A GEOMETRIC CONSTRAINT SOLVER WITH DECOMPOSABLE CONSTRAINT SET

David Podgorelec, Borut Žalik

Faculty of Electrical Engineering and Computer Science
University of Maribor, Smetanova 17
SI-2000 Maribor
Slovenia
e-mails: david.podgorelec@uni-mb.si, zalik@uni-mb.si

ABSTRACT

Abstract: In the paper, a new constructive approach to solving geometric constraints in 2D space is presented. The main step of the algorithm is pre-processing, which transforms both, geometric elements and geometric constraints, into simpler forms, and adds redundant constraints of distances and angles by solving triangles and determining sums and differences of adjacent angles. A wide variety of well-constrained problems can then be solved by a simple technique of local propagation, and over-constrained scenes and input data contradictory to some well-known mathematical theorems can also be detected in the same phase. Only with under-constrained problems and some special well-constrained cases, an additional step of merging clusters and/or geometrical relaxation is required.

Keywords: CAD, constraint-based design, geometric constraints, geometric modelling.

1. INTRODUCTION

A geometric constraint is a relation among geometric objects that should be satisfied [Freem90]. The geometry can be described by defining relations (constraints) among particular geometric elements. Automatic solving of geometric constraints represents an interface between a declarative and a procedural description of the geometry. The user only specifies object's shape and size in a declarative way, and the system takes care of making the drawing in accordance to the specification. The user specifies what to draw not how to draw it [Sunde87].

Various methods of constraint-based geometric design were presented in literature. They can be classified into three main groups: the numerical approach, the symbolic approach and the constructive approach [Gao98a]. In the *numerical approach*, constraints are translated into algebraic equations, and various numerical techniques as Newton-Raphson iteration are used to solve these equations. The *symbolic approach* is similar to the numerical approach, but the algebraic equations are transformed in the symbolic form by employing general symbolic methods as Wu-Ritt's characteristic method [Kapur88, Gao98b] or the Gröbner basis method [Ajwa95] first. After this, the transformed system is solved numerically. In the *constructive approach*, a pre-processing is performed to transform the constraint problem into a new form that is easy to draw. If a graph is employed for this task, so-called *cycles* have to be broken. The basic idea is to split the configuration into components such that each component has no cycles and all the components can be merged together in some way [Gao98a]. Efficient method of breaking the cycles was presented by Fudos and Hoffman [Fudos97], for example. Besides graphs, techniques of artificial intelligence as searching and rule-based systems can be employed [Gao98a]. In this short paper, we cannot mention all the algorithms that we have studied, but with all of them, some of the following drawbacks were noticed:

- The set of geometric elements is not rich enough for various geometric tasks. The algorithms usually employ and constrain points and, in the best case, lines or line segments and circles.
- The constraint set should describe geometry in a natural way and enable designers to use their own design styles.
- In some cases, a system does not find a solution although it exists.

- The majority of systems are not able to handle eventual multiple solutions.
- The solution should not depend on order of adding and solving constraints. Such a constraint solver is called *variational* [Bouma95].

Our new algorithm, presented in this paper, tries to fit all these criteria. Besides this, we intend to provide consistency of all levels of geometric data presentation. The algorithm is a representative of the constructive approach. We shall see in the continuation that the most of the work is done in an extensive pre-processing phase. The pre-processing is therefore the main subject of this paper. The "real" constraint solver which positions geometric elements to required mutual and/or absolute positions actually operates in a few simple steps at the end of the process. In the last part of the paper, both, the pre-processing and the constraint solving can be traced through a practical example.

2. THE METHOD

The algorithm operates in 2D space. It was developed as a successor of two constraint-based drawing systems, implemented by our group in the past – FLEXI and BFoFD (see [Žalik96]). They both were employing a *local propagation* as the basic, the oldest and the most natural graph-based method. The local propagation mostly fails when cycles are present in the configuration, and therefore requires the pre-processing. Namely, in a cycle, a group of variables cannot be determined because each of them requires that some other variable from the group is determined first [Bouma95]. Therefore, the main goal of our new algorithm is to "clean" a graph from cycles before employing the local propagation. To satisfy all distinct goals mentioned in the introduction, the algorithm was designed in several phases:

I. PREPROCESSING:
   - I.1. mapping the visible geometry onto the auxiliary geometry,
   - I.2. decomposition of complex constraints,
   - I.3. adding redundant constraints of distances and angles

II. CONSTRAINT SOLVING:
   - II.1. creation of clusters,
   - II.2. solving constraints in particular clusters by local propagation,
   - II.3. merging the clusters,
   - II.4. absolute positioning.

