Using the Regular Chains Library to build cylindrical algebraic decompositions by projecting and lifting.

Matthew England, David Wilson, Russell Bradford, and James H. Davenport

University of Bath, UK
{M.England, D.J.Wilson, R.J.Bradford, J.H.Davenport}@bath.ac.uk

Abstract. Cylindrical algebraic decomposition (CAD) is an important tool, both for quantifier elimination over the reals and a range of other applications. Traditionally, a CAD is built through a process of projection and lifting to move the problem within Euclidean spaces of changing dimension. Recently, an alternative approach which first decomposes complex space using triangular decomposition before refining to real space has been introduced and implemented within the REGULAR-CHAINS Library of MAPLE. We here describe a freely available package `ProjectionCAD` which utilises the routines within the REGULARCHAINS Library to build CADs by projection and lifting. We detail how the projection and lifting algorithms were modified to allow this, discuss the motivation and survey the functionality of the package.

1 Introduction

A *cylindrical algebraic decomposition* (CAD) is: a *decomposition* of $\mathbb{R}^n$, meaning a collection of cells which do not intersect and whose union is $\mathbb{R}^n$; *cylindrical*, meaning the projections of any pair of cells with respect to a given variable ordering are either equal or disjoint; and, *(semi)-algebraic*, meaning each cell can be described using a finite sequence of polynomial relations. CAD is best known for quantifier elimination over the reals, but has also found diverse applications such as motion planning [25] and reasoning with multi-valued functions [13].

The REGULARCHAINS Library [26] in MAPLE contains procedures to build CAD by first building a *complex cylindrical decomposition* (CCD) of $\mathbb{C}^n$ using triangular decomposition by regular chains, then refining to a CAD of $\mathbb{R}^n$. The core algorithm was developed in [11] with improvements detailed in [10] and [3].

These CAD algorithms are in contrast to the traditional approach of projection and lifting followed since Collins' original work [12]. Here, a *projection* phase repeatedly applies an operator to a set of polynomials (starting with those forming the input) each time producing another set in one fewer variables. Then the *lifting* phase builds CADs of $\mathbb{R}^i, i = 1 \ldots n$. $\mathbb{R}$ is decomposed into points and intervals corresponding to the real roots of the univariate polynomials. $\mathbb{R}^2$ is decomposed by repeating the process over each cell in $\mathbb{R}^1$ using the bivariate polynomials at a sample point. The output over each cell consists of *sections*

(where a polynomial vanishes) and *sectors* (the regions between) which together form a *stack*. The union of these stacks gives the CAD of $\mathbb{R}^2$ and the process is repeated until a CAD of $\mathbb{R}^n$ is produced. Collins defined the projection operator so the CAD of $\mathbb{R}^n$ produced using sample points this way could be concluded *sign-invariant* for the input polynomials: each polynomial has constant sign on each cell. The key tool in the proof was showing polynomials to be *delineable* in a cell, meaning the zero set of individual polynomials are disjoint sections and the zero sets of different polynomials are identical or disjoint. For developments to Collins' algorithm see for example the introduction of [4].

We use `PL-CAD` for CADs built by projection and lifting and `RC-CAD` for CADs built via CCDs. We will discuss a freely available MAPLE package PROJECTION-CAD which builds `PL-CAD`s by utilising routines developed for `RC-CAD`. We continue in Section 2 by describing the motivation for coupling these approaches before explaining the workings of the package in Section 3 and describing the current functionality in Section 4. Earlier versions of the package can be downloaded alongside [14] [15], with the latest version available from the authors. There are plans for its integration into the REGULARCHAINS Library [26] itself.

2   Motivation

PROJECTIONCAD uses routines in the REGULARCHAINS Library to build cells in the lifting phase. The advantages of utilising the routines are multiple:

- It avoids many costly algebraic number calculations by using efficient algorithms for triangular decomposition. When algebraic numbers are required (as sample points for lower dimension cells) they are represented as the unique root of a regular chain in a bounding box.
- It ensures `ProjectionCAD` will always use the best available sub-algorithms in MAPLE, such as the recently improved routines for real root isolation.
- It allows `ProjectionCAD` to match output formats with the `RC-CAD` algorithms. In particular, it allows for the use of the sophisticated *piecewise* interface [9] which highlights the tree-like structure of a CAD.

