

Numerical Decomposition of Geometric Constraints

Sebti Foufou*
Lab. Le2i, UMR CNRS 5158
Université de Bourgogne
BP 47870
21078 Dijon cedex, France
Sebti.Foufou@u-bourgogne.fr

Dominique Michelucci
Lab. Le2i, UMR CNRS 5158
Université de Bourgogne
BP 47870
21078 Dijon cedex, France
Dominique.Michelucci@u-bourgogne.fr

Jean-Paul Jurzak
IMB, UMR CNRS 5584
Université de Bourgogne
B.P. 47870
21078 Dijon cedex, France
Jean-Paul.Jurzak@u-bourgogne.fr

Abstract

Geometric constraint solving is a key issue in CAD/CAM. Since Owen's seminal paper, solvers typically use graph based decomposition methods. However, these methods become difficult to implement in 3D and are misled by geometric theorems. We extend the Numerical Probabilistic Method (NPM), well known in rigidity theory, to more general kinds of constraints and show that NPM can also decompose a system into rigid subsystems. Classical NPM studies the structure of the Jacobian at a random (or generic) configuration. The variant we are proposing does not consider a random configuration, but a configuration similar to the unknown one. Similar means the configuration fulfills the same set of incidence constraints, such as collinearities and coplanarities. Jurzak's prover is used to find a similar configuration.

CR Categories: I.6.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems

Keywords: Geometric constraints, constraints decomposition and solving, Geometry provers

1 Introduction

Solving systems of geometric constraints becomes a key issue for geometric modelers in CAD/CAM [Brderlin and Roller 1998; Sutherland 1963; Olano and Brunet 1994; Mignotte and Stefanescu 1999]. A qualitative study, also called a planning step, is needed to detect errors (*e.g.* over-constrained parts) in the system of constraints, and to decompose well-constrained parts into smaller parts, easier and faster to solve [Lin et al. 1981]. A number of computer softwares are developed to make geometry teaching, and pen-based drawings easy on computers. Proposed tools even allows modifying pen-based inputs of geometry drawings while keeping valid initial spatial and geometric constraints. Cinderella is an excellent example of such tools [Kortenkamp 1999; Kortenkamp and Richter-Gebert 1998].

Until recently, geometric constraints were typically used for the dimensioning of parts. The users were engineers who had a good intuition of what was going on, were often able to solve by hand the problems they posed and they could help the solver. Nowadays, geometric constraints are increasingly used to model objects of increasing complexity, for instance, free form curves or surfaces and their blends, and not only the dimensioning of mechanical parts. Geometric constraints are used in various fields by engineers, design artists, etc., which leads to more complex geometric systems, involving hundreds of unknowns, to solve. These systems are usually very under-constrained. It often happens that the solver does not find a solution, and is unable to give any usable explanation (for instance the offending subsystem is too big). The user tries to modify the values of some parameters, but the problem does not disappear. This situation is very frustrating for users, who can not understand the error because of the size of the system, and can not fix the problem. An offending system with a dozen of equations can be sufficiently large to prevent any human to isolate the source of the problem. Whether the user is a mathematician, an engineer or a designer does not change the situation. Note that even well known methods in computer algebra like Grobner bases, Wu-Ritt and Chou triangulations can not detect these kinds of problems, because their cost is worse than exponential.

An a posteriori explanation of the problem is that the system, which seems globally under-constrained (it has more unknowns than equations), contains an over-constrained subsystem, non detectable by today's procedures that perform the qualitative study. Basically, these procedures check that there is no subsystem which involves more equations than unknowns. Thus the faulty over-constrained subsystem has more equations than unknowns. Our explanation is that the over-constraint is due to the fact that the system of geometric constraints contains a hypothesis, and the conclusion or the negation of the conclusion, of a geometric theorem. Qualitative study procedures can not detect such dependences (redundancy or contradiction) between equations.

Moreover, since the problem persists when the user of the solver changes the values of parameters (distances, angles), one may suspect a dependence due to a projective theorem. A projective theorem uses only incidences in its hypothesis and its conclusion, and is thus independent of values of angles and distances.

Two examples of projective theorems are:

1. Desargues' theorems in 2D and 3D: if two triangles abc and ABC are perspective (aA, bB, cC concur), then $ab \cap AB, bc \cap BC, ca \cap CA$ are collinear.
2. The hexamys theorem in 2D: an hexamys is an hexagon (possibly concave or self-intersecting) such that its three opposite sides cut in three collinear points; then all permutations of the vertices of an hexamys are hexamys. Hexamys is an abbreviation for Pascal mystical hexagram. The term is due to Raymond Pouzergues [Pouzergues 2002].

*Sebti Foufou is currently a guest researcher at the Manufacturing Systems Integration Division, National Institute of Standards and Technology, Gaithersburg, MD 20899-8263. Email: sfoufou@cme.nist.gov

This paper presents two new methods to detect such dependences. Although the proposed methods are still under experimentation, they appear promising for geometric constraint dependence detection. Their polynomial time complexity and the results on a set of examples (courtesy of A. Ortuzar [Serré et al. 2001] and C. Jermann [Jerermann et al. 2003]) make us optimistic about their use for real world problems.

The remainder of the paper is organized as follows. Section 2.1 presents the limitations of the graph based qualitative study methods. Section 2.2 presents the Numerical Probabilistic Method (NPM), a method well known in rigidity theory, where only geometric constraints are the distances between points. This method is extended here to more general geometric constraints. Section 2.3 shows that NPM can also decompose the system of constraints into rigid subsystems; as far as we know, this result is new, even in the restricted framework of rigidity theory. However, the NPM (as well as the graph based methods) still has limitations due to the fact that it studies the Jacobian of the system at a random (generic) configuration. Section 2.4 introduces a new variant of the NPM to overcome these limitations: the idea of this variant is to study with the NPM not a random configuration, but a configuration satisfying the incidence constraints (collinearities and coplanarities of the system). Section 3 presents Jurzak’s prover, which is often sufficient to compute such configurations. Section 4 describes why solvers should use Jurzak’s prover. Section 5 presents work in progress on how orthogonality constraints can be reduced to incidence constraints. Section 6 provides the conclusions.

2 The problem

The qualitative study is performed on systems of constraints before the system solving is undertaken in order to detect errors (*e.g.* over-constrained parts) and to decompose well-constrained parts into smaller parts, to speed up the resolution.

