LibPoly: A Library for Reasoning about Polynomials*

Dejan Jovanović and Bruno Dutertre

SRI International

Abstract

LibPoly is a C library for computing with polynomials. It provides data structures to represent multivariate polynomials, and algorithms ranging from simple arithmetic and GCD computation, to root isolation and computation with algebraic numbers. The goal of the library is to be simple and extensible, and it is targeting tools that reason in nonlinear arithmetic. LibPoly is already successfully used in the Yices 2 nonlinear arithmetic solver. The library is freely available under a permissive open-source license. We present the basic functionality of LibPoly through the Python interface, and we describe a simple implementation of a classical cylindrical algebraic decomposition algorithm.

1 Introduction

Nonlinear arithmetic is an expressive domain with many applications in theorem proving [14], verification of hybrid systems [27], program verification and synthesis [11, 25, 10], termination analysis [24], and even cryptography [2] and verification of neural networks [20, 23]. Symbolic reasoning in nonlinear arithmetic has traditionally been the forte of computer algebra systems such as Mathematica and Maple. With the advent of satisfiability modulo theories (SMT) solvers [12] there has been a renewed effort in developing nonlinear reasoning tools that focus on satisfiability and scalability: applications usually require solving many problems very fast, where each individual problem might be large but not intrinsically hard (for example, mostly linear). One example of a method that has been successful in the SMT context is NLSAT [22].

Computer algebra systems often support more general algebraic structures, but they usually include, or rely on, a specialized library for polynomial computations. A notable example of such a is the saclib [9, 8] library that backs the qepcad [5] cylindrical algebraic decomposition (CAD) system. Other examples include CoCoA [1], Singular [17], pari/gp [26], ntl [28], flint [19], GiNaC [4], and redlog [15]. These libraries, although very powerful, provide support for certain polynomial operations (e.g., univariate factorization or Gröbner basis computation), but they lack some of the features to support an efficient implementation of a CAD-based SMT solver in the style of NLSAT. Some of the missing features are related to the software development aspects, such as, permissive open-source license and a clean C interface with good documentation that is easy to use.[1] But, more importantly, implementing an NLSAT-style solver requires full support for CAD operations, while also being flexible enough to handle polynomials with thousands of variables and a continuously changing variable order (similar to the order of variables in a SAT solver).

We present LibPoly, a C library that can support the needs of modern SMT solvers in computing with polynomials. In addition to basic operations in the ring of polynomials, LibPoly includes all the necessary ingredients for implementing CAD-based SMT reasoning engines, such as multivariate root isolation and projection of polynomials using sub-resultant sequences.

*The research presented in this paper has been supported by NASA Cooperative Agreements NNX14AI05A and by NSF grant 1528153.

[1] For example, redlog and Singular don't have a C interface and require running a separate kernel and communication channels like pipes and sockets.

The interface to the library is designed to support both the traditional, purely symbolic algorithms, and the more recent model-driven methods. Therefore, LibPoly supports operations not usually considered in traditional symbolic computer algebra systems, such as a variable order that can change freely between various operations.

To present the functionality of LibPoly, we go through a simple implementation of a cylindrical algebraic decomposition (CAD) algorithm [7] using the Python API to LibPoly.[2] We then discuss current status and future plans. We present the basic functionalities that are needed for implementing a CAD-based reasoning system, but the library supports more advanced features, such as computing with polynomial over modular arithmetic,

2 Usage and Examples

We will go through a Python implementation of the CAD construction algorithm by relying on LibPoly's Python bindings. We first demonstrate the basics of polynomial construction and manipulation. Then we show how to isolate and manipulate roots of univariate polynomials and construct the sign table of a univariate polynomial. Finally, we extend the sign-table construction to the multivariate case, by relying on CAD projections, resulting in a simple implementation of CAD construction.[3]

