constructing the shaded cell in Fig. 11b.

It must now pick a model outside of the shaded region. Again, it would be a strange choice but the model $(-2, -50)$ would be acceptable and here c_1 is violated. The necessary splitting would then require the calculation and real root isolation of the large resultant of the defining polynomials of c_1 and c_5. There is one such real root, indicating a real intersection of the two polynomials as shown in Fig. 11a, and this would necessarily be part of the boundary in the constructed cell.

We accept that the above model choices would be strange but they are a possibility for NuCAD while guaranteed to be avoided by the new algorithm. Of course, it should be possible to shift the coordinate system of the example to make these model choices seem reasonable.

5.3. Simple 3D example

The 2D examples demonstrated how the new algorithm performs less work than a CAD and is more guided by conflict than NuCAD. However, to demonstrate the benefits over NLSAT we need more dimensions.

Consider the simple problem in 3D of simultaneously satisfying the constraints:

$$c_1 : x^2 + y^2 + z^2 < 1, \qquad c_2 : x^2 + (y - \tfrac{3}{2})^2 + z^2 < 1. \tag{7}$$

These require us to be inside two overlapping spheres as on the left of Fig. 12. Let us consider how the new algorithm would find a witness.

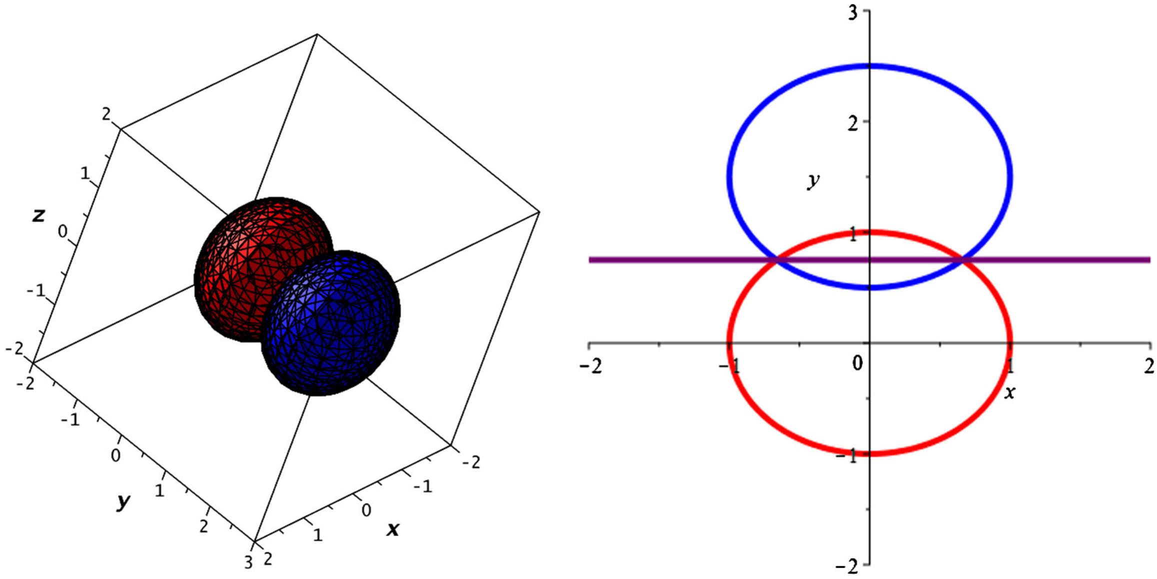


Fig. 12. On the left are the spheres defined by constraints (7) and on the right the cures of the (x, y)-plane computed as the first characterisation. (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

`get_unsat_cover`$(s = ())$. Our algorithm cannot draw any conclusions from the constraints as none are defined only in x, so it samples $x = 0$ and enters the recursive call.

`get_unsat_cover`$(s = (x \mapsto 0))$. Once again no conclusions can be drawn from the constraints yet so we sample $y = 0$ and recurse again.

`get_unsat_cover`$(s = (x \mapsto 0, y \mapsto 0))$. This time the call of `get_unsat_intervals`$(s = (x \mapsto 0, y \mapsto 0))$ does produce some conclusions. It concludes that the first constraint is unsatisfiable outside of $x \in (-1, 1)$ while the second constraint is not satisfiable anywhere over this sample. I.e. the unsat interval $(-\infty, \infty)$ is part of the output. This we may skip the main loop of `get_unsat_cover` and return to the previous call.

`get_unsat_cover`$(s = (x \mapsto 0))$ *`continued`*. For the characterisation we need calculate only the discriminant of the polynomial defining c_2, which after simplification is $x^2 + (y - \frac{3}{2})^2 - 1$ (the blue circle in the right of Fig. 12). We have no need to calculate the discriminant of the other defining polynomials, or their resultant (the other graphs on the right of Fig. 12 which would be computed by a full CAD).

When forming the interval around $y = 0$ we obtain the set $Z = \{-\infty, -1, \frac{1}{2}, \frac{5}{2}, \infty\}$ and so we generalise to $(-\infty, \frac{1}{2})$.

- Sampling for y anywhere outside $(\frac{1}{2}, 1)$ would lead to a full covering of the z-dimension after the initial querying of constraints in the recursive call obtained by analysing c_1. The discriminant of c_1 would then be taken for the characterisation and the generalisation will rule out $(1, \infty)$.
- Any sample from within $(\frac{1}{2}, 1)$ can be simply extended with $z = 0$ to a full satisfying witness.

5.3.1. Comparison to NLSAT

NLSAT would proceed similarly in sampling $(x, y) = (0, 0)$ and discovering the conflict. However, it would then immediately build a cell around $(0, 0)$ requiring the computation of the full projection of those polynomials in the conflict (i.e. not just the calculation of the discriminant above but then its discriminant also). Only then would NLSAT move to a new partial sample. In contrast, our new algorithm only computes projections with respect to the second variable once it has determined there is no possible y to extend the x-value. Since in this example there is such a y for the first x-value those projections with respect to y need never be computed. Recall that iterated projection operations is the source of the doubly exponential growth in CAD, thus for a conflict which involved multiple constraints the savings are even more significant.

5.4. More involved 3D example

We finish with a larger 3D example to demonstrate some of the facets of the algorithm not yet observed in the smaller examples. We define the three polynomials:

