30 Years after Sketchpad: Relaxation of Geometric Constraints Revisited

C.W.A.M. van Overveld
Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O.Box 513, 5600 MB, Eindhoven,
The Netherlands.
Email: wsinkvo@info.win.tue.nl

The problem of solving sets of coupled geometric constraints in the context of interactive computer aided sketching is studied. Some criteria for solution methods are given and the applicability of a relaxation method is discussed. Despite its relatively slow convergence, the relaxation method turns out to be sufficiently flexible (it works for a variety of constraint types and it can be implemented to handle both under- and over-constrained systems in a stable way) to be of at least qualitative use in an interactive design environment. Some mathematical aspects of the method are discussed and examples of applications are given.

1. Introduction; background; related work

Since the introduction of the Sketchpad system by Sutherland as early as 1963 [16] the notion of constraints has played an important role in computer graphics and computer aided design. Applications include the design of mechanisms [14], user interfaces [2] and [3], interactive dynamics [11, 17, 18], 2-D graphics design [7] and [10], and the combination of 3-D geometric design and mechanism simulation in the LEGO-system [6].

In all these applications, geometric constraints take the form of coupled non-linear equations and inequalities. Unfortunately, quoting the authors of "Numerical Recipes in C" ([13], page 379), "There are *no* good, general methods for solving systems of more than one nonlinear equation and [...] there *never will be* any good general methods".

Rather than conclude from this adage that the topic of constraint based graphics should be given a rest, all authors above have attempted to tackle the problem of dealing with coupled nonlinear equations in broader or narrower sense.

In fact, the pre-computer era already has seen a considerable amount of activities in this field: R.V. Southwell used a technique called "relaxation" in the late 1930-s and early 1940-s for a variety of engineering problems [4]. This technique was essentially inspired by a mechanical device, consisting of a network of inextensible strings which should be brought in a state of equilibrium where all strings were completely stretched. A numerical scheme based on this "analog computing device" has been used by Southwell to solve discretised approximations of partial differential equations; it can be seen to be a forerunner of finite element methods.

In his 1963 paper Sutherland describes a method to cope with constraints which very much resembles the Southwell approach; he suggests to relax each one of the constraints in succession until a tolerance criterion is met (in the Southwell technique, all constraints are relaxed simultaneously). He also observes that the method is "robust but slow", but gives no further quantitative analysis.

With the increase of interest in computer solutions for numerical problems in the 1960-s and 1970-s, several new disciplines of numerical analysis new inpulses [15, 13, 5]. In this way, both the problems of solving one non-linear equation and the problem of solving sets of linear equations have received ample attention. For the first type of problems the Newton-Raphson method is a general approach which gives quadratic convergence. The second type of problems allows a variety of canonic approaches: both Gauss-elimination with pivoting and iterative approaches (such as Gauss-Seidel or Gauss-Jacobi, using the same matrices throughout the iteration process or the conjugate-gradient method, which adjusts the direction of iteration underway) are often used. The latter method, the conjugate-gradient method, is a variation of the steepest descent method, or gradient method, and this allows application in the more complex setting of systems of non-linear equations as well. It is implemented very straightforwardly, but it has merely linear convergence. For this reason, much of the recent work on geometric constraints in computer graphics and computer aided design uses the multi-dimensional Newton Raphson method for its better asymptotic convergence (quadratic rather than linear). The price to pay for this improved convergence, however, is that at every iteration step, a (large) set of linear equations has to be solved, typically using one of the methods mentioned above, optionally adapted to deal with sparse matrices or block-matrices. So a single step of a Newton-method is much more expensive than a single step of a linear method, but to achieve a given tolerance ϵ, only fewer Newton-steps are needed[1]. Given the availability of these methods from numerical analysis, a new wave of interest in constraint resolution in computer graphics could be seen in the late 1970-s and 1980-s.

In 1985, PAVLIDIS et al. [10] introduced a system for beautifying hand-sketched drawings by imposing constraints (e.g. line segments should be parallel or perpendicular, their should be no gaps between adjacent line segments,

[1] provided ϵ is sufficiently small: for an arbitrary ϵ, a quadratic method does not necessarily have to be faster than a linear method.

etc.). In the system, constraints were dealt with in a locally linearized fashion and the coupling between the constraints was taken care of by means of solving a set of coupled linear equations. Also in 1985, NELSON [7] published a paper on the Juno system, based on a declarative language for specifying constraints for 2-D drawings. The constraints in Juno were solved by means of Newton-Raphson-based global error minimization. In 1986, BORNING and DUISBURG [2, 3] reported on the application of constraint solving in the context of user interface design.

At the same time, efforts took place in the mechanical engineering community to combine the above techniques for constraint resolution with known analytic results from the theory of multi-link systems and their kinematics to build systems for computer-assisted mechanism synthesis [8] and even interactive real-time simulation of planar linkages [14].

With the introduction of physics-based animation techniques in the late 1980's, the issue of constraints received renewed attention. Indeed: the details of mechanical motion are often dictated by constraints, and a central issue in mechanical simulation is computing the reaction forces introduced by these constraints. WITKIN et al. [17] consider the path of a dynamical system in parameter space which develops subject to global energy minimization. This means that at every subsequent time step a non-linear optimisation step (steepest descent search) has to be performed in order to find the "cheapest" way for the system to proceed. BARZEL et al. [1] apply a technique to remove the algebraic constraint equations by means of differentiating them in order to arrive at a set of evolution equations only. These are linearized at every time step and the linear sets are solved.

In 1990, WITKIN et al. [18] again published results on constraint satisfaction, this time achieved by recursively traversing a tree of constraint-dependencies in order to arrive at the independent coordinates and next to compute -in reverse order- the values for the dependent coordinates. Their system has been reported to work at interactive speeds.

For all approaches listed above, it is noteworthy that (with the exception of Witkin's 1990 method) the coupling between the constraints is dealt with by some form of global optimization, giving rise to the (repeated) solution of coupled sets of (linearized) equations.

This means that either the computational effort per iteration step is high, or a technique for sparse linear systems or block systems is to be used which introduces a considerable administration overhead in the case of topological changes in the set of constraints of the simulated system (e.g. adding or deleting constraints). The latter also holds for methods which are based on symbolic manipulations.

As a result, the problem of simulating the constraint-based behaviour of a system at near real-time speed can considered to be solved as long as the structure of the system does not vary too much over time.

Interactive applications can be imagined, however, where direct manipulation of the constraint structure by the user is important. (We will call this

sort of applications *sketch*-applications; the object to be sketched can be a mechanism, a layout of geometrical components or "just" a drawing which has some constraints imposed on it.) In this type of applications, changes of the constraint graph may frequently occur.

This means that here (a) the topological structure of the constraints may constantly vary, (b) the system of constraints can be, at any time, over- or under constrained and (c) a variety of different types of constraints should be available. Fortunately, the user will be mainly interested in visual feedback while sketching, so (d) the desired accuracy will be typically not as high as in applications like e.g. the ones in [14], as long as (e) the accuracy improves to quantitatively significant values if the user is willing to give the system a little more time.

This paper studies a possible approach for constraint relaxation in sketch-applications. It is clear from (a) that techniques based on sparse matrices or block matrices are less suited, as are symbolic algebra-type approaches due to (b). The aspects (d) and (e) allow for a re-consideration of linearly-convergent methods, provided we can cast them in a form which is sufficiently general to accommodate with (a,b, and c). In Section 2 we contrast the two most familiar linearly-convergent methods for solving sets of coupled non-linear equations: the relaxation method (in the style of Southwell) and the steepest descent method. Section 3 gives a variety of constraint-types and the associated algorithms for relaxation, and in Section 4 some examples of simulated systems are presented.

2. A RELAXATION ALGORITHM FOR SOLVING CONSTRAINTS

In this section, a simple approach for dealing with constraints on real-valued parameters is described.

In the classification of constraint resolution methods of PLATT [12], the method to be discussed here is in between (i) a *penalty method* (where a constraint is satisfied by moving a state vector in the direction of the gradient of a scalar penalty function) and (ii) the *constraint stabilisation method* (where every distinct constraint gives rise to a component of the displacement vector, and each component is weighted separately by a Lagrange multiplier).

With respect to (i), the relaxation method discussed in this paper does not classify as a penalty method, since it does not use one scalar penalty function for the entire system; it rather maintains a separate penalty term for each constraint.

With respect to (ii) we observe that the systems we consider are first order time dependent systems rather than dynamical systems; as a consequence, the computations are both conceptually and numerically expected to be simpler.

The main characteristics of the approach are its flexibility (it works for a variety of constraint-types and it can be implemented to handle both under- and over-constrained systems in a stable way) and its suitability for distributed computation. Moreover, asymptotically, given that the system converges, constraints are satisfied exactly. The method is based on the idea of simultaneous

relaxation of (geometrical) constraints.

