The design of LINETOOL, a geometric editor

Lars Warren Ericson

Projet FORMEL
INRIA
BP 105-78153
Domaine de Voluceau-Rocquencourt
Le Chesnay CEDEX FRANCE

Chee-Keng Yap

Computer Science Department
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012 USA

Abstract

We describe the design of LINETOOL, a geometric editor. Researchers in the areas of computational geometry, robotics and algebraic computation need a graphical editor for composing geometric objects which does more than simply turn pixels on and off on the screen. This system will be a tool to help researchers make and demolish conjectures, and to experiment with ideas. Our editor will allow users to define geometric scenes by declaring geometric objects built up from constants, dependent and independent variables, and geometric constraints. The system will solve for the constraints, and display the resulting scene. The user may then make queries about spatial relationships between components of geometric objects in the scene, which will be answered correctly, that is, without errors due to numerical approximations.

1 Introduction

We have been working on a graphical editing system that we hope will have an impact in computational geometry. The target audience for this system are researchers in computational geometry, robotics and algebraic computation, who often would like tools to help them visualize constructions in proofs, examples and conjectures. In this kind of research, there is a need for a graphical editor to compose geometric objects (*not* just turning pixels on/off on the screen) which satisfy specific constraints. In other words, these are abstract objects that have existence independent of their visual representation. One will be able to vary the parameters of the objects and see how they change. The system will maintain user-specified relationships among the objects (e.g. this line is always at distance 2 from the intersection of that pair of lines). The user may request comparisons of object component values (e.g. is this point above that line?). Comparisons shall be *exact*, not limited by the resolution of the screen. Objects shall be represented exactly internally, with the visual representation (at user specified resolutions) being derived from their internal representation.

As Forrest has pointed out [20], the numerical instability of geometric algorithms is a well-known problem, with the stability of solutions depending on such artifacts as the nearness of the scene being processed to the origin of the coordinate system, and proposed solutions ranging from lazy evaluation, to fixed increases in numerical precision and clever attention to the order of expression evaluation, to systems of geometric operations which keep a measure of the uncertainty of the result. However, it seems that there is only one sure solution to such problems: the use of algebraic computations founded on *exact precision arithmetic.*[1] Our system is an attempt at an 'existence proof' of the feasibility of such an approach and to bring to relief the appropriate algebraic tools. Perhaps the algebraic tools are self-evident once the exact precision goal is clearly delineated, but to our knowledge the set of tools we propose has not been used in any extant system.

Our exact approach solves, as a corollary, the problem of maintaining topological correctness [19]. Note that the papers Yao and Greene [44], Segal and Sequin [35] and Milenkovic [26] have topological correctness (a.k.a. consistent calculations) as their goal; the fundamental assumption in these papers is that of fixed precision arithmetic. Of course, our

Ericson is supported by a Bourse Chateaubriand from the Gouvernement Français. Yap is supported in part by NSF grants #DCR-84-01898 and #CCR-8703458.

[1] We choose this terminology over the alternative of 'infinite precision arithmetic' which may be misleading because our number representation remains finite precision at every step of the computation. Perhaps 'arbitrary precision arithmetic' is acceptable but we wish to emphasize that it is not really the precision that matters (since infinite precision does not imply no error) but the fact that no loss of information occurs throughout the intermediate computation.

approach is fundamentally different. However, their assumption is very important in the context of computer graphics; moreover this seems to be a necessary assumption if real-time computation is a goal.

It seems evident that in general, if we seek exact precision arithmetic then we must give up the hope of real-time operation. This is a situation that can be tolerated in the kinds of application we envision: as a tool for geometers to construct exact geometric objects for the purposes of testing hypotheses. The sketches that computational geometers often perform on paper in trying out ideas can often be done *exactly* using such a system. Several hours of background computation on the machine is often preferable to a fraction of the computational geometer's time doing the same construction imprecisely. Just as human sketches often evolve, we will see that our algebraic tools allow an evolving and interactive use of the system: each construction need not be a 'one-shot' computation, but contains intermediate results that can speed up incremental modifications in a construction. For other potential uses for such a system, see [2].

Prior graphical editing systems in which scenes are described by geometric constraints include Nelson's JUNO graphical editor [29], Borning's THINGLAB simulation system [4], and Sutherland's SKETCHPAD graphical editor [39]. These systems all fall back on numerical relaxation methods [15,36] to solve non-linear systems of equations which arise from the geometric constraints of the given scene. Borning's system is also in the line of work in Artificial Intelligence on programming languages based on the propagation and retraction of constraints on domains of discrete values, through the nodes of constraint relationship graphs. Such work is exemplified by Steele's thesis [37]. Gosling's MAGRITTE geometric editor [22] and van Wyk's IDEAL graphical typesetting language [42], on the other hand, use algebraic methods such as iterative elimination (IDEAL) and term-rewriting (MAGRITTE).

A problem domain slightly tangential to our intended application is that of geometric modelling. Although there are numerous efforts here, we mention one that seems closest to ours especially in its deployment of algebraic techniques. A 'geometric algebra system', using symbolic algebra techniques to *reduce* the cost (compared to ray casting and numerical techniques) of constructing geometric models built up from parameterized surfaces, is being designed at the University of Bath [5]. The algebraic techniques employed in the Bath system include root isolation via Sturm sequences, algebraic intersection of lines with surfaces, surface intersection via elimination via resultants, and 'generic rays', a technique analogous to the Collins cell decomposition method [14] used by Arnon to compute surface intersection [3].

