GEOMETRIC REASONING IN A PARALLEL NETWORK

Robert B Fisher
Mark J L Orr

DAI RESEARCH PAPER NO. 451

Accepted for publication in Int. Journal of Robotics Research.

Copyright (c) R B Fisher & M J L Orr, 1989.

GEOMETRIC REASONING IN A PARALLEL NETWORK

Robert B. Fisher

Department of Artificial Intelligence
University of Edinburgh
5 Forrest Hill
Edinburgh, EH1 2QL, Scotland, UK

Mark J. L. Orr

Advanced Robotics Research Ltd.
University Road
Salford M5 4PP, England, UK

Abstract

The general principles of geometric reasoning for computer vision are discussed, and some previous attempts at implementing reasoners reviewed. A new implementation is proposed that involves interval arithmetic to cope with uncertainties and has a network structure that promotes efficiency from parallelism and accuracy from feedback. Typical constraints from a 3D model/3D image vision system are discussed and used to evaluate the network performance.

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 the correct configuration*). A geometric reasoner inside a vision system is a type of "quantity knowledge base" in the terms of (Davis 1987), receiving constraints from the vision system along with requests to draw inferences from them.

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 the associated operations that can realize these tasks. The third and final stage is an implementation of the abstract types.

This paper reports our work using this design methodology. In Section 2 we discuss the kinds of tasks done by vision systems that require geometric reasoning and review some previous attempts to implement them. The next section introduces a new parallel method of doing interval arithmetic which is the basis of our own implementation. Section 4 shows how the reasoner was tailored to deal with constraints typical of 3D model matching, which are then used in Section 5 to evaluate the performance of the network.

We follow the notation of Alefeld and Herzberger (1983) using lower case letters ($a, b, c, \cdots$) for real variables and capitals ($A, B, C, \cdots$) for interval variables. An interval A is identified with its bounds (infinum and supremum):

$$A \equiv [inf(A),\ sup(A)] \quad \text{where} \quad inf(A) \leq sup(A)$$

Bold letters are used for vector quantities such as locations and directions (with three components) and quaternions (four components).

2. Tasks and Previous Implementations

We start by considering the types of tasks that seem appropriate for a vision system to delegate to a geometric reasoning package.

Before proceeding we should mention that we make certain assumptions about 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 reference frame linked to the aggregate. The key point about images is that they should contain entities that 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. Although we deal with images having 3D information (depth images) our method can be adapted to 2D images where the constraints are weaker but the principles of manipulating them are the same.

2.1 Tasks

The geometric reasoning component within a vision system is characterised by the tasks that it is expected to perform (Orr and Fisher 1987). 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 feature 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 relative 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.

Having established a set of constraints on the position of a model from its individual features, the next step is to attempt to **merge** them into a single position estimate. The attempt can fail because of inconsistent constraints, the detection of which is necessary to eliminate false hypotheses (formed, for example, by erroneous feature identifications). Even if a consistent estimate can be found it may contain degrees of freedom (e.g. rotationally symmetric objects), and there may be more than one estimate (mirror symmetric objects).

In a similar way, position estimates for subcomponents have to be aggregated into an estimate for the parent object, but with one important difference. Feature positions refer to the placement of the objects that own them while subcomponent positions refer to their own placement. Therefore the subcomponent positions must first be **transformed**, using their known positions relative to the parent object, into positions for the parent.

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 using 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 can then be 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? If not found, can their absence be explained?

Two points need mentioning here that complicate matters. First, predicted features are not necessarily pixel entities. Although observed features are derived from pixel-based information, they are normally described as symbolic entities such as points, lines, surfaces and so on. We believe image prediction should handle both types of description. Second, real images are formed from objects that have exact positions in the world, but prediction involves objects whose positions are only estimated and should therefore be able to generate descriptions that reflect this uncertainty.

To summarize, the primary geometric reasoning tasks can be described by the following set of operations:

LOCATE - for forming primitive position constraints

MERGE - for combining position constraints

TRANSFORM - for changing constraints from one reference frame to another

PREDICT - for estimating image features using previously deduced positions

2.2 Previous Implementations

Our own vision system IMAGINE (Fisher 1989) 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 of the argument positions, replacing the subintervals by their mid-points, combining all pairing of the partitioned positions by matrix multiplication and then finding the smallest rectangular box enclosing the resultant positions in six dimensional parameter space, thus producing another set of six intervals. Problems were encountered with the use of slant and tilt for rotations because when zero was contained in the range of slant angles, the tilt angle became unbounded (we now use quaternions for representing rotations). Other problems concerned the implementation of *LOCATE* 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 obscured.

ACRONYM (Brooks 1981) is a vision system that uses 2D data and 3D model primitives. Positions are represented by interval 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) processes multiple constraints symbolically leading to bounds on the individual position parameters. The operations *MERGE*, *TRANSFORM* and *PREDICT* (Sections 2.4, 2.5 and 2.6) are achieved by, respectively, unioning restriction sets, simplifying symbolic compositions of positions and bounding expressions in variables.

RAPT (Popplestone, Ambler and Bellos 1980) is an off-line programming language for planning robot assembly tasks. Embedded in RAPT is a geometric reasoner that takes assertions about the relative positions of bodies from the programming language and infers their Cartesian positions. In the programming language, relations are stated similarly to how 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 that 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 that relates two or more independent expressions for the body's position and some or all 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 that lead to algebraic equalities and not to inequalities.

Faugeras and Herbert (1983) used models and images that 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 and is interpreted in a least squares fashion 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 *LOCATE* and *MERGE* for a particular class of feature.

An alternative way of treating uncertain positions has come out of work in stochastic geometry. Durrant-Whyte (1988a) 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 approximately Gaussian. Thus, all that is needed to specify an uncertain position are the mean parameter values and a covariance matrix. The latter may be transformed by multiplication with the matrix representing the (exact) relation between the two coordinate frames. As it stands, the method is not a full implementation of the *TRANSFORM* function since the transforming position must be exact, though it seems likely that the method can be extended to overcome the limitation.

In another article Durrant-Whyte (1988b) discussed the problem of integrating information from multiple sources. His analysis results in separate models for observers (e.g. raw sensor plus noise), observer dependencies (e.g. relative sensor reference frame positions) and observer states (e.g. changes of sensor positions over time). In this way, estimating geometric quantities (e.g. positions) can be reduced to a statistical estimation problem and thus the Kalman filter is often used.

More specific applications of the statistical approach can be found in Ayache and Faugeras (1988) and Porrill (1988). These projects improved feature and observer position estimates by linearizing parameter relationships about an initial estimate, and then applying statistical methods to integrate information from multiple observations. In particular, these were applied to the problem of estimating scene positions from stereo imagery. Additionally, linearizing the effects of perturbations from estimated positions allows one to exploit constraints between features to reduce the uncertainity in correlated perturbations. Porrill exploited this to develop constraints on lines for orthogonality, intersection, equality and maintenance of rigidity under transformation.

The last three mentioned articles, and several others, can be found in a special issue of the International Journal of Robotics Research Volume 7 Number 6, 1988.

Linearizing small perturbations (or rotations) to simplify geometric relationships has also been applied successfully to symbolic algebra constraints (as in this paper) by Fleming (1988), who analyzed toleranced part relationships when in assemblies.

Although a normal probability distribution might give a more realistic representation of typical errors than intervals, we wanted to investigate the latter because of their relative computational simplicity (i.e. avoiding the use of covariance matrices). However, we see no theoretical reason why the modular parallel approach described below cannot be applied instead to an uncertainity representation scheme based on means and variances. Indeed, a likely advantage of this extension is the ability to estimate a "best" estimate, as well as the range of allowable values.

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 (modeled or observable) and unknown quantities and estimates are sought for the unknowns. To find such estimates the CMS symbolically combines and simplifies multiple constraints until the expressions they bound reduce to single quantifiers. This has 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 that confronts these problems. (The description given in this section is an updated version of the work reported in Fisher and Orr (1988)).

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 $\{x_1, x_2, \cdots x_n\}$ of variables and f_i and g_i are values or expressions involving some or all 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 given expressions (involving some or all 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 bounded and its sub-expressions. SUP returns an upper bound (supremum) and INF a lower bound (infinum).

In ACRONYM 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.

An example of how the network can iterate to better bounds over non-linear constraints than the single pass method of the CMS is the following:

$$x \le 1 + \frac{1}{y}, \quad y \ge 1 + \frac{1}{x}$$

$$x \in X = [0.1,\ 10], \quad y \in Y = [0.1,\ 10]$$

ACRONYM's CMS (somewhat simplified) finds:

$$sup(X) = 1 + \frac{1}{inf(Y)} = 1 + \frac{1}{1 + \frac{1}{sup(X)}}$$

When the CMS reaches the embedded sup(X) it recognizes the recursion and then uses the numerical bound 10 to produce:

$$sup(X) = 1 + \frac{1}{1 + 0.1} = 1.91$$

However, our network computation iterates to the (analytically) best supremum:

$$sup(X) = 1.62 = \frac{(1 + \sqrt{5})}{2}$$

Much of the work of analysing the constraints is done once through a compilation of prototypical algebraic constraints. Then, as new constraints are formed at run-time, the compile-time results can be re-used, with the appropriate numerical values from the image and model data substituted.

3.1 Structure of the network

The implementation uses a network of 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 bounding intervals shrink and the network converges asymptotically to a stable state in which the desired bounds on variables or expressions of interest can be extracted from the associated value nodes.

Operation nodes execute a simple unary or binary function and take their inputs from value nodes and other operation nodes. The operators are: "+", "-", "*", "/",*unary_minus*, *sup_of_max*, *sup_of_min*, *inf_of_max*, *inf_of_min*, *extract_sup*, *extract_inf*, *constant*, *cos*, *sin*, *sqrt*, *abs*, "<", "≤", ">", "≥", *and*, *or*, *not*, *select*, *delay*, *guard* and *enable*.

3.2 Network Creation

A network is constructed by linking together several network fragments or modules. Each module represents a particular instance of a prototypical 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 the on-line program that uses the network only has to connect instances of the modules in an appropriate way 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 (equalities being split into pairs of inequalities). Products such as:

$$x * y \leq z$$

are split into four inequalities involving the signed reciprocal (*srecip*) function:

$$x \leq z * srecip(y)$$

$$y \leq z * srecip(x)$$

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

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

This function has the definition:

$$srecip(x) = \begin{cases} 1/x & \text{if } x > 0 \\ undefined & \text{if } x \leq 0 \end{cases}$$

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:

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

meaning:

$$\text{if } (inf(z) \leq 0 \leq sup(z)) \text{ then } sup(x) \text{ is bounded by } sup(y)$$

Recursive constraints are allowed such as:

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

which becomes:

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

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

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 for introducing a 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 first defined 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*). The *enable* operator is used with *guards* and *delays* to ensure correct synchronisation.