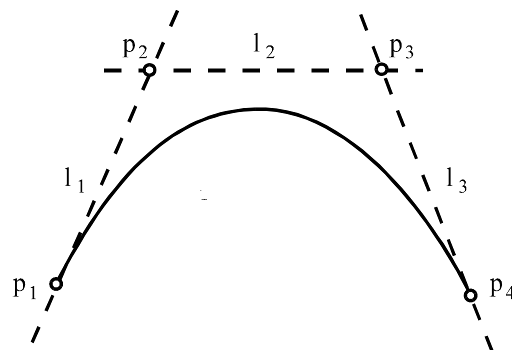
III. PREPARING THE RESULTS:
   - III.1. establishing the visible geometry from the invisible geometry.

In this paper, we deal only with constraint problems where a single cluster is obtained. Therefore, the steps of creating (II.1.) and merging the clusters (II.3.) are not described in the paper.

2.1. Visible and auxiliary geometry

With BFoFD already, a two-level organisation of geometric objects was introduced, and it is adopted in our new system as well: Bézier curves, ellipses, circles, circular arcs, and line segments form the *visible geometry*, and they are mapped onto control points and lines forming the *auxiliary geometry*. Each entity of the visible geometry has its equivalent (or more of them) in the auxiliary part, and the opposite is not necessary. The two-level organisation is important because of the following facts:

- It suffices to employ constraints that operate on elements of auxiliary geometry only. When the auxiliary geometry is constrained, the visible part is constrained, too.
- Less complex constraints can be employed, and implementation is also simplified in this way.
- The majority of the auxiliary geometry and all self-understandable constraints are created automatically by the system [Žalik96].
- New types of elements of the visible geometry can be defined in an easy way. Only the correct transformation into the auxiliary geometry has to be provided.



A Bézier curve and its auxiliary geometry
Figure 1

In Figure 1, a Bézier curve and its control polyline is presented. The auxiliary geometry consists of four points p_1 – p_4 and three lines l_1 – l_3. Besides this, six constraints establishing the topology are added automatically: On(p_1, l_1), On(p_2, l_1), On(p_2, l_2), On(p_3, l_2), On(p_3, l_3), and On(p_4, l_3).

2.2. The constraint set

After introducing the auxiliary geometry, only points and lines have to be constrained. In consequence of

this, the number of necessary constraint types has also been considerably reduced. We have implemented 25 constraint types, and they can be classified into four groups:

1. *Topological constraints* establish topology of the scene. The constraint On(p, l) that requires that the point p lies on the line l is the only representative of this group.
2. *Dimensional constraints* determine distances and angles between pairs of geometric elements. This group include constraints Distance(p_1, p_2, d), Angle(l_1, l_2, α), Parallel(l_1, l_2), Perpendicular(l_1, l_2), RelPos(p_1, p_2, x, y), Coincidence(p_1, p_2), Coincidence(l_1, l_2), and Distance(l_1, l_2, d).
3. *Positioning constraints* determine absolute positions of geometric elements i.e. point coordinates and line slopes. This group contains the constraints: Point(p, x, y), AngleValue(l, α), HLine(l), and VLine(l).
4. *Structural constraints* define relations between dimensions. In this group, we can find the constraints Symmetric(p_1, p_2, p_3), Symmetric(l_1, l_2, l_3) and different constraints which establish sums and ratios of distances or angles.

Each constraint is described by its predicate (AngleValue, for example), the list of comprised points, the list of comprised lines, and the list of numerical parameters. The length of a particular list depends on the constraint predicate. Some of the lists can be empty, but not all three of them.