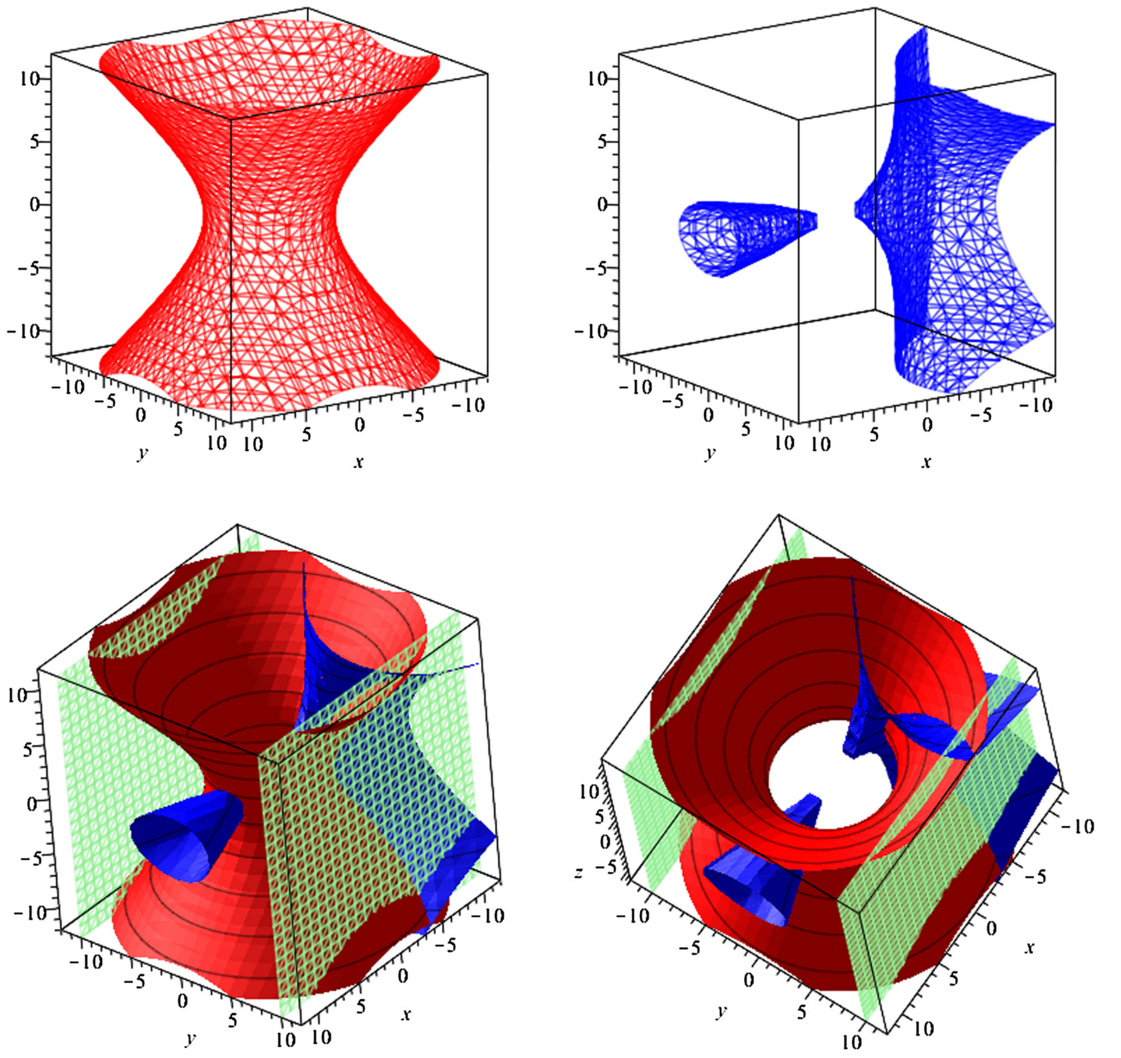


Fig. 13. Surfaces defined by the polynomials in (8).

$$\begin{aligned} f &:= -z^2 + y^2 + x^2 - 25, \\ g &:= (y - x - 6)z^2 - 9y^2 + x^2 - 1, \\ h &:= y^2 - 100, \end{aligned} \tag{8}$$

and seek to determine the satisfiability of

$$f > 0 \wedge g > 0 \wedge h < 0.$$

We use variable ordering $x \prec y \prec z$, and note that unlike the previous example we have here not only a higher dimension, but also a non-trivial leading coefficient for g and a constraint that is not in the main variable formed by h. Finally please note that z appears only as z^2 in the constraints and thus facts drawn for positive z may be applied similarly for negative.[10]

The surfaces defined by the polynomials are visualised in Fig. 13 where the red surface is for f, the blue for g and the green for h. We see that f is a hyperboloid, while g would have been a paraboloid if it were not for the leading coefficient in z. The final constraint simply bounds y to $(-10, 10)$. We see that there are many non-trivial intersections (and self intersections) between the surfaces. A full sign-invariant CAD for the three polynomials may be produced with the Regular Chains Library for Maple [43] in about 20 seconds, and contains 3509 cells.

[10] We are not suggesting our algorithm makes this simplification, we just seek to reduce the presentation of details here.

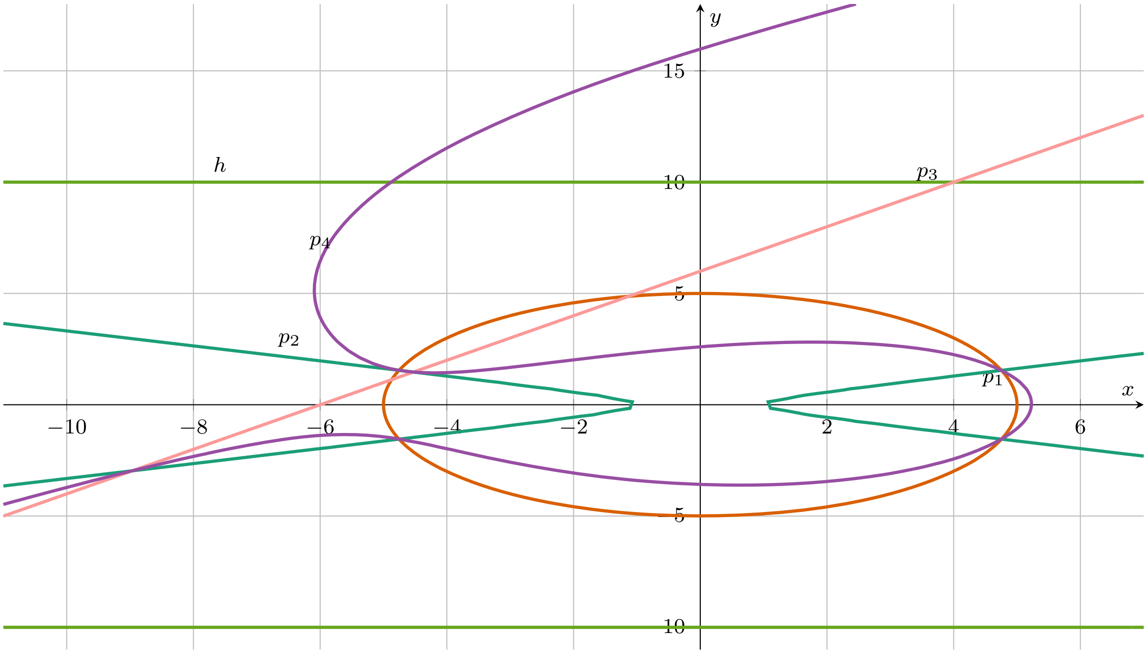


Fig. 14. Projection of the surfaces in Fig. 13.

For use in the discussion later, let us examine and put a label on all the CAD projection polynomials in (x, y) (i.e. those obtained after z is projected out)[11]:

$$\begin{aligned}
p_1 &:= y^2 + x^2 - 25\\
p_2 &:= -9y^2 + x^2 - 1\\
p_3 &:= -y + x + 6\\
p_4 &:= -y^3 + y^2x + 15y^2 - yx^2 + 25y + x^3 + 5x^2 - 25x - 149
\end{aligned}$$

Here p_1 is the discriminant of f with respect to z (at least up to a constant factor); p_2 and p_3 are factors of the discriminant of g and also the trailing and leading coefficients of g respectively. The resultant of f and g with respect to z evaluated to p_4^2. Of course, the projection also includes h itself and is shown in Fig. 14.