Yet another related problem domain is that of automatic geometry theorem provers (e.g. [12]). Without elaboration, one might say that our system lies somewhere halfway between such geometry provers and the geometric modellers.

2 Steps in the geometric editing process

The geometric editor we are implementing works as follows:

1. The user declares *independent* $\bar{U} = u_1, u_2 \ldots u_m$ and *dependent* variables $\bar{X} = x_1, x_2 \ldots x_n$. This declares our intention to work in the polynomial ring $\mathcal{Q}(\bar{U})[\bar{X}]$. with rational function coefficients.

2. Primitive geometric objects, such as points, lines, half lines and line segments, and circles and circular arcs can be declared. The objects may have painting attributes such as solid or dashed, labels, etc. These objects are parameterized by the variables $\bar{U}\bar{X}$ and may involve numerical constants.

3. Geometric relationships between objects can be declared. Although convenient short hand is provided, these are ultimately translated into systems of equations in $\mathcal{Q}(\bar{U})[\bar{X}]$. We call these *constraints*.

4. *Kinds* (parameterized collections of geometric objects and relations) are defined by the user. A kind can be *instantiated* (by substituting for parameters). Thus it provides a macro facility.

5. The dependent variables may be partitioned into groups such that a *dependency graph* can be specified. The graph is constructed by specifying its edges (i.e. the dependency of one group of variables upon another). There are restrictions on this graph (for instance, it must be acyclic).

6. The primitive objects, instantiated kinds, constraints and dependency graph are collected in *frames*. A frame can be thought of as a sheet of paper on which a geometric construction is being drawn. Provided the frame is properly defined, it may be displayed and relationships among objects in the frame may be queried. The user may save a frame in a file or generate a set of POSTSCRIPT and LaTeX files which draw a picture of the currently-displayed scene.

7. Let us now see what happens behind the scenes during a define-display-query cycle above. Note that the order of definitions leaves a considerable amount of freedom. There is the obvious bookkeeping task of keeping track of variables, constraints, objects, kinds and dependencies. In the definition of a kind, we must verify that the system of constraint equations within a kind is consistent. This amounts to computing the Gröbner basis of the equations. In the case of a frame, we verify that the dependency graph satisfies some requirements to be described later. Then, based on this graph, we construct a Gröbner basis. Inconsistent constraints will be detected here. If the Gröbner basis has dimension greater than 0 or has no solutions, we complain to the

user that there are insufficient constraints. (More generally, we may analyze the dimension of the constraint system.)

8. An *assignment procedure* is constructed which for any assignment of values to the independent variables yields corresponding values for the dependent variables. Using this procedure, the user can rapidly vary the independent variables and change the display. Graphical tools (such as sliding scales or mouse) can help in specifying the values of the independent variables. The user can also scale the display to any desired degree of fine detail.

9. The user can query the display on the relationships among the objects. The answers are always correct, since the computations are based on exact algebraic number arithmetic[25].

10. Based on the display and query answers, the user can repeat the above process of defining frames, displaying and querying. The user may add and delete constraints, objects and painting properties. The system may be able to exploit intermediate results to speed up such incremental computations.

3 Editor language

A version of LINETOOL was implemented as a sublanguage of Yale's T dialect of Lisp [32]. A subsequent version in progress will be based on the MAPLE computer algebra system of the University of Waterloo [45]. The sections below give an impression of the editing language.

3.1 Dependent and independent variables

Independent variables may be declared directly, for example by saying

```
independent x1, x2, x3;
```

We assume that variables are dependent if they are not independent. As a pragmatic matter, it seems more natural for a user to begin by constructing objects (primitive objects or instantiated kinds) before deciding on the set of independent variables. LINETOOL only insists that all declarations be completed before the frames are analyzed.

3.2 Dependency declarations

Among the dependent variables we allow the user to declare a more elaborate structure, an acyclic directed graph whose nodes represent groups of variables. The user may say

```
{X, Y, Z} depends-on {A, B};
```

to mean that set of variables $\{x, y, z\}$ depends on the set of variables $\{a, b\}$. Of course we can specify any number of variables in these sets. The totality of such declarations must satisfy certain requirements to be specified in the next section.

These dependency relations are to be regarded as 'advice' for the Gröbner basis algorithm: using them we expect the algorithm to exploit certain structure in the system of equations to be solved. Using the dependency information, the system of equations can be solved in parts. For instance, if the above declaration were the only declaration, then intuitively, we can separately solve those equations involving only the variables a, b. The burden of providing dependency information is on the user. But usually the user can provide such information since the user knows the *semantics* of the variables.

3.3 Primitive Objects

The primitive geometric objects of the language are numbers, angles, points, lines, circular arcs and circles. (We avoid more general algebraic curves in the present design.)

Numbers are either arbitrary-precision integers, rationals, or real algebraic numbers defined using isolation intervals on the roots of polynomials with integer coefficients. For instance,

```
algebraic-number(x^2 - 2, [1,2]);
```

gives the root of $x^2 - 2$ in the interval $(1, 2)$, i.e. the positive squareroot of 2.

