Cylindrical Algebraic Decomposition in *Macaulay2*

Corin Lee, Tereso del Río and Hamid Rahkooy

Abstract

`CylindricalAlgebraicDecomposition.m2` is the first implementation of Cylindrical Algebraic Decomposition (CAD) in *Macaulay2*. CAD decomposes space into 'cells' where input polynomials are sign-invariant. This package computes an Open CAD (full-dimensional cells only) for sets of real polynomials with rational coefficients, enabling users to solve existential problems involving strict inequalities. With the construction of a full CAD (cells of all dimensions), this tool could be extended to solve any real quantifier elimination problem. The current implementation employs the Lazard projection and introduces a new heuristic for choosing the variable ordering.

1 Introduction

This article documents the `CylindricalAlgebraicDecomposition` package for *Macaulay2* [GS]. Of particular note is the function `findPositiveSolution`, which allows the user to solve satisfiability problems on non-linear real arithmetic through the use of Open Cylindrical Algebraic Decomposition (Open CAD).

Cylindrical Algebraic Decomposition (CAD) is an important algorithm in symbolic computation for studying real semi-algebraic sets. It takes a set of multivariate polynomials and decomposes the solution space into disjoint regions known as *cells*, within which the initial polynomials do not change sign. In satisfiability problems, this reduces the search from the uncountable real space to a finite number of regions.

CAD was first introduced as a method for performing *quantifier elimination* (QE) over the reals in [Col75] and has applications in algebraic geometry and fields such as robotics, economics, and biology. CAD is well suited for computation and has been implemented in many widely used computer algebra packages, such as *REDUCE, Maple* and *Mathematica*.

However, CAD is computationally expensive, with a worst-case complexity that is doubly exponential in the number of variables [DH88]. This package reduces the complexity of the original CAD algorithm proposed by Collins, incorporating modern improvements. It contains an implementation of Open CAD (see Section 2.4.3), uses the contemporary Lazard projection [Laz94, MPP19] (see Section 2.4.2), and is the first implementation of the `gmods` heuristic for variable ordering [dRE22] (see Section 2.4.1).
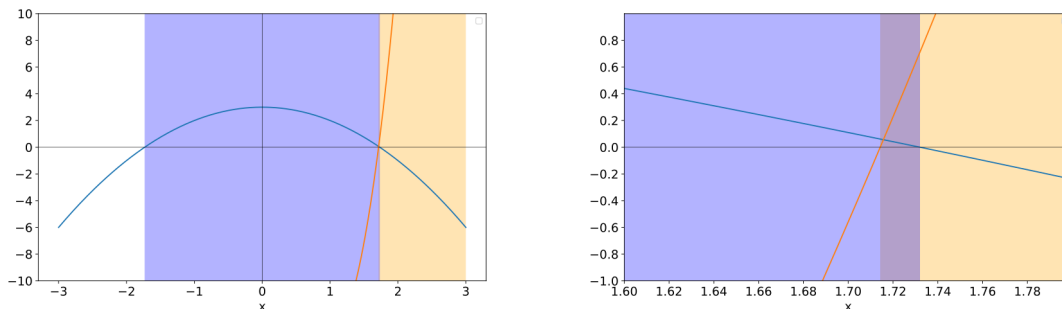
1.1 Motivating Example

Imagine we are interested in determining whether there exists an $x \in \mathbb{R}$ such that

$$g(x) = 3 - x^2 > 0 \quad \text{and} \quad h(x) = (7x - 12)(x^2 + x + 1) > 0. \tag{1}$$

Studying the problem numerically, one could sample 1001 equispaced points between -50 and 50 and find no point that satisfies the conditions. However, such a point does exist:

(a) It is not initially clear whether there is some value of x that ensures both polynomials are positive...

(b) ...however it becomes clear that such an x does exist when looking in greater detail.

Figure 1: Algebraic varieties of $3 - x^2$ (blue) and $(7x - 12)(x^2 + x + 1)$ (orange), and the associated values of x ensuring these polynomials are positive (blue and orange highlighting, respectively).

The function `findPositiveSolution` in this package can check whether a set of strict polynomial inequalities has a solution. For the univariate example above, this function outputs the following:

```
i1 : R=QQ[x]

i2 : findPositiveSolution({3-x^2,(7*x-12)*(x^2+x+1)})

                         1539
o2 = (true, HashTable{x => ----})
                          896
```

The package is not restricted to the univariate case and can find solutions for polynomials in as many variables as desired, given enough time.

2 Cylindrical Algebraic Decomposition

2.1 Brief history

A quantifier elimination (QE) problem involves turning a *quantified formula* consisting of polynomial equations and inequalities combined with logical connectives such as $\wedge, \vee, \neg$, and quantifiers $\forall$ and $\exists$, into an equivalent *quantifier-free formula*.

For example, the quantified formula

$$(\exists x) \ (ax^2 + bx + c = 0) \wedge (a \neq 0) \text{ with } a, b, c \in \mathbb{R}$$

is logically equivalent to

$$b^2 - 4ac \geq 0.$$

In the 1940s, Tarksi showed through a constructive proof that real quantifier elimination is always possible [Tar51]. However, the algorithm used had complexity that could not be bounded by a finite tower of exponents, and so was considered impractical.

In 1975, Collins [Col75] proposed CAD as an algorithm that could solve real quantifier elimination problems with doubly exponential complexity in the number of variables [BD07]. This complexity meant that the algorithm could only solve simple problems, but advances in computational power and CAD theory have allowed us to solve more challenging cases.

2.2 CAD basics

This paper covers the key parts of CAD, for a more detailed explanation of CAD see [Jir95].

- A **cell** is a connected region of $\mathbb{R}^n$.

- A **decomposition** of a space X is a collection of disjoint cells whose union is X.

- A cell is **cylindrical** with respect to the variable ordering $x_1 \prec \cdots \prec x_n$ if, for any $i < n$, it can be described as

$$\{(x_1, \ldots, x_n) \mid g_{i,1}(x_1, \ldots, x_i) \leq x_{i+1} \leq g_{i,2}(x_1, \ldots, x_i)\},$$

  where $g_{i,1}$ and $g_{i,2}$ are continuous functions.

