DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm Explanation</td>
<td>1</td>
</tr>
<tr>
<td>CimStation CAD models</td>
<td>1</td>
</tr>
<tr>
<td>Figure 1 Surfaces in U,V and X,Y,Z space</td>
<td>1</td>
</tr>
<tr>
<td>Importing inspection point data</td>
<td>2</td>
</tr>
<tr>
<td>Need for a new algorithm</td>
<td>2</td>
</tr>
<tr>
<td>Details of the algorithm</td>
<td>2</td>
</tr>
<tr>
<td>Figure 2 U,V points compared to x,y,z point</td>
<td>4</td>
</tr>
<tr>
<td>Example of algorithm during 1 iteration</td>
<td>5</td>
</tr>
<tr>
<td>Figure 3 U,V space for example</td>
<td>5</td>
</tr>
<tr>
<td>Formulas / Mathematics used for the algorithm</td>
<td>6</td>
</tr>
<tr>
<td>For distance between two points</td>
<td>6</td>
</tr>
<tr>
<td>For the comparison of the normal vectors</td>
<td>6</td>
</tr>
<tr>
<td>Algorithm Software Design Diagram</td>
<td>7</td>
</tr>
<tr>
<td>Software Function Descriptions</td>
<td>8</td>
</tr>
<tr>
<td>five_by_5()</td>
<td>8</td>
</tr>
<tr>
<td>smallest_dst()</td>
<td>9</td>
</tr>
<tr>
<td>angl_between_vecs()</td>
<td>9</td>
</tr>
</tbody>
</table>
Algorithm Explanation:

Cimstation CAD models

Silma's Cimstation running the inspection module contains geometry for the part, machine, fixtures, inspection probes, etc. such that all geometry present during an actual part inspection can be accurately programmed and simulated. The part model is generally brought into the Cimstation's CAD system from the original CAD system through translations. Silma uses similar techniques as other CAD systems to store the geometry. The actual data representing the geometry (looking deep into the layers of detail) is like a two dimensional coordinate system along with additional information needed to allow mapping these data into x,y,z space.

1. The part model geometry is stored in the Cimstation CAD system in U-V coordinate space along with surface type and other mapping data. The Cimstation software takes care of the "type" surface when mapping from U-V space to X,Y,Z. Surface types are parametric surfaces, B-splines, etc.

   Example
   Surface in U,V space

   One possible mapping of the Surface in X,Y,Z space

![Image: Figure 1 Surfaces in U,V and X,Y,Z space](image)

2. At any given U,V point on a surface, the point's normal vector can also be obtained from the Cimstation software. The U,V coordinates are passed through the appropriate, surface type dependent function to yield the x,y,z coordinates and, if needed, i,j,k normal vectors.

   \[ f(u,v) \rightarrow f(t) \] yields \( f(x,y,z,i,j,k) \)

3. Because of this method of storing surface data and also because the usual use of this system is to have the Cimstation software calculate the x,y,z information, going the other way .. that is beginning with x,y,z and trying to find an equivalent u,v surface point .. is generally not needed and therefore not fully supported. The base functions are missing that allow taking x,y,z data points and then finding the nearest surface and the nearest point on that surface to some x,y,z point.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
Importing inspection point data

4. There are times during part programming when the customer might dictate the exact places where inspections are to take place. This has been supported by Silma through the use of "importing" points. The problem has been that these "imported" points were not/ could not be verified as being correct ... that is to say verified to be exactly on the surface to be inspected. One portion of the CRADA with Silma has corrected that problem.

Need for a new algorithm

5. In order to accomplish the task of verification of these imported points translated into the need to develop two different algorithms:

(a) An algorithm to, given an x,y,z point, find the nearest surface stored in the Cimstation's CAD system.

and

(b) An algorithm to, given an x,y,z,i,j,k point and the nearest surface to that point, find the nearest surface point to the given point and ensure that it is close enough to the surface point and that the given normal vector is close enough to the surface normal vector at the nearest surface point.

For algorithm (a), existing Silma functions (generally used for collision detection) were used. This method of determining the nearest surface, after some final modifications, works well. Of all the surfaces stored in the Cimstation, the algorithm either determines the NEAREST surface or else determines that NONE of the surfaces are near enough to make it wise to continue.

For algorithm (b), after failing to find any information at all indicating that a suitable algorithm already exists available to the general public, an algorithm was created/developed. The algorithm developed methodically iterates on increasingly smaller areas of U,V space calculating each distance between a series of equally spaced (in u,v space) u,v points and a given x,y,z point. Each of the equally spaced u,v points must be transformed into x,y,z points so that the distance from them to the given x,y,z point can be calculated.

Because many types of surfaces are supported on Cimstation, each with their own unique mapping algorithms to get from u,v space to x,y,z space ... care had to be taken to NOT try to converge too quickly on the nearest point, else it might be overlooked due to rapid curvature changes in x,y,z space for some types of surfaces. The "final" version of the algorithm, after it proved to work on MMES type surfaces, was modified at Silma because it failed to work on one of the type surfaces that Silma's other customers use. It was modified to converge a little more slowly than originally planned but now works on all surfaces tested.

Details of the algorithm

6. Some details of the algorithm used to find the nearest point on the closest surface to a given, imported x,y,z point are:
* The algorithm was written to solve this one specific problem ..
  to either find that the given point is within an operator
  changeable distance tolerance (point okay .. accept it) or else
  find that the point is not within the distance tolerance (point
  NOT okay .. reject it).

* An iterative method was used and so a maximum limit on the
  number of iterations (now set to 17) had to be coded.

* The first thing required was to find the "domain" of the
  surface within it's U,V space ... a rectangular section of U,V
  space with a maximum U, maximum V, minimum U, and minimum V.

* The technique employed was to begin each iteration with an area
  of U,V space (initially the domain of the surface), and then
  through a series of calculations reduce the area of U,V space
  in which to continue a search for the nearest surface point to
  our given point. This particular algorithm can reduce the area
  by a factor of around 5 to 1 .. that is to say, each iteration
  will look at about 1/5 the space as the previous iteration ..
  thus converging on the nearest surface point.

* During each iteration a distance is calculated between the
  given imported x,y,z point and each of 36 evenly spaced points
  (evenly spaced in u,v space). The closest distance value is
  one of the values returned by the algorithm function.

* At the completion of each iteration, the closest 5 points are
  used to determine a new, smaller area of u,v space in which to
  continue the search for the next iteration. This smaller area
  is calculated by looking at the u,v coordinates of the nearest
  5 points and finding the minimum u and v and the maximum u and
  v. An adjustment to the code was made to this portion of the
  logic during the final Silma visit. (See "adjustment note")

* At the completion of each iteration, the closest surface
  distance found is compared to an operator changeable distance
  tolerance and, if it is out of tolerance, the process of
  looking further continues. If it is within tolerance the
  iteration loop is abandoned and the imported point's x,y,z
  values are accepted as close enough and the normal vector
  comparison is applied.

* If the maximum number of iterations occurs without finding a
  surface point close enough to our given point, one more try is
  made by calculating the mid-point of the remaining U,V space
  and then using that point to see if it is within the distance
  tolerance of the given point. The point is rejected if not in
  tolerance. If it is in tolerance, the point is accepted as
  okay and the normal vector comparison is applied.

(adjustment note): To allow the algorithm to work on ALL surface
  types, the area calculated as the new, smaller search area has
  been increased in size in all 4 directions (+u,-u,+v,-v) by 10%
  of the total beginning ranges to ensure it does not converge
  too quickly and miss the proper surface range.
Sketch of a portion of U,V space after it has been "mapped" into x,y,z space.

(A) Some of the 36 (marked on sketch) U,V points calculated for each iteration of the algorithm.

(B) The GIVEN IMPORTED x,y,z point.

(C) The surface point closest to the given x,y,z point. This point is seldom found exactly ... the iterations only continue until a point close enough is found or until the iteration count exceeds the maximum limit. The default tolerance is 0.001 cm and so, even if the point is not exactly found, very little error exists.

(D) The x,y,z distance between the given imported point and, in the case shown, the closest surface point (C). A distance (D) is calculated for each one of the 36 points (A).

(E) The shaded area represents the new, smaller area of U,V space in which to continue searching that would probably result from data shown in this example.

(F) Shaded points are the points that would probably have been found to be the 5 closest points to our given point. Their U,V values would then be used to calculate the new search area (E).
Example of algorithm during 1 iteration

Example: Suppose that the beginning $U,V$ ranges for the area of $u,v$ space to look at were:

$U$ between 0.0 and 1.0 and $V$ between 0.0 and 1.0

Further, suppose that the $u,v$ coordinates of the five points found to be nearest to our given $x,y,z$ point were:

0.2,0.4 0.2,0.6 0.4,0.4 0.4,0.6 and 0.0,0.4

Then the NEW, reduced range of where to look would become:

$U$ between 0.0 and 0.4 and $V$ between 0.4 and 0.6

Now, to apply the "fix" to the algorithm, we would calculate the "additional" $u$ and $v$ amounts to add to the above calculated range values. Both $U$ and $V$ ranges were the same on entry... between 1.0 and 0.0 and 1/5 of that is 0.2 and 1/2 of that is 0.1. We would now add this value to the maximums and subtract it from the minimums.... taking care NOT to allow these new values to get into areas outside of the original ranges. The result would be:

$U$ between 0.0 and 0.5 and $V$ between 0.3 and 0.7

![Figure 3 U,V space for example](image-url)
Formulas / Mathematics used for the algorithm

The only formulas/mathematics required for the code to implement the algorithm were:

For distance between two points:

Point 1 \((x_1,y_1,z_1)\) and Point 2 \((x_2,y_2,z_2)\)

Used the Square root of the sum of the squares of the differences in \(X,Y,Z\) .... distance = \(\text{SQRT}((x_1-x_2)^2 + (y_1-y_2)^2 + (z_1-z_2)^2)\)

For the comparison of the normal vectors:

Surface Point \((i_1,j_1,k_1)\) and Given Point \((i_2,j_2,k_2)\)

Used the Square root of the sum of the squares of the differences in \(I,J,K\) .... distance = \(\text{SQRT}((i_1-i_2)^2 + (j_1-j_2)^2 + (k_1-k_2)^2)\) and then, because that represents the distance between the "tips" of two unit vectors (each length is 1.0), used the arcsin function of \(1/2\) the calculated distance to find the angle between these two vectors. The operator changeable tolerance is, by default, set to \(0.00174\) radian.
*** rewritten for CRADA ***
verify imported point
FUNCTION
dverify_point()

** EXISTING **
find nearest surface
FUNCTION
d_verify_find_surface()

** EXISTING **
find domain of surf.
FUNCTION
dom()

reduce U,V search area
FUNCTION
five_x_5()

** EXISTING **
get xyz/ijk from u,v
FUNCTION
seg_pose()

find smallest distance
FUNCTION
smallest_dist()
Software Function Descriptions:

Function Description: five_by_5()

On entry, a rectangular section of U,V space is defined by two corner points P1 and P2. P1 is the upper left hand corner and P2 is the lower right hand corner of the rectangle. The software in this function then breaks this u,v space into 25 equally sized smaller rectangles. The result is 36 [nodes] intersections (see below) counting the outside legs of the rectangle. At each intersection [node], the software takes the U,V value .. through existing functions .. obtaining the X,Y,Z value at that node. In the below diagram, these nodes are labeled (a-z and 1-0).