Ordinarily, an operator will not be evaluated until all arguments have values. This causes problems when using 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 each value node are computed by finding the maximum and minimum of (typically) many arguments.

To solve this problem, the maximum and minimum computations 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 if the other argument then acquires a value later, either there is no effect or the bound improves.

3.3 Network Compilation

The compilation process consists of translating real inequalities such as:

$$x \leq f$$

$$x \geq g$$

into constraints on the bounds of the corresponding interval variables:

$$sup(X) \leq sup(F)$$

$$inf(X) \geq inf(G)$$

If f and g are expressions then the SUP and INF functions must be recursively distributed inside until they are applied only to individual variables. Additionally, the compiler must create the network structure from the processed equations. Value nodes are created for all variables occurring in the constraint list. These are connected to various operator nodes that extract values from value nodes or other operators. (In an earlier version of the compilation (Fisher and Orr 1988) we used a version of ACRONYM's CMS to obtain initial simplified bounds on the variables. However, since normally all variables are uninstantiated at compilation, this step had little effect and was also time-consuming. Hence, we now just compile directly from 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.

add:

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

subtract:

An operation node is created that subtracts the results of the recursively compiled sub-expressions. If $sup(A-B)$ is desired then $sup(A)-inf(B)$ is the result and if $inf(A-B)$ is desired the result is $inf(A)-sup(B)$.

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 arguments 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.

multiplication:

sup(AB) 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(AB) except max is replaced by min.

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

An *enable* and *select* expression is required for the reciprocal function. The expression selects its output according to a test defined at compile-time and carried out at run-time. If SUP is the desired bound, the construction is:

$$\text{if } inf(E) > 0 \text{ or } sup(E) < 0$$

$$\text{then } \frac{1}{inf(E)} \text{ else } \infty$$

If INF is the desired bound then:

$$\text{if } \mathit{inf}(E) > 0 \text{ or } \mathit{sup}(E) < 0$$

$$\text{then } \frac{1}{\mathit{sup}(E)} \text{ else } -\infty$$

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

This is the signed reciprocal function where:

$$\mathit{srecip}(x) = \begin{cases} 1/x & \text{if } x > 0 \\ \mathit{undefined} & \text{if } x \leq 0 \end{cases}$$

If SUP is the desired bound, a network fragment is created selecting:

$$\text{if } \mathit{inf}(E) > 0 \text{ then } \frac{1}{\mathit{inf}(E)} \text{ else } \mathit{undefined}$$

If INF is the desired bound then the construction is:

$$\text{if } \mathit{inf}(E) > 0 \text{ then } \frac{1}{\mathit{sup}(E)} \text{ else } \mathit{undefined}$$

V^n (V is a variable and n is an odd number):

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

V^n (V is a variable and n is an even number):

If SUP is the desired bound then sequences of × 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 an enable and select construction is made:

$$\text{if } \mathit{sup}(V) < 0 \text{ then } (\mathit{sup}(V))^n$$

$$\text{else if } \mathit{inf}(V) > 0 \text{ then } (\mathit{inf}(V))^n$$

$$\text{else } 0$$

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

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

$$\text{if } \mathit{inf}(E) \geq 0 \text{ then } \mathit{sqrt}(\mathit{sup}(E)) \text{ else } \mathit{undefined}$$

If INF is the desired bound:

$$\text{if } \mathit{inf}(E) \geq 0 \text{ then } \mathit{sqrt}(\mathit{inf}(E)) \text{ else } \mathit{undefined}$$

Other standard functions (*cos*, *sin*, *abs*, etc.) are defined in a similar manner.

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. Other simplifications occur from the recognition of commutative expressions and the reduction of multiple constraints to a single min or max function:

$$v \le e_1,\ v \le e_2,\ \cdots$$

becomes:

$$v \le min(e_1, e_2, ...)$$

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

$$a \le b - c$$

which entails the further constraints:

$$b \ge a + c$$

$$c \le b - a$$

Recursively applying the SUP and INF functions symbolically we find:

$$sup(A) \le sup(B) - inf(C)$$

$$inf(B) \ge inf(A) + inf(C)$$

$$sup(C) \le sup(B) - inf(A)$$

The compiler then produces the network shown in Figure 1. This is a trivial example; in practice modules are larger (Section 4) and more complicated (Appendix 1).

3.4 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:

$$x \le y - z$$

$$y \le z - w$$

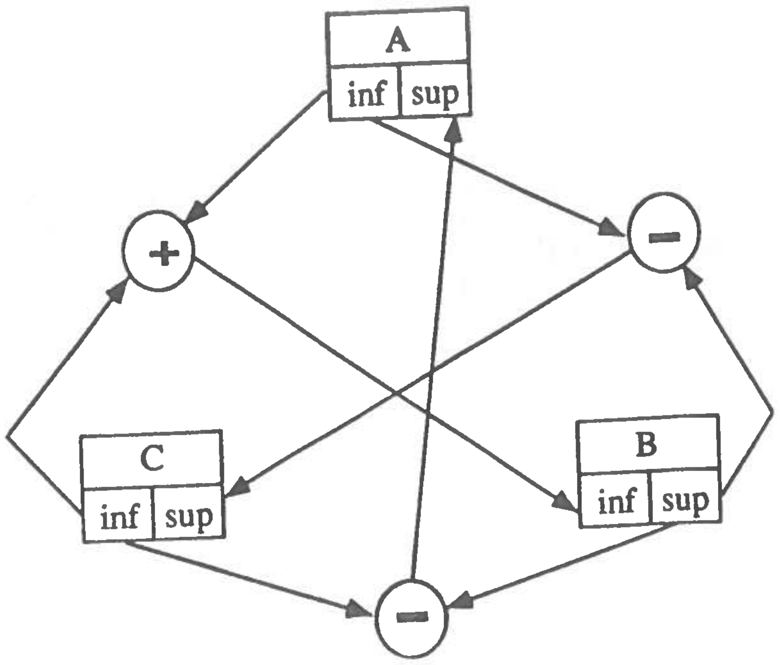


Figure 1: The network for $a \le b - c$.

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

$$a \le b - c$$

and connected. The modules can be thought of as black boxes with connections to the outside world. For the first constraint the connections $a \to x$, $b \to y$ and $c \to z$ are made, while for the second constraint $a \to y$, $b \to z$ and $c \to w$.

In practice, the networks generated as a result of geometric analysis are much more complicated. We have identified several prototypical geometric reasoning relationships and implemented them as geometric reasoning modules. These are described in more detail in Section 4. Thus, it is possible to represent a geometric relation like (i.e. the transformed model vector is close to the observed data vector):

$$T(\, v_m) \cdot v_d \ge c$$

by the connection of vector transform and dot produce constraint modules, as seen in Figure 2.

3.5 Network Evaluation

The values at each operator and value 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:

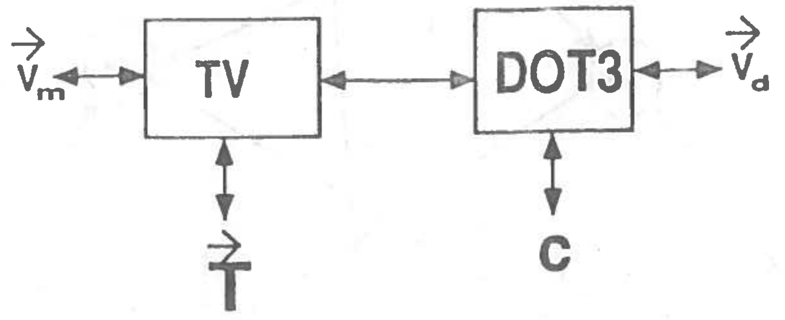


Figure 2: Two connected modules.

$$\{ a_1, a_2, \cdots \}$$

are the upper bounds on a variable a whose interval estimate is A_t at time t, 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 (where a denotes the argument of a unary function and a_1 and a_2 the arguments of a binary function).

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 $a_1 = +\infty$ & $a_2 = +\infty$ then return $+\infty$

if $a_1 = -\infty$ & $a_2 = -\infty$ then return $-\infty$

if $a_1 = +\infty$ & $a_2 = -\infty$ then indeterminate

if $a_1 = -\infty$ & $a_2 = +\infty$ then indeterminate

```
        if only one argument is infinite then return it
        else return the sum of arguments
minus:
        returns a result if both arguments are initialised:
        if $a_1 = +\infty$ & $a_2 = +\infty$ then indeterminate
        if $a_1 = -\infty$ & $a_2 = -\infty$ then indeterminate
        if $a_1 = +\infty$ & $a_2 = -\infty$ then return $+\infty$
        if $a_1 = -\infty$ & $a_2 = +\infty$ then return $-\infty$
        if only $a_1$ is infinite then return it
        if only $a_2$ is infinite then return its negative
        otherwise return the difference of arguments
unary_minus:
        returns a result if its argument is initialised:
        if $a = +\infty$ $(-\infty)$ then return $-\infty$ $(+\infty)$
        otherwise return the negative of the argument
times:
        returns a result if both arguments are initialised:
        for $i, j \in \{1, 2\}, i \neq j$:
        if $a_i = +\infty$ & $a_j > 0$ then return $+\infty$
        if $a_i = +\infty$ & $a_j = 0$ then return 0
        if $a_i = +\infty$ & $a_j < 0$ then return $-\infty$
        if $a_i = -\infty$ & $a_j > 0$ then return $-\infty$
        if $a_i = -\infty$ & $a_j = 0$ then return 0
        if $a_i = -\infty$ & $a_j < 0$ then return $+\infty$
        otherwise return product of arguments
recip:
        returns a result if the argument is initialised:
```

(where ε relates to numerical precision)

if $a = +\infty$ then return +ε

if $a = -\infty$ then return -ε

if $0 < a < +\varepsilon$ then return $+\infty$

if $-\varepsilon < a < 0$ then return $-\infty$

otherwise return 1/a

sup_of_max:

returns $max(a_1, a_2)$ 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 $min(a_1, a_2)$ if both initialised

sqrt:

returns a result if a is initialised and $a \geq 0$:

if $a = +\infty$ then return $+\infty$ otherwise return $\sqrt{a}$

cos (sin):

returns a result if the argument is initialised:

if $a = \pm\infty$ then indeterminate otherwise return $cos(a)$ ($sin(a)$)

greater:

returns a result if both arguments are initialised:

if $a_1 = -\infty$ and $a_2 = -\infty$ then indeterminate

if $a_1 = +\infty$ and $a_2 = +\infty$ then indeterminate

if $a_1 = -\infty$ then return false

if $a_2 = -\infty$ then return true

if $a_1 = +\infty$ then return true

if $a_2 = +\infty$ then return false

if $a_1 > a_2$ 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 a_1 is not initialised then return a_2

if a_2 is not initialised then return a_1

if $a_1 = true\ or\ a_2 = true$ then return true otherwise return false

not:

returns a result if its argument is initialised:

if the argument is true, then return false otherwise return true

delay:

returns its argument if it is initialised:

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 list pending evaluation. 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 occurs (*sup*(*V*) < *inf*(*V*) for some variable V).

