Efficient Solution to Systems of Multivariate Polynomials using Expression Trees

Gershon Elber
Dept. of Computer Science
Technion – IIT
Haifa 32000, Israel

Tom Grandine
The Boeing Company
Seattle

December 3, 2007

Abstract

In recent years, several quite successful attempts have been made to solve systems of polynomial constraints, using geometric design tools, by making use of *subdivision based solvers*. This broad class of methods includes both binary domain subdivision as well as the projected polyhedron method of Sherbrooke and Patrikalakis [13].

One of the main difficulties in using subdivision solvers is their scalability. When the given constraint is represented as a tensor product of all its independent variables, it grows exponentially in size as a function of the number of variables. In this work, we show that for many applications, especially geometric, the exponential complexity of the constraints can be reduced to a polynomial one by representing the underlying problem structure in the form of expression trees that represent the constraints. We demonstrate the applicability and scalability of this representation and compare its performance to that of tensor product constraint representation, on several examples.

Keywords: interval arithmetic; multivariate polynomial constraint solver; self-bisectors; Contact computation; Hausdorff distance.

1 Introduction and Related Work

Sets of multivariate polynomial constraints are frequently prescribed and solved in many applications in science and engineering in general, and in geometric modeling, in specific. A key module in many modern geometric modeling packages is a constraint solver. Constraints such as distance, orthogonality, parallelism, and convexity, are all used to capture desired properties. The constraint solver has the responsibility of finding (all) valid geometric solutions (configurations) that satisfy these constraints (the design intent), in the modeling environment.

Solving a set of non linear constraints is a difficult problem. In the univariate case, the properties of Bézier and B-spline shapes have been exploited toward this end, to a great extent [9]. Drawing on the convex hull property [4], a Bézier/B-spline function cannot contain a zero if all its coefficients are positive (or all negative). With the ability to divide a spline function into two, one is provided with a robust divide-and-conquer algorithm to converge on the zeros of the function. Hence after, we denote such solvers as *subdivision solvers*.

The ability to exhaustively examine the specified domain of a function for zeros, allows one to robustly seek all solutions, and provide a global (optimal) answer. The extension of this scheme to multivariates was first proposed in [13]. In [13], polynomial functions are considered and the domain is reduced using (projected) Bézier clipping [11]. In [7], this work was extended to handle B-spline functions as well and a proposal was made for a geometric test for a single solution existence. The division of the domain could clearly be stopped in the following cases:

- The domain has no solution (i.e. all coefficients are positive),
- The domain is small enough, below some user specified tolerance.

However, in the second case, one can only assume a solution exist in the domain. Moreover, there could be an arbitrary number of solutions to the function, in the second case. Hence, identifying a domain as having at most a single solution can aid in resolving the topology of the solution space as well as allow the early termination of the division process. In [8], an efficient algorithm was proposed to compute this proposed single solution test as

well as proposed the use of parallel hyper-planes to bound and further isolate roots in the multivariate solver.

In [10], an orthogonalization approach to precondition the constraints was suggested so that the Bézier clipping step is becoming more efficient. [10] also suggested other improvements such as the use of the upper and lower envelope of the Bézier clipping projection as control polygons of two new Bézier forms. These Bézier forms still bounds the original function and hence can be used as a tighter bound during the Bézier clipping step.

Recently, in [2], a proposal was made to bound a univariate constraint using a second order envelope and clip the domain of the constraint with the aid of the zeros of this envelope. With a super-quadratic convergence, this work was also extended to handle multivariates in [3].

Non linear constraint solvers that are subdivision base have been successfully used, in recent years, in solving quite a few geometric problems [7, 10]. Being highly robust, focused on real roots, and capable of finding the (global) optimal solutions, these subdivision solvers have captured their place as an essential tool in handling constraints. One main drawback of this type of solvers stems from their lack of scalability. By typically exploiting the tensor product representation, the size of the tensor product constraint grow exponentially with its number of degrees of freedom (variables). Let n be the number of variables in a constraint, and let k be the number of coefficients in each variable. Then, the size of the tensor product constraint is of order $O(k^n)$. This exponential complexity renders the use of tensor product impractical for anything but a few variables.

Several suggestions where made, in recent years, to use triangular hyper-patches, exploiting barycentric blending functions. For instance, see [12]. While this alternative somewhat reduces the size of the constraint, asymptotically it remains exponential. Moreover, the triangular hyper-patches representation is not capable of capturing all algebraic constraints.

In this work, we propose a different alternative to represent non linear constraints, an alternative that nicely fits the way constraints are created in geometric design. This proposed representation only shows polynomial growth with the number of variable and hence is far more promising. The rest of this paper is organized as follows. In Section 2, we introduce the basic representation which is *expression trees* (ET). In Section 3, we introduce a constraint solver based on this ET representation, and in Section 4, some examples are shown, comparing the ET representation to regular tensor product representation. In Section 5, some extensions are discussed and finally, we conclude in Section 6.