Also for use in the following discussion, we label the six real roots in y that these four polynomials have when evaluated at $x = 0$:

$b_1 = -5$ defined by p_1

$b_2 = \underline{-3.58}$ defined by p_4

$b_3 = 2.60$ defined by p_4

$b_4 = 5$ defined by p_1

$b_5 = 6$ defined by p_3

$b_6 = \underline{15.98}$ defined by p_4

Note that p_2 has no real roots when $x = 0$.

Now let us describe how the new algorithm would determine satisfiability.

get_unsat_cover$(s = ())$. Our algorithm cannot draw any conclusions from the constraints originally as none are defined only in x, so it samples $s_1 = 0$ and enters the recursive call.

[11] We do this first just to ease presentation: the algorithm would only compute a projection polynomial when it needed it.

`get_unsat_cover`$(s=(x\mapsto 0))$. This time, the call of `get_unsat_intervals`$(s=(x\mapsto 0))$ does obtain some information from the third constraint. It produces two unsat intervals: $(-\infty,-10]$ and $[10,\infty)$ both of which are characterised by h. However, since this does not cover the whole line we enter the main loop and sample outside, starting with $s_2=0$.

`get_unsat_cover`$(s=(x\mapsto 0, y\mapsto 0))$. This time the call of `get_unsat_intervals`$(s=(x\mapsto 0, y\mapsto 0))$ produces two intervals, both of which are $(-\infty,\infty)$ defined by f and g respectively. Thus the main loop is skipped and we return to the prior call.

`get_unsat_cover`$(s=(x\mapsto 0))$ *continued*. For the characterisation we need pick only one of the two intervals in $\mathbb{I}$ since both cover the whole line (in the call to `construct_cover` in Algorithm 4). Let us suppose we pick the one formed by g. Then the characterisation returns $\{p_2,p_3\}$. When forming the interval around $y=0$ we obtain the set $Z=\{-\infty,6,\infty\}$ and so we generalise to $(-\infty,6)$ with the bound defined by p_3. We still do not have a full covering so we loop again. The uncovered space is now $[6,10)$ so we must sample from here. Suppose we sample $s_2=7$ and recurse.

`get_unsat_cover`$(s=(x\mapsto 0, y\mapsto 7))$. This time the initial constraints do provide a cover but only when considered together: f requires $z\in(\underline{-4.90},\underline{4.90})$ while g requires $z\notin[\underline{-21.02},\underline{21.02}]$.

`get_unsat_cover`$(s=(x\mapsto 0))$ *continued*. This time the characterisation contains all of $\{p_1,p_2,p_3,p_4\}$ and so Z contains all of the b_i. Hence we can generalise around $x=7$ to the interval $(6,\underline{15.98})$. We now have a cover for all y values over $x=0$ except $y=6$.

`get_unsat_cover`$(s=(x\mapsto 0, y\mapsto 6))$. This time a complete cover is provided from the initial constraint defined by g so no need for the main loop.

`get_unsat_cover`$(s=(x\mapsto 0))$ *continued*. Thus the characterisation is similar to that obtained from the sample $y=0$. We actually have to do a little extra work here: because at $(x,y)=(0,6)$ the leading coefficient of g is zero we include also the trailing coefficient, but this was actually already a factor of the discriminant so we do have the same characterisation as for $y=0$. Because this characterisation produced a Z containing $y=6$ we cannot generalise beyond this point. Nevertheless, we have a full cover over $x=0$.

`get_unsat_cover`$(s=())$ *continued*. The cover is formed from five intervals: two from the initial constraints and three from the recursions as summarised below.

	ℓ	u	L	U	P_i	$P_\perp$
	$-\infty$	-10	$\emptyset$	$\{h\}$	$\{h\}$	$\emptyset$
I_1	$-\infty$	6	$\emptyset$	$\{p_3\}$	$\{p_2,p_3\}$	$\emptyset$
I_2	6	6	$\{p_3\}$	$\{p_3\}$	$\{p_2,p_3\}$	$\emptyset$
I_3	6	$\underline{15.98}$	$\{p_3\}$	$\{p_4\}$	$\{p_1,p_2,p_3,p_4\}$	$\emptyset$
I_4	10	∞	$\{h\}$	$\emptyset$	$\{h\}$	$\emptyset$

The first of these intervals is redundant: the interval $(-\infty,-10)$ from h is entirely inside $(-\infty,6)$ from the sample $y=0$. Hence it will be removed by the call to `construct_cover`, where we also order the remaining four intervals as in the table above.

We now form the characterisation at $x=0$: it contains the discriminants of p_1,p_2 and p_4; those of p_3 and h were constant and so removed. Similarly, all the leading coefficients were constant and so not required. The only within interval resultants required come from I_3. We take the resultant of p_3 with both p_1 and p_4 as both the latter produce smaller real roots. The only between interval resultant required is that between p_4 and h which protects the overlap of the final two intervals. After the standard simplifications (including factorisation) we have the following characterisation.

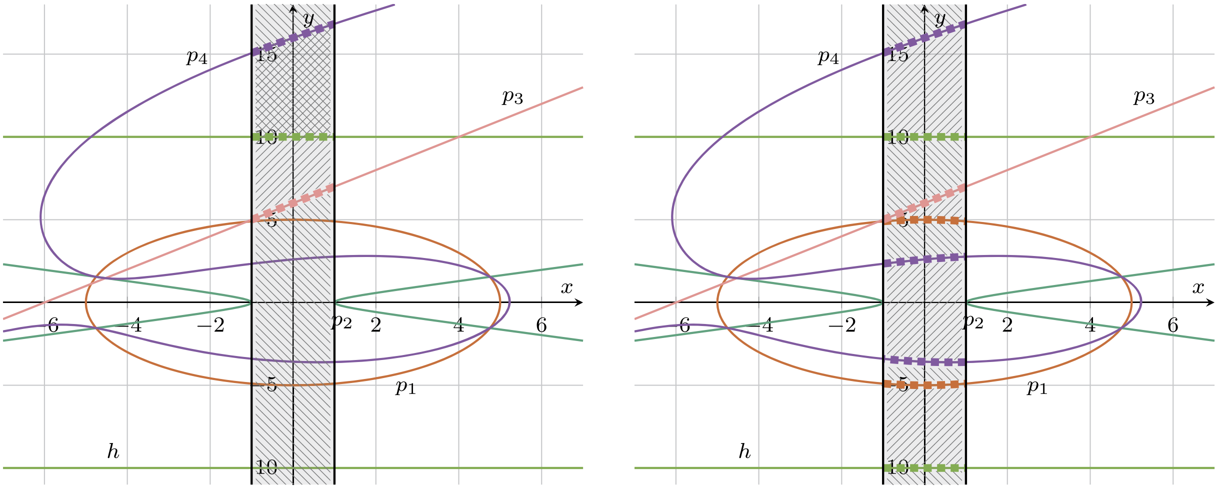
$$\begin{aligned}R := \{&x-5, x+5, x-1, x+1, 4x^6+20x^5+475x^4+6760x^3+7670x^2-198300x-655237,\\ &2x^2+12x+11, 8x^2+108x+325, x^3-5x^2+75x+601, x^3+15x^2+75x+2101\}\,.\end{aligned}$$

This produces the following set of real roots:

$$Z=\{\underline{-17.55},\underline{-8.97},\underline{-6.09},-5,\underline{-4.88},\underline{-4.87},\underline{-4.53},\underline{-1.13},-1,1,5,\underline{5.23}\}\,.$$

Hence we can generalise around $x=0$ to the interval $(-1,1)$.