- Furthermore, a cell is **algebraic** if $g_{i,1}$ and $g_{i,2}$ are polynomials, and is called a *j-cell* if it is homeomorphic to $\mathbb{R}^j$, for $0 \leq j \leq n$.

- A decomposition is **cylindrical** with respect to the variable ordering $x_1 \prec \cdots \prec x_n$ if, for any $i \leq n$, the projections of any two cells into $\mathbb{R}[x_1, \ldots, x_i]$ are either disjoint or identical.

- A CAD is a cylindrical decomposition whose cells are cylindrical and algebraic.

A CAD will only be useful if its cells satisfy some property of interest. We focus on *sign-invariance*, where each input polynomial maintains a constant sign (positive, negative, or zero) within a cell. This invariance means it is sufficient to test a representative *sample point* for each cell to determine the signs of the input polynomials.

Definition 1 (Cylindrical Algebraic Decomposition)**.**
For $\mathcal{F}_n$ a set of polynomials in variables $x_1, \ldots, x_n$, an $\mathcal{F}_n$-invariant CAD of $\mathbb{R}^n$ (or colloquially 'a CAD of $\mathcal{F}_n$'), denoted $\mathrm{CAD}(\mathcal{F}_n)$, is a cylindrical decomposition of $\mathbb{R}^n$ whose cells are cylindrical and algebraic, and over which the polynomials in $\mathcal{F}_n$ have constant sign.
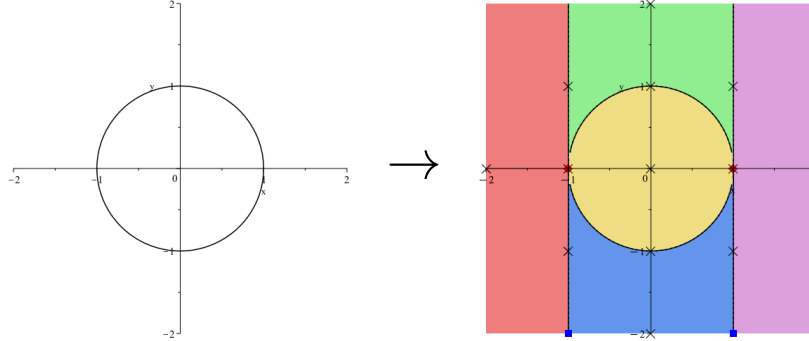

Figure 2: The graph of $\mathcal{F} = \{x^2 + y^2 - 1\}$ and its associated sign-invariant CAD of $\mathbb{R}^2$. This CAD consists of 13 cells: five 2-cells (the coloured regions), six 1-cells (the lines between them) and two 0-cells (the two red points where the lines meet). The black crosses represent the sample points for each cell. These cells stack in *cylinders* over the five cells of the CAD of $\mathbb{R}^1$ (the two points at the bottom and the regions between them). Each cell is described by constraints on x and y.

For example, the collection of cells in Figure 2 form a CAD of $\mathcal{F} = \{x^2 + y^2 - 1\}$, with sample points $(-2, 0), (-1, 1), (-1, 0), (-1, -1), (0, 2), (0, 1), (0, 0), (0, -1), (0, -2), (1, 1), (1, 0), (1, -1)$ and $(2, 0)$.

2.3   CAD construction

We will also use the term 'CAD' to refer to the algorithm that generates a sign-invariant CAD for a given set of polynomials. Collins' original algorithm (sometimes referred to as a *projection-and-lifting* CAD algorithm) uses a *projection operator* on the input polynomials to produce a set of polynomials in one fewer variable, whose properties enable a CAD of the input polynomials to be constructed from a CAD of their projection.

In one dimension, constructing a sign-invariant CAD is straightforward: its cells are the collection of roots of the polynomials and the regions either side of them. Each sign-invariant cell defines a region where the number of real roots (the 'root structure') does not change.
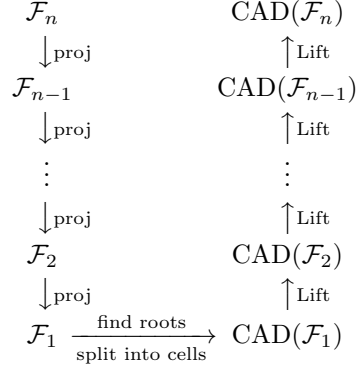
In higher dimensions, we treat the polynomials in $\mathcal{F}_n$ as univariate in the greatest variable with respect to the ordering, written as $\mathcal{F}_n \subset \mathbb{R}[x_1, \ldots, x_{n-1}][x_n]$. To find changes in the root structure in x_n, we construct a set of *projection polynomials* $\mathcal{F}_{n-1}$ in one fewer variable, whose roots indicate when changes occur (referred to as 'projecting with respect to x_n', or 'projecting away' x_n). This process is repeated, with each $\mathcal{F}_k$ obtained by projecting $\mathcal{F}^{k+1}$ with respect to x_{k+1}, until we reach $\mathcal{F}_1 \subset \mathbb{R}[x_1]$ whose roots and intervals make up a CAD of $\mathbb{R}^1$.

Due to the nature of the projection polynomials, the root structure of $\mathcal{F}_k$ does not vary over a cell of the CAD of $\mathbb{R}^{k-1}$. We can continue the process and decompose the space above the cell into a new set of sign-invariant cells. The full collection of these cells forms the CAD of $\mathbb{R}^k$.

This algorithm follows a two-phase approach:
- **Projection:** Repeatedly apply the projection operator to produce a chain of projection polynomial sets, each in one variable fewer than the previous, until reaching a univariate set of polynomials.
- **Lifting:** Starting with the univariate set, construct a CAD of the polynomials. Above this, one can create a new CAD in one dimension greater via *lifting*, made possible by Theorem 2.1.