2.1 Basic Operations

Variables. A variable in LibPoly is an object representing real-valued variables. Internally, it has an associated printable name and a unique numerical ID. The following example imports the LibPoly library in Python and creates three variables x, y, and z.

```
1 import polypy                                      # Import the library
2 x = polypy.Variable('x')                           # Variable x
3 [y, z] = [polypy.Variable(s) for s in ['y', 'z']] # Variables y and z
```

Variable Order. An important feature of LibPoly is a flexible order over the variables. By default, variables are ordered by numerical ID's (i.e., in order of creation), but this order can be changed dynamically. The order is updated by manipulating the *order list* that changes the variable order according to the following semantics:

- variables in the order list are smaller than the variables not in the order list;
- variables in the order list are ordered according to the list order; and
- variables not in the order list are ordered according to their numerical ID's.

The following example illustrates the order manipulation.

```
1 order = polypy.variable_order # order = []     , x < y < z
2 order.push(z)                 # order = [z]    , z < x < y
3 order.push(y)                 # order = [z, y], z < y < x
4 order.pop()                   # order = [z]    , z < x < y
5 order.push(x)                 # order = [z, x], z < x < y
```

[2] The presentation focuses on the Python API for clarity, but the library is implemented in C and provides a C API with equivalent functionality.

[3] Complete examples are available at https://github.com/SRI-CSL/libpoly/tree/master/examples.

Polynomials. Polynomials in LibPoly always have integer coefficients. They can be created and combined using the standard Python arithmetic operators over variables and integer constants. The following example creates two polynomials $f = (x^2 - 2x + 1)(x^2 - 2)$ and $g = z(x^2 - y^2)$.

```
f = (x**2 - 2*x + 1)*(x**2 - 2) # Univariate polynomial
g = z*(x**2 - y**2)             # Multivariate polynomial
```

The internal representation of polynomials is recursive: a polynomial is a list of coefficients in its topmost variable, where each coefficient is a polynomial in the remaining variables. This representation is updated dynamically to respect the variable order. For example, if the variable order is $x < y < z$, then g is a polynomial in $\mathbb{Z}[x, y, z]$, with z as top variable. Therefore, g is represented as $g = (-y^2 + x^2) \cdot z$. On the other hand, if the variable order is $z < y < x$, then x is the top variable, and $g = z \cdot x^2 + (-z) \cdot y^2$.

```
order.set([x, y, z]) # z is the top variable, i.e. g in Z[x,y,z]
print g              # Out: (-1*y**2 + (1*x**2))*z
order.set([z, y])    # x is the top variable, i.e. g in Z[z,y,x]
print g              # Out: (1*z)*x**2 + ((-1*z)*y**2)
```

Several operations give access to this recursive structure: The `var()` method returns the top variable of the polynomial in the current order, the `degree()` method returns the degree of the polynomial in its top variables, and the `coefficients()` method returns the list of polynomial coefficients with respect to its top variable (in decreasing degree order). The `derivative()` method computes the derivative of the polynomial with respect to its top variable. The `factor_square_free()` method returns a factorization of the polynomial where no factor is a square and the factors are pairwise co-prime.[4]

```
g.var()                 # Out: Variable('x')
g.degree()              # Out: 2
g.coefficients()        # Out: [(-1*z)*y**2, 0, 1*z]
g.derivative()          # Out: (2*z)*x
f.factor_square_free() # Out: [(1*x**2 - 2, 1), (1*x - 1, 2)]
g.factor_square_free() # Out: [(1*z, 1), (1*x**2 + (-1*y**2), 1)]
```

In the example above, f is factored (over $\mathbb{Z}$) into irreducible polynomials $(x^2 - 2)(x - 1)^2$. The polynomial g is factored into $g = z.(x^2 - y^2)$. This is not a full factorization as $(x^2 - y^2)$ could be further decomposed but the factorization is square free.