The purpose of the guard function is to enforce synchronisation when necessary. For example, an enable node needs to ensure that its test argument is synchronised with its value argument. Otherwise, problems may occur, such as when a variable in a test expression occurs at a greater expression depth than in the result expression, thus potentially allowing an invalid result to pass through the guard before the test disables it. Hence, we use guard nodes to "trap" certain propagating values until all other activity has ceased, and then propagate all values through the guard nodes synchronously. While this reduces the parallelism inherent in the network, in practice we find that it only adds about 10% to the cycle time.

It is easy to show that the networks must converge asymptotically, 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 width of V. As the bounds can at most be equal (inconsistency occurs 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. We varied the propagation threshold from 10^{-3} to 10^{-6}, but this did not significantly affect convergence times.

3.6 Implementing the Geometric Reasoning Functions

The *TRANSFORM* function is implemented as a network module. Considered 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 transformed 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 function appropriately by setting new bounds on the third set of ports. Partial bounds on all variables can also propagate to tighten other bounds.

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 actions of *LOCATE* and *PREDICT*, unlike the other operations, depend on the types of models and image entities used by the vision system and we postpone their discussion until Section 4.

3.7 Example

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

$$\mathbf{m}_1 = (-0.51, 0.83, 0.22)$$

$$\mathbf{m}_2 = (0.68,-0.23, 0.69)$$

are rotated rigidly into vectors $\mathbf{d}_1$ and $\mathbf{d}_2$ that we then observe (without error) as:

$$\mathbf{d}_1 = (-0.40, 0.91, 0.04)$$

$$\mathbf{d}_2 = (-0.52,-0.67, 0.51)$$

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

$$\mathbf{q} = (0.73, 0.25,-0.62,-0.14)$$

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

$$\mathbf{D}_1 = ([-.43,-.41], [.91, .96], [.03, .07])$$

$$\mathbf{D}_2 = ([-.53,-.49], [-.70,-.63], [.49, .55])$$

These are the above exact vectors with $\varepsilon = 0.05$ radians random error added (see Appendix 2) and satisfy $\mathbf{d}_i \in \mathbf{D}_i$, $i = 1, 2$. Evaluating a network that consists of a single module for transforming a pair of vectors leads to the following bounds on rotation:

$$\mathbf{Q} = ([.70, .76], [.20, .30], [-.71,-.54], [-.17,-.11])$$

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 simultaneously. As ε increases the bounds on $\mathbf{Q}$ diverge with $\mathbf{q} \in \mathbf{Q}$, while as ε tends to zero the bounds converge and the solution approaches the analytic result.

If we 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 could consist of three modules. This is illustrated schematically in Figure 3 where the modules are the boxes labelled "TV2" (because the module transforms two directions and no locations - see Section 4) and the circles are the linking external variables with "Q" representing the rotation parameters. Adding an extra constraint to the above can only improve the bounds. Suppose we have:

$$\mathbf{m}_3 = (\,0.43,\,0.62,-0.66)$$

$$\mathbf{d}_3 = (\,0.66,\,0.42,\,0.62)$$

and add 0.05 radians random error to $\mathbf{d}_3$ to obtain:

$$\mathbf{D}_3 = ([\,.65,\,.66],\,[\,.39,\,.46],\,[\,.61,\,.67])$$

If we now relax the network in Figure 3 we get the tighter bounds:

$$\mathbf{Q} = ([\,.70,\,.76],\,[\,.22,\,.29],\,[-.69,-.55],\,[-.16,-.11])$$

A larger example is given in section 4.5.

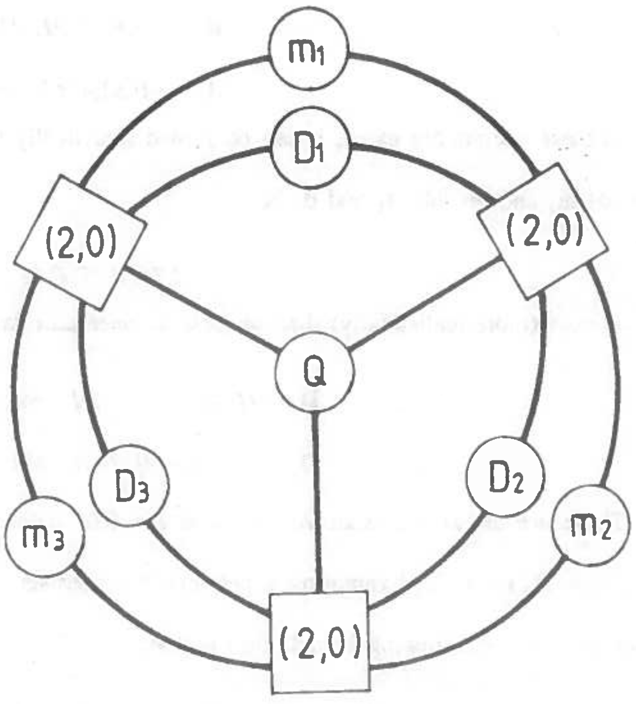


Figure 3: Network structure for three pairs of vectors.

3.8 Other Relevant Network Research

The use of algebraic inequalities to represent geometric constraints derives from Brooks' ACRONYM (Brooks 1981), as do 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 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 vision system.

Davis (Davis 1987) classified the types of constraint propagation systems. Within this framework, the system described here is an interval label constraint machine applied over full algebraic constraints (with some transcendental operations). His complexity analysis showed execution times may be doubly exponential and termination may not even occur (unless forced by truncating small changes, as is done here). However, the complexity does not appear to be a problem with our networks, presumably because we use truncation. Davis also raised the problem of disjoint parameter intervals (i.e. where there are two or more possible non-overlapping parameter ranges). We believe the geometry understanding embedded in the vision program should detect this in advance (e.g. should understand 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 SUP/INF networks can be constructed that implement the geometric reasoning operators *TRANSFORM*, 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 3D images. The implementation of these operators for this and similar systems is the subject of this section. For 2D images, as 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.

4.1 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 locations. However, since any two locations are equivalent to a single location and one direction (plus the distance between the two locations, which is not used here), an $(m, n > 1)$ constraint can always be reduced to $(m + n - 1, 1)$ by pairing up locations. Further, since two directions are enough to constrain rotation, an $(m > 2, n)$ constraint can be split into $m(m - 1)/2$ separate $(2, n)$ constraints. 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 locations, for example, would be dealt with by ten (2, 1) modules linked together.

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 enough 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 locations to fully constrain translation.

Thus, to summarise, we can cope with any (*m,n*) position constraint with some combination of four types of constraint: (1, 0), (0, 1), (1, 1), (2, 0). (These use the TV, TP, TVP and TV2 network modules discussed in Section 4.3).

To illustrate the mathematics underlying one module (the TP module transforming a location) see Appendix 1. Other less crucial modules can also be useful, such as those dealing with isotropic errors, the conversion between quaternions and other types of rotation representations, dot products and unit vector constraints (see Section 4.3).

4.2 Particular Constraints

Our implementation of the operators *LOCATE* and *PREDICT* depends on a catalogue of constraints derived 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 *PREDICT*ion, 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 1 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 (as are the curve's endpoints and the volume's bounding surfaces).

Ambiguities are caused by n-fold symmetric features and are handled in the vision system (rather than the geometric reasoner) by the creation of multiple hypotheses. Each hypothesis receives a position estimate; the erroneous ones are eventually eliminated by a lack of other hypotheses with which they can *MERGE*.

Table 1

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

4.3 Other Constraint Modules

The constraint modules described above are the most important ones for reasoning with 3D models and 3D data. However other modules for further constraint types are also useful. A module that converts pairs of locations into the direction between them (a "*P2V*" module) is required to reduce an (m, n) constraint to an $(m + n - 1, 1)$ constraint as discussed in Section 4.1 above. Figure 11 below shows a network containing instances of this module. We also use modules that directly handle isotropic errors, both for directions, based on:

$$\mathbf{d}' \cdot \mathbf{d} \geq \cos(\varepsilon)$$

and for locations, based on:

$$|\mathbf{p}' - \mathbf{p}| \leq \mu$$

where ε and μ are small and positive (see Appendix 2).

Modules for dealing with constraints from partially obscured features may also be needed. Constraints inevitably involve uncertainty in the data features (e.g. because of measurement noise), but constraints from obscured features also involve uncertainty about which model features correspond to the data. For example, suppose a line of length l is observed and identified as a model line of length $l_0 > l$, so that some fraction $\xi = (l_0 - l) / l_0$ of the length is obscured. Clearly, there cannot be a direct correspondence between $\mathbf{E}_1$ and $\mathbf{E}_2$, the measured end-points of the line in the Camera frame, and $\mathbf{e}_1$ and $\mathbf{e}_2$, the line's end-points in the local model frame. However, the corresponding model points lie in ranges that can be parameterised as follows: $\mathbf{E}_1$ corresponds with $\mathbf{e}_1 + \lambda(\mathbf{e}_2 - \mathbf{e}_1)$ and $\mathbf{E}_2$ corresponds with $\mathbf{e}_2 + (\xi - \lambda)(\mathbf{e}_1 - \mathbf{e}_2)$ for some $\lambda \in [0, \xi]$. This is an example of what we call type B constraints, that are characterised by uncertainty about the model features (whereas regular, type A, constraints have no such uncertainty). To deal with type B constraints special modules could be constructed that manage the various parameterisation schemes. Alternatively appropriate amounts of uncertainty could be added to the model parameters (which is the approach we have adopted).

Altogether, we have created geometric reasoning modules for the following relations:

SS - two scalers are close in value

PP - two points are close in location

VV - two vectors point in nearly the same direction

DOT3 - the dot product of two 3-vectors is greater than or equal a scaler

UNIT3 - enforcing a unit 3-vector constraint

UNIT4 - enforcing a unit 4-vector constraint

TP - a transformation links a pair of points (the (0, 1) constraint)

TV - a transformation links a pair of vectors (the (1, 0) constraint)

TV2 - a transformation links two pairs of vectors (the (2, 0) constraint)

TT - a transformation maps from one position to a second by a third relative position

TVP - a transformation links a pair of vectors and points (the (1, 1) constraint)

P2V - a vector can be defined by two points

QWθ - a quaternion is equivalent to an axis and an angle

4.4 Binding Degrees of Freedom

Binding degrees-of-freedom is another typical visual geometric reasoning problem (Fisher and Orr 1989), such as when estimating an intrinsic linear degree-of-freedom in a prismatic robot joint. Its under-constrained position can be modeled by aligning the degree-of-freedom with one of the coordinate axes and then using using a variable in the reference frame translation. Then, bounds on this variable can be estimated by using the network method to compute its reference frame relationship relative to its main object.