2.3. A constraint problem

While our constraint solver operates with auxiliary points and lines only, *a constraint problem* is defined as CP = (P, L, CS), where

- P is the set of points $p_1,\ldots,p_m$. A point p_i, $i=1,\ldots,m$, is presented by its absolute coordinates (x_i, y_i).
- L is the set of auxiliary lines $l_1,\ldots,l_n$. For each line l_i, $i=1,\ldots,n$, its slope $\alpha_i \in [0, \pi)$ and the absolute coordinates (lx_i, ly_i) of a point lying on the line are given. The slope 0 corresponds to a horizontal line, and angles are measured in mathematical positive (counter-clockwise) direction.
- CS is the set of constraints $c_1,\ldots,c_r$ defining relations between elements of P and L.

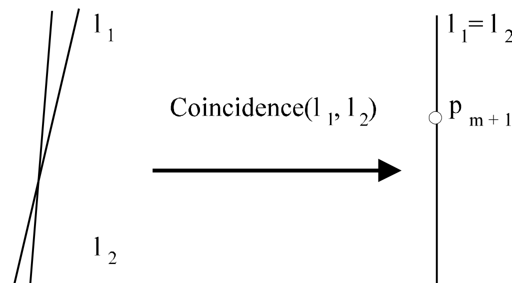
The method takes a constraint problem as input, and tries to calculate values of attributes of points and lines from the sets P and L. Initial values are obtained from the user's sketch. After solving the constraints, the solution of the constraint problem is also returned in the sets P and L. This solution is then employed to establish the visible geometry.

It is usually difficult for human designers to specify exactly geometric constraints needed to define an object unambiguously. A *well-constrained* problem has a finite number of solutions. If an infinitive number of solutions exist, a problem is *under-constrained*, and if there are not solutions, it is *over-constrained*. To define the topology, shape and dimensions of a planar scene consisting of n characteristic points, $2n-3$ independent constraints are necessary [Latha96].

3. CONSTRAINT DECOMPOSITION

In the second step of pre-processing, the constraint set CS is transformed into a form consisting only of any number of topological predicates On(p, l), dimensional predicates Distance(p_1, p_2, d), and Angle(l_1, l_2, α), no more than one positioning predicate Point(p, x, y), and also a single (or none) positioning predicate AngleValue(l, α). These predicates are directly copied from the CS into the transformed constraint set. All other predicates are defined as a conjunction of the predicates mentioned above. The transformed constraint set will be named the *minimal* constraint set CS_{MIN} in the continuation. Constraints from the set CS_{MIN} will be called *explicitely defined* constraints (just to distinguish between them and the redundant constraints inserted in the next step), and constraints from the set CS are *original* or *user-defined* constraints.

Some original constraints cannot be presented by employing the predicates from the set CS_{MIN} on existing elements of the auxiliary geometry only. They require additional geometric elements which play the same role as the elements of the auxiliary geometry, but a user need not be aware of their existence. They present so-called *invisible geometry*. They cannot be modified separately, but only by constraints that have created them. In this way, the consistency of the constraint problem is provided.



The dimensional predicate Coincidence(l_1, l_2)
Figure 2

Let us describe decomposition of the predicate

Coincidence(l_1, l_2). It requires that the lines l_1 and l_2 coincide. Additional point p_{m+1} is added first, and the following constraints are inserted into the set CS_{MIN} instead of the original predicate: On(p_{m+1}, l_1), On(p_{m+1}, l_2), Angle(l_1, l_2, 0). Of course, the number of points *m* has to be incremented by one after this substitution. The situation is shown in Figure 2.

Transformation into the minimal constraint set results in the following advantages:

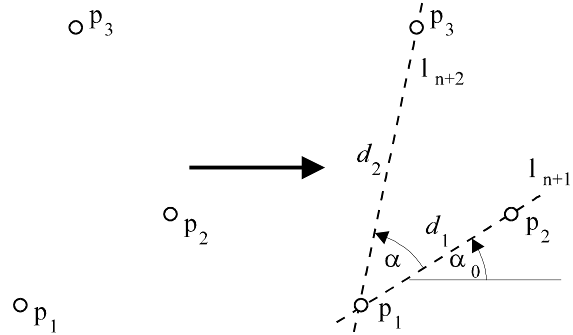
1. Employing only five constraint types simplifies design and implementation of the system.
2. Both the positioning predicates Point and AngleValue can be ignored during the solving process, and performed at the end by simple geometric transformatitions of the whole scene.
3. Constraint dependencies and behaviour of the constraint solver are much more predictable.
4. Adding new constraint types into the original constraint set is also importantly simplified. No matter how complex a new constraint is, it does not require writing and testing methods for its solving, because correct transformation already assures the desired behaviour.