Fig. 15a visualises this generalisation of the cover. We see that I_1, which was originally the y axis below 6, is generalised to the cylinder bounded at the top by p_3; while I_2, which was originally just the point $(x,y)=(0,6)$, is generalised to the line segment of $y-x-6$ that is above $x\in(-1,1)$. Recall that I_2 was a cell where the leading coefficient of g vanished and observe that this remains the case throughout the generalisation. I_3 and I_4 were overlapping segments of the y-axis and now form overlapping cylinders. These different cells are indicated by the different direction of shading in the figure.



(a) The cover over $x \in (-1, 1)$ generalised from $x = 0$.

(b) The cylinder over $x \in (-1, 1)$ as produced by CAD.

Fig. 15. Comparison of the cover over $x \in (-1, 1)$ with a cylinder from regular CAD.

Let us pause to compare this cylindrical algebraic covering with an actual CAD. A CAD of the projection $\{p_1, p_2, p_3, p_4, h\}$ would also form a cylinder over $x \in (-1, 1)$. However, this cylinder would then be split into 17 cells according to the 8 line segments in the image. We would have to work over 17 samples to determine UNSAT over the cylinder, while we obtained the same conclusion using only 4 samples in the new algorithm.

The algorithm continues in the call to `get_unsat_cover`$(s = ())$ with a sample outside of $(-1, 1)$. A natural choice here is $s_1 = 2$, for which an UNSAT cover would be found consisting of the following six intervals (after removal of redundant ones):

	ℓ	u	L	U	P_i	$P_\perp$
I_1	$-\infty$	$\underline{-0.58}$	$\emptyset$	$\{p_2\}$	$\{p_2, p_3\}$	$\emptyset$
I_2	$\underline{-4.58}$	$\underline{4.58}$	$\{p_1\}$	$\{p_1\}$	$\{p_1\}$	$\emptyset$
I_3	$\underline{0.58}$	8	$\{p_2\}$	$\{p_3\}$	$\{p_2, p_3\}$	$\emptyset$
I_4	8	8	$\{p_3\}$	$\{p_3\}$	$\{p_2, p_3\}$	$\emptyset$
I_5	8	$\underline{17.64}$	$\{p_3\}$	$\{p_4\}$	$\{p_1, p_2, p_3, p_4\}$	$\emptyset$
I_6	10	∞	$\{h\}$	$\emptyset$	$\{h\}$	$\emptyset$

The characterisation then allows a generalisation from $x = 2$ to $x \in (1, 4.75)$, promoting a third sample in the main call of $x = 5$. In the recursion a sample of $y = 0$ is unsat for the first constraint but cannot be generalised. A second sample of $y = 1$ leads to the initial constraints providing a cover for all but $z \in (-1, 1)$. Hence the main loop is entered and we sample $z = 0$ to find a satisfying witness: $(x, y, z) = (5, 1, 0)$ which evaluates f, g, h to $1, 15$ and -99 respectively.

We note that the covering produces far less cells than a traditional CAD. In particular, not every projection factor is used in every covering. This was demonstrated visually by Fig. 15a where the purple resultant was a cell boundary for positive y but not for negative y.

6. Experiments on the SMT-LIB

6.1. Aim of these experiments

We note at the outset that the purpose of these experiments is to establish which of the different algorithmic ideas discussed throughout the paper performs best on a substantial dataset of real problems. The aim is not to prove that a particular solver is the current state of the art. Given our aim it made sense to use implementations of the algorithms in a single system so that they can all draw on the same underlying data-structures and sub-algorithms. This way we can more fairly compare the effects of the high level algorithm with less risk of effects from implementation issues. Thus all the competing solvers in this section are implemented in the established SMT solver SMT-RAT [33].

6.2. Dataset

The worked examples show that the new algorithm does have advantages over existing techniques. We will now check whether this translates into a method that is also competitive in practice. This section describes experiments on the benchmarks for quantifier-free non-linear real arithmetic (QF_NRA) from the SMT-LIB [28] initiative (as of March 2020). This dataset contains 11489 problem instances from 11 different families. Note that these examples are not presented as sets of constraints but can come with more involved Boolean structure, and so the new algorithm is applied as the theory solver for the larger SMT system. Throughout we apply a time limit of 60 seconds and a memory limit of 4 GB for tackling a problem instance.

6.3. New solver CDCAC

We produced an implementation in the course of a Master's thesis [44] within SMT-RAT. The solver CDCAC consists of the regular SAT solver from SMT-RAT, combined with this implementation of the presented method as the only theory solver. We consider this implementation to be a proof of concept, as it does not implement any form of incrementality and there is scope to optimise several subroutines. In the interests of reproducibility it is available from the following URL:

https://github.com/smtrat/smtrat/tree/JLAMP-CDCAC

With regards to the completeness issue discussed in Section 4.4.6: we observed nullification upon substitution of the sample (Algorithm 5 Line 3) in a small, but significant number of examples: 82 from all 11489 instances (0.7%) involved a nullification within the time limit. This means that for these instances the theory does not guarantee UNSAT results of solver CDCAC as correct. However, we note that in all cases the results provided were still correct (i.e. they matched the declared answer stored in the SMT-LIB), which is consistent with our experience of similar CAD completeness issues in [35].

6.4. Competing solvers

We compare the new algorithm to a theory solver based on traditional CAD as described in [35]. We show this in two configurations: CAD-naive uses no incrementality (by projection factor — see Section 2.4) while CAD-full employs what was called *full incrementality* in [35]. Solver CDCAC uses no incrementality, as already discussed in Section 4.6, but the difference between the incremental and non-incremental versions of the full CAD gives a glimpse at what further improvements to CDCAC might be possible. Both of these use Lazard's projection operator.

We also compare against the NLSAT method [36]. For better comparability of the underlying algorithms, we used our own implementation of the NLSAT approach which uses the same underlying data structures as the aforementioned solvers. NLSAT uses only the CAD-based explanation backend and the NLSAT variable ordering. This uses the same projection operator as CDCAC and so has similar completeness limitations. We call this solver NLSAT and refer to [45] for a more detailed discussion of the implementation and the relation to the original NLSAT method from [36].

6.5. Absence of NuCAD

Throughout the paper we also highlighted NuCAD as a relevant competing algorithm. NuCAD is not currently implemented in SMT-RAT and so could not be included directly in the experimental evaluation. There is only a single implementation of NuCAD in existence: in the Tarski system [42]. However, Tarski only implements *open NuCAD* (full dimensional cells only) which reduces the comparability, and there are also some technical issues at the time of writing with Tarski's parsing of certain SMT-LIB benchmarks and so we are not able to report on the use of Tarski here. We hope, however, that the worked examples and discussions above made sufficiently clear the differences and advantages of the new algorithm in comparison to NuCAD.

6.6. Determinism and scrambling

We stress that the algorithms compared are all deterministic. I.e. if we ran the benchmark experiments again we would get the exact same results. Furthermore, though we did not try experimentally, we do not expect that a random scrambling of the benchmarks would cause any major change in the general observations. The only potential effect we can foresee would be to make different *heuristical* choices. For example, a renaming of the variables could cause the solver to pick a different choice in situations where the decision is based on the lexicographic order of the variables. However, if the algorithms that we compare share the same heuristics then scrambling might bring advantages or disadvantages on single problem instances, but it should not cause any fundamental changes visible on larger benchmark sets.