The more difficult case is a rotational degree-of-freedom, as in a robot revolute joint, which we now consider in more detail. The problem with this case is that there is no element of the rotation specification that exactly corresponds to the degree-of-freedom (unlike the translation case). Fortunately, the *cos*(θ/2) component of a rotation quaternion can be related to a joint's rotation.

Suppose the (uncertain) object positions, $\mathbf{P}_1$ and $\mathbf{P}_2$, the (exact) common axis in each model frame, $\mathbf{x}_1$ and $\mathbf{x}_2$, and two (exact) reference vectors $\mathbf{r}_1$ and $\mathbf{r}_2$, again, in the model frames, are given. The reference vectors satisfy:

$$\mathbf{x}_1 \cdot \mathbf{r}_1 = \mathbf{x}_2 \cdot \mathbf{r}_2 = 0$$

The problem is to determine the rotation ϕ that maps $\mathbf{r}_1$ onto $\mathbf{r}_2$.

The solution consists of first transforming the vectors into the Camera reference frame by using the estimated object positions and some TV2 modules, resulting in (uncertain) transformed vectors $\mathbf{X}_1$, $\mathbf{X}_2$, $\mathbf{R}_1$ and $\mathbf{R}_2$. In the Camera frame the axis vectors $\mathbf{X}_1$ and $\mathbf{X}_2$ should be the same (i.e. $\mathbf{X}_1 \cap \mathbf{X}_2$ is non-empty) while the angle between $\mathbf{R}_1$ and $\mathbf{R}_2$ defines the joint angle θ. Then, another TV2 module finds the rotation that maps $\mathbf{R}_1$ onto $\mathbf{R}_2$ about the common axis. This rotation is in quaternion form and to find the rotation angle a *QW*θ module converts the quaternion to a form involving an axis, ω, and an angle, θ. This module is based on the equations:

$$\theta = 2\cos^{-1}(q_0)$$

$$\omega = \frac{(q_1, q_2, q_3)}{\sin(\theta/2)}$$

$$\mathbf{q} = (\cos(\theta/2), \sin(\theta/2)\,\omega)$$

Figure 4 shows the network for the joint angle problem. The two TV2 modules at the top map each pair of model vectors to a corresponding pair of scene vectors. Two of these map to the common axis vector. From these we can estimate the joint rotation quaternion $\mathbf{Q}_j$. This is done by the third TV2 module in the middle of the network. It rotates the Camera frame $\mathbf{R}_1$ vector to the Camera frame $\mathbf{R}_2$ vector, while maintaining the direction of the global axis vector ω.

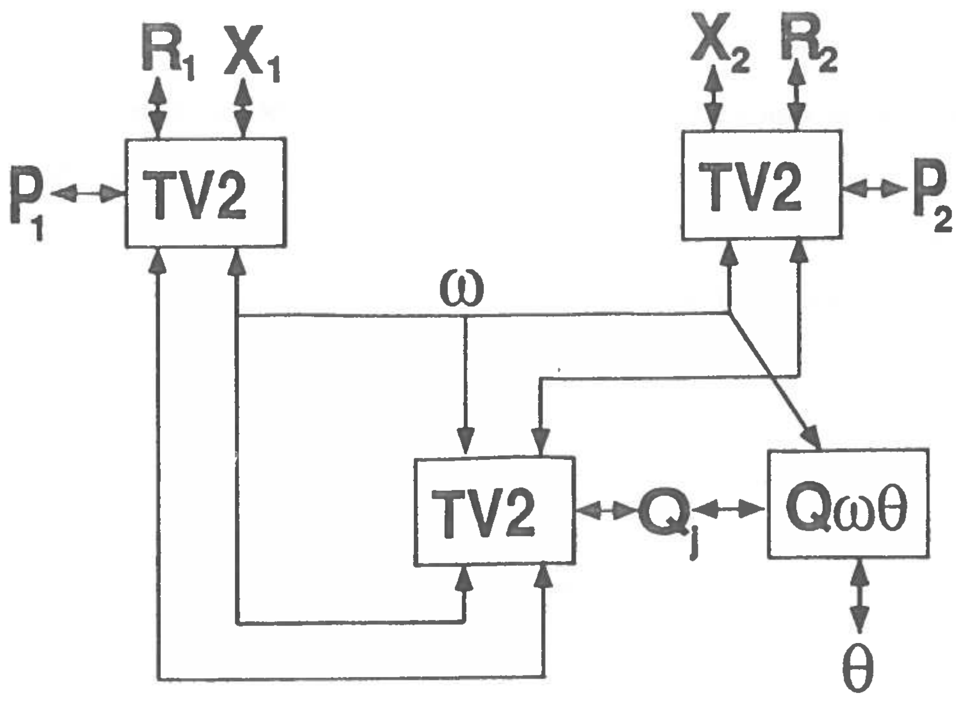


Figure 4: Network structure for the joint angle problem.

Finally, to extract the joint angle itself, we use a QWθ module that maps a rotation quaternion to a (θ,ω) axis-angle form. Here, vector **ω** is also the same as the global rotation axis, so this helps the module. (The angular output is given in the form *cos*(θ)).

With the following example

$\mathbf{x}_b=\mathbf{x}_d=(0,-1,0)$

$\mathbf{r}_b=\mathbf{r}_d=(1,0,0)$

$\mathbf{P}_B=((0.20,-0.61,-0.45,-0.61),(0,0,0))$

$\mathbf{P}_D=((0.31,-0.43,-0.39,-0.75),(0,0,0))$

with small intervals (0.00001 width) the joint network estimates the rotation: *cos*(θ)=[0.863,0.864].

4.5 A Large Network Example

Figure 5 shows the full network generated for analyzing the position of a robot in a test scene (Fisher 1989). As before, the boxes represent transformations, but there are more types used here. The TPn boxes stand for n instances of a TP module. The circular "Jn" boxes represent three identical instances of subnetworks allocated for transformations involving joint angles, which are omitted to simplify the diagram (each contains 7 network modules). The core of the subnetworks was shown in Figure 4, but some additional modules are added to align reference frames. The relative positions of objects are given by the T structures, such as T_{g-R}, which represents the position of the robot in the global reference frame. These are linked by the various transformations. Links to model or data vectors or points are represented by the unconnected segments exiting from some boxes.

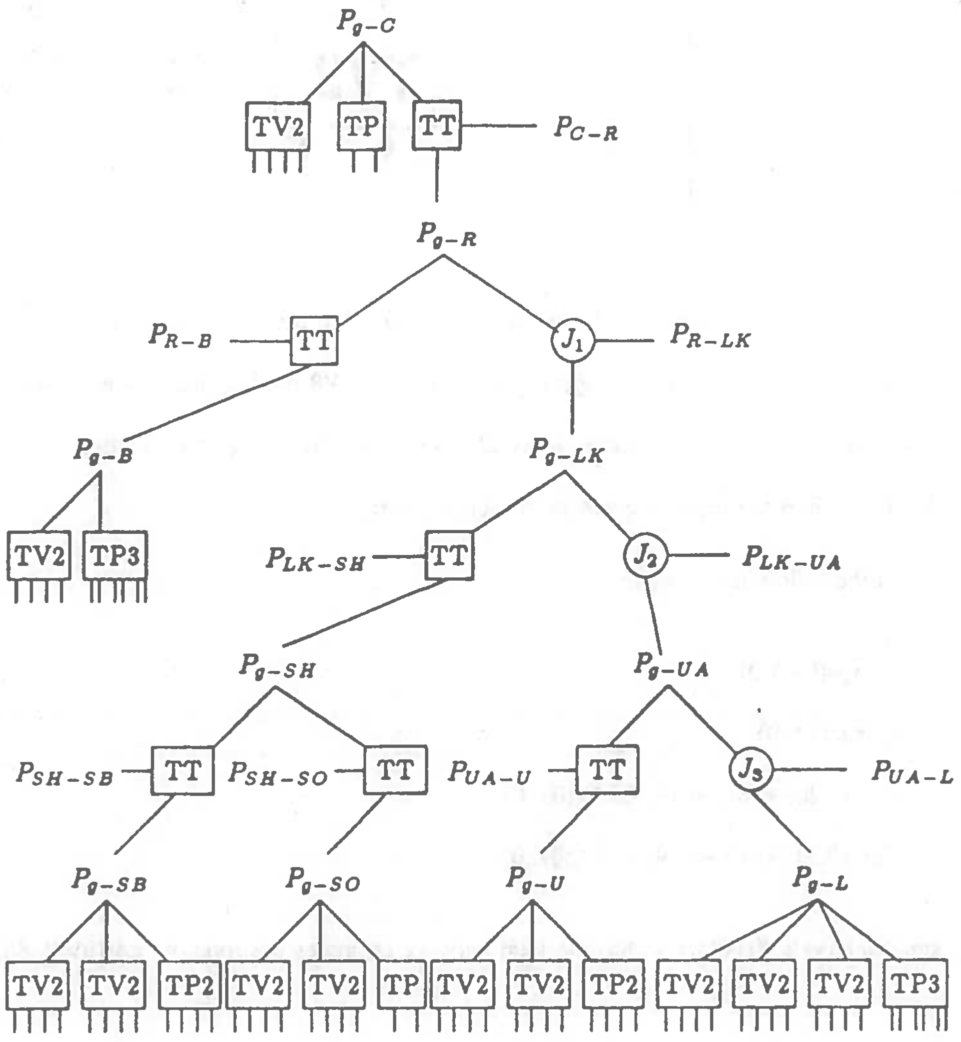


Figure 5: Network for Robot Scene

The top position T_{g-C} is the position of the camera in the global coordinate system, and the subnetwork to the left and below relates calibration features in the camera frame to corresponding ones in the global coordinate system. Below and right is the position T_{g-R} of the robot in the global coordinate system and the position T_{C-R}, of the robot in the camera coordinate system, all linked by a *TT* position transformation module. Next, to the bottom left is the subnetwork for the cylindrical robot body T_{g-B}. The "J1" node connects the robot position to the rest ("link") on the right, whose position is T_{g-LK}. Its left subcomponent is the rigid shoulder ASSEMBLY (SH) with its subcomponents, the shoulder body (SB) and the small shoulder patch (SO). To the right, the "J2" node connects to the "armasm" ASSEMBLY (UA), linking the upper arm (U) to the lower arm (L), again via another joint angle (J3). At the bottom are the modules that link model vectors and points to observed surface normals, cylindrical axis vectors, and central points, etc. Altogether, there are 61 network modules containing about 96,000 function nodes (this version of the modules did not use separate UNIT3 and UNIT4 modules).

Altogether, the network structure closely resembles the model subcomponent hierarchy, and only the bottom level is data-dependent, where new nodes are added whenever new model-to-data pairings are made, producing new constraints on feature positions.