Angles are algebraic as well: an *algebraic angle* is one whose sine is a real algebraic number. An angle can be specified in (only) one of the forms Φx where Φ is one of arcsine, arccosine or arctangent and x is a rational or algebraic number. For instance, to get 30°, we write

```
algebraic-angle(Arcsine,1/2);
```

For 210° we write

```
algebraic-angle(Arcsine,1/2,reflex);
```

where the keyword `reflex` denotes that we want the non-principal quadrant. If one wishes to compare two algebraic angles, then we can compare them to any degree of accuracy. In analogy to algebraic numbers, we store an interval to bound the angle. This interval can be refined as much as desired.

Lines are subdivided into line segments, half lines and full lines. Each line also has an optional *paint* property, such as `solid` (the default) or `dashed` or `invisible`. For example,

```
add point: A(0,0);
add point: C(x2,x3);
add line-segment: AC(A,C);
add point D(x5,x4);
add line-segment AD(A,D,longdash);
```

A circle is defined by its center and radius, and a circular arc is specified as a circle together with its 'left' and 'right' endpoints (represented as angles).

3.4 Constraints

Geometric constraints establish relationships such as that a point is on a line, that a point is at the intersection of two lines, that two lines are perpendicular, etc. It is well known that these can be expressed as polynomial equations. (Schwartz [33, §3] and Chou [12] give some common constraints, together with their algebraic translations.) In general, a *constraint* is simply a polynomial equation or inequality or inequation.

For example, the constraint that $p = (x_p, y_p)$ is the midpoint between $q = (x_q, y_q)$ and $r = (x_r, y_r)$ is expressible as the single vector equation $q + r = 2p$, or as the system of equations $\{x_q + x_r = 2x_p, y_q + y_r = 2y_p\}$. Many standard constraints are given convenient forms in LINETOOL, for example:

```
# DC and AB are line segments
add relation R1: Parallel(DC,AB);

# A, O and C are points
add relation R2: Collinear(A,O,C);

# P is a point and L a line
add relation: Incident(P,L);
```

Note that we can name a constraint (`R1, R2`) or let it be anonymous. In general, the user may freely specify constraints as equations, inequations and inequalities:

```
add relation P1: X*Y = U*V;
add relation P2: X*Y /= 0;
add relation P3: U >= 0;
add relation:  V^3 - 4 < 0;
```

3.5 Primitive functions

Another feature is the use of primitive functions. For instance, if `P1`, `P1` are points, rather than writing the expression

```
(P1.X - P2.X)^ + (P1.Y - P2.Y)^2
```

we write `Distance(P1,P2)`. The function is extended to describe the distance between a point and a line for instance: `Distance(P1,L)`.

Another useful function gives the intersection between two lines `Intersect(L1,L2)` (if the lines are coincident, the result is the line itself, and if they are only parallel, the result is a point at infinity).

Such expressions may be composed and used in constraint relations. Thus the following constraint says that two lines intersect at unit distance from the origin:

```
add relation: Distance((0,0),Intersect(L1,L2)) = 1;
```

We can define an algebraic angle with `Angle(P,Q,R)` where `P`, `Q` and `R` are points (with algebraic coordinates).

Another function which returns an angle is `slope(L)` where `L` is a line or line segment.

3.6 Kinds and Frames

A *kind* is a composite object with (optionally):

- External parameters given by a `parameters` clause.
- Geometric objects within the object given by an `locals` clause.
- Geometric constraints between objects, parameters and constants given by a `relations` clause.
- Inheritance of attributes from another kind given by an `of` clause. The `of` clauses gives the name of another kind, followed by actual parameters from the environment of the kind being defined. When an instance of kind A is created, effectively, an instance of the kind B that it is `of` is also created, and any parameters, locals or relations of the instance of kind B are accessible and utilized as if they appeared directly in the declaration of kind A. This notion of inheritance is related to the notion as developed in 'object-oriented programming' [38].

For example, the following kind declarations define triangles and equilateral triangles, respectively:

```
(kind Triangle
   (parameters point: A, B, C)
   (local line-segment: AB(A, B), BC(B,C), AC(A,C)))

(kind EquilateralTriangle
      (of Triangle(A,B,C))
      (parameters point: A, B, C)

      (relations Length(AB) = Length(BC);
                 Length(AC) = Length(BC)))
```

In order to use a kind, we *instantiate* it. This is done by invoking the name of the kind and supplying values for the external parameters. Thus:

```
add Triangle: T1(P1,P2,(0,0));
```

yields an instance of the triangle with vertices at `P1`, `P2` and the origin $(0,0)$. We may now refer to the vertices of the triangle `T1` using the dot convention, thus the three vertices of this triangle instance are `T1.A`, `T1.B`, `T1.C`. (The coordinates of `T1.A` are `T1.A.X`, `T1.A.Y`, etc.)

A *frame* is a collection of primitive objects, instantiated kinds, independent variable declarations, dependency constraints, and relations (see §3.4), together with a coordinate transformation matrix as defined in POSTSCRIPT [1]. The user creates or enters a frame by saying `open-frame F1`. From then until the user says `close-frame`, all declarations of primitive objects, etc, will be put into the frame named `F1`. Note that the declaration of kinds are global and do not get put into the currently opened frame. When a frame is closed, it can be checked and its equations solved by invoking `analyze F1`. This process is explained in more detail below. This is the most sophisticated part of the engine and errors in the definition of the frame will be detected during

this solution process. In particular, if the frame is non-zero dimensional, i.e., it is unsolvable or has infinitely many solutions, this will be reported and the user must edit the frame, that is, add and delete relations and objects, etc.