2 Expression Trees

Binary trees are a common mean in computer science to represent (arithmetic) expressions [1]. Techniques for parsing the common infix notation (where the operator is between the operands) and converting it into a binary tree are well known [1]. Expressions are converted into binary trees, where unary operators (i.e. the unary minus or the sin function) are converted into degenerate nodes in the binary trees, with a single son. See Figure 1 for the arithmetic expression example of $\sin(-z)/(x+y*5)$.

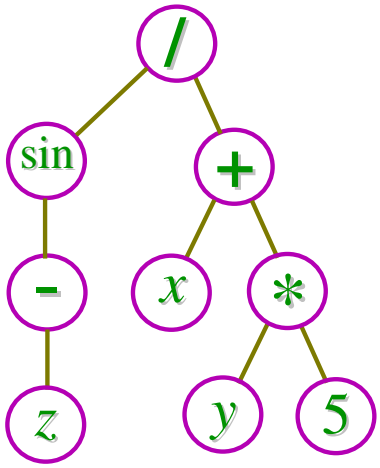

Figure 1: An example of a binary tree created for the expression $\sin(-z)/(x+y*5)$.

Having infix expressions represented as binary trees offers numerous advantages in computing. For example, symbolic differentiation is reduced to the simple task of traversing the tree and substituting each note for its derivative, recursively. As a second example, optimization of an expression could be done via the application of local rules (i.e. substitute $x/1$ by x) or by search and merge of common expressions, converting the tree into a DAG (directed acyclic graph).

In this work, we deal with algebraic expressions. Hence, the set of operators one needs to support in expression trees (ETs) is surprisingly small, namely: addition, subtraction, multiplication, and possibly a few others. Dealing with vector functions (i.e. parametric curves and surfaces in $\mathbb{R}^2$ and $\mathbb{R}^3$), one is also frequently employing inner products and possibly cross (outer) products. Division, in many cases, could be posed as multiplications and in other cases rational forms must be used.

With this representation, we will, in the next section, explore the potential benefits of ETs in using and solving sets of algebraic constraints.

3 Expression Trees based Solver

As a simple example that will follow us throughout the discussion, consider the problem of computing all intersection points of two planar parametric curves, $C_1(t) = (x_1(t), y_1(t))$ and $C_2(r) = (x_1(r), y_1(r))$. Algebraically,

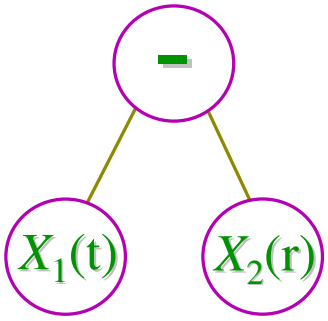

Figure 2: The ET of $x_1(t) - x_2(r)$.

the equivalent constraint are,

$$\begin{aligned} x_1(t) &= x_2(r), \\ y_1(t) &= y_2(r), \end{aligned} \tag{1}$$

having two equations and two unknowns. Assume both curves have k control points. Then, each of the two constraint in Equations (1) is a bivariate function with $O(k^2)$ coefficients, if represented as a tensor product. Now, the same constraints, represented as an ET will look as in Figure 2, with the memory cost of only order $O(k)$.

In general, a constraint with n degrees of freedoms (i.e. n curves), each of which has k coefficients (control points), will have $O(n^k)$ coefficients in all, when represented as a tensor product. In contrast, if this constraint could be represented as an ET with m operators, its size will only be of order $O(mk)$.

The increase in memory size when one represents an operation such as a difference between two curves as a tensor product, needs some clarification. Denote by a '$\diamond$' a binary operator ('$-$', '$+$' or '$*$') and let curve C_i be a Bézier or B-spline curve of the form,

$$C_i(t) = \sum_{o=0}^{k_i} Q_o^i B_o^i(t).$$

Then

$$\begin{aligned} & F(t,r) \\ &= C_1(t) \diamond C_2(r) \\ &= \sum_{o=0}^{k_1} Q_o^1 B_o^1(t) \diamond \sum_{p=0}^{k_2} Q_p^2 B_p^2(r) \\ &= \sum_{o=0}^{k_1} Q_o^1 B_o^1(t) \sum_{p=0}^{k_2} B_p^2(r) \diamond \sum_{o=0}^{k_1} B_o^1(t) \sum_{p=0}^{k_2} Q_p^2 B_p^2(r) \\ &= \sum_{o=0}^{k_1} \sum_{p=0}^{k_2} Q_o^1 B_o^1(t) B_p^2(r) \diamond \sum_{o=0}^{k_1} \sum_{p=0}^{k_2} Q_p^2 B_o^1(t) B_p^2(r) \\ &= \sum_{o=0}^{k_1} \sum_{p=0}^{k_2} (Q_o^1 \diamond Q_p^2) B_o^1(t) B_p^2(r). \end{aligned} \tag{2}$$

In other words, the $O(k^2)$ control points of the bivariate function $F(r,t) = C_1(t) \diamond C_2(r)$ are $(Q_o^1 \diamond Q_p^2)$, as an outer product of all the $O(k)$ control points of $C_1(t)$ and all the $O(k)$ control points of $C_2(r)$.

