Computers & Structures Vol. 46, No. 1, pp. 99–124, 1993
Printed in Great Britain.

0045-7949/93 $6.00 + 0.00
© 1992 Pergamon Press Ltd

GEOMETRICAL AND TOPOLOGICAL CONSISTENCY IN INTERACTIVE GRAPHICAL PREPROCESSORS OF THREE-DIMENSIONAL FRAMED STRUCTURES

M. GATTASS, G. H. PAULINO and J. C. GORTAIRE C.
Civil Engineering Department, PUC-Rio, 22453 Rio de Janeiro, Brazil

(*Received* 1 *October* 1991)

Abstract—This paper shows how interactive graphical preprocessors can treat inconsistent data to generate consistent finite element and structural models. Algorithms are described for checking the geometrical and topological consistency of three-dimensional meshes made up of one-dimensional finite elements. With the consistency guaranteed, the user of a graphical preprocessor can freely create the geometry and topology of a structure with his or her attention concentrated only on the image shown on the screen. Inconsistencies like repeated nodes or partially overlapped bars are left to the system to solve.

1. INTRODUCTION

A finite element model must be consistent for the computation to succeed. If there are errors in defining the mesh, then the computation may advance until aborting or, even worse, it may produce results which are wrong [1]. Graphical preprocessors have come a long way to minimize problems in the finite element data generation [2, 3]. With the use of these systems, the process of defining data has undoubtedly become more creative and reliable. There are, however, a few issues which remain to be solved. One of these issues is related to the consistency of the geometrical/topological model generated by the preprocessor with respect to finite element modeling rules.

Geometric functions of interactive graphical preprocessors, such as symmetry or copy of a plane of nodes and elements to a given position, may create repeated nodes or overlapped elements. These situations, that are sometimes hard to detect in an image of the structure, represent unacceptable errors in the finite element model.

The existence of more than one node in the same position, for example, would bring many problems to the man–machine graphic conversation. The user would not know with certainty to which node he or she has pointed and may have a mistaken impression that there is only one node at that point. The model would treat each node with different degrees-of-freedom, probably yielding an incorrect solution with unwanted gaps.

There is also the possibility of bar elements crossing at internal points. This situation does not characterize clearly the existence of an error. Bar crossings can be desirable in some structural types such as trusses and bracking bars in frames, as shown in Fig. 1.

On the other hand, if the preprocessor can solve for bar element crossings at internal points, the creation of the geometry/topology of a structure can be rather simple. Consider, for example, the floor of one building shown in Fig. 2. If the user is requied to create a node at each bar intersection, the input data has 17 nodes and 25 elements as shown in Fig. 2(b). This process is much less comfortable to the user than the one shown in Fig. 2(a) that involves the input of only 14 nodes and nine elements if the preprocessor can create nodes occurring at intersections.

The simple examples shown in Figs 1 and 2 illustrate the versatility required from a frame preprocessor. It must be able to cope with both cases. These types of intersections are common situations in structures such as roofs, floors and transmission towers.

There are basically two approaches for the data consistency problems in finite element preprocessors. The solution of these problems can be made during the generation of the geometry and topology of the model, or in a later time after the model has been created (*a posteriori* approach). In the former case, consistency rules are imposed as the data are created, never allowing for inconsistency to occur.

If the system restricts the actions of the user to create consistent data only, the man–machine dialogue can become complicated and cumbersome. On the other hand, if the preprocessor verifies the consistency of the data for all user's actions, the response time of the system may compromise the effectiveness of the dialogue between the user and the computer.

In the second approach mentioned above the user creates the geometry and topology of the structure without worrying about the consistency of the model. Inconsistencies arising at this stage are kept in the data base until the user explicitly asks for a check of the consistency problem or until data is to be transported to the analysis program. The ambiguous situations, such as bar crossings at internal points, are solved by

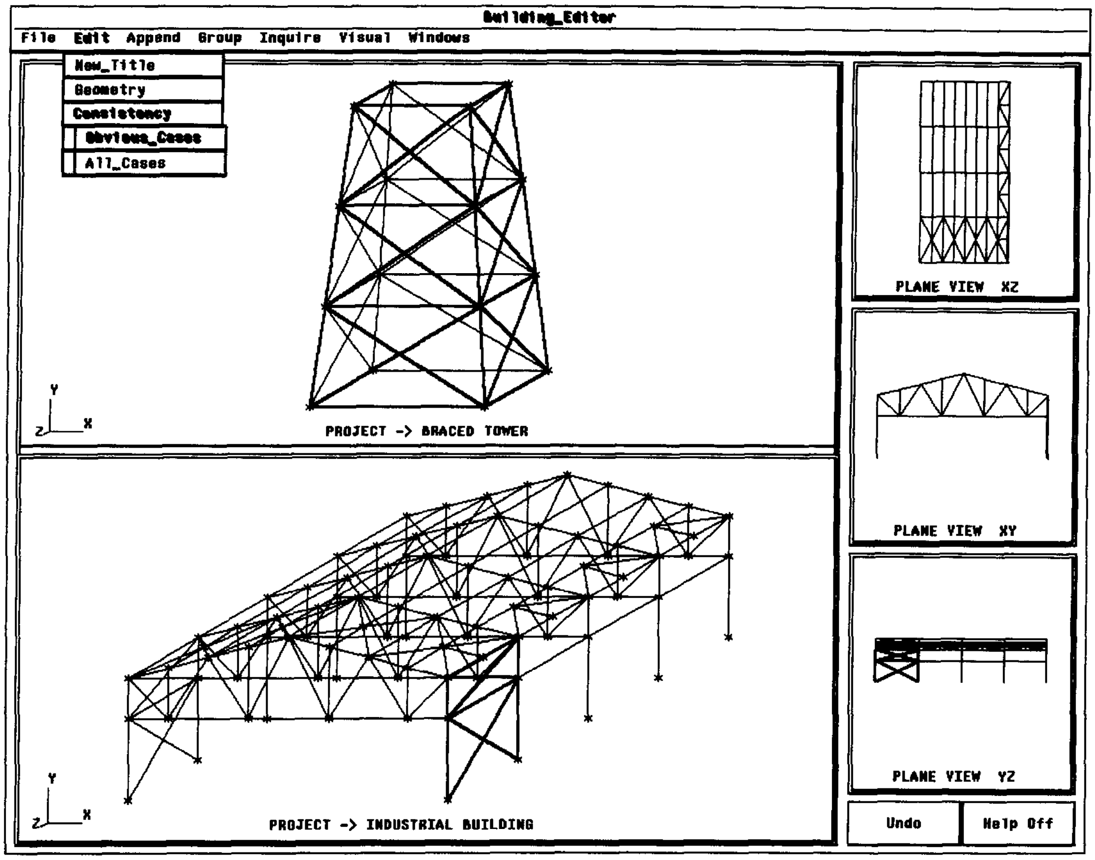

Fig. 1. Element crossings at internal points.

the user at this time. He or she may ask the system to treat only obvious inconsistencies or to decompose bars that cross.

The latter approach can also benefit many existing preprocessors that do not treat the consistency problem. A program with the algorithms for this case can be used as a filter to remove inconsistencies of data previously created. This program must be capable of transforming any wire-frame representation (inconsistent) in a consistent finite element model of frames.

This paper presents the algorithms for the *a posteriori* approach. The algorithms proposed can be part of an independent program or a specific module for a generic preprocessor. The algorithms are presented following a step-wise refinement technique using pseudo-code.

2. THE CONSISTENCY PROBLEM

There are basically two algorithms to solve the consistency problem. The first one only treats inconsistencies that are unambiguous: repeated and isolated nodes, repeated and partially overlapped bars, and nodes within bars. The second algorithm also treats the case of bars that cross at internal points.

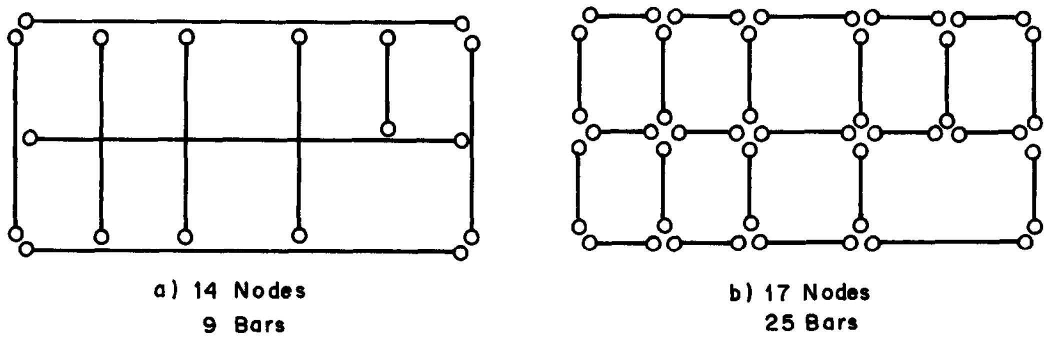

Fig. 2. Input data for beams in a building floor.

To eliminate unambiguous inconsistencies the algorithm must first eliminate repeated nodes and bars of null size. This step reduces the size of the data structure without changing the basic geometrical definition of each bar. At this stage it is also beneficial to eliminate nodes that do not belong to any bar.

Nodes that do not belong to any bar can either be an error or they can be used to define the orientation of the bars. Even if these nodes are used as reference to define the direction of the minor axis of bars, they should be removed from the list of active geometric nodes. They can be stored as special nodes to which no degree-of-freedom is attached.

With the number of nodes reduced to its minimum, the next step in the elimination of unambiguous inconsistencies is the division of bars to eliminate internal nodes. This division eliminates the problem of partial overlapping among bars. After this step two bars can either overlap completely or they do not overlap at all. Partial overlapping cannot occur since there are no internal nodes.

The final step in this process is the elimination of repeated bars. Here, once again, the elimination of repeated nodes plays an important role. Overlapping bars can be identified easily only by their topology (nodal incidence) without any inquiry about their geometry (end coordinates). Hence, the first algorithm can be decomposed as a sequence of four steps [2]:

Algorithm 1. Elimination of basic inconsistencies
Step 1: Elimination of repeated nodes and bars of null size.
Step 2: Elimination of nodes without incidence.
Step 3: Division of bars to avoid internal nodes.
Step 4: Elimination of repeated bars.

In the above algorithm, steps 2 and 3 can be interchanged yielding a different but consistent mesh. Nevertheless, if the isolated nodes are eliminated before the division of bars to avoid internal nodes, as shown, the user cannot use a *CreateNode* function to divide bars. Instead the preprocessor must provide a specific *DivideBar* function to perform this task. If the steps are interchanged the bar is divided and the isolated nodes become end nodes of the newly created bars. The proposed order, however, is preferred by the authors.

To treat the problems of bars that cross at internal points, another algorithm implemented as a sequence of seven steps is proposed here. The first four steps are the same as the first algorithm. To divide the bars that cross, three more steps are needed. The first additional step creates a node at each intersection of the bars without dividing them. If more than two bars cross at the same point this step can create repeated nodes. Thus the next step is the elimination of such nodes. The last step divides the bars to avoid internal nodes. Steps 6 and 7 need only be executed when a bar crossing is detected in step 5. Note that the elimination of repeated bars, bars of null size and isolated nodes are not necessary since they cannot occur at this stage. Hence, the proposed sequence decomposition for this algorithm is [2]:

Algorithm 2. Elimination of bar crossings
Step 1: Elimination of repeated nodes and bars of null size.
Step 2: Elimination of nodes without incidence.
Step 3: Division of bars to avoid internal nodes.
Step 4: Elimination of repeated bars.
Step 5: Creation of nodes at bars crossings.
Step 6: Elimination of repeated nodes.
Step 7: Division of bars to avoid internal nodes.

