Annales Univ. Sci. Budapest., Sect. Comp. 51 (2020) 231–252

ON PROVING INEQUALITIES BY CYLINDRICAL ALGEBRAIC DECOMPOSITION

Marcell János Uray (Budapest, Hungary)

Dedicated to the 70th birthday of Professor Antal Járai

Communicated by Imre Kátai

(Received January 30, 2020; accepted April 18, 2020)

Abstract. Cylindrical algebraic decomposition (CAD) is a basic concept in real algebraic geometry, and it has useful applications to deal with symbolic inequalities. We present a new implementation of CAD in the SageMath computer algebra system. This is not as fast as some existing implementations like QEPCAD, but it is more flexible to be embedded in certain larger calculations. One such application of CAD is a proving procedure for inequalities involving recursive functions, invented by Gerhold and Kauers. We present an implementation of this algorithm as well. This paper also gives an overview with examples about the theory behind the implemented algorithms.

1. Introduction

Collins invented cylindrical algebraic decomposition (CAD) in 1973 [5]. Its main motivation was quantifier elimination, i.e. given a quantified formula, it creates an equivalent formula without quantifiers. CAD has also several other applications regarding inequalities, some of them are discussed in this paper.

Key words and phrases: Cylindrical algebraic decomposition, inequalities, Sage package.
2010 Mathematics Subject Classification: 68W30, 13P15.
The work of the author was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-2016-2017-00002).
https://doi.org/10.71352/ac.51.231

There are already a few implementations of CAD, including QEPCAD, which is a free software written in C/C++ by Hong, Brown and others [3]; the proprietrary software Mathematica which contains the built-in command CylindricalDecomposition implemented by Strzeboński [4]; and Redlog [7] also has an implementation.

Our attention is on the SageMath computer algebra system (which will be referred to simply as Sage). It has an optional package named qepcad, which is an interface to the QEPCAD software and thus provides cylindrical algebraic decomposition in Sage [11]. That package has a strong emphasis on giving access to the rich features of QEPCAD, but it is less useful when we want to manipulate logical formulas and use CAD as a subroutine. For example, though it provides a way to build quantified logical formulas from Sage expressions, however, to the best of our knowledge, it does not support manipulating these formulas, and the output of QEPCAD is returned as a string instead of a mathematical object. Another problem is that unlike e.g. Mathematica's implementation, QEPCAD (and Sage's interface) does not support special expressions like square roots or fractions in formulas, only polynomials.

In this paper we present a new Sage package for CAD, developed by the author. It has two modes: it implements CAD on its own, but it can also invoke QEPCAD if it is installed on the machine. This package (in both modes) solves the issues mentioned above, though it has fewer capabilities to use QEPCAD's special commands. It only uses it as a component for CAD interchangably with its own implementation.

Moreover, this paper discusses a specific application of CAD: Gerhold and Kauers invented an algorithm for proving certain types of inequalities involving recursive sequences [9, 13]. Kauers implemented this algorithm in his package SumCracker [15], written in Mathematica. We give a new implementation of this in our Sage package.

The paper is built up as follows. In Section 2, we discuss the core of cylindrical algebraic decomposition. Section 3 is about using CAD in formula conversion (including quantifier elimination). Section 4 is devoted to Gerhold's and Kauers' inequality proving algorithm. In all these sections, first we discuss the theoretical background with examples, then we present our implementation briefly. Finally, in Section 5, we compare the CAD implementation with QEPCAD and give some possible future directions.

The presented Sage package, along with some test cases, is available at the online version of this article [18]. They were tested in Sage 8.9, on a machine with Pentium Dual-Core 3 GHz processor and 4 GB RAM. This paper is indended to be a general overview of the package, the full user documentation is included in the package as documentation strings, and the implementation details are described in comments. The package is distributed under the GNU General Public License 3.

2. Cylindrical algebraic decomposition

2.1. Definition of CAD

Cylindrical algebraic decomposition is a special decomposition of $\mathbb{R}^n$ into so-called cells, induced by some polynomials. More precisely:

Definition 2.1. A *decomposition* of $\mathbb{R}^n$ is a finite number of nonempty, connected and pairwise disjoint sets, $C_1, C_2, \ldots, C_N$, whose union is $\mathbb{R}^n$. These sets are called *cells*.

Definition 2.2. An *algebraic decomposition* of $\mathbb{R}^n$ is a decomposition for which there exists a finite number of n-variate polynomials $p_1, p_2, \ldots, p_m$ which all have constant sign on each cell, but not on the union of any two adjacent cells.

Definition 2.3. A *cylindrical algebraic decomposition* (CAD) of $\mathbb{R}^n$ is an algebraic decomposition which, when projected to $\mathbb{R}^{n-1}$, has the following two properties: any two cells have either the same or disjoint projections, and the distinct projected cells constitute a cylindrical algebraic decomposition of $\mathbb{R}^{n-1}$. A CAD of $\mathbb{R}^0$ is its (only) trivial algebraic decomposition.

Example 2.4. For example, the CAD of $\mathbb{R}^2$ induced by the single polynomial $x^2 + y^2 - 1 \in \mathbb{R}[x, y]$ consists of the following 13 cells (5 regions, 6 curves and 2 points):

$$\{(x,y) \in \mathbb{R}^2 \mid x < 1\},$$
$$\{(x,y) \in \mathbb{R}^2 \mid x = -1 \wedge y < 0\},$$
$$\{(x,y) \in \mathbb{R}^2 \mid x = -1 \wedge y = 0\},$$
$$\{(x,y) \in \mathbb{R}^2 \mid x = -1 \wedge y > 0\},$$
$$\{(x,y) \in \mathbb{R}^2 \mid x > -1 \wedge x < 1 \wedge y < -\sqrt{1-x^2}\},$$
$$\{(x,y) \in \mathbb{R}^2 \mid x > -1 \wedge x < 1 \wedge y = -\sqrt{1-x^2}\},$$
$$\{(x,y) \in \mathbb{R}^2 \mid x > -1 \wedge x < 1 \wedge y > -\sqrt{1-x^2} \wedge y < \sqrt{1-x^2}\},$$
$$\{(x,y) \in \mathbb{R}^2 \mid x > -1 \wedge x < 1 \wedge y = \sqrt{1-x^2}\},$$
$$\{(x,y) \in \mathbb{R}^2 \mid x > -1 \wedge x < 1 \wedge y > \sqrt{1-x^2}\},$$
$$\{(x,y) \in \mathbb{R}^2 \mid x = 1 \wedge y < 0\},$$
$$\{(x,y) \in \mathbb{R}^2 \mid x = 1 \wedge y = 0\},$$
$$\{(x,y) \in \mathbb{R}^2 \mid x = 1 \wedge y > 0\},$$
$$\{(x,y) \in \mathbb{R}^2 \mid x > 1\}.$$

2.2. CAD algorithm

Now we describe the CAD decomposition algorithm invented by Collins [5]. It takes a finite set of polynomials $\mathcal{P} \subseteq \mathbb{A}[x_1, x_2, \ldots, x_n]$ as input (where $\mathbb{A}$ denotes the set of real algebraic numbers), and returns the cells of the cylindrical algebraic decomposition induced by these polynomials.