Expression trees can have arbitrary number of operators in them. While typically small, denote the number of operators in the tree by m. Then, the actual memory complexity of an ET is of order $O(km)$, taking into account both the size of the leaves (each of order $O(k)$) and the size of the tree itself (of order $O(m)$).

Representing the expression $x_1(t) - x_2(r)$ as an ET not only reduces the memory consumption but also reduces the computation cost. In this work, we deal with expression trees where the internal nodes are, for the most part, '$\diamond$' operators, and the leaves are algebraic functions (such as $x_1(t)$ or $x_2(r)$). By keeping algebraic functions that depends on *different variables* as separated leaves in the tree, the tensor product is represented implicitly and the explosion in data size is eliminated. The tensor product representation is sparse and a large reduction in memory consumption can be gained, if avoided.

During the recursive (zero) search iterations of the subdivision solver, two principal operations are usually performed:

- Examination whether or not the expression can hold zeros. For a tensor product Bézier/B-spline representation, this could amounts to examining all the coefficients for their minimum and maximum values.

- Dividing and/or reducing the domain where the zeros are sought. For a tensor product Bézier/B-spline representation, this could amounts to applying a subdivision/clipping along, typically, one of the parameters of the multivariate.

For an n dimensional tensor product, both operations visits all the coefficients of the tensor product multivariate and hence cost $O(k^n)$ time, where k is the size of the multivariate in one direction. In the next two sections, we will show how can these two operation be performed in $O(km)$ time, eliminating the exponential complexity dependency on the number of variables, n, inherit to the tensor product representation.

3.1 Zero testing in ETs

Let $\mathcal{E}_T^1$ and $\mathcal{E}_T^2$ be two expression trees and consider some '$\diamond$' operation between them. Further, assume the interval of values $\mathcal{E}_T^i$ can assume are between $[m_i, M_i]$. Then, the following rules of interval arithmetic apply:

Lemma 3.1 *The following intervals bound the result of the following $\diamond$ operation:*

1. *$' \diamond' = '+'$. $\mathcal{E}_T^1 + \mathcal{E}_T^2$ can assume values between $[m_1 + m_2, M_1 + M_2]$.*

2. *$' \diamond' = '-'$. $\mathcal{E}_T^1 - \mathcal{E}_T^2$ can assume values between $[m_1 - M_2, M_1 - m_2)]$.*

3. $'\diamond' = '*'$. $\mathcal{E}_T^1 * \mathcal{E}_T^2$ *can assume values between* $[\min(m_1m_2, m_1M_2, M_1m_2, M_1M_2), \max(m_1m_2, m_1M_2, M_1m_2, M_1M_2)]$.

Proof: As can be seen in Equation (2), the coefficients of $\mathcal{E}_T^1 \diamond \mathcal{E}_T^2$, as a Bézier or B-spline function, are $Q_o^1 \diamond Q_p^2$, but $Q_o^1 \in [m_1, M_1]$ and $Q_o^2 \in [m_2, M_2]$. Hence, the sum cannot be smaller than $m_1 + m_2$ nor can it be larger than $M_1 + M_2$. Similarly, the difference cannot be smaller than $m_1 - M_2$ nor can it be larger than $M_1 - m_2$.

The product is a bit more involved as signs can change and hence one must examine all four possibilities of m_1m_2, m_1M_2, M_1m_2, M_1M_2, for both the minimum and maximum values.

■

These rules follow the same ideas found in interval arithmetic computations. However, examine the computational complexity of evaluating such a '$\diamond$' constraint in an ET. If $\mathcal{E}_T^i$ contains $O(k)$ coefficients, the evaluation of the interval of values $\mathcal{E}_T^1 \diamond \mathcal{E}_T^2$ can assume is reduced to evaluating the values $\mathcal{E}_T^i$, $i = 1, 2$ can assume and the cost of the interval arithmetic operator, following Lemma 3.1, which is a constant time.

For our example of intersections of two curves (Recall Figure 2), the interval computation of $C_1(t) \diamond C_2(r)$ cost $O(k) + O(k) + O(1)$ operations or $O(k)$ operations in all. If the ET has m operators (internal nodes), it also has $O(m)$ leaves, the primitive algebraic functions in hand. With each leaf having $O(k)$ coefficients, a tree of m '$\diamond$' operators could be tested for the possible values it can assume, in order $O(mk)$ time.