For both algorithms there are only five independent steps: the four presented in the first algorithm and the step that creates nodes at bar intersections. With the exception of the elimination of nodes without incidence (that was already commented on) all steps must be kept in the order shown above to avoid unwanted results.

3. METHODS OF SOLUTION

In Sec. 2, two algorithms using five independent steps were presented to solve the consistency problem. Each of these steps will be implemented using three different methods of solution. All the methods lead to a consistent finite element representation of the data structure, but differ in computational efficiency.

In the first method of solution (simple), a given object (node or bar) may be tested against all others. This is a typical situation where the algorithms have, roughly speaking, a complexity of n^2 (n being the number of nodes or bars).

To increase efficiency of the algorithms, this paper presents two alternative methods: the use of filters (filter) and the use of sorted data (order). The former method uses simple tests to avoid more expensive ones.

Table 1. Methods of solution

Implementation	Method
First	Simple
Second	Filter
Third	Order

A typical example of this strategy is the use of bounding boxes; that is, a box that contains a bar as the main diagonal. Two bars cannot intersect if their bounding boxes are disjoint.

The latter method seeks to sort the data to reduce the amount of objects (nodes or bars) that a given object must be tested against. A bar, for example, can only intersect nodes that have y coordinates greater than the bar's minimum y and smaller than its maximum y. Thus, if the nodes are sorted by their y coordinates, the test to detect if a node is internal to a given bar is restricted to a certain portion of the nodal array (or list). This strategy can also make use of filters in the reduced set of objects.

Each method of solution will be associated with a distinct computational implementation, and will be referred to according to Table 1.

4. NOTATION AND BASIC DATA STRUCTURE

This paper is intended primarily for programmers of modern computer languages such as C or Pascal, but recognizes the importance of existing FORTRAN codes. For this reason, the ideas are presented in the form of pseudo codes that are language independent.

In search for notation, the authors found that while the most common algorithmic language, pidgin algol [4], is directly useful for Pascal and C programs, the binding between this language and FORTRAN codes is not direct. For this reason, an adaptation of this algorithmic language was adopted here.

With modern languages like C and Pascal, there are a variety of ways to store nodal coordinates and element incidence. However, in order to keep compatibility with existing FORTRAN finite element codes, the algorithms presented use the conventional vector arrays. Therefore, the following variables are considered global for all the algorithms shown in this paper

```
var
    nnodes,nbars   :int;                          {number of nodes and number of bars}
    NodeI,NodeJ    :array[1...MAX_BARS] of int;           {bar incidence}
    x, y, z        :array[1...MAX_NODES] of real;   {nodal coordinates}
```

In the above declarations, MAX_BARS and MAX_NODES are global integer parameters to denote an upper bound for the number of nodes and bars, respectively. The other parameter also presented in the code is TOL, to denote tolerance. The value of this tolerance depends on the structural dimensions and it may be computed by the program according to the maximum and minimum coordinates or defined interactively by the user. For the algorithms presented herein, it was considered appropriate to use TOL $= 10^{-4}$.

In most finite element codes, node and bar attributes, such as boundary conditions and materials, are also stored in arrays indexed by the node/bar number. To solve geometrical and topological inconsistencies, the algorithms presented here delete, create and redefine nodes and bars. This procedure obviously changes the node and bar numbers thus requiring a correction to be made in the attribute arrays. However, this correction is not always simple. For instance, consider two partially overlapped bars of different materials. To solve the inconsistency problem these two bars must be transformed into three non-overlapped bars, with the overlapped region being one of them. Which material should be assigned to this new bar is not clear. Any one of the materials of the present bars is equally valid and a user action is required.

Since the *a posteriori* approach is used, and to avoid attribute conflicts and to simplify the algorithms presented, this paper assumes that the geometrical and topological inconsistencies are solved prior to the attribute setting stage. That is, in the preprocessor developed in the present work, the user can create the geometrical and topological model, enforce consistency and, only then, define the attributes and loads.

Therefore, the solution proposed for the consistency analysis algorithms is based on a temporary dissociation of the data structure used to represent simultaneously geometry/topology and the finite element model. This free dissociation is crucial to guarantee efficiency and creativity of the user when editing the structure. If the algorithms presented are used in a program that filters inconsistencies from a complete finite element model, more code should be added to handle the attribute arrays.

5. THE ALGORITHMS

For each one of the five independent steps discussed in Sec. 2, three possible implementations are proposed. The first implementation maintains the order of nodes and the bars orientation. To improve efficiency, the next implementation uses the filter strategy discussed above. The last implementation seeks to improve the performance of the algorithms by also sorting the data. The nodes are sorted according to their geometry (coordinates) and the bars according to their topology (incidences).

The step for the elimination of bar crossings represents the most complex one in the consistency analysis, and for this reason it is presented first. The discussion of these algorithms allows a full perspective of the geometrical and topological consistency problem. Steps 3, 1, 2 and 4 follow in this order.

5.1. *Elimination of bar crossings*

Bars can cross in three different kinds of patterns as illustrated by Fig. 3. Figure 3(a) shows the simplest case where only two bars cross at a point. Figure 3(b) shows the case where several bars cross at the same point. Figure 3(c) illustrates a complex crossing case where several bars cross at many different points. The last case is the general case and must be treated by a good frame preprocessor.

Angell and Griffith present a simplified algorithm to determine line segment intersections [5]. Preparata and Shamos present a very elaborate study about geometric intersection problems in a general sense (hidden-line and hidden-surface problems, intersections of polygons and line segments, etc.) [4].

The simplest algorithm to solve the complex crossing problem would test each bar (called here the reference bar) against all the others for intersections. If an intersection point is found, the algorithm should divide both the bars. A tolerance should be established to avoid the division of a bar too close to its end nodes.

The simple algorithm referred above has a basic problem: if a reference bar is divided after a certain number of tests, the newly created bars should not be submitted to these tests again. The natural way to program this solution is to use recursive procedures and data structures that are more flexible than the array list presented in Sec. 3.

To solve the intersection problem with the static data structure an adaptation must be performed. Instead of dividing the bars as intersections are detected, the proposed algorithm creates a node at each intersection and, in a latter stage, eliminates repeated nodes and divides the bars to avoid internal nodes. Pseudo code 1 shows a possible implementation of this algorithm. The logical function *LineSegX* returns *true* if the two testing bars intersect, and the intersection point is given by the parameter *t*. Appendix A presents both the method used to solve the intersection of line segments and the Pseudo code A1 for the function *LineSegX*.

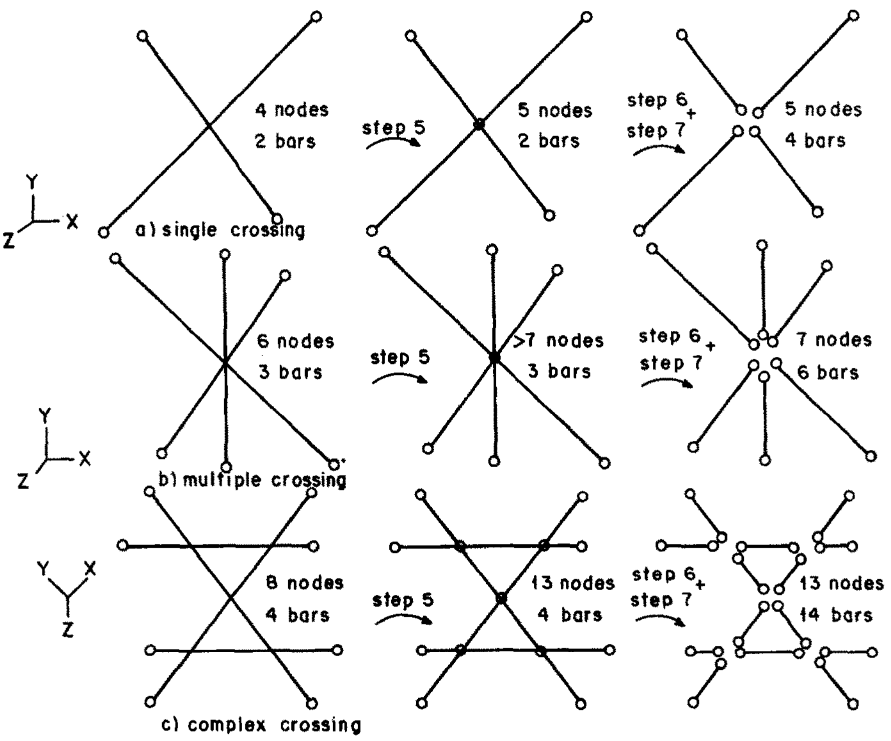

Fig. 3. Types of bar crossings.

```
proc CreateNodeAtBarX;            { first implementation—simple }
{ detect bar intersection and create nodes at this position }
var
  bar1,bar2          : int;
  t                  : real;
begin
  for bar1 from 1 incr 1 to nbars − 1 do begin
    for bar2 from bar1 + 1 incr 1 to nbars do
      if LineSegX(bar1,bar2,t) execute AddInternalNode(bar1,t);
    end for;
  end for;
end.{ CreateNodeAtBarX }
```

Pseudo code 1: Bar crossings (simple).

The procedure *AddInternalNode* simply adds a node in the node array in the parametric position *t* of the reference bar as illustrated by Pseudo code 2.

```
proc AddInternalNode ( bar: int; t: real ) ;
{ create an internal node to the bar at parametric coordinate t }
begin
  nnodes ← nnodes + 1;
  x[nnodes] ← x[NodeI[bar]] + ( x[NodeJ[bar]] − x[NodeI[bar]] )*t;
  y[nnodes] ← y[NodeI[bar]] + ( y[NodeJ[bar]] − y[NodeI[bar]] )*t;
  z[nnodes] ← z [NodeI[bar]] + ( z[NodeJ[bar]] − z[NodeI[bar]] )*t;
end. { AddInternalNode }
```

Pseudo code 2: Add a node in a bar parametric position.

The importance of developing efficient algorithms for detecting intersection is apparent as the size of structures being analyzed grows increasingly more ambitious. A typical structure may contain hundreds or thousands of bars that must be processed as quickly as possible in order to provide an answer to the user within a time that does not impair the interactive dialogue. Hence, the nearly quadratic time algorithm presented in Pseudo code 1 yields unacceptable response time.

