[23] Interval-Based Geometric Reasoning in a Parallel Network

Mark J L Orr and Robert B Fisher

Department of Artificial Intelligence
University of Edinburgh, Edinburgh EH1 2QL, UK

Reprinted, with permission of Butterworth Scientific Ltd, from parts of *Image and Vision Computing*, 1987, **5**, 100-106 and *Image and Vision Computing*,1988, **6**, 233-238

1. Introduction

Reasoning about geometry is a key process in visual perception. Not only is the discovery of geometric facts often the goal of a perceptual act (*where is the chair?*) but such facts can be used to aid the attainment of other goals such as identification (*this is a chair because it has four legs and a seat in all the right places*). A geometric reasoner inside a vision system is a kind of *quantity knowledge base* in the terms of (Davis 1987), receiving constraints from the vision system along with requests to draw inferences from them (*where in space is ...?, where in the image is ...?, can this be a ...?*).

The design of a geometric reasoner for computer vision can be split into three stages (Orr and Fisher 1987). The first stage consists in identifying the tasks to be delegated to the reasoner by the vision system. The second is the design of abstract data types and their associated operations which can carry out these tasks. The third and final stage is the design and testing of an implementation of the abstract types.

This paper reports our work using this method of design. In section 2 we discuss mainly tasks and abstract data types while also making reference to previous attempts at implementing geometric reasoning. The next section introduces a new parallel method of doing SUP/INF arithmetic which is the basis of our current implementation. Section 4 reveals how the reasoner was tailored to deal with the particular set of constraint types (which we catalogue) coming from our own vision system, or any similar, which uses 3D models and 3D data.

2. Tasks and Data Types

We divide geometric reasoning into three aspects: tasks, data types and implementation. The first aspect deals with the various tasks which seem appropriate for a vision system to delegate to a geometric reasoning package. The second involves the ideal data types and operations required in order to carry out these tasks and the third concerns the machinery of implementation.

Before proceeding we should mention that we make certain assumptions about the nature of the model and image entities used by the parent vision system. We assume that object models are built up from primitive geometric features (such as points, curves, surfaces and volumes) placed in a coordinate frame belonging to the model. We further assume that models are structured hierarchically, that is, complex models are built out of simpler ones by specifying the placing of the subcomponents in a frame pertaining to the aggregate. The key point about images is that they should contain entities which can correspond with the entities in the models at all levels: features (to correspond with model features), clusters of features (with simple models) and clusters of clusters of features (with complex models). How image segmentation into clusters is achieved or how model to image entity matches are hypothesised does not concern us here. We are also unconcerned whether the data is 2D or 3D: although the latter contains more information, the principles of geometric reasoning are the same for both.

Tasks

The nature of the geometric reasoning component within a vision system is characterised by the tasks which it is expected to carry out. Exactly which tasks come under the heading of geometric reasoning is debatable, but some stand out as obvious candidates. Included in these are establishing position estimates and image prediction.

Every identified feature in an image can be used to form position constraints, first because it is visible, and second by its measurable properties (location, shape, dimensions and so on). Take for example the identification of a point in the image with a point belonging to some object model. This hypothesis constrains the translation of the object in relation to the line of sight to the visible point, and some orientations of the object are excluded as they would cause the point to be obscured behind the object.

Having established a set of constraints on the position of a model from its individual features, the next step is to combine them into a single position estimate. The

detection of inconsistent constraints is an important task here to eliminate false hypotheses (formed, for example, due to erroneous feature identifications). If a consistent estimate can be found it may contain degrees of freedom (especially if there is any rotational symmetry), or there may be more than one estimate (mirror symmetry).

In a similar way, position estimates for subcomponents have to be aggregated into an estimate for the parent object, but with one important difference. Since the subcomponent estimates refer to the placement of the subcomponents and not (as in the case of features) to the placement of the parent, the subcomponent estimates must first be transformed, using their known positions relative to the parent, into estimates for the parent object.

Having established a position estimate for an object the next step is to predict the appearance and location of its features. This allows a critical comparison between the predicted and observed features, and affords a basis for reasoning about occlusion effects. Additionally, image prediction can be used to search for features not already found in the image and to subsequently refine the position estimate of the object on the basis of any new information obtained. As an example, suppose an estimate of the position of a bicycle is obtained from the positions of two coplanar wheels at the correct distance apart, and then used to predict the location and appearance of the saddle and handle bars. Using this prediction, the image is then searched for these subcomponents with the following questions in mind: If found, are they where they should be and can they be used to refine the position already obtained from the wheels? If not found, can their absence be explained?

Two points need mentioning here which complicate matters. Firstly, predicted features are not necessarily pixel type entities. Although observed features are derived from pixel based information, they are normally described in terms of symbolic entities such as points, lines, surfaces and so on. Image prediction must be capable of handling both pixel and symbolic descriptions. Secondly, real images are formed from objects which have exact positions in the world, but prediction involves objects whose positions are only estimated and must reflect this by being able to form uncertain descriptions.

We now come to the second level of description of our geometric reasoner - the abstract data types and operations required in order to carry out the tasks outlined above.

Positions

The first and most obvious requirement is a data type for representing positions. Positions are traditionally represented by six independent quantities, three translational and three rotational. Unfortunately, this representation is not adequate for our purposes, for two reasons. Firstly, because we want to model objects which have flexible attachments, we need to be able to represent positions with degrees of freedom, and secondly, because a certain amount of uncertainty is present in image measurements we also wish to represent positions which are uncertain.

In what follows we will be giving some simple data type specifications (Guttag, Horowitz and Musser 1978) using the operators FRAME and PLACED (capital letters will be used for all operators). Both operate on members of the set Position and return members of the set Model. The latter includes the special 'models' World and Camera so that we can have world centered and viewer centered coordinate systems as well as relative positions between models. The functionality of FRAME and PLACED are written:

FRAME: Position $\rightarrow$ Model

PLACED: Position $\rightarrow$ Model

In other words, FRAME tells us which object the given position is with respect to and PLACED tells us what object is at the specified position.

Both FRAME and PLACED are termed *observer* functions because they reveal a single aspect of a multifaceted object. Other observer functions would reveal information about particular position parameters and would map to pairs of real numbers (to denote a permitted range) or perhaps to the mean and standard deviation of a Guassian probability distribution. In the next subsection we will introduce some *constructor* functions which generate instances of the Position data type from other types or from other positions.

