A particle-spring approach to geometric constraints solving

Simon E.B. Thierry
LSIIT, UMR CNRS-UdS 7005
Université de Strasbourg
simon.thierry@unistra.fr

March 2011

Abstract

Current iterative numerical methods, such as continuation or Newton-Raphson, work only on systems for which the corresponding matrix is a square one. The geometric constraint systems need thus either to have no degrees of freedom, or to be a system the software can anchor, *i.e.* a rigid system.

In this article, we propose a new iterative numerical approach which can handle both rigid and under-rigid geometric constraint systems. It is based on the translation of the system under the form of a particle-spring system where particles correspond to the geometric entities and springs to the constraints. We show that consistently over-constrained systems are also solved.

We show that our approach is promising by giving results of a prototype implementation. We propose tracks for enhancements of the approach which could tackle its drawbacks (mainly stability).

Keywords : Geometric constraints solving, Mass-spring system, Numerical iterative computation

1 Introduction

Geometric constraint solving is a key functionality in Computer-Aided Design (CAD) software. The basic idea is to solve constraints of distance, angle, incidence, tangency, *etc.* applied to geometric elements such as points, lines, circles, planes, spheres, *etc.* A Geometric Constraint System (GCS) is a set of such constraints, generally given under the form of a technical sketch, on which the user interactively places the constraints. A solution of a GCS is a set of coordinates of the geometric elements (a *figure*) which satisfies the constraints. A GCS with a finite number of solutions is said to be well-constrained. If it allows flexions, it is said to be under-constrained. When it has no solutions, it is over-constrained.

Mathis and Thierry [20] give formal definitions of GCS and of their resolution and decomposition.

The literature contains many different approaches of the resolution of GCS, which can be roughly classified in four families: rule-based methods perform explicit geometric deductions with expert systems; graph-based methods consider the constraint graph, where a node represents a geometric element and an edge a constraint, and compile the geometric knowledge under the form of combinatorial rules; symbolic methods consider the underlying equations and solve the equation system, forgetting the geometric nature of the problem; numerical methods also translate the GCS into an equation system, but then use iterative computations to approximate the solutions. Whatever the approach is, a general trend in the last two decades has been to decompose the system [14], in order to lower the complexity of the resolution as well as to enhance the resolution power. For a more complete view of the geometric constraint solving field, the reader may refer to some surveys [13, 15].

Numerical methods are of primary importance for an industrial software, because they are complete: they are not limited to a certain class of systems and are not sensitive to geometric theorems which the developpers did not take into account. They may succeed for GCS other methods fail to solve. But current numerical methods only work on rigid systems. When they are able to handle under-constrained systems, they do it at the expense of speed.

Yet, under-constrained systems are important for interactive and intuitive solvers. Non-expert users cannot be expected to design a well-constrained system, since it is easy, especially with large constraint systems, either to add a redundant constraint by failing to realize that a part is already rigid, or to leave some parts articulated though the intent was that they be rigid. Moreover, being able to solve any system (if it has solutions) is necessary to give feedback to the user. For instance, interactive theorem provers used for geometric proofs cannot yet have a drawing feature, which would help the user better understand the current situation.

In this article, we propose a new approach to numerically handle geometric constraint systems by considering them as particle-spring systems (known also as mass-spring systems). Particle-spring systems are widely used in computer graphics to simulate the behaviour of deformable objects: muscles [22], cloth [24], hair [26], surgery tools [16] or face expressions [30], among many others. We propose to use them to find approximate solutions of geometric constraint systems: geometric elements become particles and constraints become springs. We explain the simple implementation we made and show that it gives satisfactory results. We show that it can solve under-constrained and well-constrained systems alike. We show that consistently over-constrained systems (*i.e.* systems which are generically over-constrained but yet have solutions) are in general solved more quickly. Non-consistent over-constrainedness, though, is shown to be hard to detect.

The rest of this article is organized as follows. Section 2 reviews related work by detailing the existing numerical methods. Section 3 details how we build a particle-spring system from a geometric constraint system and how we

compute its iterative states. Section 5 gives practical examples of several geometric constraint systems and of their resolution. Section 5.2 elaborates on the specific case of consistently over-constrained systems. Section 5.3 gives quantitative results which show that our approach, though naively implemented, is satisfactory. Section 6 concludes and gives perspectives.