The first idea to optimize this algorithm is to introduce a test to avoid the costly computation of line intersections. For each bar, taken as the reference, the algorithm computes a bounding box that contains this bar as the main diagonal as shown in Fig. 4. This bounding box is called here the 'bar box'. To test if other bars intersect the reference bar, the algorithm first checks to see if the bar to be tested intersects the bar box. The Pseudo code 3 shows the algorithm to compute this box. The tolerance is added here to treat the case of bars that are parallel to the Cartesian axes (x, y or z). For these cases the bar box would degenerate and floating point approximations could yield incorrect results.

```
proc ComputeBarBox ( bar:int; box: array [1 .. 6] of real )
{ computes a box with the bar as main diagonal }
```

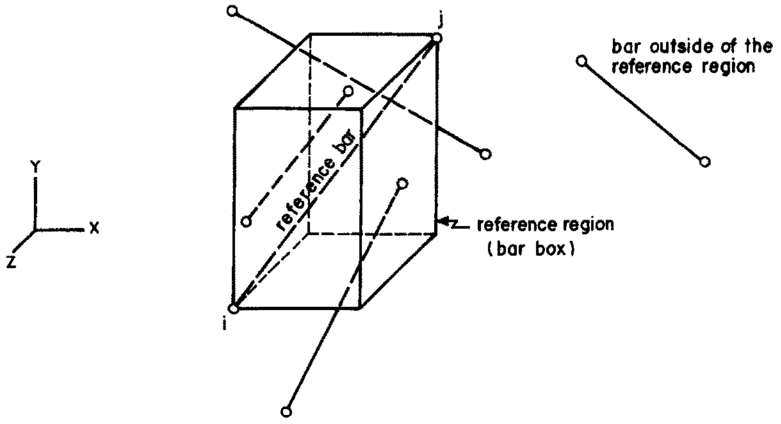
Y
X
Z
j
i
reference bar
bar outside of the reference region
reference region (bar box)

Fig. 4. Definition of a reference region (after Paulino [2]).

```
begin
box[1] ← min (x(NodeI[bar]),x(NodeJ[bar])) − TOL;
box[2] ← max (x(NodeI[bar]),x(NodeJ[bar])) + TOL;
box[3] ← min (y(NodeI[bar]),y(NodeJ[bar])) − TOL;
box[4] ← max (y(NodeI[bar]),y(NodeJ[bar])) + TOL;
box[5] ← min (z(NodeI[bar]),z(NodeJ[bar])) − TOL;
box[6] ← max (z(NodeI[bar]),z(NodeJ[bar])) + TOL;
end. { ComputeBarBox }
```

Pseudo code 3: Bounding box for bar elements.

The selection of bars that intersect a given bar box is similar to the clipping technique used in computer graphics practice. There, clipping algorithms are referred to as procedures for eliminating all parts of a defined picture outside of specified boundaries [6]. Here, the interest is to identify which bars intercept a specified region.

The methodology presented below is based on the line-clipping algorithm proposed by Cohen and Sutherland. This algorithm can be found in many references about Computer Graphics, for example [7]. In this algorithm many bars that lie outside the reference region are easily identified by the position of their end nodes. Floating point operations are required only for the remaining bars. This algorithm is especially efficient in the case of framed structures where the reference region is relatively small and most bars lie left, right, below, above, back or front of the reference region.

To detect which bars lie outside the reference region, the algorithm starts by assigning to each end point of the bar being tested a six-digit binary code, called the 'region code'. Each bit in the region code is set to *1* (TRUE) if a given relation between the end point and the reference region is true, otherwise the bit is *0* (FALSE). Pseudo code 4 illustrates this procedure.

```
procOutCode ( x, y, z: real; box : array [1 .. 6] of real;
                      var outcode : array [1 .. 6] of logical );
{ define the position of a point (x, y, z) with respect to a box }
begin
  outcode[1] ← x < box[1];   {left}
  outcode[2] ← x > box[2];   {right}
  outcode[3] ← y < box[3];   {below}
  outcode[4] ← y > box[4];   {above}
  outcode[5] ← z < box[5];   {back}
  outcode[6] ← z > box[6];   {front}
end. { OutCode }
```

Pseudo code 4: Region code definition.

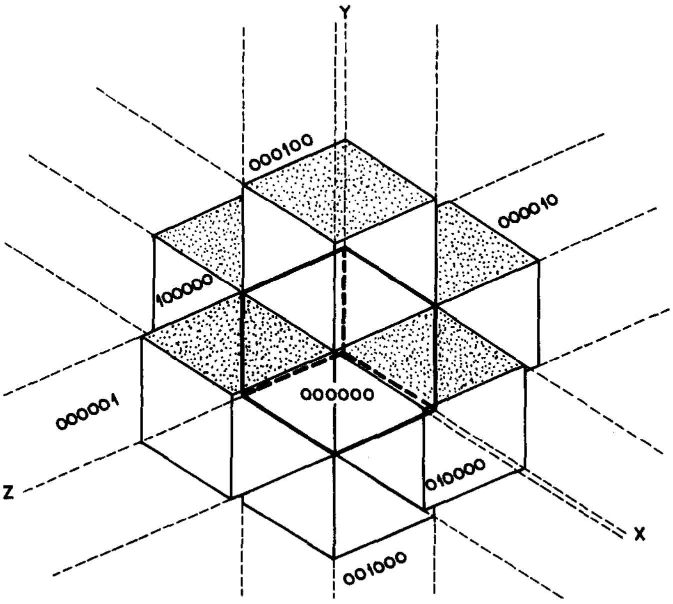

Fig. 5. Binary region codes for bar end points (after Paulino [2]).

A value of *1* in any bit position indicates that the point is in that relative position; otherwise, the bit is set to *0*. For example, a point that is to the right and below the region of reference has a region code of '011000'. If a point is within the region of reference, the region code is '000000'. Figure 5 illustrates the twenty seven possibilities for the region code. The procedure *IsNodeIn*, shown in Pseudo code 5, returns TRUE if the node is inside the bounding box.

```
function IsNodeIn ( outcode: array [1..6] of logical ) :logical;
begin
  IsNodeIn ← not outcode[1] and not outcode[2] and
             not outcode[3] and not outcode[4] and
             not outcode[5] and not outcode[6];
end. { IsNodeIn }
```

Psuedo code 5: Test if a node is inside a box.

Bars that are completely contained within the boundaries of the region of reference have a region code of '000000' for both end nodes. For these bars the line intersection test is unavoidable. On the other hand, bars that have a *1* in the same bit position in the region codes for each end point are completely outside the reference region. A *1* in the first position of both nodes means that the bar is left of the reference region, a *1* in the fourth position means above, and so on. These bars can be trivially discarded from the algorithm as they cannot intersect the bar that lies within the reference region. The algorithm, for example, discards bars that have a region code of '010101' for one end node and a code of '011000' for the other end node. Both end nodes of this line are right of the region of reference, as indicated by the *1* in the second bit position of each region code.

These tests can be efficiently implemented with the *and* bitwise operation for both end node region codes as illustrated in the Pseudo code 6. If the result is not null, the bar is completely outside the reference region. If the result is '000000', the test is not conclusive. The bar may or may not be outside the box. Figure 6 illustrates this situation.

```
function IsLineOut ( outcode1, outcode2: array [1..6] of logical ) :logical;
{ find if a line is left, right, below, above, back or front of a box }
begin
  IsLineOut ← ( outcode1[1] and outcode2[1] ) or
              ( outcode1[2] and outcode2[2] ) or
              ( outcode1[3] and outcode2[3] ) or
              ( outcode1[4] and outcode2[4] ) or
              ( outcode1[5] and outcode2[5] ) or
              ( outcode1[6] and outcode2[6] );
end. { IsLineOut }
```

Pseudo code 6: Elementary test to detect lines outside a box.

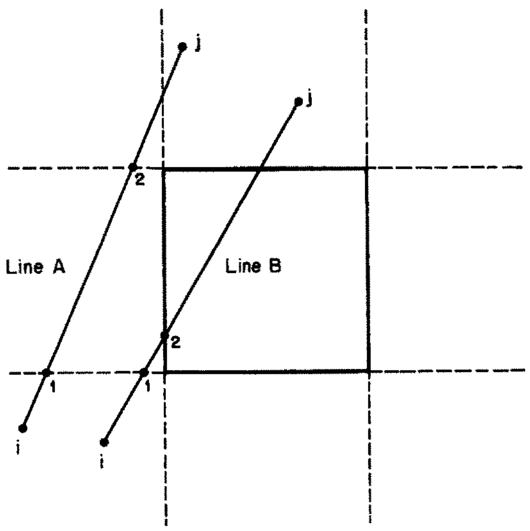

Fig. 6. Non-trivial situation for the region code analysis.

In Fig. 6, both lines A and B have the same region code, but A is outside the reference box and B is not. This is a typical situation where the region code analysis fails. To solve this problem, the algorithm recursively clips the line against a clipping plane (shown dashed in the figure) and performs the test with the clipped line. Thus if the test in the line *ij* is not conclusive, the algorithm reduces the line to 1*j*, and tests again. If the test is still non-conclusive the line is reduced to 2*j* and tested again. This process only stops when a line is either classified as trivially out (left, right, below, above, back or front) or an end node has region code '000000' (inside). In the former case the line crossing does not occur, and in the latter *LineSegX* function of Appendix A must be executed. The Psuedo code 7 illustrates this algorithm. The name *PickLine* was chosen because this algorithm can also be used to determine if a bar crosses a cursor box that the user has located on the viewing surface.

```
function PickLine ( bar :int; box : array [1..6] of real ) :logical;
{ verify if a bar intercepts a box }
var
  x1,y1,z1,x2,y2,z2     : real;
  outcode1,outcode2     : array [1..6] of logical;
  done                  : logical;
begin
  x2 ← x[NodeJ[bar]];
  y2 ← y[NodeJ[bar]];
  z2 ← z[NodeJ[bar]];
  execute OutCode( x2, y2, z2, box, outcode2);
  if IsNodeIn( outcode2 ) then
    PickLine ← TRUE;
  else begin
    x1 ← x[NodeI[bar]];
    y1 ← y[NodeI[bar]];
    z1 ← z[NodeI[bar]];
    done ← FALSE;
      execute OutCode( x1, y1, z1, box, outcode1);
      if IsLineOut( outcode1, outcode2) then begin
        Pickline ← FALSE;
        done    ← TRUE;
      else
        if outcode1[1] then
          execute ClipLineAtPlane(x1,y1,z1,x2,y2,z2,box[1]);
        else if outcode1[2] then
          execute ClipLaneAtPlane(x1,y1,z1,x2,y2,z2,box[2]);
        else if outcode1[3] then
          execute ClipLineAtPlane(y1,z1,x1,y2,z2,x2,box[3]);
        else if outcode1[4] then
          execute ClipLineAtPlane(y1,z1,x1,y2,z2,x2,box[4]);
        else if outcode1[5] then
          execute ClipLineAtPlane(z1,x1,y1,z2,x2,y2,box[5]);
        else if outcode1[6] then
          execute ClipLaneAtPlane(z1,x1,y1,z2,x2,y2,box[6]);
        else
          PickLine ← TRUE;
          done     ← TRUE;
        end if;
      end if;
    until done;
  end if;
end. { PickLine }
```

Pseudo code 7: Pick line algorithm.

The procedure *ClipLineAtPlane* clips the line going from point 1 to point 2 at a limit given in the last parameter. Note that this limit is always related to the first coordinate. Thus, to clip against constant y and z planes, a cyclic permutation must be performed in the nodal coordinates as shown in the Pseudo code 7.

```
proc ClipLineAtPlane ( var u1,v1,w1 :real; u2,v2,w2,lim :real);
{ clips the line going from point 1 to point 2 at u = lim }
var
  t : real;
begin
  t ← (lim-u1)/(u2 − u1);
  u1 ← lim;
  v1 ← v1 + t*(v2-v1);
  w1 ← w1 + t*(w2-w1);
end. { ClipLineAtPlane }
```

Pseudo code 8: Clipping against a constant plane in the first coordinate.