6.7. Experimental results

The overall performance of the solvers is summarised in Fig. 16. For each solver we show the number of solved problem instances (within the 60 second time limit), both separated into satisfiable and unsatisfiable instances and in total. Additionally, average runtimes for the solved instances are given and the final column shows the overall percentage of the dataset that was solved. The table shows that our new algorithm has led to a competitive solver. It significantly outperforms a solver based on regular CAD computation, even when it has been made incremental. Fig. 17 shows how the solvers tackled

Solver	SAT		UNSAT		Overall	
`CAD-naive`	4277	0.24 s	3407	0.95 s	7684	66.9%
`CAD-full`	4309	0.20 s	4207	1.50 s	8516	74.1%
`CDCAC`	4365	0.50 s	4373	1.20 s	8738	76.1%
`NLSAT`	4521	0.40 s	4409	1.32 s	8930	77.7%

Fig. 16. Experimental results for different CAD-based solvers.

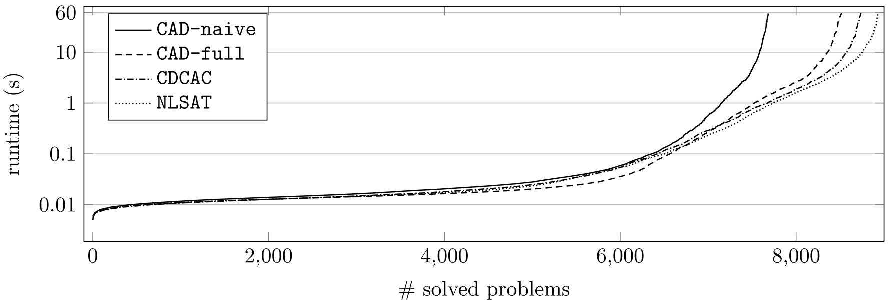


Fig. 17. Graphical depiction of the experimental results for different CAD-based solvers.

instances by their runtime, demonstrating that on trivial examples the regular CAD actually outperforms the others, but for more complex examples the savings of the adapted solvers outweigh their overheads.

We note that although we use the same SAT solver for `CAD-naive`, `CAD-full` and `CDCAC`, the overall SMT solver is not guaranteed to behave the same, i.e. make the same sequence of theory calls. This is because the theory solvers may construct different infeasible subsets which guide the SAT solver in different ways. Hence the difference seen in Fig. 16 may not be purely due to improved performance within the theory solver. Our prior experience with this issue from [46,35] indicates that the different subsets usually have only a negligible influence on the overall solver performance.

6.8. Comparison with NLSAT in experimental results

We acknowledge that our new algorithm is outperformed by the NLSAT-based solver. We note that this out-performance is greater on SAT instances than UNSAT, suggesting that the new approach may be particularly useful for proving UNSAT. We are optimistic that `NLSAT` could be beaten following improvements to our implementation. First, implementing incrementality increased the number of problems the CAD solver could tackle by 7% of the dataset, so gains to `CDCAC` from incrementality may be particularly fruitful. Second, we have yet to perform optimisation on things like the variable ordering choice for `CDCAC`, which is known to have a great effect on CAD-based technology (see e.g. [47]) and has been already performed for this NLSAT-based solver to give significant improvements as detailed in [45].

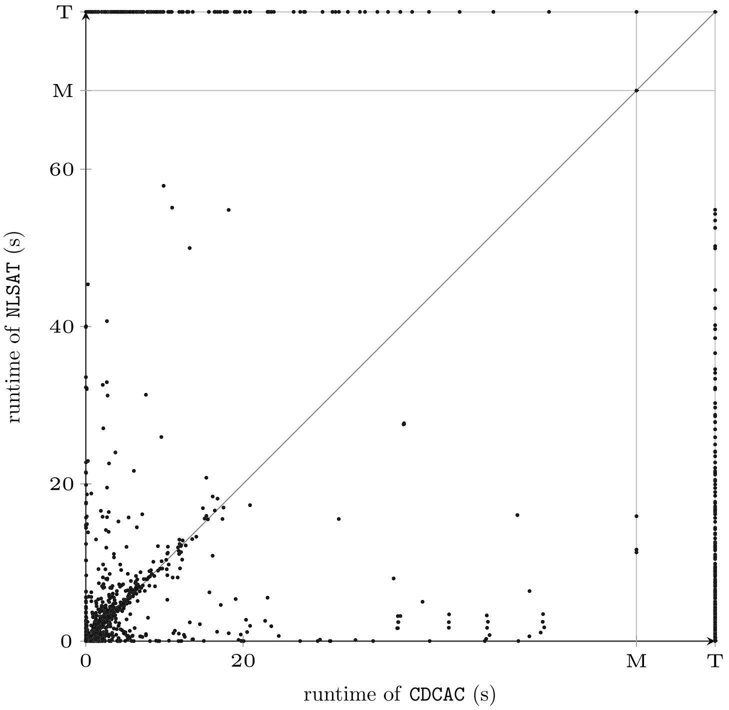
A more detailed comparison of `CDCAC` and `NLSAT` is presented in Fig. 18. In Fig. 18a we show a direct comparison of the two solvers, where every dot represents one problem instance. The cluster of points in the lower left corner represent the many *easy* problem instances that are solved by both solvers and thus is not so important for the comparison. More interesting is that the solvers perform very differently on a substantial amount of harder problem instances. Note that almost no problem instance is *solved by both* solvers after more than 20 seconds, but that both solvers can solve such problem instances *that the other cannot*. There are 555 problems on which `CDCAC` times out but NLSAT completes, and 358 problems for which NLSAT times out and `CDCAC` completes.

Figs. 18b and 18c separates the number of problem instances according to how much slower or faster `CDCAC` is when compared to `NLSAT`. The data in Fig. 18b are for problems where at least one solver took more than 10*ms*, and the data in Fig. 18c where at least one took above 100*ms* (so these exclude instances that the solvers agree are trivial). Note that the rows are cumulative (i.e. instances which are 2× faster are also 1.1× faster). Fig. 18b states that `NLSAT` was actually slower than `CDCAC` more often than it was faster! This does not contradict the data in Fig. 16 however, as `NLSAT` still completed in many of the cases it was slower while `CDCAC` would time out more often when slower. We looked for whether the stronger performance of a solver correlated to different origins of the problem instances but did not find strong evidence of this.

Altogether, the data from Fig. 18 indicates that `CDCAC` and `NLSAT` have substantially different strengths and weaknesses. There should hence be benefit in combining them, for example in a portfolio solver [48].

7. Conclusions

We have presented a new algorithm for solving SMT problems in non-linear real arithmetic based on the construction of cylindrical algebraic coverings to learn from conflicts. We demonstrated the approach with detailed worked examples and the performance of our implementation on a large dataset.



(a) Scatter plot of CDCAC and NLSAT. Here *M* and *T* signify respectively that the 4 GB memory limit or 60 s time limit was met for an instance.

# Problems for which CDCAC is		
	SAT	UNSAT
at least 10× slower	422	322
at least 2× slower	822	628
at least 1.1× slower	1907	1374
similar	922	1524
at least 1.1× faster	1778	1743
at least 2× faster	640	566
at least 10× faster	197	285

(b) Comparison with NLSAT for runtimes ≥ 10 ms.