Assuming the frame is successfully analyzed, the user may display the frame by saying `display F1`. Recall that the frame has in general several independent variables. The user will be provided with several *sliders*, one for each independent variable, on the display screen. Using a mouse or the keyboard, the user is able to freely vary these independent variables and the display will transform accordingly.

Recall too that a frame has a coordinate transformation, which is initialized to the identity matrix. This transformation may be changed freely during display. In particular, the user may zoom into detailed parts of the figure by scaling up and changing the focus (center point) of the display. This scaling up is in principal unlimited.

Finally, the user may query object relationships in frame `F1` by saying `query F1:` followed by any valid relation, for example:

```
# For point P and line L in F1; Answer is Y/N
query F1: on(P,L);

# Lines are always directed
query F1: right-of(P,L);
```

There is also a command `list-frame F1` for listing the objects and relations of a frame and information about its solution. The user may inspect the current value for the components of an object in the current frame by simply giving the name of the object.

4 Dependency

In this section we outline our method for treating dependent and independent variables. We believe their correct treatment would be critical to the practicality of LINETOOL, especially in efficiency. This is because non-trivial geometric constructions easily involve over 10 variables, which is normally beyond the scope of current algorithms for solving such systems (whether one uses Gröbner basis or other techniques). On the other hand, one can exploit structural properties in systems of equations arising from geometric constructions. These properties are little understood and hence their exploitation requires user cooperation. We provide a mechanism by which the user can advise our algorithm.

Recall that among the (number and angle) variables in a frame, we classify certain of them as independent. These are meant to be the freely varying variables.[2] It is the users' responsibility to ensure that their independent variables are meaningful since our system is not equipped to detect such irregularities. For instance, it is clear that there must not be any constraints involving only independent variables. While it is easy to check for this (syntactic) restriction, it is not a sufficient test. In illustration, if u, v are independent and x is dependent, the three constraints

$$\begin{cases} (u-x)^2+v^2 &= 0 \\ (u+x)^2+v^2 &= 0 \\ x &> 0 \end{cases}$$

together imply u must be identically zero. On the other hand, as pointed out in [24], there is a simple method based on Gröbner bases to check if a set of variables is truly independent.

Among the dependent variables, the semantics of a geometric construction can often tell us that one group of variables depends upon another group. To illustrate this, suppose that we want to construct the three squares on the sides of a right triangle T, as in a well-known proof of Pythagoras' theorem. Suppose the vertices of T are $(0,0)$, $(0,x)$ and $(y,0)$, and the other two vertices of the square on the hypotenuse are $(a,b),(c,d)$. Then it is natural to let the independent variables be x,y and declare that a,b depends on x,y and also c,d depends on x,y.

Each dependency declaration (see §3.1) is a relationship between two groups of dependent variables. We impose two requirements on these declarations:

1. The dependency declarations are used to decompose the set of dependent variables into groups. To do this, we form a set S of subsets of dependent variables: for each dependency declaration, we get two sets of variables (one depending on the other). Both of these sets are put into S. All those dependent variables that are not mentioned in any dependency declaration also form a set which is again put in S. Now we repeatedly coalesce each pair of sets in S that have some common element. The final set S forms a partition of the set of dependent variables; members of this partition are called *groups* of dependent variables. We construct a directed graph G whose nodes are labelled by these groups of dependent variables. If U, V are two groups of variables, then the edge (U,V) is in G if and only if there exists some dependency declaration which says some subset of U is dependent on some subset of V. We require the graph G to be acyclic; G is called the *dependency graph.*

2. Let E be the set of equations that occur as constraints in the frame. (For the time being, let us ignore inequations and inequalities.) For each node (associated with a group of dependent variables U), we associate a subset $E(U)$ of E as follows: $E(U)$ consists of those equations in E which involve one or more variables in U. For each equation $e \in E(U)$ we require that any variable in e must either be independent or be in some group V that is reachable by a path from U.

[2] Actually, it is sufficient that each of these number (resp. angle) variables can freely vary over some interval (resp. angular range), simultaneously. A more sophisticated system might allow this refinement.

5 Translation from geometric to algebraic terms

As noted in §3.4, geometric constraints may ultimately be translated into polynomial relations of the form $P \circ 0$, where $\circ \in \{=, \neq, <, >, \leq, \geq\}$, and $P \in \mathcal{Q}(\bar{U})[\bar{X}]$. The manipulations to get a constraint into this form are standard. Note that the original constraint may involve primitive functions such as the distance between two points or between a point and a line. These expressions involve the radical extraction which can be removed by repeated squaring, etc.

In principle, the treatment of inequations and inequalities can be reduced to equations. For each such constraint, we introduce a new (dependent) variable x_0 and write a corresponding equation:

- We reduce the inequality $P \leq 0$ to $P + x_0^2 = 0$.
- We reduce the strict inequality $P < 0$ to $x_0^2 P + 1 = 0$.
- We reduce the inequation $P \neq 0$ to $x_0^2 P^2 - 1 = 0$.

Despite its conceptual simplicity, the introduction of new variables x_0 seems to be a bad idea. This is because all known algorithms for solving systems of equations have complexity that grows exponentially in the number of variables. One heuristic for reducing the complexity is to specify the dependency: x_0 depends on the set of all dependent variables occurring in P. Another method of dealing with these inequation/inequalities will be discussed later.