Let us close this section by an example presenting the mechanism that assures that only a single predicate Point and a single predicate AngleValue can be present in the set C_{MIN}. This mechanism is implemented by the following two functions.

```
function TransformAngleValue(l, alpha);
begin
  if (num_of_fixed_lines = 0) then begin
    AddPredicate( AngleValue(l, alpha) );
    num_of_fixed_lines = 1;
  end
else  AddPredicate( Angle(l0, l, alpha – alpha0) );
(* alpha0 is the slope of the already fixed line lo *)
end;

function TransformPoint(p, x, y);
begin
  if (num_of_fixed_points = 0) then begin
    AddPredicate( Point(p, x, y) );
    num_of_fixed_points = 1;
  end
  else begin
   (*  we already have a fixed point p0(x0, y0) *)
    d = sqrt((x – x0)^2 + (y – y0)^2)    (* distance |p p0|*)
    AddPredicate( Distance(p0, p, d) );
    if (d <> 0) then begin
      add line ln+1 into the invisible geometry;
      AddPredicate( On(p, l n+1) );
      AddPredicate( On(p0, l n+1) );
      alpha = Arctg (y – y0)/(x – x0) ;
      (* the slope of the line ln+1 through p and p0 *)
      TransformAngleValue(ln+1, alpha);
      n = n + 1;
    end
  end
end
```

Let us have three points p_1, p_2, p_3, and three predicates: Point(p_1, x_1, y_1), Point(p_2, x_2, y_2), Point(p_3, x_3, y_3). The situation is handled as shown in Figure 3:



Transformation of three predicates Point
Figure 3

1. Point(p_1, x_1, y_1) is directly copied into CS_{MIN}.
2. Point(p_2, x_2, y_2) cannot be added because CS_{MIN} already contains a predicate Point. The line $l_{n+1}(p_1, p_2)$ is added, and its slope and the distance between both points are calculated. We obtain CS_{MIN} = {Point(p_1, x_1, y_1), On(p_1, l_{n+1}), On(p_2, l_{n+1}), AngleValue(l_{n+1}, α_0), Distance(p_1, p_2, d_1)}.
3. Similarly, the Point(p_3, x_3, y_3) cannot be added into CS_{MIN}. The line $l_{n+2}(p_1, p_3)$ is added into the invisible geometry, and its slope angle and the distance between the points p_1 and p_3 are calculated. But we cannot use another predicate AngleValue while one is already present in the CS_{MIN}. Instead of this, we add the predicate Angle defining the angle between the lines l_{n+1} and l_{n+2}. We obtain CS_{MIN} = {Point(p_1, x_1, y_1), On(p_1, l_{n+1}), On(p_2, l_{n+1}), AngleValue(l_{n+1}, α_0), Distance(p_1, p_2, d_1), On(p_1, l_{n+2}), On(p_3, l_{n+2}), Angle(l_{n+1}, l_{n+2}, α), Distance(p_1, p_3, d_2)}.

4. ADDING REDUNDANT CONSTRAINTS OF DISTANCES AND ANGLES

We have discussed several advantages of previous steps of pre-processing, but the main problem still remains unsolved: cycles of constraints have not been removed yet. The idea of our algorithm is to add redundant constraints of distances and angles and then to choose such a combination of constraints

which does not contain cycles. The last step of the pre-processing introduces two matrices:

The matrix of distances M_D stores the distances between all pairs of point from P. The element in the ith row and the jth column of the matrix is marked $d_{i,j}$ and presents the distance between the points p_i and p_j. The size of the matrix is $m \times m$ where m is the total number of points of the auxiliary and invisible geometry. While the distance is unsigned, the matrix is symmetric, and therefore, it suffices to use only the elements below the main diagonal. Beside the numerical value of the distance, each element $d_{i,j}$ stores its *priority* and a pointer to the line passing through the points p_i and p_j. The priority denotes a way how the value was obtained. It contains one of the following values:

0 – *the lowest priority*: the value was initialised by the approximate value from the users's sketch;
1 – *high priority*: the value was calculated by using already determined values, and cannot be changed any more.
2 – *the highest priority*: the value presents the numerical attribute of an explicitely defined predicate Distance.