2.2 Constructing a Sign Table

Root Isolation. When analyzing the behavior of a polynomial $f \in \mathbb{Z}[\vec{x}, y]$, the basic operation is finding the roots of f. A *root* of f, given an assignment of variables $\vec{x}$ to values $\vec{a}$, is a value b such that $f(\vec{a}, b) = 0$. The ordered list of roots of f can be obtained by calling the `roots_isolate()` method on the polynomial f.

```
m = polypy.Assignment()
r = f.roots_isolate(m)
print r[0] # Out: <1*x**2 + (-2), (-3/2, -5/4)>
print r[1] # Out: 1
print r[2] # Out: <1*x**2 + (-2), (5/4, 3/2)>
```

In the example above, we first create an empty `Assignment` object, and then obtain the roots of $f = (x^2 - 2)(x - 1) \in \mathbb{Z}[x]$. The result is the list $\left[-\sqrt{2}, 1, \sqrt{2}\right]$. LibPoly displays $-\sqrt{2}$ and $\sqrt{2}$ as algebraic numbers as explained in the following paragraph.

[4] This is not a full factorization, but it is sufficient for most purposes where factoring is useful. Complete factorization, although potentially useful, is not necessary for CAD and is currently not available in LibPoly.

Working with Values. The real values (including roots) in LibPoly are represented as *value objects* and are either integers, dyadic rationals[5], or algebraic numbers such as $\sqrt{2}$. Algebraic numbers are represented symbolically as pairs $\langle f, I\rangle$, where f is a univariate polynomial f, and I is an real interval in which f has a unique root. The algebraic number $\langle f, I\rangle$ is the root of f that occurs in interval I. In the previous example, the algebraic number $\sqrt{2}$ is represented as the pair $\langle x^2 - 2, (\frac{5}{4}, \frac{3}{2})\rangle$.

```
r[0].to_double()              # Out: -1.41421356237
r[0] < r[1]                   # Out: True
r[0].get_value_between(r[2]) # Out: 0
m.set_value(x, 0)             # Set x -> 0
f.sgn(m)                      # Out: -1
m.set_value(x, r[0])          # Set x -> -sqrt(2)
f.sgn(m)                      # Out: 0
```

The snippet above illustrates the basic operations on real values. Real values can be converted to floating point with the `to_double()` method, and they can be compared to each other with the usual comparison operators in Python. Given two values v_1 and v_2, method `get_value_between()` returns the a value in the interval (v_1, v_2). The method picks the "simplest" possible value, namely a dyadic rational in the interval with the smallest value of m. In particular, the method will return an integer if one exists in (v_1, v_2). Method `set_value()` assigns a value to a variable, while `unset_value()` clears the value. Method `sgn()` computes the sign of a polynomial at a point given by a full assignment.

Constructing a Sign Table. We now have all the ingredients to compute sign tables of univariate polynomials. For a set of univariate polynomials $\mathcal{F}$ a *sign table* of $\mathcal{F}$ is a decomposition of $\mathbb{R}$ into intervals $I_1, \ldots, I_n$, such that, in each interval I_k, the polynomials in $\mathcal{F}$ are *sign-invariant* on I_k (i.e., they don't change sign). To construct a sign table of $\mathcal{F}$, we first isolate the roots of all $f \in \mathcal{F}$, then we sort the computed roots by increasing value, and, finally, we evaluate the sign of the polynomials in the *intervals delimited by the roots*. In each such interval, I_k, the polynomials of $\mathcal{F}$ do not change sign, so we can evaluate the sign of each $f \in \mathcal{F}$ on I_k by evaluating the sign of f at a single *evaluation point* of I_k.

A Python function that constructs a sign table using LibPoly is presented in Figure 1. Applying this function to polynomials $f = x^2 - 2$ and $g = x^2 - 3$ will result in a sign table that resembles the following.