It is well known that in general, the transcendental functions lead to undecidability for such basic questions as root isolation and elimination of variables (see Buchberger [10, §5]). To get a decidable subset, we restrict ourselves to the algebraic angles and only allow terms involving angles which

- Add two angle terms.
- Take an integer multiple of an angle term.
- Apply a trigonometric function to an angle term.

Thus, we allow the expression $\tan(5\theta - 12\phi)$ where θ, ϕ are algebraic angles.

Note that a set of geometric constraints is thus translated into a system of equations $C_i = 0$ on dependent variables $\bar{x}$ and independent variables $\bar{u}$, which represents the logical assertion

$$(\forall \bar{u})(\exists \bar{x}) \wedge_i C_i = 0$$

Such assertions (involving only conjuncts in the matrix of the sentence) are sufficient to declare a wide variety of planar geometric scenes. However, there may be situations that call for disjunction or negation.

6 Gröbner bases and related questions

The system of equations Σ into which a set of constraints is translated generates a polynomial ideal I. The Gröbner basis of Σ (with respect to the pure lexicographic ordering) is a basis for I that is well-suited to elimination [7]. Our basic computational tool will be such a Gröbner basis of Σ. Although the fine-tuning of the algorithm is a continuing matter of experimentation, we will use a number of ideas to speed up the so-called Basic algorithm[27]:

1. Buchberger's Improved algorithm [7].
2. The use of head-reduction (this means that the normal form algorithm does not attempt to further reduce a polynomial whose head term is irreducible). For this purpose, we have developed some efficient data-structures that make this particularly effective (see below).
3. The polynomials in the Gröbner basis will be generated in order of non-decreasing (total) degree to ensure a minimum degree Gröbner basis [21].
4. Exploiting the dependency structure (see below).
5. Exploiting the fact that many geometric constructions may have simple identification of variables (i.e. constraints of the form $x = y$). We can use a union-find data structure to keep track of equivalent variables.
6. More generally, it seems that many geometric constructions involve only linear equations. In this case, Gaussian elimination techniques suffice.

Hierarchical construction of Gröbner basis. The dependency structure is exploited in the following manner. We process the nodes of the dependency graph in a topologically sorted fashion, beginning at those minimal nodes (i.e., nodes with outdegree zero). A node of the graph is identified with the group U of dependent variables associated with it. Let $D(U)$ denote the set of all variables in nodes V that can be reached with zero or more edges starting from U. Inductively assume that at each node U that has been processed, we have already computed the Gröbner basis $G(U)$ of all equations that involve at least one equation in $D(U)$. Suppose we are now processing a node V and let $V_1, \ldots, V_m$ be the nodes which are reachable from U along a single edge. (We may omit any V_i that is reachable from some other V_j). Then we may compute the Gröbner basis of the set of equations $E(U) \cup G(V_1) \cup \cdots \cup G(V_m)$.

In computing the Gröbner basis of $E(U) \cup G(V_1) \cup \cdots \cup G(V_m)$, we exploit the fact that the $G(V_i)$'s are already Gröbner bases. For instance, we do not have to compute the S-polynomial of pairs of polynomials in each $G(V_i)$. We will refer to this as the *hierarchical method* for computing the Gröbner basis.

Data structures.

We are designing data structures for various operations related to Gröbner bases computation. One particularly useful one is after the Gröbner basis G is constructed, subsequent reductions of polynomials modulo G can be rapidly carried out using the following data structure D: given a power product p, we can decide if p is divisible by some head power product of a polynomial in G. For instance, with n variables and $|G| = m$, we can achieve a space-time bound of $O(m^{n-1}, n \log m)$ or $O(2^n m, \log^{n-1} m)$.

Dimensionality analysis. The *dimension* of a system of equations is the dimension of the algebraic variety of the polynomials in system. An important requirement of LINETOOL is that the system of constraint equations must be zero dimensional, i.e., determine only a finite number of solutions to the system.[3] Note the independent variables, already transferred to the coefficient domain, do not participate in dimensionality analysis. It seems desirable to inform a user of the dimensionality of the system of constraints; in our current implementation, we only give one of three answers: zero dimensional, unsolvable (-1 dimensional), or positive dimensional (infinitely many solutions). Schwartz [33, §2] gives a probabilistic implementation of a method of van der Waerden for determining dimensionality [41]. Alternatively, the dimension can be obtained from the Hilbert function of the ideal of the system of equations; the Gröbner bases can be used to compute the Hilbert function [28].

Complexity. We should remark that the worst case complexity of computing Gröbner basis is double exponential. However it is now known (following recent work of Brownawell and Caniglia [6,11]) that solving a system of polynomial equations that has a finite set of solutions (as in our case) can be done in single exponential time using the Gröbner bases method. Such a complexity bound matches the bounds (due to John Canny of U.C. Berkeley) recently obtained for certain classical resultant methods. Hence it seems that the Gröbner bases method is just as competitive. Moreover, the Gröbner bases approach seems to be more versatile (applicable to many related computational problems, as well as amenable to incremental modification of the generator set).