The skeleton of the algorithm is the following:

1. Find the splitting points along x_1, which decompose $\mathbb{R}$ to cells (intervals and points) which are the projection of the final CAD to the line of x_1.
2. Pick a sample $x_1 \in \mathbb{A}$ (possibly $x_1 \in \mathbb{Q}$) from each of these cells.
3. For each cell, substitute the sample x_1 to the input to get polynomials in $\mathbb{A}[x_2, \ldots, x_n]$.
4. Construct the CAD recursively for these polynomials in $\mathbb{R}^{n-1}$.

Of course, the crutial part is the first step, i.e. how to get the splitting points for x_1. We need so small cells that within any given cell, all points behave basically in the same way, i.e. any sample point can be used to find out the structure of the whole cell.

For this, we use a *CAD projection operator*, which transforms a finite subset of $\mathbb{A}[x_1, \ldots, x_{n-1}, x_n]$ to a finite subset of $\mathbb{A}[x_1, \ldots, x_{n-1}]$. The goal of this operator is that the roots of the output polynomials contain all "interesting" points of the input polynomials, i.e. all $(x_1, \ldots, x_{n-1})$ for which the input, viewing them as a set of univariate polynomials in x_n, may change their nature. This may mean that one of the polynomial changes the number of distinct real roots, or two of them intersect.

For example, a CAD projection operator may project $\{x^2 + y^2 + z^2 - 1\}$ to $\{x^2+y^2-1\}$, because the "interesting" points of the unit sphere when projecting down to the (x, y)-plane are exactly the unit circle. Projecting $\{x^2 + y^2 - 1\}$ further would give $\{x^2 - 1\}$, or equivalently $\{x - 1, x + 1\}$. There are several different CAD projection operators, which we describe in the next section.

Step 1 of the algorithm above, using a CAD projection operator, can be implemented as follows. Starting from the input polynomial set $\mathcal{P} = \mathcal{P}_n$, use the CAD projection operator iteratively to get $\mathcal{P}_n \mapsto \mathcal{P}_{n-1} \mapsto \ldots \mapsto \mathcal{P}_1$, where each $\mathcal{P}_i \subseteq \mathbb{A}[x_1, \ldots, x_i]$. Then the splitting points for x_1 will be the roots of $\mathcal{P}_1$. For the recursive CAD decomposition in step 4., when calculating the splitting points for x_2, x_3 etc., there is no need to repeat this CAD projection process again for the new input polynomials, but instead it is enough to substitute the sample x_1 to $\mathcal{P}_2$, the sample x_1 and x_2 to $\mathcal{P}_3$ etc. to get univariate polynomials in x_2, x_3 etc., whose roots will be the splitting points for those steps.

2.3. CAD projection operator

Let $\mathcal{P} = \{p_1, p_2, \ldots, p_m\} \subseteq \mathbb{A}[x_1, \ldots, x_n]$ be a finite set of polynomials in n variables. We consider them as polynomials in x_n over $\mathbb{A}[x_1, \ldots, x_{n-1}]$. The CAD projection operator takes $\mathcal{P}$ as input, and gives polynomials $\mathrm{proj}(\mathcal{P}) \subseteq \mathbb{A}[x_1, \ldots, x_{n-1}]$ such that their induced CAD in $\mathbb{R}^{n-1}$ can be extended to a valid CAD for $\mathcal{P}$ in $\mathbb{R}^n$. Loosely speaking, this means that over any given cell of the $\mathbb{R}^{n-1}$-CAD, the polynomials in $\mathcal{P}$ behave in a similar way over all points of that cell. This property is called the *delineability* of the roots of $\mathcal{P}$ over the cell, and its precise definition and related theorems can be found e.g. in [5, p. 139]. Basically, the following three problems can prevent delineability:

1. some $p_i \in \mathcal{P}$ changes its degree;
2. some $p_i \in \mathcal{P}$ changes the number of distinct roots while preserving its degree;
3. two polynomials in $\mathcal{P}$ have common roots.

Therefore, a naive approach to address these problems would be to return the following polynomials as $\mathrm{proj}(\mathcal{P})$:

1. the leading coefficients of all $p_i \in \mathcal{P}$;
2. the discriminants of all $p_i \in \mathcal{P}$, which is, up to a constant, $\mathrm{res}(p_i, p_i')$;
3. $\mathrm{res}(p_i, p_j)$ for all distinct pairs of polynomials.

But this is not sufficient. For example, if $p \in \mathcal{P}$ has degree d, but in a one-dimensional cell, the leading coefficient of p becomes everywhere zero, then within that cell, problem 1 is independent from the (original) leading coefficient, but instead depends on the next coefficient. A similar problem arises when dealing with the resultants. It is zero if and only if the two polynomials have a common root. It may happen that two polynomials have a common root everywhere within a one-dimensional cell, and two common roots in exactly one point of that cell. Then this latter point cannot be detected by resultants. Therefore we need its generalization, the principal subresultant coefficients: $\mathrm{psc}_k(f, g) = 0$ if and only if f and g have at least $k + 1$ common roots, and $\mathrm{psc}_0(f, g) = \mathrm{res}(f, g)$. Resultants and principal subresultant coefficients are described e.g. in [8, Chapter 7.3].

There are several approaches to create a correct CAD projection operator. The simpliest is Collins' simple projection operator, which is a brute-force approach to address the issues mentioned above. Let $\mathrm{red}_l(p)$ be the lth reductum of p, i.e. the polynomial obtained from p by removing the l highest powers of x_n. More precisely: $\mathrm{red}_0(p) = p$ and $\mathrm{red}_{l+1}(p) = \mathrm{red}_l(p - \mathrm{lc}(p))$ where $\mathrm{lc}(p)$ is the leading coefficient of p. Then Collins' simple projection operator returns the following polynomials [5, p. 142]:

1. all coefficients of all $p_i \in \mathcal{P}$;
2. $\mathrm{psc}_k(\mathrm{red}_l(p_i), \mathrm{red}_l(p_i)')$ for all i, l, k;
3. $\mathrm{psc}_k\big(\mathrm{red}_{l_i}(p_i), \mathrm{red}_{l_j}(p_j)\big)$ for all i, l_i, j, l_j, k.

