Quantifier elimination for polynomials in real arithmetic

Octavian Ganea

January 14, 2012

The following describes the statement of the problem and the possible approaches that I found by myself or reading papers related to the subject. From these, I implemented Cylindrical Algebraic Decomposition in Scala language .

1 Statement of the problem

In [2] is discussed a method for synthesis for linear rational arithmetic, that is, a method for quantifier elimination for formulas $\exists(x_1, x_2 \dots x_n)\phi$ where ϕ is a quantifier free formula containing variables $x_1, \dots x_n$. Formulas are obtained from atomic formulas using propositional operators $\wedge, \vee, \neg$ and parantheses. Atomic formulas are of the form $f(x_1, \dots x_n)\rho g(x_1, \dots x_n)$, where ρ is one of $\neq, <, >, \leq, \geq, =$ and f and g are linear expressions (polynomials of degree 1) of form $a_0 + a_1x_1 + a_2x_2 + \dots + a_nx_n$. All parameters are real numbers and can be unknown at compilation time (so some atomic formulas are parametrized). More exactly, the method provided in the paper checks if the formula is satisfiable and, if it is, returns a a valid assignment for the variables via choose() function.

Our goal is to treat the general case where instead of linear expressions, we can have polynomials of any degree, that means of form $\sum a \cdot x_1^{b_1} x_2^{b_2} \dots x_n^{b_n}$. We want to implement a version of FindInstance command from Mathematica: [5]

2 Rewriting the initial hypothesis

Following the ideas from the above paper, we will have the following reductions in the original formula:
• Reduce atomic formulas from $f(x_1, \dots x_n)\rho g(x_1, \dots x_n)$ to $h(x_1, \dots x_n)\rho 0$ by having $h = f - g$
• Replace all $\neg$ by: $\neg(f = 0) \to (f \neq 0)$, $\neg(f \geq 0) \to (f < 0)$ and so on.
• Rewrite the formula to elliminate all $\neq, >, \geq$: $f \neq 0 \to (f > 0) \vee (f < 0)$; $f > 0 \to (-f < 0)$; $f \geq 0 \to (-f \leq 0)$.

• Replace all $f(x) \leq 0$ by introducing a new variable x_{fresh}: $f(x) + x_{fresh}^2 = 0$.
• Rewrite the formula in disjunctive normal form: $\exists x.\phi_1 \vee \phi_2 \vee \ldots$. We have to solve each $\exists x.\phi_i$ separately from now.

So, by the above, we have to deal just with solving systems of polynomial equations and inequations of form $f(x) = 0$ or $f(x) < 0$.

If we have just equations (or $\leq$ which we showed can be reduced to equality) i.e. no strict inequalities in the original formula, then we can use some of the existent algorithms (which we will briefly describe in the following) - triangular decomposition into regular chains or homotopy. For some references, please see [3], [4].

If we have strict inequalities, for some particular cases we can use Lagrange multipliers. In the most general case, the best option is CAD - cylindrical algebraic decomposition.

3 Reducing general case to quadratic case

In the following, we want to reduce a general equation of form $f(x_1, \ldots x_n)\ \rho\ 0$ to one of the following 2 forms:
• $a + \sum b_i x_i^2\ = (<)\ 0$ (type 1)
• $c + \sum d_i x_i\ = (<)\ 0$ (type 2)

In order to do this, notice that one can rewrite the equation $xy = z$ in the following form:
$a - x - \frac{1}{2}y = 0$
$b - z - \frac{1}{2} = 0$
$b^2 - z^2 - a^2 + x^2 + \frac{1}{4}y^2 - \frac{1}{4} = 0$
where a and b are fresh variables.

Now, using the above trick, we can rewrite any $x^n = y$ in the above form (write first $x^{[\frac{n}{2}]} = z_1$ and $x^{n-[\frac{n}{2}]} = z_2$ recursively, and then rewrite $y = z_1 z_2$. Then, we can apply the same argument for products of different variables of form $y = x_1 x_2 \ldots x_k$.