7 Elimination and display

We now discuss the computational work necessary for obtaining a solution to a system of equations from its Gröbner basis $G \subseteq \mathcal{Q}(\bar{U})[\bar{X}]$. In order to do elimination of variables, it is simplest to use the pure lexicographical ordering of power products in the Gröbner basis algorithm [7,27]. This is in fact what we assume. The Gröbner basis G is then a system of polynomials in triangular form which permits one to find some or all (as the user may specify) solutions to the system via successive root isolations.

[3]Really we care about real zeroes, but it is useful enough to know about the complex zeroes, which is what the theory gives us.

If there are no independent variables ($\bar{U}$ is empty) then the solution of the triangular form G proceeds in the usual manner. In general, we must solve for G for arbitrary values of the independent variables. In one mode of operation, the user will successively specify values for $\bar{U}$ (using the sliders provided for each independent variable). After substituting for these values (which we restrict to be rational), we get a new system $G' \subseteq \mathcal{Q}[\bar{X}]$. Unfortunately, this system need no longer be triangular. There are two ways to handle this: we can either call the Gröbner basis algorithm on G' again (and hope that G' is almost triangular so that the result is rapidly obtained) or we can simply ignore this system G'. The latter needs explanation: the likelihood of G' not being triangular has geometric probability zero. So we can either randomly perturb the chosen values of $\bar{U}$ to regain a triangular form or we can request that the user pick another set of substitutions.

Previously we alluded to an alternative method of treating polynomial inequations/inequalities. Basically, we do not include such constraints directly into the system of polynomial equations to be solved for in the Gröbner basis. Instead, these are set aside until the stage when we find actual solutions of the system of equations: then as solutions are generated, each is tested to see if they satisfy the polynomial inequations/inequalities. We thus need to evaluate polynomials at algebraic arguments and compare algebraic numbers with zero; both of these can be done with known algorithms.

One important special case which we would like to take advantage of is when the system turns out to be linear in the $\bar{x}$; then standard Gaussian elimination techniques suffice. In this case, we can 'precompile' an *assignment procedure* which rapidly computes an assignment for $\bar{x}$ given any value for $\bar{u}$. In experiments, we have also found that the Gröbner basis algorithm converges very quickly in this case, usually just a few iterations.

Even if the system is non-linear, full exploitation of the subset of linear constraints seems preferable.

8 Evaluating object queries

The availability of a Gröbner basis G can be exploited here. Many geometric queries can be expressed as asking if a polynomial P evaluates to 0. A sufficient (but not necessary) condition for satisfaction of this query is that P belongs to the ideal of G, that is, P reduces to 0 modulo G (cf. [24]). In general, we would have to compute a basis for the radical of the ideal.

Queries of the form $P > 0$ requires root separation and algebraic number comparison methods [25,34].

9 Example

The Connecticut Do Nothing Machine is a small toy that can be purchased at highway gift shops in Connecticut. It is a square wooden block, on one surface of which has been drilled two open wooden tracks which are perpendicular to and pass through each other. Inserted in these tracks are

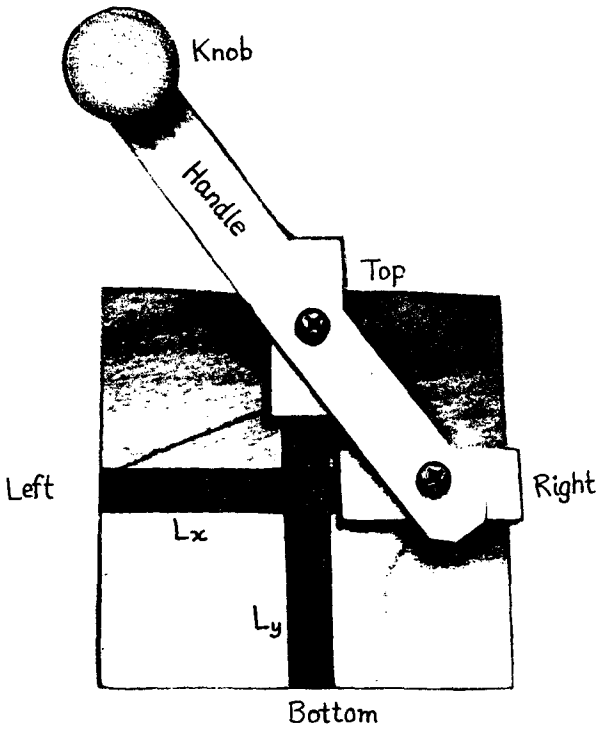

Figure 1: The Connecticut Do Nothing Machine

bevelled wooden rod segments, one for each track. In the midpoint of each of the two rod segments is a screw. Via these two screws, a straight bar is attached, with one screw at one end and in the middle, and a knob at the other end. When the bar is pushed in the plane by pressure on the knob end, the bar rotates and the knob traces an ellipse. (See Figure 1.)