The PROJECTIONCAD package was developed to implement new theory for `PL-CAD`, most notably the work in [4], [5], [6] and [24]. More details of the functionality are given in Section 4. However, it has also allowed for easy comparison of `PL-CAD` and `RC-CAD`, leading to new developments for `RC-CAD` [3] [16]. A future aim is identification of problem classes suitable for one approach or the other.

3   CAD construction in ProjectionCAD

The pseudo code in Algorithm 1 describes the framework which the main algorithms in PROJECTIONCAD follow. They apply to either polynomials or formulae. If the former then the CAD produced is sign-invariant for each polynomial. If the latter then each formula will have constant Boolean truth value on each

cell and the CAD is said to be *truth table invariant* for the formulae: a TTI-CAD. The user may also have to supply additional information (such as which projection operator to use or which equational constraint to designate [21]). All algorithms require a specified variable ordering, which can have a significant affect on the tractability of using CAD [8]. We use ordered variables $x_1 \prec \ldots \prec x_n$ and say the *main variable* is the highest ordered variable present.

Algorithm 1: PL-CAD

> **Input** : A variable ordering $x_1 \prec \ldots \prec x_n$ and F a sequence of polynomials (or quantifier-free formulae).
>
> **Output**: A CAD of $\mathbb{R}^n$ sign-invariant for the polynomials (or truth invariant for the formulae) F; or FAIL if F is not well-oriented.

1 Run the projection phase using an appropriate projection operator on F.
2 **for** $i = 1, \ldots, n$ **do**
3 $\quad$ Assign to P_i the set of projection polynomials with main variable x_i.
4 Set C_1 to be a CAD of $\mathbb{R}^1$ formed by the decomposition of the real line according to the real roots of the polynomials in P_1.
5 **for** $i = 2, \ldots, n$ **do**
6 $\quad$ **for** each cell $c \in C_{i-1}$ **do**
7 $\quad\quad$ Check any necessary well-orientedness conditions.
8 $\quad\quad$ **if** *the input is not well oriented* **then**
9 $\quad\quad\quad$ **if** $dim(c) = 0$ **then**
10 $\quad\quad\quad\quad$ Assign to L a set containing the polynomials in P_i and any (non-constant) minimal delineating polynomials.
11 $\quad\quad\quad$ **else**
12 $\quad\quad\quad\quad$ **return** FAIL.
13 $\quad\quad$ **else**
14 $\quad\quad\quad$ Set $L := P_i$.
15 $\quad\quad$ Set $S_c := \texttt{GenerateStack}(c, L)$. // Apply Algorithm 2.
16 $\quad$ Set $C_i := \bigcup_c S_c$.
17 **return** C_n.

All algorithms in PROJECTIONCAD start with a projection phase (step 1) which uses a projection operator appropriate to the input to derive a set of projection polynomials. In steps 2−3 we sort these into sets P_i according to their main variables. The remainder of the algorithm defines the lifting phase. We start by decomposing $\mathbb{R}^1$ into cells according to the real roots of P_1 (step 4) and then repeatedly lift by generating stacks over cells until we have a CAD of $\mathbb{R}^n$. All cells are equipped with a *sample point* and a *cell index*. The index is an n-tuple of positive integers that corresponds to the location of the cell relative to the CAD. Cells are numbered in each stack during the lifting stage (from most negative to most positive), with sectors having odd numbers and sections having even numbers. Therefore the dimension of a given cell can be easily determined from its index: simply the number of odd indices in the n-tuple.

Before lifting over a cell we check any conditions necessary for the correctness of the theory being implemented (step 7). These are collectively refereed to as the input being *well-oriented* and require that projection polynomials are not *nullified* (meaning a polynomial with main variable x_i is not identically zero over a cell in $\mathbb{R}^{i-1}$). Which polynomials must be checked varies with the algorithm (see [20], [21], [4], [5] for details). If the conditions are not satisfied then an error message is returned in step 12, unless the cell in question is zero-dimensional when correctness can be restored by generating the stack with respect to minimal delineating polynomials (see [7]) as well as the projection polynomials in P_i (step 10). Note that input not well-oriented for one operator may be for another, and that Collins' operator is always successful (given sufficient resources).

Stacks are built by Algorithm 2 in step 15. These are collected together in step 16 to form a CAD of $\mathbb{R}^i$, with the final CAD of $\mathbb{R}^n$ returned. The correctness of Algorithm 1 follows from the correctness of Algorithm 2 and the correctness of the various `PL-CAD` theories implemented (see the citations in Section 4).

