An implementation of Jurzak's prover

Dominique Michelucci and Jean-Paul Jurzak

Burgundy university, Dijon, France
Dominique.Michelucci@u-bourgogne.fr,Jean-Paul.Jurzak@u-bourgogne.fr

Abstract. This text reports on several implementations of the Jurzak's prover of geometric theorems, and how this prover, which is initially intended for pedagogical purposes, can be and should be used in geometric constraints solvers of CAD/CAM geometric modellers.

1 Jurzak's prover

Usually, the most powerfull and sophisticated geometry provers are based on some computer algebra machinery such as Grobner bases or triangular sets [2]. After Dieudonné and probably other mathematicians, but in a more effective (*i.e.* computational) way, Jurzak [4] remarked that numerous theorems of affine or projective geometry (Pappus', Pascal', Désargues', Beltrami's theorems) can be proved straightforwardly, *i.e.* 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 2).

1.1 The principle

The typical session in Fig. 1 illustrates the proof of Désargues' theorem, with a user point of view. See Fig. 5, 2, 3, 4 for the proofs of Pascal, Pappus, hexamy, and Beltrami' s theorems.

To prove Désargues'theorem, the program defines three independent points e, a, 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 define c as $c = e + c_a \times \overrightarrow{ea} + c_b \times \overrightarrow{eb}$. 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 $\mathbb{Z}/p\mathbb{Z}$, p 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 $c_a, c_b \ldots$, *i.e.* they are ratios of polynomials in $c_a, c_b \ldots$; these polynomials have typically rational or integer coefficients.

```
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"                    .
```

Fig. 1. Session for the proof of Désargues' theorem. In 2D, if two triangles abc and $a'b'c'$ are perspective wrt a point e (in other words, eaa', ebb' and ecc' are collinear), then the three intersection points between homolog sides $ab \cap a'b'$, $ac \cap a'c'$ and $bc \cap b'c'$ are collinear.

```
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";;                            .
```

Fig. 2. 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. Below: we see what is behind the lie_on function.

In all cases, the coefficients in the linear combinations lie in some field, from now on 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 $c = e + c_e\overrightarrow{ea} + c_b\overrightarrow{eb}$ is replaced by $c = e + 3\overrightarrow{ea} + 4\overrightarrow{eb}$.

```
let a= point "a"
let b= point "b"
let c= lie_on [a; b]
(* we need another pure point, say z (we should 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]]]
is_an_hexamy p0 p1 p2 p3 p4 p5  (* true, of course, by construction *)
if is_an_hexamy p1 p0 p2 p3 p4 p5 (* true, because of Pascal's theorem *)
then Printf.printf "hexamy is proved\n"
else Printf.printf "hexamy is wrong :(\n"                                .
```

Fig. 3. An hexagon is an hexamy if its opposite sides cut in 3 collinear points. All permutation of an hexamy is an hexamy. This form of Pascal's theorem involves only lines. It is due to Raymond Pouzergues [8].

In the case $\mathbb{K} = \mathbb{Q}$, 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: $p + x \times \overrightarrow{pq} + y \times \overrightarrow{pr} = a + z \times \overrightarrow{ab}$ with unknowns x, y, z. Replacing p, q, r, a, b with their linear combinations of some pure points $b_1, \ldots b_4$ 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, divison, 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 $X = [c_a, c_b, \ldots]$ is the set of independent parameters (c_a, c_b, etc). The word "independent" means that there is no equation relating the parameters. Actually, it is the main reason of the simplicity of the Jurzak's method: since there is no algebraic equation constraining the parameters, no sophisticated algorithm from computer algebra (Grobner 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, the difference, the product, and the division between $r_1(x_1, x_2 \ldots) = n_1(x_1, x_2 \ldots)/d_1(x_1, x_2 \ldots)$ and $r_2(x_1, x_2 \ldots) = n_2(x_1, x_2 \ldots)/d_2(x_1, x_2 \ldots)$ where n_1, d_1, n_2, d_2 are polynomials in $x_1, x_2 \ldots$ 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 $\mathbb{K} = \mathbb{Q}[X]$.