Each node X,Y,Z is compared to the given (imported) point X,Y,Z and the distance between them is stored along with references to identify the node the distance came from.

Once all 36 distances have been computed, the software then loops through them to find the FIVE closest distances .. along with the U,V coordinates of the FIVE closest distances.

From the five closest U,V points to our given (imported) point, the U's and V's of their location are then looped through to obtain the largest U, smallest U, largest V, and smallest V. These minimum and maximum U's and V's are then returned to the calling function so they may be used to define a new set of "corner points" for a much smaller area of U,V space in which to further look for the nearest U,V coordinates to our given (imported) point.

In the above diagram, the point labeled (@) represents the place in U,V space that is nearest to a given (imported) point. After running one iteration of the five_x_5 function on it, the new point corners returned to the calling function would be those represented by node [2] and node [x] which reduces the area by 2/25. Each subsequent iteration will reduce the "search" area by 2/25 or, in some cases perhaps 4/25.
Smallest distance function documentation

\texttt{smallest\_dst()}

Given an array of 36 distances, finds the smallest and returns it and the u,v coordinates where it was found. Uses the array of 36 distances as though it were a two dimensional (6 x 6) array. The first 6 entries represent (in array coordinates) 0,0 .. 1,0 .. 2,0 .. etc. The second set of entries represent 0,1 .. 1,1 .. 2,1 .. etc. and so on for 6 rows with 6 columns in each.

\textbf{Note:} It is up to the calling routine to have logic to keep from finding the same least distance over and over. This function does not reorder nor does it change any of the values in the array.

angle between vectors function documentation

\texttt{angl\_between\_vecs()}

Calculates the angle between two i,j,k unit vectors given as "point" arguments (vec1 and vec2). Outputs the angle between the two vectors in radian.

Finds the distance between the two vector (tips) using the square root of the sum of the squares (i1-i2, j1-j2, k1-k2). Then uses half that distance with the arcsin function to determine one half of the angle between the vectors.

\[
\begin{align*}
\text{angle between vectors} & \quad \text{length = 1} \\
\text{vector 1} & \quad \text{vector 2} \\
\text{length = 1} & \\
\text{length = 1} & \quad \text{length = 1} \\
\text{length = 1} & \quad \text{length = 1} \\
\end{align*}
\]

\[
\text{Keep the answer in RADIAN in case the system returns 0 for the case of 180 degrees (would occur if the vectors are equal, but in opposite direction).}
\]
CRADA (Y-1292-0162)
Software Operational Instructions

by Richard B. Riker
S.Q.A.C. Inc.

Version 3/21/94
# Table of Contents

Overview ................................................................. 1

Using the CRADA software ............................................ 2
  LOADING THE CIMSTATION ........................................... 2
  LOADING THE CRADA SOFTWARE ..................................... 2
  OTHER THINGS TO NOTICE ........................................... 2
      When picking inspection points .............................. 2
      When trying to output a feature/tolerance ............... 2
      New warnings and new error messages ..................... 3
Overview:

A CRADA between MMES and SILMA Inc. has recently been completed. The result of this CRADA is software developed jointly to enhance the Cimstation's verification of inspection part programs as they are being developed.

This software currently exists both at Silma and on the inspection part programming Cimstation in the concurrent engineering room in 9737. It was written for Cimstation (version 4.5) and should only be loaded with that version.

SILMA has already added most of this code (or the functionality of this code) to their new inspection module software thus assuring continued support for this work.

The addition of this software makes it much less likely that part programmer errors occur and features the following new tests, checks and verifications:

* Verification that imported points lie within an operator changeable distance tolerance from the part's surface.

* Verification that imported point vectors compare to the part surface's normal vector within an operator changeable tolerance amount.

* Verification that manually "picked" inspection points lie on the selected surface. Warning is issued when not true.

* Verification that the "set" of points used to measure a feature are in fact appropriate for the type of feature being inspected.

* Verification that at least the minimum required inspection points are taken for the type of feature being inspected.

* Software to put ONLY the DMIS statements allowed in a MEAS block into the MEAS block, any others are put AFTER the MEAS block.

* Software to ensure that only the tolerances allowed to be combined with each feature type are offered for application of tolerance to a given feature.

* Verification (and optional correction) that the MEAS statement's number of points agrees with the number of points measured.

* Software to issue a Warning/Reminder in cases where a tolerance is being applied and the potential need for BOUNDING exists.
Using the CRADA software:

LOADING THE CIMSTATION:

Ensure that you use the CRADA software with the proper version of the inspection module software.

the below is seen when "starting" proper version of Cimstation:

```
******** ******** ** ** ** ** ** ** ** ******** ********
..................... ********************* **
Cimstation Version 4.5
SILMA, Inc.
Copyright 1993
All rights reserved
--------------- **
** ..................... ********************
```

LOADING THE CRADA SOFTWARE:

Once the Cimstation (version 4.5) has been loaded, you must now load the software developed for the CRADA. Presently this software must be loaded in via the SIL prompt ... either in TEXT MODE or in MENUS MODE.

At the SIL prompt, type the following command:

```
Sil> ld('/usr/people/ew067/rr5/finalcode');
```

You will then see the various procedures and functions being loaded. This takes about one minute to complete. (Note: perhaps later on this code will be "compiled" into version 4.5 thus taking away the need for this step.)

OTHER THINGS TO NOTICE:

Now that the correct version of Cimstation is running and you have loaded the CRADA software, you will visually notice some differences in operation. You will also have some new menu options offered.

* When picking inspection points.

When picking inspection points notice that the "verification" of these points is automatically applied. Each time you "click" the mouse on the part the Feature on the part where the point is to lie will be colored reddish purple while the verification algorithm is at work.

* When trying to output a feature/tolerance combination.

When trying to output a feature/tolerance combination, you will notice that the DMIS feature you select will influence which of the tolerances are shown to you for selection. This prevents things like trying to apply a "flatness" tolerance on a cylinder.
* New warnings and new error messages.

Pay close attention to the area of the screen where error messages and warnings are shown. Many new messages now exist to help the verification process.

Error, Warning, and Information Messages in NEW CODE

write_msg's
These show up in the 1 line window above SIL window

Warning - CMM surface selected.
surface not identifiable
Calculated point not on surface. Point not generated.
Warning - the tolerance TOLERANCEID may require bounding.
For use with ANGLB or DISTB tolerances only.
Cannot use additional tolerances with the tolerance chosen
Need to specify an associated feature (Asoc ft) for selected tolerance
Importing point file FILENAME
Done with file FILENAME

writeln's
These show up in the SIL window

Picked section curve.
Warning - Point not on originally selected surface.
Calculated point not on surface. Point not generated.
Warning - the tolerance TOLERANCEID may require bounding.
Warning - NNNN points found to be off feature FEATURENAME
All points OK for FEATURENAME
Sorry, no test available for FEATURENAME in insert dmis stat DMISYNTAX
Rejected, Point is off surface by X.XXXX
Rejected, Normal is off X.XXXX radians from surface normal
Rejected, Point is off surface further than VARIABLETOLERANCE
*** looking at point X.XXXX, X.XXXX, X.XXXX
Warning: N Points not on surface out of NNN total points.
All NNN points found to be within tolerance

below, writeln's that can be removed from code

(for testing) Surface found = SURFACENAME
(for testing) orig.umin,umax X.XXX X.XXX vmin,vmax X.XXX X.XXX
(for testing) iteration number NN
(for testing) Point found on/near surface after iterations = NN
(for testing) distance between is X.XXXX
CRADA Y-1292-0162
SOFTWARE LISTINGS
# TABLE OF CONTENTS