Before giving the actual method, an important disclaimer should be made:

> The essential feature of the method given here is its generality. Since the method is non-linear, there is no guarantee that, for a given type of constraints, it actually will give a converging method; moreover, if the method converges, it is not guaranteed that the convergence scheme is the fastest linearly convergent method available; finally, in case the model is under-constrained, nothing is said about which solution is found. In order to be able to make quantitative statements on the performance and the accuracy, an analytical convergence analysis should precede implementation of this method.

We consider an $N-$dimensional vector of real valued parameters, the state vector $\bar{x}$, determining the state of the model under study. Typically, the coefficients are coordinates of the relevant points in the sketched object, such as locations of hinges, bearings of cog wheels, etcetera. Assume that a set of constraints should be maintained which may be written in the form

$$\bar{f}(\bar{x}) = 0;$$

$$\bar{f} = \{f_a | 0 \leq a < M\};$$

$$\bar{x} = \{x_i | 0 \leq i < N\}.$$

To distinguish the state vector from vectors in the geometrical space where the constraints are defined, the latter vectors are denoted in boldface, so if e.g. $\bar{x} = (x_0, y_0, x_1, y_1, x_2, y_2, ...)$ then $\mathbf{x_0} = (x_0, y_0)$, $\mathbf{x}_1 = (x_1, y_1)$ etcetera. The real valued functions f_a in general will depend only on a limited subset of the x_i. The f_a may be (multi-) linear or non-linear. For the method to be presented here, it is essential that with every function f_a an algorithm A_a exists such that (the predicates in braces are the pre- and postcondition of the algorithm):

$$\{f_a(\bar{x}) \neq 0\}$$

$$A_a;$$

$$\{f_a(\bar{x} + \bar{d}_a) = 0\}$$

In words: given a state vector $\bar{x}$, the algorithm A_a should find a vector $\bar{d}_a$, such that $f_a(\bar{x} + \bar{d}_a) = 0$. Those components d_{ai} of $\bar{d}_a$ corresponding to parameters x_i that do not occur in $f_a(\bar{x})$ are assumed not to contribute to $\|\bar{d}_a\|$, i.e. they are assumed to vanish. (In a practical implementation, they do not even occur). In most cases it is advisable for $\bar{d}_a$ to have in some sense a minimal norm; this turns out to be often advantageous in convergence analyses. The complete algorithm for simultaneously solving the constraints looks as follows:

repeat
    for $a := 0$ to $M-1$ do if $f_a(\bar{x}) \neq 0$ then A_a;
    $\{\forall a : 0 \leq a < M : f_a(\bar{x} + \bar{d}_a) = 0\}$
    for $a := 0$ to $M-1$ do
        $\bar{x} := \bar{x} + \rho_a \bar{d}_a$;
    cope with user interaction;

until infinity;

Notice that in the algorithm, $\bar{d}_a$ is effectively a vector in geometric space (which should be written as $\mathbf{d}_a$, padded with zeros to make it an N-dimensional vector). This distinguishes the algorithm from a penalty method such as described in [12].

The active constraints (i.e. the constraints f_a which are checked in the repeat loop and for which algorithms A_a may be invoked), together with pointers to the associated algorithms A_a, are kept in a list. A user action which involves the geometry may cause changes in the $\bar{x}$; an action which involves the constraint-topology may result in an update of the list of active constraints. If no user interaction takes place for a sufficient long time (depending on the complexity of the simulated system and the speed of the used hardware), the process continues to iterate to arrive at more and more accurate estimates of the variables, provided that convergence occurs[2]. Here, "convergence" may mean the following:

- if the system was over-constrained, a 'best-fit' solution has been found.
- if the system possesses exactly one solution, this solution has been found.
- if the system is under-constrained, one of the solutions has been found.

Depending on the algorithms A_a this may be the solution with smallest norm $\|\bar{d}\|$.

The numbers ρ_a serve to control the convergence process. The smaller the ρ_a, the slower the convergence but the more stable the process. Theoretically, the values of ρ_a could be tuned for each of the constraints individually to arrive at the fastest convergence speed. This is similar to the introduction of variable weight factors in the computation of a penalty function in steepest descent methods. The optimal values for the ρ_a's may be difficult to obtain, however, they may vary during the iteration, and in practice we work with a value which is sufficiently small (say, 0.25) to get a stable process in all but very pathological cases. We observe that for each algorithm A_a in isolation, convergence is usually obvious: in most cases, such an algorithm will arrive at a configuration that satisfies the constraints without having to re-iterate at all. For instance, the algorithms presented in Subsections 3.1 and 3.2 give the

[2] In the implementation to be studied in Section 3 also "motors" are defined. In this case, the system continues to move even without user interaction.

correct results in one step for a value of $\rho = 1$, provided their parameters do not occur simultaneously in other constraints. A proof for global convergence for networks of *coupled constraints* is far from trivial, especially if constraints of different types occur. In [9], Eric van Loon gives a detailed analysis to compute the maximal value of ρ_a in order to guarantee global convergence in the case of a set of coupled length constraints (e.i. the constraint type of Subsection 3.5).

Instead of finding solutions of the equation $f_a(\bar{x}) = 0$, the algorithms A_a may also be arranged to yield a solution of the inequality $c_1 \leq f_a(\bar{x}) \leq c_2$ with $c_1 \leq c_2$. In that case, of course, the test $f_a(\bar{x}) \neq 0$ should be replaced by $f_a(\bar{x}) < c_1$ or $f_a(\bar{x}) > c_2$.

This means that in the same algorithm both types of constraints, equations and inequalities, are dealt with in precisely the same manner.

It is obvious that concerning the performance of the constraint solver, much, if not all, depends on the choice of the algorithms A_a and on the simulated system. Nevertheless, some general characteristics of the method are:

- All types of constraints, occurring in a given application, are dealt with in the same, unified, manner. The algorithms A_a may be developed in isolation, one for every type of constraint function, which will generally be much easier than designing a constraint solving strategy for an entire problem. Once a library of available constraint-solvers has been implemented, they may be instantiated under user control during an interactive sketch-session. (If we would restrict ourselves to constraints functions that are quadratic polynomials, the algorithms can even be derived automatically be differentiating the squared constraint function.)
- The executions of all A_a in the first for-loop in the algorithm may take place independently (it can even take place in parallel on a distributed computing platform). This means that a complete de-coupling of all constraints has been achieved. The data communication between the several constraint functions takes place in the second half of the algorithm, where the resulting $\bar{d}_a$ vectors are added to the $\bar{x}$. This means that the iterative nature of the method comes in the place of the simultaneous equations that have to be solved in the conventional constraint solvers.
- The notion of order-independence in the above algorithm turns out to be rather crucial. First, it is obvious that a successive update of the components of $\bar{x}$ rather than accumulating the $\bar{d}_a$ vectors would introduce an order-dependence which destroys any symmetry that happens to be present in the simulated system. More important, however, is a significant reduction of the convergence velocity. In Figure 1, the squares in the graph labeled *order dependence* show the 10-logarithm of the error as a function of the number of iteration steps if we use the above algorithm to find the intersection of two circles[3], whereas the crosses are found with an algorithm using successive updates. In both cases, the ρ_a were

[3] one circle with midpoint at (0,0) with radius 100 and the other one at (80,0) with radius 100; the starting point was chosen at (200,10)

all set to 0.5. Although the convergence is linear, we obtain about 2/3 digit of accuracy per iteration with our relaxation algorithm, whereas the successive update strategy is about 8 times as slowly.

A deterioration of the convergence speed due to successive updating of the components of $\bar{x}$ is expected to be particularly significant if the several $\bar{d}_a$ for a given point are large and directed in near opposite directions. If the $\bar{d}_a$ are added first, the resulting vector will be relatively small which means that the point moves in small steps. When applying the $\bar{d}_a$ in this case directly to compute successive updates, the updated point would move with large zig-zag type steps, which can delay the convergence, especially if the point is already close to its optimal location.

- As noted above, the convergence of the iteration may either imply a best-fit solution, a unique solution or one (possibly smallest-norm) solution from the multi-solution space of an under-constrained problem. This means that the method assumes no prior knowledge about the dimension of the solution space.
- The character of the constraint solver as an iterative process allows for standard convergence accelerators (extrapolators) for increasing its efficiency.