	$(-\infty, -\sqrt{3})$	$-\sqrt{3}$	$(-\sqrt{3}, -\sqrt{2})$	$-\sqrt{2}$	$(-\sqrt{2}, \sqrt{2})$	$\sqrt{2}$	$(\sqrt{2}, \sqrt{3})$	$\sqrt{3}$	$(\sqrt{3}, +\infty)$
f	1	1	1	0	−1	0	1	1	1
g	1	0	−1	−1	−1	−1	−1	0	1

In the univariate case, building a sign table is a complete method for determining if a set of polynomial constraints has a solution. For example, we can solve the constraint $(f > 0) \wedge (g < 0)$ by examining the sign table. It is clear that the solutions are in the set $(-\sqrt{3}, -\sqrt{2}) \cup (\sqrt{2}, \sqrt{3})$. Moreover, in order to perform such an analysis, we only need to keep track of one evaluation point per interval. Figure 2 plots the two polynomials f and g of our example, and shows the evaluation points (red dots).

2.3 Cylindrical Algebraic Decomposition

A sign table that we computed in the previous section decomposes $\mathbb{R}$ into finitely many regions. Each region is either an open interval or a point, and in each region, the polynomials of $\mathcal{F}$ are

[5] Rationals of the form $\frac{n}{2^m}$.

```
1  def sign_table(x, polys, m):
2    # Get the roots and print the header
3    roots = set()  # Set of roots
4    output("poly/int")
5    for f in polys:
6      output(f)
7      f_roots = f.roots_isolate(m)
8      roots.update(f_roots)
9    stdout.write("\n")
10   # Sort the roots and add infinities
11   roots = [polypy.INFINITY_NEG] + sorted(roots) + [polypy.INFINITY_POS]
12   # Print intervals and signs in the intervals
13   root_i, root_j = itertools.tee(roots)
14   next(root_j)
15   for r1, r2 in itertools.izip(root_i, root_j):
16     output((r1.to_double(), r2.to_double()))
17     # The interval (r1, r2)
18     v = r1.get_value_between(r2);
19     m.set_value(x, v)
20     for f in polys: output(f.sgn(m))
21     stdout.write("\n")
22     # The interval [r2]
23     if r2 != polypy.INFINITY_POS:
24       output(r2.to_double())
25       m.set_value(x, r2)
26       for f in polys: output(f.sgn(m))
27       stdout.write("\n")
28   m.unset_value(x)
```

Figure 1: A function for printing a sign table of a set of polynomials. Code fragments where relevant LibPoly functionality is used are emphasized. For example, sorting in Line 11 is done using the Python built-in `sorted()` function that compares values using value comparison functionality from LibPoly.