Estimating Positions from Features

Each pairing of a model to a data feature produces constraints on the position of the model to which the feature belongs. We have then an operation, LOCATE, whose inputs are the model and image features:

LOCATE: Image_feature, Model_feature $\rightarrow$
Position $\cup$ {undefined}
for all
$f_i \in$ Image_Feature &
$f_m \in$ Model_Feature:

let p = LOCATE(f_i, f_m)
if p $\neq$ undefined

FRAME(p) = Camera &
PLACED(p) = m

(f_m belongs to model m).

As well as the functionality of the LOCATE operator we have stated a rule which must always apply, *viz.* that LOCATE always places the model to which the feature belongs relative to the camera frame (because the geometric features of the image are in the Camera frame), unless an illegal pairing has been attempted, an image surface with a model edge for instance, when the result is undefined. The latter possibility shows why the range of the LOCATE function has to include the undefined object.

Merging Positions

In general models consists of more than just a single feature. A surface model, for instance, might consist of several curve features to represent its boundary and two vectors for its principle axes of curvature. If some or all of these features have been identified and produced constraints on the position of the surface then there must be some way of verifying consistency and merging the separate estimates into one.

Consequently, we need an operation MERGE which operates on a set of positions and returns a position. If #(Position) is the power set of Position (the set of all possible subsets of Position) then:

MERGE: #(Position) → Position ∪
{undefined, inconsistent}
for all m_1, m_2 ∈ Model, S ∈ #(Position):

if q ∈ S → FRAME(q) = m_1 &
PLACED(q) = m_2
then
let p = MERGE(S)
if p ≠ inconsistent

FRAME(p) = m_1 &
PLACED(p) = m_2

else MERGE(S) = undefined

The result is only defined when all the input positions have the same coordinate frame and refer to the same object. Another special device - the inconsistent object - is used to signal that the positions in S are contradictory, that is, when the intersection in 6D parameter space of the volumes corresponding to the elements of S is empty.

Transforming Position Constraints

Often we know the position of two objects relative to one another, perhaps because they are parts of the same model assembly or because there is *a priori* knowledge (e.g. the position of the camera in the world). Suppose we know the position of object A relative to object B, and consider two different problems. First, if we know the position of A in the frame of some other object C, what is the position of B in this frame? Second, if instead we know the position of C in A's frame, what is the position of C in B's frame? These problems require the operations TRANSFORM and INVERSE which obey the following rules in relation to the operators FRAME and PLACED.

TRANSFORM: Position, Position →
Position ∪ {undefined}

INVERSE: Position → Position
for all p, q ∈ Position:

let t = TRANSFORM(p, q)
if PLACED(p) = FRAME(q) then

FRAME(t) = FRAME(p)
PLACED(t) = PLACED(q)

else t = undefined
for all p ∈ Position:

let q = INVERSE(p)

FRAME(q) = PLACED(p) &
PLACED(q) = FRAME(p)

Now if we represent by X/Y a position whose FRAME is X and whose PLACED object is Y, our two problems can be written as:

First problem: know A/B and C/A, want C/B:

C/B = TRANSFORM(C/A, A/B)

Second problem: know A/B and A/C, want B/C:

B/C = TRANSFORM(INVERSE(A/B), A/C)

Image Prediction

Subsumed under the heading image prediction are a number of operations, ranging from the simple to the complex, and differing by what is being predicted. Simple predictions include feature properties (the projected length of an edge for example) and visibility (whether something can be seen). The most complicated prediction would be the whole image, pixel by pixel.

The operation to be performed in any given prediction task depends not only on the task but also on the nature of the object whose image is to be predicted. To predict whether a plane surface is front-facing merely requires the calculation of surface normal projected along the line of sight. For non-planar surfaces something more complicated has to be done. We abstract all such predictions into the operation PREDICT:

PREDICT: Model_feature, Position ->
Pred_feature

where Pred_feature is a separate data type from Image_feature because it must incorporate uncertainty due to positions which are only estimated.

Finally, we illustrate the use of the operators we have introduced. Suppose the vision system and the geometric reasoner together have hypothesised and located an object based on some subset of its constituent surfaces. The position of any of its surfaces can be obtained by TRANSFORMing the known (from the model) position of the surface relative to the object by the estimated position of the object relative to the camera. Suppose that one of the object's surfaces which has not yet been found, but whose position has been estimated, is PREDICTed to be visible. The vision system then conducts a search for this image feature. If it cannot be found then some explanation for its absence (e.g. occlusion) is required if the object hypothesis is to stand up. If it is present in the image, the LOCATE

operator can estimate its position which can then be TRANSFORMed into a new estimate for the object position. The hypothesis can then fail if this estimate does not MERGE successfully with the original.

Review

Part of the motivation for an abstract specification of geometric reasoning is to make explicit the important implementation decisions. On the one hand, there may not be easy solutions to some of the problems posed in the specification. On the other, it might be possible to relax the requirements of the specification so as to permit a particular implementation solution but still retain an acceptable level of competence. Each practical system is made interesting by its own particular set of compromises between what is desired and what can be achieved. We next review some existing geometric reasoners, and discuss them in the light of the previous sections.

Our own vision system IMAGINE (see "The design of the IMAGINE II scene analysis program" in this volume) uses 3D image data and 3D models. Its current geometric reasoning engine is described below in sections 3 and 4. The old version used intervals for the six position parameters to represent uncertainty. MERGEs were done by intersecting the intervals. TRANSFORMs were achieved by partitioning the intervals, taking means, transforming each possible combination of six means by matrix multiplication and then finding the smallest rectangular box enclosing the transformed points in six dimensional parameter space. Problems were encountered with the use of slant and tilt for rotations because when zero was in the range of slant values the tilt became unbounded (we now use quaternions for representing rotations). Some of the problems with the old method were caused by the LOCATE operation which did not handle data errors well, could only operate on surface patches and required the location of the patch central point so that difficulties arose when a patch was partially occluded.

ACRONYM (Brooks 1981) is a vision system which uses 2D data and 3D model primitives. Positions are represented by variables, one for each of the six degrees of freedom. Constraints on positions are formed by relating expressions in the variables to uncertain quantities measured from the image. A constraint manipulation system (CMS) processed multiple constraints symbolically leading to bounds on the individual position parameters. The operations MERGE, TRANSFORM and PREDICT (sections 3.3, 3.4 and 3.5) were achieved by, respectively, unioning restriction sets, simplifying symbolic compositions of positions and bounding expressions in variables. Underpinning the geometric reasoner is SUP/INF or interval arithmetic which the current IMAGINE reasoner also uses although it is implemented differently and has certain advantages over ACRONYM (see section 3).