This machine may be described in LINETOOL as follows. The basic block has four edge midpoints, `Top`, `Bottom`, `Left` and `Right`, and two tracks, `Lx` and `Ly`, represented by line segments. As parameters, it also takes a length for the sliders and for the handle. The sliders and handle are attached by passing common reference points for the attachment:

```
# Connecticut Do-Nothing Machine

(kind CDNM
   (parameters
      point: Top, Bottom, Left, Right;
      number: SliderLength, HandleLength)

   (local
      point: VSAttach, HSAttach;
      line-segment: Lx(Left, Right), Ly(Top, Bottom);
      slider: VSlider(SliderLength, VSAttach, Ly),
              HSlider(SliderLength, HSAttach, Lx);
      handle: Handle(HandleLength, VSAttach, HSAttach)))
```

A slider has a certain width and center, and slides along the track defined by a given line segment. Locally it defines two endpoints for itself, and it attaches its center to the track by a relation.

```
(kind slider
   (parameters
      number: SWidth;
      point: Center;
      line-segment: Track)

   (local
      point: Left, Right)

   (relations
      On(Center, Track);
      Distance(Left, Center) = SWidth/2;
      Distance(Right, Center) = SWidth/2;
      Collinear(Center, Left, Right)))
```

A handle has a width and two attachment points and a knob. The knob is collinear with the attachment points.

```
(kind Handle
   (parameters
      number: HWidth;
      point: AttachV, AttachH)
   (local
      point: Knob;
      line-segment: Bar(AttachH, Knob))
   (relations
      Collinear(Knob, AttachV, AttachH);
      Distance(Knob, AttachV) = HWidth;
      Distance(Knob, AttachH) = HWidth/2))
```

We create an instance of the machine and place it in a frame as follows:

```
open-frame F1;
add CDNM: X((0,2),(0,0),(-1,1),(1,1), 2, 4);
```

It turns out that the knob traces out an ellipse. It is clear in any case that the motion of the knob is not entirely free. But we would like to experiment, to move the handle. So we add an angle to the frame, and relate it to the slope of the bar. Then we close the frame, analyze it, and display (the display is automatically parameterized by the only free variable in the frame):

```
add angle: t = slope(X.Handle.Bar);
independent t;
close-frame F1;
analyze F1;
display F1;
```

10 Remaining and future work

In T, we have written code for rings (polynomial, extension field and rational function); polynomials (sparse and power-product form), polynomial arithmetic, derivative, GCD, factoring by multiplicity and resultant calculations; rational functions; root isolation via Sturm sequences, isolation intervals, sign sequences and algebraic numbers; the Improved algorithm for Gröbner bases, substitutions and assignments, and lexical-order elimination; angles, points, lines, circles and frames; kinds with inheritance but without equational relationships; window graphics; and a rudimentary editor lan-

guage. Most of these facilities already exist in MAPLE, which is widely available, thus we have chosen to re-implement LINETOOL in the MAPLE system.

More code needs to be written for automatic domain interconversion; kinds with equational relationships; arbitrary admissible-order Gröbner basis elimination; algorithms for the hierarchical Gröbner basis computation; dimensionality analysis via Hilbert polynomials; faster algorithms for polynomial GCD, resultant and Sturm's series calculation (e.g. [34]); dimensionality analysis via probabilistic techniques such as those of Schwartz [33]; code to handle algebraic angles and related expressions; treatment of inequations/inequalities.

Given the geometric editor design that we have presented, it seems that it would be easy to extend it to handle non-Euclidean geometries such as the Lobachevsky [31] or Minkowski [43] geometries. One needs to provide:

- A slightly different set of geometric constraints with a (in certain cases) different translation into polynomial equations, e.g. a different metric for distance.
- Display routines with a different model of the plane, e.g. as a circle in which lines are diameters or arcs of orthogonal circles.

Acknowledgement

Lars Ericson thanks Thomas Dubé and Professor Bhūbaneswar Mishra of Courant Institute for stimulating conversations regarding methods in algebraic geometry, and Dennis Arnon of Xerox PARC for helpful comments.

References

[1] Adobe Systems Incorporated. POSTSCRIPT *Language Tutorial and Cookbook.* Addison-Wesley, 1985.

[2] Varol Akman. Geometry and graphics applied to robotics. In [18], pages 619–638.

[3] Dennis S. Arnon. Geometric reasoning with logic and algebra. In [23].

[4] Alan Borning. *Thinglab: A constraint-oriented simulation laboratory.* PhD Thesis, Stanford University. Also available as Technical Report STAN-CS-79-746, Computer Science Department, Stanford University, July, 1979.

[5] Adrian Bowyer, James Davenport, Philip Milne, Julian Padget and Andrew Wallis. A geometric algebra system. Draft, December 1987, Schools of Mechanical Engineering and Mathematical Sciences, University of Bath, Bath, Avon BA2 7AY, UK.

[6] D. Brownawell. Bounds for the degrees in the Nullstellensatz. *Annals of Math. Second Series*, **126** (3) 1987, pages 577–591.

[7] Bruno Buchberger. Gröbner bases: An algorithmic method in polynomial ideal theory. In N.K. Bose, editor, *Multidimensional Systems Theory*, pages 184–232. D. Reidel, 1985.

[8] Bruno Buchberger. What can Gröbner bases do for computational geometry and robotics? In [23].

[9] Bruno Buchberger, G. E. Collins, R. Loos, editors, with R. Albrecht. *Computer Algebra: Symbolic and Algebraic Computation* Springer-Verlag, second edition 1983.

[10] Bruno Buchberger and R. Loos. Algebraic simplification. In [9], pages 11–43.

[11] L. Caniglia, A. Galligo and J. Heintz. Some new effectivity bounds in computational geometry. Preliminary version, 1988. Working Group Noaï Fitchas, Instituto Argentino de Mathemática, Viamonte 1636, 1er cuerpo, 1er piso, (1055) Buenos Aires, Argentina.