2.1 Limitations of graph based methods

Typically, qualitative studies performs a degree of freedom analysis in a graph of constraints [Hoffmann et al. 2001a; Hoffmann et al. 2001b; Gao and Zhang 2003; Lamure and Michelucci 1998; Joan-Arinyo et al. 2004; Joan-Arinyo et al. 2003], either: (a) the bipartite graph of incidence between equations and unknowns, where an arc between an equation-vertex and an unknown-vertex means that the corresponding equation involves the corresponding unknown; or (b) a graph where vertices represent geometric unknowns (*e.g.* points, lines) and edges represent distance or angle or incidence constraints. With the second kind of graph, to account for non-binary constraints either a hypergraph must be used, or non-binary constraints must be binarized during a preprocessing step [Owen 1991; Owen 1996; Sitharam and Zhou 2004; Hoffmann et al. 2004].

Laman’s theorem [Laman 1970], which applies in 2D to distance constraints only, is extended to 3D and to more general geometric constraints since Owen’s seminal paper [Owen 1991]. Laman’s theorem states that a set of 2D points, with generic distance constraints between them, is well-constrained (rigid) iff there are $e = 2n - 3$ distance constraints and n vertices, and iff for all subsystems where e' constraints involve n' vertices, $e' \leq 2n' - 3$. This last condition means that no subsystem is over-constrained. The genericity condition is here to forbid points collinearities, cocyclicities, etc. In practice, it means that a small random perturbation of the distances will perturb slightly only the configuration corresponding to the solution.

There is an exponential number of subsystems to check if they are not over-constrained, but several polynomial time methods have been proposed, by L. Lovasz and Y. Yemini [Lovasz and Plummer 1986; Lovasz and Yemini 1982], or by B. Hendrickson [Hendrickson 1992]. The term $2n - 3$ or $2n' - 3$ in Laman’s theorem is explained by the fact that n vertices have $2n$ coordinates, and that any displacement of a point set solution is also a solution. A displacement in 2D is characterized by 3 parameters, say, a translation (2 parameters), and an angle for the orientation. In 3D, the term $3n - 6$ appears. In d dimension, the subtracted constant term is $d(d + 1)/2$: d translations, and $d(d - 1)/2$ rotations (it is the number of pairs of axes, *i.e.*, of independent planes of rotation).

Laman’s theorem is extended in 2D to more general kinds of constraints, but some difficulties are still to be considered. For instance the shape of a triangle is well defined by three lengths, but not by three angles (note however that three angles are indeed correct in 2D spherical geometry). Basically, graphs can not capture properties of geometric groups: projective transforms (which leave the cross ratio invariant), similarities (which leave angles and distance ratios invariant), translations (which leave orientations and distances invariant), rotations (which leave distances and one point invariant), and displacements (which leave invariant distances).

These difficulties are partially solvable in the 2D case [Owen 1991; Owen 1996]. Actually, even a partial characterization of correctness (partial because correctness conditions are necessary, but not sufficient) is useful, since this characterization permits the detection of some common errors, and the decomposition of systems which can not be solved otherwise into simpler solvable systems.

In 3D, Laman’s theorem can be extended as follows: for the relative location of n points to be well defined, $e = 3n - 6$ distance constraints are needed, and no subsystem is over-constrained, *i.e.*, for all subsystems with n' points and e' distance constraints, $e' \leq 3n' - 6$. This condition is necessary but not sufficient; a famous counter example is the double banana in Figure 1. Interested readers can find on the Internet pictures of the spider banana, which is another counter example (unfortunately a static figure is not sufficient to understand the spider banana). Today no combinatorial characterization (in terms of graphs) is known. However it is common to use Laman’s conditions to decompose geometric systems in 3D: after all, these conditions are necessary. Several methods have been proposed [Hoffmann et al. 2001a; Hoffmann et al. 2001b; Gao and Zhang 2003; Jerermann et al. 2003; Serré et al. 2001].

These graph based methods are misled by 3D configurations (Figure 1) such as the classical and unlikely ”double banana”, but also by more probable and equivalent configurations due to A. Ortuzar. C. Jerermann also exhibits such configurations. Modifying the graph based methods to overcome this limitation is possible but yields to not easily reproducible methods. An alternative is to use the NPM, which is polynomial in time and very easy to implement because it only uses linear algebra (Gauss elimination).

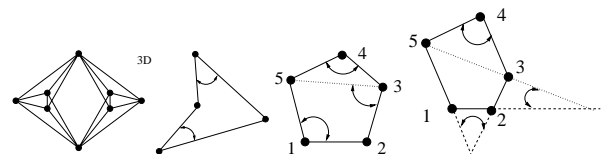


Figure 1: The double banana (left), and three Ortuzar’s configurations (courtesy of A. Ortuzar, Dassault Systèmes).

2.2 The Numerical Probabilistic Method

Equations $F(X) = 0$ are independent if the Jacobian has full rank at the searched root. Unfortunately, the latter is unknown, but it is still possible to study the Jacobian structure at a generic (random) configuration X_0 . To speed up the process, and to avoid inaccuracy difficulties, it is possible to compute a base of the Jacobian in some finite field $\mathbb{Z}/p\mathbb{Z}$, with a prime integer $p \approx 10^9$ for example.

NPM is well known in rigidity theory. It deals with the following problem: for a given dimension, for a given graph, assuming edges carry generic lengths, is the graph rigid or not? For the 2D case, Laman's theorem gives a characterization of rigidity, and several polynomial time methods have been proposed [Lovasz and Yemini 1982; Lovasz and Plummer 1986; Hendrickson 1992]. For the 3D case, a combinatorial characterization is still unknown. NPM applies in any dimension and it is still polynomial in time. Actually, it is always cubic in time, whatever the dimension. Its probabilistic nature is not a taboo for engineers.

We extend the NPM to other geometric constraints. To this end, we tried several translations of constraints into equations, and various representations (cartesian or barycentric coordinates). Here we present the simplest way, where points are represented by cartesian coordinates. There are two kinds of unknowns: (a) cartesian coordinates of points and (b) unknowns the values of which are independent of the cartesian frame used, such as distances, scalar products, signed areas, volumes and non-geometric unknowns (forces, moments). The only unknowns the values of which depend on the cartesian frame are the points coordinates. Vectorial equations $t_1 \times \vec{P}_i \vec{P}_j + t_2 \times \vec{P}_k \vec{P}_l + \dots = \vec{0}$ specify collinearities, coplanarities, incidences and intersections. For instance, to specify that I is the middle of AB , use $\vec{AI} - \vec{IB} = 0$. To specify that $J = AB \cap CD$, introduce two new scalar unknowns α, β and use equations: $\vec{AJ} = \alpha \vec{AB}, \vec{CJ} = \beta \vec{CD}$. All incidence constraints are translated into vectorial equations.

Scalar equations (*i.e.* non-vectorial equations) are used to specify distances and angles between vectors, or non-geometric constraints (*e.g.*, mechanical, physical). The constraint

$distance(A, B) = d$ is translated into the equation: $d^2 - \vec{AB} \cdot \vec{AB} = 0$.

