

## **7 Appendix**

### **7.1 Flexible Feeder**

**7.1.1 Underlit Conveyor Construction Procedure**

**7.1.2 Feeder Test Data Windowing Algorithm**

**7.1.3 Averaging Window Script**

**7.1.4 Feeder Distribution Plots**

### **7.2 Tool Offset Calculator Examples**

**7.2.1 Picking a Part from a Flexible Feeder**

**7.2.2 Placing a Part on a Fixture on a Modular Table**

**7.2.3 Placing a Part on a Fixture at an Angle**

### **7.3 Programs**

**7.3.1 V+ QC Calibration Program**

**7.3.2 Tool Offset Calculator**

## **7.1 Flexible Feeder**

### **7.1.1 Underlit Conveyor Construction Procedure**

First, the entire conveyor must be disassembled. This is a simple procedure given the modular construction employed by Dorner. The next step is to make the fluorescent light adapter plate. The top plate of the conveyor must then be shortened by  $3\frac{3}{8}$  inches. New mounting holes must be drilled into the end of the shortened plate ( $\frac{1}{4}$  of an inch from the end,  $\frac{5}{16}$  of an inch from each side). The side profile must then be cut for a length of  $3\frac{1}{2}$  inches to a depth of  $\frac{7}{16}$  of an inch (leaving the side profile in place). They must also have tapped holes to hold the diffuser plate as well as cross holes for the fluorescent mounting plate. The upper T-nut for the twin-roller tail must then be trimmed to a length of 1 inch. The bracket for the end rollers must also have a section removed for mounting bracket clearance. Finally the diffuser plates must be machined. The parts are then ready to be assembled.

### 7.1.2 Feeder Test Data Windowing Algorithm

Data collected during system operation was based on time per part. For further analysis, it was desired to have the data in a part per time format. To accomplish the conversion, a Matlab script (below) was written which read in the data, performed the conversion, then wrote out the new data. Input parameters included the name of the Matlab array (multi-dimensional) which contained the data, the size of the window (in seconds) over which the data was averaged, the step size factor (explained below), and the index of the column of the array which contained the timestamp for the test data.

The algorithm functioned as follows. Refer to Figure 7-1 for point names listed below. First, the amount of time by which the averaging window was advanced between subsequent data (output) points was determined. This was accomplished by computing the minimum distance between any two data points in the original data then multiplying this number by the step size factor (generally set between 0.5 and 0.75). The step-size factor was used so that no data was skipped over due to an overly coarse step-size. Next, the ending time and the initial value of the end of leading data point in the window, point  $j$ , was determined. A loop was then setup to step the window over the data. After each step, data points added to or removed from the loop were determined, the number of points in the loop was computed (index  $j$  - index  $i$ ), and the averages were computed.

As an example, consider determining the system throughput at the instant shown in Figure 7-1. The number of total parts in the window is  $j - i$ , the size of the window is  $W_h - W_t$ , therefore the average system throughput would be  $(j - i) / (W_h - W_t)$ .

The resulting data point would be placed at the center time of the window,  $(W_h - W_t) / 2$ .