[12] Shang-Ching Chou. Proving elementary geometry theorems using Wu's algorithm. *Contemporary Mathematics* **29**, pages 243–286. American Mathematical Society, 1984.

[13] George Collins. The calculation of multivariate polynomial resultants. *Journal of the Association for Computing Machinery* **18** (4), October 1971, pages 515-522.

[14] George Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. *Second GI Conference on Automata, Theory and Formal Languages*, pages 134–183, Springer-Verlag Lecture Notes in Computer Science 33, 1975.

[15] S. D. Conte and Carl de Boor. *Elementary Numerical Analysis.* McGraw-Hill, 1980.

[16] Thomas Dubé. A survey of methods for bounding Gröbner bases. NYU Computer Science Department, February 9, 1987. Courant Institute of Mathematical Sciences, 251 Mercer St., New York, NY 10012.

[17] Thomas Dubé, Bhubaneswar Mishra and Chee-Keng Yap. Admissible orderings and bounds on Gröbner normal form algorithm. NYU Computer Science Department Technical Report 258, Robotics Report 88, Courant Institute of Mathematical Sciences, New York, December, 1986.

[18] R.A. Earnshaw. *Theoretical Foundations of Computer Graphics and CAD.* NATO ASI Series Volume F40, Springer-Verlag, Berlin, 1988.

[19] Lars Warren Ericson. Topologically correct graphical editing in the plane. Robotics Research Technical Report No. 97, Computer Science Division, Courant Institute of Mathematical Sciences, New York University, January, 1987.

[20] A.R. Forrest. Geometric computing environments: some tentative thoughts. In [18], pages 185–197.

[21] M. Giusti. Some effectivity problems in polynomial ideal theory. *EUROSAM '84*, pages 159–171. Springer-Verlag Lecture Notes in Computer Science **174**, 1984.

[22] James Gosling. *Algebraic Constraints.* PhD Thesis, Carnegie-Mellon University. Also available as Technical Report CMU-CS-83-132, Computer Science Department, CMU, May, 1983.

[23] D. Kapur and J.L. Mundy, editors. *Geometric Reasoning*, a special issue of *Artificial Intelligence*, contents selected from a Workshop On Geometric Reasoning, Keble College, Oxford University, June 30 – July 3, 1986.

[24] B. Kutzler and S. Stifter. Automated geometry theorem proving using Buchberger's Algorithm. *SYMSAC '86*, Waterloo, Canada.

[25] R. Loos. Computing in algebraic extensions. In [9], pages 173–187.

[26] Victor Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. In [23].

[27] B. Mishra and C.K. Yap. Notes on Gröbner basis. Technical Report 257, Computer Science Department, New York University, September 1986.

[28] H.M. Möller and F. Mora. The computation of the Hilbert function. *EUROCAL '83.* Springer-Verlag Lecture Notes in Computer Science **162** (1983), pages 157–167.

[29] Greg Nelson. Juno, a constraint-based graphics system. *ACM SigGraph* 19(3):235–243, 1985.

[30] A.C. Norman. Computing in transcendental extensions. In [9], pages 169–172.

[31] A. V. Pogorelov. *Lectures on the Foundations of Geometry.* Noordhoff, 1966.

[32] Jonathan A. Rees, Norman I. Adams and James R. Meehan. *The T Manual.* Computer Science Department, Yale University, 1984.

[33] Jacob T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the Association for Computer Machinery* **27** (4), October 1980, pages 701–717.

[34] Jacob T. Schwartz and Micha Sharir. On the "Piano Movers" problem. II. General techniques for computing topological properties of real algebraic manifolds. *Advances in Applied Mathematics* **4**, 298–351. Academic Press, 1983.

[35] Mark Segal and Carlo H. Sequin. Consistent calculations for solids modeling. *ACM SigGraph*, pages 29–38, 1986.

[36] R. V. Southwell. *Relaxation Methods in Engineering Science.* Oxford University Press, 1940.

[37] Guy Lewis Steele Jr. *The definition and implementation of a computer programming language based on constraints.* PhD Thesis, Massachusetts Institute of Technology. Also available as Technical Report AI-TR-595, Artificial Intelligence Laboratory, MIT, August, 1980.

[38] Mark Stefik and Daniel G. Bobrow. Object-oriented programming: themes and variations. *The AI Magazine* ??:40–62, 1985?.

[39] Ivan Edward Sutherland. *Sketchpad: A Man-Machine Graphical Communication System.* PhD Thesis, Massachusetts Institute of Technology, January 1963.

[40] J.V. Uspensky. *Theory of Equations.* McGraw-Hill, 1948.

[41] B.L. van der Waerden. *Einführung in die Algebraische Geometrie.* Second edition, Springer-Verlag. New York, 1973.

[42] Christopher John Van Wyk. *A Language for Typesetting Graphics.* PhD thesis, Stanford University. Also available as Technical Report STAN-CS-80-803, Computer Science Department, Stanford University, June, 1980.

[43] I.M. Yaglom. *A Simple Non-Euclidean Geometry and its Physical Basis.* Heidelberg Science Library. Springer-Verlag, 1979.

[44] Francis Yao and Daniel H. Greene. Finite-resolution computational geometry. *27th Annual Symposium on Foundations of Computer Science*, October 27–29, 1986, IEEE, pages 143–152.

[45] Symbolic Computation Group, University of Waterloo. *Maple 4.0 Reference Manual.* Waterloo, Canada, 1985.