RAPT (Popplestone, Ambler and Bellos 1980) is an off-line programming language for planning robot assembly tasks. Embedded in RAPT is a geometric reasoner which takes assertions about the relative positions of bodies from the programming language and infers their Cartesian positions. In the programming language, relations are stated rather like a human might state them, e.g. *face 1 of body A is against face 2 of body B*. Internally, however, a relation is represented by a symbolic composition of translations and rotations which may involve variables to represent the unconstrained degrees of freedom. A graph is formed whose nodes are the bodies in the assembly task and whose arcs are the relations between the bodies. If at least two independent paths can be found between two nodes then there is an equation which relates two or more independent expressions for the body's position and some or all of the variables (degrees of freedom) can be eliminated. The difficulties with RAPT relations for our purposes are the absence of any mechanism for incorporating uncertainty and the restriction to relations which lead to algebraic equalities and not to inequalities.

In (Faugeras and Herbert 1983) models and images have features but are unstructured. The features (model and image) are planar surface patches characterised by surface normal and distance from the origin. The problem is to find the transformation that best maps the model features into the image features. It is interpreted as a least squares problem and elegantly solved by reducing it into the problem of finding the eigenvalues of a symmetric 3 by 3 matrix. Their work can be viewed as an implementation of the MERGE operator for a particular class of feature.

An alternative way of treating uncertain positions, reported by (Durrant-Whyte 1987) has come out of work in stochastic geometry (Harding and Kendall 1974) Durrant-Whyte tackles the problem of applying (exact) coordinate transformations to uncertain positions. Uncertainty is represented by a probability distribution in parameter space, and its functional form is chosen to be Gaussian because the transformation of a Gaussian distribution is also a Gaussian (though only to an approximation). Thus, all that is needed to specify an uncertain position are the mean parameter values and a variance-covariance matrix. The latter may be transformed by multiplication with the matrix representing the (exact) relation between the two coordinate frames. The method is not a full implementation of the TRANSFORM function since the transforming position must be exact.

3. A Network Implementation

Of the various implementation alternatives discussed above algebraic inequalities of the type used in the CMS of ACRONYM (Brooks 1981) have several desirable properties. They provide a uniform mechanism for a variety of relationships including *a priori* relationships (e.g. the position of the camera), and model variations (variable dimensions or flexible attachments). Such constraints involve known (observable) and unknown quantities and estimates are sought for the unknowns. To find such estimates the CMS symbolically combined and simplified multiple constraints until the expressions they bound reduced to single quantifiers. This had drawbacks of a high cost for symbolic processing and an inability to properly handle non-linear constraints. Below we describe a new implementation of this method which

confronts these problems.

The basic constraint solving method we use is Bledsoe's SUP-INF algorithm (Bledsoe 1975), later refined by Shostak (Shostak 1977) and Brooks (Brooks 1981). Constraints are expressed in the form:

$$x_i \leq f_i$$

or

$$x_i \geq g_i$$

where the x_i are members of a set $\{x_1, x_2, ...x_n\}$ of variables and f_i and g_i are values or expressions involving some or all of the x_i. A solution of the constraints would be a substitution of real values for the variables that maintained the truth of each inequality. The goal of the algorithm, for a given set of constraints, is:

(1) to decide whether the set of possible solutions is empty,

(2) to find bounds on the value that a given expression (involving some or all of the x_i) can attain over the solution set.

The algorithm is based on the recursive application of the functions SUP and INF on the expression to be bound and its sub-expressions. SUP returns an upper bound (supremum) and INF a lower bound (infimum). In Brooks' (Brooks 1981) program the simplification of constraints and the application of SUP and INF was handled by symbolic manipulation at run time. We present a new implementation of the SUP-INF method that transfers the cost of symbolic manipulation from run-time to compile-time, improves the performance of the algorithm for non-linear constraints and has a natural parallel structure.

Structure of the network

The implementation has the structure of a network with nodes and connections. There are two types of nodes: value nodes and operation nodes. The value nodes acquire numerical SUP and INF bounds on their associated algebraic variable or expression. The bounds are computed from connections with other value nodes or with operation nodes that receive inputs from other value or operation nodes. Each time new bounds are computed the change propagates over the network causing other nodes to acquire new bounds. The changes become smaller as the bounds get closer and the network converges asymptotically to a stable state when the desired bounds on variables or expressions of interest can be extracted from the associated value nodes.

Operation nodes implement a simple unary or binary function and take their inputs from value nodes and other operation nodes. The operators implemented are: {"+", "-", "*", "/", "sup_of_max", "sup_of_min", "inf_of_max", "inf_of_min", "extract_sup", "extract_inf", "constant", "cos", "sin", "sqrt", "<", "≤", ">", "≥", "and", "or", "select", and "enable"}.

Network Creation

A network is constructed by linking together several network fragments or modules. Each module represents a particular instance of a common constraint type and there may be more than one module of the same type in the network. The structure of modules is defined by an off-line compilation process. Consequently, an on-line program that uses the network, such as a geometric reasoner, only has to connect instances of the appropriate modules to solve the problem at hand.

A module is compiled from a list of algebraic inequalities such as:

$$x \leq y + z$$

The inequalities are written by a human programmer after due consideration of the 'problem' that the module 'solves'. An example from geometric reasoning is finding the rotation that maps one pair of direction vectors to another. The relations between all the variables occurring in the problem are expressed as inequalities. If an equality is encountered then it is split into two inequalities:

$x = expr$ becomes:

$x \leq expr \;\&\; x \geq expr$

If a product is encountered, then it is split into four inequalities involving the signed reciprocal ('srecip') function:

$x * y \leq z$ becomes:

$x \leq z * srecip(y)$

$y \leq z * srecip(x)$

$x \geq -z * srecip(-y)$

$y \geq -z * srecip(-x)$

This function has the definition:

srecip(x) = if x > 0 then 1/x
else 'undefined'

and consequently has the effect of turning off and on constraints according to the sign of its argument. Explicit conditionals are also possible such as:

if (z == 0) then $x \leq y$

so that the constraint $x \leq y$ is only turned on if z is zero (meaning $INF(z) \leq 0 \leq SUP(z)$).

