Implementing arithmetic over algebraic numbers
A tutorial for Lazard's lifting scheme in CAD

Gereon Kremer
Stanford University
Stanford, United States
gkremer@cs.stanford.edu

Jens Brandt
RWTH Aachen University
Aachen, Germany
jens.brandt@rwth-aachen.de

***Abstract*—Implementing techniques from computer algebra often requires a multitude of foundational algorithms that are neither easy to understand nor to implement. Despite great interest from other communities, the difficulty to implement novel techniques from computer algebra proves to be a significant hindrance, especially, when a modern computer algebra system cannot be used.**

We tackle cylindrical algebraic decomposition (CAD) as one such example. CAD can be, for example, applied in satisfiability modulo theories solving for nonlinear real arithmetic. However, a recent advance in CAD, the Lazard's lifting scheme, requires additional algebraic techniques that are neither available in these solvers, nor as stand-alone libraries. We close this gap by showing how to use the CoCoALib library to implement Lazard's lifting scheme outside of a modern computer algebra system like Maple.

***Keywords*-computer algebra; algebraic numbers; cylindrical algebraic decomposition; Lazard lifting; satisfiability modulo theories; CoCoALib**

I. Introduction

Among the many theories that satisfiability modulo theories solvers (SMT) tackle, nonlinear real arithmetic is arguably one of the most difficult ones, at least from a theoretical point of view. Its mathematical counterpart, the *theory of the reals*, has seen a lot of results, though only few made it into practically useful tools.

While the theory of the reals is decidable in a constructive manner, the first procedure by Tarski exhibits non-elementary runtime complexity [28]. Other approaches exist [12], [23], [5], but most have never been implemented [13]. To the best of our knowledge, the CAD approach is the only complete decision procedure used in practice and it has taken about a decade to be implemented [10], [11]. Once CAD had proven to be sufficiently efficient in practice, it has been further improved, extended, and adapted as well as implemented in a variety of tools, among others, cvc5, Maple [8], Mathematica [27], QEPCAD and QEPCAD B [6], SMT-RAT [17], and yices and z3 [14].

Given the overall complexity of both, the CAD theory and the algorithms that are foundational to it, most literature on CAD assumes a significant amount of algebraic machinery. Constructing a CAD builds on the computation of subresultant coefficients, (multivariate) polynomial greatest common divisors, (multivariate) square-free factorization, and originally real root isolation of bivariate polynomials over a real algebraic number as well as the reduction of a multiple real algebraic extension to a simple extension [10]. While Collins gives some hints and references on how to implement these, later literature assumes that an implementation of these methods is available [10].

Alas, this is not always the case: some computer algebra systems have commercial licenses which is often not practicable; many lack certain parts of the required algorithms; some cannot be used easily as libraries without a major impact on the implementation architecture; and some are no longer maintained. Therefore, most SMT solvers use their own implementations of the algebraic subroutines: SMT-RAT relies on CArL (and partially CoCoALib [1]) while cvc5, yices, and z3 use libpoly [15]. Unfortunately, most SMT solver researchers only have a limited background on computer algebra, and thus implementing new theoretic results is a significant challenge.

Contribution: This work discusses an implementation of one of the most important new developments in the CAD community: Lazard's lifting scheme [19]. It eliminates polynomial factors that vanish over the current assignment, which is required for using Lazard's projection operator in a sound way. In particular, we show how the concise mathematical algorithm can be implemented based on CoCoALib [1] that is used in cvc5, the successor to CVC4 [4]. We evaluate our implementation within cvc5 and verify that using a sound implementation of Lazard's lifting scheme has no negative impact on the practical performance.

The level of abstraction aims to be understandable given a limited amount of knowledge in computer algebra. However, we assume some familiarity with CAD and its various subroutines like projection and lifting.

II. Lazard's lifting scheme

The fundamental task of the lifting procedure in CAD is to construct $(n+1)$-dimensional sample points from an n-dimensional real algebraic sample point $a \in \overline{\mathbb{Q}}^n$ and a polynomial $q \in \mathbb{Q}[x_0, \dots, x_n]$. Conceptually, lifting works by substituting a into q to obtain a univariate polynomial in x_n and then isolate its real roots. Lazard's lifting scheme

enhances this simple procedure by eliminating certain factors of q that make q vanish during lifting: assume $q = f \cdot r$ where f is a polynomial that vanishes identically over a, then the real roots from r are lost and cannot be used to construct $(n+1)$-dimensional sample points over a.