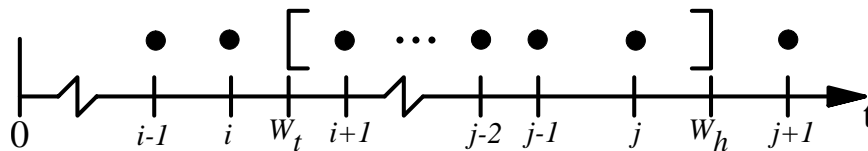


Figure 7-1: Averaging Window Moving Through the Data

### 7.1.3 Averaging Window Script

```

function [output] = window_average(data_in,win_size,step_size_factor,offset)
% window_average will compute the average of values of data which fall
% within the width of the window. The data can be at non-uniform
% intervals. The function operates by determining the smallest distance
% between any two data points and then using some fractional amount of that
% distance to determine the amount to move the window at each step.
%
% Usage: output = window_average(data_in,win_size,step_size_factor,offset)
% where output is the resulting averaged data
% data_in is the data to average
% win_size is the size of the window in seconds
% step_size_factor is the factor which is multiplied to the minimum
% distance between any two data points to determine the
% size of the step the window moves at each averaging point
% offset is the offset into the input array which points to the
% column containing the overall time data (1 or 2)
%
%
%
%
% Set up some variables
half_win_size = win_size / 2.0;
top = 1;
bottom = 1;
counter = 1;
offset=offset+1;

% Determine the minimum distance between any two data points
delta=min(data_in(2:length(data_in(:,1)),offset)-...
    data_in(1:length(data_in(:,1))-1,offset));

% Set up the bounds of the loop
step_size = delta*step_size_factor;
end_value = data_in(length(data_in(:,1)),offset)-win_size;

% Determine the initial value of top
while data_in(top,offset) <= win_size
    top = top + 1;
end
top = top - 1;

% Move the window over the data
for tail = 0:step_size:end_value

    head = tail+win_size;

    if data_in(bottom+1,offset) < tail
        bottom=bottom+1;
    end

    if data_in(top+1,offset) <= head
        top = top+1;
    end

    % Number of points in the window
    num_points = top - bottom;

    % Bottom + 1
    bottom_p = bottom+1;

    % Compute the averages for this window
    output(counter,1) = (tail+half_win_size)/60.0;
    output(counter,2) = (num_points) / (win_size / 60);
    output(counter,3) = (num_points)/(sum(data_in(bottom_p:top,offset+2))/60);
    output(counter,4) = (num_points)/(sum(data_in(bottom_p:top,offset+3))/60);
    output(counter,5) = (num_points)/(sum(data_in(bottom_p:top,offset+4))/60);

    % Update the counter
    counter = counter + 1;

end

% All finished
return