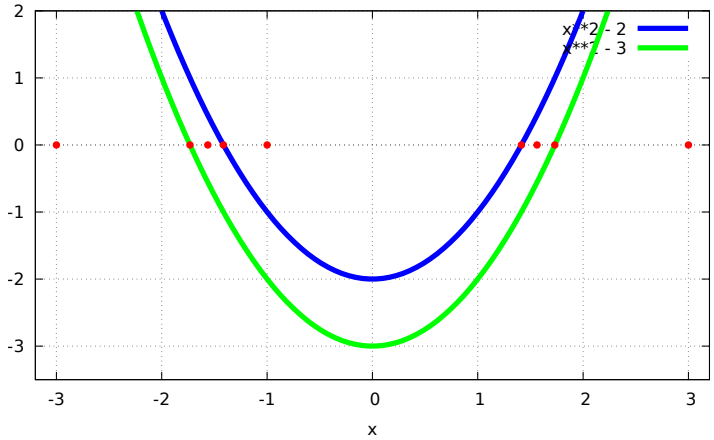

Figure 2: A plot of polynomials $f = x^2 - 2$ and $f = x^2 - 3$, along with the evaluation points during sign table construction.

```
# Lift the first variable, update the assignment, and lift recursively
def lift_first_var(poly_map, vars, m):
  if len(vars) == 0:
    # Assignment complete, print the evaluation point
    print(m)
    return
  # The first unassigned variable
  x = vars[0]
  # Get the roots of polynomials where x is the top variable
  roots = set()
  for f in poly_map[x]:
    f_roots = f.roots_isolate(m)
    roots.update(f_roots)
  # Sort the roots and add infinities
  roots = [polypy.INFINITY_NEG] + sorted(roots) + [polypy.INFINITY_POS]
  # Make a 'sign table' for x, and lift remaining variables recursively
  r_i, r_j = itertools.tee(roots)
  next(r_j)
  for r1, r2 in itertools.izip(r_i, r_j):
    # The interval (r1, r2), also called 'sector'
    v = r1.get_value_between(r2);
    m.set_value(x, v)
    # Lift recursively over the remaining variables
    lift_first_var(poly_map, vars[1:], m)
    # The interval [r2], also called 'section'
    if r2 != polypy.INFINITY_POS:
      m.set_value(x, r2)
      lift_first_var(poly_map, vars[1:], m)
  m.unset_value(x)

# Do the lifting
def lift(poly_map, vars):
  m = polypy.Assignment()
  lift_first_var(poly_map, vars, m)
```

Figure 3: CAD lifting procedure: generalized sign-tabling for n variables. Differences compared to the `sign_table()` function are underlined.

sign-invariant. The sign table completely characterizes the behavior of $\mathcal{F}$ on $\mathbb{R}$. A natural question arises: *can we extend the concept of sign table (with evaluation points) to the multivariate case, so that we can solve polynomials constraints with several variables*? The answer to this question is positive, and the generalization of sign tables to $\mathbb{R}^n$ is called cylindrical algebraic decomposition (CAD) [7].

CAD Lifting. A naïve attempt at constructing a CAD (sign table) would be to extend the sign-tabling as follows. Let $\mathcal{F} \subseteq \mathbb{Z}[x_1, \ldots, x_n]$ be a set of polynomials. We partition $\mathcal{F}$ according to the top variable of each polynomial:

$$\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2 \cup \cdots \cup \mathcal{F}_n \ .$$

Every polynomial $f \in \mathcal{F}_k$ contains only variables $x_1, \ldots, x_k$, and x_k is the top variable of f. With this setup, we can directly extend the `sign_table()` function of Figure 1 into a recursive procedure that builds the CAD of $\mathcal{F}$ one variable at a time. We do so by building a sign table T_1 for polynomials in $\mathcal{F}_1$. Then, for each evaluation point α in T_1, we build a sign table $T_{\alpha,2}$ for polynomials in $\mathcal{F}_2$, and so on. In CAD terminology this procedure is called *lifting*.

The function `lift()` from Figure 3 takes as input a map `poly_map` that maps x_k to $\mathcal{F}_k$ as above, and a list `vars` of variables $[x_1, \ldots, x_n]$. The function outputs the computed evaluation points. It shouldn't be too hard to see that this lifting procedure on its own does not produce

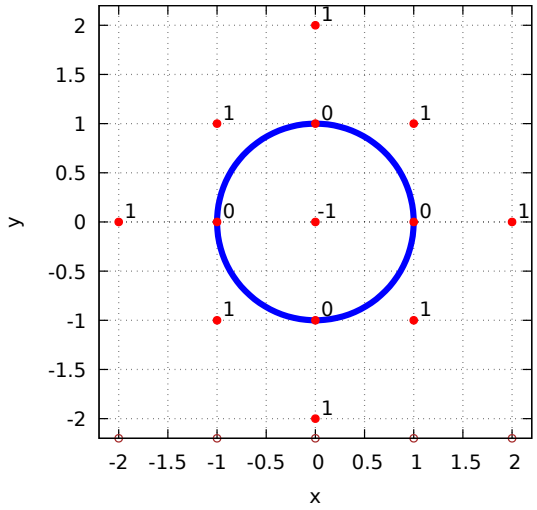

Figure 4: A plot of the roots of the polynomial $f = x^2 + y^2 - 1$ in blue, and the evaluation points (with f sign annotation) from the correct CAD of f in red. The evaluation points from the sign table of the projection polynomial $x^2 - 1$ are shown in brown at the bottom.