The constraint $angle(\vec{AB}, \vec{CD}) = \theta$ is translated into equations:

$$\begin{aligned} \vec{AB} \cdot \vec{CD} - l_{AB} l_{CD} \cos \theta &= l_{AB}^2 - \vec{AB} \cdot \vec{AB} \\ &= l_{CD}^2 - \vec{CD} \cdot \vec{CD} \\ &= 0 \end{aligned} \quad (1)$$

Scalar products are translated trivially, *e.g.*, in 2D: $\vec{AB} \cdot \vec{CD}$ is rewritten as $(x_A - x_B)(x_D - x_C) + (y_A - y_B)(y_D - y_C)$. Finally, vectorial equations are straightforwardly translated into d scalar equations in dimension d . This translation of constraints into equations is systematic and automatic. The user specifies which names are parameters, *i.e.*, have known values; all coordinates are unknown. The language used to specify constraints guarantees that all constraints are indeed independent of the cartesian frame, *e.g.*, the user can not specify point coordinates. With this formulation, NPM indeed detects the bad constrainedness in all pathologic configurations of Figure 1 (the classic NPM already detected the problem with the double banana), and their variants.

It is tempting to square equation (1) to get $(\vec{AB} \cdot \vec{CD})^2 - k_\theta^2 \times (\vec{AB} \cdot \vec{AB})^2 \times (\vec{CD} \cdot \vec{CD})^2 = 0$, since it decreases the number of equa-

tions and unknowns. However it is not a good idea: with the formulation of equation (1), there are some linear dependences between gradient vectors of the Jacobian which are detected by NPM and which disappear with equation squaring.

Other representations can be tried: Grassmann Cayley coordinates for instance, or intrinsic approaches which discard all coordinates [Yang 2002; Michelucci and Foufou 2004].

For a given system of equations, NPM works as follows. It chooses random values X_0 for unknowns X , then computes with Gauss elimination a base of the lines of the Jacobian $F'(X_0)$, with the convention that each line of the Jacobian is the gradient vector of an equation. If a line reduces to zero during elimination, a dependence has been discovered (with a small risk of error, as discussed below); this dependence is due either to a redundancy, or to a contradiction, but the NPM can not distinguish between these two cases. If no line vanishes, then all equations are independent, and then there is no possible error; of course, independent equations may have no real solution, such as $x^2 + 1 = 0$; but detecting the absence of real solutions is in general an untractable problem.

The Gram-Schmit's procedure can also be used, instead of the Gauss method, if the underlying field is \mathbb{Q} . However, we prefer to compute in a finite field, and in this case, isotropic vectors (non-null vectors with a vanishing norm) can occur, and prevent the use of the Gram-Schmit's procedure. Indeed, the Gauss triangulation method is sufficient to compute a base; moreover, it also permits the expression of a vector v in a given base $b_1 \dots b_k$: just solve the linear system $v = t_1 b_1 + \dots t_k b_k$ with k unknowns $t_1 \dots t_k$ with the Gauss method.

MNP is probabilistic at several levels. First, there is the risk that for the chosen random values in \mathbb{Q} , the gradient vectors in the Jacobian are dependent, whereas they are independent almost everywhere else. Of course, guessing at random a zero of a polynomial (the determinant of a minor of the Jacobian) has a very low probability, and many engineers will simply consider it impossible. To gain more confidence, the MNP test can be performed several times. In passing, note that to probabilistically prove that a multivariate polynomial is identically zero, one evaluation at a random value chosen in a huge set is sufficient. A deterministic proof needs much more time. Randomness is a very powerful friend of computer scientists. Second, in finite fields $\mathbb{Z}/p\mathbb{Z}$, there is the risk that some vectors are dependent modulo the prime integer used p , although they are not in \mathbb{Q} ; for instance, a square matrix can have a non-null determinant which is a multiple of p . It is very unlikely and it never happened in our experiments, but in theory this case can not be discarded. Here again, the simplest solution is to iterate the NPM when it says it detects a dependence; let ε be the probability of error, *i.e.*, that MNP wrongly detects a dependence. Performing k independent tests with NPM, with k wrong detections of dependences has probability ε^k . If the random values and p are chosen in a set with cardinality Ω , then ε has order of magnitude Ω^{-1} ; it is easy to reach $\Omega \geq 10^9$ (we skip the details); then the risk of k wrong decisions is $\varepsilon^k = 10^{-9k}$. A value $k = 10$ should suffice to reassure all anxious users.

2.3 Decomposition with NPM

NPM can also detect rigid subsystems in polynomial time. Two points P and Q are "relatively fixed" by a system $F(X) = 0$ iff the gradient vectors of the equations resulting from the constraint $\vec{PQ} \cdot \vec{PQ} - l_{PQ}^2 = 0$, where l_{PQ} is a parameter, lie in the vectorial space spanned by the Jacobian base; as usual, all computations are done at some random configuration.

The following greedy method finds in polynomial time maximal (for inclusion) rigid parts: in 2D, start from an "anchor" of two relatively fixed points A and B , and add all points M which are relatively fixed with A and with B . In 3D, we can proceed in the same way except that the anchor is made of three relatively fixed points A, B and C . To decompose a maximal rigid part, remove a constraint and compute the maximal rigid parts; see Figure 2 for an example. The simplicity of this decomposition method contrasts with the complexity of graph based methods.

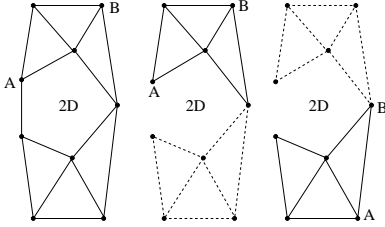


Figure 2: Decomposition with the NPM.

Researchers have proposed to decompose modulo geometric groups, and not only modulo the usual group of displacements, which yields the rigid subsystems. Schramm and Schreck [Schramm and Schreck 2003] investigate the decomposition modulo the group of similarities: it turns out that some sub-figures well-constrained for this group, *i.e.*, well-constrained up to scale, provide information (ratios between lengths and angles) which can be used to solve other parts of the system. Further, we suggest to decompose the systems of constraints modulo the group of projective mappings.

Here a natural question arises: is it possible to decompose modulo a specified geometric group with NPM?

2.4 Remaining limitations

However, NPM and a number of graph based methods are still misled by geometric theorems: if constraints contain the hypothesis and the conclusion (or its negation) of a theorem, NPM can not detect the dependence between constraints. Basically, geometric theorems do not reduce the rank of the Jacobian everywhere, whereas NPM computes the rank only at a generic (random) configuration.