2 Numerical solving methods

Numerical methods look for an approximate solution of system $F(X, U)$, with X the set of geometric elements (the unknowns) and U the set of metric values, such as distances and angles (the parameters). F is the equation system corresponding to the GCS. The best-known and most commonly used numerical method is the Newton-Raphson method [29]. It consists in approximating F by its tangent hyperplanes when searching for a root: from an initial figure f_0 and parameters u, it consists in approaching a root of F by computing the series $f_{n+1} = f_n - F'(f_n, u)^{-1}F(f_n, u)$ until a sufficiently near-zero figure is found. It has convergence issues (see [17, Fig. 1]) and its attractions basins are fractals, which may cause it to be counter-intuitive(see [17, Fig. 3]).

Another commonly used method is homotopy, also known as continuation. Introduced in the field of geometric constraint solving by Lamure and Michelucci [17], it was used by various authors [7, 8]. For given values of the parameters, the method considers the function $H(X, t) = t \times F(X) + (1 - t) \times (F(X) - F(f_0))$. It induces a linear interpolation between $H(X, 0) = F(X) - F(f_0)$, which is zero for $X = f_0$, and $H(X, 1) = F(X) = 0$. The continuation method consists in following the curves defined by the equation system $H(X, t) = 0$, from $t = 0$ and $X = f_0$ to $t = 1$. More details on continuation methods can be found in [3].

Other purely numerical methods were proposed but are not often used [1, 4, 19, 23]. Hybrid methods were proposed, combining numerical iterations with graph-based or rule-based reasoning: Schreck *et al.* [25] describe a multi-agent system where numerical methods are used when other formal solvers cannot solve the system. Lee *et al.* [18] on the one hand, Ait-Aoudia *et al.* [2] on the other hand, enhance their graph-based method by performing numerical computations for solving steps which are not feasible combinatorially. Likewise, Fabre and Schreck [9] extend the work of Gao *et al.* [10] to solve quasi-indecomposable systems: they remove a set of constraints so as to be able to decompose the system and replace them by an equivalent number of new constraints; then they use Newton-Raphson iterations to change the values of the parameters of the new constraints in order to satisfy the previously removed constraints.

All of these methods require the system to have as many variables as unknowns, *i.e.* the system must be generically rigid so that we can add three/six equations in 2D/3D (these additional equations anchor the system in the plane/space). If the number of variables differs from the number of equations, special techniques must be used, which are costly [17].

Methods able to numerically handle under-rigid systems are not many: Ge *et al.* [11] consider the sum of squares of the different equations and then test two

optimization methods. An evolutionary approach was proposed by Cao *et al.* [5] but the results are not yet satisfactory. Genetic algorithms are used for classical (*i.e.* non-geometric) constraint solving [6], but none of these methods are specific to geometric constraints.

3 Particle-spring systems and geometric constraints

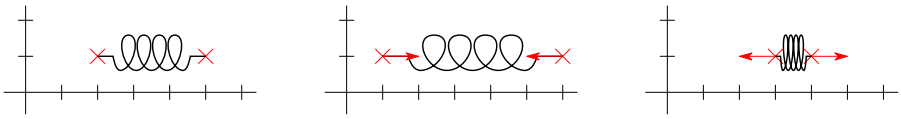
A geometric constraint system (GCS) consists in a set X of geometric elements (the unknowns), a set U of metric values (the parameters) and a set C of constraints. We note $G = (C, X, U)$. The goal of a geometric constraint solver is to yield valid figures, that is, for a valuation of U, a valuation of each element of X such that the constraints of C are satisfied.

A particle-spring system consists in a set P of particles, with no mass, and a set R of springs, each spring being linked to two or more particles. It can be represented as a graph (or a hypergraph for springs linked to more than two particles). We thus note $S = (P, R)$. Each spring has a (possibly infinite) set of stable states, according to the relative positions of the associated particles. When it is not in a stable state, a spring applies forces on its particles, pushing or pulling them towards one of its stable states. A particle-spring system is said to be in a stable state if for each particle $p \in P$, the sum of the forces applied on p by the springs which are not in a stable state is 0. This happens when all springs are in a stable state or when the forces applied by the springs cancel each other.