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

Fig. 4. 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"                          .
```

Fig. 5. 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 $B(t)$.

1.2 Empty implementation for $\mathbb{K} = \mathbb{Q}[X]$

In the zero-th implementation, $\mathbb{K} = \mathbb{Q}[X]$ is not implemented! Only $\mathbb{K} = \mathbb{Q}$ is available (ocaml, the language used here, provides a library for arithmetic in $\mathbb{Q}$). Of course, we can do no symbolic computation with parameters, we can just verify the theorem at hand with some random value for each parameter. This seemingly stupid implementation, or no-implementation, is actually the best one, as we will ironically conclude.

1.3 Classical implementation for $\mathbb{K} = \mathbb{Q}[X]$

Elements in $\mathbb{K} = \mathbb{Q}[X]$ are classically represented by a couple 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 couple 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 computations, the degrees of polynomials increase quickly, and fills the memory space. Actually, the proof of Beltrami's theorem fails because of unsufficient memory. A solution is to simplify the ratios, but it needs complicated methods from computational algebra (gcd of polynomials). Note the same kind of problem occurs at the rational number arithmetic level: rational numbers should be reduced during computations to avoid memory failure.

1.4 DAG implementation for $\mathbb{K} = \mathbb{Q}[X]$

The DAG (Directed Acyclic Graph) representation (see for instance [9]) is used to solve the previous problem. Polynomials are no more represented by their list of monomials, but by a tree 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 [10] is used: the DAG is evaluated (in time proportionnal 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).

Of course, to reassure anxious and paranoiacs, and to gain more confidence, this stochastic test can be repeated several times. Note 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 of the degree, say d, 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's book) 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 probabilist) has an exponential cost.

Another optimization is to compute not 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. Some care should be taken to not evaluate several times shared nodes. These considerations yield to the third implementation.

1.5 Final implementation for $\mathbb{K} = \mathbb{Q}[\boldsymbol{X}]$

To get the third implementation, it suffices to realize that the DAG implementation is just a complicated way to use the zero-th implementation! We can simply run several times the session of the proof with different random numerical values (in $\mathbb{Q}$ or in some $\mathbb{Z}/pZ$ field) for parameters at each time. We can also represent each parameter by an array of k (say $k = 10$) numbers, and performs all computations "in parallel" with this array of examples. Note also each of the k examples can live in a specific field, $\mathbb{Q}$ or some finite field. For simplicity, we use $k = 10$ examples in the same finite field $\mathbb{Z}/32573\mathbb{Z}$. 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 on some screen.

The probabilistic paradigm is also used in the qualitative study of systems of geometric constraints [5]: 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. More on that in section 2.

1.6 Discussion

Even if Jurzak cleverly avoids the need for non linear algebra (gcd, elimination in non linear equations, etc), the daemon of complexity forbids to use deterministic and exact computations in $\mathbb{Q}[X]$ for complicated theorems. Instead some probabilist 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 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 $\mathbb{Z}/pZ$, is needed. It has been proposed by Denis Bouhineau in his PhD thesis [1], and by D. Michelucci [6]. These arithmetics permit to check geometric theorems on examples, thus to probabilistically prove geometric theorems involving circles (for instance: the 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.

2 Why solvers should use Jurzak's prover

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

Geometric constraints solvers assume that constraints (algebraic equations) are independent: actually, they fail in presence of dependent (contradicting or redundant) constraints. Of course, a qualitative study [5] of the system to be solved is first performed, before solving, to detect the more obvious errors (too much or not enough equations, etc), or to decompose it into smaller subsystems when possible. This qualitative study uses some graph machinery and/or the numeric probabilistic method (NPM for short) this paper just alludes above. The graph based methods are combinatorial in nature and can not detect even trivial algebraic dependences (for instance $f(x, y) = (2 \times f)(x, y) = 0$).

The numeric probabilistic method is here more powerful than the graph based approach. The NPM chooses random values (*i.e.* generic values) for the unknowns and studies with linear algebra (for instance Gauss method) the rank and the structure of the jacobian at this random configuration. For instance, for square (there is as much equation as unknowns) systems, the NPM detects when the jacobian is identically zero: this proves the system is redundant or contradictory. For non square systems, the NPM can still computes some base of the lines (assuming a convention where lines are derivatives) of the jacobian, for some random values of the unknowns, and it will indeed detect present dependences, which unknowns are fixed by the system, and the remaining number of DOF (degree of freedom) for unknowns which are not. Unfortunately, the jacobian is studied only at random points, since the solution is unknown.