Similarly, all the angles between pairs of lines are stored in *the matrix of angles M_A*. The element in the ith row and the jth column of the matrix is marked $a_{i,j}$ and presents the angle between the lines l_i and l_j. The size of the matrix is $n \times n$ where n is the total number of lines of the auxiliary and the invisible geometry. Angles are oriented, and therefore, the matrix is not symmetric. But diagonally symmetric elements are still strongly correlated: $a_{i,j} = 2\pi - a_{j,i}$. Elements of the matrix M_A also contain priorities.

The idea is to fill the matrices as much as possible. Basic operations for filling the matrices are solving triangles and determining the sums and the differences of angles sharing common edges. The matrix is full if it contains only the elements with the high or the highest priority. The elements with the high priority represent numerical parameters of *redundant* dimensional constraints. The redundant constraints play an important role because they can be employed to determine other values in both matrices, and they can be chosen for constraint solving instead of explicitly defined constraints. In this way, cycles can be avoided.

4.1. Solving the triangles

The shape and the size of a triangle are described by six dimensions: three sides a, b and c, and three interior angles α, β and γ. The angle α lies against the side a, β against the side b, and γ against the side c. We shall use the same labels a, b, c for the sides and their lengths, and α, β, γ for the angles and their values. Only three of these six dimensions are independent, and they are enough to define a triangle. Other three dimensions are calculated by using the following well-known relations:

a) the cosine statement:
   $a^2 = b^2 + c^2 - 2bc \cos \alpha$;
   $b^2 = a^2 + c^2 - 2ac \cos \beta$;
   $c^2 = a^2 + b^2 - 2ab \cos \gamma$;
b) the sine statement:
   $\frac{a}{\sin\alpha} = \frac{b}{\sin\beta} = \frac{c}{\sin\gamma}$;
c) the sum of interior angles is π:
   $\alpha + \beta + \gamma = \pi$.

In this phase of the algorithm, when the absolute coordinates are not important yet, a triangle can be defined in five different ways:

1) a side and both neighbouring angles are known;
2) a side, the angle against it and one of the neighbouring angles are known;
3) two sides and the angle between them are known;
4) all three sides are known;
5) two sides and the angle against one of them are known. Note that this case can produce two different solutions

The step of solving the triangles tests all possible triangles defined by arbitrary three points, and solves those triangles where at least three dimensions are known. A distance or an angle is known if its priority is high or the highest. If more than three dimensions are known then they have to fit some required relation (for example the cosine statement) already, otherwise the constraint problem is recognised as over-constrained. This step also detects configurations that are *contradictory to some well-known mathematical theorems*. This problem reflects in one of the following errors:

1. A side of a triangle is longer than the sum of other two sides.
2. In the sine or cosine statement, the attribute of a trigonometrical function is not in the range [-1, 1].

Sides of triangles are directly stored into the matrix of distances M_D, but angles require some additional consideration before being written into the matrix of angles M_A or read from it. Namely, the angle between the carrier lines of two sides of a triangle usually does not present an interior angle (for example α) of the triangle. It can also present one of the following three angles:

- the exterior angle of a triangle ($\pi - \alpha$),
- the difference between the full angle and the

interior angle ($2\pi - \alpha$),
- the difference between the full angle and the exterior angle ($\pi + \alpha$).

Whenever an angle from the matrix M_A is transformed to an internal angle of a triangle or the inverse transformation is done, all four angles are compared with the corresponding angle from the user's sketch, and the closest one is selected. Of course, this is correct and reasonable only if the user designs strictly enough.

4.2. Determining the sums and the differences of angles sharing common edges

Usually, a big amount of distances and angles is calculated by solving the triangles, but this operation is not always sufficient to completely fill the matrices M_D and M_A. For this reason, it is assisted by a simple operation of *determining the sums and the differences of angles sharing common edges*. This trivial operation is based on the following rule:

If the angle between lines l_1 *and* l_2 *is* α, *and the angle between lines* l_2 *and* l_3 *is* β, *then the angle between the lines* l_1 *and* l_3 *is* $\alpha + \beta$.