There are mainly two ways to represent a particle-spring system [27]: explicitly, each iteration consists in computing the forces that the different springs apply on the particles and displacing the particles accordingly ; implicitly, the differential equations of the particles displacements are considered and solved. Implicit representations yield more stable techniques but are less intuitive. Because our goal was to build a prototype and see if the particle-spring approach can be a satisfactory solving technique, we considered an explicit representation, which is easier to implement.

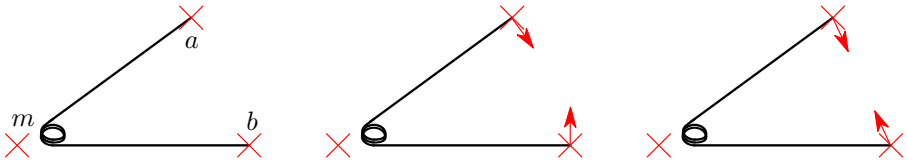
The particle-spring system $S = (P, R)$ associated to a GCS $G = (C, X, U)$ is built naturally by transforming each $x \in X$ into a particle $p \in P$ and by transforming each constraint $c \in C$ into a spring $r \in R$. The stable states of a spring r associated to a constraint c are defined as the states where the position of the particles of r satisfy c.

We give here examples of how to transform constraints into springs. Distance constraints are the most straightforward: they are associated with classical helical springs. A helical spring is in a stable state when the distance between its particles is exactly the metric of the corresponding constraint. If the distance between the two particles is bigger, the helical spring pulls the particles towards each other. If the distance is shorter, the spring pushes the particles apart. Figure 1 illustrates those three cases with a helical spring corresponding to a distance constraint with a metric of 3. Said otherwise, if there is a distance constraint with metric k between points p_1 and p_2, the corresponding helical



a **b** **c**

Figure 1: Helical spring corresponding to a 3-distance constraint: stable state (**a**) and two unstable states with the corresponding forces applied to the particles (**b** and **c**).



a **b** **c**

Figure 2: Torsion spring (**a**) and two ways to simulate its action: tangential displacement of a and b (**b**) ; replacement by a helical spring between a and b (**c**).

spring applies a force of $\overrightarrow{p_1p_2} \times \frac{k-|\overrightarrow{p_1p_2}|}{|\overrightarrow{p_1p_2}|} \times d$ on p_1, d being a damping factor (see section 4).

We take angle constraints into account by torsion springs. In our prototype, we did not explicitly consider "line particles" and define a line by two points. Thus, angles are between three points. Figure 2**a** illustrates a torsion spring in a stable state. There are several ways to simulate the action of a torsion spring corresponding to a constraint on angle $\widehat{amb}$:

1. the physics-inspired way is to apply a force on a (resp. b) which is orthogonal to $\overrightarrow{ma}$ (resp. $\overrightarrow{mb}$), *i.e.* simulate a displacement along the tangent to the m-centered circle with radius $|\overrightarrow{ma}|$ (resp. $|\overrightarrow{mb}|$) ; it is represented on figure 2**b**,
2. another classical way is to simulate the action of the torsion spring with a helical spring between a and b ; it is represented on figure 2**c**,
3. a force can be applied on particle m along the angle bisector,
4. the torsion spring can be simulated by two helical springs m-a and m-b.

Of course, hybrid ways can be considered. Choosing a way gives the directions of the force vectors applied on the particles. Whatever way is chosen, their norm is computed according to the law of cosines:

$$|\overrightarrow{ab}|^2 = |\overrightarrow{ma}|^2 + |\overrightarrow{mb}|^2 - 2|\overrightarrow{ma}||\overrightarrow{mb}|\cos(\widehat{amb})$$

To our knowledge, there is no real spring corresponding to incidence and tangency constraints. We transform these constraints into helical springs with

gliding anchor points. For instance, a point-line incidence constraint corresponds to a helical spring between the point and its orthogonal projection on the line. This spring has a zero-distance stable state.