The optimized version of the simple method to add nodes at bar crossings is then given by the Pseudo code 9. Note that in the inner loop the compiler should only test for line crossing if the functions *PickLine* and *UnCoincNode* return TRUE. The latter seeks to avoid computing intersections between two bars that cross at their end nodes.

```
proc CreateNodeAtBarX;          {second implementation—uses filter }
{ detect bar intersections and create nodes at this position }
var
  box                 : array [1 .. 6] of real;
  bar1, bar2          : int;
begin
  for bar1 from 1 incr 1 to nbars − 1 do begin
    execute ComputerBarBox(bar1,box);
    for bar2 from bar1 + 1 incr 1 to nbars do
      if PickLine(bar2,box) then
        if UnCoincNode ( bar1,bar2) then
          if LineSegX( bar1,bar2,t) execute AddInternalNode(bar1,t);
        end if;
    end if; end for;
  end for;
end. { CreateNodeAtBarX }

proc UncoincNode ( bar1, bar2 : int );
{ detect if the incidence of bar1 and bar2 are not the same }
begin
  UncoincNode ← NodeI[bar1] ≠ NodeI[bar2] and
                NodeI[bar1] ≠ NodeJ[bar2] and
                NodeJ[bar1] ≠ NodeI[bar2] and
                NodeJ[bar1] ≠ NodeJ[bar2];
end. { UncoincNode }
```

Pseudo code 9: Add node at bar crossings using filters.

The third implementation of this procedure sorts the nodes and bars, changing the node numbers and the bar incidences. The nodes are sorted in y, x and z directions. That is, first the nodes are ordered according to their y coordinate (usually vertical). Nodes with the same y are then sorted according to their x coordinate. Finally, those with the same x and y are sorted in z. Appendix B shows an efficient implementation of the quick sort algorithm to do this task.

The bars are sorted by their end nodes. The bar incidences are first adjusted so that node i is always less than node j and bars with smaller node i will precede those with larger node i. If two bars have the same node i the one that has a smaller node j will precede the other.

With that kind of sorting, a given bar can only intersect a reference bar if node i of this bar is less than or equal to node j of the reference bar. Otherwise, the bar is 'above' of the reference bar. The ' ' are put here to remind that 'above' has a broader sense once the nodes are ordered also in x and z directions. Similarly, these two bars can only intersect if the node j of the given bar is greater than or equal to node i of the reference bar. That is, the given bar is not 'below' the reference bar. Note, however, that when two bars have a common end node (the equality condition) they need not be tested for intersection at an internal node. Furthermore, the *UnCoincNode* test can be reduced to the tests shown in the inner loop of the Pseudo code 10.

```
proc CreateNodeAtBarX;  { third implementation—uses order and filter }
{ detect bar intersections and create nodes at this position }
var
  box             : array [1 .. 6] of real;
  bar1,bar2       : int;
  t               : real;
begin
  for bar1 from 1 incr 1 to nbars-1 do begin
    execute ComputeBarBox ( bar1, box );
    for bar2 from bar1 + 1 incr 1 to nbars do
      if (NodeI[bar2] < NodeJ[bar1]) then
        if (NodeJ[bar2] > NodeI[bar1]) ) then
          if PickLine(bar2,box) then
            if (NodeI[bar1] ≠ NodeI[bar2] and
              NodeJ[bar1] ≠ NodeJ[bar2] and
              LineSegX(bar1,bar2,t) ) then
              execute AddInternalNode(bar1,t);
            end if;
          end if;
        end if;
      end if;
    end for;
  end for;
end. { CreateNodeAtBarX }
```

Pseudo code 10: Add node at bar crossings using sorting and filters.

5.2. *Division of bars to avoid internal nodes*

Free use of geometrical functions in a preprocessor can create nodes internal to bars. If the frame model incorporates these nodes, that point in space will have two sets of displacements, one from the node with its primary degrees-of-freedom and one from the bar internal displacements. Figure 7 illustrates basic inconsistencies arising from nodes in the middle of bars. The algorithms presented here aims to correct any combinations of these occurrences.

The presence of a node internal to a bar can be detected by several geometric algorithms. Probably, the simplest algorithm uses Schwarz's inequality and states that a node P is internal to a bar $b_i b_j$ if and only if

$$\|\overline{Pb_i}\| + \|\overline{Pb_j}\| = \|\overline{b_i b_j}\| \tag{1}$$

as shown in Fig. 8.

The approach based on eqn (1) requires the computation of three vector norms and is too expensive. The procedure *PickLine*, in Pseudo code 7, presents a less expensive test to verify if a node is internal to a bar. Furthermore, the *PickLine* algorithm has imbedded in it a simpler geometric interpretation for the floating point tolerance. It is half of the size of the edges of a cube that has the node in the center. A tolerance in eqn (1) would be associated with an ellipsoid with foci at the end nodes.

With a geometric test defined, the algorithm to divide bars to avoid internal nodes selects each bar as a reference bar and tests if this bar has internal nodes. This can be done by testing all nodes against the reference bar one at each time. If an internal node is found, the bar must then be divided in two: one going from node

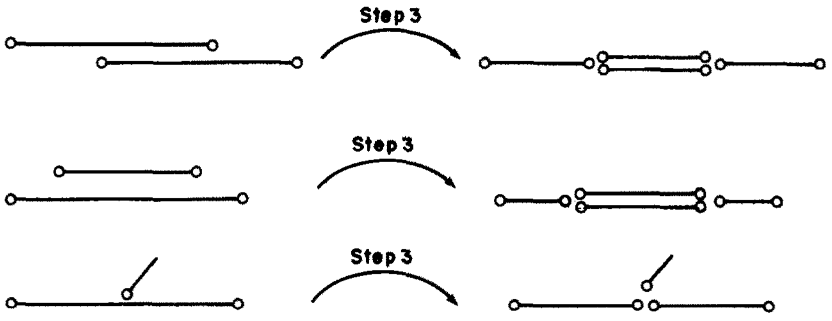

Fig. 7. Basic types of inconsistencies with nodes in the middle of the bars.

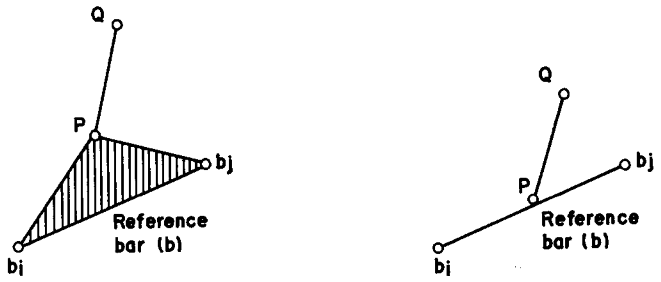

Fig. 8. Verification of internal node to the bar.

i of the reference bar to the internal node and another going from the internal node to the node *j* of the reference bar.

In the procedure just described the size of the bar array (list) is variable and can change in every step of the algorithm. If a recursive procedure could be established, the algorithm would test the newly created bars only with the remainder of the nodal list.

To avoid the use of recursive procedures (not available in FORTRAN) the procedure shown in the Pseudo code 11 keeps the first newly created bar as the current reference bar and puts the second one at the end of the bar list. After each change, the bar incidence array is thus updated. The process is repeated until all bars in the model are examined, including the new bars created. The numbers in brackets at the right of the Pseudo code 11 are line numbers put here for reference.

```
proc DivideBarToEliminateInternalNodes;  { first implementation } [L 0]
{ verify if a bar has an internal node; if so, divide the bar }
var                                                              [L 1]
  bar,node      : int;                                           [L 2]
  NodeBox       : array [1 .. 6] of real;                        [L 3]
begin                                                            [L 4]
  bar ← 0;                                                       [L 5]
  repeat begin                                                   [L 6]
    bar ← bar + 1;                                               [L 7]
    for node from 1 incr 1 to nnodes do                          [L 8]
      if NodeI[bar] ≠ node and NodeJ[bar] ≠ node then begin      [L 9]
        execute ComputeNodeBox (x[node],y[node],z[node], NodeBox) [L 10]
        if PickLine ( bar, NodeBox ) then begin                  [L 11]
          nbars ← nbars + 1;                                     [L 12]
          NodeI[nbars] ← node;                                   [L 13]
          NodeJ[nbars] ← NodeJ[bar];                             [L 14]
          NodeJ[bar] ← node;                                     [L 15]
        end if;                                                  [L 16]
      end if;                                                    [L 17]
    end for;                                                     [L 18]
  until bar > = nbars;                                           [L 19]
end. { DivideBarToEliminateInternalNodes }                       [L 20]
```

Pseudo code 11: Elimination of bar internal nodes (simple).

Note that line L9 of this algorithm excludes incident nodes of the reference bar from any further test. The procedure *ComputeNodeBox* is given in the Pseudo code 12.

```
proc ComputeNodeBox ( x_ref,y_ref,z_ref:real; var box:array [1 .. 6]:of real );
{ compute a box around a node with a given tolerance }
begin
  box[1] ← x_ref − TOL;
  box[2] ← x_ref + TOL;
  box[3] ← y_ref − TOL;
  box[4] ← y_ref + TOL;
  box[5] ← z_ref − TOL;
  box[6] ← z_ref + TOL;
end. { NodeBox }
```

Pseudo code 12: Compute a node box.

To improve efficiency, the second implementation uses the filter method. Instead of testing the reference bar against all node boxes, the algorithm can include a cheaper pre-test that checks if the node lies inside a bar box. The *PickLine* test is performed only for nodes meeting this condition. The new Pseudo code would then have the following lines included:

```
BarBox      : array [1..6] of real;                                   [L 2a]
  execute ComputeBarBox( bar, BarBox );                               [L 7a]
            execute ComputeBarBox( bar,BarBox ); { update BarBox }    [L 15a]
```

and the line L9 replaced by

```
if NodeI[bar] ≠ node and NodeJ[bar] ≠ node and                        [L 9]
  PickNode ( x[node],y[node],z[node], BarBox ) then begin             [L 9a]
```

The function *PickNode* is shown in Pseudo code 13. The name was chosen because this is the same function used in interactive graphical programs to select a node that lies in a given position on the viewing surface.

```
function PickNode(px,py,pz:real; box array[1..6]:real):logical;
begin
  PickNode ← (px > box[1]) and (px < box[2]) and
             (py > box[3]) and (py < box[4]) and
             (pz > box[5]) and (pz < box[6]);
end. {PickNode}
```

Pseudo code 13: Test if a node is within a box.

The third implementation assumes that the nodes and the bars are ordered in the same manner as shown in Pseudo code 11. As a consequence of this ordering, a reference bar always has the node i smaller than node j, and nodes with a smaller number have a smaller y, x or z coordinate. That is, if two nodes have number a and b ($a < b$) then y coordinate of a is less than or equal to the y coordinate of b. In case the y coordinates are equal the x coordinate of node a must be less than or equal to the x coordinate of node b. Finally, in case the x coordinates are also equal, the z coordinate of node a must be smaller. This order guarantees that only nodes between node i and node j of a reference bar can be internal to that bar. Psuedo code 14 shows the final algorithm after changes to consider ordered nodes.

```
proc DivideBarToEliminateInternalNodes;  { third implementation—uses order and filters }
{ verify if a bar has an internal node; if so, divide the bar. }
var
  bar,node          : int;
  nb,noi,noj        : int;
  NodeBox,BarBox : array [1..6] of real;
begin
  nb ← nbars;
  for bar from 1 incr 1 to nb do
    execute ComputeBarBox( bar, BarBox );
    noi ← NodeI[bar] + 1;
    nof ← NodeJ[bar] − 1;
    for node from noi incr 1 to nof do
      if PickNode (x[node],y[node],z[node],BarBox) then begin
        execute ComputeNodeBox (x[node],y[node],z[node], NodeBox)
        if PickLine ( bar, NodeBox ) then begin
          nbars ← nbars + 1;
          NodeI[nbars] ← NodeI[bar];
          NodeJ[nbars] ← node;
          NodeI[bar]   ← node;
          execute ComputeBarBox( bar, BarBox );
        end if;
      end if;
    end for;
  end for;
end. { DivideBarToEliminateInternalNodes }
```

Pseudo code 14: Elimination of bar internal nodes using sorting and filters.