```

7.1.4 Feeder Distribution Plots

7.1.4.1 Overall System Throughput

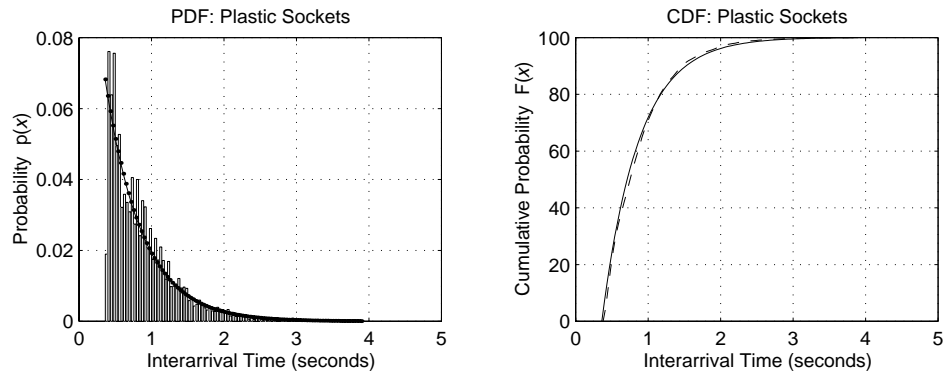


Figure 7-2: PDF and CDF of the Plastic Disks

7.1.4.2 Conveyor System Throughput

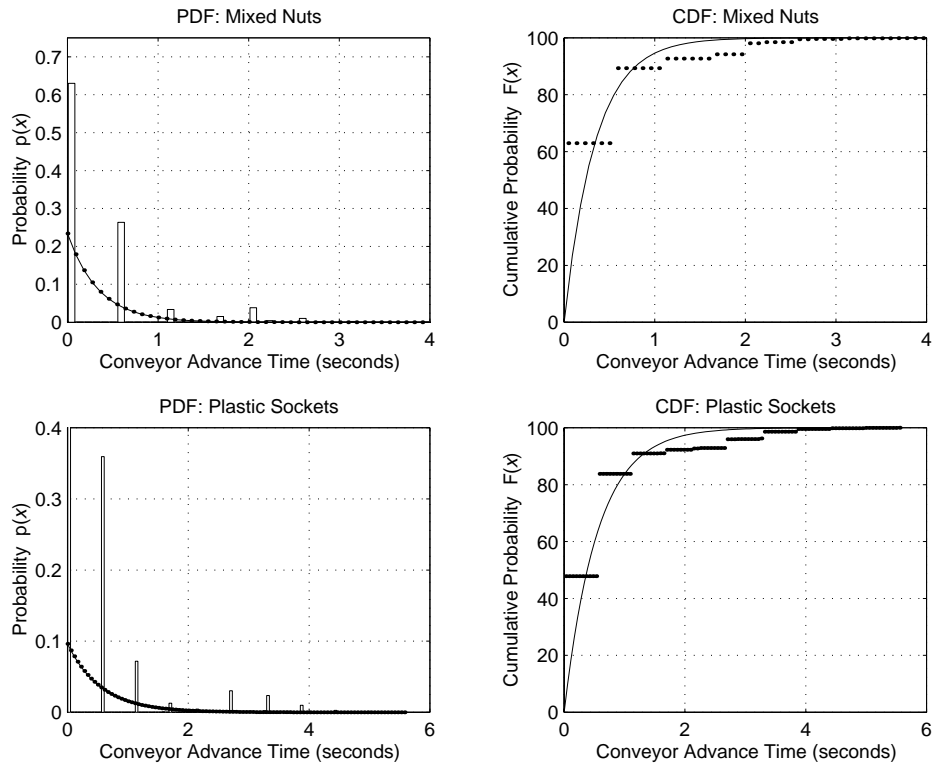


Figure 7-3: PDF and CDF for Various Parts

7.1.4.3 *Vision System Throughput*

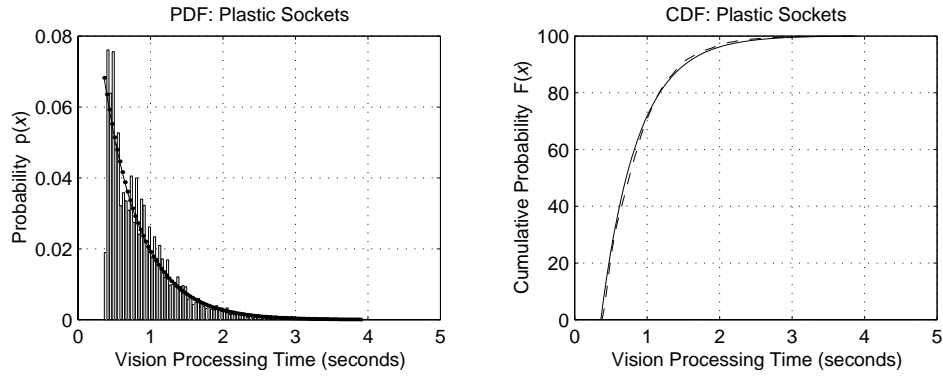


Figure 7-4: PDF of the Vision Distribution of the Plastic Sockets

7.1.4.4 *Robot System Throughput*

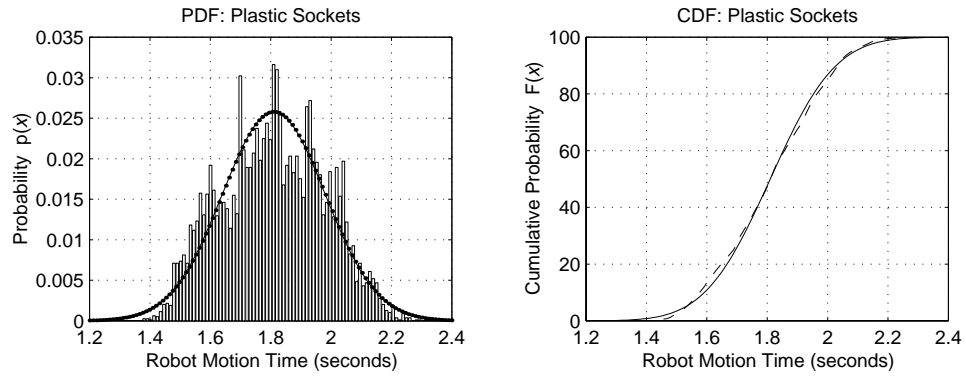


Figure 7-5: PDF and CDF of the Plastic Sockets

## 7.2 Tool Offset Calculator Examples

Entering data into the program is a simple procedure. Each offset is entered as an  $x$ ,  $y$ ,  $z$ , yaw, pitch, and roll. For example, if it has been previously determined that the quick connector offset is 10 in the  $x$ , -5 in the  $y$ , 30 in the  $z$ ,  $65^\circ$  of yaw, and no pitch or roll, then the input would be as follows. In the quick connector input section, in the  $x$  box, 10 would be entered, in the  $y$  box, -5, in the  $z$  box, 30, and in the yaw box, 65. The pitch and roll boxes may be left blank. A input box left blank is interpreted as a 0.