Evaluating the complete network from the raw data requires about 1,000,000 node evaluations in 800 "clock-periods" (thus implying over 1000-way parallelism). Given the simplicity of operations in a node evaluation, a future machine should be able to support easily a 1 microsecond cycle time. This suggests that an approximate answer to this complicated problem could be achieved in about one millisecond.

Because the resulting intervals are not tight, confidence that the mean interval value is the best estimate is reduced, though the bounds are correct, and the mean interval values provide useful position estimates. To tighten estimates, a post-processing phase progressively shrinks the bounds on the intervals. Each position variable was examined to see if its interval was tight. If not, it was reduced by 10% and the new bounds were then allowed to propagate through the network. For the robot example, this required an additional 12,000 cycles, implying a total solution time of about 13 milliseconds on our hypothetical parallel machine.

Using the geometric reasoning network, the numerical results for the whole robot in the test scene are summarized in Table 2. Here, the values are given in the global reference frame rather than in the camera reference frame.

Table 2: Measured And Estimated Spatial Parameters

PARAMETER	MEASURED	ESTIMATED
X	488 (cm)	487 (cm)
Y	89 (cm)	87 (cm)
Z	554 (cm)	550 (cm)
Rotation	0.0 (rad)	0.038 (rad)
Slant	0.793 (rad)	0.702(rad)
Tilt	3.14 (rad)	2.97 (rad)
Joint 1	2.24 (rad)	2.21 (rad)
Joint 2	2.82 (rad)	2.88 (rad)
Joint 3	4.94 (rad)	4.57 (rad)

The results of the position estimation can been seen more clearly if we look at Figure 6, which shows the estimated robot position overlaying the original scene. Now, we are using the camera position of the whole robot assembly plus the estimated joint angles from Table 2. While the positions of the individual components are reasonable, the accumulated errors in the position and joint angle estimates cause the predicted position of the gripper to drift somewhat from the true position. The iterative bounds tightening procedure described above produced this result, which was slightly better than the one-pass method.

5. Performance

In this section we discuss the performance of the network for some typical problems in geometric reasoning.

5.1 Completeness

The SUP/INF algorithm is an incomplete decision procedure. If a network is inconsistent (*inf*(*V*) > *sup*(*V*) for any variable *V*) then there is definitely no solution to the problem it is solving. However, a consistent network does not guarantee that a solution exists. This is a problem because the network's ability to detect inconsistent sets of constraints relates to the vision program's ability to reject incorrect hypotheses. We would therefore like to investigate two questions about sets of constraints that

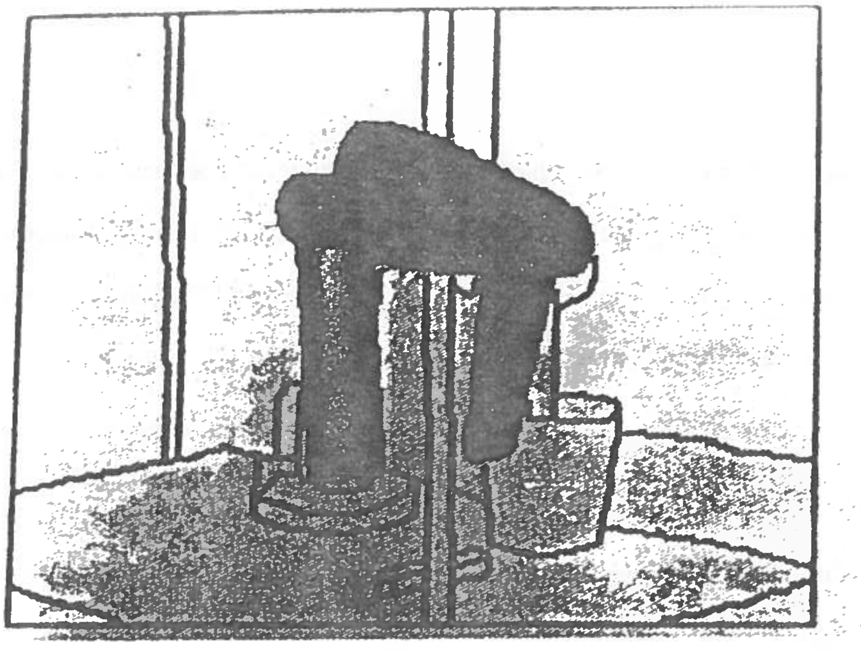


Figure 6: Recognized Complete Robot Using Network Method

have no solution but nevertheless lead to consistent networks: (1) the likelihood of such problems arising and (2) typically "how close to being soluble" (defined below) are they.

As our test case we take the problem of finding the rotation that maps a pair of 3D model direction vectors into a corresponding pair of 3D image vectors. This is not only a typical problem in geometric reasoning but also a basic one in that other constraints involving multiple matched pairs of directions and locations can be solved by decomposing them into one or more problems of this sort (Section 4). For our present purposes we assume the model vectors are known precisely while the image vectors are known only within certain errors. We further assume the errors are isotropic, that is, a nominal vector **v** is enclosed within an error cone of internal angle ε to yield the interval vector $\langle \mathbf{v}, \varepsilon \rangle_I$ (see Appendix 2).

A solution exists only if the model vectors can be rotated to lie within the error cones of their corresponding image vectors. Let $\mathbf{v}_1$ and $\mathbf{v}_2$ be the (exact) model vectors and $\langle \mathbf{v}'_1, \varepsilon \rangle_I$ and $\langle \mathbf{v}'_2, \varepsilon \rangle_I$ be the (inexact) image vectors with isotropic error ε radians. Then a solution exists if:

$$\theta' - 2\varepsilon \le \theta \le \theta' + 2\varepsilon \qquad (5.1)$$

where $\theta = \cos^{-1}(\mathbf{v}_1 \cdot \mathbf{v}_2)$ and $\theta' = \cos^{-1}(\mathbf{v}'_1 \cdot \mathbf{v}'_2)$

For different values of ε we estimate the probabilities that randomly chosen $\mathbf{v}_k$ and $\mathbf{v}'_k$, $k \in \{1, 2\}$, will result in a soluble problem and a consistent network by performing 1000 trials. The results are shown in the graph of Figure 7. (Note - this empirical probability is dependent on the particular implementation of the network modules.) The squares show the probability of network convergence while the dots show the probability of generating a soluble problem. The latter can be calculated analytically and is shown by the curve in Figure 7 which is the function:

$$p(\varepsilon) = \tfrac{1}{4}[2 - 2cos(2\varepsilon) + (\pi - 2\varepsilon)sin(2\varepsilon)] \tag{5.2}$$

The probability of the network being inconsistent given that the problem has a solution is zero for any value of ε, but the probability of the network being consistent when no solution exists is significant, as seen in Figure 7.

We now define:

$$s = \frac{|\theta - \theta'|}{2\varepsilon} \tag{5.3}$$

as a measure of "distance from solution". The problem is soluble iff $s \leq 1$. For values of s larger than but close to 1 the problem is "only just" insoluble. By again performing 1000 trials (for fixed $\varepsilon = 0.1$) we compiled histograms of values for s in trials that were predicted by (5.1) to have no solution and lead to a network state that was (a) inconsistent or (b) consistent. The results are shown in Figures 8(a) and 8(b).

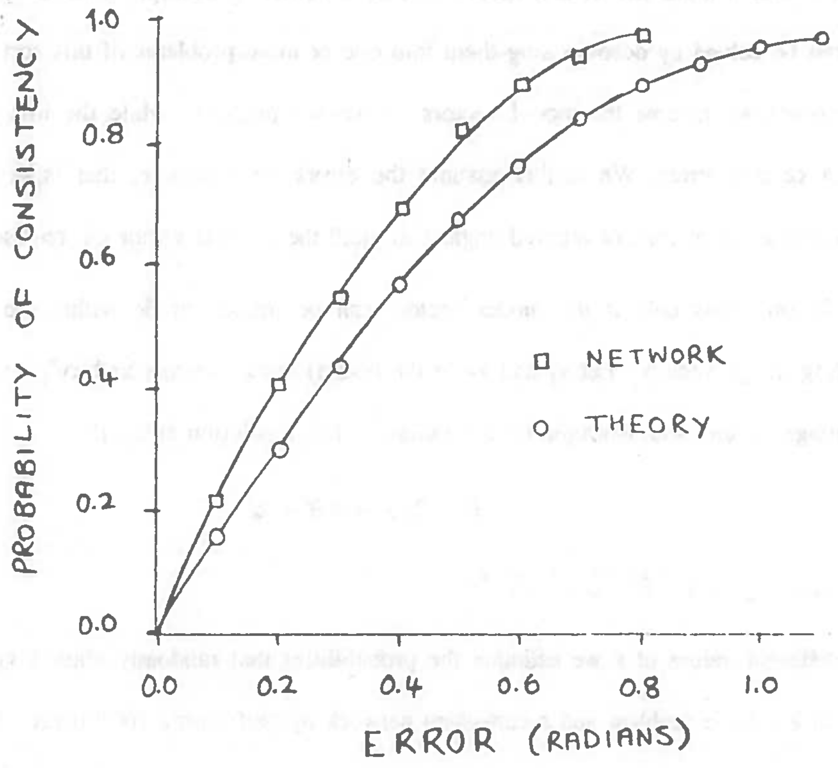


Figure 7: The probability of consistency as a function of ε

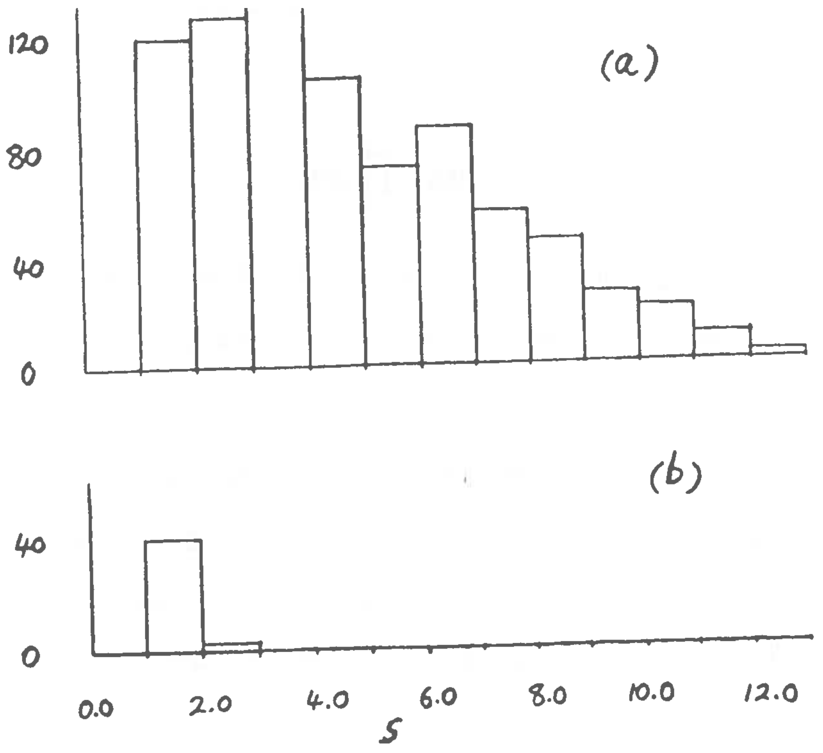


Figure 8: The distribution of s for insoluble problems with (a) inconsistent and (b) consistent networks.