The core idea of Lazard's lifting scheme is to process the sample point a dimension-wise and, before substituting the ith value into q, check whether q would vanish after this substitution [19]. If so, we conclude that $(x_i - a_i)$ divides q and we set q to $q/(x_i - a_i)$, i.e.

for $i := 0$ **to** $n-1$ **do**
  $v_i := \arg\max_{v \in \mathbb{Z}} (x_i - a_i)^v$ divides q
  $q := q/(x_i - a_i)^{v_i}$
  $q := subst(a_i, x_i, q)$
isolate real roots of q (now univariate in x_n)

The real roots of the resulting polynomial q are then used to construct sample points over a in $n+1$ dimensions.

III. Algebra in practice

While Lazard's lifting scheme looks fairly innocent, it has a subtle issue: it assumes that real algebraic numbers are seamlessly integrated in the polynomial arithmetic, otherwise substituting a_i for x_i may not be possible. Mathematically speaking, it assumes that we not only have polynomial arithmetic over the rational field but also over extensions of it. Note that the issue is not with implementing polynomial arithmetic over these extension fields, as a proper implementation of these extension fields' operations is not trivial in itself. We refer to [16, section 2.5] for a more thorough discussion of the difficulties.

One important insight is that $K(a_i)$, the field extension of K with a_i, for some field K and a real algebraic number a_i, is isomorphic to the quotient ring $K[x_i]/\langle p_i \rangle$, for $p_i \in K[x_i]$ the minimal polynomial of a_i over K. Whenever we go from K to $K(a_i)$, we thus construct $K[x_i]/\langle p_i \rangle$ and map all relevant polynomials into this quotient ring. While this does not actually substitute a_i into the polynomial q, it takes care of all cancellations (c.f. [16, section 2.5.2.1]). The naive recursive approach may fail, though: while we usually have the minimal polynomial $p_i \in \mathbb{Q}[x_i]$ for a_i over $\mathbb{Q}$, Lazard's construction requires the minimal polynomial for a_i over the field K. The remedy is to factor p_i over K and take the unique factor that vanishes over a_i to obtain the minimal polynomial $p_i^* \in K[x_i]$ for a_i.

While this approach takes care of most issues with Lazard's lifting scheme, it ultimately generates a polynomial q that was stripped of all factors that vanish over a, but may still contain the variables $x_0, \ldots, x_{n-1}$. We can use either Gröbner bases with an appropriate term ordering or iterated resultants to obtain a univariate polynomial $q^* \in \mathbb{Q}[x_n]$ such that the real roots of q^* are a superset of the real roots of q over a, which is sufficient for our application (c.f. [16, section 2.5.2.2]).

IV. Related work

Real algebraic numbers are a fundamental part of many methods in computer algebra and related fields, and have been studied extensively in the past. Unfortunately, such research usually either takes real algebraic numbers for granted or focusses on specific aspects of their implementation.

In many cases, real algebraic numbers are merely a by-product of constructing extension fields or algebraic closures, and their practical implementation is not discussed [22], [3], [29]. Literature about more complex algorithms often assumes an existing implementation of real algebraic numbers [2], [5], [8]. If they are the research subject themselves, the discussion mostly focusses on very specific operations and is never sufficient to allow for a proper reimplementation [24], [9].

Inversely, implementations either exist in commercial products (like Maple or Mathematica) or as third-party extensions that are usually meant for a very specific purpose, are no longer maintained, or lack proper documentation [25], [21]. We thus observe a shortage of literature about real algebraic numbers that is constructive and both detailed and exhaustive enough for an effective and reasonably efficient implementation of real algebraic numbers.

Collins and Loos propose to reduce a multiple extension to a simple extension and then perform all arithmetic operations in this simple extension [10], [20]. Elsewhere, individual operations on real algebraic numbers for multiple extensions are discussed [18], [26].

V. Algebraic setup

We first describe how the ideas from section III can be implemented. A major part of this consists in figuring out which computations are performed in which polynomial ring, and how to properly map polynomials from one ring into the other. We also use a subroutine that needs to deal with real algebraic numbers in a numeric manner (c.f. [16, section 2.5]) by checking which polynomial factor evaluates to zero over the current assignment. This computation needs to be performed outside of the presented framework, most likely also outside of CoCoALib, e.g., using libpoly.