Any units may be used for the translation part of the offset as long as they are consistent. If millimeters are used on for the input, then the output will be in millimeters. If inches are used for the inputs, then the output will be in inches. The rotation parts of the transforms need to be input as degrees. Using radians as the input units will cause the output to be invalid.

The output of the program is in three forms. The first is three translations and three Euler angles. As shown, two tool offsets are displayed. The first offset is only rotation and the second offset is only translation. Equation (3.41) was used to generate the translational components given the final tool offset as a single transform. It is important that two offsets be used when entering the data into a robot controller or the incorrect offset will be used. The units of the translation part of the transform are the same as those used for data entry. The units of the rotation part of the transform are degrees.

The V+ form of the transform is used to make it easy to insert the resulting offset into a V+ program. Simply copy the transform verbatim into the program. Because the translation part of the returned transform is relative to the rotated coordinate frame, the two transforms cannot be combined into one. Doing so will cause the incorrect offset to be used in the program with the end result being a possible robot crash. The units of the translation part of the transform are the same as those used for data entry. The units of the rotation part of the transform are degrees.

The final reporting of the resulting transform is in matrix form. This matrix is the complete transform, translation first then rotation. It is listed this way since altering a matrix transform by hand is not intuitive. Therefore reporting the transform as two part would only cause an unnecessary amount of data to be presented. The units of the translation part of the transform are the same as those used for data entry. Since the 3x3 rotation sub-matrix is the result of Sines and Cosines, there are no units associated with the rotation part.

Three examples will be examined as a guide to determining offsets and using the program. The first example will be retrieving a part from a flexible parts feeder. The second example will be placing a part on a fixture on a modular table, and the third will be placing a part on a fixture at an angle.

#### 7.2.1 Picking a Part from a Flexible Feeder

Begin by placing an example part into the gripper on the robot. If the jaws of the gripper do not self-center the part (e.g. a flat plate, parallel jaw gripper), make sure the part is properly oriented in the jaws of the gripper. Now move (by hand or drive using the manual control pendant) the robot to a position from which it will retrieve the part. That is, the location at which the robot can close its gripper jaws and have a secure grasp of the part. Request that the robot's controller display the current position of the robot's flange. Make sure that the position returned is with the  $z$  axis of the robot pointed up. This usually requires that the flange coordinate frame fix offset, described in Section 3.7.2.6, be enabled before requesting the flange position. Also note the offset associated with the quick connector which is attached to the robot. This offset should have been determined when the quick connector was attached to the robot. After the location of the robot has been determined, carefully open the jaws of the gripper and move the robot to a location outside of the vision window. It is important to not disturb the part as the robot is being moved. The part should be in the same position as it was when the robot's location was noted. Lastly, take a picture of the part with the vision system. The results

from the vision system should be a world location of the part. The actual location returned can be any point on the part, it should, however, be chosen such that it is easy to locate and identify with the vision system and returns a consistent result in the presence of noise and variability in the part image.

Assume, as an example, the data displayed in Table 7-1 was determined.

	Quick Connect Offset	Robot Flange Location	Part Location on the Flex Feeder
<i>x</i>	0.25	-56.346	-58.48
<i>y</i>	-0.36	123.435	119.043
<i>z</i>	-35.61	234.903	98.354
yaw	1.284	-125.98	-22.564
pitch	0	0	0
roll	0	0	0