From its structure, the relaxation algorithm very much resembles a steepest descent method. Indeed, as we will see in Section 3, the applied algorithms A_a in several cases compute a step vector (i.e. a part **d** of the step vector $\bar{d}$ that is going to be added to the state vector $\bar{x}$) in the direction of the gradient of the constraint function f_a. However, consider again the problem of finding the point of intersection of two circles. The graph in Figure 2, labeled *factor 20*, depicts the convergence of the relaxation algorithm (squares) and the steepest descent algorithm (crosses). The steepest descent algorithm was equipped with a line search process [13] to establish the optimal step size at each iteration. Nevertheless, it performs significantly inferior when compared with the relaxation method. In order to study this phenomenon, we apply linearisation, both of the steepest descent method and the relaxation method. For the steepest descent algorithm, we set

$$E(\bar{x}) = \sum_a f_a^2(\bar{x}).$$

The step-direction is

$$\bar{d} = \lambda \nabla E = 2\lambda \sum_a f_a \nabla f_a,$$

where λ defines the step size. Assume that the system of equations indeed has a solution, $E(\bar{x} + \bar{d}) = 0$, and that we are sufficiently close to it. Then we can compute λ using first order Taylor expansion:

$$0 = E(\bar{x} + 2\lambda \sum_a f_a \nabla f_a)$$

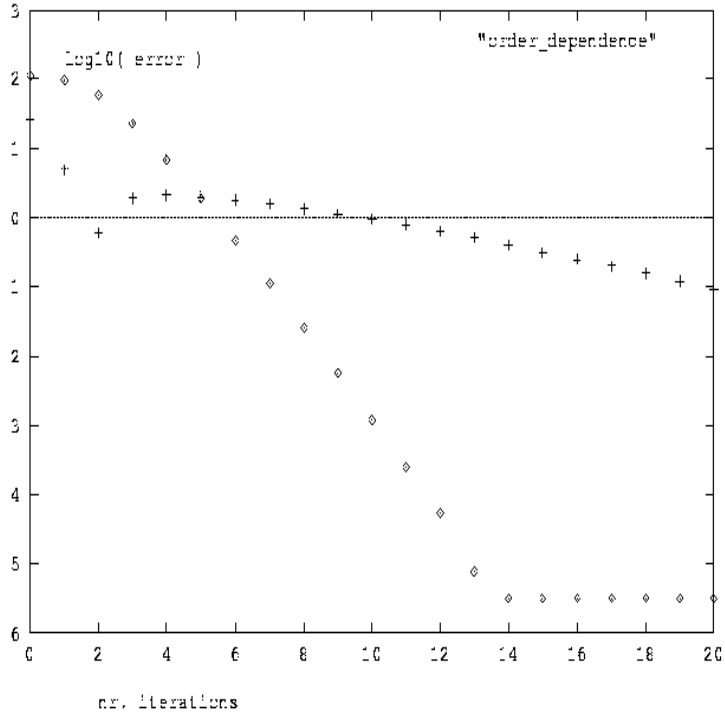

FIGURE 1. The effect of order indepence on the convergence of the relaxation algorithm

$$\approx E(\bar{x}) + 2\lambda(\sum_a f_a \nabla f_a, \sum_a f_a \nabla f_a)$$

so

$$\lambda \approx -\frac{\sum_a f_a^2}{2(\sum_a f_a \nabla f_a)^2}.$$

Hence

$$\bar{d} \approx -\frac{\sum_a f_a^2}{\| \sum_a f_a \nabla f_a \|} \bar{e}_{\nabla E}$$

where $\bar{e}_{\nabla E}$ is the unit vector in the direction of the gradient of E.

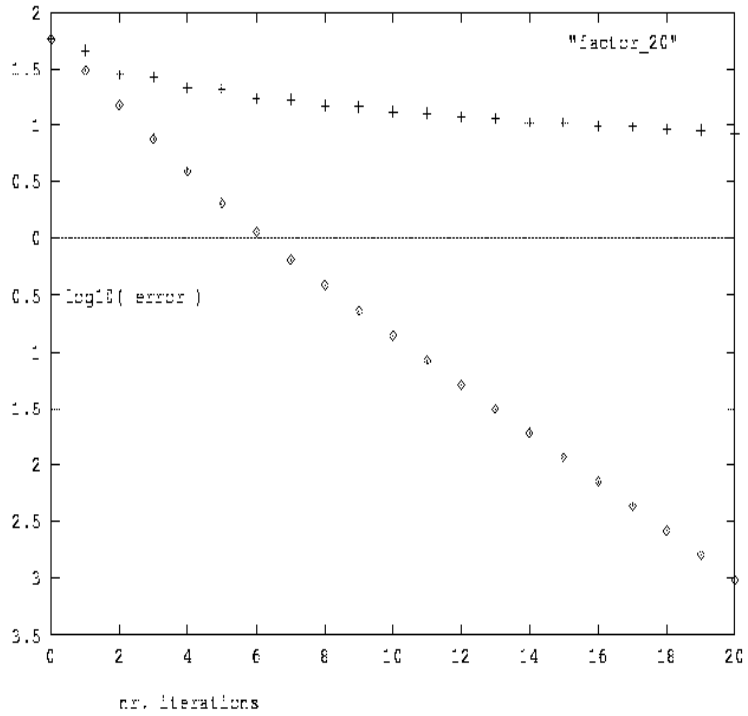

FIGURE 2. The convergence of relaxation compared with steepest descent for computing the intersection of two circles with radii in a ratio of 1:20

On the other hand, under the same assumptions we find for the step vector in the case of relaxation:

$$f_a(\bar{x} + \bar{d}_a) = 0,$$

When linearising:

$$\bar{d}_a \approx -\frac{f_a \nabla f_a}{(\nabla f_a)^2},$$

so, even if we take all ρ_a equal to the same ρ:

$$\bar{d} \approx -\rho \sum_a \frac{f_a \nabla f_a}{\|\nabla f_a\|^2}.$$

Now although the expressions for $\bar{d}$ in both cases have a similar structure, there is one significant difference: in steepest descent, the contributions to the gradient of all participating constraints are weighted the same, whereas in relaxation, the weight of every constraint is taken proportionally to $(\nabla f_a)^{-2}$. This means if one of the constraints is much more sensitive to a change in (part of) the configuration vector $\bar{x}$, its associated gradient will contribute less to the total step vector. Especially in cases where constraints with strongly different "sensitivities" are competing, like in the example above, where the radii of the two circles differ a factor 20, this difference between the steepest descent and relaxation methods seems to work in favour of relaxation. If we repeat the experiment with a much less extreme difference in radii, we observe that the relaxation and steepest descent methods become similar (see Figure 3; here the radii differed only a factor 2).

Of course, in hindsight this should not surprise us: it is assumed that the bad performance of steepest descent is essentially due to the fact that the penalty function has been constructed without applying weight factors to the individual terms.

Notice, in passing, that we applied linearisation in order to arrive at a feasible analysis. The relaxation method itself deals directly with the non-linear constraints as they are given. This means that e.g. in the case constraints happen to be uncoupled, in which case the associated ρ_a can be set to 1, relaxation finds a converged solution in 1 step, whereas a scheme based on linearisation still would do several steps towards convergence.

Ignoring the variations of the $(\nabla f_a)^{-2}$ during iteration, we can envisage the relaxation method as a steepest descent method with a penalty function that is a weighted sum over the terms f_a^2, where the weight factors are chosen such as to correct for the differences in sensitivities of the different constraints. In other words, again ignoring variations of the $(\nabla f_a)^{-2}$ during iteration, relaxation is a weighted steepest descent method with "automatically" computed weights (i.e. without having to tune any of the ρ_a). In the application we study, where a constraint network is constantly close to satisfaction because the parameter changes due to user interaction are slow when compared with the iteration update frequency, the condition of nearly constant $(\nabla f_a)^{-2}$ is expected to be fulfilled. This also means that under these same assumptions, we can infer convergence properties of the relaxation method from the corresponding properties of steepest descent.

Apart from the numerical differences between steepest descent and relaxation, we mention that

- A steepest descent implementation based on a stand-alone numerical library routine, where the value of ∇E is computed numerically by repeatedly evaluating E for $\bar{x}$ each time with a small difference in one of the components x_a takes $\mathcal{O}(M \times N)$ calculations whereas relaxation only takes $\mathcal{O}(M + N)$ calculations per iteration. Of course, a more efficient steepest descent implementation is possible, but then the evaluation of

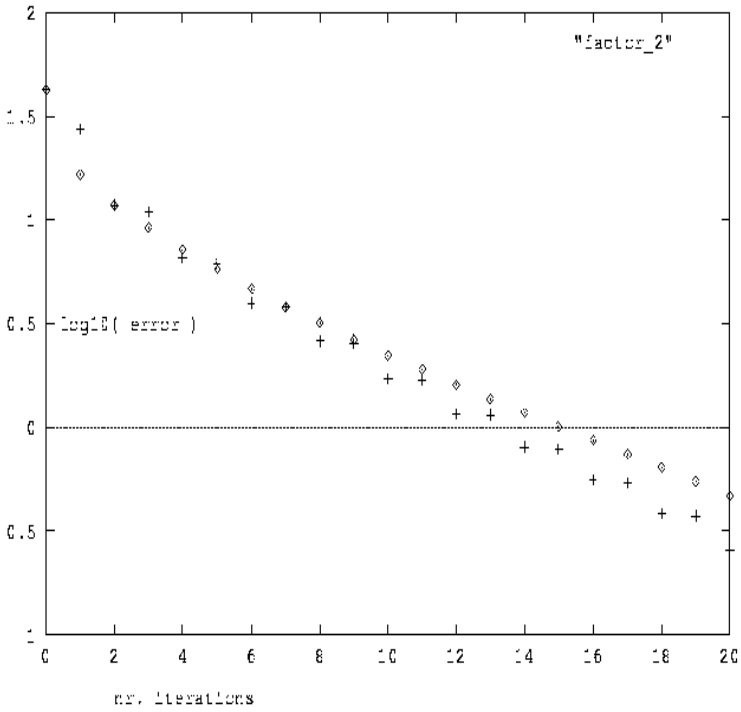

FIGURE 3. The convergence of relaxation compared with steepest descent for computing the intersection of two circles with radii in a ratio of 1:2

the several constraint functions has to be hard-coded into the steepest descent algorithm (as is the case with relaxation).