We start with a number of polynomial rings and mappings between these, and then give a detailed description of a C++ implementation of the procedure from section II. Let $n \in \mathbb{N}$, $x_0, \ldots, x_n$ be real variables, $a_0, \ldots, a_{n-1} \in \overline{\mathbb{Q}}$ real algebraic numbers with minimal polynomials $p_0, \ldots, p_{n-1}$ over $\mathbb{Q}$. We define

$$K_0 = \mathbb{Q}, \qquad R_i = K_i[x_i],$$
$$K_{i+1} = R_i/\langle p_i^* \rangle, \qquad J_i = K_i[x_i, \ldots, x_n],$$

with p_i^* minimal polynomials for a_i over K_i.

We furthermore consider the K_i-linear mappings: next to the quotient map $R_i \hookrightarrow K_{i+1}$, we have the epimorphism $\varphi_i : J_i \to J_{i+1}$ and its right inverse $C_i : J_{i+1} \to J_i$ that

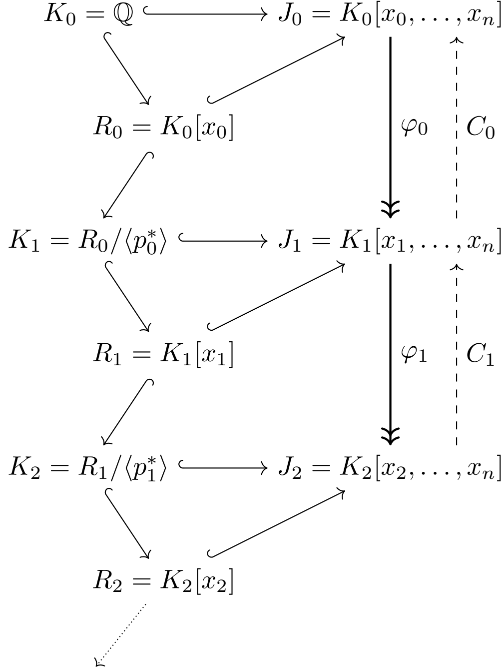

Figure 1. Schematic overview

maps each polynomial to the canonical representative of its preimage under φ_i, i.e., the polynomial with smallest degree in x_i. A schematic overview of this setup is given in figure 1 and we conclude this section by an example.

Example: Let $n = 1$, x_0, x_1 be real variables and $a_0 = \sqrt{2}$ a real algebraic number with minimal polynomial $p_0(x_0) = x_0^2 - 2$ over $\mathbb{Q}$. We obtain

$$K_0 = \mathbb{Q}, \quad R_0 = \mathbb{Q}[x_0], \quad J_0 = \mathbb{Q}[x_0, x_1],$$
$$K_1 = \mathbb{Q}[x_0]/\langle x_0^2 - 2\rangle \cong \mathbb{Q}(\sqrt{2}), \text{ and}$$
$$R_1 = J_1 = (\mathbb{Q}[x_0]/\langle x_0^2 - 2\rangle)[x_1] \cong \mathbb{Q}(\sqrt{2})[x_1].$$

In this example, the epimorphism φ_0 maps $x_0 \in J_0$ to $x_0 + \langle x_0^2 - 2\rangle \in J_1$. This operation is isomorphic to the substitution of x_0 by $\sqrt{2}$, e.g., $\varphi_0((x_0^2 - 2)x_1) = 0$.

VI. Implementation

In this section, we use CoCoALib to present an implementation of the construction from section V. The code samples are almost fully functional C++ source code snippets; the only missing parts are those that need to be computed outside of CoCoALib, e.g., numeric computations involving real algebraic assignments.

The implementation is separated into four parts: creating the extension fields K_i and polynomial rings R_i over them, creating the polynomial rings J_i and mappings, implementing the mapping C_i, and eliminating vanishing factors according to section II.