We also considered circles, with a center particle and a radius. Tangency constraints are then also zero-distance helical springs between the center of the circle and its orthogonal projection on the line, together with a virtual helical spring between the center of the circle and its perimeter.

4 Iteration algorithm

The iteration algorithm we considered is straightforward, since we use explicit representation of the springs. At each step, a loop considers each spring. Each spring computes the forces to be applied on each of its particles. After this loop, a second loop considers each particle, in order to sum the forces and apply them on the particle. Algorithm 1 gives the pseudo-code for this iterative process.

Alg. 1: One iteration step of the solver

Input: $S = (P, R)$: a particle-spring system
Result: S': S after one iteration step of the solver

foreach *spring* $r \in R$ **do**
  foreach *particle* p *linked to* r **do**
    $\overrightarrow{f} \leftarrow$ force vector applied by r on p
    Store vector $\overrightarrow{f}$ in p
foreach *particle* $p \in P$ **do**
  $\overrightarrow{v} \leftarrow \Sigma_{\overrightarrow{f} \in F}(d \times \overrightarrow{f})$, with
    - F the set of force vectors stored in p
    - d the damping factor (see below)
  $p \leftarrow p + \overrightarrow{v}$
return S

In section 3, we explained how to compute the direction of the forces applied on the particles. It also is easy to compute the norm of these vectors if one is to put the particles in the right place in only one step: for distance constraints, for instance, the norm of each vector is half of the error between the actual distance and the constraint distance. If one were to do this, however, it would result in great instability when different forces are applied in the same global direction, moving a particle beyond the wanted position.

To avoid this problem, a damping factor must be used. In order to make sure that this damping factor is small enough to prevent a particle from going beyond the point where the forces reverse, we use a $\frac{1}{n}$ damping factor, where n is the highest number of springs linked to a same particle.

We do not consider the kinetic energy of the particles, since they have no

mass. This means that, due to the damping factor, when there are two springs or more, we cannot reach an exact solution and can only tend to a zero-error.

It is possible for the user to ask that some particles do not move. If a particle p is anchored, the forces applied by the springs linked to p must me modified accordingly, so that p does not move, and the other particles move more.

After each step of algorithm 1, we compute the error of the system. We consider the minimal error (the error on the spring which is nearest to a stable state) and the maximal error. We also consider the mean error and the root mean square (RMS) error. To compute the error on torsion springs, we normalize the angle values to the largest distance constraint metric. These values cannot be measured as an error ratio, due to incidence and tangency constraints.

This leads to several possible stopping conditions, according to the different error statistics and to the user's will. If a very precise figure is needed, the user may want the solver to stop only when the maximal error is below a given small threshold. If the user only needs a rough idea of what a solution looks like, a small RMS error is enough.

Due to the possibility of instabilities, we also consider two other stopping conditions: reaching a given amount of iterations, and reaching a stable state without having reached a satisfying error value. To identify the latter, we compare the error modification of each spring after a step of algorithm 1. If the error modifications are all below a given ε, we consider the system to have reached a stable state. Note that the system may actually be globally moving, if the forces applied by the different springs define a rigid motion.

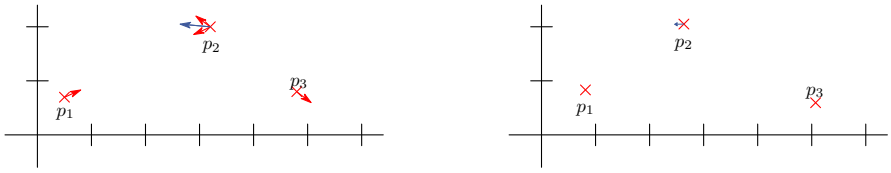
Algorithm 2 gives the pseudo-code of the overall solver.