It should be noted that the interval bounds established in Lemma 3.1 for these three '$\diamond$' operators in ETs are as tight as the tensor product representation. This, because both the tensor product representation and the ET representation examines the coefficients with the '$\diamond$' operator applied to them, by exhaustively examining all possible pairing of $Q_o^1 \diamond Q_p^2$.

Beyond the three basic '$\diamond$' operators of '$-$', '$+$' and '$*$', it is frequent to employ the *inner-product* operator between geometric entities, when forming geometric constraints. While one can see inner- (and cross-) products as generalized sets of products and summations, this is not always the case as these vector functions intermix different, sometimes independent, functions. Dealing with vector functions, the interval arithmetic computation becomes more complex.

Let $\mathcal{E}_T^1$ and $\mathcal{E}_T^2$ be two vector ETs and consider the inner product operator $\bullet$ between them. Further, let the interval of values the j'th element in vector function $\mathcal{E}_T^i$ can assume be between $[m_i^j, M_i^j]$. Then,

Lemma 3.2 *the interval of values vector functions* $\mathcal{E}_T^1$ *and* $\mathcal{E}_T^2$ *can assume in* $\mathcal{E}_T^1 \bullet \mathcal{E}_T^2$ *is bounded to be between*

$$\left[\sum_j \min(m_1^j m_2^j, m_1^j M_2^j, M_1^j m_2^j, M_1^j M_2^j), \right. \\ \left. \sum_j \max(m_1^j m_2^j, m_1^j M_2^j, M_1^j m_2^j, M_1^j M_2^j), \right]. \quad (3)$$

Proof: To see that Equation (3) indeed bounds the possible values an inner product can assume, one should first note we compute the minimum and maximum values each element can assume, in the vectors, like in a regular product, for all the (j'th) elements of the vector.

Then, the minimum bounding value of the inner product is set to be the sum of all the minimum values each element can assume whereas the maximum bounding value of the inner product is set to be the sum of all the maximum values each element can assume.

■

Unlike the three '$\diamond$' scalar operations, the interval bounds we established in Equation (3) for the inner product operation between ETs is not as tight as inner products between tensor products. Internal cancellation between the different elements of the inner product of the two vector functions are likely to yield a smaller interval than the bound we compute for ETs. Nonetheless, this bound is also established in a constant time (assuming the size of the vector is constant). We will show, in Section 4 that this bound is sufficiently tight to yield better results than tensor products constraints, in the vast majority of cases.

Needless to say an ET can hold zeros, if and only if the interval of values it can assume contains the zero. In the next section, we also show how to perform subdivision operations over ETs.

3.2 Subdivision of ETs

With the ability to efficiently determine if an ET can hold a zero, the subdivision of the domain of an ET along one variable, u, is even simpler. The following traversal of ET needs to be perform:

- Every algebraic expression (leaf) that holds u is locally divided.
- Every algebraic expression (leaf) that does not hold u is locally copied.
- Every operator (internal) node is locally copied.

The cost of subdividing an algebraic expression (leaf) with $O(k)$ control points is $O(k)$ (assuming the degree is constant). Obviously, copying a leaf of $O(k)$ coefficients costs $O(k)$. Hence, the overall cost of the proposed traversal is bounded by the number of coefficients in the entire ET that needs to be either copied or subdivided. This number is of order $O(mk)$.

Consider the curve curve intersection example $F(r, t) = x_1(t) - x_2(r)$ (recall Figure 2). A subdivision of $F(r, t)$ along t (r) can clearly be done in $O(k)$ time: traverse the

tree, copy the root operator and the leaf $C_2(r)$ ($C_1(t)$) and subdivide the other leaf $C_1(t)$ ($C_2(r)$). The total cost is hence $O(1) + O(k) + O(k)$ or $O(k)$ in all.

Being able to perform the two principal operations of a subdivision solver in a computational cost that is no longer exponential in the number of variables opens the way to employ this type of solver on a much larger scale problems. The use of such a solver for more than a few variables is becoming possible. The next section presents a few examples of solving non-linear algebraic constraints using both the tensor product representation and using this newly introduced ET approach, and compare the two schemes for scalability in terms of both memory use and computation times.

4 Examples

In this section, we present several examples of sets of algebraic constraints formed out of geometric problems. All measurements where made on a modern PC with 2G of memory.