First, we create the extension fields K_i, the polynomial rings R_i and the minimal polynomials p_i^*.

```
vector<RingElem> p; // p_0 ... p_{n-1}
vector<ring> K;     // K_0 ... K_n
vector<ring> R;     // R_0 ... R_n

K[0] = RingQQ();

// assigned variables x_0,...x_{n-1}
for (size_t i = 0; i < n; ++i)
{
  R[i] = NewPolyRing(K[i],
    {NewSymbol()});
  RingElem mipo = /* from R_i */;
  auto facs = factor(mipo);
  p[i] = /* fac that vanishes */;
  K[i+1] = NewQuotientRing(R[i],
    ideal(p[i]));
}

// free variable x_n
R[n] = NewPolyRing(K[n], {NewSymbol()});
```

The variable `mipo` holds the minimal polynomial of a_i over $\mathbb{Q}$. To obtain the vanishing factor stored in `p[i]`, we need to inspect the factors in `facs.myFactors()`. These factors contain one unique element that evaluates to zero over the assignment given by $a_0, \ldots, a_i$. Note that, to simplify debugging, one can use `symbol(name)` instead of `NewSymbol()` to have proper variable names when printing CoCoALib objects.

Second, we construct the polynomial rings J_i as well as the mappings from section V. Most can be obtained directly from CoCoALib: $K_i \hookrightarrow R_i$ and $K_i \hookrightarrow J_i$ using `CoeffEmbeddingHom`, and $R_i \hookrightarrow K_{i+1}$ using `QuotientingHom`. For $R_i \hookrightarrow J_i$, we need to map x_i manually by constructing the embedding using `PolyAlgebraHom` where the single indeterminant of R_i is mapped to the first indeterminant of J_i. A similar problem arises for $\varphi_i : J_i \twoheadrightarrow J_{i+1}$ where we need to provide a mapping $K_i \hookrightarrow K_{i+1}$ and the identity of the variables $x_{i+1}, \ldots, x_n$. We store the two mappings $R_i \hookrightarrow J_i$ and $\varphi_i : J_i \twoheadrightarrow J_{i+1}$ in `RJ` and `JJ`, respectively.

```
vector<ring> J;      // J_0 ... J_n
vector<RingHom> RJ;  // R_i ↪ J_i
vector<RingHom> JJ;  // J_i ↠ J_{i+1}

// x_0...x_n, in the loop x_i...x_n
vector<symbol> syms;
for (size_t i = 0; i <= n; ++i)
{
   syms[i] = symbols(R[i])[0];
}
```

```
for (size_t i = 0; i <= n; ++i)
{
  J[i] = NewPolyRing(K[i], syms, lex);
  syms.erase(syms.begin());
  RJ[i] = PolyAlgebraHom(R[i], J[i],
    {indet(J[i], 0)});
  if (i == 0) continue;
  // K_{i-1} ↪ R_{i-1}
  auto K2R = CoeffEmbeddingHom(R[i-1]);
  // R_{i-1} ↪ K_i
  auto R2K = QuotientingHom(K[i]);
  // K_i ↪ J_i
  auto K2J = CoeffEmbeddingHom(J[i]);
  // x_{i-1}, x_i ... x_n where x_{i-1} ∈ R_{i-1}
  vector<RingElem> indets = {
    K2J(R2K(indet(R[i-1], 0))) };
  for (size_t j = 0; j < n-i; ++j)
    indets.push_back(indet(J[i], j));
  JJ[i] = PolyRingHom(J[i-1], J[i],
    R2K(K2R), indets);
}
```

Third, we provide an auxiliary function for mapping elements from J_i into J_0 via the composition $C_0 \circ \cdots \circ C_{i-1}$, compare section V. For this, it is sufficient to implement every function C_j individually by manually deconstructing $p \in J_{j+1}$ and reconstructing it in J_j. Therefore, we replace every coefficient by its canonical representative and map the kth indeterminant of J_i to the $k+1$st indeterminant of J_{i-1}. We use this auxiliary function `toJ0` in the final step of the reduction where we compute a Gröbner basis over J_0 to obtain univariate polynomials.

```
RingElem pushDown(RingElem p, size_t i)
{
  assert(owner(p) == J[i]);
  RingElem res(J[i-1]);    // 0 ∈ J_{i-1}
  for (auto it = BeginIter(p);
       !isEnded(it); ++it)
  {
    RingElem c = coeff(it);
    c = CanonicalRepr(c); // K_i → R_{i-1}
    c = RJ[i-1](c);         // R_{i-1} ↪ J_{i-1}
    auto pp = PP(it);    // power product
    vector<long> ind = IndetsIn(pp);
    for (auto k = 0; k<ind.size(); ++k)
    { // map power product J_i → J_{i-1}
      long e = exponent(pp, ind[k]);
      auto in = indet(J[i-1], ind[k]+1);
      c *= power(in, e);
    }
    res += c;
  }
  return res;
}
```

```
RingElem toJ0(RingElem p, size_t i)
{
  for (; i > 0; --i) p = pushDown(p, i);
  return p;
}
```