Note that, due to the order, the newly created bars do not need to be tested against any node. All nodes between node *i* and node *j* have already been tested when the bar is created.

5.3. *Elimination of repeated nodes (and bars with null size)*

The elimination of repeated nodes aims to eliminate any node that, within a certain tolerance, has the same coordinates as a previous one. Bars of null size with equal or different incident nodes can also be eliminated from the model. If the incident nodes in one bar of null size are different, this step will transform the bar definition to the case where the incident nodes are equal. Its elimination is then just a topological (non-geometrical) problem.

To implement this routine, the nodes with repeated coordinates are detected first. Afterwards, the element incidence list and the nodal coordinate list are updated. Pseudo code 15 shows three procedures, one for each one of those conditions. The procedure *NewNodeNumber* must be executed before the procedures *UpdateBarIncidence* and *UpdateNodalCoordinates*. The procedure *Zero* assigns zero for all elements of an integer vector array.

```
proc NewNodeNumber (var NewNumber : array [1 .. MAX_NODES] of int );
                                        { first and second implementation }
{ compute the new number of an existing node if repeated coordinates
  are eliminated from the coordinate arrays ( x, y, z )
  (the new number of current node i is NewNumber[i]) }
var
  node    : int; { Counter for the new list }
  box[6]  : real; { Tolerance box around a node }
  i, j    : int;
begin
  node ← 0;
  execute Zero( MAX_NODES, NewNumber );
  for i from 1 incr 1 to nnodes do begin
    if NewNumber[i] = 0 then begin
      node ← node + 1;
      NewNumber[i] ← node;
      execute ComputeNodeBox(x[i],y[i],z[i],box);
      for j from i + 1 incr 1 to nnodes do
        if PickNode(x[j],y[j],z[j],box) then do NewNumber[j] ← node;
      end for;
    end if;
  end for;
end. { NewNodeNumber }
proc UpdateBarIncidence ( NewNumber : array [1 .. MAX_NODES] of int );
{ update bars incidence according to the vector NewNumber }
var
  i : int;
begin
  for i from 1 incr 1 to nbars do begin
     NodeI[i] ← NewNumber[NodeI[i]];
     NodeJ[i] ← NewNumber[NodeJ[i]];
  end for;
end. { UpdateBarIncidence }
proc UpdateNodalCoordinates ( NewNumber : array[1 .. MAX_NODES] of int );
{ update nodal coordinates according to the vector NewNumber }
var
  i,node : int;
begin
  node ← 1;
  for i from 1 incr 1 to nnodes do
    if NewNumber[i] = node then begin
      x[node] ← x[i];
      y[node] ← y[i];
      z[node] ← z[i];
      node ← node + 1;
    end if;
  end for;
  nnodes ← node − 1;
end. { UpdateNodalCoordinates }
```

Pseudo code 15: Elimination of repeated nodes (simple and filter).

With the repeated nodes removed, the bars with null size can also be eliminated from the bar list. This elimination needs to be done only the first time that this step is executed (step 1) in both algorithms in Sec. 2. The bars with null size do not need to be checked in step 7 of the algorithm that treats bars that cross at internal points because that situation does not occur at this stage. Pseudo code 16 shows how this elimination can be performed.

```
proc EliminateBarNullSize;          {first and second implementation}
var { eliminate bars of null size }
  bar  : int; { counter for the new bar list }
  i    : int; { counter for the old bar list }
begin
  bar ← 0;
  for i from 1 incr 1 to nbars do
    if  NodeI[i] ≠ NodeJ[i] then begin
        bar ← bar + 1;
        NodeI[bar] ← NodeI[i];
        NodeJ[bar] ← NodeJ[i];
    end if;
  end for;
  nbars ← bar;
end. { EliminateBarNullSize }
```

Pseudo Code 16: Elimination of bar with null size (simple and filter).

The geometric test to detect if two nodes are coincident is the *PickNode* test presented above. This test is so simple that no filter strategy needs to be done. Therefore, this step needs only two distinct methods: one without (1st and 2nd implementation), and one with (3rd implementation) sorting of nodes and bars.

With the nodes sorted (see Appendix B), coincident nodes occupy consecutive positions in the node list. Therefore, it is sufficient to test each node only with the previous node of the list. Repeated nodes have a predecessor in the same position. The algorithm to remove these nodes is shown in the Pseudo code 17. Note that this algorithm uses an integer nodal label computed at the *SortNodes* procedure that uniquely defines a (x,y,z) position. That is, two nodes occupy the same position if and only if their labels are equals. Moreover, if the label of a node is greater than the label of another, the first node comes after the second in the sorted list.

```
proc NewNodeNumber ( LabelList : array [1 .. MAX_NODES] of int;
                     var NewNumber : array [1 .. MAX_NODES] of int );
                                       { third implementation—uses order }
{ compute the new number of an existing node if the repeated coordinates are eliminated from the
  coordinate arrays ( x, y, z )
  (the new number of current node i is NewNumber[i]) }
var
  node : int;
  i    : int;
begin
  node ← 1;
  NewNumber [1] ← 1;
  for i from 2 incr 1 to nnodes do
    if LabelList[i] = LabelList[i − 1] then
       NewNumber[i] ← NewNumber[i − 1];
    else begin
        node ← node + 1;
        NewNumber[i] ← node;
    end if;
  end for;
end. { NewNodeNumber }
```

Pseudo Code 17: Elimination of repeated nodes (order).

Note also that this algorithm has one loop less than that the one shown in Pseudo code 15. The precedures *UpdateBarIncidence* and *UpdateNodalCoordinates* for this implementation are similar to the ones presented in Pseudo code 15.

5.4. *Elimination of isolated nodes*

Another step that may be required for consistent data is the elimination of nodes without incidence (isolated nodes). This step may not be performed if the analysis program does require such nodes. An example of the use of isolated nodes in the analysis is the node K used in the SAP program. In this situation the existence of an error may not be so clear. The degrees of freedom of the isolated nodes should be checked to avoid singular stiffness matrices.

If, however, the analysis program does not require isolated nodes, a proper action would be to delete them from the node list. Their existence increases unnecessarily the graphical data structure and the application data structure.

An algorithm to eliminate isolated nodes can be developed in three steps:

1. on the basis of the bar incidence, mark all nodes that are not isolated;
2. compute a new number for these nodes (omitting the isolated ones);
3. update the node list and the bar list.

The Pseudo code 18 illustrates this algorithm. Note that this algorithm only uses topological tests that are too simple to use filters. The sorting of nodes and bars also does not improve this algorithm; thus, only one implementation is presented for this step.

```
proc DeleteIsolatedNodes;        {first, second, and third implementation}
{ eliminate nodes without incidence (isolated) }
  NewNumber   : array [1 .. Max_a] of int; { New mode number }
  node,i, j   : int;
  update_flag : logical;
begin
 { mark nodes without incidence with NewNumber equal to 0, the others will have in NewNumber their new
  number in the list that excludes the isolated nodes }
 { step 1: mark connected nodes }
  execute ZERO ( MAX_NODES, NewNumber );
  for j from 1 incr 1 to nbars do begin
    NewNumber[NodeI[j]] ← −1;
    NewNumber[NodeJ[j]] ← −1;
  end for;
 { step 2: mark isolated nodes and set update flag }
  node ← 0;
  update_flag ← FALSE;
  for i from 1 incr 1 to nnodes do
    if NewNumber[i] = −1 then begin
      node ← node + 1;
      NewNumber[i] ← node;
    else
      update_flag ← TRUE;
    end if;
  end for;
 { step 3: Update Data Structure (if necessary) }
  if update_flag then begin
   execute UpdateBarIncidence(NewNumber);
   execute UpdateNodalCoordinates(NewNumber);
  end if;
end. { DeleteIsolatedNodes }
```

Pseudo code 18: Elimination of nodes without incidence.

5.5. *Elimination of repeated bars*

A procedure to eliminate repeated bars of a model that do not have repeated nodes can rely only on the topology of the model. The geometric questions have already been solved by the elimination of repeated nodes.

The algorithm proposed here first computes for each bar an integer code (*LabelList*) that is a function of its incidences (nodes *i* and *j*). This label must be the same in the case of bars with equal or inverse incidence. The Pseudo code 19 illustrates this computation.

```
proc ComputeBarLabel ( var LabelList : array [1 .. MAX_BARS] of int );
{ compute a label for each bar }
var
  i          : int;
begin
  for i from 1 incr 1 to nbars do
    if NodeI[i] < NodeJ[i] then
      LabelList[i] ← ISHFL(NodeI[i],15) + NodeJ[i];
    else
      LabelList[i] ← ISHFL(NodeJ[i],15) + NodeI[i];
    end if;
  end for;
end. { ComputeBarLabel }
```

Pseudo code 19: Compute bar label (without change of incidence).

The function *ISHFL*, shown in the code, shifts the binary representation of a number fifteen positions to the left. This function is available in both C or FORTRAN and is equivalent to a multiplication to 2^{15} (32768). The obvious restriction of this approach is that node the number must be less than this value. This restriction can be waived if a longer integer representation is used. Unsigned integer in the C language, for example, would allow numbers up to 2^{16} (65536).

With the labels computed, each bar is then tested with the following bars of the list. When two codes are coincident, the repeated bar is eliminated. Here again the use of filters is not applicable. Topological tests are too simple to be replaced. Pseudo code 20 presents this algorithm.

```
proc EliminateRepeatedBars ( var LabelList : array [1 .. MAX_BARS] of int );
                                             { first and second implementation }
 { eliminate bars with same incidence or reverse incidence }
var
  label  : int;
  bar    : int;
  i, j   : int;
begin
  bar ← 1;
  for i from 2 incr 1 to nbars do begin
     j ← 0;
     repeat begin
       j ← j + 1;
     until j > bar or LabelList[j] = LabelList[i]
     if j > bar then begin
       bar            ← bar + 1;
       NodeI[bar]     ← NodeI[i];
       NodeJ[bar]     ← NodeJ[i];
       LabelList[bar] ← LabelList[i];
     end if;
  end for;
  nbars ← bar;
end. { EliminateRepeatedBars }
```

Pseudo code 20: Elimination of repeated bars (simple and filter).