This recursive approach ultimately leads to a complete CAD for the original polynomial set.

$$
\begin{array}{ccc}
\mathcal{F}_n & & \mathrm{CAD}(\mathcal{F}_n) \\
\downarrow{\mathrm{proj}} & & \uparrow{\mathrm{Lift}} \\
\mathcal{F}_{n-1} & & \mathrm{CAD}(\mathcal{F}_{n-1}) \\
\downarrow{\mathrm{proj}} & & \uparrow{\mathrm{Lift}} \\
\vdots & & \vdots \\
\downarrow{\mathrm{proj}} & & \uparrow{\mathrm{Lift}} \\
\mathcal{F}_2 & & \mathrm{CAD}(\mathcal{F}_2) \\
\downarrow{\mathrm{proj}} & & \uparrow{\mathrm{Lift}} \\
\mathcal{F}_1 & \xrightarrow[\text{split into cells}]{\text{find roots}} & \mathrm{CAD}(\mathcal{F}_1)
\end{array}
$$

Theorem 2.1 ([Col75], Theorem 5).
Let $\mathcal{F}$ be a non-empty set of non-zero real polynomials in $n \geq 2$ real variables. Let S be a connected subset of $\mathbb{R}^{n-1}$. If every element of the projection of $\mathcal{F}$ is sign-invariant on S, then the polynomials in $\mathcal{F}$ are delineable over S.

Definition 2.
*A set of polynomials is **delineable** over S if the number of distinct collective roots remains constant, i.e., their roots do not intersect or disappear (see Figure 3).*

delineable over region   not delineable over region   delineable over refined regions
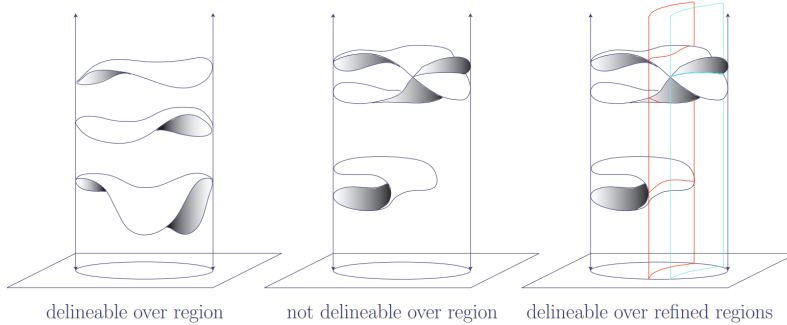
Figure 3: Illustration of delineability, from [Bro04].

This theorem allows us to obtain an n-dimensional sign-invariant CAD by studying a finite set of sample points for the cells in an $(n-1)$-dimensional decomposition. For example, the sign-invariant CAD of $\mathcal{F} = \{x^2 + y^2 - 1\}$ seen in Figure 2 can be obtained by studying a point over each sign-invariant cell of the CAD of the projection polynomials $\{x - 1, x + 1\}$.

2.4 Choices made for the package

Certain decisions have been made in designing the CAD algorithm to best suit this package. These decisions are explained below.

2.4.1 Variable ordering

In order to compute a CAD, one must decide on the variable ordering. The ordering chosen can significantly impact the number of cells in a CAD (see Figure 4), and thus the resources and time needed to produce it. Considerable work has been done trying to find the best way to choose the variable ordering [HEW$^+$15, EF19, CZC20].
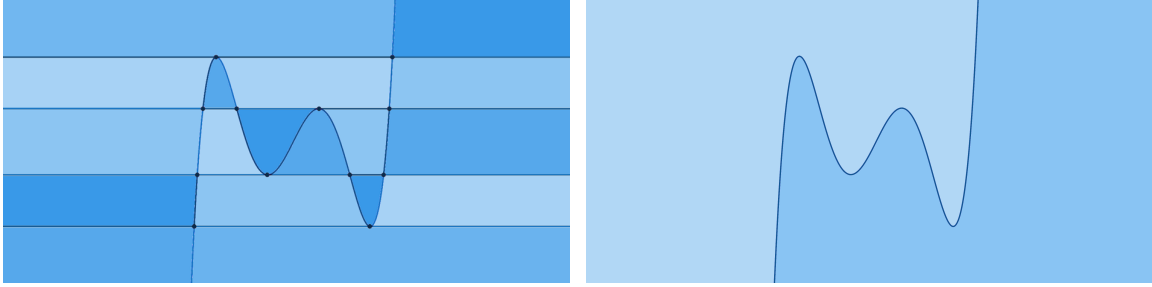
Figure 4: CADs sign-invariant for the set of polynomials $\{x^5 + 5x^4 + 5x^3 - 5x^2 - 6x - 2y\}$. Using ordering $y \prec x$, we obtain a CAD with 57 cells (18 2-cells, 27 1-cells and 12 0-cells). Using the ordering $x \prec y$ generates only three cells (two 2-cells and one 1-cell).

`CylindricalAlgebraicDecomposition` employs `gmods`, a strategy developed by Del Río and England [dRE22], which first projects the variable with the lowest degree sum across the set of polynomials.

2.4.2 Lazard projection

This package uses the *Lazard projection*, originally proposed by Lazard [Laz94], later validated by McCallum et al. [MPP19]. Constructing an open CAD eliminates the need for extra checks that would otherwise be needed when applying the Lazard projection in a full CAD.

Definition 3 (Lazard projection).
*When projecting, we consider polynomials in $\mathbb{R}[x_1, \ldots, x_n]$ as univariate in x_n with coefficients in $\mathbb{R}[x_1, \ldots, x_{n-1}]$, hence the notion of 'leading coefficient' is in terms of x_n.*

Let $\mathcal{F}_n$ be a finite set of irreducible pairwise coprime polynomials in $\mathbb{R}[x_1, \ldots, x_n]$, with $n \geq 2$. The Lazard projection of $\mathcal{F}_n$, $P_L(\mathcal{F}_n) \subset \mathbb{R}[x_1, x_2, \ldots, x_{n-1}]$, comprises the following polynomials:

1. *all leading coefficients of the elements of $\mathcal{F}_n$,*

2. *all trailing coefficients (coefficients independent of x_n) of the elements of $\mathcal{F}_n$,*

3. *all discriminants of the elements of $\mathcal{F}_n$, and*

4. *all resultants of pairs of distinct elements of $\mathcal{F}_n$.*

$$P_L(\mathcal{F}_n) = \{\mathrm{lc}(f_i)\} \cup \{\mathrm{tc}(f_i)\} \cup \{\mathrm{disc}(f_i)\} \cup \{\mathrm{res}(f_i, f_j \mid i \neq j)\}.$$

The roots of these projection polynomials sufficiently capture the points at which the root structure of $\mathcal{F}_n$ changes, such as when two distinct roots meet or the polynomial degree drops. In `CylindricalAlgebraicDecomposition`, the set of input and output polynomials are replaced by their nonconstant irreducible factors, ensuring they are irreducible and pairwise coprime (see 2.4.2).