# Problems for which CDCAC is		
	SAT	UNSAT
at least 10× slower	422	322
at least 2× slower	599	546
at least 1.1× slower	629	704
similar	28	823
at least 1.1× faster	395	808
at least 2× faster	356	430
at least 10× faster	197	284

(c) Comparison with NLSAT for runtimes ≥ 100 ms.

Fig. 18. Direct comparison of CDCAC and NLSAT.

7.1. Comparison with traditional CAD

There are clear advantages over a more traditional CAD approach (even one that is made incremental for the SMT context). The new algorithm requires fewer projection polynomials, fewer cells, and fewer sample points to determine unsatisfiability in a given cylinder. It can even sometimes avoid the calculation of projection polynomials entirely, when they represent combinations of polynomials not required to determine unsatisfiability. These advantages manifested in substantially more examples solved in the experiments.

7.2. Comparison with NuCAD

The worked examples made clear how the new algorithm more effectively learns from conflicts than the NuCAD approach [27] applied to our setting. Although NuCAD also saves in projection and cell construction when compared to traditional CAD, its savings are not maximised as the search is less directed. The conflicts of separate cells in NuCAD are never combined and so the search within NuCAD is not guided to the same extent. However, NuCAD is able to work in a larger QE context, while this new algorithm is applicable only in a restricted setting for checking the consistency of polynomial constraint sets, as outlined in Section 2.3.

7.3. Comparison with NLSAT

Our ideas are closest to those of the NLSAT method [36] which likewise builds partial samples and when these cannot be extended explains the conflict locally and generalises from the point. For several of the worked examples the new algorithm and NLSAT performed similarly, with any differences due to luck or heuristic choices not fundamental to the theory. However, the methods are very different in their presentation and computational flow. There are a number of potential advantages of the new approach compared to NLSAT.

First, on a practical note, our method is SMT compliant and thus allows for a (relatively) easy integration with any existing SMT solver. NLSAT on the other hand is usually separate from the regular SMT solving engine which makes it hard to combine with other approaches, e.g. for theory combination. Second, our approach can avoid some projections performed by NLSAT in cases where a change in the sample in one dimension will find a satisfying witness (see the example in Section 5.3).

Most fundamentally, our approach keeps the theory reasoning away from the SAT solving engine while NLSAT burdens it with many lemmas for the (intermediate) theory conflicts. For a problem instance that is fully conjunctive our method may determine the satisfiability alone while NLSAT will still require a SAT solver. It is possible that NLSAT may cause the Boolean reasoning to slow down or use excessive amounts of memory by forcing it to deal with a large number of clauses. Further, once we move on to a different area of the solution space most of the previously generated lemmas are usually irrelevant, but their literals may still leak into the theory reasoning in NLSAT. An interesting claim about NLSAT is that explanations for a certain theory assignment can sometimes be reused in another context [49]. While this may be the case, our own experiments suggest that this impact is rather limited. Our new algorithm can not reuse unsatisfiable intervals in such a way, although we can for example retain projection factors to avoid their recomputation.

The experimental results have the NLSAT based solver performing best overall, but our new algorithm was already competitive with only a limited implementation,[12] and outperformed NLSAT on a substantial quantity of problems instances as detailed in Fig. 18.

7.4. Future work

Our next step will be to optimise the implementation of the new algorithm, in particular to have it work incrementally. We will also consider how to optimise some of the heuristic choices required for the algorithm. This most notably includes the variable ordering, critical for the performance of all CAD technology.[13] The new algorithm also introduces its own choices which could be made heuristically, such as whether to remove redundant intervals (Section 4.5) and more generally which intervals to select for a covering.

Another imminent step is to edit the algorithms and arguments so they use Lazard projection theory [40] in place of McCallum theory, in order to avoid the completeness issues of Section 4.4.6. At the time of writing this was not undertaken as the additional trailing coefficients of Lazard projection made it more expensive but we note the recent work [53] which addresses when and how these coefficients can be removed.

Future software development will include the use of the new algorithm and NLSAT together, since the experiments showed significant data sets where one substantially outperformed the other. It will be necessary to develop a heuristic to analyse an instance and pick the appropriate solver.

As an application of the new algorithm, we are keen to explore whether it could provide guidance to formal proof systems. Recent preliminary work in [54] suggests that a trace of the new algorithm provides a sequence of arguments closer to human intuition than those of the alternatives.

Other open questions include the complexity of our new algorithm, and whether more general extensions are possible. Can the approach outlined here be adapted to more general quantifier elimination? Can we go from “merge intervals for existential quantifier” to “merge intervals for universal quantifier”? Can we use cylindrical coverings to study all first order formulae (instead of just sets of constraints representing conjunction) and thus remove the need for the SAT solver entirely?

Declaration of competing interest

None.

Acknowledgements

We thank all three anonymous reviewers whose comments helped improve this paper. We also thank Jasper Nalbach for his feedback on the paper.

The work was not funded directly by the SC^2 project [32] but the authors were brought together by it. Matthew England was supported by EPSRC Project EP/T015748/1 (*Pushing Back the Doubly-Exponential Wall of Cylindrical Algebraic Decomposition*).

References

[1] T. Sturm, New domains for applied quantifier elimination, in: Proceedings of the 9th International Workshop on Computer Algebra in Scientific Computing, CASC 2006, in: LNCS, vol. 4194, Springer, 2006, pp. 295–301, https://doi.org/10.1007/11870814_25.

[2] R. Bradford, J.H. Davenport, M. England, H. Errami, V. Gerdt, D. Grigoriev, C. Hoyt, M. Košta, O. Radulescu, T. Sturm, A. Weber, A case study on the parametric occurrence of multiple steady states, in: Proceedings of the 42nd International Symposium on Symbolic and Algebraic Computation, ISSAC 2017, ACM, 2017, pp. 45–52, https://doi.org/10.1145/3087604.3087622.

[3] C.B. Mulligan, J.H. Davenport, M. England, TheoryGuru: a Mathematica package to apply quantifier elimination technology to economics, in: Proceedings of the 6th International Congress on Mathematical Software, ICMS 2018, in: LNCS, vol. 10931, Springer, 2018, pp. 369–378, https://doi.org/10.1007/978-3-319-96418-8_44.

[4] N.H. Arai, T. Matsuzaki, H. Iwane, H. Anai, Mathematics by machine, in: Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation, ISSAC 2014, ACM, 2014, pp. 1–8, https://doi.org/10.1145/2608628.2627488.

[5] A. Tarski, A Decision Method for Elementary Algebra and Geometry, 2nd ed., Univ. Cal. Press, 1951, reprinted in: B.F. Caviness, in: J.R. Johnson (Eds.), Quantifier Elimination and Cylindrical Algebraic Decomposition, Springer, 1998, pp. 24–84, https://doi.org/10.1007/978-3-7091-9459-1_3.

[12] Referring to the lack of incrementality which meant recomputed data for different theory solver calls.

[13] We note recent work on the intersection of variable ordering between the theory and Boolean solvers [45] and the use of machine learning to make CAD variable ordering decisions [50,47,51,52].

[6] G.E. Collins, Quantifier elimination for real closed fields by cylindrical algebraic decomposition, in: B.F. Caviness (Ed.), Proceedings of the 2nd GI Conference on Automata Theory & Formal Languages, in: LNCS, vol. 33, Springer, 1975, pp. 134–183, reprinted in: J.R. Johnson (Ed.), in: Quantifier Elimination and Cylindrical Algebraic Decomposition, Springer, 1998, pp. 85–121, https://doi.org/10.1007/978-3-7091-9459-1_4.