This operator simply considers all reducta of the polynomials and all principal subresultant coefficients. This often results in a huge amount of polynomials, and the vast majority of them are unneccessary and create superfluously many cells. However, it is easy to implement.

Collins himself proposed an improved version, noticing that we can stop taking reductions and psc's when it is safe to do so [5, p. 176]. For example, if a coefficient is a nonzero constant, we can stop taking reductions since the degree will never drop below this coefficient. More generally, if we can prove that the first few coefficients have no common roots, or even only finitely many (as individual vanishing points do not cause any problem), then we can stop. The same simplification applies to the sequence of psc's.

McCallum reduces the projection operator further, but at the same time, makes difficulties with the other parts of the algorithm. His proposed projection is the following [16]:

1. all coefficients of all $p_i \in \mathcal{P}$ (with Collins' improvement);
2. the discriminants of all $p_i \in \mathcal{P}$;
3. $\mathrm{res}(p_i, p_j)$ for all distinct pairs of polynomials.

This is almost the same as the naive approach (only the first one differs), but there are some restrictions about the algorithm to make this valid. First, the input polynomials must be squarefree and pairwise disjoint. This is not a real restriction as it can be achieved e.g. by factorization and removing duplicates. Note that for the earlier projection operators, although factorization is not neccessary, it may also improve the efficiency of the algorithm. The other problem with McCallum's projection is that it works only for well-oriented input, which means that the polynomials in the input or after projection have no vertical lines contained in their zero set (i.e. along the last variable), or there are only finite many of them [16, p. 258], where the latter case is handled by a modification of the algorithm. Up to three variables, all inputs are well-oriented [17], and in higher dimensions most inputs fulfil this property either.

Brown improved McCallum's projection operator even further, removing the non-leading coefficients of the polynomials, thus making it formally the same as the naive version. In the same time, it requires further changes in the CAD algorithm, namely inserting certain individual points into the decomposition which are not roots of any projection polynomials, but are calculated separately. [1]

We illustrate the point of improving the projection operator by an example from [16, Section 7.1]. Let $\mathcal{P} = \{x^2 + y^2 + z^2 - 1, z^3 + xz + y\} \in \mathbb{Q}[x, y, z]$. Our implementation, which is to be described in the next section, performs as follows on $\mathcal{P}$ using different projection operators:

Projection operator	$\lvert\mathcal{P}_2\rvert, \lvert\mathcal{P}_1\rvert, \lvert\mathcal{P}_0\rvert$	**Cells**	**Running time**
Collins' simple	$2, 6, 30$	*unknown*	> 1 hour
Collins' improved	$2, 4, 10$	1393	62 sec
McCallum's	$2, 3, 9$	971	33 sec

The test with Collins' simple projection operator did not terminate within an hour.

2.4. CAD in the package

In our Sage package, the CAD implementation is invoked by the function `cylindrical_algebraic_decomposition()` from `cad.sage`. It expects a list of polynomials or a single polynomial. Here is the execution of Example 2.4:

```
sage:  var('x,y');
sage:  cad = cylindrical_algebraic_decomposition(x^2 + y^2 - 1)
sage:  for cell in cad.cells():
          print cell.dimension(), cell.sample(), cell.formula()
2 (-2, 0) x < -1
1 (-1, -1) x == -1 /\ y < -sqrt(-x^2 + 1)
0 (-1, 0) x == -1 /\ y == -sqrt(-x^2 + 1)
1 (-1, 1) x == -1 /\ y > -sqrt(-x^2 + 1)
2 (0, -2) x > -1 /\ x < 1 /\ y < -sqrt(-x^2 + 1)
1 (0, -1) x > -1 /\ x < 1 /\ y == -sqrt(-x^2 + 1)
2 (0, 0) x > -1 /\ x < 1 /\ y > -sqrt(-x^2 + 1) /\ y < sqrt(-x^2 + 1)
1 (0, 1) x > -1 /\ x < 1 /\ y == sqrt(-x^2 + 1)
2 (0, 2) x > -1 /\ x < 1 /\ y > sqrt(-x^2 + 1)
1 (1, -1) x == 1 /\ y < -sqrt(-x^2 + 1)
0 (1, 0) x == 1 /\ y == -sqrt(-x^2 + 1)
1 (1, 1) x == 1 /\ y > -sqrt(-x^2 + 1)
2 (2, 0) x > 1
```

The essence of cylindrical algebraic decomposition is implemented in the `CADCell` class (in `cad-cell.sage`). It represents a single cell of the CAD in $\mathbb{R}^n$, or any of its projection in $\mathbb{R}^{n-1}, \ldots, \mathbb{R}^0$. Each projection `CADCell` stores the list of all `CADCell`s in one higher dimension whose projection it is. This recursively gives a tree of all higher-dimensional cells over this cell. An important method of `CADCell` is `CADCell._split()`, which, for a cell in $\mathbb{R}^{k-1}$, constructs its children cells in $\mathbb{R}^k$ using the algorithm described in Section 2.2.

The function that puts everything together is `cad_polys_to_cells()`. First it generates the projections of the input polynomials iteratively. All polynomials, either from the input or from the projection, are factored, and constant fac-

tors and duplicates are removed (this is implemented by the `PolyFactorList` class in `poly.sage`). Then, it creates the single cell of $\mathbb{R}^0$ as a starting point, and gerenates the decomposition by calling repeatedly `CADCell._split()`. It finally returns the single cell of $\mathbb{R}^0$, which then contains the tree of the complete decomposition.

The package implements three projection operators: the two versions of Collins' projection operator and McCallum's projection. They are all defined at the end of `cad.sage`.

3. Formula conversion

Cylindrical algebraic decomposition can be used to handle semi-algebraic sets, i.e. subsets of $\mathbb{R}^n$ described by polynomial equalities and inequalities. Using CAD, we can convert such a formula into an equivalent form having a very special structure, which gives useful information about the set, such as whether it is empty, finite, bounded etc., its dimension, the number of components, a sample point in each component etc. This formula conversion itself, which is an application of CAD, is sometimes also called cylindrical algebraic decomposition [14, 4].