The third implementation uses the bar data observed as illustrated by the Pseudo code 21. After sorting, node *i* is always less than node *j* and the bars are ordered with respect to node *i*. Bars with the same node *i* are ordered with respect to node *j*.

```
proc SortBars ( var LabelList : array [1 .. MAX_BARS] of int );
 { sort the bars according to their incidence }
var
  i,temp       : int;
begin
 { step 1: label all bars uniquely according to their node numbers }
  for i from 1 incr 1 to nbars do
    if NodeI[i] > NodeJ[i] then begin
      temp ← NodeI[i];
      NodeI[i] ← NodeJ[i];
      NodeJ[i] ← temp;
    end if;
    LabelList [i] → ISHFL(NodeI [i], 15) + Node J [i];
  end for;
 { step 2: sort the list of bars with respect to array LabelList }
   execute QSortB(LabelList,NodeI,NodeJ,1,nbars);
end. { SortBars }
```

Psuedo code 21: Sort bars according to their incidence.

The procedure *QSortB*(E,F,G,$n1$,$n2$) is given in Appendix B. This procedure sorts the integer vector arrays E, F and G as a function of the elements of E, between the lower and upper position limits, $n1$ and $n2$. The sorting routine used here is an adaptation of the quick sort algorithm presented by Houlsby and Sloan [8] and Sedgewick [9].

With the bar labels sorted, repeated bars occupy consecutive positions in the bar label list. This fact is used to develop Pseudo code 22 that is an improved version of Pseudo code 20.

```
proc EliminateRepeatedBars ( var LabelList : array [1 .. max_a] of int );
                                             { third implementation—uses order }
{ eliminate bars with same incidence or reverse incidence }
var
  label        : int;
  bar          : int;
  i, j,t       : int;
begin
  bar     ← 1;
  for i from 2 incr 1 to nbars do begin
    if LabelList[i] ≠ LabelList[i − 1] then begin
         bar             ← bar + 1;
         NodeI[bar]      ← NodeI[i];
         NodeJ[bar]      ← NodeJ[i];
         LabelList[bar] ← LabelList[i];
    end if;
  end for;
  nbars ← bar;
end. { EliminateRepeatedBars }
```

Pseudo code 22: Elimination of repeated bars (order).

6. EXAMPLES

To numerically evaluate the algorithms proposed in this paper, three implementations were produced using the following methods: simple, filter, and order (sorting or filters plus sorting), as described in Sec. 3. The general outline of the implementation using filters plus sorting is given by:

Step A:	Label and sort nodes;
Step B:	Elimination of repeated nodes;
Step C:	Elimination of bars of null size;
Step D:	Elimination of nodes without incidence;
Step E:	Change bar orientation;
Step F:	Division of bars to avoid internal nodes;
Step G:	Label and sort bars;
Step H:	Elimination of repeated bars;

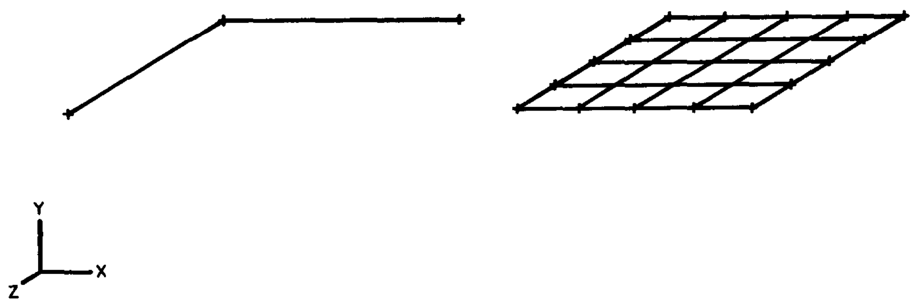

Fig. 9. Example 1: orthogonal grid.

Table 2. Results of the consistency analysis for the four meshes of example 1

Example 1	Initial numbers		Final numbers		PC/AT time (sec)			SUN time (sec)		
Mesh	Nodes	Bars	Nodes	Bars	Simple	Filter	Order	Simple	Filter	Order
4 × 4	19	10	25	40	1.8	0.67	0.28	0.09	0.01	0.00
8 × 8	35	18	81	144	16	4.0	1.1	0.63	0.17	0.06
16 × 16	67	34	289	544	185	38	4.7	7.1	1.6	0.28
32 × 32	131	66	1089	2112	2566	453	23	99	20	1.2

Step I:	Creation of nodes at bars crossings;
Step J:	Label and sort nodes;
Step K:	Elimination of repeated nodes;
Step L:	Division of bars to avoid internal nodes.

To improve efficiency, steps J, K and L are only executed if step I does create nodes at bar intersections, that is, if the problem does have bars intersecting at internal nodes.

The simple algorithm and the algorithm with filters can be derived from the outline just presented by eliminating steps A, E and J and by replacing the remaining steps by their proper version. In those versions, step G only computes bar labels.

The language chosen for the numerical implementation was FORTRAN 77. The programs were implemented on a micro-computer PC/AT 286 running MS-DOS 4.0 at 12 MHz and also in a SUN 4/370.

Four examples were chosen to evaluate these implementations: two in plane and two in three-dimensional space. All the examples were created using a frame preprocessor. The first example is an orthogonal floor gridwork created in the manner illustrated by Fig. 9. In this figure, the girders were created with lines defined by the end points shown as + (plus character). To create the initial mesh, the user defined first the top and left lines as illustrated in Fig. 9. These lines were then copied in the vertical and horizontal directions, respectively.

Table 2 presents the total number of nodes and bars before (initial) and after (final) the consistency was imposed for the four meshes of the type shown in Fig. 9. That table also shows the time obtained for the three methods: (simple), using filters (filter), and using sorting plus filters (order) on both the PC/AT and the SUN workstation. Obviously all three implementations produced the same results. However, the names

Table 3. Detailed results of the consistency analysis of the 32 × 32 mesh of example 1

Example 1—32 × 32 mesh	Initial numbers		Final numbers		PC/AT time (sec)		
Step	Nodes	Bars	Nodes	Bars	Simple	Filter	Order
Label and sort nodes (1)	131	66	131	66			0.17
Elim. repeated nodes	131	66	128	66	0.98	0.99	0.00
Elim. bars null size	128	66	128	66	0.00	0.00	0.06
Elim. nodes without incidence	128	66	128	66	0.00	0.00	0.00
Change bar orientation (1)	128	66	128	66			0.00
Div. bars to avoid internal nodes	128	66	128	190	26.04	6.31	0.99
Label and sort bars (2)	128	190	128	190	0.00	0.00	0.11
Elim. of repeated bars	128	190	128	190	0.17	0.22	0.00
Create nodes at bar crossings	128	190	1089	190	25.21	16.43	7.15
Label and sort nodes (1)	1089	190	1089	190			3.02
Elim. repeated nodes	1089	190	1089	190	57.62	57.72	0.00
Div. bars to avoid internal nodes	1089	190	1089	2112	2455.66	371.79	11.53
Total					2565.69	453.46	23.03

Notes: (1) Only for the implementation with sort. (2) For simple and filter implementations no sorting is performed.

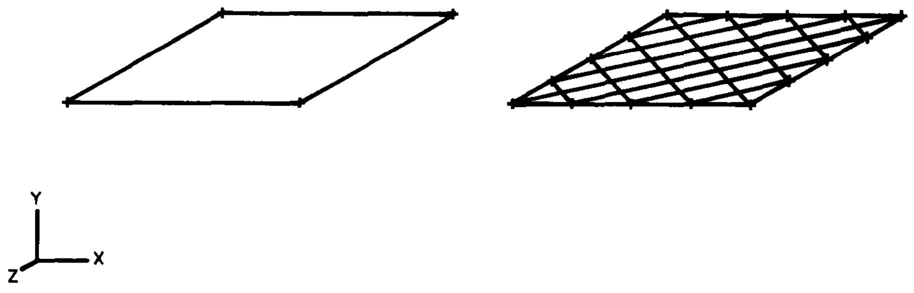

Fig. 10. Example 2: non-orthogonal grid.

(numbers) of the nodes and bars are different in the last method (order). Table 3 shows the results after each step of the algorithms for the 32 × 32 mesh of the orthogonal grid. The times presented in this table refer to the PC/AT implementation.

The second example deals with non-orthogonal grids of bars of the type illustrated by Fig. 10. The process of creation of the initial mesh starts with the definition of the external bars at left, top, right and bottom. These bars are then divided into 4, 8, 16 or 32 parts depending on the mesh. Bars are then placed joining these nodes as illustrated in Fig. 10 for a 4 × 4 mesh. Note that in this figure the boundary nodes are also marked with a + (plus) character.

Table 4 shows the total number of nodes and bars and the time for the consistency analysis for the four meshes of the second example. Table 5 details these numbers for each step of the analysis for the 32 × 32 mesh. Table 4 presents both PC/AT and SUN times and Table 5 only the former.

The third example is obtained by duplicating the bar cube shown in Fig. 11 in the x and z directions. Four meshes are analyzed: 2 × 2, 4 × 4, 6 × 6 and 8 × 8. Figure 11 also illustrates the 4 × 4 mesh of this example. Note that diagonal bars are placed on the cube faces to produce bar crossings.

Table 6 shows the total number of nodes and bars and the time for the consistency analysis for the four meshes of the third example. Table 7 details these numbers for each step of the analysis for the 8 × 8 mesh. Table 6 presents both PC/AT and SUN times and Table 7 only the former. For this particular example, Fig. 12 shows a qualitative comparison among the three methods used in this paper.

The last example is a roof generated by duplicating the pyramid bar shown in the upper left corner of Fig. 13 in the x and z directions. After the duplication, the outstanding top bars are trimmed to produce the roof also shown in Fig. 13. Four meshes are analyzed in this example: 2 × 2, 4 × 4, 6 × 6 and 8 × 8. Note that in this example both algorithms presented in Sec. 2 are equivalent because there are no bar crossings.

Table 4. Results of the consistency analysis for the four meshes of example 2

Example 2	Initial numbers		Final numbers		PC/AT time (sec)			SUN time (sec)		
Mesh	Nodes	Bars	Nodes	Bars	Simple	Filter	Order	Simple	Filter	Order
4 × 4	16	30	41	80	4.8	2.0	0.8	0.18	0.08	0.05
8 × 8	32	62	145	288	53	17	4.4	2.0	0.67	0.16
16 × 16	64	126	545	1088	697	202	25	27	7.4	1.12
32 × 32	128	254	2113	4224	10252	2809	159	395	103	7.1

Table 5. Detailed results of the consistency analysis of the 32 × 32 mesh of example 2

Example 2—32 × 32 mesh	Initial numbers		Final numbers		PC/AT time (sec)		
Step	Nodes	Bars	Nodes	Bars	Simple	Filter	Order
Label and sort nodes (1)	128	254	128	254			0.44
Elim. repeated nodes	128	254	128	254	0.99	0.94	0.00
Elim. bars null size	128	254	128	254	0.00	0.05	0.00
Elim. nodes without incidence	128	254	128	254	0.05	0.00	0.00
Change bar orientation (1)	128	254	128	254			0.00
Div. bars to avoid internal nodes	128	254	128	254	37.63	12.58	7.09
Label and sort bars (2)	128	254	128	254	0.00	0.05	0.06
Elim. of repeated bars	128	254	128	254	0.33	0.33	0.00
Create nodes at bar crossings	128	254	2113	254	46.74	35.32	21.97
Label and sort nodes (1)	2113	254	2113	254			7.14
Elim. repeated nodes	2113	254	2113	254	225.47	225.69	0.05
Div. bars to avoid internal nodes	2113	254	2113	4224	9941.57	2533.88	122.10
Total					10252.7	2808.84	158.86

Notes: (1) Only for the implementation with sort. (2) For simple and filter implementations no sorting is performed.