Consider the intersection location of three surfaces in $\mathbb{R}^3$, $S_i(u_i, v_i) = (x_i(u_i, v_i), y_i(u_i, v_i), z_i(u_i, v_i))$, $i = 1, 2, 3$. Finding the solution location could be formulated using the following six constraints:

$$\begin{aligned} x_1(u_1, v_1) &= x_2(u_2, v_2), \\ x_1(u_1, v_1) &= x_3(u_3, v_3), \\ y_1(u_1, v_1) &= y_2(u_2, v_2), \\ y_1(u_1, v_1) &= y_3(u_3, v_3), \\ z_1(u_1, v_1) &= z_2(u_2, v_2), \\ z_1(u_1, v_1) &= z_3(u_3, v_3), \end{aligned} \tag{4}$$

having six equation and six unknowns. Figure 3 shows an example with three quadratic by cubic B-spline surfaces with 31×12 control points. With over 600 control points, each of the six constraints in Equations (4), is a 4-variate with over $360,000 = 600^2$ control points (or over a million coefficients), as a tensor product.

Solving this specific problem using a tensor product representation takes several seconds and consumes about 600 megabytes of data. In contrast, using expression trees, the same system is solved in a fraction of a second and only a few megabytes of memory.

Contact point computations between freeform shapes play, for instance, a major role in design, robotics, and NC machining. In this second example, we seek to derive the contact point between two C^1 freeform surface shapes, $S_1(u_1, v_1)$ and $S_2(u_2, v_2)$, in $\mathbb{R}^3$, when one surface is stationary and the second is moving according to some scale and translation transformation $T(t)$. At the contact points, the surfaces are tangent to each other, a condition that can be formulated algebraically as,

$$\begin{aligned} T(t)[x_1(u_1, v_1)] &= x_2(u_2, v_2), \\ T(t)[y_1(u_1, v_1)] &= y_2(u_2, v_2), \end{aligned}$$

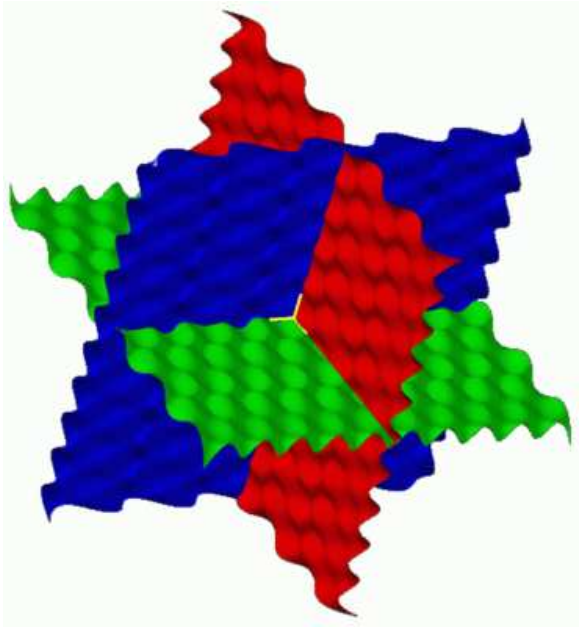

Figure 3: An example of three general B-spline surfaces intersecting in a single point (in yellow).

$$\begin{aligned} T(t)[z_1(u_1, v_1)] &= z_2(u_2, v_2), \\ \left\langle \frac{\partial T(t)[S_1(u_1, v_1)]}{\partial u_1}, \frac{\partial S_2(u_2, v_2)}{\partial u_2} \times \frac{\partial S_2(u_2, v_2)}{\partial v_2} \right\rangle &= 0, \\ \left\langle \frac{\partial T(t)[S_1(u_1, v_1)]}{\partial v_1}, \frac{\partial S_2(u_2, v_2)}{\partial u_2} \times \frac{\partial S_2(u_2, v_2)}{\partial v_2} \right\rangle &= 0, \end{aligned} \tag{5}$$

where $T(t)[\cdot]$ denotes the transform (translation and scale) operator.

In Figure 4, the two surfaces are biquadratic B-spline surfaces with a mesh size of 14×7. The motion curve is a cubic B-spline curve with 29 control points. The attempt to solve these constraints using expression trees exploited four Megabyte of memory and took about 60 seconds. The same attempt to solve Constraints (5) using a tensor product representation failed after five minutes of computation due to luck of memory (in a 2 gigabyte machine).

Our third example deals with the Hausdorff distance between planar curves. In [5], we show how the Hausdorff distance between two planar curves could be computed by isolating all the events where this extreme distance can occur. Consider two planar parametric curves $C_1(t)$ and $C_2(r)$. One case when the Hausdorff distance between $C_1(t)$ and $C_2(r)$ can occur is when $C_1(t)$ intersects with the bisector curve of $C_2(r)$. This condition could be formulated using the following set of constraints,

$$\begin{aligned} &\langle C_1(r) - C_2(t), C_1(r) - C_2(t) \rangle \\ &\qquad = \langle C_1(r) - C_2(s), C_1(r) - C_2(s) \rangle, \\ &\langle C_1(r) - C_2(t), C_2'(t) \rangle = 0, \\ &\langle C_1(r) - C_2(s), C_2'(s) \rangle = 0, \end{aligned} \tag{6}$$