Example 3.1. $x^2+y^2+z^2 \leq 2$ is converted by CAD to the following equivalent formula:

$$
\begin{aligned}
&(x = -\sqrt{2} \wedge y = 0 \wedge z = 0) \vee \\
&(x > -\sqrt{2} \wedge x < \sqrt{2} \wedge (\\
&\quad (y = -\sqrt{2 - x^2} \wedge z = 0) \vee \\
&\quad (y > -\sqrt{2 - x^2} \wedge y < \sqrt{2 - x^2} \wedge z \geq -\sqrt{2 - x^2 - y^2} \wedge z \leq \sqrt{2 - x^2 - y^2}) \vee \\
&\quad (y = \sqrt{2 - x^2} \wedge z = 0) \\
&)) \vee \\
&(x = \sqrt{2} \wedge y = 0 \wedge z = 0)
\end{aligned}
$$

This form directly gives some information about the set (of course we do not need CAD for such an easy example, this is just an illustration):

- it has points only for $-\sqrt{2} \leq x \leq \sqrt{2}$;
- for either of $x = \pm\sqrt{2}$, there is exactly one point in the set;
- for $-\sqrt{2} < x < \sqrt{2}$, there are infinitely many possible y values;
- for any such x, the formula directly gives the possible y values, and for any such x and y, it gives the possible z values;

- the set is three-dimensional;
- the set is bounded;
- its exact bounding box is $[-\sqrt{2}, \sqrt{2}]^3$.

CAD can also deal with formulas with quantifiers, and convert them to equivalent quantifier-free form. This is called quantifier elimination, and it is one of the major applications of CAD.

Example 3.2. The formula $\exists y : \exists z : x^2 + y^2 + z^2 \leq 2$ is converted to:

$$x \geq -\sqrt{2} \wedge x \leq \sqrt{2}.$$

Now we define the structure of the formulas precisely. The input has a very general form:

Definition 3.3. A *Tarski formula* is a logical formula consisting of variables, rational numbers, three arithmetic operators ($+$, $-$, $\cdot$), relational symbols ($<$, $>$, $\leq$, $\geq$, $=$, $\neq$), logical operators ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$), logical constants (*true*, *false*) and quantifiers ($\forall$, $\exists$) built according to the standard mathematical syntax.

In short, Tarski formulas are logical combination of polynomial equalities and inequalities such as $\exists x : x > 1 \wedge (x+2)^2 y^3 > x - y^2$. The output of CAD is an equivalent formula of the following special form [14]:

Definition 3.4. A *CAD formula* in n variables $x_1, x_2, \ldots, x_n$ is of the form $(\phi_1 \wedge \psi_1) \vee (\phi_2 \wedge \psi_2) \vee \ldots \vee (\phi_m \wedge \psi_m)$, where

1. ϕ_i are univariate in x_1;
2. each ϕ_i describes a non-empty interval or point of $\mathbb{R}$, bounded by algebraic numbers (examples: $x_1 > \alpha$, $x_1 \leq \beta$, $x_1 \geq \alpha \wedge x_1 < \beta$, $x_1 = \alpha$, *true* etc., where $\alpha < \beta$ are algebraic numbers);
3. all $\phi_i \wedge \phi_j$ are inconsistent for $i \neq j$;
4. and all ψ_i, when x_1 is replaced by any γ algebraic number for which $\phi_i(\gamma)$ holds, is a satisfiable CAD formula in $x_2, \ldots, x_n$.

A CAD formula in 0 variables is either *true* or *false*.

Note that a CAD formula can be a logical constant also for $n > 0$: if $m = 0$, i.e. for disjunction of length zero, it is considered constant *false*; if $m = 1$ and $\phi_1 = \mathit{true}$ recursively, we get constant *true*. Conversely, the only unsatisfiable CAD formula is the constant *false*, so converting to a CAD formula gives a satisfiability test. The constant *true* is not so simple, e.g. $x \geq 1 \vee (x < 1 \wedge (y > x \vee y \leq x))$ is a CAD formula equivalent to *true*, but such formulas can be easily simplified to *true*. We assume that the CAD conversion implementation makes this simplification automatically.

3.1. Conversion algorithm

How can we convert a Tarski formula to a CAD formula using cylindrical algebraic decomposition (as defined in Definition 2.3)?

First, consider Example 2.4 about the CAD of $x^2 + y^2 - 1$, and notice that all cells were described by CAD formulas. It can be done in general, i.e. in any given CAD decomposition, it is straightforward to describe the cells by CAD formulas. Using this, we can convert a quantifier-free Tarski formula to a CAD formula by the following steps:

1. Extract all polynomials from the Tarski formula after reducing all atomic formulas to zero on the right (e.g. extract $f(x) - g(x)$ from $f(x) < g(x)$).
2. Calculate the CAD decomposition induced by these polynomials.
3. Select those cells which satisfy the input formula by evaluating it on one sample point in each cell.
4. Construct the CAD formula for these cells.
5. Combine these formulas by disjunction and simplify them using the rule of distributivity.

Note that due to the sign-invariance of the CAD (by Definition 2.2), the input formula has constant truth value on each cell, so Step 3 indeed works. The simplification in Step 5 ensures requirement 3 of Definition 3.4. Note that Steps 4-5 can be combined by creating the simplified formula directly.

It is not hard to extend the algorithm to quantified Tarski formulas. It must be ensured that the variables are ordered so that the quantified variables come last (i.e. deepest in the CAD tree), and then it is straightforward to evaluate the cells according to the quantifiers, and the formula is built for the free variables as described above.

When CAD is used to convert formulas, especially quantified formulas, lazy evaluation can optimize the process by creating only those cells which contribute to the result, thus creating only a partial decomposition. If the formula certainly has a constant truth value on some cell, there is no need to calculate its subcells. For example, if the input formula is like $x_1 > 1 \wedge \phi$, then the cells $x_1 < 1$ and $x_1 = 1$ need no further subdivision.

3.2. More about CAD formula

Recall Example 3.1, which converted $x^2 + y^2 + z^2 \leq 2$ to a CAD formula. The result contained square roots, which showed that a CAD formula is not neccessarily a Tarski formula. It is a problem if we want to feed an output CAD formula as input for another CAD (in a slightly modified form), as we will want to do in Section 4.

But there is another problem: in general, not even radicals are sufficient to express roots of polynomials. For example, consider the input formula $x^6+y^6 < y$. For arbitrary x, the solution for y cannot be expressed in terms of radicals. We can formally solve this problem by introducing a new notation:

Definition 3.5. For a polynomial $p \in \mathbb{R}[x_1, x_2, \ldots, x_n, x]$, let $\mathrm{rootof}(p, x, i)$ be a function in $x_1, x_2, \ldots, x_n$ whose value is the ith distinct real root of $p(x_1, x_2, \ldots, x_n, x)$ as a univariate polynomial in x, with the natural ordering of the roots, counted from 0. The value is undefined for negative i or for those $x_1, x_2, \ldots, x_n$ where the polynomials has $\leq i$ distinct real roots.