| Initialization variables | 1 |
| function check_feat_min_pts | 1 |
| function d_check_hit | 3 |
| function d_verify_find_surface | 3 |
| procedure c_init_verify_stuff | 4 |
| function dselect_surf_points | 4 |
| function dselect_surf_points | 6 |
| function check_tol_feat | 7 |
| function check_tol_feat | 7 |
| function cneck_tol_feat | 8 |
| function cneck_tol_feat | 8 |
| function check_tol_feat | 8 |
| function check_tol_feat | 8 |
| function check_tol_feat | 8 |
| function check_tol_feat | 9 |
| function check_tol_feat | 9 |
| function check_tol_feat | 9 |
| function check_tol_feat | 9 |
| function check_tol_feat | 9 |
| function check_tol_feat | 10 |
| function check_tol_feat | 10 |
| function check_tol_feat | 10 |
| function check_tol_feat | 10 |
| function check_tol_feat | 10 |
| function check_tol_feat | 10 |
| function get_tolerance | 11 |
| function check_tol_for_feat | 11 |
| function check_tol_for_feat | 11 |
| procedure check_tol_for_bound | 12 |
| function dselect_tol_name | 12 |
| function dselect_tol_name_list | 13 |
| procedure doutput_menu_handler | 14 |
| function verify_point_on_feat | 16 |
| function verify_point_on_feat | 17 |
| function verify_point_on_feat | 17 |
| function verify_point_on_feat | 17 |
| function verify_point_on_feat | 17 |
| function verify_point_on_feat | 18 |
| function verify_point_on_feat | 18 |
| function verify_point_on_feat | 18 |
| function verify_point_on_feat | 19 |
| function verify_point_on_feat | 19 |
| function verify_point_on_feat | 19 |
| procedure verify_all_feat_pnts | 20 |
| procedure verify_pnts_menu_handler | 20 |
TABLE OF CONTENTS
(continued)

function is_valid_meas_member ............... 21
procedure check_valid_insert ............... 21
procedure insert_dmis_stmt ................. 22
procedure init_gen_pt_set ................. 22
procedure dgen_pt_settings_menu_handler ... 23
procedure create_cur_cs_point ............. 24
dsmallest = lrecord ....................... 25
function smallest_dst .................... 26
dcorners = lrecord ....................... 26
function five_x_5 ......................... 27
function angl_between_vecs ................. 30
function dverify_point .................... 31
procedure dimport_points ................. 33
{ Initialization variables for MMES Crada work.
   11.jan.94   jt
   edits:
}
d_verify_point_tol == .001;
d_verify_normal_tol == .01745;

{ feature minimum inspection points needed: }
circle_minimum_points := 3;
plane_minimum_points := 3;
arc_minimum_points := 3;
cone_minimum_points := 6;
cylinder_minimum_points := 6;
line_minimum_points := 2;
ellipse_minimum_points := 5;
sphere_minimum_points := 4;
rectangle_minimum_points := 9;
point_minimum_points := 1;
parallel_planes_minimum_points := 4;
pattern_minimum_points := 2;
gcurve_minimum_points := 2;
psurf_minimum_points := 3;

d_is_manual_select == false;

{ ------------------------------- }

{ FUNCTION: check_feat_min_pts }
{ ------------------------------- }
function check_feat_min_pts(ftype: id; num_pts: integer): anno_integer;
var is_ok: anno_integer;
   min_pts: integer;
begin
   is_ok := mk_anno_integer(true, 0);
   case ftype of
    "circle": begin
        is_ok.anno := (num_pts >= circle_minimum_points);
        min_pts := circle_minimum_points;
        end;
    "plane": begin
        is_ok.anno := (num_pts >= plane_minimum_points);
        min_pts := plane_minimum_points;
        end;
    "arc": begin
        is_ok.anno := (num_pts >= arc_minimum_points);
        min_pts := arc_minimum_points;
        end;
    "cone": begin
        is_ok.anno := (num_pts >= cone_minimum_points);
        min_pts := cone_minimum_points;
        end;
    "cylindr": begin
        is_ok.anno := (num_pts >= cylinder_minimum_points);
        min_pts := cylinder_minimum_points;
        end;
    "line": begin
        is_ok.anno := (num_pts >= line_minimum_points);
        end;
    end;
end;

---
min_pts := line_minimum_points;
end;
"ellips: begin
  is_ok.anno := (num_pts >= ellipse_minimum_points);
  min_pts := ellipse_minimum_points;
end;
"sphere: begin
  is_ok.anno := (num_pts >= sphere_minimum_points);
  min_pts := sphere_minimum_points;
end;
"rectngl: begin
  is_ok.anno := (num_pts >= rectangle_minimum_points);
  min_pts := rectangle_minimum_points;
end;
"point: begin
  is_ok.anno := (num_pts >= point_minimum_points);
  min_pts := point_minimum_points;
end;
"gsurf: begin
  is_ok.anno := (num_pts >= gsurf_minimum_points);
  min_pts := gsurf_minimum_points;
end;
"gcurve: begin
  is_ok.anno := (num_pts >= gcurve_minimum_points);
  min_pts := gcurve_minimum_points;
end;
"patern: begin
  is_ok.anno := (num_pts >= pattern_minimum_points);
  min_pts := gcurve_minimum_points;
end;
end;
if not(is_ok.anno) then
  is_ok.val := min_pts;

check_feat_min_pts := is_ok;
end;

{ file cr_io.sil 18.nov.93  jt
  reads in points from user files and creates inspection points
  Currently only works for ANVIL bulk point data example.

  Complies with Section 1 of Crada Statement of Work.

  edits:
  ** Modified for MMES Crada 11.jan.94  jt **
}
3

** ThP
tile most not contain blank lines at end.

** Oata must be in the current coordinate system or part cs.

** Currency limitations units should be the same as in the data file. **

data in file looks like:

```
12.232 3.423 43.399 0.000 0.000 1.000
```

{ This will find a surface that the point lies near. }
{ Currently being used as the only check until NMES writes a better one. }

{ ----------------------------- }  
{ FUNCTION: d_check_hit }  
{ ----------------------------- }  

'' temp fix... ** uncomment if you are running less than Inspection 1.4.1 **

function d_check_hit(x, y: shape): boolean;
var ret: boolean;
begin
if (hidden(x) or is_empty_shape(x)) then d_check_hit := false
else begin
  v_inquire_coldet_11(x.seg,y.seg,vgp_collision_info);
  if ((vgp_collision_info.segl as_type ob) = (x.seg as_type ob)) then
    ret := true
  else ret := false;
  d_check_hit := ret;
end;
end;  
end of temp fix }

{ example usage: }

** The file must not contain blank lines at end. **
** Data must be in the current coordinate system or part cs. **
** Current cimation units should be the same as in the data file. **

{ ----------------------------- }  
{ FUNCTION: d_verify_find_surface }  
{ ----------------------------- }  

function d_verify_find_surface(pt, nm: point): shape;
begin
  moveto(d_coll_fellow, pt);
  if d_is_manual_select then
    d_verify_find_surface := selected_shape
  else
    d_verify_find_surface := d_get_hit(d_coll_fellow, false, ' ');
end;

verify_col_det_size := 0.55;

{ Complies with section 2 of Crada Statement of Work }

do_manual_select_verify := true;
procedure c_init_verify_stuff();
begin
  if is_object('detector_1') then remove('detector_1');
  if not(in_world(d_coll_fellow)) then add(d_coll_fellow,'detector_1');
end;

function dselect_surf_points(do_pnt, do_verify: boolean):seg3dr;

var
  pnt, vt: point;
  uvguess: param2dr;
  uguess, vguess: real;
  ans, verify, point_ok: boolean;
  ret: seg3dr;
  temps: shape;
{ dfeat_part is shape - feature of part }
begin
  verify := do_verify;
  ret := mk_seg3dr(null_point, null_point);
  if (not(null(selected_shape_to_view_psurface))) then
    begin
      uvguess := guess_parm(selected_shape as_type psurface);
      uguess := uu(uvguess);
      vguess := vv(uvguess);

      pnt := seg_pose(selected_shape) * eval_at(selected_shape as_type psurface, uguess, vguess);
      vt := ornt(seg_pose(selected_shape)) * normal_at(selected_shape as_type psurface, uguess, vguess);
      ret.end0 := pnt;
      ret.end1 := vt;
    end
  else if good_slice_pick() then
    begin { points on cross sections }
      verify := false; [ no way we can check for this ]
      writeln('Picked section curve.');

      if not(facet_was_selected) then error('both curve and surface must be displayed');
      if (not(null(selected_facet_shape_to_view_psurface))) then
        begin
          uvguess := guess_parm_on_slice(selected_facet_shape as_type psurface, selected_shape);
          uguess := uu(uvguess);
          vguess := vv(uvguess);

          pnt := seg_pose(selected_facet_shape) * eval_at(selected_facet_shape as_type psurface, uguess, vguess); [ in global ]
          vt := ornt(seg_pose(selected_facet_shape)) * normal_at(selected_facet_shape as_type psurface, uguess, vguess);
        end
    end
  end
end;

else if (not(null(selected_facet_shape to_view dsurface))) then
begin
pnt := selected_facet_point;
vt := selected_facet_normal;
end;
ret.end0 := pnt;
ret.end1 := vt;
end
else if (facet_was_selected) then
begin
verify := false; { no way we can check for this }
if (do_pnt and not(expert_user)) then
begin
temps := selected_object;
while parent(temps) <> world do
temps := parent(temps);
if is_robot(temps) then write_msg('Warning - CMM surface selected.');</end;
pnt := selected_facet_point;
vt := selected_facet_normal;
ret.end0 := pnt;
ret.end1 := vt;
end
else if (not(null(selected_shape to_view point_shape))) then
begin [ start off in global coords ]
verify := false; { no way we can check for this }
pnt := null_point rel pose(selected_shape);
vt := d-vector; { no normal associated with this type of point }
ans := mget_graphics_pick('select point on normal');
if ans then
begin
vt := null_point rel pose(selected_shape);
vt := normalize( (vt - pnt ) as_type point);
end;
ret.end0 := pnt;
ret.end1 := vt;
end
else if (not(null(selected_shape to_view opoint_shape))) then
begin
verify := false; { no way we can check for this }
pnt := null_point rel seg_pose(selected_shape);
vt := ornt(seg_pose(selected_shape)) * d-vector;
ret.end0 := pnt;
ret.end1 := vt;
end
else if (not(null(selected_shape to_view d_point))) then
begin
verify := false; { no way we can check for this }
pnt := null_point rel pose(selected_shape);
vt := ornt(pose(selected_shape)) * d-vector;
ret.end0 := pnt;
ret.end1 := vt;
end
else
write_msg('surface not identifiable');

if selected_shape <> dfeat_part then
writeln('Warning - Point not on originally selected surface.');
if verify then
begin
    c_init_verify_stuff();
    d_is_manual_select := true;
    point_ok := dverify_point(pnt, vt);
    d_is_manual_select := false;
end
else point_ok := true;

if (do_pnt and point_ok) then
begin
    { pnt and vt in world cs }
    create_ga_inspect_point(pnt, vt, true, true);
    refresh();
end;
if (do_pnt and not(point_ok)) then
begin
    write_msg('Calculated point not on surface. Point not generated.');
    writeln('Calculated point not on surface. Point not generated.');
end;
dselect_surf_points := ret;
end;