Here is the simplest example. In 3D, specifying that lines AB and CD meet at point I implies that A, B, C and D are coplanar. But NPM does not detect the redundancy of the coplanarity constraint on A, B, C and D . More sophisticated theorems are not detected as well.

However, if NPM is used not at a random configuration, but at a configuration fulfilling vectorial equations (*i.e.* incidence constraints), it detects the dependence between equations, *i.e.* the rank deficiency. Thus a natural idea is to find a configuration fulfilling vectorial equations, and to study it with NPM. To compute such a configuration we use Jurzak's prover.

3 Jurzak's prover

The most powerful and sophisticated geometry provers are usually based on precise computer algebra machinery such as Grobner bases or triangular sets [Chou 1988]. After Dieudonné and probably other mathematicians, but in a more effective computational way, Jurzak [Jurzak 2003] remarked that numerous theorems

```

let e= point "e"
let a= point "a"
let b= point "b"
let c= lie_on [ e; a; b]
let a' = lie_on [e; a]
let b' = lie_on [e; b]
let c' = lie_on [e; c]
let c'' = inter [[a; b]; [a'; b']]
let a'' = inter [[b; c]; [b'; c']]
let b'' = inter [[a; c]; [a'; c']]
if 2 = rank [a'' ; b'' ; c'' ] then
Printf.printf "Desargue's theorem is true\n"
else Printf.printf "Desargue's theorem is
wrong :\n" .

```

Figure 3: Session for the proof of Desargues' theorem. In 2D, if two triangles abc and $a'b'c'$ are perspective with respect to a point e (in other words, ead' , ebb' and ecc' are collinear), then the three intersection points between homologous sides $ab \cap a'b'$, $ac \cap a'c'$ and $bc \cap b'c'$ are collinear.

of affine or projective geometry (such as Beltrami's theorems) can be proved using only linear algebra, if a vectorial representation is used. It yields a simple and fast prover, easily programmable by students in computer science, also well suited for geometry teaching. This prover can also be used by solvers of geometric constraints (see section 4).

3.1 The principle

The typical session in Figure 3 illustrates the proof of Desargues' theorem, from a user point of view. See Figures 4, 5, 6 and 7 for the proofs of Pascal, Pappus, hexamys, and Beltrami's theorems. To prove Desargues's theorem, the program defines three independent points e , a and b (Jurzak call them pure points). Then other points are constrained to lie on flats (lines, planes, etc) spanned by previously defined points. If 3 (4, 5...) pure points are created, we are in 2D (3D, 4D...) geometry. The functions `inter` for intersection, and `rank` have self explanatory names. Actually, the definition `let c = lie_on [e; a; b]` creates two independent parameters c_a and c_b , and defines c as $c = e + c_a \times \vec{ea} + c_b \times \vec{eb}$. All dependent points are defined this way, as linear combinations of base points, which are pure symbols. In the simplest case, coefficients in the combinations are numerical values (rational numbers, or numbers in some finite field $\mathbb{Z}/p\mathbb{Z}$, where p is a prime integer). To prove theorems in a more general way, it is needed (or at least it seems, a priori) to compute with symbolic parameters; in other words, coefficients in the combinations are rational functions of the parameters c_a, c_b, \dots , *i.e.* they are ratios of polynomials in c_a, c_b, \dots ; these polynomials have typically rational or integer coefficients.

In all cases, the coefficients in the linear combinations lie in a field, called \mathbb{K} . In the simplest case, \mathbb{K} is a numerical field like \mathbb{Q} , the field of rational numbers, or some finite field, thus no symbolic parameter occurs. They are replaced by explicit numerical values; for instance the expression $c = e + c_e \vec{ea} + c_b \vec{eb}$ is replaced by $c = e + 3\vec{ea} + 4\vec{eb}$.

In the case of $\mathbb{K} = \mathbb{Q}$, it is trivial to implement the two main functions: `inter` and `rank`: they just need linear algebra with numbers in $\mathbb{K} = \mathbb{Q}$ or $\mathbb{K} = \mathbb{Z}/p\mathbb{Z}$. For instance, the expression `let i=inter [[p;q;r]; [a;b]]`, which defines i as the intersection point between the plane (p, q, r) and the line (a, b) , needs to solve the linear system: $p + x \times \vec{pq} + y \times \vec{pr} = a + z \times \vec{ab}$ with un-

```

let i= point "i"
let a= point "a"
let b= lie_on [ i; a]
let c= lie_on [ i; a]
let a'= point "a'"
let b'= lie_on [ i; a']
let c'= lie_on [ i; a']
let a''= inter [[b; c']; [b'; c]]
let b''= inter [[c; a']; [c'; a]]
let c''= inter [[a; b']; [a'; b]]
if 2 = rank [ a'' ; b''; c'' ]
then "Pappus is true" else "Pappus is wrong" .

let i= point "i"
let a= point "a"
let b= let p_b=parameter "p_b" in
        i +% p_b *(a -% i)
let c= let p_c=parameter "p_c" in
        i +% p_c *(a -% i)

let a'= point "a'"
let b'= let p_b'=parameter "p_b'" in
        i +% p_b' *(a' -% i)
let c'= let p_c'=parameter "p_c'" in
        i +% p_c' *(a' -% i)

let a''= inter [[b; c']; [b'; c]]
let b''= inter [[c; a']; [c'; a]]
let c''= inter [[a; b']; [a'; b]]
if 2 = rank [ a'' ; b''; c'' ]
then "Pappus is true" else "Pappus is wrong";;.

```

Figure 4: Proof of Pappus's theorem: if a, b, c are 3 collinear points, as well as a', b', c' , then $a'' = bc' \cap b'c$, $b'' = ac' \cap a'c$, $c'' = ab' \cap a'b$ are also collinear.

knows x, y, z . Replacing p, q, r, a, b with their linear combinations of some pure points b_1, \dots, b_4 gives 4 linear equations in the three unknowns x, y, z . Any one of these equations is a linear combination of the three others and can be ignored. Solving for x, y, z gives the intersection point. All computations needed for linear algebra (sum, product, subtraction, division, nullity test) are made in $\mathbb{K} = \mathbb{Q}$ or $\mathbb{K} = \mathbb{Z}/p\mathbb{Z}$. The typical \mathbb{K} is $\mathbb{Q}(X)$, where $X = \{c_a, c_b, \dots\}$ is the set of independent parameters. The word "independent" means that there is no equation relating the parameters. Actually, it is the main reason of the simplicity of Jurzak's method: since there is no algebraic equation constraining the parameters, no sophisticated algorithm from computer algebra (Groebner bases, triangular sets, resultants, gcd, etc) is needed to solve these equations.