2.4.3 Open CAD

We focus on the notion of an *Open CAD*, that is, open solution sets of systems of strict polynomial inequalities [Str00]. In other words, an Open CAD will only consist of open, full-dimensional n-cells.

At present, *Macaulay2* does not support polynomial division over the reals or symbolic root storage, making a full CAD impossible due to the inability to store or lift over algebraic irrational roots. Despite

this limitation, many real-world applications are not interested in these zero-measure regions, so an Open CAD is enough for most cases.

As a result, we have implemented an Open CAD algorithm, using the existing `RealRoots` package [GMSY24] to isolate roots within rational intervals, allowing rational sample points to be chosen which represent every open cell. Additionally, inefficiencies and issues within the `RealRoots` package have been corrected and improved.

2.5 `RealRoots` improvements

`CylindricalAlgebraicDecomposition` calls the existing `RealRoots` package to perform real root isolation. However, there are some limitations and inefficiencies in `RealRoots` that we have addressed:

- **Only works on one polynomial:** `CylindricalAlgebraicDecomposition` simply performs this real root isolation on $\prod f_i$, where each f_i is reduced to its irreducible factors during the projection phase.
- **Error in root bound:** The original root bound failed when the leading coefficient was negative. This was later fixed in the JSAG release of the package, but has also been fixed here.
- **Inefficient root bound:** The original root bound was weaker than necessary, and would scale badly in cases of coefficient blowup. The fix improves the existing bound, implements another which better handles large coefficients, and the command now takes the smaller bound, thus speeding up each real root isolation step.
- **Inefficient with close roots:** The original algorithm would continue to refine every interval in cases when one root required a smaller interval, creating extra work. Now, once a root has been sufficiently isolated, its interval is left alone.
- **Roots not always ordered:** Root intervals are now consistently listed lowest to highest.

3 The `CylindricalAlgebraicDecomposition` package

This section provides a detailed description of the `CylindricalAlgebraicDecomposition`. This package allows the user to produce an open, sign-invariant CAD with respect to a list of input polynomials in $\mathbb{Q}[x_1, \ldots, x_n]$ by constructing a tree of sample points in $\mathbb{Q}^n$, representing the open cells of a CAD of $\mathbb{R}^n$. It also allows users to step through each phase of the Open CAD algorithm and check for points where all input polynomials are positive.

3.1 Package Overview

```
findPositiveSolution
├── openCAD
│   ├── projectionPhase
│   │   ├── gmodsHeuristic
│   │   └── lazardProjection
│   │       ├── leadCoeff
│   │       ├── factorsInList
│   │       │   └── factors
│   │       ├── discriminant
│   │       └── resultant
│   ├── liftingPoint
│   │   ├── evaluatePolynomials
│   │   └── samplePoints
│   │       └── realRootIsolation
│   └── positivePoint
│       └── evaluatePolynomials
└── hashify
```
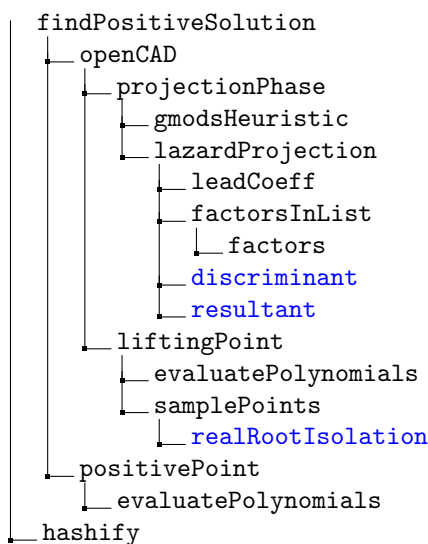
Figure 5: List of functions contained in the `CylindricalAlgebraicDecomposition` package. Methods labelled in blue are from external packages.

6

The package's top-level function is `findPositiveSolution`, which takes a list of input polynomials and returns a point in which all input polynomials are strictly positive or informs the user that no such point exists.

This is built upon two other functions: `openCAD`, which constructs an Open CAD for the given list of polynomials as a tree of sample points, and `positivePoint` which takes the tree (or a branch with leaves) and checks if such a positive point exists. These commands are also available to the user, allowing them to construct Open CADs and check the cells as needed.

Now we proceed to describe these functions in more detail before continuing with the remaining functions in the package, providing inputs, outputs, and examples. To illustrate the algorithm, we will use the following system of polynomials $\mathcal{F} = \{f_1, f_2\}$, taken from [Jir95]:

$$\begin{array}{ll} f_1 = & x_1^2 + x_2^2 - 1 \\ f_2 = & x_1^3 - x_2^2. \end{array} \tag{2}$$

```
R=QQ[x1,x2];
f1:=x1^2+x2^2-1; f2:=x1^3-x2^2;
F={f1,f2};
```

3.1.1 openCAD

Input: List of polynomials $\mathcal{F}_n = \{f_1, \ldots, f_r\}$ in variables $x_1, \ldots, x_n$.

Output: Tree of sample points representing the Open CAD of $\mathbb{R}^n$ with respect to $\mathcal{F}_n$.

The command `openCAD` produces an open, projection-and-lifting CAD, and consists of a projection phase (see Section 3.3) and a lifting phase (see Section 3.4).