FUNCTION: dselect_surf_points

function dselect_surf_points(do_pnt: boolean):seg3dr;
beg
    dselect_surf_points := dselect_surf_points(do_pnt, do_manual_select_verify);
end;

do_tol_verify := true;
tol_verify_default := true;

tols_ok_for_arc := list("anglb, "cirlty, "concen, "cortol, "crnout, "diam, "distb, "flat, "parlel, "perp, "pos, "prof1, "profs, "rad, "strght, "trnout, "width");
do_tol_verify := true;
tol_verify_default := true;

tols_ok_for_circle := list("anglb, "cirlty, "concen, "cortol, "crnout, "diam, "distb, "pos, "rad");

tols_ok_for_cone := list("anglb, "anglr, "cirlty, "concen, "cortol, "diam, "distb, "parlel, "perp, "pos, "rad");

tols_ok_for_cylindr := list("anglb, "anglr, "concen, "cortol, "cylcty, "diam, "distb, "parlel, "perp, "pos, "rad, "trnout");

tols_ok_for_ellips := list("anglb, "cortol, "diam, "distb, "pos, "rad");

tols_ok_for_gcurve := list("prof1");

tols_ok_for_gsurf := list("profs");
tols_ok_for_line  == list("anglb, "anglr, "concen, "cortol, "distb, "parlel, "perp, "pos, "strght); 

tols_ok_for_parpln  == list("distb, "pos, "width); 

tols_ok_for_text  == list("compos); 

tols_ok_for_plane  == list("anglb, "anglr, "cortol, "distb, "flat, "parlel, "perp); 

tols_ok_for_point  == list("cortol, "distb, "pos); 

tols_ok_for_rectngl  == emptylist(id); 

tols_ok_for_sphre  == list("cirlty, "cortol, "diam, "distb, "pos, "rad); 

{ ----------------------------------------------- 
{ FUNCTION: check_tol_feat } 
{ ----------------------------------------------- 
function check_tol_feat(f: d_bar; tol: id): boolean; 
var ret: boolean; 
begin 
    if member(tol, tols_ok_for_bar) then ret := true 
    else ret := false; 
    check_tol_feat := ret; 
end; 

{ ----------------------------------------------- 
{ FUNCTION: check_tol_feat } 
{ ----------------------------------------------- 
function check_tol_feat(f: d_circle; tol: id): boolean; 
var ret: boolean; 
begin 
    if member(tol, tols_ok_for_circle) then ret := true 
    else ret := false; 
    check_tol_feat := ret; 
end; 

{ ----------------------------------------------- 
{ FUNCTION: check_tol_feat } 
{ ----------------------------------------------- 
function check_tol_feat(f: d_cone; tol: id): boolean; 
var ret: boolean; 
begin 
    if member(tol, tols_ok_for_cone) then ret := true 
    else ret := false; 
    check_tol_feat := ret; 
end; 

{ ----------------------------------------------- 
{ FUNCTION: check_tol_feat } 
{ ----------------------------------------------- 
function check_tol_feat(f: d_cylinder; tol: id): boolean; 
var ret: boolean; 
begin 
    if member(tol, tols_ok_for_cylinder) then ret := true 
    else ret := false; 
    check_tol_feat := ret; 
end;
function check_tol_feat(f: d_ellipse; tol: id): boolean;
var ret: boolean;
begin
  if member(tol, tols_ok_for_ellipse) then ret := true
  else ret := false;
  check_tol_feat := ret;
end;

FUNCTION: check_tol_feat

function check_tol_feat(f: d_gcurve; tol: id): boolean;
var ret: boolean;
begin
  if member(tol, tols_ok_for_gcurve) then ret := true
  else ret := false;
  check_tol_feat := ret;
end;

FUNCTION: check_tol_feat

function check_tol_feat(f: d_gsurf; tol: id): boolean;
var ret: boolean;
begin
  if member(tol, tols_ok_for_gsurf) then ret := true
  else ret := false;
  check_tol_feat := ret;
end;

FUNCTION: check_tol_feat

function check_tol_feat(f: d_line; tol: id): boolean;
var ret: boolean;
begin
  if member(tol, tols_ok_for_line) then ret := true
  else ret := false;
  check_tol_feat := ret;
end;

FUNCTION: check_tol_feat

function check_tol_feat(f: d_plane; tol: id): boolean;
var ret: boolean;
begin
  if member(tol, tols_ok_for_plane) then ret := true
  else ret := false;
  check_tol_feat := ret;
end;
function check_tol_feat(f: d_point; tol: id): boolean;
var ret: boolean;
begin
  if member(tol, tols_ok_for_point) then ret := true
  else ret := false;
  check_tol_feat := ret;
end;

function check_tol_feat(f: d_sphere; tol: id): boolean;
var ret: boolean;
begin
  if member(tol, tols_ok_for_sphere) then ret := true
  else ret := false;
  check_tol_feat := ret;
end;

function check_tol_feat(f: d_rectbox; tol: id): boolean;
var ret: boolean;
begin
  if member(tol, tols_ok_for_rectngl) then ret := true
  else ret := false;
  check_tol_feat := ret;
end;

function check_tol_feat(f: d_circle; tol: id): boolean;
var ret: boolean;
begin
  if member(tol, tols_ok_for_circle) then ret := true
  else ret := false;
  check_tol_feat := ret;
end;
{ FUNCTION: check_tol_feat }
{ ----------------------------- }
function check_tol_feat(f: d_circle; tol: id): boolean;
var ret: boolean;
begin
  if member(tol, tols_ok_for_circle) then ret := true
  else ret := false;
  check_tol_feat := ret;
end;

{ FUNCTION: check_tol_feat }
{ ----------------------------- }
function check_tol_feat(f: d_circle; tol: id): boolean;
var ret: boolean;
begin
  if member(tol, tols_ok_for_circle) then ret := true
  else ret := false;
  check_tol_feat := ret;
end;

{ FUNCTION: check_tol_feat }
{ ----------------------------- }
function check_tol_feat(f: d_circle; tol: id): boolean;
var ret: boolean;
begin
  if member(tol, tols_ok_for_circle) then ret := true
  else ret := false;
  check_tol_feat := ret;
end;

{ FUNCTION: check_tol_feat }
{ ----------------------------- }
function check_tol_feat(f: d_circle; tol: id): boolean;
var ret: boolean;
begin
  if member(tol, tols_ok_for_circle) then ret := true
  else ret := false;
  check_tol_feat := ret;
end;

{ FUNCTION: check_tol_feat }
{ ----------------------------- }
function check_tol_feat(f: d_circle; tol: id): boolean;
var ret: boolean;
begin
  if member(tol, tols_ok_for_circle) then ret := true
  else ret := false;
  check_tol_feat := ret;
end;
function check_tol_feat(f: d_circle; tol: id): boolean;
var ret: boolean;
begin
  if member(tol, tols_ok_for_circle) then ret := true
  else ret := false;
  check_tol_feat := ret;
end;

function check_tol_feat(f: d_circle; tol: id): boolean;
var ret: boolean;
begin
  if member(tol, tols_ok_for_circle) then ret := true
  else ret := false;
  check_tol_feat := ret;
end;

function get_tolerance(tol: string): id;
var ret: id;
found_one: boolean;
begin
  ret := empty_id;
  found_one := false;
  for i in program_buffer while not(found_one) do
    if not( null(i to_view dmis_def) ) then
      if (itag(i as_type dmis_def) = "t") then
        if (label_of(i as_type dmis_def) = intern(tol)) then
          begin
            ret := car(minors(i as_type dmis_def)) as_type id;
            found_one := true;
          end;
  get_tolerance := ret;
end;

function check_tol_for_feat(f: feature; tol: id): boolean;
var ret: boolean;
xun: universal;
begin
  xun := applyn("check_tol_feat, list(f as_type universal, tol as_type universal));
  if not(is_fail(xun)) then
    ret := xun as_type boolean
  else ret := tol_verify_default;
  check_tol_for_feat := ret;
end;
function check_tol_for_feat(f, tol: string): boolean;
var the_feat: feature;
  the_tol: id;
  stmt: dmis_stmt;
begin
  the_feat := get_feature(f) as_type feature;
  the_tol := get_tolerance(tol);
  if the_tol <> empty_id then
    begin
      check_tol_for_feat := check_tol_for_feat(the_feat, the_tol)
    end;
  check_tol_for_feat := tol_verify_default;
end;

procedure check_tol_for_bound(tol: string);
var the_tol: id;
begin
  the_tol := get_tolerance(tol);
  if the_tol <> empty_id then
    begin
      if member(the_tol, bound_tols_list) then
        begin
          writeln('Warning - the tolerance ', tol, ' may require bounding.');
          write_msg('Warning - the tolerance ', tol, ' may require bounding.');
        end;
    end;
end;

for i in program_buffer do
  if not( null(i to_view dmis_def) ) then
    if (itag(i as_type dmis_def) = "t") then
      begin
        if do_tol_verify then
          begin
            temp := car(minors(i as_type dmis_def)) as_type id;
            if check_tol_for_feat(the_feat, temp) then
              stuff := append(stuff, list(id_to_string(label_of(i as_type