The rule establishes relation between three elements of the matrix of angles: $a_{i,j} + a_{j,k} = a_{i,k}$. Of course, it can be also employed to calculate the difference of two known angles sharing a common edge. Beside to the trivial situation when all three lines intersect in a common point, the rule handles alternate and corresponding angles to some known angles, provides transitivity of parallelism, or fits the relation that an external angle of a triangle is equal to the sum of both opposite internal angles.

4.3. Order of solving the triangles

To visit all the triangles, the matrix M_D has to be passed, and after that the sums and differences of angles are calculated during passing the matrix M_A. Calculated dimensions are employed to solve other triangles and calculate the sums or the differences of other pairs of angles. Some triangle that was previously recognised as unsolvable, can become solvable after determining some dimensions. For this reason, more passes of matrices are performed, and if there are not any changes in a particular pass, the step of adding redundant dimensions is terminated.

The main disadvantage of the method is its time complexity while it calculates much more redundant constraints than it is necessary to solve the cycles. Between n points, $\binom{n}{3} = \frac{n(n-1)(n-2)}{6} = O(n^3)$ different triangles are possible, and the algorithm tries to solve them all. Similarly, all combinations of three lines are tested to calculate the sums and differences of angles sharing common edges. Other steps of the algorithm are much faster.

To speed up the process and improve the interactivity, some parts of the algorithm are designed incrementally. An *incremental* constraint solver is able to take advantages of previous steps and does not calculate all the values after each user's interaction [Freem90]. For example, the majority of values from the matrices M_D and M_A determined after a particular designer's interaction need not be calculated again after the next operation.

5. CONSTRAINT SOLVING BY LOCAL PROPAGATION

After the preprocessing, the algorithm splits the sets of lines L and points P into the *clusters*, solvable with local propagation. Each cluster contains lines with known mutual angles and all the points lying on them. In this paper, we only discuss configurations where a single cluster is obtained. While the angles between all pairs of lines in a cluster are known, this situation appears exactly when the matrix M_A is full. On the other hand, the distances between pairs of points need not be determined. Regarding the priorities of distances, a cluster consists of one or more subsets of points. We name them *the connected subsets of points*. With well-constrained problems, the matrix M_D is also full, and the cluster contains a single connected subset.

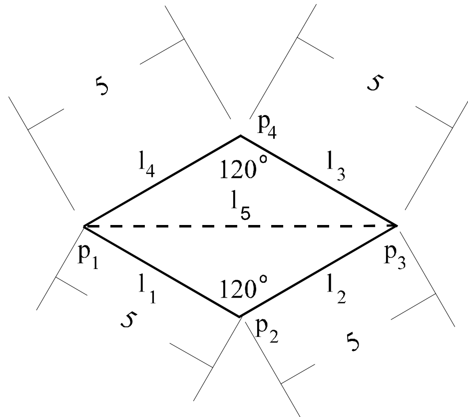
The subset of the constraint set CS_{MIN} belonging to a particular cluster contains the predicates On, Angle and Distance only. These constraints are solved in the following way.

1. The predicates Angle are treated first. The slope of the first line remains unchanged, and all other lines are rotated to establish required angles with the first line. Besides this, the lines are translated to pass through the origin, and all the points are also moved to the origin.
2. One of the points (the reference point p_R) is left in the origin and each other point p_i is moved along the line (p_R, p_i) to the prescribed distance. Together with the point, all the lines passing through it and not being moved yet, are translated. If the observed point and p_R belong to different connected subsets, a distance with the lowest priority is employed, and the translated point is chosen as the reference point for all the points of the same connected subset. In this way, a three-level hierarchical structure is obtained. The structure does not contain cycles, and a cluster is solvable for sure.

The last step of the algorithm solves eventual positioning predicates AngleValue and Point. The predicate AngleValue requires rotation of the whole scene around the origin, and the predicate Point is satisfied by translation of the desired point into the required position, and then translation of all other points and lines for the same offset.