Algorithm 2 Stacks are generated following Algorithm 2. It finishes with a call to `RegularChains:-GenerateStack`, described in Section 5.2 of [11] (and implemented in MAPLE's `RegularChains` library). Algorithm 2 requires the input be projection polynomials: implying they satisfy the delineability conditions necessary for the cells produced when lifting to have the required invariance condition. The regular chains algorithm has stricter criteria, requiring in addition that the polynomials *separate above the cell*, meaning they are coprime and squarefree throughout. Hence Algorithm 2 must first pre-process to meet this condition.

In steps 1 and 2 we extract information from c. We identify those dimensions of the cell which are restricted to a point by consulting the cell index (indices with even integers) and collect together the equations defining these restrictions in steps $3-7$. There is no ambiguity in the ordering of the polynomials in E since a regular chain is defined by polynomials of different main variables [1]. If the cell is of full dimension then there is no need to process since the polynomials are delineable and taken from a squarefree basis. Otherwise, we process using Algorithms 3 and 4 in steps 10 and 11. The restriction is identified using a regular chain $\hat{rc}$ (step 9) together with the original bounding box. Note that $\hat{E}$ defines a single regular chain since the equations were extracted from one.

Algorithm 3 In order to make the polynomials coprime we use repeated calls to a triangular decomposition algorithm in step 3 (described in [19] and part of the `RegularChains` Library). Given lists of polynomials L_1 and L_2 and a regular chain, it returns a decomposition of the zeros of L_1 which are also also zeros of the regular chain but not zeros of L_2. We use $\hat{rc}$ for the regular chain, so we work on the restriction, and build up a list of coprime polynomials by ensuring existing ones (L_2) are not zeros in decompositions of the next one (L_1). Each time the decomposition is a list of either regular chains or *regular systems* (a regular chain and an inequality regular with respect to the chain [23]).

Algorithm 2: GenerateStack

Input : A cell c from a CAD of $\mathbb{R}^k$ and a set P of projection
                polynomials in $x_1 \ldots x_{k+1}$ (part of a squarefree basis).
Output: A set of cells $\mathcal{S}$ of $\mathbb{R}^{k+1}$ comprising a stack over c. The
                polynomials in P are sign-invariant on each cell of $\mathcal{S}$.

1 Set I and sp to be the cell index and sample point of c.
2 Set rc and bb to be the regular chain and bounding box encoding sp.
3 Set E to be the set of k polynomials whose zeros define rc, ordered by
    increasing main variable.
4 Set $\hat{E} := \{ \, \}$.
5 **for** $i = 1, \ldots, k$ **do**
6 | **if** *the i'th integer in I is even* **then**
7 | | Add the ith polynomial in E to $\hat{E}$.

8 **if** $\hat{E} \neq \{ \, \}$ **then**
9 | Set $\hat{rc}$ to be the regular chain formed by $\hat{E}$.
10 | $\hat{P} := \texttt{MakeCoprime}(P, \hat{rc}, c)$. // Apply Algorithm 3.
11 | $\hat{P} := \texttt{MakeSquareFree}(P, \hat{rc}, c)$. // Apply Algorithm 4.
12 $\mathcal{S} := \texttt{RegularChains:-GenerateStack}(c, \hat{P}, k+1)$ // From [11].
13 **return** $\mathcal{S}$.

Algorithm 3: MakeCoprime

Input : A set of polynomials P, a regular chain $\hat{rc}$ and a cell c.
Output: A set of polynomials $\hat{P}$ which describe the same set of varieties,
                but which are coprime over c.

1 Set $\hat{P} = \{ \, \}$.
2 **for** *polynomial* $p \in P$ **do**
3 | $T := \texttt{Triangularize}([p], \hat{P}, \hat{rc})$. // From [19].
4 | **for** *component* $\mathcal{C}$ *of* T **do**
5 | | **if** $\text{mvar}(\mathcal{C}) \neq \text{mvar}(p)$ **then**
6 | | | next $\mathcal{C}$.
7 | | **if** $\mathcal{C}$ *has a zero compatible with the sample point of* c **then**
8 | | | Add the polynomial in $\mathcal{C}$ with same main variable as p to $\hat{P}$.

9 **return** $\hat{P}$.

We consider each of these components in turn. If the main variable is lower then the solution is discarded. Otherwise we check if the component has a solution compatible with the sample point for the cell (as it may be a solution of $\hat{rc}$ other than one isolated by bb). This means isolating the real solutions (of the component excluding the top dimension) and refining their bounding boxes until they are either within bb or do not intersect at all. It is achieved using the `RealRootIsolate` command in the **RegularChains** Library (see [2]). Finally if

the component passes these tests then the polynomial in the main variable is extracted and added to the set returned from Algorithm 3 in step 8.