- It is not a priori clear how a steepest descent algorithm should cope with inequalities whereas this is evident for relaxation.
- By just adding together all squared constraint functions in order to compute one scalar penalty term E in steepest descent, we throw away a lot of detailed information which is preserved (somewhat more) in relaxation, since there the direction in which a point p is going to move for the next step is derived only from the constraints that are related to p, and constraints that are not coupled to p have no influence. In steepest descent, all information for computing the next step is taken from the

penalty function which depends on *all* constraints at once.

We summarise:

In applications of geometric constraints where frequent configuration changes occur, and where a fast approximate result of constraint resolution is desired (eventually to converge more slowly to an accurate result), a linearly convergent method may be useful; of these methods relaxation is preferable above steepest descent methods, and care should be taken to implement relaxation with simultaneous update of the state variables rather than sequential update.

3. The formulation of constraints for simulating mechanisms

When using the constraint resolution algorithm as outlined in the previous section, all that remains to be done is to formulate the functions f_a and the associated algorithm A_a for a series of constraint types that are of practical use. A formulation of the algorithms is chosen that guarantees a symmetrical behavior, i.e. in case several points are subject to one constraint, the $\mathbf{d}-$vectors of these points will be computed such that all of them will be affected in a way that is "physically meaningful".

In several cases the expressions for $\mathbf{d}$ (and hence $\bar{d_a}$) can be obtained by simply differentiating the squared constraint function f_a to its arguments. Care should be taken, however, by constraint functions that are not quadratic functions. E.g. it is left to the reader to show that the result that is obtained by differentiating the squared orientation constraint (see Subsection 3.6 and 3.7) gives an undefined result in case the input vectors are parallel.

Most of the following examples are relatively straightforward; they are chosen such that both a variety of planar mechanisms can be constructed and simulated and a large collection of theorems from planar geometry (see e.g. Subsection 4.4) can be illustrated. Notice also that non-holonomic constraints are dealt with in the same framework (non-holonomic constraints are constraint expressions where time derivatives of the constrained variables occur).

For describing the algorithms, the convention is used that the parameters of the functions are coordinates of the relevant points (except for the cog wheel constraint). The $\mathbf{d}$-vector associated with point $\mathbf{x}_i = (x_i, y_i)$ is $\mathbf{d}_i = (d_{x_i}, d_{y_i})$; this $\mathbf{d}$-vector is a segment of the $\bar{d}$-vector that is going to be used to update the state vector $\bar{x}$.

3.1. Coordinate constraint

This constraint expresses that a point $\mathbf{x}_1 = (x_1, y_1)$ should be tied to a certain location $\mathbf{c} = (c_x, c_y)$ in space. It takes the form

$$f(\mathbf{x}_1) = \|\mathbf{x}_1 - \mathbf{c}\|;$$

The algorithm reads:

$$\mathbf{d}_1 := \mathbf{c} - \mathbf{x}_1$$

(A similar result would be obtained by differentiating the squared constraint function).

3.2. Coincidence constraint

The coincidence constraint expresses that points $\mathbf{x}_1$ and $\mathbf{x}_2$ should coincide. It takes the form

$$f(\mathbf{x}_1, \mathbf{x}_2) = \|\mathbf{x}_1 - \mathbf{x}_2\|.$$

The algorithm reads:

$$\mathbf{d}_1 := (\mathbf{x}_2 - \mathbf{x}_1)/2;$$

$$\mathbf{d}_2 := -\mathbf{d}_1;$$

(A similar result would be obtained by differentiating the squared constraint function).

3.3. Equivalence constraint

In geometrical constructions, the equivalence of two difference vectors is often encountered, *equivalent* here means that the two vectors are parallel and of equal length. Indicating the first vector by $\mathbf{x}_2 - \mathbf{x}_1$ and the second one by $\mathbf{x}_4 - \mathbf{x}_3$, then

$$f(\mathbf{x}_1 \ldots \mathbf{x}_4) = \|\mathbf{x}_4 - \mathbf{x}_3 - \mathbf{x}_2 + \mathbf{x}_1\|.$$

The following algorithm[4] is easily seen (in the absence of any other constraints) to yield

$$f(\mathbf{x}_1 + \mathbf{d}_1, \ldots, \mathbf{x}_4 + \mathbf{d}_4) = 0.$$

As follows:

$$\mathbf{d}_4 := \mathbf{d}_1 := (\mathbf{x}_2 + \mathbf{x}_3 - \mathbf{x}_4 - \mathbf{x}_1)/4;$$

$$\mathbf{d}_3 := \mathbf{d}_2 := (\mathbf{x}_1 + \mathbf{x}_4 - \mathbf{x}_2 - \mathbf{x}_3)/4;$$

(This follows as well from differentiating the squared constraint function.)

3.4. Equal distance constraint

A use for the equal distance constraint could happen during sketching a mechanism: it expresses that two point-tuples have equal distances, without stating anything about the actual value of this distance. Let the argument points have the same meanings as in §3.3, then

$$f(\mathbf{x}_1, ..., \mathbf{x}_4) = \|\mathbf{x}_1 - \mathbf{x}_2\| - \|\mathbf{x}_3 - \mathbf{x}_4\|.$$

The algorithm operates by first computing the average distance of the two tuples, and next computing the $\mathbf{d}$'s in order to have the new locations of the points such that these distances hold. In doing so, the orientation of both $\mathbf{x}_1 - \mathbf{x}_2$ and $\mathbf{x}_4 - \mathbf{x}_3$ remains the same; similarly for the centres of gravity of the two tuples. Even though the latter requirement does not follow from demanding $f = 0$, it obviously helps to keep the algorithm remain stable. In the following, $\mathbf{e}_{ij}$ denotes the unit vector in the direction of $\mathbf{x}_j - \mathbf{x}_i$, and $l_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|$.

[4] a:=b:=c is short for b:=c; a:=b

$\Delta := (l_{12} - l_{34})/4;$

$\mathbf{d}_4 := \mathbf{d}_1 := \Delta \mathbf{e}_{12};$

$\mathbf{d}_2 := \mathbf{d}_3 := -\Delta \mathbf{e}_{12};$

(This follows as well from differentiating the squared constraint function.)

3.5. Length constraint
The length constraint is typically used to define (rigid) rods. Notice that the length constraint in the form of an inequality may be used to express the geometric behaviour of a chord of given length (imposing a maximal distance between its end points) or of a telescope with a given minimal and maximal length. For l the length of the rod, and $\mathbf{x}_1$ and $\mathbf{x}_2$ its end points, f reads:

$$f(\mathbf{x}_1, \mathbf{x}_2) = \|\mathbf{x}_1 - \mathbf{x}_2\| - l.$$

The algorithm is similar to the algorithm of §3.4; it will therefore be omitted here. Again, satisfying the length constraint does not introduce any spurious rotations or translations: the orientation of the segment $\mathbf{x}_1\mathbf{x}_2$ as well as its centre of gravity remain fixed.