For example, the polynomial $x^5 + x^3 + 1 < 0$ will be reduced to the system below:
$t_1 - z_1 = \frac{1}{2}$
$t_1^2 - x^2 - z_1^2 = \frac{1}{4}$ (this means that $z = x^2$)
$t_2 - x - \frac{1}{2}z = 0$
$t_3 - z_2 = \frac{1}{2}$
$t_3^2 - z_2^2 - t_2^2 + x^2 + \frac{1}{4}z^2 = \frac{1}{4}$ (this means that $z_2 = x^3 = xz$)
$t_4 - z_2 - \frac{1}{2}z = 0$
$t_5 - z_3 = \frac{1}{2}$
$t_5^2 - z_3^2 - t_4^2 + z_2^2 + \frac{1}{4}z^2 = \frac{1}{4}$ (this means that $z_2 = x^5 = z_2 \cdot z$)
$z_2 + z + 1 < 0$

So we showed that we can reduce any system of polynomials to a system of (more) polynomials of type 1 and type 2 by introducing some fresh variables. This is a simple form of the quadratic case equivalent to any system of polynomials. One problem with this approach is the complexity and the big number of fresh variables and equations introduced, but an advantage is the simple form of the resulting system. The number of obtained equations can be approximate by the linear in the sum of degrees of all terms of each polynomial. That is an unacceptable complexity for most of the examples.

4 Why we cannot use the same ideas as in synthesis for linear rational arithmetic

We are now in the quadratic case - each polynomial has degree at most 2).

The main idea from [2] was to eliminate at each step one variable from the entire system using some kind of Gaussian elimination. The approach works in the following way: if $f(x_1, \dots x_i) = 0$ then, because f is linear in its variables, rewrite it as $g(x_1, \dots x_{i-1}) = x_i$ and apply one point rule by taking the other expressions that contain x_i. Remark that, because f is linear, g will also be linear in its variables. For $f(x_1, \dots x_i) < 0$ the idea is the same : rewrite it as $g(x_1, \dots x_{i-1}) < (>) x_i$.

However, this approach cannot work in the quadratic case because $f(x_1, \dots x_i) = 0$ rewriten as $g(x_1, \dots x_{i-1}) = x_i$ will imply that g will no longer be linear or quadratic. In fact, for the base case $\exists x.(ax^2 + bx + c = 0) \wedge \phi$ we have the following equivalent formula:

$$(a = 0 \wedge b = 0 \wedge c = 0) \vee (a = 0 \wedge b \neq 0 \wedge \phi[-\frac{c}{b}/x]) \vee$$
$$(a \neq 0 \wedge b^2 - 4ac \geq 0 \wedge (\phi[\frac{-b + \sqrt{b^2 - 4ac}}{2a}/x] \vee \phi[\frac{-b - \sqrt{b^2 - 4ac}}{2a}/x]))$$

If a,b,c are polynomials in other variables (in fact a cannot be more than a constant and b more than a linear polynomial, because $ax^2 + bx + c$ has degree 2), then the substitution $\phi[\frac{-b+\sqrt{\Delta}}{2a}/x]$ will make each equation(inequation) from ϕ loose its quadratic form, but instead get the form $\frac{p+q\sqrt{\Delta}}{r} = (<)0$ where p, q, r and Δ are quadratic polynomials.

In [12] is described a quantifier elimination following this approach discussed here.

The main idea is that $\frac{p+q\sqrt{\Delta}}{r} = 0$ is equivalent to $pq \leq 0 \wedge (p^2 - q^2\Delta = 0)$ while $\frac{p+q\sqrt{\Delta}}{r} < 0$ is equivalent to

$$(pr < 0 \wedge (p^2 - q^2\Delta > 0)) \vee (qr \leq 0 \wedge (p^2 - q^2\Delta < 0 \ \vee \ pr < 0))$$

However, while this seems the natural approach, the polynomial $p^2 - q^2\Delta$ will no longer be quadratic if p is quadratic or q is quadratic, or q and Δ are at least linear. An idea is to add some simple optimizations that will increase the chances of the above method to be applicable (to always have quadratic constraints):
• Any variable that appears JUST linear in all equations will be eliminated exactly like it is described in the paper [2] and discussed above
• Any variable x_i that appears in ϕ JUST quadratically will rewritten as : $z \geq 0 \wedge \phi[z/x_i^2]$ and eliminated after, as described in the above step.

The author specifies in section 4 of the above paper that this method will not work in many cases.

5 A simple case:

One of the first ideas that I had was to deal with the case where we have just one expression of type 2 and one of type 1, but this one having just positive coefficients and more or the same number of variables than the first equation:
• $\sum_{i=1}^{n} a_i x_i^2 = (\leq) A$
• $\sum_{i=1}^{k} b_i x_i = B$
where $k \leq n$ and $a_i > 0, \forall i$.

For this simple case, the first step is to make k = n by introducing $b_j = 0, \forall j > k$.

Now, the above system has the following equivalent quantifier free formula:

$$A(\sum_{i=1}^{n} \frac{b_i^2}{a_i}) \geq B^2$$

This comes from Cauchy-Schwartz inequality. Moreover, from the above form one can obtain a satisfying assignment.