[7] M. England, R. Bradford, J.H. Davenport, Improving the use of equational constraints in cylindrical algebraic decomposition, in: Proceedings of the 40th International Symposium on Symbolic and Algebraic Computation, ISSAC 2015, ACM, 2015, pp. 165–172, https://doi.org/10.1145/2755996.2756678.

[8] M. England, J.H. Davenport, The complexity of cylindrical algebraic decomposition with respect to polynomial degree, in: Proceedings of the 18th International Workshop on Computer Algebra in Scientific Computing, CASC 2016, in: LNCS, vol. 9890, Springer, 2016, pp. 172–192, https://doi.org/10.1007/978-3-319-45641-6_12.

[9] M. England, R. Bradford, J. Davenport, Cylindrical algebraic decomposition with equational constraints, in: Davenport, et al. (Eds.), [55], 2019, pp. 38–71 https://doi.org/10.1016/j.jsc.2019.07.019.

[10] C.W. Brown, J.H. Davenport, The complexity of quantifier elimination and cylindrical algebraic decomposition, in: Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation, ISSAC 2007, ACM, 2007, pp. 54–60, https://doi.org/10.1145/1277548.1277557.

[11] S. Basu, R. Pollack, M.F. Roy, Algorithms in Real Algebraic Geometry, Algorithms and Computations in Mathematics, Springer, 2006, https://link.springer.com/book/10.1007/3-540-33099-2.

[12] H. Hong, Comparison of several decision algorithms for the existential theory of the reals, Tech. rep., RISC, Linz (Report Number 91-41), 1991, ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1991/91-41.ps.gz.

[13] A. Cimatti, A. Griggio, A. Irfan, M. Roveri, R. Sebastiani, Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions, ACM Trans. Comput. Log. 19 (3) (2018) 19:1–19:52, https://doi.org/10.1145/3230639.

[14] A. Maréchal, A. Fouilhé, T. King, D. Monniaux, M. Périn, Polyhedral approximation of multivariate polynomials using Handelman's theorem, in: Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2016, in: LNCS, vol. 9583, Springer, 2016, pp. 166–184, https://doi.org/10.1007/978-3-662-49122-5_8.

[15] M. Fränzle, C. Herde, T. Teige, S. Ratschan, T. Schubert, Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure, J. Satisf. Boolean Model. Comput. 1 (2007) 209–236, https://doi.org/10.3233/SAT190012.

[16] V.X. Tung, T. Van Khanh, M. Ogawa, raSAT: an SMT solver for polynomial constraints, Form. Methods Syst. Des. 51 (3) (2017) 462–499, https://doi.org/10.1007/s10703-017-0284-9.

[17] V. Weispfenning, Quantifier elimination for real algebra — the quadratic case and beyond, Appl. Algebra Eng. Commun. Comput. 8 (2) (1997) 85–101, https://doi.org/10.1007/s002000050055.

[18] F. Corzilius, E. Ábrahám, Virtual substitution for SMT solving, in: Proceedings of the 18th International Symposium on Fundamentals of Computation Theory, FCT 2011, in: LNCS, vol. 6914, Springer, 2011, pp. 360–371, https://doi.org/10.1007/978-3-642-22953-4_31.

[19] P. Fontaine, M. Ogawa, T. Sturm, X.T. Vu, Subtropical satisfiability, in: Proceedings of the 11th International Symposium on Frontiers of Combining Systems, FroCoS 2017, in: LNCS, vol. 10483, Springer, 2017, pp. 189–206, https://doi.org/10.1007/978-3-319-66167-4_11.

[20] J. Abbott, A.M. Bigatti, CoCoALib: a C++ library for computations in commutative algebra .. and beyond, in: Proceedings of the 3rd International Congress on Mathematical Software, ICMS 2010, in: LNCS, vol. 6327, Springer, 2010, pp. 73–76, https://doi.org/10.1007/978-3-642-15582-6_15.

[21] D. Wilson, R. Bradford, J.H. Davenport, M. England, Cylindrical algebraic sub-decompositions, Math. Comput. Sci. 8 (2014) 263–288, https://doi.org/10.1007/s11786-014-0191-z.

[22] G.E. Collins, H. Hong, Partial cylindrical algebraic decomposition for quantifier elimination, J. Symb. Comput. 12 (30) (1991) 299–328, https://doi.org/10.1016/S0747-7171(08)80152-6.

[23] G.E. Collins, Quantifier elimination by cylindrical algebraic decomposition — twenty years of progress, in: B.F. Caviness, J.R. Johnson (Eds.), Quantifier Elimination and Cylindrical Algebraic Decomposition, Springer, 1998, pp. 8–23, https://doi.org/10.1007/978-3-7091-9459-1_2.

[24] S. McCallum, On projection in CAD-based quantifier elimination with equational constraint, in: Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation, ISSAC 1999, ACM, 1999, pp. 145–149, https://doi.org/10.1145/309831.309892.

[25] R.J. Bradford, J.H. Davenport, M. England, S. McCallum, D.J. Wilson, Truth table invariant cylindrical algebraic decomposition, J. Symb. Comput. 76 (2016) 1–35, https://doi.org/10.1016/j.jsc.2015.11.002.

[26] D. Jovanović, Solving nonlinear integer arithmetic with MCSAT, in: Proceedings of the 18th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2017, in: LNCS, vol. 10145, Springer, 2017, pp. 330–346, https://doi.org/10.1007/978-3-319-52234-0_18.

[27] C.W. Brown, Open non-uniform cylindrical algebraic decompositions, in: Proceedings of the 40th International Symposium on Symbolic and Algebraic Computation, ISSAC 2015, ACM, 2015, pp. 85–92, https://doi.org/10.1145/2755996.2756654.

[28] C.W. Barrett, P. Fontaine, C. Tinelli, The satisfiability modulo theories library (SMT-LIB), http://www.smt-lib.org, 2016.

[29] A. Seidenberg, A new decision method for elementary algebra, Ann. Math. 60 (2) (1954) 365–374, https://doi.org/10.2307/1969640.

[30] C. Barrett, R. Sebastiani, S. Seshia, C. Tinelli, Satisfiability modulo theories, in: A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, in: Frontiers in Artificial Intelligence and Applications, vol. 185, IOS Press, 2009, pp. 825–885, https://doi.org/10.3233/978-1-58603-929-5-825.

[31] E. Ábrahám, Building bridges between symbolic computation and satisfiability checking, in: Proceedings of the 40th International Symposium on Symbolic and Algebraic Computation, ISSAC 2015, ACM, 2015, pp. 1–6, https://doi.org/10.1145/2755996.2756636.

[32] E. Ábrahám, J. Abbott, B. Becker, A.M. Bigatti, M. Brain, B. Buchberger, A. Cimatti, J.H. Davenport, M. England, P. Fontaine, S. Forrest, A. Griggio, D. Kroening, W.M. Seiler, T. Sturm, SC^2: satisfiability checking meets symbolic computation, in: Proceedings of the 9th International Conference on Intelligent Computer Mathematics, CICM 2016, in: LNCS, vol. 9791, Springer, 2016, pp. 28–43, https://doi.org/10.1007/978-3-319-42547-4_3.