Regrettably, some dependences between equations do not make vanish the jacobian everywhere (or do not reduce its rank everywhere, in case of non square systems). Consider for instance the very simple examples: the first square system $xy - 1 = 0, y = 0$, with jacobian y, has no solution; the second square system $xy - y = 0, y = 0$, with jacobian y, has a continuum of solutions. The NPM can not detect this kind of dependences between equations.

Regrettably, constraints systems met in CAD/CAM contain too often constraints with this kind of dependences, which the NPM is unable to detect. We conjecture that these dependences are due to incidence constraints (collinearites, coplanarities), either explicitly specified in the constraint system, or forced by geometric theorems from affine or projective geometry (Désargues, Pappus, hexamys, etc). The previous section just shows that Jurzak's prover detects very efficiently such hidden dependences (collinearites, coplanarities) induced by theorems. The core idea is thus to apply the NPM to a configuration which respects the explicit, specified "projective constraints" (collinearities, coplanarities) – and which will also respect incidence constraints due to geometric theorems

(Désargues, Pappus, hexamys, etc). The NPM probably works much better at such configurations.

Now, is it possible to find a configuration respecting the specified projective constraints? For CAD/CAM applications, very often indeed. It is due to the facts that systems are almost well constrained, and that there is a majority of metric constraints (for cotations of CAD/CAM parts, it is hardly a surprise), thus there are relatively few projective constraints. 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.

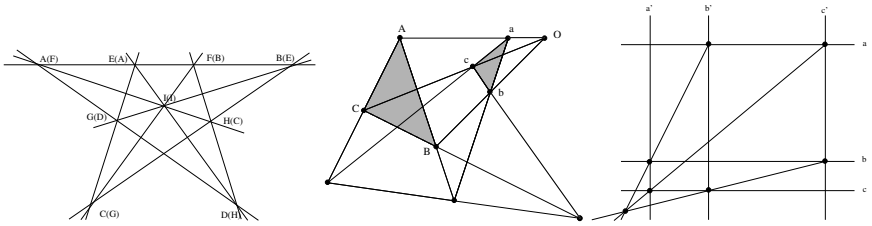

Fig. 6. Projective configurations: a non rational one, Désargues, Pappus.

Of course, the previous method of construction may fail in some cases. Fig. 6 shows a classical example, from Grunbaum's book. 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 Désargues's configurations, when the line of the conclusion is added, are other examples which make fail the greedy method, with this peculiarity that they become easily soluble if the conclusion line is removed, and, due to symmetry, all lines can be considered as the conclusive line.

Again, projective systems which make fail the greedy method are very unlikely in CAD/CAM but we can not resist the temptation to discuss this topic for the fun. Is it possible to decompose projective systems, as we do for rigid systems with Owen's method [7] or the HLS (Hoffmann, Lomonosov, Sitharam [3]) method? Note that a "projectively well constrained" part has 8 remaining DOF in 2D, while a rigid part has 3 remaining DOF in 2D as 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 Désargues'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 pos-

sible to built a dictionary of "atomic" projective configurations, and to recognize quickly them? Actually, we promote here the study of projective constraints.

3 Conclusion

This paper has shown how a simple prover, easy to implement, can very efficiently detect forced collinearities or coplanarities, which are 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.

Finally, if the numerical probabilistic method is used not at a random configuration, but at a configuration which fulfills at least the explicit projective constraints (collinearities, coplanarities) of the system to be solved (and thus the induced projective constraints induced by theorems of affine or projective geometry), then it gives more relevant diagnosis.

This poses the question of solving the projective part of systems of geometric constraints. For CAD/CAM applications, a greedy obvious method almost always solve the projective part, because the latter is under constrained. However the question of decomposing and solving projective systems (only incidence constraints) is funny and deserves further study.

Finally should we consider (and how) other incidence constraints, to circles or conics in 2D, to spheres or quadrics in 3D, to cubics?

References

1. D. Bouhineau. *Constructions automatiques de figures géométriques et programmation logique avec contraintes*. PhD thesis, Université Joseph Fourier, Grenoble, 1997.
2. S.-C. Chou. *Mechanical Geometry theorem Proving*. D. Reidel Publishing Company, 1988.
3. C. M. Hoffmann, A. Lomonosov, M. Sitharam, Decomposition Plans for Geometric Constraint Systems, Part I: Performance Measures for CAD, *Journal of Symbolic Computation*, 31:367-408, 2001, Academic Press.
4. Jean-Paul Jurzak. Géométrie vectorielle algorithmique : La géométrie vectorielle par l'informatique. Web site http://passerelle.u-bourgogne.fr/publications/webgeometrie/, Burgundy University, Dijon, December 2003.
5. H. Lamure and D. Michelucci. Qualitative study of geometric constraints. In *Geometric Constraint Solving and Applications*. Springer-Verlag, 1998.
6. Dominique Michelucci. The robustness issue. Unpublished draft. *http://www.emse.fr/~micheluc/robustness.ps.gz*, 1998.
7. J.C. Owen. Algebraic Solution for Geometry from Dimensional Constraints In *Solid Modeling*, 397-407, 1991.
8. R. Pouzergues. Les hexamys (in french). *available on the web http://hexamys.free.fr/accueil.htm*
9. A. Rege. A complete and practical algorithm for geometric theorem proving. In *ACM Symp. Computational geometry*, 1995.
10. J.T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 4(27):701–717, 1980.