If equality occurs in the above condition and $\sum_{i=1}^{n} a_i x_i^2 = A$, then the only possible values for x_i are $x_i = \frac{A}{B}\frac{b_i}{a_i}$. If the inequality is strict, then there are an infinite number of solutions of x_i of the form $x_i = \frac{A}{B}\frac{b_i}{a_i} + \epsilon_i$ (with some restrictions on ϵ_i).

6 Triangular Decomposition

How can we use the particular form described in the previous section - reducing a system of polynomials to just a system of equations/inequations of type 1 and

type 2 ?

If we have just equalities (so no $<$), we may be able to reduce the system to a regular chain of triangular system of equations (see [6] and [3]):

$$\begin{cases} f_1(x_1) = 0 \\ f_2(x_1, x_2) = 0 \\ f_3(x_1, x_2, x_3) = 0 \\ \dots \\ f_n(x_1, \dots x_n) = 0 \end{cases}$$

If we can reduce a system to this triangular form, then solving it is easy: solve the first equation to find possible values of x_1, then, for each such value, replace x_1 in the rest of equations and repeat the process with $x_2, x_3 \dots$ until we are done. We can see that for the linear case, the above is a straight-forward Gaussian elimination.

However, the above situation is difficult for general polynomials and also there is the problem if the number of equations is different than the number of variables. An idea is to reduce the original system to the above form of equations of type 1 and type 2 (as described in section 3):

$$\begin{cases} \sum_{i=1}^{n} a_{ij} x_i = A_j, \forall j = 1 \dots k_1 \end{cases} \begin{cases} \sum_{i=1}^{n} b_{ij} x_i^2 = B_j, \forall j = 1 \dots k_2 \end{cases}$$

So we obtained 2 systems (dependent on each other) on the same set of variables - one of equations of type 1 (linear) and the other of equations of type 2. It is hard to apply Gaussian elimination on the entire set of equations (both linear and quadratic) because the substitutions can break the nice form of the system. However, by applying Gaussian elimination separately, but with the same lexicographic order on the variables, on the first system and on the second system, we will get:

$$\begin{cases} \phi_n(x_1, \dots x_n) \\ \phi_{n-1}(x_1, \dots x_{n-1}) \\ \dots \\ \phi_i(x_1, \dots x_i) \end{cases}$$

Where $\phi_j(x_1, \dots x_j)$ is either a linear equation in its variables of type 1, or a quadratic equation of type 2 or a conjunction of both.

If the last $\phi_i(x_1, \dots x_i)$ contains exactly one variable (i = 1), then the above approach will work, because we will find possible values of x_1, substitute each of them in the remaining equations and continue the procedure. If not, then try to see if this $\phi_i(x_1, \dots x_i)$ has an unique solution - maybe we are in the equality case of Cauchy inequality described above. If not - this means that it will have an infinite number of solutions which we cannot propagate to the rest of equations in a trivial way.

7 Lagrange multipliers

The next idea that I had was to search for Lagrange multipliers and their applicability, more specifically to a generalization of them that permits inequalities in the constraints - KKT conditions (see references [7] or [8]):

The main goal is to:
- maximize f(x) subject to constraints of the form
- $g_i(x) = 0, h_j(x) \leq 0,\ i = 1 \dots m,\ j = 1 \dots p$

where $x = (x_1, \dots x_n)$.

Then one can build the Lagrangian $L(x, \lambda, \mu) = f(x) - \sum_{i=1}^{m} \lambda_i g_i(x) - \sum_{j=1}^{p} \mu_j h_j(x)$ where λ and μ are fresh variables.

Then, a necessary condition for x^* that maximizes f(x) is

$$\nabla L(x, \lambda, \mu) = 0$$

$$\mu_j^* h_j(x^*) = 0$$

$$\mu_j^* \geq 0$$

where the differential is taken with respect to x_i and represents the vector of partial derivatives: $\nabla q(x_1, \dots x_n) = (\frac{\partial q}{\partial x_1}, \dots \frac{\partial q}{\partial x_n})$. This happens because any point of maximization of function f(x) with the above constraints is also an extreme point of L in all its variables (including λ, μ), that is $\nabla L(x, \lambda, \mu) = 0$. Remark that we can easily convert maximization to minimization by considering -f instead of f.