Fourth and last, everything is in place to eliminate the vanishing factors from the input polynomial $q^{(0)} \in J_0$ via a straight-forward implementation of the method described in section II (c.f. [19]), resulting in $q^{(n)} \in J_n$. We then map $q^{(n)}$ and all p_i into J_0 using the auxiliary function `toJ0` and compute a reduced Gröbner basis of these polynomials. Constructing J_0 with the lexicographical term ordering (c.f. [16, section 2.5.2.2]) allows us to retrieve polynomials from the Gröbner basis that are univariate in x_n and characterize the solution space of the input polynomials in x_n. Therefore, the projection of the real roots of q onto x_n is contained in the set of real roots of these univariate polynomials.

```
RingElem q = /* irreducible in J_0 */;
for (size_t i = 0; i < n; ++i)
{
   RingElem cur = RJ[i](p[i]); // p_i ∈ J_i
   while (IsDivisible(q, cur)) q /= cur;
   q = JJ[i](q);                // q ∈ J_{i+1}
}
assert(owner(q) == J.back());  // q ∈ J_n
vector<RingElem> id = { toJ0(q) };
for (size_t i = 0; i < n; ++i)
{
   id.emplace_back(toJ0(p[i]));
}
auto basis = ReducedGBasis(ideal(id));
return { /* polynomials from basis that
  are univariate in x_n */ };
```

The retrieved polynomials serve as input for real root isolation algorithms, and their real roots then are used in the subsequent lifting process of CAD.

VII. Practical efficiency

In section VI, we described a basic implementation of Lazard's lifting scheme. However, the practical efficiency can be further improved by the following techniques.

A. Reusability

We oftentimes want to process multiple polynomials over the same sample point a (c.f. the outer loop in `get_unsat_intervals` [2, algorithm 3]). The aforementioned description nicely separates the construction of all polynomial rings and mappings from the processing of a polynomial q. In this case, we can thus construct the whole algebraic setup once and execute Lazard's lifting scheme multiple times.

B. Skip levels

If the assignment a_i is an element of K_i, we can set $K_{i+1} = K_i$ and substitute x_i directly by a polynomial expression in $a_0, \dots, a_{i-1}$ with rational coefficients (c.f. [16, algorithm 2.2]), i.e., there is no need for the complex construction in section VI.

To identify such cases, it suffices to check whether the minimal polynomial of a_i over K_i is linear in x_i, i.e., in the implementation, we check whether `p[i]` is linear in x_i. In these cases, we copy `K[i]` to `K[i+1]` instead of using `NewQuotientRing` and compute the assignment for x_i as the quotient of the constant term and the leading coefficient of the polynomial `p[i]`.

However, level skipping requires several other adaptions:

- The construction of `JJ[i]` is simplified: the coefficient rings of J_{i-1} and J_i are identical and `JJ[i]` can thus be constructed as a `PolyAlgebraHom` that maps x_i to its assignment.
- When mapping the coefficients of p from K_i to J_{i-1} in the `pushDown` method, we can use `CoeffEmbeddingHom(J[i-1])` instead of obtaining the canonical representative and applying the mapping `RJ[i-1]`.
- In the actual reduction, we directly construct the (partial) evaluation homomorphism `hom` on J_i that assigns x_i to its assignment as a `PolyAlgebraHom`. We replace the divisibility check by `IsZero(hom(q))` and use `q = hom(q)` to embed q into J_{i+1}.

Please note that we could increase the number of these cases by selecting a suitable variable ordering. However, the correctness of Lazard's lifting scheme requires using the same variable ordering as the overall CAD construction (c.f. [16, section 5.3.2]).

C. Factorization

We feel that it is also worth mentioning that the discussed methods exploit polynomial factorization beyond what is necessary to identify the minimal polynomials p_i over K_i.

Note that we assume in the code the input polynomial q to be irreducible (in J_0) and returned the polynomials from the reduced Gröbner basis separately. We could accept a reducible polynomial q, and return the product of all univariate polynomials from the Gröbner basis. However, keeping these factors separate makes sense in practice as the real root isolation methods that eventually process these polynomials may be able to retrieve rational roots more often and are usually faster on polynomials of smaller degree.