Table 7-1: Example Locations - Flexible Feeder Example

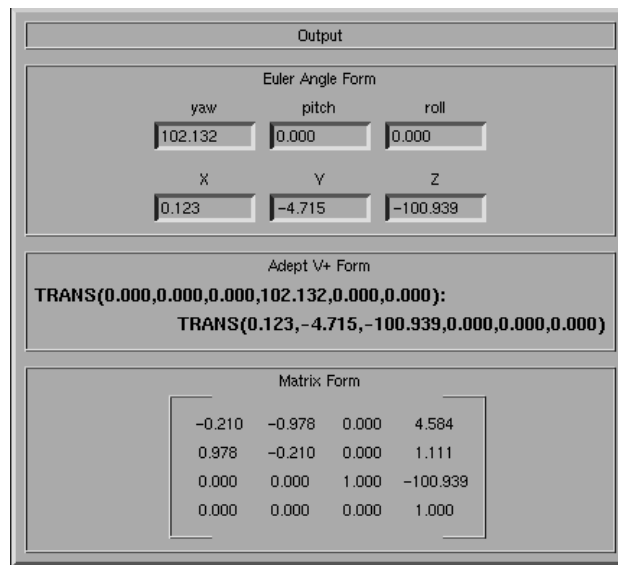


Figure 7-6: Results of the Tool Offset Calculator

After determining the data, the tool offset program may be used to determine the offset. Make sure the flex feeder mode of the program is selected and enter the data into the relevant sections of the program. Leave the pitch and roll sections of the input blank. If the flex feeder mode was not selected, then there will be no flex feeder part location input section. After inputting the data, click on the "Compute" button to calculate the

gripper transform. Figure 7-6 shows the results. If programming in V+, the transform may be copied directly into the program from the V+ Form section. When using the transform with another system use either the Euler form or the matrix form depending on the requirements of that particular system.

### 7.2.2 Placing a Part on a Fixture on a Modular Table

Begin by instructing the robot to register the modular table into its world coordinate system. Note the location of the origin of the table in the world system. Next, instruct the robot to retrieve the part from wherever it is grasped. It is important to allow the robot to grasp the part itself so that the position of the part in the gripper jaw is not altered by the act of putting the part into the jaw by hand. Next drag the robot into the final position. Make sure the robot is in the location from which it can open its jaws and release the part. Request the robot's controller to display the current location of the robot's flange. This usually requires that the flange coordinate frame fix offset, described in Section 3.7.2.6, be enabled before requesting the flange position. Also note the offset associated with the quick connector which is attached to the robot. This offset set should have been determined when the quick connector was attached to the robot.

Assume, as an example, the data displayed in Table 7-2 was determined.

	Quick Connect Offset	Robot Flange Location	Modular Table Frame Origin	Table Relative Location
<i>x</i>	0.25	154.946	45.564	100.0
<i>y</i>	-0.36	109.8	35.081	75.0
<i>z</i>	-35.61	187.45	-40.354	80.0
yaw	1.284	108.96	-85.453	45.0
pitch	0	0	0	0
roll	0	0	0	0

Table 7-2: Example Locations - Modular Table Example

Note that the location of the fixture on the modular table is exact. This is because the location should be read from the mechanical drawing and entered per print. Any inaccuracies in the assembly of the table will be accounted for in the gripper transform being determined.

Make sure the modular table mode of the program is selected and enter the data into the relevant sections of the program. Leave the pitch and roll sections of the input blank. If the modular table mode was not selected, there will be no table frame origin location input section and the table relative position input section will be grayed out. After the data has been entered, press the “Calculate” button to compute the gripper transform. Figure 7-7 shows the results. If programming in V+, the transform may be copied directly into the program from the V+ Form section. When using the transform with another system use either the Euler form or the matrix form depending on the requirements of that particular system.

The screenshot shows the output of a Tool Offset Calculator. It is divided into three sections:

- Euler Angle Form:**

yaw	pitch	roll
-150.679	0.000	0.000
X	Y	Z
88.907	-145.994	-112.194
- Adept V+ Form:**

**TRANS(0.000,0.000,0.000,-150.679,0.000,0.000):**  
**TRANS(88.907,-145.994,-112.194,0.000,0.000,0.000)**
- Matrix Form:**

-0.872	0.490	0.000	-149.010
-0.490	-0.872	0.000	83.753
0.000	0.000	1.000	-112.194
0.000	0.000	0.000	1.000

Figure 7-7: Results of the Tool Offset Calculator

### 7.2.3 Placing a Part on a Fixture at an Angle

An example of this type of action would be putting a cylindrical part into a hole which is at a 45° angle to the horizontal. Also assume that a SCARA type robot is being used, therefore it is assumed that a special gripper is used to hold the part at the proper insertion angle. If a 5 or 6 DOF robot was used, then the pitch term would appear in the flange location of the robot and in the table relative location term. The resulting tool

offset would then have no pitch term. Other than the introduction of a pitch term, this procedure is identical to the previous.