Why is this method helpful? Because it permits inequalities.
If we can rewrite the original system of polynomials as one quadratic constraint f(x) (quadratic polynomial = or < than 0) and the rest just linear constraints (equalities $g_i(x)$ or inequalities $h_j(x)$), then the above method will allow us to find a (finite) set of candidates x^* for $x = (x_1, \dots x_n)$. The optimal x is in this set, so we can try each element x^* and see which minimizes or maximizes f(x). Then refer to the original constraint of f(x) and see if we found a good solution (a solution that satisfies all our equality and inequality constraints). If none of the candidates work, then the original system has no solution.

Remark that all the constraints need to be linear in order to assure that the equivalent Lagrangian system described above contains linear equations (which we know how to solve by Gaussian methods and linear arithmetic); that is, to assure that $\nabla L(x, \lambda, \mu)$ is linear in all x_i. For this, we need to have f(x) at most quadratic, and constraints linear in their variables (of type 1).

An example of how to apply this method is the following:

$$\exists x.((x-2)^2 + 2(y-1)^2 \leq \frac{3}{2}) \wedge (x + 4y \leq 3) \wedge (x \geq y))$$

Using the above idea we can set $f(x,y) = -(x-2)^2 - 2(y-1)^2, h_1(x,y) = x + 4y - 3, h_2(x,y) = y - x$ and find the candidates in the system:
- $-2(x-2) - \mu_1 + \mu_2 = 0$
- $-4(y-1) - 4\mu_1 - \mu_2 = 0$
- $\mu_1(3 - x - 4y) = 0$
- $\mu_2(x - y) = 0$
- $x + 4y \leq 3$
- $-x + y \leq 0$
- $\mu_1 \geq 0$
- $\mu_2 \geq 0$

We have a finite number of solutions for this system, and only one gives a maximum for f(x,y): $x = \frac{5}{3}, y = \frac{1}{3}$. In this case, this is the same as the minimum of $(x-2)^2 + 2(y-1)^2$ which is 1. So we also obtained a satisfying asignment for the original constraint.

However, as said above, this method is difficult to use if we have more quadratic constraints.

8 Numerical methods - homotopy

For this method I attached a few references and presentations in the e-mail (there are many papers on this topic on the internet): [13] and [14]

Homotopies is a numerical method used to approximate solutions of systems of polynomial equations using a predictor correcto method. It is applicable if all the expressions in the initial formula are equalities (or can be reduced to equalities. For example for $f(x) \geq 0$ can be reduced to equality by introducing a fresh variable s and rewriting $f(x) + s^2 = 0$) and the number of equations is equal to the number of variables (see the above references). But I claim that it can be used even if the number of equations is strictly less than the number of variables (see below). That is, if we have a system of polynomials of form:

$$\begin{cases} f_1(x_1, \dots x_n) = 0 \\ \qquad\qquad \dots \\ f_k(x_1, \dots x_n) = 0 \end{cases}$$

where f_i are polynomials. Define $x = (x_1, \dots x_n)$ and $f(x) = (f_1(x), \dots f_k(x))$. The idea is to start with known solutions of a known start system and then track those solutions as we deform the start system into the system that we wish to solve. That is, we taks a system g(x) = 0, whose solutions we know, and make

use of a homotopy $H(x,t) = (1-t)g(x) + tf(x)$. We see that we know the solutions of $H(x,0) = 0$ and wish to find the solutions of $H(x,1) = f(x) = 0$. We can move our t slightly from 0 to 1 and see how the solutions previous found are deformed (changed) during these steps. With a good predictor-corrector method, we can approximate the final solutions of $H(x,1) = 0$.

Written differently, we need to approximate as good as possible the vector $x(t)$ such that $H(x(t),t) = 0$ when t goes from 0 to 1.

$H(x(t),t) = 0$ means in fact that the derivative of H with respect to t is 0:

$$0 = \frac{dH(x(t),t)}{dt} = \sum_{i=1}^{n} \frac{\partial H}{\partial x_i}\frac{dx_i}{dt} + \frac{\partial H}{\partial t}$$

To compute the paths we can use ODE methods for differential equations to predict and Newtons method to correct (see slide 14 of the PDF presentation attached for more details). Essentially, let's suppose we split the interval [0,1] in T equal regions. Then we take $t_0 = 0, t_T = 1, t_j = t_{j-1} + \frac{1}{T}$ and the initial $x(0)$ being the solutions of g(x) which we already know.