3.6. Orientation constraint
(This is the first constraint type where the distinction between 2-D and 3-D matters; the implementation discussed here is 2-D)

An orientation constraint may serve to express that two components are mounted on the same axis, rotating with the same speed and having thus a constant relative angle, say ϕ, as a function of time. In this context, it would suffice to define the orientation constraint in terms of three points: the common rotation centre and two endpoints. A somewhat more general definition, however, allows the orientation constraint to be applicable also to express parallelism or perpendicularity of two vectors, as well as any other angular relation. Again introducing two point tuples with the relevant vectors $\mathbf{x}_1, ..., \mathbf{x}_4$, f is expressed as

$$f(\mathbf{x}_1, \ldots, \mathbf{x}_4) = \angle(\mathbf{e}_{12}, \mathbf{e}_{34}) - \phi.$$

Here, the function $\angle$, unlike the inner product, is taken to be anti-symmetric in its arguments. The algorithm should leave the lengths of the segments $\mathbf{x}_1\mathbf{x}_2$ and $\mathbf{x}_3\mathbf{x}_4$ alone; therefore the $\mathbf{d}$'s should be computed such as to constitute rotations of these segments only. Also, in order to avoid net rotations or translations of the system as a whole, these rotations should be equal and of opposite signs for the two segments, and the centres of mass of the two segments should not be affected. By implementing these requirements, care should be taken that the sign of the angle between $\mathbf{e}_{12}$ and $\mathbf{e}_{34}$ is properly taken into account. An outline of the algorithm reads as follows:

$\mathbf{m}_{12} := (\mathbf{x}_1 + \mathbf{x}_2)/2;$

$\mathbf{m}_{34} := (\mathbf{x}_3 + \mathbf{x}_4)/2;$

$\delta := (\phi - \text{current angle between } \mathbf{x}_1\mathbf{x}_2 \text{ and } \mathbf{x}_3\mathbf{x}_4)/2;$

now the segments $\mathbf{x}_1\mathbf{x}_2$ and $\mathbf{x}_3\mathbf{x}_4$ should rotate over $-\delta$ and δ, respectively.

$\mathbf{d}_1 := \mathbf{m}_{12} + \text{rotate}(\mathbf{x}_1 - \mathbf{m}_{12}, -\delta) - \mathbf{x}_1;$

and similar for the $\mathbf{d}_2$, $\mathbf{d}_3$, and $\mathbf{d}_4$.

3.7. Motor constraint

Of all types of motions, circular motion plays a very prominent role. The motor constraint expresses that $\mathbf{x}_2$ circulates around $\mathbf{x}_1$ with a given radius r, a given frequency ω, and a given starting phase ϕ_0. By treating this relative motion as a constraint, the case where $\mathbf{x}_1$ is not fixed in space is covered as well. It is assumed, however, that the frame of reference of the motor, located at $\mathbf{x}_1$, remains parallel with a fixed frame of reference. In other words, the point $\mathbf{x}_2$ experiences no difference if the casing of the motor would be revolving. A better interpretation of this constraint therefore would be that the orientation of $\mathbf{x}_1\mathbf{x}_2$ rotates in space. The function f for this constraint reads

$$f(\mathbf{x}_1, \mathbf{x}_2) = \angle(\mathbf{e}_{12}, \mathbf{e}_x) - \phi_0 - \omega t.$$

Here t stands for the simulation time, and $\mathbf{e}_x$ is the horizontal unit vector. The algorithm is a simplified version of the previous algorithm; it is therefore omitted here.

3.8. Glider constraint

A well-known example of a glider constraint is found in a piston, where an object has one translational degree of freedom over a limited domain. Note that this is again a symmetric constraint: the piston is constrained by the cylinder it moves in, but the cylinder (in case it would be allowed to move) is also constrained by the piston. Moreover, this constraint is the first example where both an equation and two inequalities occur. This means that introducing a glider constraint gives rise to 3 functions: f_1 expressing that the point $\mathbf{x}_1$ must remain on the line through $\mathbf{x}_2\mathbf{x}_3$, and the functions f_2 and f_3 expressing that the projection of $\mathbf{x}_1$ onto that line falls within the interval $\mathbf{x}_2\mathbf{x}_3$. The functions f read:

$$f_1(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) = (\mathbf{x}_1 - \mathbf{x}_2, \mathbf{x}_3^P - \mathbf{x}_2^P),$$

$$f_2(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) = (\mathbf{x}_1 - \mathbf{x}_3, \mathbf{x}_3 - \mathbf{x}_2),$$

$$f_3(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) = (\mathbf{x}_1 - \mathbf{x}_2, \mathbf{x}_3 - \mathbf{x}_2).$$

Here, $\mathbf{a}^P$ means the vector obtained by rotating $\mathbf{a}$ over $\pi/2$. f_1 should be equal to 0, $f_2 \leq 0$ and $f_3 \geq 0$. A realisation of all of these three (in-)equalities is given by the following algorithm:

- First compute the projection $\mathbf{t}$ of $\mathbf{x}_1$ onto the line through $\mathbf{x}_2\mathbf{x}_3$. To this aim, write $\mathbf{t} = \mathbf{x}_2 + \mu(\mathbf{x}_3 - \mathbf{x}_2)$

  $$\mu := \frac{(\mathbf{x}_1 - \mathbf{x}_2, \mathbf{x}_3 - \mathbf{x}_2)}{\|\mathbf{x}_3 - \mathbf{x}_2\|^2};$$

  if $\mu < 0$ then $\mu := 0$;

  if $\mu > 1$ then $\mu := 1$;

  $\mathbf{t} := \mathbf{x}_2 + \mu(\mathbf{x}_3 - \mathbf{x}_2) - \mathbf{x}_1$;

  the difference between the point $\mathbf{t}$ and $\mathbf{x}_1$ should be distributed over $\mathbf{d}_1$, $\mathbf{d}_2$, and $\mathbf{d}_3$. First, assume $\mathbf{d}_1$ be equal to $\mathbf{t}$. In order for the corrected $\mathbf{x}_1$ to end onto the corrected line segment $\mathbf{x}_2\mathbf{x}_3$, $(1-\mu)\mathbf{d}_2 + \mu\mathbf{d}_3 = \mathbf{t}$ should hold. Moreover, by interpreting the difference vector $\mathbf{t} - \mathbf{x}_1$ as some sort of reaction force, the force distribution law from classical statics should be used. This yields: $\mu\mathbf{d}_2 = (1-\mu)\mathbf{d}_3$. The above two equations yield: $\mathbf{d}_2 = \frac{\mu\mathbf{t}}{2\mu^2+1-2\mu}$ and $\mathbf{d}_3 = \frac{(1-\mu)\mathbf{t}}{2\mu^2+1-2\mu}$
- Finally, in order to keep the centre of gravity at rest, adjust both $\mathbf{d}_1$, $\mathbf{d}_2$ and $\mathbf{d}_3$ with the same translation vector, $\frac{-\mathbf{t}/3}{2\mu^2+1-2\mu}$ for their sum to become 0. The final result reads:

  $$\mathbf{d}_1 := \frac{\mathbf{t}/3}{2\mu^2 + 1 - 2\mu};$$

  $$\mathbf{d}_2 := -\frac{(\mu - 1/3)\mathbf{t}}{2\mu^2 + 1 - 2\mu};$$

  $$\mathbf{d}_3 := -\frac{(2/3 - \mu)\mathbf{t}}{2\mu^2 + 1 - 2\mu};$$

3.9. Hinge constraint

Both in the simulation of mechanisms and in the "simulation" of geometrical theorems, an often occurring relation holds that a point $\mathbf{x}_1$ resides at the line through $\mathbf{x}_2$ and $\mathbf{x}_3$, with some fixed ratio between $\|\mathbf{x}_1 - \mathbf{x}_2\|$ and $\|\mathbf{x}_1 - \mathbf{x}_3\|$. Both for the formulation of the constraint functions and for the algorithm, this is a special case of §3.8, namely the case where μ has a prescribed value.

3.10. Cog wheel constraint

The essential property for cog wheels is that they have (ideally) one point, say $\mathbf{t}$, in common; moreover, their velocities in $\mathbf{t}$ are the same. Apart from this constraint, their centres may either both be fixed, or one cog wheel may roll over the other one, or both may roll over another. Up to now, all constraints were easily expressed in the coordinates of the constituent points; this is not the

case anymore for cog wheels. The cog wheel constraint will have to be expressed in terms of the equality of the velocities of $\mathbf{t}$ when viewed from one cog wheel and the other (it is therefore an example of a non-holonomic constraint).

The cog wheels are parameterised by the velocities of the points $\mathbf{x}_1$ (the centre of the first cog wheel), $\mathbf{x}_2$ (a point on the first cog wheel, different from the centre) and similar $\mathbf{x}_3$ and $\mathbf{x}_4$ for the second cog wheel.