(This notation is based on [2, Definition 17.], but slightly modified.)

Example 3.6.

$$\mathrm{rootof}(x^2+y^2-1, y, 0) = -\sqrt{1-x^2} \qquad (-1 \leq x \leq 1)$$
$$\mathrm{rootof}(x^2+y^2-1, y, 1) = +\sqrt{1-x^2} \qquad (-1 < x < 1)$$

This notation completes the notion of CAD formula.

Example 3.7. $x^6+y^6 < y$ can be converted to the following equivalent CAD formula:

$$x > -\frac{\sqrt[6]{5}}{\sqrt[5]{6}} \wedge x < \frac{\sqrt[6]{5}}{\sqrt[5]{6}} \wedge$$
$$\wedge\, y > \mathrm{rootof}(x^6+y^6-y, y, 0) \wedge y < \mathrm{rootof}(x^6+y^6-y, y, 1).$$

But this still does not solve the problem that a CAD formula is not a Tarski formula. For this, we generalize CAD formulas to allow implicit expressions (cf. Definition 3.4):

Definition 3.8. A *generalized CAD formula* in n variables $x_1, x_2, \ldots, x_n$ is of the form $(\phi_1 \wedge \psi_1) \vee (\phi_2 \wedge \psi_2) \vee \ldots \vee (\phi_m \wedge \psi_m)$, where

1. ϕ_i are univariate in x_1;
2. each ϕ_i describes a non-empty subset of $\mathbb{R}$ by logical combinations of equalities and inequalities involving only algebraic numbers (e.g. $x_1^2 + \alpha x_1 > \beta$);
3. all $\phi_i \wedge \phi_j$ are inconsistent for $i \neq j$;
4. and all ψ_i, when x_1 is replaced by any γ algebraic number for which $\phi_i(\gamma)$ holds, is a satisfiable generalized CAD formula in $x_2, \ldots, x_n$.

A generalized CAD formula in 0 variables is either *true* or *false*.

Note that the difference from the CAD formula is that ϕ_i are not neccessarily linear. This allows the elimination of radicals and rootof-expressions:

Definition 3.9. A *polynomial CAD formula* is a generalized CAD formula which contains only polynomial expressions.

Now a polynomial CAD formula is also a Tarski formula.

Example 3.10. $x^6+y^6 < y$ (cf. Example 3.7) can be converted to the following polynomial CAD formula:

$$x > -\frac{\sqrt[6]{5}}{\sqrt[5]{6}} \wedge x < \frac{\sqrt[6]{5}}{\sqrt[5]{6}} \wedge x^6 + y^6 < y,$$

or equivalently:

$$x^{30} < \frac{5^5}{6^6} \wedge x^6 + y^6 < y.$$

Example 3.11. $x^2+y^2+z^2 \leq 2$ has the following equivalent polynomial CAD formula:

$$\begin{aligned}
&(x^2 = 2 \wedge y = 0 \wedge z = 0) \vee \\
&(x^2 < 2 \wedge (\\
&\qquad (x^2 + y^2 = 2 \wedge z = 0) \vee \\
&\qquad (x^2 + y^2 < 2 \wedge x^2 + y^2 + z^2 \leq 2) \\
&))
\end{aligned}$$

Compared to the equivalent CAD formula in Example 3.1, most information can still be extracted from this, but in a less explicit way, so the exact values can be determined by finding roots of univariate polynomials.

Creating a polynomial CAD formula is much more difficult than creating a CAD formula. The basic idea is to try to use the existing polynomials, i.e. from the input and its projections, maybe after factorization. On each level, we try to characterize each cell by the signs of the available polynomials. Or rather, not neccessarily individual cells, but combination of cells whose subformulas are identical. E.g. the cells $x = -\sqrt{2}$ and $x = \sqrt{2}$ in the example above could be combined, because they both have the subformula $y = 0 \wedge z = 0$, and their common characterization is that $x^2 - 2$ has zero sign. If these two cells had different subformula, they would need to be differentiated by introducing a new polynomial, such as x, whose sign decides between the two. A CAD is called *projection definable* if the available polynomials are sufficient to create a polynomial CAD formula. If not, Brown developed an algorithm [2, Chapter 4.]

that calculates the neccessary new polynomials to be added, keeping their number as low as possible. Its basic tool is taking derivatives, e.g. x above is (up to a constant) the derivative of $x^2 - 2$. Practical experiences show however that most CAD's are projection definable, so an implementation might choose not to implement Brown's algorithm, making the formula conversion only partial. An interesting tradeoff is that the better (i.e. smaller) the projection operator we use, the more likely that we get a projection-undefinable CAD is, because we have less available projection polynomials.

3.3. Extended Tarski formula

The algorithm described so far expects a Tarski formula as input, i.e. logical combination of polynomial equalities and inequalities (Definition 3.3). But we can extend the formula conversion to the following more general input:

Definition 3.12. An *extended Tarski formula* is a logical formula consisting of variables, rational numbers, the four arithmetic operators ($+$, $-$, $\cdot$, $/$), the mathematical functions min, max, absolute value, square root and nth root, relational symbols ($<$, $>$, $\leq$, $\geq$, $=$, $\neq$), logical operators ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$), logical constants (*true*, *false*) and quantifiers ($\forall$, $\exists$) built according to the standard mathematical syntax.

(Note that this notion is different from the extended Tarski formula in [2, Definition 18.], which is used by QEPCAD.)

Note that the difference from the Tarski formula is that the extended version supports wider range of mathematical functions than just polynomials. It does not make the CAD algorithm itself any harder, because extended Tarski formulas can be converted to Tarski formulas in a preprocessing step, as described below.

A rational function can be simply multiplied by the common denominator, but special care must be taken to the sign. For example:

$$A/B \geq 0 \;\longmapsto\; (A \geq 0 \wedge B > 0) \vee (A \leq 0 \wedge B < 0).$$

The functions min and max can be removed as follows. For example:

$$f(\max(A, B)) > 0 \;\longmapsto\; (A \geq B \wedge f(A) > 0) \vee (B \geq A \wedge f(B) > 0).$$

Simplier formula can be obtained if f can be proved to be monotonic, e.g. if f is monotically increasing:

$$f(\max(A, B)) > 0 \;\longmapsto\; f(A) > 0 \vee f(B) > 0.$$

The expression $|A|$ can be simply replaced by $\max(A, -A)$.