Recursive constraints are allowed such as:

$$x^2 \geq 1 - y^2$$

which becomes:

$$x \geq (1 - y^2) * srecip(x)$$

$$x \leq (y^2 - 1) * srecip(-x)$$

but are treated differently by bound simplification (see below).

A parallel network based case construction is needed because some operations produce different output according to conditions on their input. Two special operation types were used to implement the case structure. One is the 'enable' operation, a function of a test argument and a result argument whose output is the result only if the test is true. The other is the 'select' operation that returns the first of its arguments to become defined. Using these, the 'enable' operation turns on and off results according to their applicability (as determined by the test argument), and the 'select' operation passes through the "true" value. The logical value of the test argument is generated by using a numerical comparison operator (e.g. "<") or a logical operator (e.g. "and").

Ordinarily, an operator will not be evaluated until all arguments have values. This causes problems when using the operators that may not evaluate, such as the 'srecip' function. A problem also occurs at initial startup, because not all operators have all arguments ready, which may block the evaluation of other nodes, which may in turn block the evaluation of the operator, resulting in deadlock. The problem occurs often because the bounds on value nodes are "max" and "min" operators of (typically) many arguments.

To solve this problem, the "max" and "min" operators are evaluated differently according to whether the SUP or INF is desired. The "sup_of_max" ("inf_of_min") operator does not evaluate until all arguments are ready, because increasing the upper (decreasing the lower) bound may be necessary as other arguments become ready, and the SUP (INF) bound is only allowed to decrease (increase). However, the "inf_of_max" ("sup_of_min") operator can evaluate when one argument is ready, because later arguments either have no effect or improve the bound.

Symbolic Manipulation

Before compiling the network, the list of inequalities is checked for correct syntax, simplified and processed by the functions SUP and INF. In general this is a hard problem but the constraint manipulation system (CMS) of Brooks' program ACRONYM (Brooks 1981) at least provides some competence. We have extended this CMS to cope with square roots, powers of variables, the unsigned reciprocal function, conditionals, the undefined value and 'minus'!

Simplification is only applied to non-recursive constraints where the variable on the left hand side of the inequality does not appear anywhere in the right hand side. Recursive constraints are difficult to handle and if simplified would generally just lead to the trivial:

$$-\infty \le x \le +\infty$$

The CMS could be used directly (as in ACRONYM) by the on-line program. Measurements made by the program would add new constraints providing more scope for simplification and eventually to bounds on variables and expressions that are not measured directly. However, symbolic reasoning is computationally expensive and not suited to wide scale parallelism.

A more compelling reason for using a network is that it can iterate to better bounds over non-linear constraints than the single pass method of the CMS. Consider the following example.

$$x \le 1 + 1/y$$

$$y \ge 1 + 1/x$$

$$0.1 \le x \le 10$$

$$0.1 \le y \le 10$$

The CMS (somewhat simplified) finds:

$$SUP(x) = 1 + 1/INF(y)$$

$$= 1 + 1/(1 + 1/SUP(x))$$

When it gets to the embedded SUP(x) it uses the numerical bound 10 to produce:

$$SUP(x) = 1 + 1/(1 + 1/10)$$

$$= 1.91$$

However the network computation iterates to the (analytically) best bound:

$$SUP(x) = 1.62$$

$$= (1 + \sqrt{5})/2$$

Network Compilation

Value nodes are created for all variables occurring in the constraint list. These are connected by various operator nodes that extract values from value nodes or other operators. The connections are determined by the expressions found in the constraints. The following is a list of the actions taken by the compiler when it encounters the specified expression type:

constant:
An operation node (with no inputs) is created that supplies the given constant.

variable:
An operation node is created that extracts the SUP (or INF) of the associated value node.

plus: An operation node is created that adds the results of the recursively compiled sub-expressions.

max (or min):
SUP(max(list)) is compiled to be max(SUP(list)) (analogously for INF and 'min'). Thus subfragments for each sub-expression in the list are created and linked to a series of connected binary 'max' (or 'min') nodes. Network evaluation is different for max (or min) nodes created from SUP or INF in their use of defaults when not all argu-

ments are evaluated (which may arise from timing delays or alternative expressions being undefined). The INF max function returns a value if at least one argument is evaluated; the SUP max function only returns a value when all arguments are evaluated.

times:

SUP(A*B) is expanded to:

max(INF(A) * INF(B),
INF(A) * SUP(B),
SUP(A) * INF(B),
SUP(A) * SUP(B))

and then compiled. The same for INF(A*B) except 'max' is replaced by 'min'.

recip(E) (where E is an expression):

A test-case node is required for the reciprocal function. Test-case nodes select their output according to a test defined at compile-time and carried out at run-time. If SUP is the desired bound, the test-case construction is:

if INF(E)>0 or SUP(E)<0

then 1/INF(E)

else plus_infinity

If INF is the desired bound then:

if INF(E)>0 or SUP(E)<0

then 1/SUP(E)

else minus_infinity

srecip(E) (where E is an expression):

This is the signed reciprocal function where:

srecip(x) = if x > 0 then 1/x
else 'undefined'

If SUP is the desired bound, a test-case node is created selecting:

if INF(E)>0

then 1/INF(E)

else 'undefined'

If INF is the desired bound then the test-case construction is:

if INF(E) > 0

then 1/SUP(E)

else 'undefined'

v^n (where v is a variable and n is odd):

A sequence of 'times' operation nodes are created and linked to the SUP (or INF) of the variable. The output of each 'times' operation becomes the input to the next.

v^n (where v is a variable and n is even):

If SUP is the desired bound then sequences of 'times' nodes are created and linked to both the INF and SUP of the variable and a final 'max' node linked to the output of each sequence. If INF is the desired bound then a 'test-case' node is created selecting:

if SUP(v) < 0

then $[SUP(v)]^n$

else if INF(v) > 0

then $[INF(v)]^n$

else 0

square_root(E) (where E is an expression):

The positive square root is assumed. If SUP is the desired bound then:

if SUP(E) ≥ 0

then sqrt(SUP(E))

else 'undefined'

If INF is the desired bound:

if INF(E) ≥ 0

then sqrt(INF(E))

else 'undefined'

As the same expressions may be used more than once in different constraints in the same module, the recursive compiler uses a previous compilation for a expression if one exists, thus avoiding duplication. Another simplification is the reduction of multiple constraints to a single 'min' or 'max' function:

$v \le E_1$, $v \le E_2$, ... becomes:

$v \le \min(E_1, E_2, ...)$

A similar simplification is performed for lower bounds using the 'max' function.