Algorithm 4 In order to make the polynomials squarefree we use repeated calls to an algorithm which does this modulo a regular chain ($\hat{rc}$: so that we are working on the restriction). It is an analogue of Musser's [22] with the gcd calculations performed modulo the regular chain as described in [18]. It assumes the polynomial is regular modulo the chain and so we first test for this. If not regular (the leading coefficient vanishes) then we consider the `tail` (polynomial minus the leading term) in step 5, if still in the main variable. The output of the factorization is either: rc and a list of polynomials forming a squarefree decomposition of p modulo rc; or a list of pairs of regular chains and squarefree decompositions where the regular chains are a decomposition of rc. In the latter case only one will be relevant for the root isolated by bb and we identify which using the `RealRootIsolate` command, similarly to Algorithm 3.

Algorithm 4: MakeSquareFree

> **Input**   : A set of polynomials P, a regular chain $\hat{rc}$ and a cell c.
> **Output**: A set of polynomials $\hat{P}$ which describe the same set of varieties, but which are each squarefree over c.

1 Set $\hat{P} = \{\,\}$.
2 while *P is not empty* **do**
3   Remove a polynomial p from P.
4   **if** *p is not regular over $\hat{rc}$* **then**
5   Set $\hat{p} = \mathtt{tail}(p)$
6   **if** $\mathrm{mvar}(\hat{p}) = \mathrm{mvar}(p)$ **then**
7   Add $\hat{p}$ to P and `continue` from step 2.
8 $T := \mathtt{SquarefreeFactorization}(p, \hat{rc})$.
9   Select $\mathcal{C}$ as the component in T compatible with the sample point of c.
10   Set $\hat{p}$ to be the product of polynomials in the decomposition in $\mathcal{C}$.
11   Add $\hat{p}$ to $\hat{P}$.
12 return $\hat{P}$.

---

4   Functionality of ProjectionCAD

We finish by listing some of the functionality of within PROJECTIONCAD, focusing on aspects not usually found in other CAD implementations:

- Sign-invariant CADs can be built using the Collins [12] or McCallum [20] projection operators.
- CADs can be built with the stronger property of *order-invariance* (where each polynomial has constant order of vanishing on each cell) [20].
- *Equational constraints* (ECs) are equations logically implied by the formula. They can be utilised via McCallum's reduced projection [21] and a more efficient lifting phased (detailed in Section 5 of [5]).

- TTICADs can be built for sequences of formulae, making use of ECs in each [4] [5]. TTICAD can be both a desired structure for applications [17] and an efficient way to build a truth-invariant CAD (allowing savings from ECs for conjunctive sub-formulae, not ECs of the whole formula).
- Minimal delineating polynomials [7] are built automatically, avoiding unnecessary failure declarations (which can occur in QEPCAD). See [14] for an example of this.
- User commands for stack generation and the construction of *induced CADs* (a CAD of $\mathbb{R}^i, i < n$ produced en route to a CAD of $\mathbb{R}^n$), allowing for easy experimentation with the theory.
- *Layered CADs* contain cells of only a certain dimension or higher. They can be produced (more efficiently than a full CAD) [24].
- *Variety CADs* contain only those cells that lie on the variety defined by an EC. They can be produced (more efficiently than a full CAD) [24].
- Layered and manifold TTICADs as well as layered-manifold CADs can be produced [24] (combining the savings from the different theories).
- Heuristics are available to help with choices such as variable ordering, EC designation and breaking up parent formulae for TTICAD [6].

Details can be found in the citations above and the technical reports [14], [15].