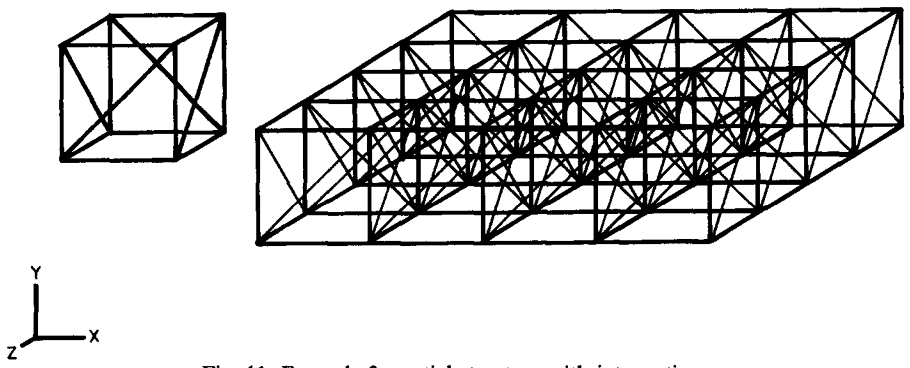

Fig. 11. Example 3: spatial structure with intersections.

Table 6. Results of the consistency analysis for the four meshes of example 3

Example 3	Initial numbers		Final numbers		PC/AT time (sec)			SUN time (sec)		
Mesh	Nodes	Bars	Nodes	Bars	Simple	Filter	Order	Simple	Filter	Order
2 × 2	32	64	22	57	4.2	1.6	0.9	0.13	0.07	0.03
4 × 4	128	256	74	217	51	17	10	1.9	0.79	0.53
6 × 6	288	576	158	481	234	75	41	8.6	3.4	2.26
8 × 8	512	1024	274	849	705	220	118	26	10	6.6

Table 7. Detailed results of the consistency analysis of the 8 × 8 mesh of example 3

Example 3—8 × 8 mesh	Initial numbers		Final numbers		PC/AT time (sec)		
Step	Nodes	Bars	Nodes	Bars	Simple	Filter	Order
Label and sort nodes (1)	512	1024	512	1024			5.27
Elim. repeated nodes	512	1024	162	1024	4.84	4.83	0.06
Elim. bars null size	162	1024	162	1024	0.00	0.00	0.00
Elim. nodes without incidence	162	1024	162	1024	0.00	0.00	0.00
Change bar orientation (1)	162	1024	162	1024			0.05
Div. bars to avoid internal nodes	162	1024	162	1024	175.21	23.61	5.99
Label and sort bars (2)	162	1024	162	1024	0.11	0.05	0.49
Elim. of repeated bars	162	1024	162	625	3.08	3.08	0.06
Create nodes at bar crossings	162	625	274	625	271.55	151.70	96.34
Label and sort nodes (1)	274	625	274	625			1.81
Elim. repeated nodes	274	625	274	625	3.85	3.79	0.00
Div. bars to avoid internal nodes	274	625	274	849	246.01	32.47	8.18
Total					704.65	219.53	118.31

Notes: (1) Only for the implementation with sort. (2) For simple and filter implementations no sorting is performed.

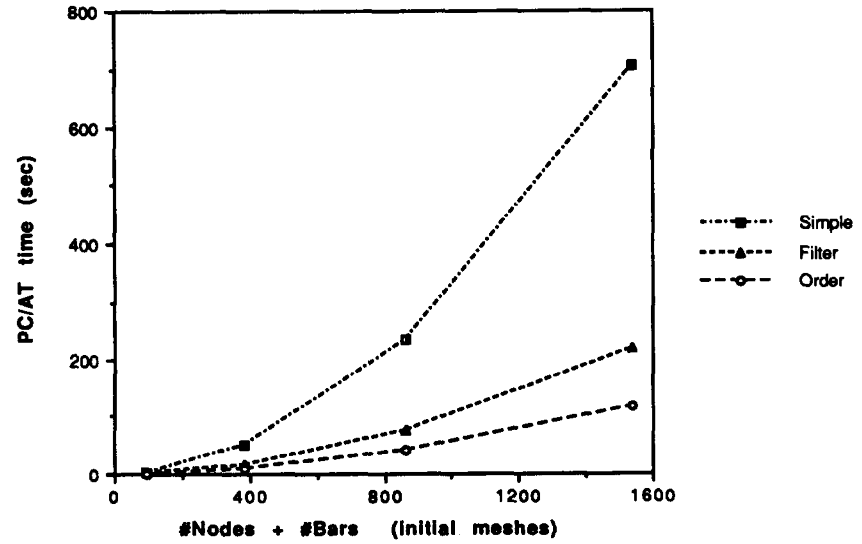

Fig. 12. Comparison of the methods for consistency (example 3).

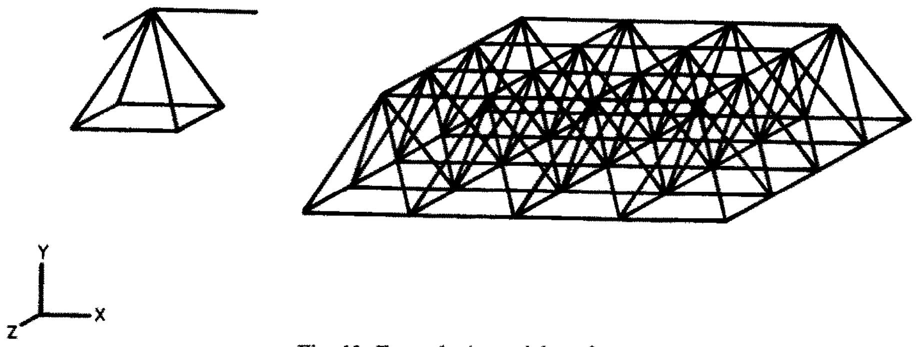

Fig. 13. Example 4: spatial roof.

Table 8. Results of the consistency analysis for the four meshes of example 4

Example 4	Initial numbers		Final numbers		PC/AT time (sec)			SUN time (sec)		
Mesh	Nodes	Bars	Nodes	Bars	Simple	Filter	Order	Simple	Filter	Order
2 × 2	24	36	13	32	1.2	0.6	0.3	0.04	0.04	0.00
4 × 4	104	152	41	128	18	7.6	4.6	0.67	0.35	0.24
6 × 6	240	348	85	288	89	38	21	3.2	1.7	1.16
8 × 8	432	624	145	512	279	118	67	10	5.2	3.6

Table 9. Detailed results of the consistency analysis of the 32 × 32 mesh of example 4

Example 4—8 × 8 mesh	Initial numbers		Final numbers		PC/AT time (sec)		
Step	Nodes	Bars	Nodes	Bars	Simple	Filter	Order
Label and sort nodes (1)	432	624	432	624			2.81
Elim. repeated nodes	432	624	145	624	3.68	3.68	0.05
Elim. bars null size	145	624	145	624	0.00	0.00	0.00
Elim. nodes without incidence	145	624	145	624	0.00	0.00	0.00
Change bar orientation (1)	145	624	145	624			0.00
Div. bars to avoid internal nodes	145	624	145	624	95.29	12.24	2.63
Label and sort bars (2)	145	624	145	624	0.06	0.06	0.27
Elim. of repeated bars	145	624	145	512	1.54	1.59	0.06
Create nodes at bar crossings	145	512	145	512	178.34	100.46	59.86
Total					278.91	118.03	65.68

Notes: (1) Only for the implementation with sort. (2) For simple and filter implementations no sorting is performed.

Table 8 shows the total number of nodes and bars and the time for the consistency analysis for the four meshes of the last example. Table 9 details these numbers for each step of the analysis for the 8 × 8 mesh. Table 8 presents both PC/AT and SUN times and Table 9 only the former.

7. CONCLUSIONS

Several techniques have been proposed to treat the geometric and topological inconsistencies of three dimensional frame models. These techniques are based on a simple data structure which is familiar to finite element programmers.

The results, for the examples presented here, have shown that a simple and naive approach to the problem (first implementation) can severely compromise the effectiveness of the user-computer dialog in interactive systems. Based on the examples presented, this conclusion is more strongly noticed in the following steps listed by order of importance: 'Division of bars to avoid internal nodes', 'Creation of nodes at bar crossings', and 'Elimination of repeated nodes'. Therefore, the search for efficient algorithms to produce a consistent finite element data structure is very important.

Although effective solutions were proposed for the step of 'Division of bars to avoid internal nodes', the authors feel that there is still room for improvement in order to reduce the response time of this step.

The use of 'filters' is effective, but the use of 'order' is much better. The time taken to sort the data is not relevant if compared with the savings obtained in any of the steps.

The authors envisage that the techniques presented here can be also useful in engineering education. If these algorithms are implemented as a separate module of an interactive graphical frame preprocessor, they can be used to verify if the new students have input the data for analysis properly.

Acknowledgements—The work presented in this paper is part of the research carried out by the last two authors in partial fulfillment of the requirements for M.Sc. degrees in Civil Engineering at the Pontifícia Universidade Católica do Rio de Janeiro. The authors wish to acknowledge CNPq, CAPES and STC (Brazilian Science and Technology Agencies) for funds to support this research. They wish also to acknowledge the collaboration of Miguel José Reis Lopes, Waldemar Celes Filho and Mark L. Valenzuela.

REFERENCES

1. K. Preiss, Checking the topological consistency of a finite element mesh. *Int. J. Numer. Meth. Engng* **14,** 1805–1812 (1979).
2. G. H. Paulino, Preprocessing of three-dimensional framed structures, with nodal reordering, using interactive computer graphics (in Portuguese). M.Sc. dissertation, Civil Engineering Department, PUC-Rio (1988).
3. C. I. Pesquera, W. McGuire and J. F. Abel, Interactive graphical preprocessing of three-dimensional framed structures. *Comput. Struct.* **17,** 1–12 (1983).
4. F. P. Preparata and M. I. Shamos, *Computational Geometry*. Springer, New York (1985).
5. I. O. Angell and G. Griffith, *High-resolution Computer Graphics using FORTRAN 77.* Macmillan, London (1987).
6. D. Hearn and M. P. Baker *Computer Graphics*. Prentice-Hall, Englewood Cliffs, NJ (1986).
7. J. D. Foley, A. van Dam, S. K. Feiner and J. F. Hughes, *Computer Graphics: Principles and Practice*, 2nd Edn. Addison-Wesley, Reading, MA (1990).
8. G. T. Houlsby and W. Sloan, Technical note: efficient sorting routines in FORTRAN 77. *Adv. Engng Software* **6,** 198 (1984).
9. R. Sedgewick, Implementing quick sort programs. *Commun. ACM* **21,** No. 10 (1978).