To illustrate the creation of a network module, suppose we are interested in the 'problem':

$A \le B - C$

which entails the further constraints:

$B \geq A + C$

$C \leq B - A$

This list of constraints would be the input to the CMS, that would have little to simplify but would recursively apply the SUP and INF functions symbolically to find:

$SUP(A) = SUP(B) - INF(C)$

$INF(B) = INF(A) + INF(C)$

$SUP(C) = SUP(B) - INF(A)$

The compiler then produces the network shown in figure 1. This is a trivial example that even fails to compute both bounds on the parameters involved. In practice (see section 3) modules are larger and more complicated.

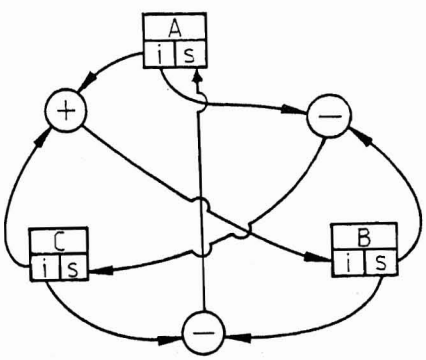

Figure 1: the network for $A \leq B - C$.

Modularisation

The run-time program constructs and evaluates its own networks according to the problems it is presented with. We assume that problems can be broken down into several parts each of which can be managed by an instance of some previously compiled module. Suppose we have the following two constraints:

$x \leq y - z$

$y \leq z - w$

A network for this problem would be constructed out of two instances of the module defined above for the constraint type:

$A \leq B - C$

and connected as shown in figure 2. The modules can be thought of as black boxes with connections to the outside world. For the first constraint the connections A->x, B->y and C->z are made, while for the second constraint A->y, B->z and C->w.

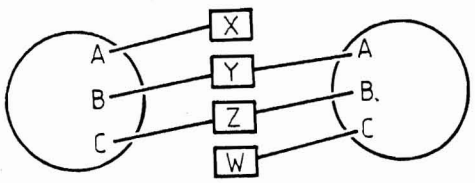

Figure 2: two connected modules.

Network Evaluation

The values at each node are computed using the values at the connecting nodes. The SUP (INF) computation chooses the minimum (maximum) of each of its current bounds and its current value. Including the current value in the calculation ensures that bounds can only get tighter. Thus if:

$SUP(A) \leq a_1$, $SUP(A) \leq a_2$, ...

then:

$SUP(A_{t+1}) = \min(SUP(A_t), a_1, a_2, ...)$

is the updating function for the supremum of A from time t to time t+1.

The following defines the evaluation functions for the different operation types:

constant:
returns a constant value

extract_sup (extract_inf):
returns the SUP (INF) of the referenced value

plus:
returns a result if both arguments are initialised:
if $+\infty + +\infty$, then return $+\infty$
if $-\infty + -\infty$, then return $-\infty$
if $+\infty + -\infty$, then indeterminate
if $-\infty + +\infty$, then indeterminate
if only one argument is infinite then return it
otherwise return the sum of arguments

minus:
returns a result if both arguments are initialised:
if $+\infty - +\infty$, then indeterminate
if $-\infty - -\infty$, then indeterminate
if $+\infty - -\infty$, then return $+\infty$
if $-\infty - +\infty$, then $-\infty$
if only the first argument is infinite then return it
if only the second argument is infinite then return its negative
otherwise return the difference of arguments

times:
returns a result if both arguments are initialised:
if one argument is $+\infty$ and the other is > 0, then return $+\infty$
if one argument is $+\infty$ and the other is $= 0$, then return 0
if one argument is $+\infty$ and the other is < 0, then return $-\infty$
if one argument is $-\infty$ and the other is > 0, then return $-\infty$
if one argument is $-\infty$ and the other is $= 0$, then return 0
if one argument is $-\infty$ and the other is < 0, then return $+\infty$
otherwise return product of arguments

recip:
returns a result if the argument is initialised:
if argument is $\pm\infty$, then return 0
if $0 <$ argument $< +\varepsilon$, then return $+\infty$
if $-\varepsilon <$ argument < 0, then return $-\infty$ otherwise return 1/argument

sup_of_max:
returns the largest of the arguments if both initialised

sup_of_min:
returns the largest of any initialised arguments

inf_of_max:
returns the smallest of any initialised arguments

inf_of_min:
returns the smallest of the arguments if both initialised

sqrt:
returns a result if the argument is initialised and greater than or equal to0:
if argument is $+\infty$, then return $+\infty$
otherwise return $\sqrt{}$(argument)

cos (sin):
returns a result if the argument is initialised:
if argument is $\pm\infty$, then indeterminate
otherwise return cos(argument) (sin(argument))

greater:
returns a result if both arguments are initialised:
if first argument is $-\infty$, then return false
if second argument is $-\infty$, then return true
if first argument is $+\infty$, then return true
if second argument is $+\infty$, then return false
if first argument > second argument, then return true
otherwise return false
(similarly for *greatereq, less, lesseq*)

and:
returns a result if both arguments are initialised:
if both arguments are true, then return true
otherwise return false

or:
returns a result if at least one argument is initialised:
if the first argument is not initialised, then return the second
if the second argument is not initialised, then return the first
if either argument is true, then return true
otherwise return false

enable:
returns the value argument if both arguments are initialised and the test argument is true

select:
returns the value of any initialised argument (arbitrary if more than one).

The networks of modules are designed to be evaluated in parallel. The whole network could be evaluated synchronously or asynchronously in a MIMD processor with non-local connectivity. Ideally, each node would be stored in a separate processor, continually polling its inputs and updating its output if appropriate.

So far we only simulate the network serially. To increase efficiency each node contains a list of its dependent nodes and when its value changes its dependents are put on a 'pending evaluation' list. When the change at a node drops below a preset threshold its dependent nodes no longer require re-evaluation. When the pending evaluation list is empty the network has reached a stable state and processing can stop. Alternatively, the network stops when inconsistency is detected when a pair of bounds cross over (the SUP of some value node becomes lower than its INF).