[33] F. Corzilius, G. Kremer, S. Junges, S. Schupp, E. Ábrahám, SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving, in: Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing, SAT 2015, in: LNCS, vol. 9340, Springer, 2015, pp. 360–368, https://doi.org/10.1007/978-3-319-24318-4_26.

[34] P. Fontaine, M. Ogawa, T. Sturm, V. Khanh To, X. Tung Vu, Wrapping computer algebra is surprisingly successful for non-linear SMT, in: Proceedings of the 3rd Workshop on Satisfiability Checking and Symbolic Computation (SC^2 2018), no. 2189 in CEUR Workshop Proceedings, 2018, pp. 110–117, http://ceur-ws.org/Vol-2189/.

[35] G. Kremer, E. Ábrahám, Fully incremental CAD, in: Davenport, et al. (Eds.), [55], 2019, pp. 11–37 https://doi.org/10.1016/j.jsc.2019.07.018.

[36] D. Jovanović, L. de Moura, Solving non-linear arithmetic, in: Proceedings of the 6th International Joint Conference on Automated Reasoning, IJCAR 2012, in: LNCS, vol. 7364, Springer, 2012, pp. 339–354, https://doi.org/10.1007/978-3-642-31365-3_27.

[37] M. Jaroschek, P.F. Dobaland, P. Fontaine, Adapting real quantifier elimination methods for conflict set computation, in: Proceedings of the 10th International Symposium on Frontiers of Combining Systems, FroCoS 2015, in: LNCS, vol. 9322, Springer, 2015, pp. 151–166, https://doi.org/10.1007/978-3-319-24246-0_10.

[38] S. McCallum, An improved projection operation for cylindrical algebraic decomposition, in: B. Caviness, J. Johnson (Eds.), Quantifier Elimination and Cylindrical Algebraic Decomposition, in: Texts & Monographs in Symbolic Computation, Springer-Verlag, 1998, pp. 242–268, https://doi.org/10.1007/978-3-7091-9459-1_12.

[39] H. Hong, An improvement of the projection operator in cylindrical algebraic decomposition, in: Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC 1990, ACM, 1990, pp. 261–264, https://doi.org/10.1145/96877.96943.

[40] S. McCallum, A. Parusińiski, L. Paunescu, Validity proof of Lazard's method for CAD construction, J. Symb. Comput. 92 (2019) 52–69, https://doi.org/10.1016/j.jsc.2017.12.002.

[41] D. Lazard, An improved projection for cylindrical algebraic decomposition, in: C. Bajaj (Ed.), Algebraic Geometry and Its Applications: Collections of Papers from Abhyankar's 60th Birthday Conference, Springer, Berlin, 1994, pp. 467–476, https://doi.org/10.1007/978-1-4612-2628-4_29.

[42] C. Brown, Projection and quantifier elimination using non-uniform cylindrical algebraic decomposition, in: Proceedings of the 42nd International Symposium on Symbolic and Algebraic Computation, ISSAC 2017, ACM, 2017, pp. 53–60, https://doi.org/10.1145/3087604.3087651.

[43] C. Chen, M. Moreno Maza, Cylindrical algebraic decomposition in the RegularChains library, in: H. Hong, C. Yap (Eds.), Mathematical Software – ICMS 2014, in: LNCS, vol. 8592, Springer, Heidelberg, 2014, pp. 425–433, https://doi.org/10.1007/978-3-662-44199-2_65.

[44] H. Franzen, Conflict driven cylindrical algebraic coverings for nonlinear arithmetic in SMT solving, Master's thesis, RWTH Aachen University, 2020, https://ths.rwth-aachen.de/wp-content/uploads/sites/4/franzen_master.pdf.

[45] J. Nalbach, G. Kremer, E. Ábrahám, On variable orderings in MCSAT for non-linear real arithmetic, in: Satisfiability Checking and Symbolic Computation, in: CEUR Workshop Proceedings, vol. 2460, 2019, pp. 6:1–6:7, http://ceur-ws.org/Vol-2460/paper5.pdf.

[46] W. Hentze, Computing minimal infeasible subsets for the cylindrical algebraic decomposition, Bachelor's thesis, RWTH Aachen University, 2017, https://ths.rwth-aachen.de/wp-content/uploads/sites/4/teaching/theses/hentze_bachelor.pdf.

[47] M. England, D. Florescu, Comparing machine learning models to choose the variable ordering for cylindrical algebraic decomposition, in: C. Kaliszyk, E. Brady, A. Kohlhase, C. Sacerdoti (Eds.), Intelligent Computer Mathematics, in: LNCS, vol. 11617, Springer International Publishing, 2019, pp. 93–108, https://doi.org/10.1007/978-3-030-23250-4_7.

[48] L. Xu, F. Hutter, H. Hoos, K. Leyton-Brown, SATzilla: portfolio-based algorithm selection for SAT, J. Artif. Intell. Res. 32 (2008) 565–606, https://doi.org/10.1613/jair.2490.

[49] L. de Moura, D. Jovanović, A model-constructing satisfiability calculus, in: Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2013, in: LNCS, vol. 7737, Springer, 2013, pp. 1–12, https://doi.org/10.1007/978-3-642-35873-9_1.

[50] Z. Huang, M. England, D. Wilson, J. Bridge, J. Davenport, L. Paulson, Using machine learning to improve cylindrical algebraic decomposition, Math. Comput. Sci. 13 (4) (2019) 461–488, https://doi.org/10.1007/s11786-019-00394-8.

[51] D. Florescu, M. England, Algorithmically generating new algebraic features of polynomial systems for machine learning, in: J. Abbott, A. Griggio (Eds.), Proceedings of the 4th Workshop on Satisfiability Checking and Symbolic Computation (SC^2 2019), no. 2460 in CEUR Workshop Proceedings, 2019, pp. 4:1–4:12, http://ceur-ws.org/Vol-2460/, 2019.

[52] D. Florescu, M. England, Improved cross-validation for classifiers that make algorithmic choices to minimise runtime without compromising output correctness, in: D. Slamanig, E. Tsigaridas, Z. Zafeirakopoulos (Eds.), Mathematical Aspects of Computer and Information Sciences, in: LNCS, vol. 11989, Springer International Publishing, 2019, pp. 341–356, https://doi.org/10.1007/978-3-030-43120-4_27.

[53] C. Brown, S. McCallum, Enhancements to Lazard's method for cylindrical algebraic decomposition, in: F. Boulier, M. England, T. Sadykov, E. Vorozhtsov (Eds.), Computer Algebra in Scientific Computing, in: Lecture Notes in Computer Science, vol. 12291, Springer International Publishing, 2020, pp. 129–149, https://doi.org/10.1007/978-3-030-60026-6_8.

[54] E. Ábrahám, J. Davenport, M. England, G. Kremer, Z. Tonks, New opportunities for the formal proof of computational real geometry?, in: Proceedings of the 5th Workshop on Satisfiability Checking and Symbolic Computation (SC^2 2020), no. 2752 in CEUR Workshop Proceedings, 2020, pp. 178–188, http://ceur-ws.org/Vol-2752/.

[55] J. Davenport, M. England, A. Griggio, T. Sturm, C. Tinelli (Eds.), Symbolic computation and satisfiability checking: Special Issue, J. Symb. Comput. 100 (2020), https://doi.org/10.1016/j.jsc.2019.07.017.