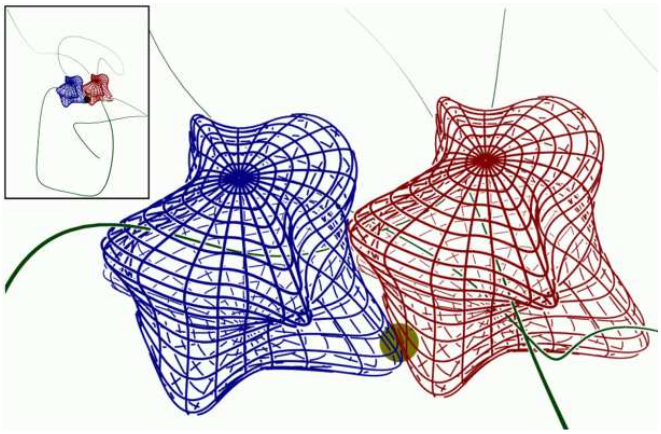

Figure 4: An example of seeking the contact point between two similar freeform shapes. One shape is stationary (blue) and the other is moving (red) along some freeform trajectory (in green). One computed contact locations is displayed, and the contact point itself is highlighted in yellow. The top left corner shows a snap-shot of the entire path.

having three equations and three unknowns. Note C_2 is independently parameterized twice, for the two foot points of its bisector curve. The first constraint in Equations (6) vanishes for $t = s$ (and the last two coalesce), a self-intersection problem that is dealt with in [6] and is beyond this paper. Nonetheless, attempting to solve this set of constraint using a tensor product representation of Equations (6), for curves of different number of control points yields the following result:

k (# Ctl. Pts.)	5	10	20	50
Time (Secs.)	2.93	2.13	8.47	326
Space (Mb.)	14	14.5	77	1470

Using expression trees for the same set of constraints yields the following comparable result:

k (# Ctl. Pts.)	5	10	20	50
Time (Secs.)	2.44	1.81	2.97	23
Space (Mb.)	7	7	7.5	11.5

Examining the scalability of the two representations, the expected benefits are obvious. Even for small curves the reductions in time and in memory use are clear. Moreover, for moderate to large curves, the difference is now between the ability (a few megabytes of memory) and inability (gigabytes of memory) to solve the problem, consuming huge amount of memory in the case of tensor products. Because the first constraint in Equations (6) is a tensor (outer) product of essentially three curves, if $n = 50$, this trivariate constraint, as a tensor product, has $125,000 = 50^3$ coefficients. In practice, it has more because the inner product in the constraint doubles the degrees.

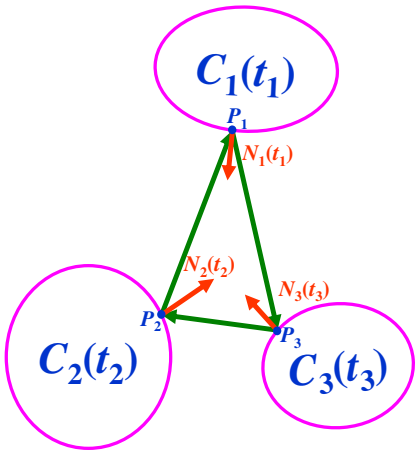

Figure 5: the concept of a ray trap between three planar curves.

Our last geometric example is the problem of *ray-traps* also known as the *bouncing billiard balls* problem. The general problem could be consider for k general objects in $\mathbb{R}^m$. Herein, we consider it for planar curves in $\mathbb{R}^2$: Given k planar parametric curves, $C_i(t_i)$, $i = 1, k$, in the xy plane, a ray-trap of length k is a set of k points $P_i = C_i(t_i)$ such that a bouncing ray off P_i towards $P_{(i+1)\mathrm{mod}k}$ will be reflected at $P_{(i+1)\mathrm{mod}k}$ toward $P_{(i+2)\mathrm{mod}k}$. See Figure 5 that conceptually shows the case of a ray trap between three planar curves.

Let N_i denote a normal field of curve C_i. Then, this problem (See also [7]) could be formulated using the following constraint, for each P_i contact,

$$\frac{\left\langle P_{(i-1)\mathrm{mod}k} - P_i, N_i \right\rangle}{||P_{(i-1)\mathrm{mod}k} - P_i||} = \frac{\left\langle P_{(i+1)\mathrm{mod}k} - P_i, N_i \right\rangle}{||P_{(i+1)\mathrm{mod}k} - P_i||}, \tag{7}$$

or in terms of the k parametric curves,

$$\frac{\left\langle C_{(i-1)\mathrm{mod}k}(t_{(i-1)\mathrm{mod}k}) - C_i(t_i), N_i \right\rangle}{||C_{(i-1)\mathrm{mod}k}(t_{(i-1)\mathrm{mod}k}) - C_i(t_i)||} = \frac{\left\langle C_{(i+1)\mathrm{mod}k}(t_{(i+1)\mathrm{mod}k}) - C_i(t_i), N_i \right\rangle}{||C_{(i+1)\mathrm{mod}k}(t_{(i+1)\mathrm{mod}k}) - C_i(t_i)||}. \tag{8}$$

Due to the normalization factors in the denominator, the square of the equation is used from now on, which is algebraic. While C_i is required to be regular for N_i to be defined, N_i need not be a unit vector fields as it appears in both sides of the constraint.