$\mathbb{Q}(X)$ is the set of ratios between polynomials with coefficients in \mathbb{Q} and variables in X . To implement linear algebra with $\mathbb{K} = \mathbb{Q}(X)$, *i.e.*, to solve linear systems for the functions `inter` and `rank`, we just need, as before, the four basic "field operations" in $\mathbb{K} = \mathbb{Q}(X)$: the sum, difference, product, and division between $r_1(x_1, x_2, \dots) = n_1(x_1, x_2, \dots)/d_1(x_1, x_2, \dots)$ and $r_2(x_1, x_2, \dots) = n_2(x_1, x_2, \dots)/d_2(x_1, x_2, \dots)$ where n_1, d_1, n_2, d_2 are polynomials in x_1, x_2, \dots with rational coefficients; the nullity test (or the equality test) in the field \mathbb{K} is also required. The next section discusses several implementations for Jurzak's method: it is just a discussion about the corresponding implementations of the computation in the field $\mathbb{K} = \mathbb{Q}(X)$.

3.2 Empty implementation for $\mathbb{K} = \mathbb{Q}(X)$

In the zeroth implementation, $\mathbb{K} = \mathbb{Q}(X)$ is not implemented. Only $\mathbb{K} = \mathbb{Q}$ is available (ocaml, the language used here, provides a li-

```

let a= point "a"
let b= point "b"
let c= lie_on [a; b]
(* we need another point z. take z=a1, ok) : *)
let z=point "z"
(* generate points in the plane : *)
let a1= lie_on [a; b; z]
let a2= lie_on [a; b; z]
let b1= lie_on [a; b; z]
let b2= lie_on [a; b; z]
let c1= lie_on [a; b; z]
let c2= lie_on [a; b; z]
(* define the 6 vertices of the hexamy: *)
let p0 = inter [[a;a1]; [c;c2] ]
let p1 = inter [ [b;b1]; [c;c2] ]
let p2 = inter [ [b;b1]; [a;a2] ]
let p3 = inter [ [c;c1]; [a;a2] ]
let p4 = inter [ [c;c1]; [b;b2] ]
let p5 = inter [ [a;a1]; [b;b2] ]
let is_an_hexamy a b c d e f = 2 =
rank[inter[[a;b]; [d;e]];
inter[[b;c]; [e;f]]; inter[[c;d]; [f;a]]]
(* true, by construction *)
is_an_hexamy p0 p1 p2 p3 p4 p5
(* true, by Pascal's theorem *)
if is_an_hexamy p1 p0 p2 p3 p4 p5
then Printf.printf "hexamy is proved\n"
else Printf.printf "hexamy is wrong :(\n".

```

Figure 5: An hexagon is an hexamy if its opposite sides cut in 3 collinear points. Any permutation of an hexamy is an hexamy. This form of Pascal's theorem involves only lines.

```

(* (a,b) is the first black line *)
let a= point "a" let b= point "b"
(* (c,d) is the second black line *)
let c= point "c" let d= point "d"
(* e lie in 3D space spanned by a, b, c, d *)
let e= lie_on [a; b; c; d]
(* (e,f) is the third black line *)
let f= lie_on [a; b; c; d]
(* (p,p') is the first white line *)
let p = lie_on [a; b]
let p' = inter [[p;c;d]; [e;f]]
(* (q,q') is the second white line *)
let q = lie_on [a; b]
let q' = inter [[q;c;d]; [e;f]]
(* (r,r') is the third white line *)
let r = lie_on [a; b]
let r' = inter [[r;c;d]; [e;f]]
(* (t,t') cuts the 3 black lines *)
let t= lie_on [a; b]
let t'= inter [[t;c;d]; [e;f]]
(* (g,h) cuts the 3 white lines *)
let g= lie_on [p; p']
let h= inter [[g;q;q']; [r;r']]
(* (t,t') and (g,h) are coplanar, they cut *)
if rank [t; t'; g; h]=3 then
Printf.printf "Beltrami's theorem is true\n" else
Printf.printf "Beltrami's theorem is wrong\n".

```

Figure 6: Proof of Beltrami's theorem (the 16 points theorem). Three (non-coplanar) white lines cut three (non coplanar) black lines in 9 points. A line is said black (white) if it cuts the 3 white (black) lines. All black lines cut all white lines.

```

let o= point "o" let a= point "a"
let b= point "b" type 'pt conic =
{ a0 : elt; a1: elt; a2: elt; b0 : elt;
  b1: elt; b2: elt; a : (elt * 'pt) list;
  b : (elt * 'pt) list; c : (elt * 'pt) list
}
let conique = { a0 = of_int 0; a1= of_int 0;
  a2=of_int 1; b0 = of_int 0; b1= of_int 1;
  b2=of_int 0; a = a; b=b; c=o }
let pt_on_conic { a0=a0; a1=a1;a2=a2; b0=b0;
  b1=b1; b2=b2;a=a; b=b; c=c }
t =
let a_coeff= a2 * / t * / t + / a1 * / t + / a0 in
let b_coeff= b2 * / t * / t + / b1 * / t + / b0 in
let c_coeff = (of_int 1) - / a_coeff - / b_coeff
in a_coeff *% a +% b_coeff *% b +% c_coeff *% c
let p0 = pt_on_conic conique (parameter "t0 ")
let p1 = pt_on_conic conique (parameter "t1 ")
let p2 = pt_on_conic conique (parameter "t2 ")
let p3 = pt_on_conic conique (parameter "t3 ")
let p4 = pt_on_conic conique (parameter "t4 ")
let p5 = pt_on_conic conique (parameter "t5 ")
let i = inter [[p0; p1]; [p3; p4]]
let j = inter [[p1; p2]; [p4; p5]]
let k = inter [[p2; p3]; [p5; p0]]
if 2 = rank [ i; j; k] then
Printf.printf "Pascal' theorem is proved\n"
else Printf.printf "Pascal' theorem is wrong!\n".

```

Figure 7: Session for the proof of Pascal's theorem: the opposite sides of any hexagon inscribed in a conic cut in three collinear points. Since the theorem is projective, any conic can be used, for instance a parabola (it avoids the problem of solving quadratic equations). We define conics, on the fly, as the set of points $A(t)a + B(t)b + (1 - A(t) - B(t))c$, where a, b, c are three non collinear points, and $A(t), B(t)$ two quadratic functions in the parameter t . Each point on the conic is defined by its value for the t parameter. It should also be possible to use a symbolic conic, with symbolic parameters a_2, a_1, a_0 for $A(t) = a_2t^2 + a_1t + a_0$, and idem for $B(t)$.

rary for arithmetic in \mathbb{Q}). Of course, we can do no symbolic computation with parameters, we can just verify the theorem at hand with a random value for each parameter. This seemingly stupid implementation, or no-implementation, is actually the best one, as we will ironically conclude.