It is easy to show that the networks must converge asymtotically, that is, not oscillate. At any time when a new bound becomes available for some variable V, if it is a larger upper bound than the current SUP or a smaller lower bound than the current INF then it has no effect, as it makes no sense to increase the range of potential values for V. As the bounds can at most be equal (inconsistency is declared if they cross), each bound has a limit so must converge. In practice, when the change in a value is below a threshold, no change is recorded, thus forcing finite termination. Further details can be found in (Fisher 1987b).

Implementing the Geometric Reasoning Functions

The TRANSFORM function is implemented as a network module. Looked at as a black box, it has three sets of ports to the outside world representing three positions (18 parameters in total): the position being transformed, the transforming position and the resulting position. When operating in the context of an evaluating network, if any two of the sets of ports receive bounds from outside, the module will reflect the new situation by setting new bounds on the third set of ports. The INVERSE function can be implemented using TRANSFORM and the 'bi-directional' nature of network modules. Recall the second of the two problems relating to TRANSFORM and INVERSE which were discussed in section 2:

Second problem: we know A/B and A/C and want B/C:

B/C = TRANSFORM(INVERSE(A/B), A/C)

(X/Y = the position of Y relative to X)

By rearranging we can eliminate the INVERSE function:

A/C = TRANSFORM(A/B, B/C)

Now if we set up a TRANSFORM module for this problem, because of bi-directionality, it does not matter which of the three positions are constrained the other(s) will be forced into agreement by evaluation. In particular, we can always generate constraints on B/C given constraints on A/B and A/C.

In general we cannot always achieve the elimination of INVERSE, for example if:

We know A/B and B/C and want C/A, then:

C/A = INVERSE(TRANSFORM(A/B, B/C))

and rearranging will not remove the INVERSE operator. In this case we must use the identity position (I) and solve the problem with two linked TRANSFORM modules implementing the relation:

I = TRANSFORM(
TRANSFORM(A/B, B/C), C/A)

The MERGE function is carried out at the nodes linking the ports from different modules. Each port is 'saying something' about the bounds on some variable and if two or more ports are linked then they either agree (the bounds intersect and the intersection improves the estimate) or disagree. In the latter case, an inconsistency has been detected - precisely what the MERGE function was designed to do.

The functionalities of LOCATE and PREDICT, unlike the other operations, depend on the types of models and image entities used by the vision system. We therefore postpone discussion of these operators until the next section when we discuss a particular vision system.

Example

We illustrate the foregoing with an example of estimating an object's 3D orientation. Assume the following (exact) model direction vectors

$\underline{m}_1$ = (-0.51, 0.83, 0.22)
$\underline{m}_2$ = (0.68,-0.23, 0.69)

are rotated rigidly by rotation Q to give the vectors $Q(\underline{m}_1)$ and $Q(\underline{m}_2)$. Then, assume we observe two (exact) data vectors

$\underline{d}_1$ = (-0.40, 0.91, 0.04)
$\underline{d}_2$ = (-0.52,-0.67, 0.51)

Because these vectors are exact, it can be shown analytically that the rotation (represented as a quaternion) which maps $\underline{m}_1$ and $\underline{m}_2$ into $\underline{d}_1$ and $\underline{d}_2$ is:

Q = (0.73, 0.25,-0.62,-0.14)

Now suppose (more realistically) that we observe *uncertain* data vectors

$\underline{d}_1$ =
Low (-0.450, 0.892,-0.005)
High (-0.359, 0.932, 0.095)
$\underline{d}_2$ =
Low (-0.566,-0.711, 0.476)
High (-0.481,-0.637, 0.561)

These are the above exact vectors with $\varepsilon = 0.05$ radians isotropic error added. Evaluating a network which consists of a single module for transforming a pair of vectors (see section 4) the following bounds are achieved on the rotation:

Q =
Low (0.674, 0.177,-0.761,-0.209)
High (0.784, 0.342,-0.497,-0.087)

The result required 46 network update cycles with an average of 85 operation node evaluations per cycle. In a true parallel implementation (we can only simulate) the node evaluations in each cycle can be done in parallel. As ε increases the bounds on Q diverge, while as ε tends to zero the bounds converge and the solution approaches the analytic result.

If we had had three pairs of vectors instead of just two, there would be three different ways of pairing them and therefore the network for this constraint would consist of three modules. This is illustrated schematically in figure 3 where the modules are the boxes labelled "(2,0)" (the name is due to the module transforming two directions and no locations - see section 4) and the circles are the linking external variables with "Q" representing the rotation parameters.

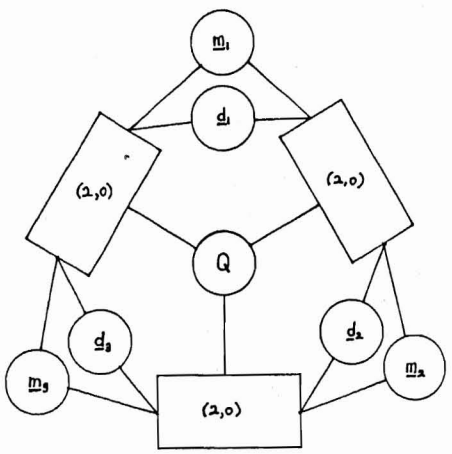

Figure 3: the network structure
for three pairs of vectors.

In general for *n* matched pairs of vectors there are *(n-1)n/2* different pairings and the complexity of the corresponding network is of order n^2. Tests have revealed the existence of at least one heuristic which can be used to reduce the complexity, which is to discard the

pairings of matched vectors which have the high uncertainty.

Related Work

The use of algebraic inequalities to represent geometric constraints derives from Brooks' ACRONYM (Brooks 1981), as does the symbolic constraint manipulation methods. The network computation is similar to the many relaxation or constraint satisfaction algorithms that are suitable for parallel processing. However, it differs from the relaxation algorithms in that it is not a probabilistic labelling computation and from constraint satisfaction in that there is reduction of an infinite continuous range of values rather than selection from a finite set of discrete values. While the network relies on connections between units, the computation is not in the distributed connectionist form where the results are expressed as states of the network. Instead, the results are the values current at selected processors.

The work presented here differs significantly from two other network based geometric reasoning systems. Hinton and Lang (1985) learned and deduced positions of 2D patterns using a distributed connectionist network, whose intermediate nodes represented object position and gated connections between iconic image and model representations. Ballard and Tanaka (1985) demonstrated a 3D reasoning network whose nodes represent instances of parameter values and whose connections represent consistency according to model-determined algebraic relationships. In both cases, patterns of network activity result, with the dominant pattern accepted as the answer (unlike here, where the result is explicit). Both systems also simultaneously select a model, which is treated separately in our analysis.