Begin by instructing the robot to register the modular table into its world coordinate system. Note the location of the origin of the table in the world system. Next, instruct the robot to retrieve the part from wherever it is grasped. It is important to allow the robot to grasp the part itself so that the position of the part in the gripper jaw is not altered by the act of putting the part in to the jaw by hand. Next drag the robot into the final position. Make sure the robot is in the location from which it can open its jaws and release the part. Request the robot's controller to display the current location of the robot's flange. This usually requires that the flange coordinate frame fix offset, described in Section 3.7.2.6, be enabled before requesting the flange position. Also note the offset associated with the quick connector which is attached to the robot. This offset set should have been determined when the quick connector was attached to the robot.

Assume, as an example, the data displayed in Table 7-3 was determined.

	Quick Connect Offset	Robot Flange Location	Modular Table Frame Origin	Table Relative Location
$x$	0.25	154.946	45.564	100.0
$y$	-0.36	109.8	35.081	75.0
$z$	-35.61	187.45	-40.354	80.0
yaw	1.284	108.96	-85.453	45.0
pitch	0	0	0	45.0
roll	0	0	0	0

Table 7-3: Example Locations - Modular Table Example

Make sure the modular table mode of the program is selected and enter the data into the relevant sections of the program. Leave the pitch and roll sections of the input blank except for the 45° pitch in the table relative input section. If the modular table mode was not selected, there will be no table frame origin location input section and the table relative position input section will be grayed out. After the data has been entered, press the "Calculate" button to compute the gripper transform. Figure 7-8 shows the resulting gripper transform. Again, the transform may be copied directly into a V+

program from the V+ Form section. When using the transform with another system use either the Euler form or the matrix form depending on the requirements of that particular system.

The screenshot shows the output of a Tool Offset Calculator. It is divided into three sections:

- Euler Angle Form:** Displays yaw, pitch, and roll values in input fields, and X, Y, and Z coordinates in output fields.

yaw	pitch	roll
-150.679	45.000	0.000
X	Y	Z
142.200	-145.994	-16.467
- Adept V+ Form:** Shows two TRANS commands: `TRANS(0.000,0.000,0.000,-150.679,45.000,0.000):` and `TRANS(142.200,-145.994,-16.467,0.000,0.000,0.000)`.
- Matrix Form:** Displays a 4x4 transformation matrix.

-0.617	0.490	-0.617	-149.010
-0.346	-0.872	-0.346	83.753
-0.707	0.000	0.707	-112.194
0.000	0.000	0.000	1.000

Figure 7-8: Results of the Tool Offset Calculator

## 7.3 Programs

### 7.3.1 V+ QC Calibration Program

This program was used to determine the translational portion of the quick connector offset as described in Chapter 3 of the dissertation.

```

PROGRAM test()
; Make sure the correct robot is being used.
  DETACH ()
  SELECT ROBOT = 1
  ATTACH ()
; Make sure no tool offsets are in place, then call in the
; tool offset which flips the coordinate system back to the
; positive z up orientation. Finally, call in the offset
; that is being calculated. Initially, this offset should
; be set to 0
  TOOL NULL
  TOOL gripfix:qcfix
; Set the speed of the robot to something reasonable so that
; the indicator is not being banged around
  SPEED 25
; Create a dummy position that is away from the indicator,
; but that is purely in either the x or y direction from the
; indicator.
; Move to this position to begin the testing.
  MOVE dummy
  BREAK
; Loop thru 8 different angles (multiples of 45 deg) and wait
; at each angle for the measurement to be recorded. Call the
; position of the indicator "temp" and only change the rotational
; part of the position for each point. Move to "dummy" between
; each movement to "temp". Also, move to dummy at the end of
; the process.
  FOR i = 1 TO 8
    TYPE i
    MOVE temp[i]
    BREAK
    TIMER 1 = 0
    WAIT TIMER(1) >= 3
    MOVE dummy
    BREAK
  END
END