3.3 Classical implementation for $\mathbb{K} = \mathbb{Q}(X)$

Elements in $\mathbb{K} = \mathbb{Q}(X)$ are classically represented by a pair of polynomials, the numerator and the denominator. Polynomials (the set of polynomials is noted $\mathbb{Q}[X]$) are represented by a list of monomials; a monomial is a pair made of a coefficient (some rational number) and a power product. x^2yz^3 is a typical power product. All that is very well known in computer algebra. The cons of this representation is also well known. During computation, the degrees of the polynomials increase rapidly, and fill the memory space. Actually, the proof of Beltrami's theorem fails because of insufficient memory. A solution is to simplify the ratios, but it needs complicated methods from computational algebra (gcd of polynomials). Note that the same kind of problem occurs at the rational number arithmetic level: rational numbers should be reduced during computations to avoid memory failure.

3.4 DAG implementation for $\mathbb{K} = \mathbb{Q}(X)$

The DAG (Directed Acyclic Graph) representation (see for instance [Rege 1995]) is used to solve the previous problem. Polynomials are no more represented by their list of monomials, but by trees the leaves of which carry parameters or numbers, and the nodes of which carry binary operators: $+, -, \times$ and possibly $/$ for division. Thus each operation in $+, -, \times, /$ is done in constant time and space. The remaining question is the one of evaluation, more precisely the one of deciding if a given DAG represents the identically zero expression or not. The probabilistic paradigm [Schwartz 1980] is used: the DAG is evaluated (in time proportional to its size) for random values of the parameters. Clearly the probability for guessing a root of the DAG is negligible; thus if the DAG evaluates to zero with a random binding, the underlying expression is very likely zero (and the DAG can be replaced by the simple DAG zero).

To reassure anxious users, and to gain more confidence, this stochastic test can be repeated several times. This test can also be made deterministic, in several ways. Consider to simplify a polynomial in only one variable: if we have an upper bound d of the degree, and if the polynomial vanishes in $d + 1$ distinct sample values, then it can only be the zero polynomial; or, if we have an upper bound for the magnitude of the coefficients, we can compute also (with some formula; see Mignotte and Stefanescu's book [Mignotte and Stefanescu 1999]) an upper bound for the magnitude of the root module: so if the DAG vanishes for a number larger than this upper bound, then the polynomial can only be the zero polynomial. This principle for univariate polynomials can be extended to the multivariate case. However, making these tests deterministic (no more probabilistic) has an exponential cost.

Another optimization is not to compute with rational numbers (their size increases quickly during computation) but modulo some prime numbers.

However the DAG representation is not a panacea. It uses a huge amount of memory. Care should be taken to not evaluate shared nodes several times. These considerations yield to the third implementation.

3.5 Final implementation for $\mathbb{K} = \mathbb{Q}(X)$

To get the third implementation, it suffices to realize that the DAG implementation is just a complicated way to use the zeroth implementation! We can simply run several times the session of the proof with different random numerical values (in \mathbb{Q} or in some $\mathbb{Z}/p\mathbb{Z}$ field) for parameters at each time.

This solution is by far the simplest and fastest one. All theorems are (probabilistically) proved in less than a fraction of a second. It is also possible to use a floating point implementation for one of the k indices to display it.

The probabilistic paradigm is also used in the qualitative study of systems of geometric constraints [Lamure and Michelucci 1998]: the determinant of the Jacobian matrix of the system of equations to be solved, or the rank and structure (which subsets of equations are dependent or not) of the Jacobian matrix, is computed for random values of the unknowns. Section 4 gives more details on this point.

3.6 Discussion

Even if Jurzak cleverly avoids the need for non-linear algebra (gcd, elimination in non-linear equations, etc), the daemon of complexity

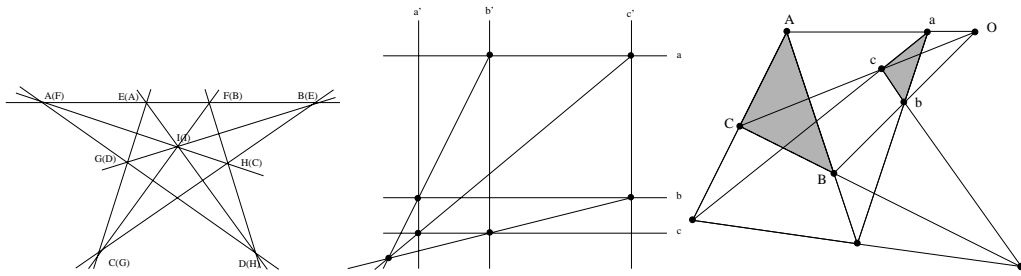


Figure 8: Projective configurations: a non-rational one, Pappus, Desargues.

forbids the use of deterministic and exact computations in $\mathbb{Q}(X)$ for complicated theorems. Instead, some probabilistic and polynomial time ($O(n^3)$ since we use mainly Gauss method) can be used, and ironically the best implementation of $\mathbb{Q}(X)$ is the empty one.

A note in passing, if we just compute with (representative, generic) instances, we can also admit some non-linear operations for the construction of points, such as taking the square root: it permits to compute ruler and compass constructions, with line/circle or circle/circle intersections. Of course a quadratic arithmetic, able to compute in the quadratic closure of \mathbb{Q} or $\mathbb{Z}/p\mathbb{Z}$, is needed. It has been proposed by D. Bouhineau in [Bouhineau 1996]. These arithmetics permit to check geometric theorems on examples, thus to probabilistically prove geometric theorems involving circles (for instance: that three common chords to 3 circles concur). Unfortunately, these quadratic arithmetics have an exponential cost: a number in $\mathbb{Q}[\sqrt{a_1}] \dots [\sqrt{a_k}]$ is represented by 2^k coefficients in \mathbb{Q} , the lengths of which are also exponential. Moreover these arithmetics can not be used to compute with general conics (not circles); the latter need more general (and more complex) algebraic arithmetics.

4 Why solvers should use Jurzak's prover

This section is devoted to the question: Why geometric solvers should use Jurzak's prover (we mean the stochastic implementation of Jurzak's prover).

The core idea is to apply NPM to a configuration which respects the explicitly specified "projective constraints" such as collinearities and coplanarities, and thus will also respect incidence constraints due to geometric theorems (Desargues, Pappus, hexamys, etc). The NPM works much better on such configurations. We can use Jurzak's prover to compute a generic configuration fulfilling the vectorial equations.