a satisfactory result. As an example, let's apply it to the polynomial $f = x_1^2 + x_2^2 - 1$, i.e. let $\mathcal{F} = \{f\}$ which partitions into $\mathcal{F}_1 = \emptyset$ and $\mathcal{F}_2 = \{f\}$.

lift x_1: Since $\mathcal{F}_1$ is empty, the set of roots for x_1 is also empty, and we can choose any value in $(-\infty, +\infty)$ as evaluation point for x_1. For example, let $x_1 \mapsto 2$ be the evaluation point for x_1.

lift x_2: $\mathcal{F}_2 = \{f\}$, and we search for the roots of the polynomial f when $x_1 \mapsto 2$. This amounts for finding the roots of $f(2, x_2) = x_2^2 + 3$. Since this polynomial does not have roots in x_2, we can, as before, pick any evaluation point. We choose $x_2 \mapsto 0$.

In the above example, the procedures computes a single evaluation point, namely, $(x_1, x_2) = (2, 0)$. At this point, f has positive sign/ This single evaluation point clearly does not capture the behavior of f over $\mathbb{R}^2$, since f is not uniformly positive.

CAD Projection. The above example shows that building sign tables variable by variable is not in itself sufficient to construct a proper sign-invariant decomposition for a set of polynomials $\mathcal{F}$. The problem lies in the fact that the lower-level polynomial sets $\mathcal{F}_k$ do not capture enough information about the behavior of the whole set $\mathcal{F}$, and the sign-tabling can miss important evaluation points. We can fix this, while keeping the same overall procedure, by adding carefully crafted polynomials to the sets $\mathcal{F}_k$. These additional polynomials will "guide" the construction of sign tables to cover all possible behaviors of the original set $\mathcal{F}$. In the seminal paper on CAD construction [7], George Collins showed how to do this completion, by projecting polynomials from higher levels into the lower levels.

Definition 2.1 (Collins Projection)**.** *Given a set of polynomials $\mathcal{F} \subset \mathbb{Z}[\vec{y}, x]$, the Collins*

projector operator $\mathsf{P}_c(\mathcal{F}, x)$ *is defined as*

$$\bigcup_{f\in\mathcal{F}} \mathsf{coeff}(f,x) \cup \bigcup_{\substack{f\in\mathcal{F}\\ g\in\mathsf{R}^*(f,x)}} \mathsf{psc}(g,g'_x,x) \cup \bigcup_{\substack{i<j\\ g_i\in\mathsf{R}^*(f_i,x)\\ g_j\in\mathsf{R}^*(f_j,x)}} \mathsf{psc}(g_i,g_j,x) \ .$$

The projection operator above, when applied to $\mathcal{F}$, introduces enough additional polynomials that capture the necessary missing behavior. The individual operations in P_c are advanced polynomial operations (for more details see, e.g., [3, 6]), that are all available in LibPoly. The set of coefficients $\mathsf{coeff}(f,x)$ can be obtained with the `coefficients()` method, while the derivative can be obtained with the `derivative()` method. A reductum[6] $\mathsf{R}(f,x)$ of a polynomial can be obtained using the `reductum()` method, and the principal subresultant coefficients $\mathsf{psc}(f,g)$ of two polynomials can be obtained with the `psc()` method.