Similarly, many operations on real algebraic numbers are faster if their defining polynomials have smaller degrees, at least if the implementation allows for defining polynomials that are not irreducible.

Configuration		Results		
lifting	**projection**	**sat**	**unsat**	**total**
libpoly	McCallum	5064	5378	10442
libpoly	Lazard	5062	5377	10439
Lazard	McCallum	5088	5370	10458
Lazard	Lazard	5090	5371	10461

Figure 2. Experimental results

VIII. Some experiments

We integrate our code in the implementation of cylindrical algebraic coverings [2] in cvc5 and compare four configurations that vary on the one hand in the lifting technique (c.f. `get_unsat_intervals` [2]) and on the other hand in which coefficients are included in the projection (c.f. `required_coefficients` [2]).

To isolate the real roots of a polynomial, we either use `infeasible_regions` from libpoly or a reimplementation based on Lazard's lifting as presented in section VI. For the coefficient choice, we use `required_coefficients` [2] which implements a slight refinement of McCallum's projection, or follow Brown and McCallum's refinement of Lazard's projection (c.f. [7, definition 4]).

The four configurations were tested on the SMT-LIB benchmark set for `QF_NRA`, consisting of 11552 input files, of which only 4752 ever made it to the nonlinear solver within a time limit of ten minutes. All other instances were solved by the simplex-based linear solver alone. About 99.6% of all variable assignments could be skipped as discussed in section VII-B. Only 925 instances see any non-direct assignments at all, which emphasizes the importance of this optimization. The removal of vanishing factors, the core improvement of Lazard's lifting scheme, occurs on only 664 instances where almost $750k$ vanishing factors are removed altogether.

Figure 2 shows that employing Lazard's lifting scheme solves slightly more satisfiable instances than the others while losing a few unsatisfiable ones. On commonly solved instances, there is no discernible performance difference in either computation time or memory usage. Using the smaller projection sets from [7] has no significant impact, as the two tested variants are identical on all but 52 input files, corresponding to merely 0.45%.

Note that we have not seen a single incorrect result in our experiments, although only the configurations that use Lazard's lifting scheme are technically sound. This is consistent with previous experience [16], [17]: using an incomplete combination of projection and lifting schemes can only cause soundness issues by missing sample points and incorrectly determine unsatisfiability. In practice, this is only the case when all satisfying sample points are missed in this way across all theory calls, which is highly unlikely.

These experiments thus verify, that using a proper implementation of Lazard's lifting scheme allows employing Lazard's projection operator in a sound way without negatively affecting practical performance.

IX. Conclusion

We showed how one of the most significant recent advances for CAD, Lazard's lifting scheme that allows to use Lazard's projection operator, can be implemented with reasonable effort outside of a full-fledged computer algebra system, instead using the free and open source C++ library CoCoALib.

The experiments show that for most examples the assignments allow to avoid constructing the field extensions as discussed in section VII-B. Even if they are necessary, though, they have no discernible performance impact.

References

[1] John Abbott and Anna M. Bigatti. CoCoALib: a C++ library for doing Computations in Commutative Algebra. Available at `http://cocoa.dima.unige.it`.

[2] Erika Ábrahám, James H. Davenport, Matthew England, and Gereon Kremer. 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*, 119, 2021. `doi:10.1016/j.jlamp.2020.100633`.

[3] Michael Artin. *Algebra*. Pearson, 2011.

[4] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification*, volume 6806 of *LNCS*, pages 171–177, 2011. `doi:10.1007/978-3-642-22110-1_14`.

[5] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. On the combinatorial and algebraic complexity of quantifier elimination. *Journal of the ACM*, 43:1002–1045, 1996. `doi:10.1145/235809.235813`.

[6] Christopher W. Brown. QEPCAD B: A program for computing with semi-algebraic sets using cads. *ACM SIGSAM Bulletin*, 37:97–108, 2003. `doi:10.1145/968708.968710`.

[7] Christopher W. Brown and Scott McCallum. Enhancements to Lazard's method for cylindrical algebraic decomposition. In *Computer Algebra in Scientific Computing*, volume 12291 of *LNCS*, pages 129–149, 2020. `doi:10.1007/978-3-030-60026-6_8`.

[8] Changbo Chen, Marc Moreno Maza, Bican Xia, and Lu Yang. Computing cylindrical algebraic decomposition via triangular decomposition. In *International Symposium on Symbolic and Algebraic Computation*, pages 95–102, 2009. `doi:10.1145/1576702.1576718`.