The format of `openCAD` is a tree, represented in *Macaulay2* as a collection of nested hash tables with n levels. Each level k, for $1 \le k < n$, represents the CAD of $\mathbb{R}^k$, containing the projection polynomials $\mathcal{F}_k$, the cell below (as a $(k-1)$-dimensional sample point) and the collection of sample points in x_k obtained by lifting over that cell. Each leaf node (level n) represents an n-dimensional sample point of $\mathcal{F}_n$.

```
i3 : openCAD(F)

o3 = MutableHashTable{...7...}
```

The output of `openCAD` is substantial, and thus is left unprinted in order to minimise clutter (see 3.2 for the full output). In this example, the first level consists of $\mathcal{F}_1$ (`polynomials`), the (empty) cell below (`point`), and the five open sample points of the CAD of $\mathbb{R}^1$.

```
i4 : peek(openCAD(F))

o4 = MutableHashTable{point => MutableHashTable{}                         }
                                                        3     2
                      polynomials => {x1 - 1, x1 + 1, x1, x1  + x1  - 1}
                        3
                      - - => MutableHashTable{...5...}
                        4
                        5
                      - - => MutableHashTable{...2...}
                        2
                      3
                      - => MutableHashTable{...7...}
                      8
                      15
                      -- => MutableHashTable{...7...}
                      16
                      17
                      -- => MutableHashTable{...5...}
                       8
```

3.1.2 positivePoint

Input 1: List of polynomials $\mathcal{F}_n = \{f_1, \ldots, f_r\}$ in variables $x_1, \ldots, x_n$.
Input 2: A tree of sample points representing an Open CAD (or a subtree reaching down to level n).

Output: If every input polynomial is positive at an n-dimensional sample point, that sample point is returned. If not, '*no point exists*' is returned.

```
i5 : positivePoint(F,openCAD(F))

o5 = MutableHashTable{...2...}
```

Yes, such a point exists. The point is:

```
i6 : peek positivePoint(F,openCAD(F))

o6 = MutableHashTable{x2 => 0 }
                              17
                      x1 => --
                              8
```

3.1.3 findPositiveSolution

Input: List of polynomials $\mathcal{F}_n = \{f_1, \ldots, f_r\}$ in variables $x_1, \ldots, x_n$.

Output: Returns a boolean indicating if there exists a point in which all polynomials are strictly positive, and if so, it returns one such point as a hash table.

The command `findPositiveSolution` combines the previous commands, first producing an Open CAD from the input polynomials, then checking this for a point where the input polynomials are positive.

```
i7 : findPositiveSolution(F)

                            17
o7 = (true, HashTable{x1 => --})
                             8
                      x2 => 0
```

$x_1:$ $-\frac{5}{2}$ $-\frac{3}{4}$ $\frac{3}{8}$ $\frac{15}{16}$ $\frac{17}{8}$

$x_2:$ 0 $-\frac{2\,237}{1\,024}$ 0 $\frac{2\,237}{1\,024}$ $-\frac{2\,003}{1\,024}$ $-\frac{979}{2\,048}$ 0 $\frac{979}{2\,048}$ $\frac{2\,003}{1\,024}$ $-\frac{16\,159}{8\,192}$ $-\frac{7\,967}{16\,384}$ 0 $\frac{7\,967}{16\,384}$ $\frac{16\,159}{8\,192}$ $-\frac{33}{8}$ 0 $\frac{33}{8}$

Figure 6: **Top:** A visualisation of a (full) CAD of $\mathcal{F}$. The coloured regions represent the open cells, the green diamond represents the sample point from `positivePoint`/`findPositiveSolution`, and the blue diamonds represent the remaining sample points from `openCAD`.
Bottom: A representation of these sample points in a tree structure.

3.2   hashify

Input: A data structure.

Output: The same data structure with any mutable hash tables converted to immutable ones.

The recursive nature of this Open CAD algorithm greatly benefits from the ability to copy and modify previous steps, something only possible with a *mutable* hash table. Additionally, mutable hash tables do not display their entries by default, avoiding the otherwise overwhelming output given by `openCAD` and allowing the user to expand the parts of interest as they see fit.

However, mutable hash tables are incomparable and cannot be fully expanded. The `hashify` command was implemented to address this, converting mutable hash tables into immutable ones which are able to be viewed in their entirety.