Cog wheels can touch at the outside or at the inside; inside touching is expressed by setting r (the ratio of the radii) negative. Given r, $\mathbf{t} = (r\mathbf{x}_1 + \mathbf{x}_3)/(r+1)$, provided $r \neq -1$. For well-chosen matrices A_1 and A_2 (see Appendix A for the construction of A_i) the velocity $\mathbf{t}'$ of the point $\mathbf{t}$ can be expressed in terms of the velocities $\mathbf{x}_i'$ of the points $\mathbf{x}_i$ as follows:

$$\mathbf{t}' = \mathbf{x}_1' + A_1(\mathbf{x}_2' - \mathbf{x}_1') = \mathbf{x}_3' + A_2(\mathbf{x}_4' - \mathbf{x}_3').$$

This immediately gives the constraint function, this time expressed in the $\mathbf{v}_i$ rather than the $\mathbf{x}_i$:

$$f(\mathbf{v}_1, \ldots, \mathbf{v}_4) = \|\mathbf{v}_1 + A_1(\mathbf{v}_2 - \mathbf{v}_1)\| - \|\mathbf{v}_3 + A_2(\mathbf{v}_4 - \mathbf{v}_3)\|.$$

The construction of the algorithm now is based on the observation that, assuming the Euler discretisation scheme, $\mathbf{d}_i = h\mathbf{v}_i$. Thus, given current estimates of the $\mathbf{v}_i$, based on the difference between the current locations and the previous locations, the algorithm should compute new estimates in order to make the constraint hold for the next time step as well. The following algorithm can easily be seen to do so; (the new estimates for the velocities are indicated by $\tilde{\mathbf{v}}_i$):

$$\tilde{\mathbf{v}}_1 := (I - A_1)^{-1}((I - A_2)\mathbf{v}_3 + A_2\mathbf{v}_4 - A_1\mathbf{v}_2);$$

$$\tilde{\mathbf{v}}_3 := (I - A_2)^{-1}((I - A_1)\mathbf{v}_1 + A_1\mathbf{v}_2 - A_2\mathbf{v}_4);$$

$$\tilde{\mathbf{v}}_2 := {A_1}^{-1}((I - A_2)\mathbf{v}_3 + A_2\mathbf{v}_4 - (I - A_1)\mathbf{v}_1);$$

$$\tilde{\mathbf{v}}_4 := {A_2}^{-1}((I - A_1)\mathbf{v}_1 + A_1\mathbf{v}_2 - (I - A_2)\mathbf{v}_3);$$

next compute the $\mathbf{d}$'s from these new velocities; notice that the values of $\mathbf{v}_i$ hold in case every single one has to cope with fulfilling the entire constraint relation. The constraint is fulfilled, however, by means of a cooperation of both $\mathbf{x}_1$, $\mathbf{x}_2$, and $\mathbf{x}_3$, $\mathbf{x}_4$, hence every $\mathbf{d}_i$ is only responsible for half the change:

$$\mathbf{d}_1 := h(\tilde{\mathbf{v}}_1 - \mathbf{v}_1)/2;$$

and similar for $\mathbf{d}_2$, $\mathbf{d}_3$, and $\mathbf{d}_4$.

4. Implementation; some examples

In order to test the above ideas a prototype implementation of a sketch-system based on constraint relaxation has been built at Technical University of Eindhoven.

It provides the following functions:

- compact specification of a mechanism by means of a command language;
- direct manipulation of the constraint structure via updates in the command-language and via freezing and thawing of points (For visual feedback, a frozen point is labeled with a triangle; in the relaxation algorithm, no d-vector is computed for frozen points);
- direct manipulation of the geometry via dragging of frozen points while the simulation runs;
- autonomous motion via motor constraints.

The examples in this section are realised with this system. The figures are obtained by direct screen dumps from the simulations; the lettering has been added later.

Interactive control is offered over the coefficient ρ, which controls the convergence of the algorithm[5], and over the number of iterations the algorithm is allowed to make between any two subsequent frame updates. The larger this number, the more accurate the constraints are fulfilled in the drawn frames, but the slower the simulation. On a SUN SPARC station, a simulation with 10 to 15 constraints with a number of iteration steps set to 30 gives a typical frame update rate of 20 frames/second, including the drawing. Except for the cog wheel constraints, this gives a hardly noticeable deviation from the constraints even in the case of rapid motion (i.e., fast dragging or high-speed motor constraints). The cog wheel constraints sometimes are a little bit more sensitive to the simulation speed and/or the number of steps; this is believed to be due to the poor characteristics of the Euler integrator.

4.1. Guilloche engine turning

Figure 4 shows a screen-dump of the guilloche engine. It consists of 5 motors, 4 of them are at frozen locations (motor1,...,motor4). Motor1 causes point p1 to move in a circle, and similar for p2, p3, and p4. The lengths p1-q1, p2-q1, p3-q2, and p3-q2 are defined by length constraints, so the paths of q1 and q2 follow. Next, motor5 is mounted on the joint of two rods, again with fixed lengths, originating in q1 and q2, so it describes a curve in the plane. Finally, the point r is driven by motor 5, and its path traces out a rather complex curve in the plane (part of which is indicated by the dot-curve). During operation it is possible to drag the motors over the screen, to change rotation velocities, etc.

4.2. Motor-driven crankshaft coupled with four pistons

Figure 5 shows a snapshot of this mechanism. It consists of a motor which causes point p to describe a circular motion. In p, four rods are connected: p-q1, p-q2, p-q3, and p-q4. The points q1,...,q4 are restricted to move along the gliders glider1,...,glider4. All extremes of the gliders happen to be frozen in this case, thawing some of them makes the system under constrained. Furthermore,

[5] all ρ_a from section 2 are set to the same value; practical values are between 0.50 and 0.75

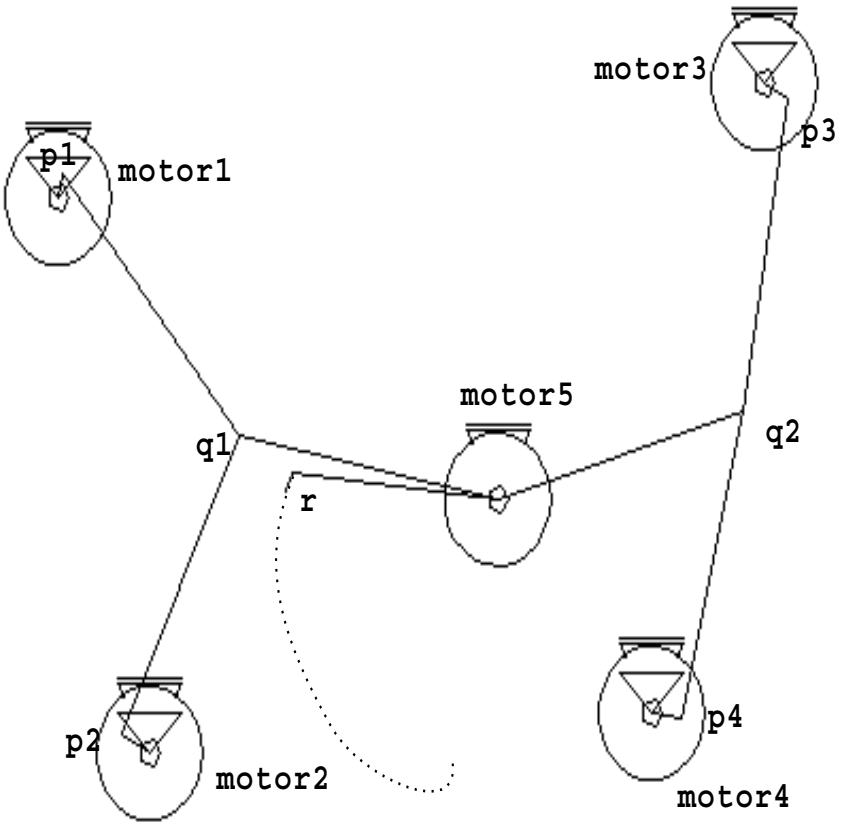

FIGURE 4. Guilloche engine turning

there are 4 line segments a1-b1, a2-b2, a3-b3, and a4-b4. The length of each of these is coupled (via a length constraint) to the distance of one of the q to the far end of the corresponding glider (the distance a1-b1 equals the distance between q1 and the lower most extreme of glider1, and so on). Since the a1,...,a4 are frozen, the b1,...,b4 move in the form of a "running wave" while the motor is running. This example nicely demonstrates the principle of reversibility of the constraint algorithms we use: if the motor constraint is switched off, the user can "push" each of the points b1,...,b3 up repeatedly and hence again cause the point p to rotate, thus demonstrating the conversion of a linear motion into a circular motion due to constraints.

4.3. Planet gear set of cog wheels coupled with a glider

Figure 6 shows a screen-dump of the system. It consists of two cog wheels.