Now, for the predictor method, suppose we computed in the previous step $x(t_{j-1})$ and wish to compute $x(t_j)$. Then the above formula can be rewritten as the following system with unknowns $\Delta(x_i,t_j) = x_i(t_j) - x_i(t_{j-1})$:

$$0 = \sum_{i=1}^{n} \frac{\partial H}{\partial x_i} \cdot (x_i(t_j) - x_i(t_{j-1})) + \frac{\partial H}{\partial t} \cdot (t_j - t_{j-1})$$

Now, if the number of equations (k) is smaller or equal than the number of variables (n), then the above system with unknowns $\Delta(x_i,t_j)$ ca be solved as to approximate one solution of $x(t_j) = x(t_{j-1}) + \Delta(x_i,t_j)$ (note that if the number of equations is smaller than the number of variables, then we do not care to much because we need just one valid solution for $\Delta(x_i,t_j)$).

However, $x(t_j)$ the value what we computed above is a predictor value $x^p(t_j)$ which has to be corrected to get the real approximation for $x(t_j)$. Correction means that we need our final $x(t_j)$ to satisfy: $|H(x(t_j),t_j)| < \epsilon$ where ϵ is an acceptable error. For corrector we can use the Newton method with a maximum permitted number of iterations: see [9]

In summary, we start with $x_0 = x^p(t_j)$ as a column vector of n elements. Then we find an x_i based on x_{i-1} by solving the system:

$$(\nabla H)(x_i)(x_i - x_{i-1}) = -H(x_i)$$

where $(\nabla H)(x_i)$ is the kxn matrix of partial derivates of H with respect to x_i when parameter t is constant set to t_j. The above approach is a generalization of the 1 dimensional case described on wikipedia, in which $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$. The idea is to run this corrector method just a maximum number of times (at most 3) and check if the desired accuracy is obtained. If not, then it is a big chance that this path from 0 to 1 converges to infinity at some point, so we need to try other initial solutions x(0) of g(x). If all paths converge to infinity, then we can conclude (with a big chance) that our initial system of equations has not solutions.

As far as I read, this is one of the best existent numerical methods for approximating solutions of a system of equations. I showed above how it can be applied in our case (for the initial set of polynomials with no reductions needed). However, is just a numerical method, nothing symbolic is used here.

9 Univariate polynomials case - towards cylindrical algebraic decomposition

How can we deal with the case where we have just one variable and multiple polynomials: $f_1(x) = 0, f_2(x) < 0 \dots f_n(x) \geq 0$? The best idea will be, for each $f_i(x)$, to split the set of real numbers into intervals such that on each interval the polynomial $f_i(x)$ has constant sign - that is it is +, - or 0 (degenerate in this case). These intervals are bordered by $\pm\infty$ or roots of $f_i(x)$. For example, for $f(x) = x^2 - 3x + 2$ a valid representation will be $([-\infty, 1], +), ([1, 1], 0), ([1, 2], -), ([2, 2], 0), ([2, \infty], +)$. If we can build such a list for every polynomial $f_i(x)$, then we can combine all the lists at the end as to obtain another split of R in intervals such that none of the polynomials $f_j(x)$ changes its sign on the entire interval. Having this final list solves the original problem, as we need to scan each interval and see if the desired signs for $f_i(x)$ are all happening in the respective interval. If it does, than just return a sample point of the interval as a possible solution x of our system. If none of the intervals corresponds to our desired signs, then we will know that our formula is unsatisfiable. This method has also necessary and sufficient conditions for the satisfiablity of the initial expression.

It remains the question of how to find best the roots of a polynomial. If all polynomials have degree at most 4, then we know that there exists solutions expressible with radicals. Else, if the degree is greater than 5 we do not have a closed formula in general.

The approach I implemented is one of the most used in practice - Sturm chains. It is a numerical method rather than a symbolic one. However, if the degree of the polynomial is greater than 5, then numerical methods are the best we can hope.

Definition If $f_1, f_2, \dots f_r \in R[x]$ have the following properties with respect to the interval (a,b), then we call this sequence a Sturm chain of (a,b):
(i) $f_k(x) = 0 \Rightarrow f_{k-1}(x)f_{k+1}(x) < 0, a < x < b$
(ii) $f_r(x) \neq 0, a < x < b$

An interesting example of such Sturm chain is formed from any polynomial f_1 as following:
$f_2 = f_1'$
$f_3 = -f_1 \% f_2$ (that is, f_3 is minus the remainder of f_1 divided by f_2)
$f_4 = -f_2 \% f_3$
...
$f_r = -f_{r-2} \% f_{r-1}$
$0 = f_{r-1} \% f_r$

where it follows from the above that $f_r = gcd(f_1, f_2)$. This sequence is a Sturm chain in any interval (a,b) by construction.