Square roots and nth roots are handled as follows. The general solution is to introduce a new variable, e.g.:

$$f(\sqrt[n]{A}) > 0 \longmapsto \exists z : z^n = A \wedge z \geq 0 \wedge f(z) > 0$$

if n is even, otherwise the $z \geq 0$ part is omitted. The problem is that the CAD is very sensitive to the number of variables, its worst-case complexity is doubly exponential in it, so it is preferable to avoid adding new variables whenever possible. For this, we first try to rewrite the expression as follows: $A\sqrt[n]{B} + C\sqrt[n]{D} > 0$ (the relational symbol may differ), where A and C contains no roots. This can be converted to rootless formulas of A, B, C and D. If n is odd, it is simply equivalent to $A^n B + C^n D > 0$. For even n, it is more complicated, because it depends on the sign of A, B, C and D. For example:

$$x\sqrt{y} + (y - x) > 0 \longmapsto y \geq 0 \wedge ((x > 0 \wedge (y - x > 0 \vee x^2 y > (y - x)^2)) \vee (x \leq 0 \wedge y - x > 0 \wedge x^2 y < (y - x)^2)).$$

3.4. Formula conversion in the package

In the package, the CAD formula conversion is invoked by the same function as CAD itself, `cylindrical_algebraic_decomposition()`. If it gets a logical formula as an input, it converts it to an the equivalent CAD formula. For example:

```
sage:  var('x,y,z')
sage:  cylindrical_algebraic_decomposition(x^2 + y^2 <= 2)
(x == -sqrt(2) /\ y == -sqrt(-x^2 + 2)) \/ (x > -sqrt(2) /\
x < sqrt(2) /\ y >= -sqrt(-x^2 + 2) /\ y <= sqrt(-x^2 + 2)) \/
(x == sqrt(2) /\ y == -sqrt(-x^2 + 2))
```

We get a polynomial CAD formula using the `output="poly"` parameter:

```
sage:  var('x,y,z')
sage:  cylindrical_algebraic_decomposition(x^2 + y^2 <= 2, output="poly")
(x^2 == 2 /\ y^2 == -x^2 + 2) \/ (x^2 < 2 /\ y^2 <= -x^2 + 2)
```

Compound formulas can be created using the `logic_and()`, `logic_or()` and `logic_not()` functions:

```
sage:  cylindrical_algebraic_decomposition(logic_and(x^2 + y^2 < 1,
       x^3 == y^2, 2*x <= 1))
(x == 0 /\ y == -sqrt(x^3)) \/ (x > 0 /\ x <= (1/2) /\
(y == -sqrt(x^3) \/ y == sqrt(x^3)))
```

The package can handle extended Tarski formulas:

```
sage:  cylindrical_algebraic_decomposition(sqrt(x) + sqrt(y) >= 1)
(x >= 0 /\ x <= 1 /\ y >= x - 2*sqrt(x) + 1) \/ (x > 1 /\ y >= 0)
```

Quantified formulas can be formed by `logic_forall()`/`logic_exists()`:

```
sage:  cylindrical_algebraic_decomposition(logic_exists(y,
       logic_and(x^2 + y^2 == 3, x + y < 0)))
x >= -sqrt(3) /\ x < 1/2*sqrt(6)
sage:  cylindrical_algebraic_decomposition(logic_exists(x,
       logic_forall(y, x^2 - y^2 < 1)))
True
sage:  cylindrical_algebraic_decomposition(logic_exists(x,
       logic_forall(y, x^2 - y^2 < z)))
z > 0
```

When `cylindrical_algebraic_decomposition()` gets a formula, it calls `cad_formula_reduce()` from `cad.sage` to do the CAD formula conversion. It first calls `convert_formula_to_poly()` from `formula.sage` to convert the input from extended Tarski formula to Tarski formula as described in Section 3.3. Then it executes the algorithm in Section 3.1, using `cad_polys_to_cells()` for the decomposition. The CAD formula or the polynomial CAD formula is generated recursively by the functions `CADCell._to_rooty_formula()` and `CADCell._to_poly_formula()`, respectively. The latter is incomplete: only the projection-definable case is implemented (see Section 3.2), but it turns out that it suffices for most problems.

This package provides a way to express logical formulas in Sage, which has no built-in types for that. We define the following classes in `logic.sage`: `Logic` (base class), `LogicConstant` (true or false), `LogicRelation` (e.g. $x^2y > (x-1)^2$), `LogicChain` ($\wedge$ or $\vee$) and `LogicQuantified`. These classes should not be used directly, instead the following functions and constants should be used: `logic_true`, `logic_false`, `logic_and()`, `logic_or()`, `logic_not()`, `logic_implies()`, `logic_forall()` and `logic_exists()`. More details can be found in the documentation strings in `logic.sage`.

In addition to providing a CAD implementation, this package can also use QEPCAD as an external tool to do the CAD, as mentioned in the introduction. In this way, we can combine the power of QEPCAD with some features of this package, e.g. we can give an input formula with square roots, which QEPCAD is unable to handle on its own, but this package can convert it to a Tarski formula and pass it to QEPCAD, and then convert the result to a `Logic` object. In order to use QEPCAD instead of the built-in implementation, we need to pass the parameter `algorithm="qepcad"` to `cylindrical_algebraic_decomposition()`. This requires QEPCAD to be installed on the machine and be accessible on the command line via the `qepcad` command. For example:

```
sage:  cylindrical_algebraic_decomposition(logic_exists(z,
       x == sqrt(1 - y^2 - z^2)), algorithm="qepcad");
x >= 0 /\ x^2 + y^2 - 1 <= 0
```

Note that QEPCAD returns the formula in a different format than the built-in implementation. It produces neither a CAD formula nor a polynomial CAD formula, but instead by default it tries to return as simple formula as possible, using polynomials only.

4. Inequality proving

An interesting application of CAD is a proving procedure of inequalities involving recursive functions like factorials and sums, an algorithm invented by Gerhold and Kauers [9, 13]. A simple example to illustrate the algorithm is Bernoulli's inequality:

$$1 + nx \leq (1 + x)^n \quad (x \geq -1, n \in \mathbb{N}).$$

The inequality on its own lies outside of the scope of CAD, since it contains non-polynomial expressions (and does not even fit into our notion of extended Tarski formulas), because the exponent n is a variable. The idea is to use induction on n, as we would do by hand, but prove the steps by CAD. After checking the trivial $n = 0$ case, we need to prove the following:

$$n \geq 0 \wedge x \geq -1 \wedge 1 + nx \leq (1 + x)^n \Longrightarrow 1 + (n + 1)x \leq (1 + x)^{n+1}.$$

At first glance it seems that we did not make any progress, because we still have the non-polynomial expressions. But here comes the trick. We assign new variables to the non-polynomial expressions, so let $y := (1+x)^n$. Now the formula becomes:

$$n \geq 0 \wedge x \geq -1 \wedge 1 + nx \leq y \Longrightarrow 1 + (n + 1)x \leq (1 + x)y,$$

and now we can apply CAD, which returns *true*, thus proves the statement.