For n curves, one needs to handle n constraints of the form shown in Equation (8). Herein, we consider an example with $n = 3$ and for curves with increasing complexity (number of control points). See Figure 6 as an actual example for a case of $n = 3$.

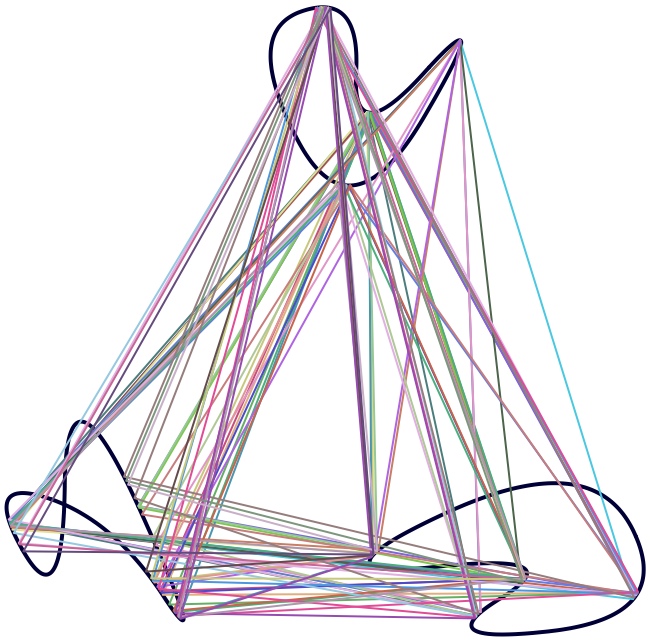

Figure 6: An example of all ray traps between three planar quadratic B-spline curves, each with 7 control points. 44 ray traps are identified.

Using a tensor product representation of constraints of the square of Equation (8) we get,

k (# Ctl. Pts.)	13	23	33	43
Time (Secs.)	3	40	152	Died
Space (Mb.)	67	463	1650	> 1700

Using expression trees for the same set of (squared) Constraints (8) yields the following comparable result:

k (# Ctl. Pts.)	13	23	33	43
Time (Secs.)	7	44	139	327
Space (Mb.)	5	7	10	18

The reduction in memory use is again substantial. However, in this case, for small scale problems, the tensor product computation is faster. This could be explained by the fact that the expression tree here is larger. Hence, the benefit in using the ETs is insufficient to compensate for the traversal cost of large ETs. Nonetheless, for moderate and large size curves, the computation time using ETs is again becoming better than in the tensor product representations.

5 Extensions

The use of expression trees in solving sets of non linear (geometric) constraints can greatly enjoy the vast list of techniques developed in computer science [1] to process parsed data and optimize their evaluations. In the case of arithmetic expression trees, the evaluations of leaves, being numbers, is typically very simple. Herein, however, the leaves are parametric forms, and reducing leaves' processing can greatly benefit the overall computation times. Merge of common sub-expression, which is reduced to finding identical sub-trees in the ETs, can do just that. This practice is well known for arithmetic expression evaluations and can be clearly adapted here as well. The expressions will no longer be represented as trees but rather as directed acyclic graphs (DAGs).

Another example where the processing of ETs could be further optimized is when operation are applied to only certain nodes/leaves in the graph. Reconsider the ET in Figure 2. If we divide/reduce the domain of this ET in t, only $x_1(t)$ is to be divided while $x_2(r)$ remains unmodified. Clearly, one can reference the current $x_2(r)$ instead of copying it, using pointer reference counting. With this data structures' accounting, the overall complexity of the subdivision/domain reduction computation in t (r) is down to the size of only $x_1(t)$ ($x_2(r)$).

Not always ETs should be preferred. If a binary expression contains operands with the same variables on both sides, one can consider converting this expression to a tensor product as no exponential growth is expected. For example, reconsider the last two constraints in Equation (5). The cross product of $\frac{\partial S_2(u_2,v_2)}{\partial u_2} \times \frac{\partial S_2(u_2,v_2)}{\partial v_2}$ could be precomputed as a tensor product as it stays dependent on only two variables (u_2, v_2). Moreover, the cross product is a difficult operation to bound. During the execution of the solver, the processing of $\frac{\partial S_2(u_2,v_2)}{\partial u_2} \times \frac{\partial S_2(u_2,v_2)}{\partial v_2}$ as an ET is eliminated while the size of this tensor product is only slightly larger than that of the operands (due to the raised degrees).