Alg. 2: Particle-spring geometric constraint solver

Input:
  $G = (C, X, U)$: a geometric constraint system
  X_0: initial figure (valuation of the unknowns)
Output:
  X_s: approximate solution
  b: boolean indicating if the solver succeeded

$S = (P, R) \leftarrow$ particle-spring system corresponding to G and X_0
$e \leftarrow$ error statistics
$i \leftarrow 0$
while *e is not satisfying* **do**
  $S \leftarrow$ solving step using algo. 1
  $i \leftarrow i + 1$
  $e \leftarrow$ error statistics
  if *modifications of e are too small* **or** *i is too high* **then**
    return P, **false**
return P, **true**

The complexity of the algorithm is as follows: algorithm 1 works in $\mathcal{O}(|p| +$



a **b**

Figure 3: One resolution step with 2 springs

$|r|$) since it traverses each spring once and each particle $n+1$ times at most, n being the maximal number of springs linked to a particle, *i.e.* a small constant. Algorithm 2 uses algorithm 1 and computes errors at each iteration. Computing errors means traversing each spring and is thus in $\mathcal{O}(|r|)$. The overall complexity of each iteration of algorithm 2 is thus $\mathcal{O}(|p|+2|r|)$. Since the number of particles is similar to the number of springs, the complexity of the particle-spring solver is $\mathcal{O}(3|p|)$, *i.e.* $\mathcal{O}(|p|)$.

5 Practical examples

We give here practical examples of how our particle-spring prototype solver behaves. We then focus on the specific cases of torsion springs and detail the consequences of redundant constraints.

Let us consider a GCS with three points $p_1 \dots p_3$ and two distance constraints: the distance between p_1 and p_2 is constrained to be 2, and the distance between p_2 and p_3 must be 3. In this example, the damping factor is $\frac{1}{2}$, 2 being the maximal number of springs attached to a single particle. On figure **3a**, the initial distance between p_1 and p_2 is 3 and the distance between p_2 and p_3 is 2. The first helical spring thus applies on p_1 a force directed towards p_2 with a norm of 1 (error to the constrained value) $\times\frac{1}{2}$ (the force is shared among two particles) $\times\frac{1}{2}$ (damping factor) $= \frac{1}{4}$. It applies a symmetric force on p_2. The second helical spring applies a force vector of norm $\frac{1}{4}$ on particle p_2, pushing it apart from p_3, and a symmetric force on p_3. Those four forces are shown in red on figure **3a**, the blue arrow representing the sum of the forces applied on p_2.

After applying these forces and displacing the particles, we obtain the positions shown on figure **3b**. The new distance between p_1 and p_2 is approximately $\frac{11}{5}$ and the new distance between p_2 and p_3 is approximately $\frac{14}{5}$. The new force vectors have a norm of 0.1 (spring p_1–p_2) and 0.09 (spring p_2–p_3). Only the sum of the forces applied on p_2 is shown, since the other ones would be too small to be visible on the figure.

Systems containing only distance constraints are very satisfyingly solved. For instance, the system represented on figure 4 leads to a maximal error of less than 10^{-4} in about 200 solving steps with random initial values, in about 150 solving steps with an initial solution taken from a user sketch. Note that it is an under-rigid system.

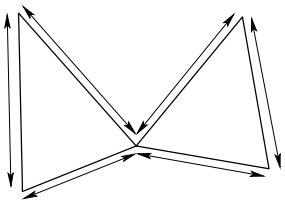


Figure 4: Articulated GCS with two rigid triangles

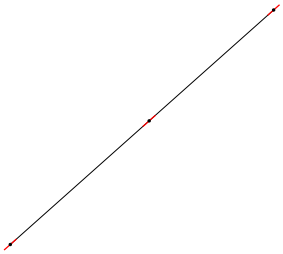


Figure 5: Stable non-solution system not respecting triangular inequality

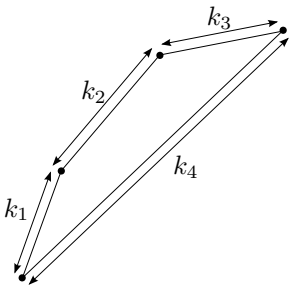


Figure 6: Implicit incidence constraint when $k_4 = \Sigma_{i=1}^{3} k_i$

It quickly leads to a stable non-solution state in the case of a triangle which does not respect the triangular inequality. Figure 5 shows the stable state obtained, with the applied forces represented in red. It succeeds to find a solution with a maximal error of 10^{-5} for a system made only of incidence constraints and representing a sketch of the Pappus theorem, in about 200 iterations.