Before defining the projection function, we first define two utility function. First, we define a function that, given a polynomial f, returns all non-constant reductums of f in variable x.

```
# Returns reductum closure of f, excluding constants
def get_reductums(f, x):
  R = []
  while f.var() == x: R.append(f); f = f.reductum()
  return R
```

Additionally, instead of adding polynomials directly to the sets $\mathcal{F}_k$, we will add them trough the following proxy function that performs factorization first. The square-free factorization simplifies the polynomials and ensures that some unnecessary complexity is removed.

```
# Add polynomials to projection map
def add_polynomial(poly_map, f):
  # Factor the polynomial f
  for (f_factor, multiplicity) in f.factor_square_free():
    # Add non-constant polynomials to poly_map
    if (f_factor.degree() > 0):
      x = f_factor.var()
      if (x not in poly_map): poly_map[x] = set()
      poly_map[x].add(f_factor)

# Add a collection of polynomials to projection map
def add_polynomials(poly_map, polys):
  for f in polys: add_polynomial(poly_map, f)
```

We are now ready to define our overall projection function `project()`. This function projects the sets $\mathcal{F}_n, \ldots, \mathcal{F}_1$ in sequence by applying the projection operator P_c to each of them.

```
# Project: go down the variable stack and project
def project(poly_map, vars):
  for x in reversed(vars):
    # Project variable x
    x_polys = poly_map[x]
    # [1] coeff(f) for f in poly[x]
    for f in x_polys:
      add_polynomials(poly_map, f.coefficients())
    # [2] psc(g, g') for f in poly[x], g in R(f, x)
    for f in x_polys:
      for g in get_reductums(f, x):
        g_d = f.derivative()
        if (g_d.var() == x):
          add_polynomials(poly_map, g.psc(g_d))
    # [3] psc(g1, g2) for f1, f2 in poly[x], g1 in R(f1, x), g2 in R(f2, x)
    for (f1, f2) in itertools.combinations(x_polys, 2):
      f1_R = get_reductums(f1, x)
      f2_R = get_reductums(f2, x)
```

[6] Reductum of a polynomial f is f without its highest-degree term.

```
19        for (g1, g2) in itertools.product(f1_R, f2_R):
20          add_polynomial(poly_map, g1.psc(g2))
```

Applying the `project()` function to the polynomial set $\mathcal{F} = \{x_1^2 + x_2^2 - 1\}$, from the failed example, results in $\mathcal{F}$ being enlarged to $\mathcal{F} = \{x_1^2 - 1, x_1^2 + x_2^2 - 1\}$. The newly added polynomial adds two critical roots to the sign table of x_1, which allows the lifting to succeed in CAD construction.

CAD Construction. All the tools are now in place for a full CAD construction. The main CAD construction routine `cad()` takes a list of polynomials and outputs the evaluation points corresponding to the CAD of the polynomials. The CAD is constructed by first projecting the input polynomials, and then constructing the CAD evaluation points over the projects set using the lifting procedure.

```
1  # Run the CAD construction
2  def cad(polys, vars):
3    polypy.variable_order.set(vars)
4    poly_map = {}
5    add_polynomials(poly_map, polys)
6    project(poly_map, vars)
7    lift(poly_map, vars)
```

Applying the procedure above on our example set $\mathcal{F} = \{x^2 + y^2 - 1\}$ results in the evaluation points depicted in Figure 4.

3 Current Status and Plans for Further Development

LibPoly is freely available on GitHub[7] under the LGPL license, with more information available at the LibPoly web page.[8] The implementation is about 15KLOC of C code, with the only external dependency being the widely used GMP library [18]. The library is stable and is currently used successfully in the yices2 [16] SMT solver to back the nonlinear reasoning capabilities through the native C API (see e.g. [21]).

Although the library is efficient[9], there are many parts of the library that could benefit from algorithmic improvements or new features. For example, more efficient algorithms for polynomial factorization and GCD computation, and more efficient computation with algebraic numbers (and support of infinitesimal numbers [13]). As the library is open source, we invite contributions and suggestions for future development at the LibPoly GitHub page.