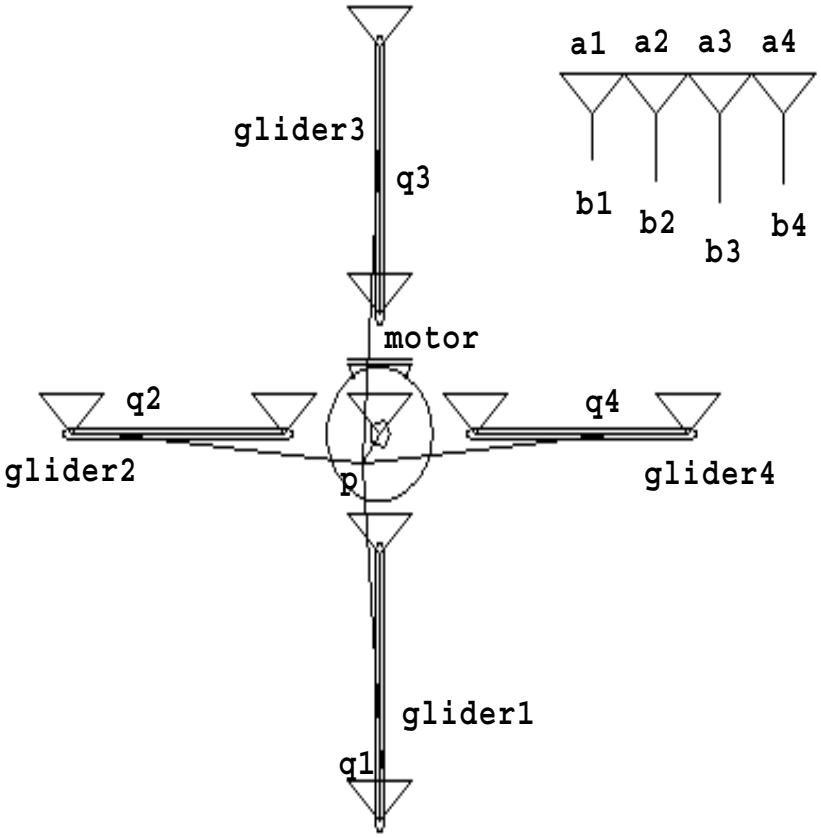

FIGURE 5. Motor driven crankshaft

The largest one is cogwheel1; its centre is the frozen point p1. Cogwheel1 is prohibited from rotating since point a (which is a point on the cogwheel1) is frozen. Concentric with p1, a motor is defined which causes the point p2 to move in a circle within cogwheel1. The point p2 is the centre of a second, smaller cog wheel, cogwheel2, which is hence forced to roll within cogwheel1. The radius of cogwheel2 is 0.25 times the radius of cogwheel1. The point b is a fixed point with respect to cogwheel2, and its trajectory in space is thus defined (it describes a four-leave rosette, provided no numerical errors are made; in practice, a small phase shift may occur due to the bad performance of the Euler method for integrating the velocities). This point b in turn is constrained to move within a glider with frozen extreme g1, and as a result the point g2 oscillates in a complex fashion.

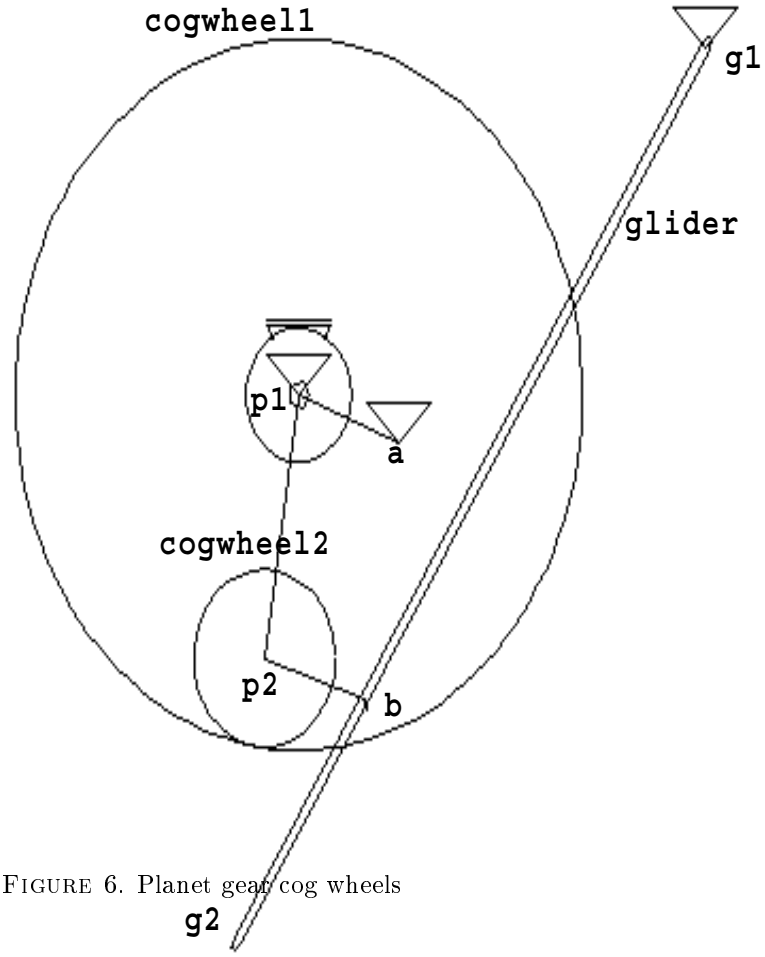

FIGURE 6. Planet gear cog wheels

4.4. Illustration of a geometrical theorem

In Figure 7 we have depicted a multi-exposure screen dump of this system. Two tangent circles have been given, circle1 has a radius which is twice as large as the radius of circle2. In the centre of circle2, two motors are defined. One motor drives point s along circle2 with velocity w; the other motor drives point u along the same circle with velocity 2w. Point r is the tangent point of the two circles; via a hinge constraint, the point t is defined on the line through r and s such that the distance r-s equals the distance s-t. As is easily seen, then t moves along the larger circle. A simple theorem from planar geometry now states that the point u is always on the line through t and p, where p is the centre of circle1. The simulation shows that this is indeed the case.

4.5. The recursive subdivision algorithm for cubic Bezier curves.

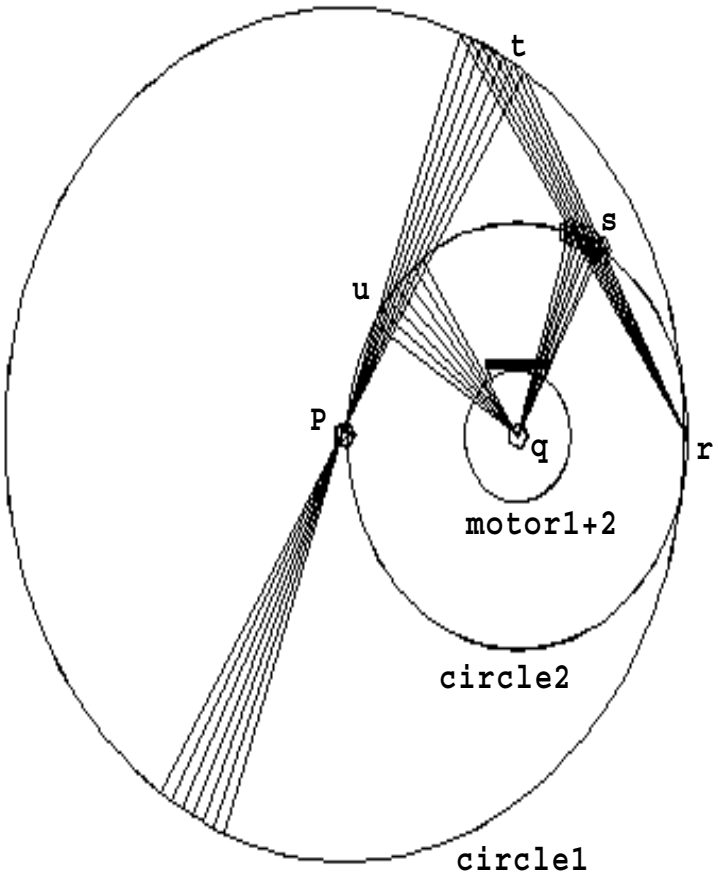

FIGURE 7. Illustration of a geometrical theorem

The last example is taken from a geometrical design application. Suppose we want to construct a cubic Bezier curve of which we know three of its control points and the midpoint. Figure 8 shows a solution to this problem, defined in terms of hinge constraints. Points p1,p2 and p3 are frozen; they represent the given control points. The point q is frozen as well; this is the given midpoint of the curve. The recursive subdivision is implemented via hinge-constraints, each hinge constraint requiring a line segment to be subdivided into two equal halves. So p12 is midway between p1 and p2, p23 midway between p2 and p3, and so on. The relaxation algorithm computes the location of p4, and again we can study the dependence of the location of p4 on the input data by dragging the frozen points; or we can experiment with other sets of input data (including, e.g., points that are computed in the second subdivision step).