6 Conclusions and future work

In this work, we have presented an alternative representation to sets of non-linear constraints, that yields a significant reduction in memory and computational costs. This representation makes it possible to handle larger sets of constraints (and variables) than are feasible with the current tensor product representation.

Constraints could be created from geometric entities and their relations. In geometric context, the construction of ETs is fairly straightforward. However, in other cases, the sources of the constraint might be unknown and/or the tensor product of the constraint might be the only available input. How to efficiently decompose a tensor product constraint into its independent variables, if at all possible, is a question that deserve further investigation. Clearly one can intermix tensor products inside ETs but it is desirable to better understand when and if a given tensor product is decomposable into its independent variables.

In this work, we have shown how one can handle the basic operators ('+', '-', '*', '$\bullet$'). The interval bound for the inner product, $\bullet$, operator is not as tight as in tensor

products and a possible tighter interval bound should be sought. Other operators, such as the cross product, are not support and might be useful as well. Interestingly enough, the support of non algebraic operators, such as a square root or the sine function, is also feasible under ETs and worth further investigation. This extension to non-algebraic constraints opens a whole new horizon, going into a completely new uncharted domain.

During the solution search of the subdivision solver, several higher end algorithms are applied at times, such as gradient estimation [8], verification of a single solution existence in a domain [8], or quadratic clipping [2]. The expansion of the ETs representation to handle such higher level algorithms, if possible, should be investigated as well.

Finally, it worth pointing out that the traditional subdivision based surface-surface intersection (SSI) algorithms operate much like expression trees. These algorithms converges to the intersection locations by *independently* dividing along the parameter space of one freeform surface or the other. At no time, one process an explicit 4-variate tensor product representation of the constraint. Instead, the constraint is implicitly evaluated for possible intersections, much like the ETs scheme. A subdivision of one surface does not affect the other and vice versa. In summary, the ETs scheme is a generalized framework of handling non linear constraints with a complexity that captures the complexity of traditional subdivision based SSI algorithms.

Acknowledgment

This research was supported in part by the Software Technology Center of Excellence, Technion, in part by European FP6 NoE grant 506766 (AIM@SHAPE), and in part by the Israel Science Foundation (grant No. 346/07).

References

[1] A. V. Aho, R. Sethi, J. D. U. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Menlo Park, CA, 1986.

[2] Bartoň, M., and Jüttler, B. Computing roots of polynomials by quadratic clipping. *Comput. Aided Geom. Des. 24*, 3 (2007), 125–141.

[3] Bartoň, M., Jüttler, B., and Moore, B. Polynomial solvers with superquadratic convergence. The tenth SIAM Conference on Geometric Design & Computing, 2007.

[4] E. Cohen, R. F. Riesenfeld, G. E. *Geometric Modeling with Splines*. A. K. Peters, New York, 2001.

[5] Elber, G., and Grandine, T. Hausdorff and minimal distances between parametric freeforms in $\mathbb{R}^2$ and $\mathbb{R}^3$. Submitted for publication in Geometric Modeling and Processing 2008.

[6] Elber, G., Grandine, T., and Kim, M. S. Surface self-intersection computation via algebraic decomposition. Submitted for publication in Solid and Physical Modeling 2008.

[7] Elber, G., and Kim, M.-S. Geometric constraint solver using multivariate rational spline functions. In *Proceedings of the Symposium on Solid Modeling and Applications 2001* (Ann Arbor, Michigan, 2001), pp. 1–10.

[8] Hanniel, I., and Elber, G. Subdivision termination criteria in subdivision multivariate solvers. *Computer Aided Design 39* (2007), 369–378.

[9] Lane, J., and Riesenfeld, R. Bounds on a polynomial. *BIT 21* (1981), 112–117.

[10] Mourrain, B., and Pavone, J. P. Subdivision methods for solving polynomial equations. Tech. Rep. RR-5658, INRIA Sophia-Antipolis, http://hal.inria.fr/inria-00070350/en/, 2006.

[11] Nishita, T., Sederberg, T. W., and Kakimoto, M. Ray tracing trimmed rational surface patches. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1990), ACM, pp. 337–345.

[12] Reuter, M. Subdivision multivariate solver in the barycentric bernstein basis. The tenth SIAM Conference on Geometric Design & Computing, 2007.

[13] Sherbrooke, E. C., and Patrikalakis, N. M. Computation of the solutions of nonlinear polynomial systems. *Computer Aided Geometric Design 10*, 5 (1993), 279–405.