APPENDIX A: INTERSECTION OF LINE SEGMENTS

The intersection tests are based on the parametric equations of the line segments. Figure A1 illustrates the proposed process: the reference bar is named bar$_1$ and the testing bar is named bar$_2$. The parametric equations of the two line segments of bar$_1$ and bar$_2$ are:

$$\begin{aligned} x &= xi_1 + (xj_1 - xi_1)^*t & \qquad x &= xi_2 + (xj_2 - xi_2)^*u \\ y &= yi_1 + (yj_1 - yi_1)^*t \quad \text{and} & y &= yi_2 + (yj_2 - yi_2)^*u \\ z &= zi_1 + (zj_1 - zi_1)^*t & z &= zi_2 + (zj_2 - zi_2)^*u. \end{aligned}$$

The intersection point, whether existent, can be obtained making equal x, y and z in the two sets of equations above

$$\begin{aligned} (xj_1 - xi_1)^*t - (xj_2 - xi_2)^*u &= xi_2 - xi_1 \\ (yj_1 - yi_1)^*t - (yj_2 - yi_2)^*u &= yi_2 - yi_1 \\ (zj_1 - zi_1)^*t - (zj_2 - zi_2)^*u &= zi_2 - zi_1 \end{aligned} \tag{A1}$$

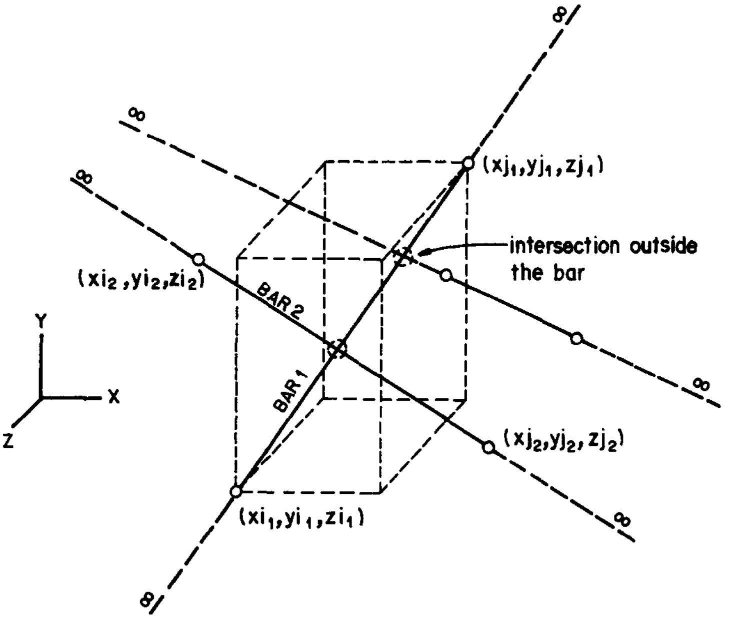

Fig. A1. Bar crossings (after Paulino [2]).

which specifies a system with three equations and two unknowns

$$\begin{bmatrix} (xj_1 - xi_1) & -(xj_2 - xi_2) \\ (yj_1 - yi_1) & -(yj_2 - yi_2) \\ (zj_1 - zi_1) & -(zj_2 - zi_2) \end{bmatrix} \begin{Bmatrix} t \\ u \end{Bmatrix} = \begin{Bmatrix} xi_2 - xi_1 \\ yi_2 - yi_1 \\ zi_2 - zi_1 \end{Bmatrix}$$

or

$$\begin{bmatrix} dx_1 & -dx_2 \\ dy_1 & -dy_2 \\ dz_1 & -dz_2 \end{bmatrix} \begin{Bmatrix} t \\ u \end{Bmatrix} = \begin{Bmatrix} dx_i \\ dy_i \\ dz_i \end{Bmatrix}, \qquad \text{(A2)}$$

where

$$dx_\alpha = xj_\alpha - xi_\alpha, \quad dy_\alpha = yj_\alpha - yi_\alpha, \quad dz_\alpha = zj_\alpha - zi_\alpha, \quad \text{for } \alpha = 1,2$$

and

$$dx_i = xi_2 - xi_1, \quad dy_i = yi_2 - yi_1, \quad dz_i = zi_2 - zi_1.$$

To solve the above system of equations one can solve three systems of two equations and two unknowns. Using Cramer's rule is necessary to compute three determinants from matrices of order 2×2.

The existence of intersections between bars may be examined through three possibilities of occurrence of one system of two equations linearly independent. Each system represents equations in one of the planes xy, xz or yz. The existence of intersection in the plane does not guarantee the existence of intersection in space. For this reason the solution needs to be checked in the third remaining equation.

For instance, suppose that the two first equations in (A1) are linearly independents. If $\det_0$, $\det_t$ and $\det_u$ are the determinants given by:

$$\det_0 = \mathrm{DET}\begin{bmatrix} dx_1 & -dx_2 \\ dy_1 & -dy_2 \end{bmatrix}$$

$$\det_t = \mathrm{DET}\begin{bmatrix} dx_i & -dx_2 \\ dy_i & -dy_2 \end{bmatrix}$$

$$\det_u = \mathrm{DET}\begin{bmatrix} dx_1 & dx_i \\ dy_1 & dy_i \end{bmatrix}$$

then

$$t = \frac{\det_t}{\det_0} \quad \text{and} \quad u = \frac{\det_u}{\det_0}.$$

For internal points to the line segment defined by bar_1 and bar_2 the following condition must be true

$$u, t \in (0, 1). \qquad \text{(A3)}$$

If the condition (A3) is obeyed, the values of u and t must still be checked in the third equation. If u and t satisfy the three equations in (A1), then the coordinates of the intersection point between the bars are calculated and one additional node is created. If the first two equations in (A1) are linearly dependent, the algorithm must test the two remaining systems of two equations each.

The Pseudo code A1 was prepared according to the concepts above. The logical function *LineSegX* returns TRUE if the two testing bars intersect, and the intersection point is given by the parameter t.

```
function LineSegX ( bar1, bar2 : int; var t : real ) : logical;
{ verify and treat line intersections }
var
  dx1,dy1,dz1        : real;
  dx2,dy2,dz2        : real;
  dxi,dyi,dzi        : real;
  u                  : real;
  error              : real;
  solution,internal  : logical;
begin
  execute Vector(NodeI[bar1],Nodej[bar1],dx1,dy1,dz1);
  execute Vector(NodeI[bar2],NodeJ[bar2],dx2,dy2,dz2);
  execute Vector(NodeI[bar1],NodeI[bar2],dxi,dyi,dzi);
  { solve in xy }
  execute Solve( dx1, -dx2,dy1, -dy2,dxi,dyi,t,u,solution,internal );
  if solution then
    if internal then begin
      error ← dz1*t - dzu2*u - dzi;
      LineSegX ← ABS (error) < TOL;
    end if;
  else begin { solve in xz }
    execute Solve( dx1, -dx2,dz1, -dz2,dxi,dzi,t,u,solution,internal );
  if solution then
    if internal then begin
      error ← dy1*t - dy2*u - dyi;
      LineSegX ← ABS (error) < TOL;
    end if;
    else begin { solve in yz }
      execute Solve( dy1, -dy2,dz1, -dz2,dyi,dzi,t,u,solution,internal );
      if solution then
        if internal then begin
          error ← dx1*t - dx2*u - dxi;
          LineSegX ← ABS (error) < TOL;
        end if;
      end if;
    end if;
  end if;
end. { LineSegX }
proc Vector ( node1, node2 : int; var dx, dy, dz : real );
begin
  dx ← x[node2] - x[node1];
  dy ← y[node2] - y[node1];
  dz ← z[node2] - z[node1];
end. { Vector }
proc Solve ( a11, a12, a21, a22, b1, b2 : real; var t, u : real; var solution, internal : logical );
{ solve an equation system of two equations and two unknowns: t,u }
var
  det0,dett,detu          :real
begin
  det0 ← a11*a22 - a21*a12;
  solution ← ABS (det0) > TOL;
  if solution then begin
    dett     ← b1*a22 - b2*a12;
    detu     ← a11*b2 - a21*b1;
    t        ← dett / det0;
    u        ← detu / det0;
    internal ← ( t > 0 and t < 1 ) and (u > 0 and u < 1 );
  else
    internal ← FALSE;
  end if;
end. { Solve }
```

Pseudo code A1: Determination of bar crossings.

APPENDIX B: NODAL SORTING

In the second implementation of all steps presented in this paper the list of nodes is sorted according to their y, x, and z coordinates, respectively (y is assumed to be the vertical direction in buildings). That is, first the nodes are ordered according to their y coordinates. Nodes with the same y are then sorted according to their x coordinates. Finally, those with the same x and y are sorted in z.

This type of sorting of nodes can be accomplished by replacing the comparison statement of a conventional sorting routine by a function that does the comparisons. Another interesting approach assigns a unique integer label for every node in such a way that the order of the labels is the same as the nodes. Psuedo code B1 illustrates a procedure to compute this integer label.

```
proc SortNodes ( var LabelList : array [1 .. MAX_NODES] of int );
{ computes a code for each node and sort the nodes }
var
  max,min                   : real;
  ix,iy,iz                  : int;
  factorx,factory,factorz   : real;
  NewNumber                 : array [1 .. MAX_NODES] of int;
  i, j,n,b                  : int;
begin
  { step 0 : Put in the NewNumber vector of the original position of the nodes }
  for i from 1 incr 1 to nnodes do
      NewNumber[i] ← i;
  end for;
  { step 1 : compute maximum and minimum coordinates }
  xmin,xmax ← x[1];
  ymin,ymax ← y[1];
  zmin,zmax ← z[1];
  for i from 2 incr 1 to nnodes do begin
      if xmin < x[i] then xmin ← x[i];
      if xmax > x[i] then xmax ← x[i];
      if ymin < y[i] then ymin ← y[i];
      if ymax > y[i] then ymax ← y[i];
      if zmin < z[i] then zmin ← x[i];
      if zmax > z[i] then zmax ← x[i];
  end for;
  { step 2 : compute factors that will reduce data to the range [0,1024] }
  if ( xmax – xmin ) > TOL then
      factorx ← 1024 / ( xmax – xmin );
  else
      factorx ← 1024;
  end if;
  if ( ymax – ymin ) > TOL then
      factory ← 1024 / ( ymax – ymin );
  else
      factory ← 1024;
  end if;
  if ( zmax – zmin ) > TOL then
      factorz ← 1024 / ( zmax – zmin );
  else
      factorz ← 1024;
  end if;
  { step 3 : transform data to 10 bit integers and compute the labels }
  for  i from incr 1 to nnodes do begin
       ix ← x[i] * factorx;
       iy ← y[i] * factory;
       iz ← z[i] * factorz;
       LabelList[i] ← ISHFL(iy,20) + ISHFL(ix,10) + iz;
  end for;
  { step 4 : sort according to the labels }
  execute QSortN( LabelList,NewNumber,x,y,z,1,nnodes );
  { step 5 : update the bar incidence according to NewNumber }
  for b from 1 incr 1 to nbars do begin
       n ← 0;
       repeat
           n ← n + 1;
           until ( NodeI[b] = NewNumber[b] );
           NodeI[b] ← n;
       n ← 0;
       repeat
           n ← n + 1;
           until ( NodeJ[b] = NewNumber[n] );
           NodeJ[b] ← n;
  end for;
end. { SortBars }
```

Pseudo code B1: Nodal Sorting.

The procedure *QSortN*(A,B,C,D,E,$n1$,$n2$) sorts the integer vector arrays (A and B) and the real vector arrays (C, D and E) as a function of the elements of A, between the lower and upper position limits, $n1$ and $n2$. The sorting routine used here is an adaptation of the quick sort algorithm presented by Houlsby and Sloan [8] and Sedgewick [9].

Note that the integer coordinates (ix,iy,iz) computed for each node have a geometric interpretation. They represent a snap of all nodes in a grid that divides the bounding box of the structure into 1024 × 1024 × 1024 cells. The label list computed here is also useful to identify repeated nodes, as discussed in Sec. 4.5 of this paper.