Note that if we had introduced the new variable right in the original inequality, we would have got $n \geq 0 \wedge x \geq -1 \wedge 1 + nx \leq y$, and although that is a Tarski formula, CAD is not able to prove that, because it has no way to know the relation between y and x. In the induction formula however, the meaning of y appeared when we expressed the shifted version of y as $(1+x)y$. This illustrates the requirement on the original formula: the recursive expressions must have a recurrence relation in a form that is acceptable by our CAD. These are called by Kauers *nested polynomially recurrent sequences* [12] (though we may allow non-polynomial expressions in the scope of the extended Tarski formulas, but as we have seen before, it is equivalent to a Tarski formula).

Another example of a nested polynomially recurrent sequence is in the following inequality (from [9, Section 4]):

$$n! \sum_{k=0}^{n} \frac{(-1)^k}{k!} > 0 \quad (n \geq 2).$$

We introduce new variables: $x := n!$, $y := \sum_{k=0}^{n} (-1)^k/k!$ along with the shifted versions:

$$x' = (n+1)! = (n+1)x,$$
$$y' = \sum_{k=0}^{n+1} \frac{(-1)^k}{k!} = y + \frac{(-1)^{n+1}}{(n+1)!} = y + \frac{z}{(n+1)x}$$

with the new variable $z = (-1)^{n+1}$, noting that $z' = -z$.

The initial case ($n = 2$) is again trivial (though in general, we may need to use CAD for the initial case as well). The induction step however is not as simple as in the previous example. If we try to prove the following by CAD:

$$n \geq 2 \wedge xy > 0 \Longrightarrow (n+1)x \left(y + \frac{z}{(n+1)x} \right) > 0,$$

we get back a similar formula instead of the desired *true*. This means that the formula still fails to hold for some $x, y, z \in \mathbb{R}$, ignoring their meaning. It is not so surprising, since we did not use the recurrence relation of z, for instance. To use that, we need to go one step further, i.e. basically do a two-step induction. We do that in an optimized way, by using the output of the above CAD, say ϕ. First we check ϕ for $n = 0$ (trivial), then the following:

$$n \geq 2 \wedge xy > 0 \wedge \phi \Longrightarrow \phi',$$

where ϕ' is the shifted version of ϕ, i.e. all variables are replaced by their shifted version. Executing CAD on this input, it finally gives *true*, proving the original inequality.

The algorithm in general is the following [9, Section 3.]. Assume that the input is a formula involving nested polynomially recurrent sequences in n, and it is to be proved for integers $n \geq n_0$.

1. Identify all recursive expressions in the input, introduce new variables for each one, and let ϕ be the resulting extended Tarski formula. Generate also the shifted versions of the variables.
2. $\psi := (n \geq n_0)$
3. While $\phi \neq \mathit{true}$ do

(a) Apply CAD to ϕ substituted at $n = n_0$. If the result is not *true*, return false (the statement is refuted).
(b) $\psi := \psi \wedge \phi$
(c) $\phi = \mathrm{CAD}(\psi \Longrightarrow \phi')$

4. Return true (the statement is proved).

Note that there is no guarantee that the algorithm terminates, but if it does so, the result is always correct.

4.1. The package

This algorithm is implemented by `try_to_prove()` in `ineq.sage`. E.g. the previous example can be proved like this:

```
try_to_prove(factorial(n) * sum((-1)^k/factorial(k), k, 0, n) > 0, n, 2);
```

The last two parameters tell that the induction is in `n` starting from 2.

It is also possible to define new recursive functions. The following example defines the Fibonacci polynomial:

```
rec = Recursifier(n);
function("fibpol", nargs=2);
rec.add_recurrence([
   fibpol(n+1, x) == x*fibpol(n, x) + fibpol(n-1, x),
   fibpol(0, x) == 0, fibpol(1, x) == 1
]);
```

We can use this to prove an inequality from [9]:

```
try_to_prove(fibpol(n, x)^2 <= (x^2 + 1)^2*(x^2 + 2)^(n-3), n, 3, rec=rec);
```

In `ineq.sage`, there are lots of other tested examples. Many of them are from [9], e.g.:

$$\frac{1}{4n} < \frac{\binom{2n}{n}^2}{16^n} < \frac{1}{3n+1} \quad (n \geq 2)$$

$$\sqrt{n - \frac{3}{4}} \leq R_n - \frac{1}{2} \leq \sqrt{n + \frac{1}{4}}$$
$$R_1 := 1,\ R_{n+1} := 1 + n/R_n \quad (n \geq 1)$$

Inequalities like $A \leq B \leq C$ must be written as $A \leq B \wedge B \leq C$. Binomial coefficients must be expanded to factorials.

We can also prove a bit more general version of Bernoulli's inequality:

$$1 + nx \leq (1 + x)^n \quad (x \geq -2, n \in \mathbb{N}).$$

Interestingly, the $x \geq -1$ case needed only one step of the algorithm, but the $x \geq -2$ case requires two steps.

Another proven inequality is about the Fibonacci numbers from [6]:

$$2\left(F_n^4 + F_{n+1}^4 + F_{n+2}^4\right)\left(\frac{1}{F_n^2} + \frac{1}{F_{n+1}^2} + \frac{1}{F_{n+2}^2}\right)^2 > 100 \quad (n \geq 1)$$

It is interesting that the algorithm does not terminate on this inequality on its own, but it does so when we prepend the trivial condition $F_n > 0 \wedge F_{n+1} > > 0 \Longrightarrow \ldots$.