It takes, however, a long time to get a satisfying solution in cases where distance constraints lead to an incidence: figure 6 shows such a system. Since the forces applied on middle points, are directed towards the other points, the closer the point gets to the biggest segment, to smallest the force attracting it gets. It takes our prototype 3000 iterations to get from a 7×10^{-3} maximal error to a 10^{-3} maximal error on this system.

Besides, our prototype finds a solution to the ten spheres problem (see [17, Fig.1]) for initial values where the Newton-Raphson method diverges. It gets a 10^{-3} maximal error in 600 iterations and a 10^{-6} maximal error in 800 iterations.

5.1 Torsion springs

Angle constraints are the weakest point of our prototype, since we could not find a generally satisfying way to simulate torsion springs, among the ones cited in section 3. Indeed, using way 4 (replacing the torsion spring by two helical springs $a - m$ and $b - m$) leads to a quick resolution of the system of figure 7. The three other ways lead either to unstable state or to stable non-solution states. On the other hand, the system of figure 8 quickly converges towards a solution with the three other ways but leads to a stable non-solution state with way 4 or with any hybrid way partially using it.

5.2 Consistent over-constraints

Unlike most solving methods, our solver accepts consistent over-constraints: the corresponding springs apply forces which are consistent with the other forces. Actually, consistent over-constraints may even lead to more precision, at the

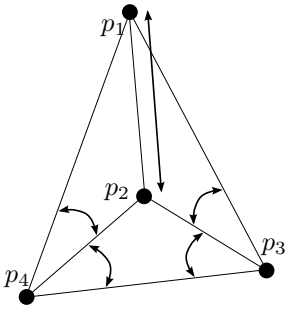
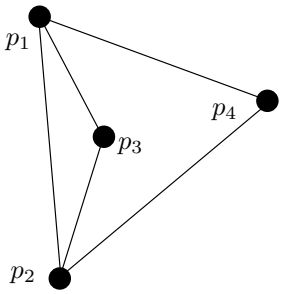


Figure 7: Rigid system solved with way 4 for angle constraints; sketch (left) and initial values (right)



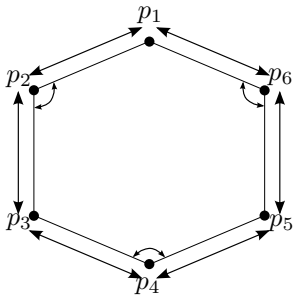


Figure 8: Rigid system solved with any way but way 4

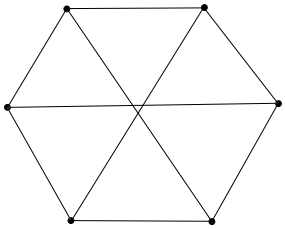


Figure 9: Rigid undecomposable 2D system

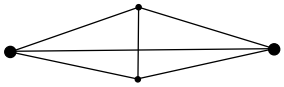


Figure 10: Addition of redundant constraints

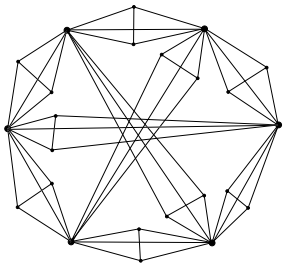


Figure 11: Highly redundant version of the GCS

possible extense of resolution speed. For instance, let us consider the 2D system of figure 9: it is made of six points and 9 distances. Our solver only succeeds to solve it with a maximal error of 10^{-3}. If we add, for each of the 9 constraints, a double-triangle (figure 10 shows how the distance constraint between the two thick points leads to the creation of two new points and 5 new distance constraints), leading to the system of figure 11, then the system is solved with a maximal error of 10^{-7}.

Likewise, even without adding new geometric entities, we noticed that adding consistent constraints leads to more precision. For instance, adding redundant constraints to a 3D system representing a Stewart platform [28] helped us reduce the maximal error from 10^{-2} to 10^{-6}.

This means that when using a particle-spring solver, the user needs not worry about addint too much information, whereas on classical solvers, it is necessary to detect redundancy and get rid of it.