Davis (Davis 1987) has classified the types of constraint propagation systems. The system described here is an interval label constraint machine applied over full algebraic constraints (with some transcendental operations). It is used for geometric reasoning without dependence on "sin" and "cos" because rotations are represented by quaternions. His complexity analysis indicates execution times may be doubly exponential and termination may not even occur (unless forced by truncating small changes, as is done here).

Here, the complexity does not appear to be a problem, with execution time of the order of network size, presumably due to the truncation of small changes. Davis also raises the problem of disjoint parameter intervals. We believe the geometry understanding embedded in the vision program will detect most cases of this in advance (e.g. will know about n-fold symmetry) and create separate hypotheses with only single intervals.

4. 3D Models and 3D Images

In the previous section it has been shown how, with the aid of a special module for transforming positions, SUP/INF networks can be constructed which implement the geometric reasoning operators TRANSFORM, INVERSE and MERGE and the data type Position. It remains to be shown how the remaining operators, LOCATE and PREDICT, can be implemented. These operators depend on the type of models and image data used by the vision system. Our own system has 3D models (Fisher 1986 and "SMS: a suggestive modeling system for object recognition" in this volume) and 3D images and the implementation of these operators for this and similar systems is the subject of this section. For 2D images, in ACRONYM for example (Brooks 1981), the operators must account for projection from the camera frame onto the image plane as well as transformations from the model frames to the camera frame.

General Constraints

Since we are dealing with 3D geometric entities the general position constraint from a match between a model feature and an image feature involves *m* matched directions and *n* matched points. However, since any two points are equivalent to a single point and one direction, an (m, n>1) constraint can always be reduced to (m+n-1, 1) by pairing up points. Further, since two directions are sufficient to constrain rotation, an (m>2, n) constraint can be split into m(m-1)/2 separate (2, n) constraints (or less if we use heuristics). Consequently, we lose no generality if we only have network modules for the constraints (1, 0), (0, 1), (1, 1), (2, 0) and (2, 1). Three matched vectors and three matched points, for example, would be dealt with by ten (2, 1) modules linked together. There is a lot of redundancy in such a constraint but in the presence of noise the redundancy helps.

An important point to note is that two linked modules representing constraints (m_1, n_1) and (m_2, n_2) are not, in general, equivalent to one module representing the constraint (m_1+m_2, n_1+n_2). The equivalence only exists when the separate constraints are individually sufficient to fully constrain the unknown quantity. For example, for PREDICT, two (1, 0) constraints are equivalent to one (2, 0) constraint because a rotated vector is completely determined by the rotation and the vector to be rotated. However, for LOCATE, a single pair of matched vectors is not sufficient to fully constrain the rotation and so the equivalence no longer holds. Curiously, a (2, 1) module *is* equivalent to linked (2, 0) and (0, 1) modules, even though neither fully constrains position. This works because rotation is fully constrained by the (2, 0) module from which it can be 'exported' to the (0, 1) module where it combines with the pair of matched points to fully constrain translation. More details can be found in (Orr 1987a).

Thus, to summarise, we can cope with any (m, n) position constraint with some combination of four types of module: (1, 0), (0, 1), (1, 1), (2, 0). For geometric reasoning we need these four modules plus the module implementing the TRANSFORM and INVERSE operators as discussed in section 3. The number of operation nodes in each of these modules is listed in table 1.

For illustration the mathematics underlying one of the modules, the (0, 1) module (transformation of a location), is given in an appendix. Other less crucial modules may also be defined, such as those dealing with isotropic errors (location spheres, direction cones) (Fisher 1987a).

Table 1	
Module	Nodes
(1, 0)	1704
(0, 1)	1080
(1, 1)	3016
(2, 0)	3155
TRANSFORM	2088

Table 2		
Feature	Constraint	Symmetry
points	(1, 0)	none
curves		
lines	(0, 2)	2-fold
circular arc	(2, 1)	2-fold
ellipses	(2, 1)	4-fold
surface patches		
plane	(1, 0)	none
cylinder	(1, 0)	2-fold
cone	(1, 1)	none
torus	(1, 1)	none
volumes		
stick	(1, 1)	2-fold
bent stick	(1, 2)	2-fold
plate	(1, 1)	2-fold
bent plate	(1, 1)	none
blob	(3, 1)	8-fold

Particular Constraints

Our implementation of the operators LOCATE and PREDICT uses a catalogue of constraints from all legal pairings between model and image features. For each pairing the catalogue lists:

1) what vectors to extract from the model,
2) what vectors to extract from the image,
3) what modules to use, how they link to the vectors and how they link with each other.

In PREDICTing, the position of the model (in the Camera frame) is known while some of the data vectors are not. The opposite is true for the LOCATE operator - the vectors are known and an estimate is sought for the position. The same network solves both problems because it is inherently 'bi-directional'.

Table 2 lists legal pairings of features in our modeling system with image features, their constraint types and possible rotation ambiguities. Note that the boundaries of a surface patch are not part of the patch feature but separate features themselves.

Ambiguities are caused by n-fold symmetric features and are handled by the vision system (rather than the geometric reasoner) by the creation of n-fold multiple hypotheses. Each hypothesis receives a position estimate, the wrong ones are eventually eliminated by the lack of other hypotheses with which they can MERGE. More details are in (Orr 1987b) which includes some discussion of isotropic data errors and methods of overcoming partial occlusion.

5. Conclusions

The methodology we have investigated is summarised here. We start with sets of algebraic constraints associated with particular geometric relationships. (For reasoning with 3D models and 3D images there seem to be at least five of these: the four vector combination transformations of section 4 and the position transformation of section 3.) Image observables are represented by variables at this stage. These constraints are then processed by a CMS to produce symbolic bounds on each variable. The bounds are compiled into a network module where the structure of the module reflects the structure of the expressions for the bounds. All the foregoing is an off-line process and need not be repeated unless new relationships or constraints are added. The on-line program solves geometric problems with networks created by connecting compiled modules together according to the structure of each problem. When observable variables get bound to measured values the other variables (position or model parameters) are forced into consistency by evaluating the networks.

ACRONYM's CMS was optimal when producing numerical bounds on single variables over sets of linear constraints. Since we reproduce the symbolic reasoning in the network, only substituting data values later, the network must have the same performance over linear constraint sets. Over non-linear constraints, as we have here, we cannot expect optimality, but our extensions to the CMS and iterative evaluation in the network improve the performance.