Fully expanding the output of `openCAD` illustrates the structure of the computed Open CAD (compare this with Figure 6), while also demonstrating that `hashify` should be used with care:

```
i8 : openCAD(F)

o8 = MutableHashTable{...7...}

i9 : hashify openCAD(F)
```

```
                3              2237                                             3
o9 = HashTable{- - => HashTable{- ---- => HashTable{point => HashTable{x1 => - -  }}} }
                4              1024                                            4
                                                                          2237
                                                            x2 => - ----
                                                                          1024
                                                                        3
                             0 => HashTable{point => HashTable{x1 => - -}}
                                                                        4
                                                            x2 => 0
                            2237                                          3
                            ---- => HashTable{point => HashTable{x1 => - - }}
                            1024                                          4
                                                                    2237
                                                        x2 => ----
                                                                    1024
                                                              3
                            point => HashTable{x1 => - -}
                                                              4
                                       2    7       2    27
                            polynomials => {x2  - --, - x2  - --}
                                               16        64
          5
        - - => HashTable{0 => HashTable{point => HashTable{x1 => - -}}}
          2                                                        2
                                                        x2 => 0
                                                        5
                            point => HashTable{x1 => - -}
                                                        2
                                       2    21      2    125
                            polynomials => {x2  + --, - x2  - ---}
                                               4          8
      3              979                                            3
    - => HashTable{- ---- => HashTable{point => HashTable{x1 => - -     }}}
      8              2048                                           8
                                                                  979
                                                      x2 => - ----
                                                                  2048
                           2003                                          3
                         - ---- => HashTable{point => HashTable{x1 => - -     }}
                           1024                                          8
                                                                    2003
                                                        x2 => - ----
                                                                    1024
                                                                 3
                            0 => HashTable{point => HashTable{x1 => -}}
                                                                 8
                                                        x2 => 0
                            979                                       3
                            ---- => HashTable{point => HashTable{x1 => -    }}
                            2048                                      8
                                                                  979
                                                      x2 => ----
                                                                  2048
                           2003                                       3
                           ---- => HashTable{point => HashTable{x1 => -    }}
                           1024                                       8
                                                                   2003
                                                        x2 => ----
                                                                   1024
                                                            3
                            point => HashTable{x1 => -}
                                                            8
                                       2    55      2    27
                            polynomials => {x2  - --, - x2  + ---}
                                               64         512
```

```
      15              7967                                            15
      -- => HashTable{- ----- => HashTable{point => HashTable{x1 => --    }}}
      16              16384                                           16
                                                                   7967
                                                       x2 => - -----
                                                                   16384
                           16159                                      15
                         - ----- => HashTable{point => HashTable{x1 => --    }}
                           8192                                        16
                                                                    16159
                                                        x2 => - -----
                                                                    8192
                                                                  15
                            0 => HashTable{point => HashTable{x1 => --}}
                                                                  16
                                                        x2 => 0
                            7967                                      15
                            ----- => HashTable{point => HashTable{x1 => --   }}
                            16384                                     16
                                                                   7967
                                                        x2 => -----
                                                                   16384
                            16159                                     15
                            ----- => HashTable{point => HashTable{x1 => --   }}
                            8192                                      16
                                                                   16159
                                                        x2 => -----
                                                                   8192
                                                            15
                            point => HashTable{x1 => --}
                                                            16
                                       2    31      2    3375
                            polynomials => {x2  - ---, - x2  + ----}
                                               256         4096
      17              33                                            17
      -- => HashTable{- -- => HashTable{point => HashTable{x1 => --  }}}
      8               8                                             8
                                                                  33
                                                      x2 => - --
                                                                  8
                                                                17
                            0 => HashTable{point => HashTable{x1 => --}}
                                                                8
                                                        x2 => 0
                            33                                      17
                            -- => HashTable{point => HashTable{x1 => --}}
                            8                                       8
                                                                  33
                                                        x2 => --
                                                                  8
                                                            17
                            point => HashTable{x1 => --}
                                                            8
                                       2    225     2    4913
                            polynomials => {x2  + ---, - x2  + ----}
                                               64         512
          point => HashTable{}
                                      3    2
          polynomials => {x1 - 1, x1 + 1, x1, x1  + x1  - 1}
```
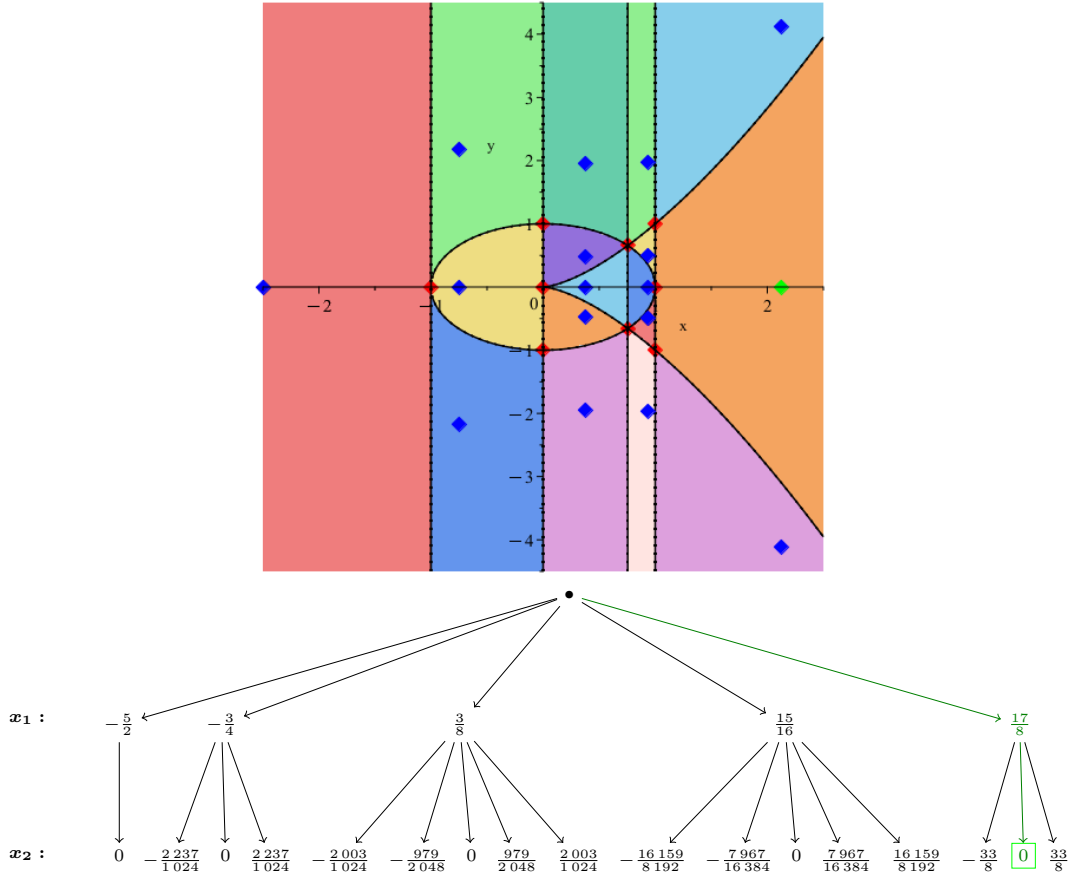
We are now able to see the full structure of `openCAD`: each level of this tree corresponds to a variable, listing sample points based on those of the level below. In this example we can see the full list of open sample points of the CAD of $\mathcal{F}$. For a deeper explanation of this structure see 3.4

3.3 Projection phase

The entire projection process is handled by the command `projectionPhase`, which takes the list of input polynomials $\mathcal{F}_n$ and returns the full collection of projection polynomial lists $\{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$, along with the corresponding variable ordering $x_1 \prec \cdots \prec x_n$. At each step, `gmodsHeuristic` selects a variable to eliminate, and `lazardProjection` constructs the resulting projection polynomials.

3.3.1 `gmodsHeuristic`

Input 1: A list of polynomials $\mathcal{F}_n = \{f_1, \ldots, f_r\}$ in n variables.
Input 2: The list of variables $x_1, \ldots, x_n$ appearing in $\mathcal{F}_n$.