Table 1: Number of iterations needed to reach given precisions

	Normal version		With redundancy	
	10^{-2}	10^{-6}	10^{-2}	10^{-6}
Fig. 4	89	252		
Fig. 9	353	*	902	8776
2D desklamp	153	*	253	7567
Pappus theorem	121	280		
4-connected GCS	2092	6054	2348	8035
Ten spheres	515	818		
3D cube	127	492	346	930
3D pyramid	96	343		
Stewart platform	724	*	1027	952

5.3 Quantitative results

Table 1 gives the number of iterations needed to reach given threshold error, with or without redundancy, for a series of geometric constraint systems. A star indicates that the software reached a stable state before reaching this precision, or that it reached 10 000 iterations. An empty case indicates we dit not try this configuration.

The systems mentioned in table 1 are the following ones:

- the 2D desklamp, the 3D cube and the 3D pyramid are classical examples,
- the "Pappus theorem" system consists in 9 points and eight incidence constraints of a point to a line passing by two points,
- the "4-connected GCS" system corresponds to the system of Fig. 9a of [21],
- the "ten spheres" system corresponds to the system of Fig. 1 of [17],
- the "Stewart platform" systems corresponds to a system as described in [28].

6 Conclusion and perspectives

We presented a new approach to solve geometric constraint systems, based on their translation under the form of a particle-spring system. We implemented a prototype, using an explicit representation of particle-spring systems. Though this leads to cases of instability, we gave results showing that this approach is promising and already satisfying for applications where a very small precision is not needed, for instance when the user only wants to get rough feedback on what the solutions look like.

It works on 2D and 3D systems, rigid or articulated. It accepts redundancy, which even leads to more precise results, yet at the expense of resolution

speed. Note that particle-spring systems can benefit from the high parallelization of GPU [12]. A disadvantage of the approach is that non-consistent over-constrainedness leads to unstable states, but since it is not the only source of unstable states, it cannot be detected.

We intend to further develop our prototype, by first adding other kinds of particles (lines, planes, spheres) and the corresponding constraints. We also intend to test an implicit representation, which would help solve the stability problems [27] we encounter with torsion springs.

Finally, we want to add features to our prototype, so that the user can interact with the solutions during the solving process, by moving particles. This way, the user can get a very effective feedback on the articulations of the system, by moving a point and seeing which parts of the system are modified.

References

[1] S. Ait-Aoudia. Numerical solving of geometric constraints. In *IV '02: Proceedings of the 6th international conference on Information Visualisation*, pages 125–129, London, England, United Kingdom, 2002.

[2] S. Ait-Aoudia, H. Badis, and M. Kara. Solving geometric constraints by a hybrid method. In *IV '01: Proceedings of the 5th international conference on Information Visualisation*, pages 749–753, London, England, United Kingdom, 2001.

[3] E. L. Allgower and K. Georg. Continuation and path following. *Acta Numerica*, 2:1–64, 1993.

[4] A. H. Borning. The programming language aspects of Thinglab, a constraint oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, 1981.

[5] C. H. Cao, W. H. Li, and B. Cong. The geometric constraint solving based on hybrid genetic algorithm of conjugate gradient. In G. R. Liu, V. B. C. Tan, and X. Han, editors, *Computational Methods*, chapter 17, pages 1117–1121. Springer, 2006.

[6] C. A. Coello Coello. Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 19(11-12):1245–1287, 2002.

[7] C. Durand. *Symbolic and numerical techniques for constraint solving*. PhD thesis, Purdue University, West Lafayette, Indiana, USA, 1998.

[8] C. Durand and C. M. Hoffmann. A systematic framework for solving geometric constraints analytically. *Journal of Symbolic Computation*, 30(5):493–519, 2000.

[9] A. Fabre and P. Schreck. Combining symbolic and numerical solvers to simplify indecomposable systems solving. In *SAC '08: Proceedings of the 23rd ACM Symposium on Applied Computing*, pages 1838–1842, Fortaleza, Brazil, 2008.

[10] X.-S. Gao, C. M. Hoffmann, and W.-Q. Yang. Solving spatial basic geometric constraint configurations with locus intersection. *Computer-Aided Design*, 36(2):111–122, 2004.