Definition If $a_1, a_2 \dots a_r$ are real numbers, then the number of variations (between strict negative and strict positive) in sign of the sequence a_i is denoted by $var(a_1, \dots a_r)$. Also, denote for a sequence of polynomials $f_1, f_2, \dots f_r$ by $V(a) = var(f_1(a), f_2(a), \dots f_r(a))$

Theorem Let f_i be the Sturm chain of polynomials formed as described in the example above starting from an arbitrary polynomial f_1. Then the number of distinct real zeros of f_1 in (a,b) is V(a) - V(b).

We can use this to search and numerically find all roots of a polynomial. The ideea is to start with the interval $(-\infty, +\infty)$ and, knowing exactly how many roots we are searching for, perform sort of a binary search at each step: if we need to find K roots of f_1 in interval (a,b) then take $mid = \frac{a+b}{2}$ and look how many roots we should discover in (a, mid) and search for them and look how many roots we should find in (mid, b) and look for them. When the interval becomes smaller than a desired ϵ and there is just one root to find in that interval, then return any point from the interval.

While the above approach looks good at first sight, it has some problems. Here are these and how I solved them in my implementation:

1) What is ∞ ? We cannot say something like MAXINT because we would get errors when evaluating $f_1(\infty)$. So what to choose in order to avoid overflow errors? The main problem is that we need to have accurate values for $f_1(\pm\infty)$. What I did was to find a not so big upper bound for all real roots of an univariate polynomial. It is not hard to see by using standard absolute value inequality that the following is a good candidate for ∞ for a polynomial $f_1(x) = a_n x^n + a_{n-1}x^{n-1} + \dots a_1 x + a_0$:

$max(1, \sum_{i=0}^{n-1} |\frac{a_i}{a_n}|)$

For $-\infty$, just consider minus the above value.

2) When the interval (a,b) is too small, then evaluating the sign of $f_i(a)$ (needed for V(a)) may give errors because the actual value of $f_i(a)$ is too close to 0 to be computed accurately. In this case, the number of real roots will not be the correct one, thus leading to incorrect results or infinite loops. So, my ideea was to do the following things:
• i) Evaluate the polynomials using Horner Scheme in order to minimize numerical errors propagation. See [10]
• ii) When the number of roots in interval (a,b) becomes 1 (we expect this to happen when a and b are far one from another compared to ϵ) and the interval (a,b) is not too small, then switch to a standard numerical method to find that root - for example I implemented Newton's method. This is more stable for small intervals than the binary search.

3) How many epsilons to use ? What should be the relation between them. We cannot use just one epsilon because we have different notions of what is $a == b$ or $f(a) == 0$ or when 2 polynomials are equal (when their coefficients are almost equal). What I did was to try different values for these different situations and see what kind of results I obtain. The relation between these epsilons are very important. In some cases, the final results can be wrong if these epsilons are not set in a correct relation. Intuitively , the epsilon for $a == b$ should be much smaller than the one for $f(a) == 0$, but what is a correct decision ?

10 Cylindrical algebraic decomposition

Cylindrical algebraic decomposition (CAD) seems the best choice for the general case of polynomials in many variables and with different degrees as it is used in Maple or Mathematica and appears in many references. I used in my implementation the theoretical notions presented in a good paper with theory related to it [11].

For the case of 2 variables the procedure is clear:

Suppose again that we have multiple relations with just 2 vars x and y and in form of type $f_1(x, y) = 0, f_2(x, y) < 0 \dots f_n(x, y) \geq 0$

We want to split the entire R^2 into "rectangles" (product of intervals): $[a_i, a_{i+1}] \times [b_i(x), b_{i+1}(x)]$, where the first interval corresponds to x and the second to y. Note that for each x, the interval corresponding to y changes, as it is dependent on x. For example, the region inside a the unit circle in R^2 will

be described as $[-1,1] \times [-\sqrt{1-x^2}, \sqrt{1-x^2}]$.

Such "rectangles" are called in the following "cells". A cell is a region in R^2 such that all polynomials f_i do not change their respective signs on the entire cell (they have the same sign for every point (x,y) in the cell). If we can obtain something a split of the entire R^2 into a finite set of cells, then we will do essentially the same as described in the previous section : we have a goal of desired signs (maybe equalities with 0) for our polynomials - so we scan the entire set of cells; if we find one where the polynomials take our desired signs, then we return true and some arbitrary values of the 2 variables from the rectangle for for choose() function; if we do not find, then we return false - the formula is unsatisfiable. Note that because the polynomials have constant signs for every point in a cell, then we can keep a representative sample point (x,y) from each cell that we will return as the result of the choose() function. The main ideea of the algorithm is to find just these sample points, rather than computing the entire formula describing a cell. For example, instead of computing the entire unit circle as $[-1,1] \times [-\sqrt{1-x^2}, \sqrt{1-x^2}]$, the algorithm just finds the representative (0,0) for this cell. Thus, if the input inequation is $1 - x^2 - y^2 < 0$, the procedure will scan all cells until it scans the cell corresponding to the unit circle. Here it will just try to see if (0,0) works for our inequation (if the sample point of a cell does not give the desired signs, then any other point from that cell will not work). In this case the algorithm will return (0,0) for the formula above.