[FUNCTION: dselect_tol_name ]
[-------------------------- ]
function dselect_tol_name(the_feat: feature): string;
var
  ans: string;
  stuff: list of string;
  temp: id;
begin
  ans := ";
  stuff := emptylist(string);
  for i in program_buffer do
    if not( null(i to_view dmis_def) ) then
      if (itag(i as_type dmis_def) = "t") then
        begin
          if do_tol_verify then
            begin
              temp := car(minors(i as_type dmis_def)) as_type id;
              if check_tol_for_feat(the_feat, temp) then
                stuff := append(stuff, list(id_to_string(label_of(i as_type

end
}
endif
do

ans := get_dmenu_choice('select tolerance ', stuff);
ans := uppercase(ans);
check_tol_for_bound(ans);
if (ans = '!ABORT!') then ans := '---';
dselect_tol_name := ans;
end;

{ ------------------------------ }
{ FUNCTION: dselect_tol_name_list }
{ ------------------------------ }
function dselect_tol_name_list(the_feat: feature): list of string; var
  ans: string;
  ans_list: list of string;
  stuff: list of string;
  temp: id;
begin
  ans := ' ';  
  stuff := empty_list(string);
  for i in program_buffer do
    if not( null(i to_view dmis_def) ) then
      if (itag(i as_type dmis_def) = "t") then
        begin
          if do_tol_verify then
            begin
              temp := car(minors(i as_type dmis_def)) as_type id;
              if check_tol_for_feat(the_feat, temp) then
                stuff := append(stuff, list(id_to_string(label_of(i as_type dmis_def))));
            end;
          else stuff := append(stuff, list(id_to_string(label_of(i as_type dmis_def))));
        end;
      ans := get_dmenu_choice_stay_up('Select Tolerances', stuff);
      ans := uppercase(ans);
      while (ans <> '!ABORT!') do
        begin
            ans_list := append(ans_list, list(ans));
            ans := get_dmenu_choice_stay_up('Select Tolerances', stuff);
            ans := uppercase(ans);
            check_tol_for_bound(ans);
        end;
        menu_return();
dselect_tol_name_list := ans_list;
end;
PROCEDURE: doutput_menu_handler

procedure doutput_menu_handler(msg: string);

var
    dmsg, assoc_tol_name, assoc_feat_name: string;
    tt, ft: id;  
    stmt: output_stmt;
    tags, labs: list of id;
    temp: integer;
    stuff_ok: boolean;
begin
    case msg of
        '!INIT!': begin
            if not(global_params_set) then
                check_for_no_support("output");
                dtol_list := emptylist(string);
                init_output();
                global_params_set := true;
            end;
        'Type': begin
            dmsg := menu_get_val(doutput_menu, 'Type :');
            case dmsg of
                'unmeas': set_msg(doutput_menu, 'Type : ', 'meas');
                'meas': set_msg(doutput_menu, 'Type : ', 'unmeas');
            end;
        end;
        'Feat': begin
            dmsg := dselect_feat_name();
            if dmsg <> '!ABORT!' then
                set_msg(doutput_menu, 'Feat : ', dmsg);
            end;
        end;
        ' Tol ': begin
            dmsg := dselect_tol_name(get_feature(menu_get_val(doutput_menu, 'Feat :')) as_type feature);
            set_msg(doutput_menu, ' Tol : ', dmsg);
            if not(is_associative_tol(dmsg)) then set_msg(doutput_menu, 'Asoc ft:', '---');
        end;
        'Asoc ft': begin
            if is_associative_tol(menu_get_val(doutput_menu, ' Tol :')) then
                begin
                    dmsg := dselect_feat_name();
                    if dmsg = '!ABORT!' then dmsg := '---';
                    set_msg(doutput_menu, 'Asoc ft:', dmsg);
            else write_msg('For use with ANGLB or DISTB tolerances only.'); end;
        end;
        'ADDITIONAL TOLS': begin
            if not(is_associative_tol(menu_get_val(doutput_menu, ' Tol :'))) then
                begin
                    dtol_list := dselect_tol_name_list(get_feature(menu_get_val(doutput_menu, 'Feat :'))
                        as_type feature);
            end
            else write_msg('Cannot use additional tolerances with the tolerance chosen');
        end;
    end;
end;
end;
'Report': get_string(doutput_menu, 'Report:');
'OK': begin
  stmt := mk_default_output();
  tags := emptylist(id); labs := emptylist(id);
  dmsg := menu_get_val(doutput_menu, 'Type:');
  case dmsg of
    'meas': begin
      ft := 'FA;
      tt := 'TA;
    end;
    'unmeas': begin
      ft := 'F;
      tt := 'T;
    end;
  end;
  dmsg := menu_get_val(doutput_menu, 'Feat:');
  assoc_feat_name := dmsg;
  if dmsg <> '---' then begin
    tags := append(tags, list(ft));
    labs := append(labs, list( intern(dmsg) ));
  end;
  dmsg := menu_get_val(doutput_menu, 'Tol:');
  assoc_tol_name := dmsg;
  if dmsg <> '---' then begin
    tags := append(tags, list(tt));
    labs := append(labs, list( intern(dmsg) ));
  end;
  if is_associative_tol(assoc_tol_name) then
    dtol_list := emptylist(string);
  for i in dtol_list do begin
    tags := append(tags, list(tt));
    labs := append(labs, list( intern(i) ));
  end;
  dmsg := menu_get_val(doutput_menu, 'Report:');
  if dmsg <> '---' then begin
    tags := append(tags, list("R"));
    labs := append(labs, list( intern(dmsg) ));
  end;
  stmt.ta_label_list := tags;
  stmt.t_label_list := labs;
  stuff_ok := true;
  if assoc_tol_name <> '---' then
    if is_associative_tol(assoc_tol_name) then
      if assoc_feat_name = '---' then
        stuff_ok := false;
if stuff_ok then
begin
  temp := edit_or_simulate;
  edit_or_simulate := d_modify;
  update_position();
  update_edit_env(stmt);
  edit_or_simulate := temp;
  dtol_list := emptylist(string);
  global_params_set := false;
  menu_return();
end
else write_msg('Need to specify an associated feature (Asoc ft) for selected tolerance');
end;
'
'!ABORT!':: begin
  global_params_set := false;
  dtol_list := emptylist(string);
  menu_return();
end;
end;

{ Verifies if a point is actually on the analytic description of a feature.

Complies with Section 3 of Crada Statement of Work

12.jan.94   jt

edits:
}

{ pnt should be in terms of feature pose, values in cm }  
{ ----------------------------------------------- }
{ FUNCTION: verify_point_on_feat }  
{ ----------------------------------------------- }
function verify_point_on_feat(f: d_circle; pnt: point): boolean;
var ret: boolean;
  rad, ss, rs: real;
begin
  ret := true;
  if not(dabout_zero(pnt.zc)) then ret := false;
  if ret then
    begin
      rad := diam(f) / 2.0;
      ss := pnt.xc * pnt.xc + pnt.yc * pnt.yc;
      rs := rad * rad;
      if not(dabout_zero(rs - ss)) then
        ret := false;
    end;
  verify_point_on_feat := ret;
end;
function verify_point_on_feat(f: d_arc; pnt: point): boolean;
  var ret, rad, ss, rs: real;
begin
  ret := true;
  if not(dabout_zero(pnt.zc)) then ret := false;
  if ret then
    begin
      rad := radius(f);
      ss := pnt.xc * pnt.xc + pnt.yc * pnt.yc;
      rs := rad * rad;
      if not(dabout_zero(rs - ss)) then
        ret := false;
    end;
  verify_point_on_feat := ret;
end;

function verify_point_on_feat(f: d_plane; pnt: point): boolean;
var ret: boolean;
begin
  ret := true;
  if not(dabout_zero(pnt.zc)) then ret := false;
  verify_point_on_feat := ret;
end;

function verify_point_on_feat(f: d_poly; pnt: point): boolean;
var ret: boolean;
begin
  ret := true;
  if not(dabout_zero(pnt.zc)) then ret := false;
  if not(dabout_zero(pnt.xc)) then ret := false;
  if not(dabout_zero(pnt.yc)) then ret := false;
  verify_point_on_feat := ret;
end;

function verify_point_on_feat(f: d_cylinder; pnt: point): boolean;
var ret: boolean;
rad, ss, rs: real;
begin
  ret := true;
  if ret then
    begin
      rad := diam(f) / 2.0;
      ss := pnt.xc * pnt.xc + pnt.yc * pnt.yc;
      ret := true;
      verify_point_on_feat := ret;
    end;
end;
rs := rad * rad;
if not(dabout_zero(rs - ss)) then
    ret := false;
end;
verify_point_on_feat := ret;
end;

{ ------------------------------ }
{ FUNCTION: verify_point_on_feat }
{ ------------------------------ }
function verify_point_on_feat(f: d_sphere; pnt: point): boolean;
var ret: boolean;
    rad, ss, rs, r2s, zs: real;
begin
    ( rad^2 = r2 + z^2 )
    ret := true;
    if ret then
        begin
            rad := diam(f) / 2.0;
            rs := rad * rad;
            ss := pnt.xc * pnt.xc + pnt.yc * pnt.yc;
            zs := pnt.zc * pnt.zc;
            if not(dabout_zero(rs - ss - zs)) then
                ret := false;
            end;
            verify_point_on_feat := ret;
        end;
end;

{ ------------------------------ }
{ FUNCTION: verify_point_on_feat }
{ ------------------------------ }
function verify_point_on_feat(f: d_ellipse; pnt: point): boolean;
var ret: boolean;
a, b2, c, ss: real;
tp: point;
begin
    ret := true;
    if not(dabout_zero(pnt.zc)) then ret := false;
    if ret then
        begin
            tp := mk_point(f.xp, f.yp, f.zp) rel pose(wlkup(f.dreference)) in_frame pose(f);
            c := abs(tp.xc);
            a := diam(f) / 2.0;
            b2 := a * a - c * c;
            ss := ((pnt.xc * pnt.xc) / (a*a)) + ((pnt.yc * pnt.yc)/ b2);
            if 1.001 < ss then
                ret := false;
            end;
            verify_point_on_feat := ret;
        end;
FUNCTION: verify_point_on_feat

function verify_point_on_feat(f: d_cone; pnt: point): boolean;
var ret: boolean;
angl, rad, ss, rs: real;
begin
  ret := true;
  if ret then
    begin
      angl := ang(f) / 2.0 * dtor;
      rad := pnt.zc * tan(angl);
      ss := pnt.xc * pnt.xc + pnt.yc * pnt.yc;
      rs := rad * rad;
      if not(dabout_zero(rs - ss)) then
        ret := false;
    end;
  verify_point_on_feat := ret;
end;