Is it possible to find a configuration respecting the specified projective constraints? For CAD/CAM applications, very often the answer is yes. It is due to the fact that, for dimensioning of CAD/CAM parts, systems are almost well-constrained, and there is a majority of metric constraints, thus there are relatively few projective constraints. For the modeling of free form curves and surfaces, systems are typically very under-constrained. The greedy method which follows will often suffice. For simplicity, it is explained in the 2D case: if a point is constrained to lie on only 2 lines, we can forget this point, solve the rest of the projective system, then compute this point as the intersection point between the two lines. The same thing holds for the dual case, where a line passes by only two points of the configuration.

Of course, the previous method of construction may fail in some cases. Figure 8 shows a classical example, from Grunbaum's book [Grunbaum 1967]. This configuration is probably the simplest one

with no rational (nor integer) realizations (coordinates). It also proves that projective systems are not linear systems. Pappus' or Desargues's configurations, when the line of the conclusion is added, are other examples which make the greedy method fail, with the peculiarity that they become easily solvable if the conclusion line is removed, and, due to symmetry, all lines can be considered as the conclusive line.

Again, projective systems which make the greedy method fail are very unlikely in CAD/CAM but we can not resist to the temptation to discuss this topic. Is it possible to decompose projective systems, as we do for rigid systems with Owen's [Owen 1991] or the HLS (Hoffmann, Lomonosov, Sitharam [Hoffmann et al. 2001b]) methods? Note that a "projectively well- constrained" part has 8 remaining degrees of freedom in 2D (15 in 3D), while a rigid part has 3 remaining degrees of freedom in 2D as is well known; thus, decomposition methods into rigid parts have to be changed. Is it possible to temporarily forget one of the projective constraints in order to simplify the problem, solve the remaining problem, and finally take into account the forgotten constraint? If we are lucky, the forgotten constraint may even be a consequence (by Desargues's theorem, or Pappus, etc) of the other constraints, and Jurzak's prover detects it very quickly. Does this last idea give a usable method? Is it possible to build a dictionary of "atomic" projective configurations, and to quickly recognize them? Actually, we promote here the study of projective constraints and the decomposition of constraints modulo the group of projective transforms: the invariant in this group is the cross ratio.

5 Handling orthogonality with Jurzak's prover

The core idea is to apply NPM on a configuration similar to the unknown one. Jurzak's prover permits to find a configuration similar in the sense that it fulfils the same projective constraints. It should be even better if we are able to take into account orthogonality constraints. A priori orthogonality constraints are not projective, but metric.

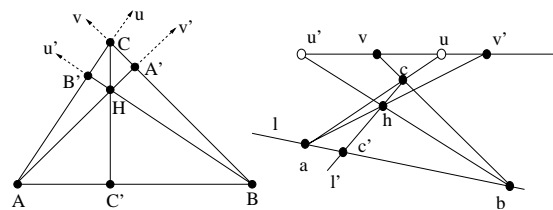


Figure 9: Pouzergues's construction.

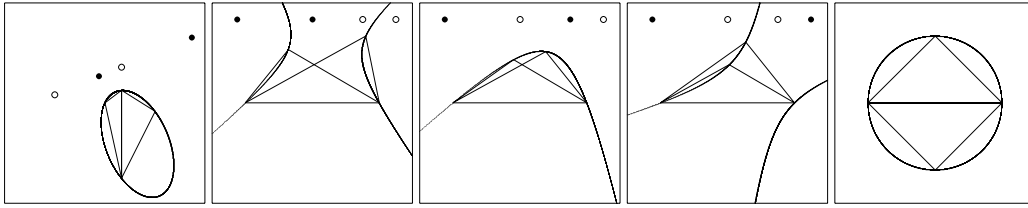


Figure 10: Projective circles. Small black disks are two directive points in involution. Idem for small white disks. The order of directed points along the vanishing line determines the kind of the conic.

It seems that Jurzak’s prover is intrinsically unable to prove metric theorems such as:

- Orthocentre theorem: the three altitudes of a triangle cut at a common point (the orthocentre).
- Three chords theorem: the pairwise common chords to three circles concur.
- Clifford theorem: if three circles with radius 1 have a common point, the three other intersection points between pairs of circles lie on a circle with radius 1.

Jurzak’s prover can at least prove theorems involving orthogonality. This is due to the fact that orthogonality constraints can be translated into incidence constraints, which is not obvious at all. This fact permits Raymond Pouzegues to prove (projective reformulations of) metric theorems.

Basically, two pairs of points (u, u') and (v, v') along a line define a unique involution α on this line: for any point p on this line, it is possible to construct $p' = \alpha(p)$, the image of p by this involution; geometrically speaking, this construction only uses the ruler; algebraically speaking, this construction lies in the initial field of the coordinates of u, u', v, v' .

An orthogonality in the projective plane (in spite of the paradoxical appearance of such a name) can then be defined by an involution on some line (intuitively, on the vanishing line of the plane). All lines with vanishing point u (or v) are perpendicular to all lines with vanishing point u' (or v'). A construction illustrated in Figure 9 gives the vanishing point $p' = \alpha(p)$ of all lines perpendicular to lines with given vanishing point p . More precisely, let l be any line through p , and c any point outside l and the line u, u', v, v' . Then the line perpendicular to l and passing through c is $c \vee h$, where $h = (b \vee u') \cap (a \vee v')$, with $b = l \cap (c \vee v)$ and $a = l \cap (c \vee u)$ (The notation $c \vee v$ means the line joining c and v). The point $p' = \alpha(p)$ is $(u \vee u') \cap (c \vee h)$. The point p' depends on u, u', v, v', p but does not depend on either l , or c . The point h is the ”orthocentre” (for the orthogonality induced by α) of the triangle abc . Other equivalent constructions (exchanging u and u' , or v and v') are possible: these equivalences are theorems. Since all these constructions need only \vee and \cap operations, they can be performed by Jurzak’s prover (as far as the barycentric coordinates for u, u', v, v' lie in a computable field). And indeed Jurzak’s prover proves all these equivalence theorems.

Such a projective definition of orthogonality suggests this projective definition of a ”circle” (*i.e.* conic) with given diameter ab as the set of points p such that ap is orthogonal to pb , for some involution α (see Figure 10). All conics can be defined this way.

We are currently extending this reduction of 2D orthogonality constraints into incidence constraints to 3D.

6 Conclusion

Owen [Owen 1991] introduced the use of graphs in geometric constraints solving for the 2D case; before, people solved directly with numerical methods. In some sense, NPM proposed here can be seen as a return to this previous situation. It is justified by the difficulty of the rigidity problem in 3D, the complications of graph-based methods, and their intrinsic limitations. On the other hand, NPM is very appealing because of its generality and simplicity: it only needs the Gauss elimination method or some variant. Moreover, when applied to a configuration satisfying vectorial equations (incidence constraints), NPM detects dependences which are outside the scope of most of graph based methods and of computer algebra methods due to their cost. NPM can also decompose the system into rigid subsystems in polynomial time as well.