Output: The variable to project as suggested by the `gmods` heuristic (see Section 2.4.1).

```
i10 : gmodsHeuristic(F,support(F))
```

```
o10 = x2
```

3.3.2 `lazardProjection`

Input 1: List of polynomials $\mathcal{F}_n = \{f_1, \ldots, f_r\}$ in variables $x_1, \ldots, x_n$.
Input 2: The variable to be projected away, typically the highest variable x_n.

Output: A list of projection polynomials $\mathcal{F}_{n-1}$ in $x_1, \ldots, x_{n-1}$, obtained via the *Lazard projection* (Definition 3) with respect to x_n.

This command ensures polynomials are irreducible and pairwise coprime using the subcommand `factorsInList` to break them into their unique irreducible factors, discarding constants. The Lazard projection is then applied with respect to x_n, returning the list of factors of the leading and trailing coefficients, discriminants and resultants.

Projecting $\mathcal{F}$ with respect to x_2 returns:

```
i11 : lazardProjection(F,x2)

                             3    2
o11 = {x1 - 1, x1 + 1, x1, x1  + x1  - 1}
```

where

- $x_1 - 1$ and $x_1 + 1$ are factors of the trailing coefficient and discriminant of f_1,

- x_1 is a factor of the trailing coefficient of f_2, and

- $x_1^3 + x_1^2 - 1$ is the resultant of f_1 and f_2.

3.3.3 `projectionPhase`

Input: List of polynomials $\mathcal{F}_n = \{f_1, \ldots, f_r\}$ in variables $x_1, \ldots, x_n$.

Output 1: List of projection polynomial lists $\{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$ (see Section 2 for more details).
Output 2: The associated variable ordering $x_1 \prec \cdots \prec x_n$.

```
i12 : projectionPhase(F)

                       3    2          2    2         3    2
o12 = ({{x1 - 1, x1 + 1, x1, x1  + x1  - 1}, {x1  + x2  - 1, x1  - x2 }}, {x1,x2})
```

3.4 Lifting phase

Like with the projection phase, the full lifting phase is handled by one command. Given a list of projection polynomial lists, their associated ordering, and a $(k-1)$-dimensional sample point representing the cell below, the command `liftingPoint` recursively constructs cells above each sample point until a full collection of n-dimensional sample points is obtained.

Ordinarily (as is the case for `openCAD`, see 3.1.1), an empty (0-dimensional) sample point representing the root node of the tree is used, leading to a CAD of $\mathbb{R}^n$. This command also allows the construction of a 'sub-CAD' of the space above any provided sample point.

Each sample point is stored as a hash table mapping variables to values. The command `liftingPoint` calls `evaluatePolynomials` to evaluate polynomials above the sample point, returning a list of univariate polynomials. This is then passed to `samplePoints`, which isolates roots and returns the open sample points of the space.

3.4.1 `samplePoints`

Input: List of univariate polynomials $\mathcal{F} = \{f_1, \ldots, f_r\} \subset \mathbb{R}[x]$.

Output: A list of open sample points for the CAD of the (one-dimensional) space containing $\mathcal{F}$.

This command applies `realRootIsolation` from `RealRoots` to the polynomials, isolating each real root within a rational interval, then returns sample points either side of these intervals. If applied to our projection polynomials $\mathcal{F}_1$, these are the open sample points of CAD$(\mathcal{F}_1)$, but if applied to $\mathcal{F}_k$ evaluated at a $(k-1)$-dimensional sample point, these are the open sample points of the CAD of the space above that point.

For example, we can obtain the open sample points of our projection polynomials obtained in 3.3.2:

```
i13 : samplePoints(lazardProjection(F,x2))

          5      3  3   15   17
o13 = {- -,  - -,  -,  --,  --}
          2      4  8   16    8
```

These are the values of x_1 seen in Figure 6.

3.4.2 evaluatePolynomials

Input 1: List of polynomials $\mathcal{F}_n = \{f_1, \ldots, f_r\}$ in variables $x_1, \ldots, x_n$.
Input 2: A k-dimensional point.

Output: The list of polynomials evaluated at this point.

Above one of these sample points, our new univariate polynomials become:

```
alpha = new MutableHashTable; alpha#x1 = -3/4;

i14 : evaluatePolynomials(F,alpha)

          2     7       2    27
o14 = {x2   - --,  - x2   - --}
              16            64
```

We can then look at the open sample points of x_2 above this:

```
i15 : samplePoints(evaluatePolynomials(F,alpha))

          2237        2237
o15 = {- ----,  0,  ----}
          1024        1024
```

3.4.3 liftingPoint

Input 1: List of projection polynomial lists $\{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$.
Input 2: The associated variable ordering $x_1 \prec \cdots \prec x_n$.
Input 3: A $(k-1)$-dimensional point.

Output: An Open CAD (see Section 2) over the given point.

Above each sample point of the previous level, a cell is constructed. This level k cell contains:

- the $(k-1)$ dimensional sample point below (`point`),

- the polynomials of $\mathcal{F}_k$ evaluated at this sample point (`polynomials`),

- the values of x_k for each k-dimensional sample point above `point`. The command `liftingPoint` is called again with these new sample points.

At the top level (level n), each cell only contains `point`, the sample point for that open cell.

Lifting above the previous sample point:

```
(PP,ord) = projectionPhase(F);

i16 : hashify(liftingPoint(PP,ord,alpha))
```

12

```
                    2237                                              3
o16 = HashTable{- ---- => HashTable{point => HashTable{x1 => - -    }}}
                    1024                                              4
                                                                  2237
                                                      x2 => - ----
                                                                  1024
                                                       3
             0 => HashTable{point => HashTable{x1 => - -}}
                                                       4
                                                 x2 => 0
             2237                                               3
             ---- => HashTable{point => HashTable{x1 => - - }}
             1024                                               4
                                                            2237
                                                x2 => ----
                                                            1024
                                         3
             point => HashTable{x1 => - -}
                                         4
                         2    7      2    27
             polynomials => {x2  - --, - x2   - --}
                             16           64
```

This demonstrates the lifting phase: we first isolate their real roots of the univariate polynomials in $\mathcal{F}_1$ and select sample points from each open cell around the roots. These open sample points are then substituted into $\mathcal{F}_2$ to produce a new list of univariate polynomials, and the process repeats until a full Open CAD is constructed.