Figure 8(a) shows that there is a wide distribution of s values between $s = 1$ and $s \approx 12$ for insoluble problems correctly detected by the network. On the other hand, Figure 8(b) shows that insoluble problems that the network could not detect are concentrated mostly between $s = 1$ and $s = 2$ and are consequently close to being soluble. We therefore conclude that the network will not give misleading results when problems are far from solution and that in practice the incompleteness problem has only a limited effect.

5.2 Error Build Up

The evaluation of an expression containing intervals of finite width will always lead to a result that also has non-zero width. In other words, if the input is uncertain then the output will be too. Since networks are effectively expression evaluators, involving feedback and iteration, it is sensible to ask what the relationship is between input and output errors. We investigated this for geometric reasoning networks, again using the module for rotating a pair of directions as an initial test case. We choose at random four real vectors v_1, v_2, v'_1 and v'_2 with the condition that $v_1 \cdot v_2 = v'_1 \cdot v'_2$. The condition ensures the existence of a rotation that transforms v_1 into v'_1 and v_2 into v'_2. We then add errors to the transformed vectors v'_k to form interval vectors V'_k, $k \in \{1, 2\}$. We again use isotropic errors of size ε (Appendix 2) so that $V'_k = \langle v'_k, \varepsilon \rangle_I$. The problem is solved by the network producing a rotation represented as an interval quaternion Q, the volumetric error of which can be calculated as:

$$g(Q) = \prod_{i=0}^{i=3} d(Q_i) \qquad (5.4)$$

where $d(Q_i) = sup(Q_i) - inf(Q_i)$, $i \in \{0, 1, 2, 3\}$, is the width of the ith component of the quaternion. Results for a range of values for ε are shown in Figure 9 for the following choice of $v_k \to v'_k$, $k \in \{1, 2\}$:

$$(-.510, .830, .226) \to (-.405, .913, .045)$$

$$(.680, -.230, .696) \to (-.524, -.675, .519)$$

The slope of the line drawn through the points is exactly 4 suggesting, as g is the product of the widths of four intervals, that the width of each component of Q is linearly related to the widths of the input vectors (which are proportional to ε). The line in Figure 9 is expressed by the equation:

$$g = (1.1\varepsilon)^4$$

Using random errors (Appendix 3) instead of isotropic errors ($V'_k = \langle v'_k, \varepsilon \rangle_R$) makes little difference except the points in Figure 9 are more scattered about.

Figure 9 refers to a network that contains only one module and is therefore relatively simple. Figure 10 shows similar results for the more complicated network (shown in Figure 11) that calculates the translation and rotation between three pairs of interval location vectors: $\mathbf{L}_k$ and $\mathbf{L'}_k$, $k \in \{1, 2, 3\}$. Note the use of "*P2V*" modules, each of which converts a pair of locations into a direction and a length. The following

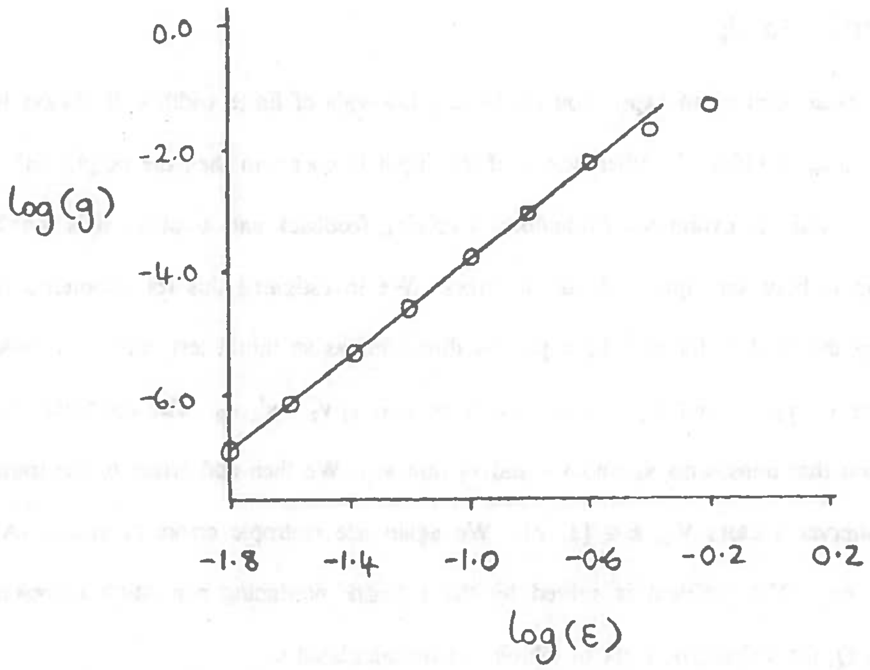


Figure 9: Error function g for two pairs of directions.

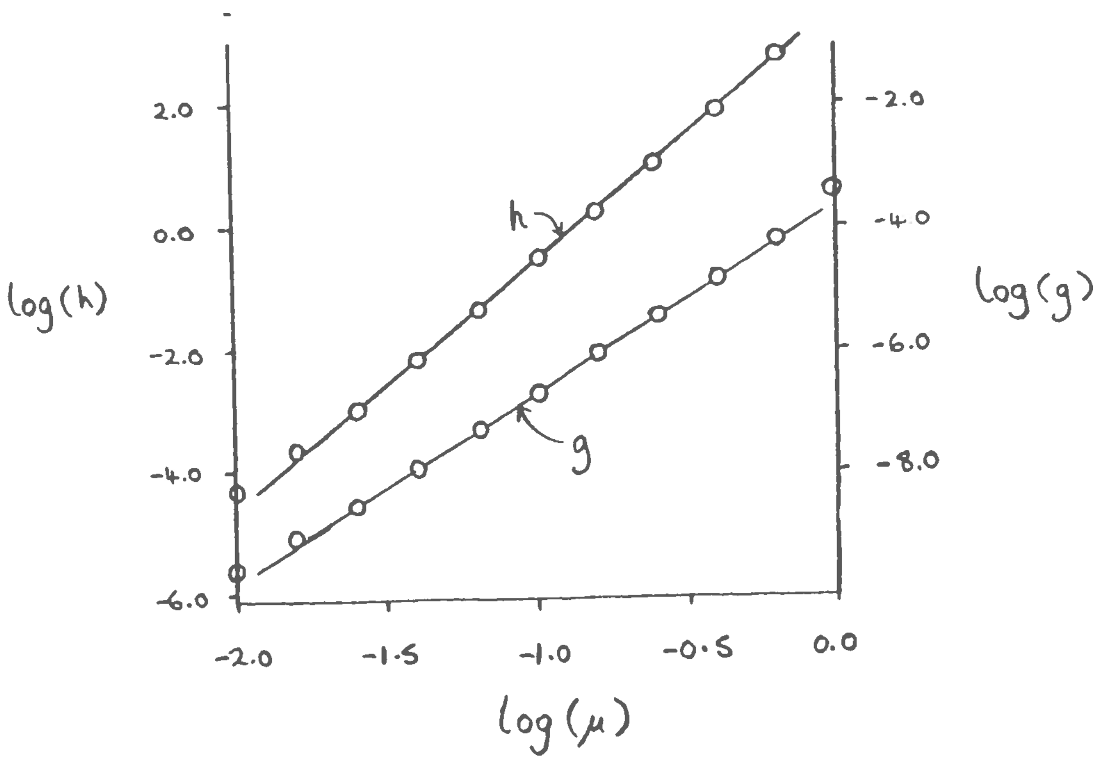


Figure 10: Errors functions g and h for three pairs of locations.

Fig. 11 - see next page.

Figure 11: Network for three pairs of locations.

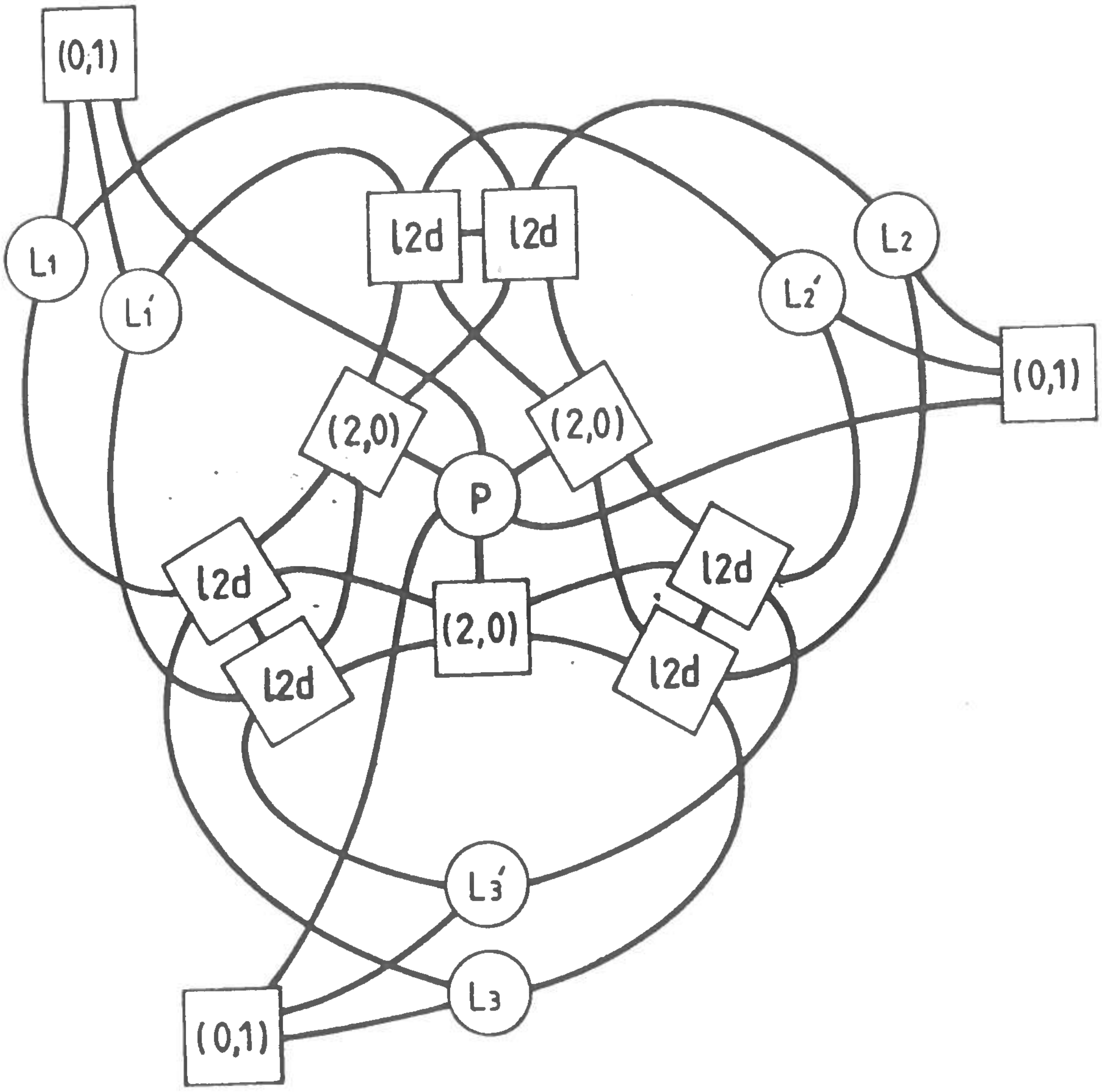
pairs were chosen by (a) randomly choosing the input vectors, a rotation axis, a rotation angle and a translation vector and (b) deducing the output vectors by applying the rotation and translation to the input vectors (using matrix algebra).