References

[1] J. Abbott and A. M. Bigatti. CoCoALib: a C++ library for computations in commutative algebra... and beyond. In *International Congress on Mathematical Software*, pages 73–76. Springer, 2010.

[2] G. Barthe, E. Fagerholm, D. Fiore, J. Mitchell, A. Scedrov, and B. Schmidt. Automated analysis of cryptographic assumptions in generic group models. In *International Cryptology Conference*, pages 95–112. Springer, 2014.

[3] S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in real algebraic geometry*, volume 20033. Springer, 1996.

[7] https://github.com/SRI-CSL/libpoly

[8] http://sri-csl.github.io/libpoly/

[9] yices2 has won the nonlinear divisions (QF_NRA, QF_NIA) of the 2016 SMT competition.

[4] C. Bauer, A. Frink, and R. Kreckel. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *Journal of Symbolic Computation*, 33(1):1–12, 2002.

[5] C. W. Brown. QEPCAD B: a program for computing with semi-algebraic sets using CADs. *ACM SIGSAM Bulletin*, 37(4):97–108, 2003.

[6] B. Buchberger, G. E. Collins, R. Loos, and R. Albrecht, editors. *Computer algebra*. Symbolic and algebraic computation. Springer, 1982.

[7] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In *Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern, May 20–23, 1975*, pages 134–183. Springer, 1975.

[8] G. E. Collins. History of SACLIB. 1998.

[9] G. E. Collins, M. Encarnacion, H. Hong, J. Johnson, W. Krandick, A. Mandache, A. Neubacher, and H. Vielhaber. SACLIB user's guide. *RISC Linz, Johannes Kepler University, Linz*, 1993.

[10] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma. Linear invariant generation using non-linear constraint solving. In *International Conference on Computer Aided Verification*, pages 420–432. Springer, 2003.

[11] E. Darulova and V. Kuncak. Sound compilation of reals. *Acm Sigplan Notices*, 49(1):235–248, 2014.

[12] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[13] L. De Moura and G. O. Passmore. Computation in real closed infinitesimal and transcendental extensions of the rationals. In *International Conference on Automated Deduction*, pages 178–192. Springer, 2013.

[14] W. Denman and C. Muñoz. Automated real proving in pvs via metitarski. In *International Symposium on Formal Methods*, pages 194–199. Springer, 2014.

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

[16] B. Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014.

[17] W. ecker, G.-M. Greuel, G. Pfister, and H. Schönemann. Singular 4-1-0 — A computer algebra system for polynomial computations. http://www.singular.uni-kl.de, 2016.

[18] T. Granlund et al. *GNU MP 6.0 Multiple Precision Arithmetic Library*. 2015.

[19] W. Hart, F. Johansson, and S. Pancratz. FLINT: Fast Library for Number Theory, 2016. Version 2.4.0, http://flintlib.org.

[20] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety verification of deep neural networks. *arXiv preprint arXiv:1610.06940*, 2016.

[21] D. Jovanović. Solving nonlinear integer arithmetic with MCSAT. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 330–346. Springer, 2017.

[22] D. Jovanović and L. De Moura. Solving non-linear arithmetic. In *International Joint Conference on Automated Reasoning*, pages 339–354, 2012.

[23] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. *arXiv preprint arXiv:1702.01135*, 2017.

[24] J. Leike and M. Heizmann. Ranking templates for linear loops. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 172–186. Springer, 2014.

[25] J. Leike and A. Tiwari. Synthesis for polynomial lasso programs. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 434–452. Springer, 2014.

[26] The PARI Group, Univ. Bordeaux. *PARI/GP version 2.9.0*, 2016. available from http://pari.math.u-bordeaux.fr/.

[27] A. Platzer, J.-D. Quesel, and P. Rümmer. Real world verification. In *International Conference on Automated Deduction*, pages 485–501. Springer, 2009.

[28] V. Shoup. NTL: A library for doing number theory, 2001.