[11] J.-X. Ge, S.-C. Chou, and X.-S. Gao. Geometric constraint satisfaction using optimization methods. *Computer-Aided Design*, 31(14):867–879, 1999.

[12] J. Georgii and R. Westermann. Mass-spring systems on the GPU. *Simulation Modelling Practice and Theory*, 13(8):693–702, 2005.

[13] C. M. Hoffmann and R. Joan-Arinyo. A brief on constraint solving. *Computer-Aided Design and Applications*, 2(5):655–663, 2005.

[14] C. Jermann, G. Trombettoni, B. Neveu, and P. Mathis. Decomposition of geometric constraint systems: a survey. *International Journal on Computer Graphics and Application*, 16(5,6):379–414, 2006.

[15] R. Joan-Arinyo. Basics on geometric constraint solving. In *EGC '09: XIII Encuentros de Geometria Computacional*, Zaragoza, Spain, 2009. Oral presentation. Paper available at `http://metodosestadisticos.unizar.es/~egc09/index_archivos/Trabajos/robert.pdf`.

[16] T. Jund, D. Cazier, and J.-F. Dufourd. Edge collision detection in complex deformable environments. In *VRIPHYS '10: Proceedings of the Workshop on Virtual Reality Interactin and Physical Simulation*, Copenhagen, Denmark, 2010.

[17] H. Lamure and D. Michelucci. Solving geometric constraints by homotopy. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):28–34, 1996.

[18] K.-Y. Lee, O.-H. Kwon, J.-Y. Lee, and T.-W. Kim. A hybrid approach to geometric constraint solving with graph analysis and reduction. *Advances in Engineering Software*, 34(2):103–113, 2003.

[19] R. Light, V. Lin, and D. C. Gossard. Variational Geometry in CAD. *Computer Graphics*, 15(3):171–175, 1981.

[20] P. Mathis and S. E. B. Thierry. A formalization of geometric constraint systems and their decomposition. *Formal Aspects of Computing*, 22(2):129–151, 2010.

[21] D. Michelucci, P. Schreck, S. E. B. Thierry, C. Fünfzig, and J.-D. Génevaux. Using the witness method to detect rigid subsystems of geometric constraints in CAD. In *SPM '10: Proceedings of the 15th ACM Conference on Solid and Physical Modeling*, Haïfa, Israël, 2010.

[22] L. P. Nedel and D. Thalmann. Real-time muscles deformation using mass-spring systems. In *CGI '98: Proceedings of the 5th edition of Computer Graphics International*, pages 156–165, Hannover, Germany, 1998.

[23] G. Nelson. Juno, a constraint-based graphic system. *ACM SIGGRAPH Computer Graphics*, 19(3):235–243, 1985.

[24] X. Provot. Deformation constraints in a mass-spring model to describe rigid cloth behaviour. In *GI '95: Proceedings of the 14th edition of Graphics Interface*, pages 147–154, Québec, Canada, 1995.

[25] P. Schreck, J.-F. Dufourd, and P. Mathis. Using a numerical tool in a formal construction method with decomposition. In B. Brüderlin and D. Roller, editors, *Proceedings of the 2nd International Conference on Computer Graphics and Artificial Intelligence*, pages 211–233. Springer, Limoges, France, 1998.

[26] A. Selle, M. Lentine, and R. Fedkiw. A mass spring model for hair simulation. *ACM Transactions on Graphics*, 27(3):64.1–64.11, 2008.

[27] M. Shinya. Theories for mass-spring simulation in computer graphics: stability, costs and improvements. *IEICE Transactions on Informatics and Systems*, E88-D(4):767–774, 2005.

[28] D. Stewart. A platform with six degrees of freedom. *Aircraft Engineering and Aerospace Technology*, 38(4):30–35, 1966.

[29] T. J. Ypma. Historical development of the Newton-Raphson method. *SIAM Review*, 37(4):531–551, 1995.

[30] Y. Zhang, E. C. Prakash, and E. Sung. Real-time physically-based facial expression using mass-spring system. In *CGI '01: Proceedings of the 8th edition of Computer Graphics International*, pages 347–350, Hong-Kong, China, 2001.