$$(7.27, 1.32, -6.52) \rightarrow (26.73, -0.18, -8.73)$$

$$(7.53, -6.59, 9.32) \rightarrow (26.28, 4.70, 8.29)$$

$$(-3.30, 1.86, -0.84) \rightarrow (16.97, 6.25, -5.95)$$

fig. 11



Isotropic errors of size μ were then added to each of the output vectors and the pairs of vectors used as input to the network of Figure 11 to solve for the rotation and translation. Here μ is the radius of the error spheres surrounding the point locations and is analogous to ε for direction vectors (see Appendix 2). The functions g (equation (5.4)) and h in Figure 10 show the variation of rotation and translation error with μ. The latter function is defined:

$$h(\mathrm{T}) = \prod_{j=1}^{j=3} d(T_j) \tag{5.5}$$

where T_j, $i \in \{1, 2, 3\}$, is the jth component of translation. For this network the behaviour is the same as the simpler example above, namely a linear relation between the uncertainties of the input and output variables. The two lines in Figure 10 are:

$$g = (0.2\mu)^4$$

$$h = (7\mu)^3$$

Notice the error magnification for the translation is large. This is a consequence of the way translation is calculated after rotation depending not only on the input vector uncertainties but also on the derived rotation errors and the length of the vectors being rotated.

By examining the algebraic relationships involved in the composition of reference frame module (TT), it can be shown analytically that if each orientation component of a random position T_{a-b} has error δ and each orientation component of a random position T_{b-c} has error ε, then each component of the resulting position $T_{a-c} = T_{a-b} * T_{b-c}$ has (statistical) error:

$$\frac{16\pi}{3}(\delta + \varepsilon)$$

assuming both δ and ε are small. This result has been verified by simulation. Figure 12 shows the empirical results, which taper off as the maximum error volume is reached. In essence, the results mean that the output error width is about 1.7 times the input error width.

A similar analysis of the transformation of a location (TP module) shows that each translation component has output error:

$$\frac{16L}{\pi}\varepsilon$$

where ε is the input rotation quaternion error on each component, and L is the distance of the input point see next page

Figure 12: TT output quaternion error volume versus input error

from the origin. This is sensible, because small rotation errors amplify along a long baseline, to produce larger translation errors. This result has also been verified empirically.

5.3 Non-Optimality

The width of an evaluated interval expression is sometimes larger than it need be. To be more precise, let f be a real function of a vector of real variables $\mathbf{x} = (x_1, x_2, ..., x_n)$ and let F be the corresponding interval evaluation over the interval vector $\mathbf{X} = (X_1, X_2, ..., X_n)$. The strongest statement that can be made about the range of values of $f(\mathbf{x})$ for all possible choices of $\mathbf{x} \in \mathbf{X}$ and the evaluation $F(\mathbf{X})$ is the following (Alefeld and Herzberger, 1983):

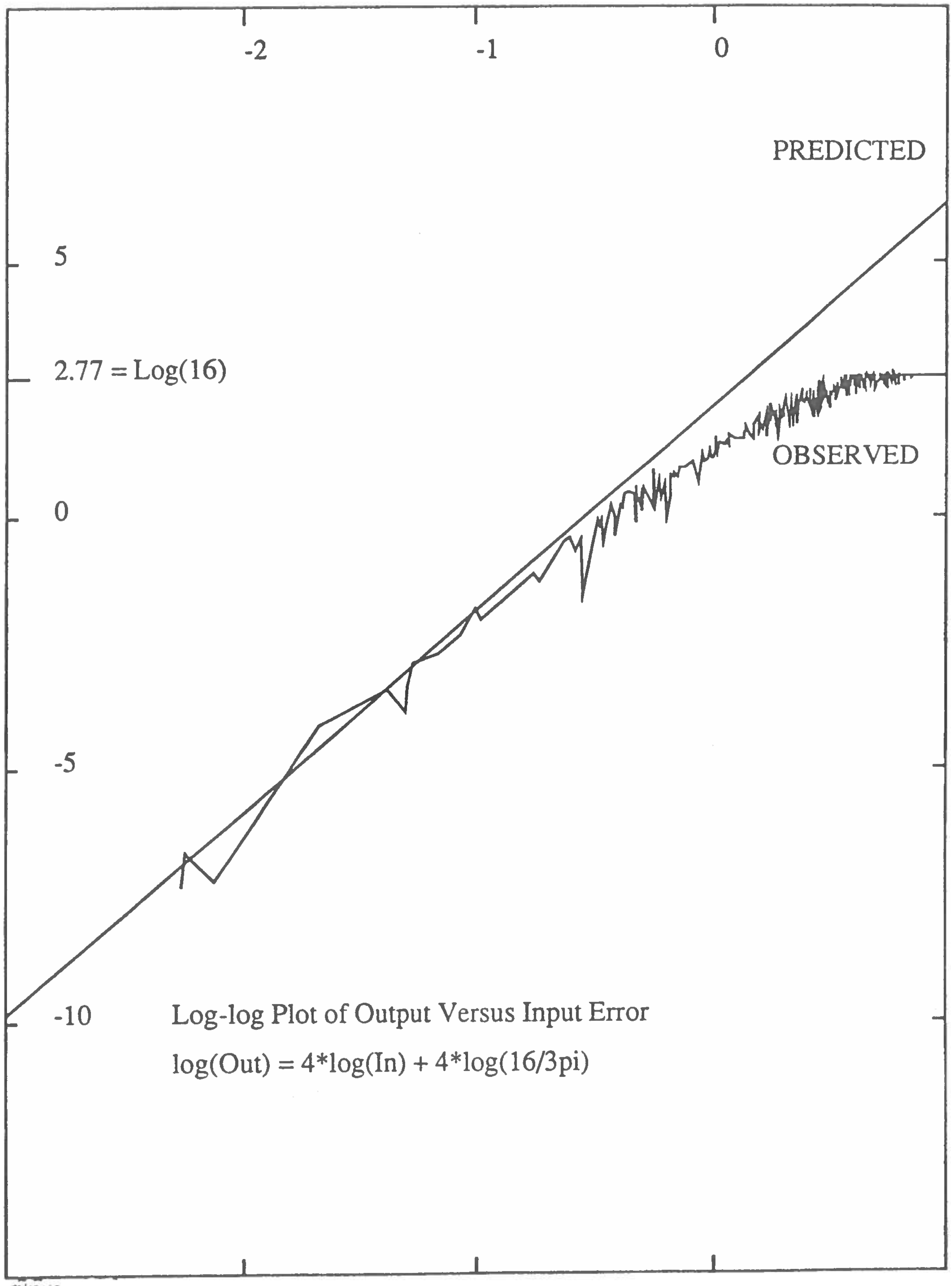
$$\{y = f(\mathbf{x}) \mid x_i \in X_i, i \in \{1, 2, ..., n\}\} \subseteq F(\mathbf{X})$$

The reason for this is the appearance in the expression of variables that depend on one another, either because the same variable occurs more than once or because variables are related through other expressions. A simple example is the following (where $n = 1$):

$$f(x) = \frac{x}{1 + x}$$

$$X = [1, 2]$$

Figure 12



-2
-1
0
PREDICTED
5
2.77 = Log(16)
OBSERVED
0
-5
-10
Log-log Plot of Output Versus Input Error
log(Out) = 4*log(In) + 4*log(16/3pi)

$$F(X) = \frac{[1, 2]}{[1, 1] + [1, 2]} = \frac{[1, 2]}{[2, 3]} = [0.33, 1]$$

$$\{y = f(x) \mid x \in X\} = [0.5, 0.67] \subseteq F(X)$$

Some expressions can be rearranged to decrease to one the number of occurrences of a particular variable. In the above example we can write:

$$f(x) = \frac{1}{1 + x^{-1}}$$

in which case $\{y = f(x) \mid x \in X\} = F(X)$. But such transformations are not always possible.

The geometric reasoning modules also suffer from this problem as can easily be shown. We take four real vectors v_1, v_2, v'_1 and v'_2 as before where $\mathrm{v}_1 \cdot \mathrm{v}_2 = \mathrm{v}'_1 \cdot \mathrm{v}'_2$ and form four corresponding interval vectors:

$$\mathbf{V}_k = <\mathrm{v}_k, 0>_I$$

$$\mathbf{V}'_k = <\mathrm{v}'_k, \varepsilon>_I$$

$k \in \{1, 2\}$. From these we use the network to deduce an interval quaternion for the rotation that maps the first pair of interval vectors to the second pair. Then we reverse the process and use the same network to estimate new vectors $\mathbf{V}''_k$ given the quaternion estimate and the unrotated vectors $\mathbf{V}_k$. Now, if the quaternion had been optimally bounded then $\mathbf{V}''_k = \mathbf{V}'_k$ would hold and in particular $h(\mathbf{V}'_k)$ and $h(\mathbf{V}''_k)$ would be identical (h is defined in equation (5.5)). In practice, the quaternion is not optimally bounded and the ratio of volumetric errors is typically a few hundred:

$$\frac{h(\mathbf{V}''_k)}{h(\mathbf{V}'_k)} \approx 7^3$$

implying a magnification factor of about 7 for the width of each vector component.

5.4 Termination

On a parallel machine, the speed of the network is roughly proportional to the cycle frequency in our serial simulation. In each cycle all nodes that require re-evaluation are updated once and it is these updates that can be done in parallel. Thus the number of cycles taken for the network to terminate is proportional to the run time on a parallel machine.

In practice we find termination time to be more or less independent of (a) the input errors and (b) the threshold value.