[9] Cyril Cohen. Construction of real algebraic numbers in Coq. In *Interactive Theorem Proving*, volume 7406 of *LNCS*, pages 67–82, 2012. `doi:10.1007/978-3-642-32347-8_6`.

[10] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Automata Theory and Formal Languages*, volume 33 of *LNCS*, pages 134–183, 1975. `doi:10.1007/3-540-07407-4_17`.

[11] George E. Collins. The SAC-2 computer algebra system. In *European Conference on Computer Algebra*, volume 204 of *LNCS*, pages 34–35, 1985. `doi:10.1007/3-540-15984-3_235`.

[12] D. Yu. Grigor'ev and N.N. Vorobjov. Solving systems of polynomial inequalities in subexponential time. *Journal of Symbolic Computation*, 5:37–64, 1988. `doi:10.1016/S0747-7171(88)80005-1`.

[13] Hoon Hong. Comparison of several decision algorithms for the existential theory of the reals. RISC Report Series 91-41, Johannes Kepler University, 1991.

[14] Dejan Jovanović and Leonardo de Moura. Solving non-linear arithmetic. In *International Joint Conference on Automated Reasoning*, volume 7364 of *LNCS*, pages 339–354, 2012. `doi:10.1007/978-3-642-31365-3_27`.

[15] Dejan Jovanovic and Bruno Dutertre. Libpoly: A library for reasoning about polynomials. In *Satisfiability Modulo Theories*, volume 1889 of *CEUR Workshop Proceedings*, 2017. URL: http://ceur-ws.org/Vol-1889/paper3.pdf.

[16] Gereon Kremer. Cylindrical algebraic decomposition for nonlinear arithmetic problems. Phd thesis, RWTH Aachen University, 2020. `doi:10.18154/RWTH-2020-05913`.

[17] Gereon Kremer and Erika Ábrahám. Fully incremental cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 100:11–37, 2020. `doi:10.1016/j.jsc.2019.07.018`.

[18] Lars Langemyr. Algorithms for a multiple algebraic extension. In *Effective Methods in Algebraic Geometry*, pages 235–248. 1991. `doi:10.1007/978-1-4612-0441-1_16`.

[19] Daniel Lazard. An improved projection for cylindrical algebraic decomposition. In *Algebraic Geometry and its Applications*, pages 467–476. 1994. `doi:10.1007/978-1-4612-2628-4_29`.

[20] Rüdiger Loos. Computing in algebraic extensions. In *Computer Algebra: Symbolic and Algebraic Computation*, volume 4 of *COMPUTING*, pages 173–187. 1982. `doi:10.1007/978-3-7091-3406-1_12`.

[21] Ulrich Loup and Erika Ábrahám. GiNaCRA: A C++ library for real algebraic computations. In *NASA Formal Methods*, volume 6617 of *LNCS*, pages 512–517, 2011. `doi:10.1007/978-3-642-20398-5_41`.

[22] Ray Mines, Fred Richman, and Wim Ruitenburg. *A Course in Constructive Algebra*. Springer, 1988. `doi:10.1007/978-1-4419-8640-5`.

[23] James Renegar. A faster pspace algorithm for deciding the existential theory of the reals. In *Symposium on Foundations of Computer Science*, pages 291–295, 1988. `doi:10.1109/SFCS.1988.21945`.

[24] Renaud Rioboo. Towards faster real algebraic numbers. *Journal of Symbolic Computation*, 36(3–4):513–533, 2003. doi:https://doi.org/10.1016/S0747-7171(03)00093-2.

[25] Eberhard Schrüfer. Algebraic number fields. https://reduce-algebra.sourceforge.io/reduce38-docs/arnum.pdf.

[26] Adam Strzeboński and Elias Tsigaridas. Univariate real root isolation in multiple extension fields. In *International Symposium on Symbolic and Algebraic Computation*, pages 343–350, 2012. doi:10.1145/2442829.2442878.

[27] Adam W. Strzeboński. Solving systems of strict polynomial inequalities. *Journal of Symbolic Computation*, 29:471–480, 2000. doi:10.1006/jsco.1999.0327.

[28] Alfred Tarski. A decision method for elementary algebra and geometry. Technical report, RAND Corporation, 1951. URL: https://www.rand.org/pubs/reports/R109.html.

[29] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2013. doi:10.1017/CBO9781139856065.