FUNCTION: verify_feat_pnts

function verify_feat_pnts(f: feature): boolean;
var is_cool: boolean;
cs: frame;
pnt: point;
xun: universal;
um_bad: integer;
begin
  num_bad := 0;

  xun := applyn("verify_point_on_feat, list(f as-type universal, null_point as-type universal)");
  if not(is_fail(xun)) then  { test to see if a variant exists for a feature type }
    begin
      cs := pose(f);
      for i in children(f) do
        if (is_probe_step(i) and not(is_goto(i))) then
          begin
            is_cool := true;
            pnt := null_point rel pose(i) in_frame cs;
            xun := applyn("verify_point_on_feat, list(f as-type universal, pnt as-type universal)");
            if not(is_fail(xun)) then
              is_cool := xun as-type boolean
            else is_cool := true;  { assume its OK if we don’t know better }
          end;
    end;
  end;
end;
else writeln('All points OK for ', name(f));
end
else writeln('Sorry, no test available for ', name(f));

verify_feat_pnts := (num_bad = 0);
end;

[ ----------------------------- ]
[ PROCEDURE: verify_all_feat_pnts ]
[ ----------------------------- ]
procedure verify_all_feat_pnts();
var ans: boolean;
begin
  for i in children(d_part) do
    if is_feature(i) then
      ans := verify_feat_pnts(i as-type feature);
  end;

[ ------------ menus ------------ ]
verify_pnts_menu == mk_imenu('VERIFY FEATURE PNTS', ulist(
                     'ONE FEATURE',
                     'ALL'),
                     "verify_pnts_menu_handler");

[ ----------------------------- ]
[ PROCEDURE: verify_pnts_menu_handler ]
[ ----------------------------- ]
procedure verify_pnts_menu_handler(msg: string);
var dmsg: string;
begin
  case msg of
    'ONE FEATURE': begin
      dmsg := dselect_feat_name();
      if dmsg <> '!ABORT!' then
        verify_feat_pnts(get_feature(dmsg) as_type feature);
      end;
    'ALL': verify_all_feat_pnts();
    '!ABORT!': begin
      menu_return();
    end;
  end;
end;

add_application_utility_menu(verify_pnts_menu);

{ --------- dont add bad words to meas block ------- }
do_meas_check := true;
{ ------------------------------- }
{ FUNCTION: is_valid_meas_member }
{ ------------------------------- }
function is_valid_meas_member(ds: dmis_stmt): boolean;
var ret: boolean;
begin
  ret := false;
  if not(nul(ds to_view aclrat_stmt)) then
    ret := true;
  else if ( not(nul(ds to_view goto_stmt)) or not(nul(ds to_view finpos_stmt))) then
    ret := true;
  else if ( not(nul(ds to_view ptmeas_stmt)) or not(nul(ds to_view rotab_stmt)) ) then
    ret := true;
  else if ( not(nul(ds to_view snslt_stmt)) or not(nul(ds to_view from_stmt)) ) then
    ret := true;
  else if ( not(nul(ds to_view fedrat_stmt)) or not(nul(ds to_view rotset_stmt)) ) then
    ret := true;
  else if ( not(nul(ds to_view text_stmt)) or not(nul(ds to_view hll_stmt)) ) then
    ret := true;
  else if ( not(nul(ds to_view gchome_stmt)) or not(nul(ds to_view save_stmt)) ) then
    ret := true;
  else if ( not(nul(ds to_view rapid_stmt)) or not(nul(ds to_view wkplan_stmt)) ) then
    ret := true;
  else if ( not(nul(ds to_view gotarg_stmt)) or not(nul(ds to_view snset_stmt)) ) then
    ret := true;
  else if ( not(nul(ds to_view comment_stmt)) or not(nul(ds to_view endmes_stmt)) ) then
    ret := true;
  else if ( not(nul(ds to_view recall_stmt)) or not(nul(ds to_view endgo_stmt)) ) then
    ret := true;
  is_valid_meas_member := ret;
end;

{ ------------------------------- }
{ PROCEDURE: check_valid_insert }
{ ------------------------------- }
procedure check_valid_insert(ds: dmis_stmt);
var cpos: integer;
begin
  cpos := where_is_cursor();
  if ((cpos = 2) or (cpos = 1)) then
    if not(is_valid_meas_member(ds)) then
      leave_meas_prep();
  end;

  { I think, that no matter what the input mechanism, all new statements must go through this, so this is what we'll use. }
procedure insert_dmis_stmt(d: dmis_stmt);
begin
  if insp_debug then writeln('in insert dmis stmt ',d.syntax);

  if d_add_to_end then
    add_dmis_stmt(d)
  else
    begin
      if (do_meas_check and not(d_loading)) then check_valid_insert(d);
      pos := dget_selected_line() + 1;
      dmis_insert(pos - 1, d);  
      { inserted AFTER arg }
      move_cursor_to(dmis_text, pos-1, 0);
      dwrite_to_window(d.syntax);
      if not(null(d_delete_rec.statement)) then  
        { inc if new statements are added }
        if (pos -1) : d_delete_rec.line_num then
          d_delete_rec.line_num := d_delete_rec.line_num + 1;
          dlast_position := pos;
      end;
      if not(d_loading) then
        any_dmis_change := true;
    end;
end;

{ ----- add verify global tolerances to utilities menu ------- }
dgen_pt_settings_menu := mk_imenu('POINT GENERATION', ulist(  
  list('CLEAR PT', 'INFO'),  
  list('USE OFST', 'INFO'),  
  list('OFST VAL', 'INFO'),  
  list('CLEAR VECT', 'INFO'),  
  list('Verify Dist', 'INFO'),  
  list('Verify Ang', 'INFO'),  
  'DEFINE CLEARANCE VECT',  
  list('GOTO_S', 'INFO')), "dgen_pt_settings_menu_handler");

{ PROCEDURE: init_gen_pt_set }
procedure init_gen_pt_set();
begin
  set_msg_yn(dgen_pt_settings_menu, 'CLEAR PT', clearance_points);
  set_msg_yn(dgen_pt_settings_menu, 'USE OFST', offset_ptmeas);
  set_msg(dgen_pt_settings_menu, 'OFST VAL', d_offset);
  set_msg_yn(dgen_pt_settings_menu, 'GOTO_S', prima_straight);
  set_msg(dgen_pt_settings_menu, 'CLEAR VECT', d_use_clear);
  set_msg(dgen_pt_settings_menu, 'Verify Dist', d_verify_point_tol / current_unit);
  set_msg(dgen_pt_settings_menu, 'Verify Ang', d_verify_normal_tol);
end;
{ ---------------
PROCEDURE: dgen_pt_settings_menu_handler
{ ---------------
procedure dgen_pt_settings_menu_handler(msg: string);
var
dmsg: string;
vI: real;
nd: integer;
begin
case msg of
'!INIT!': begin
    init_gen_pt_set();
end;
'CLEAR PT': begin
    dmsg := menu_get_val(dgen_pt_settings_menu, 'CLEAR PT');
    if ((dmsg = 'yes') or (dmsg = 'YES')) then
        begin
            set_msg(dgen_pt_settings_menu, 'CLEAR PT', 'no');
            clearance_points := false;
        end
    else
        begin
            set_msg(dgen_pt_settings_menu, 'CLEAR PT', 'yes');
            clearance_points := true;
        end;
end;
'USE OFST': begin
    dmsg := menu_get_val(dgen_pt_settings_menu, 'USE OFST');
    if ((dmsg = 'yes') or (dmsg = 'YES')) then
        begin
            set_msg(dgen_pt_settings_menu, 'USE OFST', 'no');
            offset_ptmeas := false;
        end
    else
        begin
            set_msg(dgen_pt_settings_menu, 'USE OFST', 'yes');
            offset_ptmeas := true;
        end;
end;
'OFST VAL': begin
    get_real(dgen_pt_settings_menu, 'OFST VAL');
    d_offset := menu_get_real(dgen_pt_settings_menu, 'OFST VAL');
end;
'Verify Dist': begin
    get_real(dgen_pt_settings_menu, 'Verify Dist');
    d_verify_point_tol := menu_get_real(dgen_pt_settings_menu, 'Verify Dist') * current_unit;
end;
'verify Ang': begin
    get_real(dgen_pt_settings_menu, 'Verify Ang');
    d_verify_normal_tol := menu_get_real(dgen_pt_settings_menu, 'Verify Ang');
end;
'GOTO_S': begin
    dmsg := menu_get_val(dgen_pt_settings_menu, 'GOTO_S');
    if ((dmsg = 'yes') or (dmsg = 'YES')) then
        begin
            set_msg(dgen_pt_settings_menu, 'GOTO_S', 'no');
            prima_straight := false;
        end;
else
  begin
    set_msg(dgen_pt_settings_menu, 'GOTO_S', 'yes');
    prima_straight := true;
  end;
end;

'CLEAR VECT': begin
  dmsg := menu_get_val(dgen_pt_settings_menu, 'CLEAR VECT');
  if ((dmsg = 'yes') or (dmsg = 'YES')) then
    begin
      set_msg(dgen_pt_settings_menu, 'CLEAR VECT', 'no');
      d_use_clear := false;
    end
  else
    begin
      set_msg(dgen_pt_settings_menu, 'CLEAR VECT', 'yes');
      d_use_clear := true;
    end;
end;

'DEFINE CLEARANCE VECT': call(dclear_vect_menu);

'! ABORT!': menu_return();
end

{ SIL Function name: create_cur_cs_point

Purpose: From Jerry's original code.