4.2. Recursifier

An important part of the implementation is the `Recursifier` class in `recur.sage`. It performs Step 1 of the algorithm, i.e. it takes a formula with recursive functions as an input, substitutes variables to the non-Tarski parts, and stores their recurrence relations. For example (cf. beginning of Section 4):

```
sage:  var('n,x');
sage:  rec = Recursifier(n);
sage:  ineq = rec.recursify(1 + n*x <= (x + 1)^n); ineq
a*b + 1 <= c
sage:  for v in rec.variables():
....:      print v, "->", rec.shift(v);
a -> a + 1
b -> b
c -> (b + 1)*c
sage:  rec.shift(ineq);
(a + 1)*b + 1 <= (b + 1)*c
```

`Recursifier` walks through the expression tree of the input to search for non-Tarski subexpressions, and tries to find a recurrence relation for them. It uses a set of known recursive functions, including those given by the user in `rec.add_recurrence()` (see the example of `fibpol` above). Built-in recursive functions like $n!$ and a^n, but also $\sum_{k=a}^{n} f(k)$ and $\prod_{k=a}^{n} f(k)$, are defined similarly in `init_known_recurrences()`. Parameters of the recursive functions can be, apart from simply n, also $an + b$ with $a \geq 0$ and b integer constants. Some examples that `Recursifier` cannot handle and returns error for: n^n, $(n^2)!$, 2^{2^n}, $\sum_{k=0}^{n} nk$, $\sum_{k=-n}^{0} k$.

5. Final words

The package introduced in this paper provides a new implementation of cylindrical algebraic decomposition, and also of the algorithm of Gerhold and Kauers. As we have shown, it is capable of solving several problems involving inequalities in a user-friendly manner.

5.1. Comparison with QEPCAD

Now we compare the CAD implementation of this package with QEPCAD [3] and its already existing Sage interface [11].

First, we compare them by running time on some examples (they both use McCallum's projection, and QEPCAD is invoked through our package):

Input	**Package**	**QEPCAD**
$x^2 + y^2 < 1 \wedge x^3 = y^2$	0.27 s	0.46 s
$(x-1)^2 + (y-1)^2 + (z-1)^2 \leq 1 \wedge xyz = 1$	3.89 s	0.51 s
$\forall x : \exists y : x^2 + xy + b > 0 \wedge ay^2 + b + x \leq 0$	6.56 s	0.48 s
$\forall x : x^4 + ax^3 + bx^2 + cx + d > 0$	1.97 s	0.90 s

QEPCAD is clearly faster, partly because it is written in C, and also because it has more optimized algorithms. But there is an overhead of invoking QEPCAD as an external application, which makes it slower for small examples.

An other advantage of QEPCAD is that it has a great variety of options to customize the algorithm (e.g. six different projection operators are available).

Our package has other advantages, for example it has cleaner code, and it is written in a high-level language, Sage (while QEPCAD is written in C). This makes it easier to modify, extend and optimize later.

QEPCAD is an interactive command-line application, where the user enters the input and gets back the output as a string in QEPCAD's own syntax. The existing Sage interface wraps it into a Sage function, but it still returns the output as a string. Our implementation however provides a flexible and convertible Sage type for the input and the output, so it can be used as a CAD subroutine in larger calculations, such as the algorithm of Gerhold and Kauers.

The input of QEPCAD (and its Sage interface) must be a polynomial formula (more precisely, a Tarski formula, see Definition 3.3), while our package has a much wider domain, and accepts e.g. fractions and roots in the input (see Section 3.3). This package returns the output in a very structured form (see Definiton 3.4 and 3.9), which allows many properties of the set to be obtained mechanically (see Example 3.1). QEPCAD has a different approach, it tries to return as short formula as possible.

5.2. Future improvements

In the future, we consider the following improvements and optimizations for the presented package:

- Implement better projection operators like Brown's improvement (see Section 2.3).

- Complete the implementation of generating polynomial CAD formula, i.e. Brown's algorithm to introduce new polynomials (see Section 3.2).
- Examine whether the CAD implementation can be optimized by using precomputed number fields instead of Sage's general algebraic number type (`AA`).
- Improve the interface for QEPCAD, e.g. by giving more access to its special functionality, and handling error messages properly.

References

[1] **Brown, C. W.,** Improved projection for cylindrical algebraic decomposition, *Journal of Symbolic Computation*, **32** (2001), 447–465.

[2] **Brown, C. W.,** *Solution Formula Construction for Truth Invariant CADs*, PhD thesis, 1999.

[3] **Brown, C. W.,** QEPCAD B – a program for computing with semi-algebraic sets using CADs, *ACM Sigsam Bulletin*, **37** (2003), 97–108.

[4] CylindricalDecomposition – Wolfram Language Documentation. `https://reference.wolfram.com`

[5] **Collins, G. E.,** Quantifier elimination for real closed fields by cylindrical algebraic decomposition, In:*Brakhage H.* (*eds*) *Automata Theory and Formal Languages. Lecture Notes in Computer Science* **33** (1975), Springer, Berlin, Heidelberg, 134–183.

[6] **Cooper, C. T., F. F. Ngwane and W. Lai,** An improved Fibonacci inequality, *Journal of South Carolina Academy of Science* **11** (2) (2013), 8–9.

[7] **Dolzmann, A. and T. Sturm,** Redlog: Computer algebra meets computer logic, *ACM Sigsam Bulletin*, **31** (1997), 2–9.

[8] **Geddes, K. O., S. R. Czapor and G. Labahn,** *Algorithms for Computer Algebra*, Kluwer Academic Publishers, 1992.

[9] **Gerhold, S. and M. Kauers,** A procedure for proving special function inequalities involving a discrete parameter, *Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation* (2005), 156–162.

[10] **Hong, H.,** An improvement of the projection operator in cylindrical algebraic decomposition, *Proceedings of the International Symposium on Symbolic and Algebraic Computation* (1990), 261–264.

[11] Interface to QEPCAD – Sage Reference Manual v9.0. `http://doc.sagemath.org`

[12] **Kauers, M.,** An algorithm for deciding zero-equivalence of nested polynomially recurrent sequences, *ACM Transactions on Algorithms* 3, 2 (2007).

[13] **Kauers, M.,** Computer algebra for special function inequalities, In: *Tapas in Experimental Mathematics, Contemporary Mathematics 457*, (2008), 215–235.

[14] **Kauers, M.,** How to use cylindrical algebraic decomposition, *Séminaire Lotharingien de Combinatoire*, **65** (2011).

[15] **Kauers, M.,** SumCracker: A package for manipulating symbolic sums and related objects, *Journal of Symbolic Computation*, **41** (2006), 1039–1057.

[16] **McCallum, S.,** An improved projection operation for cylindrical algebraic decomposition, *EUROCAL '85*, (1985), 277–278.

[17] **McCallum, S.,** An improved projection operation for cylindrical algebraic decomposition of three-dimensional space, *Journal of Symbolic Computation*, **5** (1988), 141–161.

[18] **Uray, M. J.,** Sage package
`http://ac.inf.elte.hu/Vol_051_2020`

M. J. Uray
Department of Computer Algebra
Faculty of Informatics
Eötvös Loránd University
H-1117 Budapest
Pázmány P. sétány 1/C
Hungary
`uray.janos@inf.elte.hu`