In general case for k variables, the idea described above is essentially the same: we need to split R^k into cells - non-intersecting regions that cover the entire R^k - such that the given set of polynomials has the same constant sign on each point of a cell. The CAD algorithm will not compute the equations of the cells, but instead just compute a representative sample point $(x_1, \ldots x_k)$ for each cell. This is sufficient because of the above definition of a cell. Now, if we have this set of representative points of all possible cells, then in order to find if our system of equations/inequations is satisfiable, we just need to scan all cells and for each cell we try the representative point of it and see if we obtain the desired signs. If yes, we return this representative point of R^k as the result of choose() function. If after the entire scan of all cells we were not succesfull, then we know for sure that the input formula is unsatisfiable.

So, the hardest point is to compute the representatives for the entire set of cells. This is what the CAD algorithm does.

Now, the procedure to build these cells is a bit complicated and is best described in the reference I used - [11]. I will try to summarize it here:

The algorithm has 3 phases: projection, base and extension. As said above, it takes as input a set F of initial n-variate polynomials (with n variables).

• The projection phase consists of n-1 steps; in each step a set of polynomials

with one less variable is constructed. The zero sets of the resulting polynomials of each step is the projection of "significant" points of the zero set of the preceeding polynomials - that is self-crossings, isolated points, vertical tangent points, etc. After n-1 steps we will obtain just univariate polynomials (in one variable). All the roots and sample points of the combined intervals between roots of these univariate polynomials are representative points of x variable corresponding to a valid set of cells of R^k. So, we start with the initial set $F = F_n \subset R[x_1, \ldots x_n]$, then from F_n find $F_{n-1} \subset R[x_1, \ldots x_{n-1}]$, and so on until we find $F_1 \subset R[x_1]$

• The base phase consists of isolation of real roots of the monovariate polynomials obtained in the previous phase. Take this set of points together with one point in each interval between two consecutive of these roots. We will call this final set the sample points of a decomposition of R^1 and denote the points by α_i. This phase is using the procedure I described in the previous section and will not be detailed more.

• Extension phase consists of n-1 steps. In the first step, a sample point $(\alpha_i, \beta_j) \in R^2$ of each cell of the "stack" over the cells of the base phase is constructed. That is, for each α_i found in the base phase, we replace x_1 by this value in all polynomials of F_2 and we obtain a set of univariate polynomials in x_2. We find all roots of polynomials in F_2 and sample points in the intervals between these roots - say we obtain a set $\beta_i(j)$ in this case. Then, the set $(\alpha_i, \beta_i(j)) \in R^2$ will be a set of first 2 coordinates of representative points for all cells in R^k that will correspond to the initial set of polynomials. We repeat the above procedure to find sample points for x_3 also, and so on until we find representative points of form $(x_1, x_2, \ldots x_n)$ for the entire set of cells in R^k. Then we are done.

The projection phase is the most complicated one and I will just sketch it here (it is more detailed in the given reference):

The ideea is that at each step we have a set of polynomials $F \subset R[x_1 \ldots x_n]$ and we would like to find a set of polynomials $proj(F) \subset R[x_1 \ldots x_{n-1}]$ in one less variable such that this set has a property: we know for sure that taking just the first n-1 coordinates from any sample point from any cell corresponding to the set F in R^k will cover all sample points of all cells corresponding to the set proj(F) in R^{k-1}. This assures that extension phase of the CAD algorithm is correct. In the following, the set F of polynomials will be seen as polynomials in variable x_n with coefficients polynomials in $R[x_1 \ldots x_{n-1}]$

In order to find the set proj(F) we need to consider 3 facts:

• Total number of complex roots of each polynomial $f(x_n) \in F$ has to remain constant. That is, the degree of each $f(x_n)$ should remain constant. Equivalent, the sign of the dominant coefficient of $f(x_n)$ should remain constant. So, why not add all coefficients of each $f(x_n) \in F$ in proj(F).