Written by: Jerry Turner

generates inspection points associated with inspecting a feature.
* Approach and retract points will not be generated if clearance_points = false
  or if you call the function with gen_approach and/or gen_retract = false.
* Point - should be the point on the surface- no compensation for prcomp needed.
* Point and normal in current coordinate system coords!
* Feature should already be defined and be the current feature.
* Radius compensation should be on.
}

procedure create_cur_cs_point(pt, nm: point; gen_approach, gen_retract: boolean);

  var pt2: point;
  dist: real; [ the distance from the pt meas point to the clearance point ]
  prad: real; [ the given radius of the probe tip ]
  do_it: boolean;

begin
  do_it := true;
  if check_for_holes then
    do_it := not(over-hole(pt rel pose(cur_frame), ornt(pose(cur_frame)) * nm));
  if do_it then
    begin
      if offset_ptmeas then [ offset for sheet metal with no thickness ]
        pt := pt + (d_offset * nm);
      if length(cur_probe.tiplist) > 0 then
        prad := (diam(select(cur_probe.tipList, cur_probe.activeTip -1))) / 2.0
      else
        prad := 0.5;
      { approach point }
      if (gen_approach and clearance_points) then
begin
  dist := (cur_modal.approach * current_unit) + prad;
  pt2 := pt + (dist * nm);  \{gs always inner -this works out cool for planes etc.\}
  make_new_point(pt2, nm, wlkup(cur_feat.dreference) as_type coord_sys, cur_feat, true);
end;

\{ inspection point \}
  make_new_point(pt, nm, wlkup(cur_feat.dreference) as_type coord_sys, cur_feat, false);

\{ retract point \}
  if (gen_retract and clearance_points) then
    begin
      dist := (cur_modal.retract * current_unit) + prad;
      pt2 := pt + (dist * nm);
      make_new_point(pt2, nm, wlkup(cur_feat.dreference) as_type coord_sys, cur_feat, true);
    end;

  refresh();  \{ this refresh statement is optional - it repaints the graphics screen. Remove to speed up. \}
end;
end;

\{ define the lrecord returned by the smallest_dst function \}
  type dsmallest contains indexes to where smallest distance found
  dsmallest.uval -  u index (2d) of smallest distance point
  dsmallest.vval -  v index (2d) of smallest distance point
  dsmallest.kval -  k index (1d) of smallest distance point

\} type
  dsmallest = lrecord
  .uval, .vval, .kval: integer
end;

\{ SIL Function name: smallest_dst  \(version using lrecord)\}

\Purpose: Given an array (36) of distances, find the smallest and return it and the u,v where it was found.\n
\Written by: Richard Riker\n  date: 2 - 14 - 94\n  for: MMES/SILMA CRADA work\n
\Input data: \n  node_dist array  \(36\) array of real distances\n
\Output data: \n  sdist real \((\text{smallest})\) currently in the array\n  dsmallest LRecord containing:
    .uval integer \(u\)-value where smallest distance found
    .vval integer \(v\)-value where smallest distance found\n
\Revision History: \}
function smallest_dst(node_dist:array of real; s_dst:dsmallest): real;
    var i, j, k: integer;
    tmpdist: real;
    begin
        tmpdist := 99999.;
        for i := 0 to 5 do begin
            for j := 0 to 5 do begin
                k := ( j * 6 ) + i;
                if node_dist[k] < tmpdist then begin
                    tmpdist := node_dist[k];
                    s_dst.uval := i;
                    s_dst.vval := j;
                    s_dst.kval := k;
                end;
            end;
        end;
        smallest_dst := tmpdist;
    end;

{ define the lrecord returned by the five_x_5 function

    type dcorners contains real u,v values of surface area used
    dcorners.umn - Minimum u value
    dcorners.umx - Maximum u value
    dcorners.vmn - Minimum v value
    dcorners.vmx - Maximum v value
    dcorners.nearp - distance from (closest) node to given point
    dcorners.unp - U value of closest node to given point
    dcorners.vnp - V value of closest node to given point
}
} type
    dcorners = lrecord
        umn, umx, vmn, vmx, nearp, unp, vnp: real
    end;

{ SIL Function name: five_x_5

Purpose: Given point1 and point2 (U,V), divide the rectangle defined by the two points into 25 smaller rectangles of equal size. Check all 36 "nodes" for distance (in xyz space) from the given point (being verified) and use the U,V coordinates of the five closest points to define a new, much smaller area of U,V space where the given point will lie.

Written by: Richard Riker
date: 2-14-94
for: MMPES/SILMA CRADA work

Input data:
    gpt          point          given point
    surf         shape         name of nearest surface
    surf_corners lrecord containing
        umn    real    minimum U value
        umx    real    maximum U value
        vmn    real    minimum V value
        vmx    real    maximum V value
Output data:
boolean TRUE or FALSE
where TRUE means successful reduction occurred
surf_corners record containing
umn real minimum U value
umx real maximum U value
vmm real minimum V value
vmmx real maximum V value

Revision History:
}
function five_x_5(surf_corners:dcorners; gpt:point; surf:shape): boolean;

var u, v, u_inc, v_inc, umin, umax, vmin, vmax, o_umn, o_umx, o_vmn, o_vmx, tmp:real;
i, j, k: integer;
pt: point;
ret: boolean;

s_dst: dsmsallest;
node_dist, least7, nodesu, nodesv: array of real;
least7u, least7v: array of integer;

begin

least7 := array_create(real,5); { closest 7 distances }
node_dist := array_create(real,35); { 36 node distances }
nodesu := array_create(real,5); { U values }
nodesv := array_create(real,5); { V values }
least7u := array_create(integer,6); { closest 7 U indexes }
least7v := array_create(integer,6); { closest 7 V indexes }
s_dst := mk_dsmsallest(0,0,0);
ret:=false;

o_umn := surf_corners.umn; { Keep the original input corner points }
o_umx := surf_corners.umx;
o_vmn := surf_corners.vmn;
o_vmx := surf_corners.vmx;

umin := surf_corners.umn; { set up the limits }
umax := surf_corners.umx;
vmin := surf_corners.vmn;
vmax := surf_corners.vmx;

{ only do this as long as some delta exists between corners 10 iterations reduces area of U,V by a factor of more than 6 to the 10th power }
if ( (umax-umin) > 0 ) AND ( (vmax-vmin) > 0 ) then begin
u_inc:=(umax-umin) / 5.0; { calculate the increment in u and v }
v_inc:=(vmax-vmin) / 5.0;

{ loop checking the xyz distance from nodes to our given point }
for j:=0 to 5 do begin
v:=vmin + (j*v_inc); { calculate next v coord }
nodesv[j]:=v;
for i:=0 to 5 do begin
u:=umin + (i*u_inc); { calculate next u coord }
nodesu[i]:=u;
pt:=seg_pose(surf) *apply("eval_at", surf as_type obj, u, v)
as_type point;
k := ( j * 6 ) + i;
node_dist[k]:=length ( pt - gpt );
now, check for the closest 7 distances (use only 5) and use their u,v
coords to determine the new u,v area in which to continue the search for
the portion of the surface "close" to the given point.

for i := 0 to 6 do begin
    least7[i] := 99999.;
    least7u[i] := -1;
    least7v[i] := -1;
end;

nodesu[n] contains REAL U values, nodesv[n] contains REAL V values and
node_dist[n] contains REAL distances between surf and given point now,
find the closest 7 points . . but only use 5 for now must store the index
to U and V for each of the distances determined to be one of the seven
nearest points

for K := 0 to 6 do begin
    least7[k] := smallest_dist(node_dist, s_dst);
    least7u[k] := s_dst.uval;  [ save the index to U value ]
    least7v[k] := s_dst.vval;  [ save the index to V value ]
    i := s_dst.kval;
    node_dist[i] := 99999.;  [ cover up the smallest found ]
end;

found the closest 7 nodes to our given point, recalculate the U,V
boundaries for next pass ... use only the closest 5 for now,
may have to change to 6 or 7 later

i := least7u[0];
umax := nodesu[i];  [ initialize them all to smallest ]
umin := nodesu[i];  [ distance point U and V ]
j := least7v[0];
vmax := nodesv[j];
vmin := nodesv[j];

{ update the record with closest information }
surf_corners.nearp := least7[0];  [ store closest node distance ]
surf_corners.unp := umax;          [ store U of closest node ]
surf_corners.vnp := vmax;          [ store V of closest node ]

for K := 1 to 4 do begin
    i := least7u[k];     [ index to this U value ]
    tmp := nodesu[i];
    if tmp > umax then umax := tmp;
    if tmp < umin then umin := tmp;
    j := least7v[k];     [ index to this V value ]
    tmp := nodesv[j];
    if tmp > vmax then vmax := tmp;
    if tmp < vmin then vmin := tmp;
end;
{ added 3/15/94 ... open up the window in cases where the area seems to be a straight line }

if (umax + (u_inc/2)) < o_umx then begin
    umax := umax + ( u_inc / 2 );
end;
if (umin - (u_inc/2)) > o_umn then begin
    umin := umin - ( u_inc / 2 );
end;
if (vmax + (v_inc/2)) < o_vmx then begin
    vmax := vmax + ( v_inc/2);
end;
if (vmin - (v_inc/2)) > o_vmn then begin
    vmin := vmin - (v_inc/2);
end;

{ Now reset the corner points of our section of U,V space }

surf_corners.umn := umin;
surf_corners.vmx := vmax;
surf_corners.umn := umax;
surf_corners.vmn := vmin;

end;

{ return true if either of the input array corner points changed }

if (((o_umn <> umin) OR (o_umx <> umax)) OR
    (((o_vmn <> vmin) OR (o_vmx <> vmx))) then
    ret := true
else
    ret := false;

five_x_S := ret;
end;

{ SIL Function name: angl_between_vecs }

Purpose:  To calculate the angle between two i,j,k unit vectors in radians.