The threshold (a small positive value) determines when termination has taken place. When the changes between cycles are less than the threshold, evaluation halts. We found that termination time increased by only one or two cycles (out of 50-100) as the threshold was lowered from 10^{-3} to 10^{-6}. Davis (1987) suggested that, for the type of networks dealt with here (interval label constraint machines), termination may never occur (without a threshold) which would seem to imply that termination time should increase (to infinity) as the threshold tends to zero.

We find that the time to convergence is mainly determined by the maximum distance between value nodes in the network. This determines the maximum time for changes to propagate over the network and is related to the greatest depth in the algebraic expressions whose compilation formed the network.

6. Conclusions

The methodology we have investigated is summarised here. We start with sets of algebraic constraints associated with particular geometric relationships. Observables are represented by variables at this stage. These constraints are compiled into network modules where the structure of the module reflects the structure of the expressions for the bounds. All these steps are done in an off-line process and need not be repeated unless new constraint types are added. The on-line program solves geometric problems with networks created by connecting pre-compiled modules together according to the structure of each problem. When observable variables are bound to measured values other variables are forced into consistency as the network evaluates.

The networks implement the necessary functions of geometric reasoning for computer vision, as identified in section 2. It has also been successfully used in a vision system to estimate object positions and detect false hypotheses.

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. We have shown that although the SUP/INF algorithm cannot always detect that a given problem is insoluble, this defect may have limited effect in practice.

While the error analysis given in Section 5 showed that errors usually do not grow too quickly, when we look at the robot network (Figure 5), we still find a problem. In particular, it turns out that the chain of reference frame transformations from the robot in the global reference frame (T_{g-R}) to the data vectors involved in the lower arm position (below T_{g-L}) involve ten TV2 and three TT modules. Assuming each has an output interval width of 1.7 times the input interval width means that the ultimate contribution of the lowerarm evidence to the robot has an error interval width of $(1.7)^{13} \approx 1000$ times the input error interval. As the maximum output interval can be only [-1,+1] for the rotation, this implies that the input uncertainity can be at most about 0.001, which is unreasonably strict.

However, this error analysis must also consider that many of the reference frame transformations are the identity transformation, which create little error (and need not even be introduced into the network). Further, though the estimates may not propagate all the way to the robot global position node, they will help constrain more closely linked structures.

It should be noted that geometric reasoning involving a multiple joint articulated object is a genuinely complicated problem, whereas most object recognition problems have a much flatter model hierarchy, involving fewer compound degrees-of-freedom, if any at all.

Acknowledgements

Figures 7, 8 and 12 are reprinted from Fisher and Orr (1989) with permission from International Joint Conferences on Artificial Intelligence, Ltd. Copies of this and other IJCAI proceedings are available from Morgan Kaufmann Publishers, Inc. PO Box 50490, San Mateo California, 94403.

Figures 5 and 6 are reprinted from Fisher (1989) by permission of John Wiley and Sons Ltd.

Figure 1 is reprinted from Fisher and Orr (1988) by permission of Butterworth Scientific Ltd.

References

Alefeld, G. and Herzberger, J. 1983. *Introduction to interval computations.* London: Academic Press.

Ayache, N, and Faugeras, O. D. 1988. Building, Registering and Fusing Noisy Visual Maps, *Int. J. of Robotics Research*, 7(6):45-65.

Ballard, D. H. and Tanaka, H. 1985. Transformational form perception in 3D: constraints, algorithms and implementation. *Proc. 9th Int. Joint Conf. on Artificial Intelligence.* Los Angeles: Morgan Kaufmann, p964-968.

Brooks, R. A. 1981. Symbolic reasoning among 3-D models and 2-D images. *Artificial Intelligence* 17(1): 285-349.

Davis, E. 1987. Constraint propagation with interval labels. *Artificial Intelligence* 32(3): 281-331.

Durrant-Whyte, H. F. 1988a. Uncertain geometry in robotics, *IEEE J. of Robotics and Automation* 4(1):23-31.

Durrant-Whyte, H. F. 1988b. Sensor Models and Multi-Sensor Integration, *Int. J. of Robotics Research*, 7(6):97-113.

Faugeras, O. D. and Hebert, H. 1983. A 3-D recognition and positioning algorithm using geometrical matching between primitive surfaces. *Proc. 8th Int. Joint Conf. on Artificial Intelligence* Karlsruhe: Morgan-Kaufmann, pp996-1002.

Fisher, R. B. 1986. SMS - a suggestive modeling system for object recognition. *Image and Vision Computing* 5(2):98-104.

Fisher, R. B. 1989. From surfaces to objects: computer vision and three dimensional scene analysis. Chichester: John Wiley.

Fisher, R. B. and Orr, M. J. L. 1988. Solving Geometric Constraints in a Parallel Network. *Image and Vision Computing* 6(2).

Fisher, R. B. and Orr, M. J. L. 1989. Experiments with a network-based geometric reasoning engine. *Proc. 11th Int. Joint Conf. on Artificial Intelligence.* Detroit: Morgan Kaufmann, pp 1623-1628.

Fleming, A. D. 1988. Analyses of Uncertainities and Geometric Tolerances in Assemblies of Parts. PhD Dissertation, University of Edinburgh, Department of Artificial Intelligence.

Hinton, G. E. and Lang, K. 1985. Shape recognition and illusory conjunctions. *Proc. 9th Int. Joint Conf. on Artificial Intelligence*. Los Angeles: Morgan Kaufmann, p252-259.

Orr, M. J. L. and Fisher, R. B. 1987. Geometric Reasoning for Computer Vision. *Image and Vision Computing* 5(3):233-238.

Popplestone, R. J., Ambler, A. P. and Bellos, I. M. 1980. An interpreter for a language describing assemblies. *Artificial Intelligence* 14(1):79-107.

Porrill, J. 1988. Optimal Combination and Constraints for Geometrical Sensor Data. *Int. J. of Robotics Research*, 7(6):66-77.

Appendix 1. The TP module

Here, for illustration, we write out the mathematics underlying one geometric reasoning module and briefly describe the modifications necessary to enable its compilation into a module. The module transforms a single location vector by a position. We use bold letters for both quaternions and vectors leaving the context to distinguish which is meant. A pure vector $\mathbf{x}=(x_1,x_2,x_3)$ is equivalent to the quaternion $\mathbf{x}=(0,x_1,x_2,x_3)$, while the quaternion $\mathbf{q}=(q_0,q_1,q_2,q_3)$ has scalar part q_0 and vector part $\omega=(q_1,q_2,q_3)$. The symbols "*" and "′" stand for the quaternion operations of multiplication and conjugation, where $\mathbf{r} * \mathbf{s}=(r_0 s_0 - \omega_r \cdot \omega_s,\ \omega_r \times \omega_s + r_0\omega_s + s_0\omega_r$, $((q_0, \omega)' = (q_0, -\omega))$, and "·" and "×" are, respectively, the inner and cross vector products.

A position consists of a rotation $\mathbf{q}$ and translation $\mathbf{t}$ where $\mathbf{q} * \mathbf{q}' = \mathbf{q}' * \mathbf{q} = 1$ and $t_0 = 0$. Let $\mathbf{u}$ be a location vector and $\mathbf{v}$ be the transformation of $\mathbf{u}$ by rotation $\mathbf{q}$ and translation $\mathbf{t}$. Then:

$$\mathbf{v} = \mathbf{q} * \mathbf{u} * \mathbf{q}' + \mathbf{t} \tag{A1.1}$$

It follows from equation A1.1 that:

$$\mathbf{t} = \mathbf{v} - \mathbf{q} * \mathbf{u} * \mathbf{q}'$$

$$(A1.2)$$

$$\mathbf{u} = \mathbf{q}' * (\mathbf{v} - \mathbf{t}) * \mathbf{q} \qquad (A1.3)$$

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

From this last equation we deduce the vector equations:

$$\boldsymbol{\omega}\cdot\mathbf{u} = \boldsymbol{\omega}\cdot(\mathbf{v} - \mathbf{t}) \qquad (A1.4)$$

$$q_0(\mathbf{v} - \mathbf{t} - \mathbf{u}) = \boldsymbol{\omega} \times (\mathbf{v} - \mathbf{t} + \mathbf{u}) \qquad (A1.5)$$

Equations A1.1-5 constitute the underlying mathematics for the module. To prepare this as input to the network compiler the following must be done:

write each vector (quaternion) equation as three (four) separate scalar equations and

rewrite each equation with an expression on the left as a series of equations with only a single variable or a product of two variables on the left.

For example, equation A1.5 is a vector equation, the first component of which is:

$$q_0(v_1-t_1-u_1) = q_2(u_3+v_3-t_3) - q_3(u_2+v_2-t_2)$$

If we are interested in bounds on q_0, and noting that the sign of $(v_1-t_1-u_1)$ is not known ***a priori***, we use the ***srecip*** function and write:

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

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

Appendix: 2. Isotopic errors

Let $\mathbf{v}$ be a real direction vector of unit norm with components v_k and $\mathbf{V}$ be a interval direction vector with interval components $V_k = [inf(V_k),\ sup(V_k)]$. If isotropic error of size $\varepsilon \geq 0$ is added to $\mathbf{v}$ to obtain $\mathbf{V}$, then:

$$\mathbf{V} \equiv \langle \mathbf{v}, \varepsilon \rangle_I$$

$$inf(V_k) = \cos(\min(\pi, \phi_k + \varepsilon))$$

$$sup(V_k) = \cos(\max(0, \phi_k - \varepsilon))$$

where $\phi_k = \cos^{-1}(v_k) \in [0, \pi]$.

Similarly, if **p** and **P** are real and interval location vectors respectively and if isotropic error of size $\mu \geq 0$ is added to **p** to obtain **P** then:

$$\mathbf{P} \equiv \langle \mathbf{p}, \mu \rangle_I$$

$$inf(P_k) = p_k - \mu$$

$$sup(P_k) = p_k + \mu$$

Appendix: 3. Random errors

Let **v** be a real direction vector of unit norm with components v_k and **V** be a interval direction vector with interval components $V_k = [inf(V_k), sup(V_k)]$. Let ε_{kj}, $k \in \{1, 2, 3\}$, $j \in \{1, 2\}$, be randomly generated numbers in the range $[0, \varepsilon]$. If random error of size ε is added to **v** to obtain **V**, then:

$$\mathbf{V} \equiv \langle \mathbf{v}, \varepsilon \rangle_R$$

$$inf(V_k) = \max((v_k - \varepsilon_{k1}), -1)$$

$$sup(V_k) = \min((v_k + \varepsilon_{k2}), 1)$$

Similarly, if **p** and **P** are real and interval location vectors respectively, if μ_{kj}, $k \in \{1, 2, 3\}$, $j \in \{1, 2\}$, are random numbers in the range $[0, \mu]$ and if random error of size μ is added to **p** to obtain **P** then:

$$\mathbf{P} \equiv \langle \mathbf{p}, \mu \rangle_R$$

$$inf(P_k) = p_k - \mu_{k1}$$

$$sup(P_k) = p_k + \mu_{k2}$$