Acknowledgements

References

1. P. Aubry, D. Lazard, and M. Moreno Maza. On the theories of triangular sets. *Journal of Symbolic Computation*, 28(1-2):105–124, 1999.
2. F. Boulier, C. Chen, F. Lemaire, and M. Moreno Maza. Real root isolation of regular chains. In *Proc. ASCM '09*, pages 15–29, 2009.
3. R. Bradford, C. Chen, J.H. Davenport, M. England, M. Moreno Maza, and D. Wilson. Truth table invariant cylindrical algebraic decomposition by regular chains. *Submitted*. Preprint: `http://opus.bath.ac.uk/38344/`, 2014.
4. R. Bradford, J.H. Davenport, M. England, S. McCallum, and D. Wilson. Cylindrical algebraic decompositions for boolean combinations. In *Proc. ISSAC '13*, pages 125–132. ACM, 2013.
5. R. Bradford, J.H. Davenport, M. England, S. McCallum, and D. Wilson. Truth table invariant cylindrical algebraic decomposition. *Submitted*.
   Preprint: `http://opus.bath.ac.uk/38146/`, 2014.
6. R. Bradford, J.H. Davenport, M. England, and D. Wilson. Optimising problem formulations for cylindrical algebraic decomposition. In *Intelligent Computer Mathematics* (LNCS 7961), pages 19–34. Springer Berlin Heidelberg, 2013.
7. C.W. Brown. The McCallum projection, lifting, and order-invariance. Technical report, U.S. Naval Academy, Compt. Sci. Dept., 2005.

8. C.W. Brown and J.H. Davenport. The complexity of quantifier elimination and cylindrical algebraic decomposition. In *Proc. ISSAC '07*, pages 54–60. ACM, 2007.

9. C. Chen, J.H. Davenport, J. May, M. Moreno Maza, B. Xia, R. Xiao, and Y. Xie. User interface design for geometrical decomposition algorithms in Maple. In *Proc. Mathematical User-Interface Workshop*, 12 pages, 2009.

10. C. Chen and M. Moreno Maza. An incremental algorithm for computing cylindrical algebraic decompositions. *Proc. ASCM '12*. To appear in LNAI, Springer. Preprint: `http://arxiv.org/abs/1210.5543`, 2012.

11. C. Chen, M. Moreno Maza, B. Xia, and L. Yang. Computing cylindrical algebraic decomposition via triangular decomposition. In *Proc. ISSAC '09*, pages 95–102. ACM, 2009.

12. G.E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Proc. 2nd GI Conference on Automata Theory and Formal Languages*, pages 134–183. Springer-Verlag, 1975.

13. J.H. Davenport, R. Bradford, M. England, and D. Wilson. Program verification in the presence of complex numbers, functions with branch cuts etc. In *Proc. SYNASC '12*, pages 83–88. IEEE, 2012.

14. M. England. An implementation of CAD in Maple utilising McCallum projection. Technical report, Uni. of Bath, Dept. Comp. Sci., 2013-02.
Available at `http://opus.bath.ac.uk/33180/`, 2013.

15. M. England. An implementation of CAD in Maple utilising problem formulation, equational constraints and truth-table invariance. Technical report, Uni. of Bath, Dept. Comp. Sci., 2013-04. Available at `http://opus.bath.ac.uk/35636/`, 2013.

16. M. England, R. Bradford, C. Chen, J.H. Davenport, M. Moreno Maza, and D. Wilson. Problem formulation for truth-table invariant cylindrical algebraic decomposition by incremental triangular decomposition. *To Appear:* Proc. CICM '14 (LNAI 8543), pages 45–60, 2014.

17. M. England, R. Bradford, J.H. Davenport, and D. Wilson. Understanding branch cuts of expressions. In *Intelligent Computer Mathematics* (LNCS 7961), pages 136–151. Springer Berlin Heidelberg, 2013.

18. X. Li, M. Moreno Maza, and W. Pan. Computations modulo regular chains. In *Proc. ISSAC '09*, pages 239–246. ACM, 2009.

19. M. Moreno Maza. On triangular decompositions of algebraic varieties. Presented at: *Effective Methods in Algebraic Geometry* (MEGA), 2000.

20. S. McCallum. An improved projection operation for cylindrical algebraic decomposition. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Texts & Monographs in Symbolic Computation, pages 242–268. Springer-Verlag, 1998.

21. S. McCallum. On projection in CAD-based quantifier elimination with equational constraint. In *Proc. ISSAC '99*, pages 145–149. ACM, 1999.

22. D.R. Musser. Multivariate polynomial factorization. *Journal of the ACM*, 22(2):291–308, 1975.

23. D. Wang. Computing triangular systems and regular systems. *Journal of Symbolic Computation*, 30(2):221–236, 2000.

24. D. Wilson, R. Bradford, J.H. Davenport, and M. England. Cylindrical algebraic sub-decompositions. *To appear: Mathematics in Computer Science*, Springer 2014.

25. D. Wilson, J.H. Davenport, M. England, and R. Bradford. A "piano movers" problem reformulated. In *Proc. SYNASC '13*. IEEE, 2014.

26. The REGULARCHAINS Library: `http://www.regularchains.org`