Written by: Richard Riker
date: 02/23/94
for: HMES/SILMA CRADA work

Input data:
    vec1 point  vector one
    vec2 point  vector two

Output data:
    angl real  angl between them in radians

Revision History:
function angl_between_vecs(vecl, vec2: point): rangle;
var dist: real;
    ang: rangle;

begin
  begin
    { Method: find the distance between the two vector (tips), use 1/2 that 
      distance with arcsin function to get 1/2 of the angle between the vectors. }
    d = sqrt((i1-i2)**2 + (j1-j2)**2 + (k1-k2)**2) ;
    1/2d = 1/2d and }
    ang = 2 * (arcsin(1/2d / 1))

  { distance between vector tips ... vectors are type POINT }
  dist := length(vecl - vec2);

  { get the absolute value of 2 times the arcsin of 1/2 dist in radians }
  ang := 2.0 * asin(dist / 2.0);

  { NOTE: Probably should keep this in RADIANS because, if the value were pi 
     radians, and it was converted to degrees, it might be considered 0 degrees 
     even though the vectors were reversed in direction and could result in 
     erroneous acceptance of a bad vector. }
  angl_between_vecs := ang;
end;

max_verify_iterations := 18;

{ SIL Function name: dverify_point

Purpose: Given a point and vector, verify that the point is within 
a global tolerance of the surface of the nearest surface. 
Also verify that the vector is within a global tolerance 
of the normal vector of the point on the surface.

Written by: Richard Riker
    date: 2 - 23 - 94
    for: MMES/SILMA CRADA work

Input data:
    pt  point  the given point
    norm point  the vector of the point (normalized)

Output data:
    boolean TRUE or FALSE
    where TRUE means the point is within tolerance and FALSE means 
    that the point should be rejected.

Revision History:
function dverify_point(pt, norm: point): boolean;

var the_surf: shape;
bounds: rbnds2d;
umin, umax, vmin, vmax, u, v, off_surf, tmp: real;
ang1_off: rangle;
surf_corners: dcorners;
ptsurf, norm_surf: point;

tempd: dir3dr;

iterations: integer;
ret, notfound: boolean;

begin

surf_corners := mk_dcorners(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
ret := false;
iterations := 0;
the_surf := d_verify_find_surface(pt, norm); { find nearest surface }
highlight(the_surf); reentrant();

writeln('Surface found = ', the_surf);
unhighlight(the_surf);

if the_surf <> null_shape then begin

{ Found a surface, now see if the point is within the tolerance of the surface using iterations of function five_x_5. }
bounds := dom(the_surf as type psurface); { get surface domain }
umin := lb(xbnds(bounds)); { find the min and max limits }
vmin := lb(ybnds(bounds)); { find the min and max limits }
umax := ub(xbnds(bounds)); { find the min and max limits }
vmax := ub(ybnds(bounds)); { find the min and max limits }

if umin > umax then begin
  tmp := umin;
  umin := umax;
  umax := tmp;
end;
if vmin > vmax then begin
  tmp := vmin;
  vmin := vmax;
  vmax := tmp;
end;
surf_corners.umn := umin; { set up original corners in Lrecord }
surf_corners.umx := umax;
surf_corners.vmn := vmin;
surf_corners.vmx := vmax;

writeln('orig.umin,umax ','umin',' ','umax',' vmin,vmax',' vmin',' ','vmax');
notfound := true;
ret := true;

while (iterations < max_verify_iterations) AND (ret) do begin
  writeln(' iteration number ', iterations);
end;
end;
iterations := iterations + 1;
ret:=five_x_5 (surf_corners,pt,th[ee_surf);
{
    see if the closest point is within tolerance and leave this while loop
    if it is, otherwise continue
}
if surf_corners.nearp < d_verify_point_tol then begin
    notfound:= false;
u := surf_corners.unp;
v := surf_corners.vnp;
ret := false;
if inspection_info then writeln('Point found on/near surface after iterations = ');
end;
umin := surf_corners.umm;  { update corners }
umax := surf_corners.umx;
umin := surf_corners.vmn;
vmax := surf_corners.vmx;
end;
if (notfound) then begin
    Completed x iterations without finding a node within tolerance.
    See what the distance is from the center of the final U,V rectangle.
    umin,vmax
        +------------------+
    midpoint ----  .  .  *
        +------------------+
        .  umax,vmin
}
u := umin + ( (umax - umin) / 2.0 );
v := vmin + ( (vmax - vmin) / 2.0 );  { mid-point of rectangle }
end;
ptsurf := seg_pose(the_surf) *apply("eval_at, the_surf as_type obj, 
    u, v) as_type point;
off_surf := length ( ptsurf - pt );
if inspection_info then writeln('distance between is ',off_surf);
{ Get the normal vector at surface point selected by u,v coordinates. }
tempd:= apply("normal_at, the_surf as_type obj, u, v) as_type dir3dr;
norm_surf := (ornt(seg_pose(the_surf)) * tempd) as_type point;
{ compare distance to the global tolerance here .. if out of tolerance, 
    set ret to false .. if in tolerance ... go on and test normal }
if off_surf <= d_verify_point_tol then
    ret:=true
else begin
    ret:=false;
    writeln('Rejected.Point is off surface by ',off_surf);
if ret then begin
  angl_off := angl_between_vecs(norm,norm_surf);
  if (angl_off as_type real) <= a_verify_normal_tol then
    ret := true
  else begin
    ret := false;
    writeln('Rejected, Normal is off ',angl_off,' radians from surface normal');
  end;
end; { JT addition: couldn't find a surface with in verify_col_det_size }
else writeln('Rejected, Point is off surface further than ', verify_col_det_size / current_unit);

d_verify_point := ret;
end;

{ SIL Procedure name: dimport_points

Purpose: From Jerry's original code.

Written by: Jerry Turner

this can:
- import points to be ptmeas or feat/point,
- do this relative to the part_cs or the current cs
- and can verify if points are "on" a surface
  (should use existing stuff instead)
}

procedure dimport_points(fname, i_type: string; part_cs, verify: boolean);
var
  x,y,z,i,j,k, cscale: real;
  the_file: text;
  pt, nm, msg_pt: point;
  fname: string;
  point_ok: boolean;
  total_pnts, rejected_pnts: integer;
begin
  if verify then
begin
    if is_object('detector_1') then remove('detector_1');
    cscale := d_coll_fellow_size; { in cm }
    d_remake_coll_fellow(verify_col_det_size / current_unit); { make him a little bigger for this }
    if not(in_world(d_coll_fellow)) then add(d_coll_fellow,'detector_1');
    end;
    total_pnts := 0; rejected_pnts := 0;
    the_file := mk_file(lib_link * '/target_code/' * i_fname);
    open(the_file,'input');
    write_msg('Importing point file ' * fname);
while(not(eof(the_file))) do
  begin
    readln(the_file, x,y,z,i,j,k);
    pt := mk_point(x,y,z);

    if inspection_info then writeln('*** looking at point ', pt*current_unit);

    msg_pnt := pt;
    total_pnts := total_pnts + 1;
    { make sure normal is a unit vector with normalize() }
    nm := normalize(mk_point(i,j,k)) as_type point;

    if not(part_cs) then { in terms of cur coord sys }
      begin
        if verify then
          point_ok := dverify_point((pt * current_unit) rel pose(cur_frame),
           xform_vect(nm, pose(cur_frame), pose(world)) )
        else point_ok := true;
        if point_ok then
          begin
            if itype = 'insp_pnt' then
              begin
                pt := pt * current_unit; { convert to cm }
                create_cur_point(pt, nm, true, true);
              end
            else
              begin
                fname := get_unique_pnt_name('pnt');
                make_point_feature(fname,'cart',list(x, y, z, nm.xc, nm.yc, nm.zc));
                mk_point_points(cur_names as_type d_point);
              end;
          end { point ok }
        else rejected_pnts := rejected_pnts + 1;
      end { in terms of cad model cs }
    begin
      pt := pt * current_unit; { convert to cm - before transforming }
      nm := ornt(seg_pose(wlup(dcad_part))) * nm; { convert to global coords }
      pt := pt rel seg_pose(wlup(dcad_part));
      if verify then
        point_ok := dverify_point(pt, nm)
      else point_ok := true;
      if point_ok then
        begin
          if itype = 'insp_pnt' then
            create_gs_inspect_point(pt, nm, true, true)
          else { make point feature }
            begin
              nm := invert(ornt(pose(cur_frame))) * nm;
              pt := pt in_frame pose(cur_frame);
              pt := pt / current_unit; { convert to current units }
              fname := get_unique_feat_name('pnt');
              make_point_feature(fname,'cart',list(pt.xc, pt.yc, pt.zc, nm.xc, nm.yc,
                   nm.zc));
              mk_point_points(cur_names as_type d_point);
            end;
        end { point ok }
      else rejected_pnts := rejected_pnts + 1;
    end;
close(the_file);
write_msg('Done with file '*fname);
if verify then
  begin
    if is_object('detector_1') then remove('detector_1');
d_make_coll_fellow(cscale / current_unit);

{ edit ... r.b. riker 3/8/94 to make the message more appropriate }

if rejected_pnts > 0 then begin
  writeln('Warning: ',rejected_pnts,' Points not on surface out of ',
    total_pnts, ' total points.');
  end
else
  writeln(' All ',total_pnts,' points found to be within tolerance');
end;
end;
Distribution:

Lockheed Martin Energy Systems, Inc./DOE-OR
Marshall B. Biddix, MS-8202, 9112
Brian B. Bovee, MS-8242, 701SCA
Erwin Epple, MS-8084, 9203
A. K. Lee/DOE-OSTI, MS-8175, 9983-30 (2 copies)
Robert M. Poteat, DOE Office of Patent Counsel, FOB

Silma, Inc.
Jerry Turner, 1601 Saratoga-Sunnyvale Road, Cupertino, CA 95014 (5 copies)