The SUP/INF method is only a partial decision procedure in the sense that although consistent data will always lead to a consistent network, inconsistent data can occasionally also lead to a consistent network. We are currently engaged in evaluating the importance of this limitation and preliminary results suggest if the data errors are low (less than about 10%) or if there is enough data to over constrain the problem then the likelyhood of reaching a consistent network state with an inconsistent set of data is small (about 10% or less). Probably these 10% of incorrect evaluations involve data which is 'nearly consistent', although we have yet to demonstrate this.

We intend to apply the network formulation to the problem of camera calibration by compiling a module to do simultaneous equation solving. The coefficients in the equations will depend on the uncertain components of matched points in space and in the image and the variables will be the camera parameters. The key question will be the extent to which errors in the reference points are magnified in the camera parameters.

Further work is required to analyse the constraints from feature matches in which a parameterised range of model vectors corresponds to a measured image vector. This often occurs when a feature is partially obscured (e.g. a circular arc whose endpoints are not visible). Such constraints are only important for heavily obscured objects where there are few alternative constraints.

References

Ballard, D. and Tanaka, H., 1985, "Transformational Form Perception in 3D: Constraints, Algorithms and Implementation", Proc. 9th Int. Joint Conf. on Artif. Intel., p964.

Brooks, R.A., 1981, "Symbolic reasoning among 3-D models and 2-D images", Artificial Intelligence, **17**, p285.

Davis, E., 1987, "Constraint Propagation with Interval Labels", Artificial Intelligence, **32**, p281.

Durrant-Whyte, H.F., 1987, "Uncertain geometry in robotics", Proceedings of the IEEE Conference on Robotics and Automation, vol.2, p851.

Faugeras, O.D. and Hebert, H., 1983, "A 3-D recognition and positioning algorithm using geometrical matching between primitive surfaces", IJCAI Proceedings, p996.

Fisher, R.B, 1986, "SMS - a suggestive modeling system for object recognition", Image and Vision Computing, **5**, p98.

Fisher, R.B, 1987a, "Solving algebraic constraints in a parallel network, as applied to geometric reasoning", Working paper No. 205, Department of Artificial Intelligence, Edinburgh University.

Fisher, R.B, 1987b, "Details of a network engine for algebraic and geometric reasoning", Working paper (forthcoming), Department of Artificial Intelligence, Edinburgh University.

Guttag, J.V., Horowitz, E. and Musser, D.R., 1978, "The design of data type specifications", in "Current trends in programming methodology", IV, (Ed. Yeh, R.), Prentice-Hall.

Harding, E.F. and Kendall, D.G., 1974, "Stochastic Geometry", Wiley.

Hinton, G. and Lang, K., 1985, "Shape recognition and illusory conjunctions", Proc. 9th Int. Joint Conf. on Artif. Intel., p252.

Orr, M.J.L., 1987a, "Coordinate transforms using quaternions", Working Paper No. 204, Department of Artificial Intelligence, University of Edinburgh.

Orr, M.J.L., 1987b, "Geometric constraints in 3D computer vision", Working Paper No. 203, Department of Artificial Intelligence, University of Edinburgh.

Orr, M.J.L. and Fisher, R.B., 1987, "Geometric Reasoning for Computer Vision", Image and Vision Computing, **5**, p233.

Popplestone, R.J., Ambler, A.P. and Bellos, I.M., 1980, "An interpreter for a language describing assemblies", Artificial Intelligence, **14**, p79.

Appendix

Here, for illustration, we write out the mathematics underlying one of the geometric reasoning modules and briefly describe the modifications necessary to enable its compilation into a module. The module is for transforming a single location vector by a position P. We use quaternions (in bold letters) and vectors (underlined). If **q** is a quaternion then q_0 is its scalar part and $\underline{q}$ its vector part. "*" and "'" stand for the quaternion operations of, respectively, multiplication and conjugation (where the sign of the vector part is reversed). Let:

$\mathbf{P} = (\mathbf{r}, \mathbf{t})$

where:

r is a unit quaternion (the rotation)

t is a pure vector ($t_0 = 0$) (the translation)
and let:

u be the untransformed vector ($u_0 = 0$)

v be the transformed vector ($v_0 = 0$)
then:

$$\mathbf{v} = \mathbf{r} * \mathbf{u} * \mathbf{r}' + \mathbf{t} \qquad \text{(A1)}$$

It follows from equation A1 that:

$$\mathbf{t} = \mathbf{v} - \mathbf{r} * \mathbf{u} * \mathbf{r}' \qquad \text{(A2)}$$

$$\mathbf{u} = \mathbf{r}' * (\mathbf{v} - \mathbf{t}) * \mathbf{r} \qquad \text{(A3)}$$

$$(\mathbf{v} - \mathbf{t}) * \mathbf{r} = \mathbf{r} * \mathbf{u}$$

From this last equation we deduce that:

$$\underline{r}.\underline{u} = \underline{r}.(\underline{v} - \underline{t}) \qquad \text{(A4)}$$

$$q_0(\underline{v} - \underline{t} - \underline{u}) = \underline{q} \times (\underline{v} - \underline{t} + \underline{u}) \qquad \text{(A5)}$$

Equations A1-5 constitute the underlying mathematics for the module. To prepare this as input to the network compiler the following must be done:

1) write each vector (quaternion) equation as three (four) separate scalar equations,
2) for product expressions on the left hand side take one subexpression to the right hand side operated on by *srecip* or *recip*,
3) replace each scalar equation by two equivalent inequalities ($\leq$ and $\geq$),

For example, equation A5 is a vector equation, the first component of which is:

$$q_0(v_1 - t_1 - u_1) =$$

$$q_2(u_3 + v_3 - t_3) - q_3(u_2 + v_2 - t_2)$$

Since the sign of $(v_1 - t_1 - u_1)$ is not known *a priori* we use the *srecip* function and write:

$$q_0 = (q_2(u_3 + v_3 - t_3) - q_3(u_2 + v_2 - t_2)) * \text{srecip}(v_1 - t_1 - u_1)$$

$$q_0 = (q_3(u_2 + v_2 - t_2) - q_2(u_3 + v_3 - t_3)) * \text{srecip}(t_1 + u_1 - v_1)$$

Finally, these two equations are replaced by four inequalities.