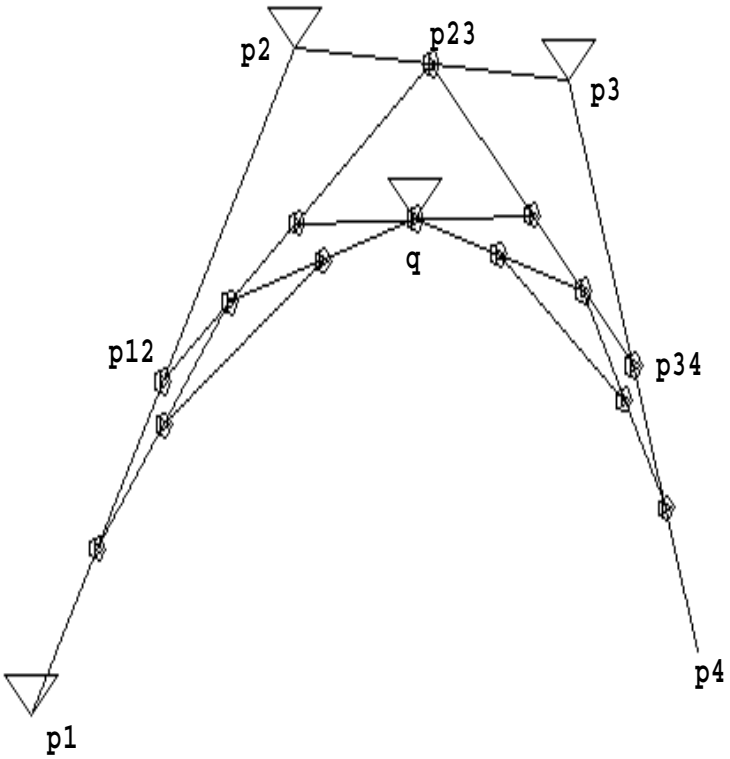

FIGURE 8. Recursive subdivision of a cubic Bezier curve segment

5. SUMMARY AND CONCLUSIONS

An algorithm for solving sets of coupled (non-) linear equations, based on an iterative scheme and the presence of reasonable starting values for the unknowns, is given. Such an algorithm can prove to be useful in a constraint-based simulation context, where variables change in a slowly manner compared with the simulation time. For a set of constraint types with geometric meanings (length constraints, orientation constraints, vector equivalence constraints, etcetera) a simple tool for the interactive design of 2-dimensional mechanisms has been built. Based on the algorithm as mentioned above, a SUN SPARC-station implementation succeeds in giving a frame update rate of approximately 20 frames per second for sketched objects consisting of 10 to 15 constraints, thereby using approximately 30 iteration steps per time frame. This amount of iteration steps proves to be sufficient to yield a (visually) correct behavior, except in

the case of cog wheels: there, an additional handicap comes in in the form of the differential equation which should be solved, introducing the need for some additional 20-40 iteration steps in order to guarantee stable motions.

Of course, a complete discussion of the merits of relaxation methods should be based on a wider array of applications than geometric constraints for kinematic simulation only. But maybe more interesting, a comparison with some more recent techniques such as simulated annealing or symbolic manipulation would be useful to deepen the understanding of constraint satisfaction methods in general. A non-trivial issue here is that of performance metrics: depending on the type of application, the trade-off between speed, accurateness, robustness and simplicity might favour one method or another. As an example, consider the applications of this paper, i.e. interactive sketching of mechanisms in motion. Assume that, while dragging some components of the mechanism, temporarily a configuration is obtained which cannot be made to meet with the constraints because, say, length constraints are violated. A symbolic constraint solver then could do nothing but report that no solution can be found, whereas a numerical scheme such as steepest descent or relaxation automatically generates approximate solutions that are, in a sense, optimal, thus mimicking the behaviour of deformable material components. Similarly, simulated annealing would be very useful to avoid getting caught in local optima in configurations that are still far from optimal; in slowly-dragging type of applications, where the configuration is constantly close to satisfaction, it would introduce considerable overhead.

Appendix A: the matrices A_i

The matrix A_1 serves to express $\mathbf{t}'$ as

$$\mathbf{t}' = \mathbf{x}'_1 + A_1(\mathbf{x}'_2 - \mathbf{x}'_1),$$

A_2. Let $\mathbf{r} = \mathbf{x}_2 - \mathbf{x}_1$. Moreover, $\mathbf{v}_1 = \mathbf{x}'_1$, $\mathbf{v} = \mathbf{x}'_2 - \mathbf{x}'_1$, $\mathbf{V} = \mathbf{t}' - \mathbf{x}'_1$ and $\mathbf{R} = \mathbf{t} - \mathbf{x}_1$. Then $\mathbf{t}'$ may also be written as:

$$\mathbf{t}' = \mathbf{v}_1 + A_1\mathbf{v}.$$

The matrix A_1 should be such that $\mathbf{V}$ is a scaled and rotated version of $\mathbf{v}$. This is expressed by

$$A_1 = s_1 R_1$$

where s_1 is a scalar,

$$s_1 = \frac{\|\mathbf{V}\|}{\|\mathbf{v}\|} = \frac{\|\mathbf{R}\|}{\|\mathbf{r}|}$$

and R_1 is a rotation matrix,

$$R_1 = \begin{pmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{pmatrix}$$

where $\cos\phi = \frac{(\mathbf{R},\mathbf{r})}{\|\mathbf{R}\|\|\mathbf{r}\|}$, and $\sin\phi = \frac{\|\mathbf{R}\times\mathbf{r}\|}{\|\mathbf{R}\|\|\mathbf{r}\|}$.

REFERENCES

1. RONEN BARZEL and ALAN H. BARR (1988). A Modeling System Based On Dynamic Constraints. *Computer Graphics (Proc. SIGGRAPH 88)* **22** (4), pp. 179–188.
2. A. BORNING and R DUISBURG (1986). Constraint based tools for building user interfaces. *ACM Transactions on Graphics* **5** (4), pp. 345–374.
3. A. BORNING and R DUISBURG (1988). A modeling system based on dynamic constraints. *ACM Computer Graphics (Proceedings SIGGRAPH '88)* **22** (5), pp. 179–188.
4. D.G. CHRISTOPHERSON and R.V. SOUTHWELL (1938). Relaxation methods applied to engineering problems. *Proceedings of the Royal Society A* **168**, pp. 317–350.
5. C.E. FRÖBERG (1985). *Numerical Mathematics.* Benjamin/Cummingh, Menlo Park, California.
6. N. FULLER and P. PRUSINKIEWICZ (1989). Application of Euclidean constructions to computer graphics. *The Visual Computer* **5**, pp. 53–67.
7. G. NELSON (1985). Juno, a constraint based graphics system. *Computer Graphics (Proc. SIGGRAPH 85)* **19**, pp. 235–243.
8. D. G. OLSON, T. R. THOMPSON, D. R. RILEY, and A. G. ERDMAN (1985). An algorithm for automatic sketching of planar kinematic chains. *Transactions of the ASME: Journal of Mechanisms, Transmitions and Automation in Design* **107**, pp. 106–111.
9. C.W.A.M. VAN OVERVELD and ERIK VAN LOON (1992). Hanging Cloth and Dangling Rods: A Unified Approach to Constraints in Computer Animation. *The Visual Computer* **3**, pp. 45–79.
10. T. PAVLIDIS and C. VAN WIJK (1985). An automatic beautifier for drawings and illustrations. *Computer Graphics (Proc. SIGGRAPH 85)* **19**, pp. 225–234.
11. J. PLATT and ALAN BARR (1988). Constraint methods for flexible models. *ACM Computer Graphics (Proceedings SIGGRAPH '88)* **22** (5), pp. 279–288.
12. J. PLATT (1992). A Generalization of Dynamic Constraints. *Graphical Models and Image Processing* **54** (6), pp. 516–525.
13. WILLIAM H. PRESS, SAUL A. TEUKOLSKY, WILLIAM T. VETTERLING and BRIAN P. FLANNERY (1992). *Numerical Recipes in C.* Cambridge University Press, Cambridge, New York, Victoria.
14. A. J. RUBEL and R. E. KAUFMAN (1977). Kinsyn iii: A new human-engineered system for interactive computer aided design of planar linkages. *Transactions of the ASME: Journal of Engineering for Industry*, pp. 440–448.
15. J. STOER and R. BULIRSCH (1980). *Introduction to Numerical Analysis.* Springer Verlag, New York.

16. IVAN E. SUTHERLAND (1980). Sketchpad: A man-machine graphical communication system. In HERBERT FREEMAN, editor, *Tutorial and Selected Readings in Interactive Computer Graphics*, pages 2–19. IEEE Computer Society, University of Utrecht, 1980. Reprinted from Proceedings, Spring Joint Computer Conference, 1963.
17. A. WITKIN, K. FLEISCHER and A. BARR (1989). Energy constraints on parameterized models. *ACM Computer Graphics (Proc. SIGGRAPH 87)* **21** (4), pp. 225–243.
18. A. WITKIN, M. GLEICHER and W. WELCH (1990). Interactive dynamics. *Computer Graphics ACM SIGGRAPH; special issue on 1990 symposium on interactive 3D graphics* **24** (2), pp. 11–21.