This paper has shown how a simple and easy to implement prover can very efficiently compute configurations fulfilling incidence constraints (or vectorial equations) and detect forced collinearities or coplanarities due to geometric theorems. On the other hand, the (fast) detection of dependence between equations is a key issue for geometric solvers. Jurzak’s prover detects some of these dependences. However, this poses the question of solving the projective part of systems of geometric constraints. For CAD/CAM applications, a greedy obvious method almost always solves the projective part, because the latter is under-constrained. The question of decomposing and solving projective systems (only incidence constraints) is interesting and deserves further study.

Some open questions remain to be answered: is it possible to account for orthogonality constraints, since they can be reduced to incidence constraints? Is it possible to use NPM to decompose geometric constraints modulo any geometric transform group? Should we consider (and how) other incidence constraints, to circles or conics in 2D, to spheres or quadrics in 3D, to cubics?

References

- BOUHINEAU, D. 1996. Solving geometrical constraint systems using clp based on linear constraint solver. In *Proc. of Third international Conference on Artificial Intelligence, and Symbolic Mathematical Computation, AISMC-3*, LNCS 1138, Springer, Steyr, Austria.
- BRDRLIN, B., AND ROLLER, D., Eds. 1998. *Geometric Constraint Solving & Applications*. Springer Verlag, June.
- CHOU, S.-C. 1988. *Mechanical Geometry theorem Proving*. D. Reidel Publishing Company.
- GAO, X.-S., AND ZHANG, G. 2003. Geometric constraint solving via c-tree decomposition. In *ACM Solid Modelling*, ACM Press, New York, 45–55.

- GRÜNBAUM, B. 1967. *Convex polytopes*. London Interscience.
- HENDRICKSON, B. 1992. Conditions for unique realizations. *SIAM J. Computing* 21, 1, 65–84.
- HOFFMANN, C., LOMONOSOV, A., AND SITHARAM, M. 2001. Decomposition plans for geometric constraint problems, part II : New algorithms. *J. Symbolic Computation* 31, 409–427.
- HOFFMANN, C., LOMONOSOV, A., AND SITHARAM, M. 2001. Decomposition plans for geometric constraint systems, parts I : Performance measures for CAD. *J. Symbolic Computation* 31, 367–408.
- HOFFMANN, C., SITHARAM, M., AND YUAN, B. 2004. Making constraint solvers useable: overconstraints. *Comp. Aided Design*.
- JERMANN, C., NEVEU, B., AND TROMBETTONI, G. 2003. Algorithms for identifying rigid subsystems in geometric constraint systems. *IJCAI*, 233–238.
- JOAN-ARINYO, R., SOTO-RIERA, A., VILA-MARTA, S., AND VILAPLANA-PASTO, J. 2003. Transforming an under-constrained geometric constraint problem into a well-constrained one. In *Proc. of Symposium on Solid Modeling and Applications*, ACM, 33–44.
- JOAN-ARINYO, R., SOTO-RIERA, A., VILA-MARTA, S., AND VILAPLANA-PASTO, J. 2004. Revisiting decomposition analysis of geometric constraint graphs. *Computer-Aided Design* 36, 2, 123–140.
- JURZAK, J.-P. 2003. Géométrie vectorielle algorithmique : La géométrie vectorielle par l’informatique. Tech. Rep. <http://passerelle.u-bourgogne.fr/publications/webgeometrie/>, Université de Bourgogne, Dijon.
- KORTENKAMP, U., AND RICHTER-GEBERT, J. 1998. *The Interactive Geometry Software Cinderella*. Springer-Verlag.
- KORTENKAMP, U. 1999. *Foundations of Dynamic Geometry*. PhD Thesis No 13403, Swiss Federal Institute of Technology, Zurich, Switzerland.
- LAMAN, G. 1970. On graphs and rigidity of plane skeletal structures. *J. Engineering Math.* 4, 331–340.
- LAMURE, H., AND MICHELUCCI, D. 1998. Qualitative study of geometric constraints. In *Geometric Constraint Solving and Applications*, Springer-Verlag.
- LIN, V. C., GOSSARD, D. C., AND LIGHT, R. A. 1981. Variational geometry in computer-aided design. In *Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, ACM Press, 171–177.
- LOVASZ, L., AND PLUMMER, M. 1986. *Matching Theory*. North-Holland.
- LOVASZ, L., AND YEMINI, Y. 1982. On generic rigidity in the plane. *SIAM J. Alg. Discr. Meth.* 3, 1, 91–98.
- MICHELUCCI, D., AND FOUFOU, S. 2004. Using Cayley-Menger determinants for geometric constraint solving. In *ACM Symposium on Solid Modeling*.
- MIGNOTTE, M., AND STEFANESCU, D. 1999. Polynomials: An algorithmic approach. In *Discrete Mathematics and Theoretical Computer Science Series*, vol. XI. Springer.
- OLANO, S., AND BRUNET, P. 1994. Constructive constraint-based model for parametric cad systems. *Computer-Aided Design* 26, 8, 614–621.
- OWEN, J. 1991. Algebraic solution for geometry from dimensional constraints. In *Proc. of the Symp. on Solid Modeling Foundations and CAD/CAM Applications*, 397–407.
- OWEN, J. 1996. Constraints on simple geometry in two and three dimensions. *Int. J. of Computational Geometry and Applications* 6, 4, 421–434.
- POUZERGUES, R. 2002. Les hexamys. Tech. rep., <http://hexamys.free.fr/>.
- REGE, A. 1995. A complete and practical algorithm for geometric theorem proving. In *ACM Symp. Computational geometry*.
- SCHRAMM, E., AND SCHRECK, P. 2003. Solving geometric constraints invariant modulo the similarity group. In *Proceedings of Computational Science and Its Applications - ICCSA 2003, International Conference, Part III, Montreal, Canada, May 2003*, Springer, vol. 2669 of *Lecture Notes in Computer Science*, 356–365.
- SCHWARTZ, J. 1980. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM* 4, 27, 701–717.
- SERRÉ, P., RIVIERE, A., DUONG, A., AND ORTUZAR, A. 2001. Analysis of a geometric specification. In *Proceedings of the 27th Design Automation Conference, ASME Design Engineering Technical Conferences*.
- SITHARAM, M., AND ZHOU, Y. 2004. A tractable, approximate characterization of combinatorial rigidity in 3D. In *Automated Deduction in Geometry*.
- SUTHERLAND, I. 1963. *Sketchpad- A Man-Machine Graphical Interface*. PhD thesis, MIT.
- YANG, L. 2002. Distance coordinates used in geometric constraint solving. *Automated Deduction in Geometry*, 216–229.