6. AN EXAMPLE: CONSTRAINING A PARALLELOGRAM

Let us highlight the described algorithm and explain some additional details by the following example. A designer wants to construct a parallelogram shown in Figure 4. He/she inserts four line segments where each pair is sharing a common end point to form a polygon. The original constraint set is initialised by topological constraints, and the user inserts the dimensional constraints. All the constraints are directly copied into the CS_{MIN}, and we obtain: CS_{MIN} = {On(p_1, l_1), On(p_2, l_1), On(p_2, l_2), On(p_3, l_2), On(p_3, l_3), On(p_4, l_4), On(p_1, l_4), Distance(p_1, p_2, 5), Distance(p_2, p_3, 5), Distance(p_3, p_4, 5), Angle(l_2, l_1, 120°), Angle(l_4, l_3, 120°)}. The matrices M_D and M_A are initialised from the initial sketch and from the set CS_{MIN}. The situation is presented in Tables 1 and 2. For the elements determined during the initialisation, the normal font style is employed.



The well-constrained parallelogram
Figure 4

M_D	p_1	p_2	p_3	p_4
p_1	0 $_0$			
p_2	5 $_0$	0 $_0$		
p_3	**8.6** $_1$	5 $_0$	0 $_0$	
p_4	**5** $_3$	**5** $_6$	5 $_0$	0 $_0$

Filling the matrix of distances M_D
Table 1

M_A	l_1	l_2	l_3	l_4	l_5	l_6
l_1	0 $_0$	240 $_0$	**0** $_5$	**240** $_5$	**210** $_1$	**300** $_6$
l_2	120 $_0$	0 $_0$	**120** $_5$	**0** $_5$	**330** $_1$	**60** $_7$
l_3	**0** $_5$	**240** $_5$	0 $_0$	240 $_0$	**210** $_3$	**300** $_7$
l_4	**120** $_5$	**0** $_5$	120 $_0$	0	**330** $_3$	**60** $_6$
l_5	**150** $_1$	**30** $_1$	**150** $_3$	**30** $_3$	0 $_0$	**90** $_8$
l_6	**60** $_6$	**300** $_7$	**60** $_7$	**300** $_6$	270 $_8$	0 $_0$

Filling the matrix of angles M_A
Table 2

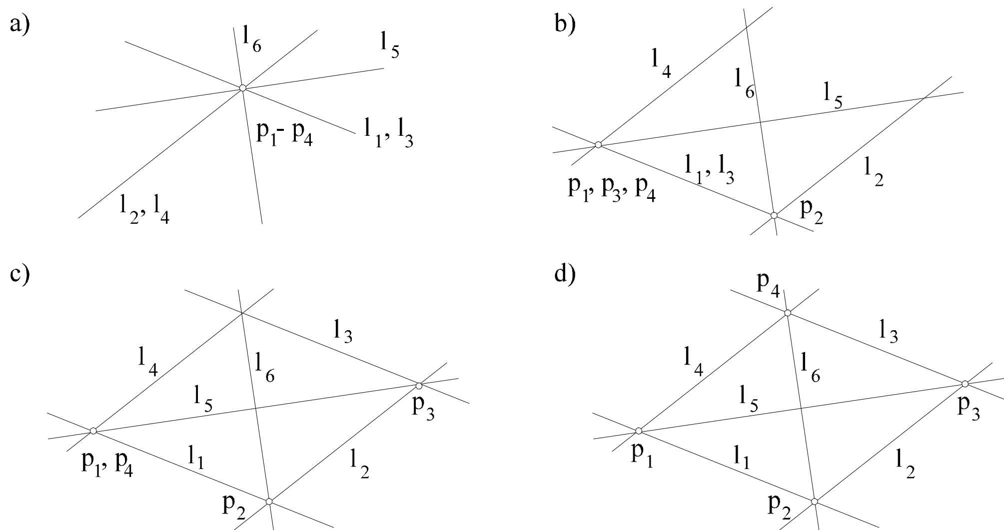
Values calculated after particular steps of adding redundant constraints are written bold. The numbers in the the right bottom corners correspond to numbering of the below steps:

1. The triangle $p_1p_2p_3$ is solved. Two sides p_1p_2, p_2p_3 and the angle between them are known.
2. The triangle $p_1p_2p_4$ cannot be solved yet.
3. The triangle $p_1p_3p_4$ is solved while two sides p_1p_3, p_3p_4 are known. Note that the side p_1p_3 calculated in the step 1 is used now.
4. The triangle $p_2p_3p_4$ cannot be solved yet.
5. The whole M_A, except the last row and column, is filled in the first iteration of determining sums and differences of adjacent angles.
6. The triangle $p_1p_2p_4$ is solved. Two sides p_1p_2, p_1p_4 and the angle between them are known.
7. The triangle $p_2p_3p_4$ is solved while all three sides (and one angle also) are known.
8. The matrix M_A is full after the second iteration of determining sums and differences.