• Total number of distinct complex roots of each $f(x_n) \in F$ has to remain constant. That means first that the degree of $f(x_n)$ should remain constant and that the degree of the greatest common divisor of $f(x_n)$ and its derivative should remain constant - meaning the number of complex roots of f and its derivative is constant. So, add to proj(F) the dominant coefficient of the greatest common divisor between f and its derivative. For computing the dominant coefficient of the greatest common divisor between 2 polynomials there is a standard procedure described in the mentioned paper based on subresultant chains. I implemented it like it was described there.

• Total number of common complex roots between each $f_i(x_n), f_j(x_n) \in F$ has to remain constant. That means that we have to consider the dominant coefficient of the greatest common divisor between these 2 polynomials.

This is how proj(F) is built.

For more details and a complete set of formal definitions please see the main reference: [11] .

11 Results and conclusion

For the code and tests of the project that I implemented, please see: `https://svn.epfl.ch/svn/cylindrical-algebraic-decomp/trunk/`. A set of tests and the output obtained are there.

As an example successfully solved by my program, I gave the following equation:

$$x_1^2 - 4x_1 + 4 + x_2^2 - 4x_2 + 4 + x_3^2 - 4x_3 + 4 - 1 = 0$$

The program solved it by splitting R^3 in 25 cells:
Map(x1 – 0.0, x2 – 0.0, x3 – 0.0)
Map(x1 – 0.0, x2 – 1.0, x3 – 1.0)
Map(x1 – 1.0, x2 – 2.0, x3 – 1.0)
Map(x1 – 2.0, x2 – 2.0, x3 – 1.0)
Map(x1 – 3.0, x2 – 2.0, x3 – 1.0)
Map(x1 – 0.0, x2 – 3.0, x3 – 1.0)
Map(x1 – 0.0, x2 – 0.0, x3 – 2.0)
Map(x1 – 1.0, x2 – 1.0, x3 – 2.0)
Map(x1 – 2.0, x2 – 1.0, x3 – 2.0)
Map(x1 – 3.0, x2 – 1.0, x3 – 2.0)
Map(x1 – 0.0, x2 – 2.0, x3 – 2.0)
Map(x1 – 1.0, x2 – 2.0, x3 – 2.0)
Map(x1 – 2.0, x2 – 2.0, x3 – 2.0)

Map(x1 – 3.0, x2 – 2.0, x3 – 2.0)
Map(x1 – 4.0, x2 – 2.0, x3 – 2.0)
Map(x1 – 1.0, x2 – 3.0, x3 – 2.0)
Map(x1 – 2.0, x2 – 3.0, x3 – 2.0)
Map(x1 – 3.0, x2 – 3.0, x3 – 2.0)
Map(x1 – 0.0, x2 – 4.0, x3 – 2.0)
Map(x1 – 0.0, x2 – 1.0, x3 – 3.0)
Map(x1 – 1.0, x2 – 2.0, x3 – 3.0)
Map(x1 – 2.0, x2 – 2.0, x3 – 3.0)
Map(x1 – 3.0, x2 – 2.0, x3 – 3.0)
Map(x1 – 0.0, x2 – 3.0, x3 – 3.0)
Map(x1 – 0.0, x2 – 0.0, x3 – 4.0)

And choosing the final result:
Map(x1 – 2.0, x2 – 2.0, x3 – 1.0)

In the end, I can say that the implementation of CAD algorithm was successfull, even though numerical errors were a hard point to deal with.

References

[1] Code of the project: https://svn.epfl.ch/svn/cylindrical-algebraic-decomp/trunk/

[2] "Functional synthesis for linear arithmetic and sets" - V. Kuncak and all

[3] http://en.wikipedia.org/wiki/Systems_of_polynomial_equations

[4] http://reference.wolfram.com/mathematica/tutorial/RealPolynomialSystems.html

[5] http://reference.wolfram.com/mathematica/ref/FindInstance.html

[6] http://en.wikipedia.org/wiki/Regular_chain

[7] http://en.wikipedia.org/wiki/KarushKuhnTucker_conditions

[8] http://mat.gsia.cmu.edu/classes/QUANT/NOTES/chap4/node6.html

[9] http://en.wikipedia.org/wiki/Newton's_method

[10] http://en.wikipedia.org/wiki/Horner_scheme

[11] "Cylindrical Algebraic Decomposition - an Introduction" - Mats Jirstrand and all

[12] "Quantifier elimination for real algebra - quadratic case and beyond" - V. Weispfenning

[13] "Numerical solution of multivariate polynomial systems by homotopy continuation methods" - T.Y.Li

[14] Finding all solutions to polynomial systems and other systems of equations - Garcia, Zangwill