4 Performance - Hyperspheres

It is known that CAD has doubly exponential complexity, meaning that only problems in fewer variables can be solved. To measure the performance of our package, we created CADs for the following intersecting n-dimensional spheres (of which the three-dimensional case is depicted in Figure 7):

$$\mathcal{F}_n = \left\{ \sum_{i=1}^n (x_i - 1)^2 - 4, \sum_{i=1}^n (x_i + 1)^2 - 4 \right\}$$

The runtimes achieved by the package are the following:

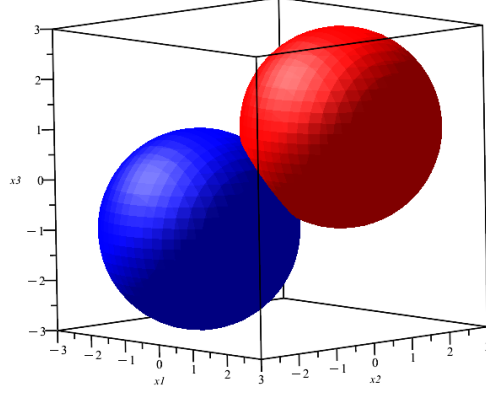| Variables	Runtime (seconds)	n-cells
1	0.0308762	5
2	0.335376	29
3	6.06833	467
4	1883.13	7370

Figure 7: A plot of the algebraic variety of $\mathcal{F}_n$.
Two intersecting spheres centered at (1,1,1) and (-1,-1,-1) respectively.

5 Future work

Some possible directions for expanding the package include:

- **Improved Output:** Enhancing `liftingPoint` by evaluating the input polynomials at the top-level sample points, providing more information to users and streamlining `positivePoint`. This would also simplify manual sign-checking of cells and future related commands.

- **Extension to Full CAD:** Expanding from Open CAD to a full CAD with cells of every dimension. This would require the implementation of real polynomial division and a way to symbolically store and retrieve real roots, but would enable the following two applications:

- **Quantifier Elimination:** A full CAD could be used to answer any nonlinear arithmetic quantified questions through quantifier elimination (given sufficient resources).

- **Connectivity Analysis of Algebraic Varieties:** By combining a full CAD with an adjacency algorithm, such as the one described in [Str17], one could determine the number of connected components in algebraic varieties, offering new insights into their topological properties.

These ideas represent potential avenues for further research and development in the domain, aimed at enhancing our understanding and capabilities in computational algebraic geometry.

References

[BD07]   Christopher W. Brown and James H. Davenport. The complexity of quantifier elimination and cylindrical algebraic decomposition. In *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, ISSAC '07, page 54–60, New York, NY, USA, 2007. Association for Computing Machinery.

[Bro04]   C. W. Brown. ISSAC 2004 Tutorial: Cylindrical Algebraic Decomposition. CADs More Formally. https://www.usna.edu/Users/cs/wcbrown/research/ISSAC04/slides3F.pdf, 2004. Figure 3, Accessed: 2025-02-28.

[Col75]   George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In H. Brakhage, editor, *Automata Theory and Formal Languages*, pages 134–183, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg.

[CZC20]   Changbo Chen, Zhangpeng Zhu, and Haoyu Chi. Variable Ordering Selection for Cylindrical Algebraic Decomposition with Artificial Neural Networks. In Anna Maria Bigatti, Jacques Carette, James H. Davenport, Michael Joswig, and Timo de Wolff, editors, *Mathematical Software – ICMS 2020*, pages 281–291, Cham, 2020. Springer International Publishing.

[DH88]   James H. Davenport and Joos Heintz. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5(1):29–35, 1988.

[dRE22]   Tereso del Río and Matthew England. New Heuristic to Choose a Cylindrical Algebraic Decomposition Variable Ordering Motivated by Complexity Analysis. In François Boulier, Matthew England, Timur M. Sadykov, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, pages 300–317, Cham, 2022. Springer International Publishing.

[EF19]   Matthew England and Dorian Florescu. Comparing Machine Learning Models to Choose the Variable Ordering for Cylindrical Algebraic Decomposition. In Cezary Kaliszyk, Edwin Brady, Andrea Kohlhase, and Claudio Sacerdoti Coen, editors, *Intelligent Computer Mathematics*, pages 93–108, Cham, 2019. Springer International Publishing.

[GMSY24]   Jordy Lopez Garcia, Kelly Maluccio, Frank Sottile, and Thomas Yahl. Real solutions to systems of polynomial equations in Macaulay2. *Journal of Software for Algebra and Geometry*, 14(1):87–95, 2024.

[GS]   Daniel R. Grayson and Michael E. Stillman. Macaulay2, a software system for research in algebraic geometry. Available at http://www2.macaulay2.com.

[HEW⁺15]   Zongyan Huang, Matthew England, David Wilson, James H. Davenport, and Lawrence C. Paulson. A Comparison of Three Heuristics to Choose the Variable Ordering for Cylindrical Algebraic Decomposition. *ACM Communications in Computer Algebra*, 48(3/4):121–123, February 2015.

[Jir95]   Mats Jirstrand. Cylindrical Algebraic Decomposition - an Introduction. Technical Report 1807, Linköping University, Department of Electrical Engineering, Automatic Control, 1995.

[Laz94]   D. Lazard. *An Improved Projection for Cylindrical Algebraic Decomposition*, volume 26 of *Springer Series in Computational Mathematics*, pages 467–476. Springer New York, New York, NY, 1994.

[MPP19]   Scott McCallum, Adam Parusiński, and Laurentiu Paunescu. Validity proof of Lazard's method for CAD construction. *Journal of Symbolic Computation*, 92:52–69, 2019.

[Str00]   Adam Strzeboński. Solving Systems of Strict Polynomial Inequalities. *Journal of Symbolic Computation*, 29(3):471–480, 2000.

[Str17]   Adam Strzeboński. CAD Adjacency Computation Using Validated Numerics. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computa-

tion, ISSAC '17, page 413–420, New York, NY, USA, 2017. Association for Computing Machinery.

[Tar51]   Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry: Prepared for Publication with the Assistance of J.C.C. McKinsey.* RAND Corporation, Santa Monica, CA, 1951.