Both the matrices are full, and the problem is well-defined. A single cluster is obtained. The next step of solving the constraints first groups all the lines and the points in the coordinate origin. The situation is shown in Figure 5a. The slope of the line l_1 is kept, and all other lines are rotated regarding the values from the matrix M_A. In Figure 5b, the point p_2 is translated along the line $l_1 = (p_1, p_2)$. Together with the point, the lines l_2 and l_6 are moved. Figure 5c shows the situation after moving the point p_3 and the line l_3. All the lines are positioned already, but the point p_4 is still in the origin. It is moved to the required position in the last step shown in Figure 5d.

7. CONCLUSIONS

In the paper, a part of our new geometric constraint solver operating in 2D space is presented. The described steps transform a constraint problem into a form that can be solved by employing local propagation. We believe that this pre-processing presents an original and efficient contribution in the field of geometric constraint solving.



Determining relative positions of points and lines by local propagation
Figure 5

Various well-constrained problems can be described and solved with rich, natural and understandable constraint set. Over-constrained problems and configurations contradictory to some well-known mathematical theorems are detected in the pre-processing phase already, and many under-constrained problems are treated successfully by the algorithm as well. Besides this, the pre-processing facilitates defining new types of geometric elements and geometric constraints. Our future work will be oriented in improving incrementality, solving particular groups of conditional constraints (not mentioned in the paper) that define a triangle in an unique way, and improving the step of merging the clusters. While the problem consists of solving the system of equations presenting distances only, we are studing different geometric relaxation methods.

ACKNOWLEDGEMENTS

This research has been supported by the Ministry of Science and Technology of Republic of Slovenia and by the British Council Slovenia inside the Valvazor/ALIS project (ALIS 63).

REFERENCES

[Ajwa95] Ajwa,IA, Liu,Z, Wang,PS: Gröbner Bases Algorithm, *ICM Technical Reports Series*, 1995.

[Bouma96] Bouma,W, Fudos,I, Hoffmann,C, Cai,J, Paige,R: Geometric Constraint Solver, *Computer-Aided Design*, Vol.27, No.6, pp.487-501, 1995.

[Freem90] Freeman–Benson,BN, Maloney,J, Borning,A: An Incremental Constraint Solver. *Communications of the ACM*, Vol.33, No.1, pp.54–63, 1990.

[Fudos97] Fudos,I, Hoffmann,CM: A Graph-constructive Approach to Solving Systems of Geometric Constraints, *ACM Transactions on Graphics*, Vol.16, No.2, pp.179–216, 1997.

[Gao98a] Gao,X–S, Chou,S–C: Solving geometric constraint systems. I. A global propagation approach, *Computer-Aided Design*, Vol.30, No.1, pp.47–54, 1998.

[Gao98b] Gao,X–S, Chou,S–C. Solving geometric constraint systems. II. A symbolic approach and decision of Rc-constructibility, *Computer-Aided Design*, Vol.30, No.2, pp.115–122, 1998.

[Kapur88] Kapur,D, Mundy,JL: Wu's Method and Its Application to Perspective Viewing, Kapur,D, Mundy,JL (eds.), *Geometric Reasoning, Elsevier Science*, 1988.

[Latha96] Latham,RS, Middletich,AE: Connectivity analysis: a tool for processing geometric constraints, *Computer-Aided Design*, Vol.28, No.11,pp.917–928, 1996.

[Sunde87] Sunde,G: A CAD System with Declarative Specification of Shape, *Eurographics workshop on Intelligent CAD Systems*, Noordwijkerhout, The Nederlands, 1987.

[Žalik96] Žalik,B, Guid,N, Clapworthy,G: Constraint-based Object Modelling. *Journal of Engineering Design* Vol.7, No.2, pp.209–232, 1996.