```

### 7.3.2 Tool Offset Calculator

```

/* PROGRAM NAME: tool_offset_calculator
FILE NAME: tool_offset_calculator_cb.c
AUTHOR: Greg Causey (gcc)
DATE: 3/2/1998
DESCRIPTION:

This file contains the routines that do all the work of the
program. It contains the callback routines that are executed when
buttons are pressed on the main form. See each individual routine
for an explanation of that routine.

REVISIONS:

*/

/* includes */
#include "forms.h"
#include "tool_offset_calculator.h"
#include <stdlib.h>
#include <math.h>
#include <string.h>

/* Definition of pi incase the system doesn't know about it. This is
needed to compile the program on the suns. It is not needed on
linux systems. */
/* #define PI 3.141592645 */

/* Global pointer to the base frame of the help window */
FD_Help_Window *global_pointer_to_the_help_window;

/* global variable which indicates whether the flex feeder radio
button is pressed. This is used in the compute values subroutine to
determine if it necessary to read inputs from the table relative
input section. */
int flex_feed=1;

/* declaration of functions used to determine the offset */
void make_z_rot_matrix(double output[4][4],double input);
void make_y_rot_matrix(double output[4][4],double input);
void make_trans_matrix(double output[4][4],double inputx,
double y, double z);
void euler_parms_find(double input[4][4], double output[6]);
void matrix_mulp(double input_1[4][4],double input_2[4][4],
double output[4][4]);
void homo_matrix_invert(double input[4][4],double output[4][4]);

/* This is the callback attached to the compute button. This is the
routine that actually calculates the tool offset given the input
data. */
void compute_values(FL_OBJECT *ob, long data)
{
/* This sets the local pointer form to the main window of the
program. This is used to allow acces to any part of the form. */
FD_Tool_Offset_Calculator *form=ob->u_vdata;

/* First set up all the variables that will be needed. Offsets are
stored internally in two forms. The first is what is referred to
as the euler form. This consists of thre translational offsets
(x, y, and z) and three rotational offsets (yaw, pitch, and
roll). The z-y-z (or 3-2-3) euler angle set (as used by Adept
robotics) is always used. The second form is a homogeneous
coordinate matrix. This is 4x4 matrix. This is the form which is
necessary if calculations are to be performed. The euler form is
more "human friendly". Routines are used to convert between the
two forms.

The following is an explanation of the variables. The six member
arrays are used to hold offsets in the euler angle form while the
4x4 arrays are used to hold offsets in the matrix form.

qc: array holding the quick connector offset.
table: array holding the location of the origin of the modular
table or pallet in euler form.

```

```

table_rel: array holding the location of a fixture on a modular
           table or on a pallet relative to the origin of the
           table or pallet in euler form.
flange: array holding the location of the flange of the robot
        in euler form.
euler_form: array holding the euler angles and translational
            pieces of the resulting tool offset in euler form.
matrix_out: 4x4 matrix used to store the resulting tool offset
            in matrix form.
matrix_qc: 4x4 matrix used to store the quick connector offset
           in matrix form.
matrix_table: 4x4 matrix used to store the location of the
             modular table or pallet in matrix form.
matrix_flange: 4x4 matrix used to store the location of the
              flange of the robot in matrix form.
matrix_table_rel: 4x4 matrix used to store the location of
                 a fixture on a modular table or on a pallet
                 relative to the table or pallet origin in
                 matrix form.
temp_matrix_X: Five (where X is a number) 4x4 matrices used for
              temporary storage of homogeneous transforms.
i,j: integer counters used for looping
euler_form_char: array of character strings which hold an ascii
                version of the resulting euler form tool offset.
                Ascii is needed to display properly on the
                resulting program window.
v_plus: two character strings used to hold the Adept V+ version
        of the euler form transform. This is used to display
        the resulting transform for easy entry into a V+ program.
matrix_out_char: 4x4 matrix of character strings used to hold the
                matrix form of the resulting tool offset.
*/

double qc[6];
double table[6];
double flange[6];
double table_rel[6];
double euler_form[6];
double matrix_out[4][4];
double matrix_qc[4][4];
double matrix_table[4][4];
double matrix_flange[4][4];
double matrix_table_rel[4][4];
double temp_matrix_1[4][4];
double temp_matrix_2[4][4];
double temp_matrix_3[4][4];
double temp_matrix_4[4][4];
double temp_matrix_5[4][4];
int i,j;

char euler_form_char[6][10];
char v_plus[2][70];
char matrix_out_char[4][4][10];

/* initialize the variables to 0.0 */

for(i=0;i<6;i++) {
  qc[i]=0.0;
  table[i]=0.0;
  flange[i]=0.0;
  table_rel[i]=0.0;
  euler_form[i]=0.0;
  if(i<4) {
    for(j=0;j<4;j++) {
      matrix_out[i][j] = 0.0;
      matrix_qc[i][j] = 0.0;
      matrix_table[i][j] = 0.0;
      matrix_flange[i][j] = 0.0;
      matrix_table_rel[i][j] = 0.0;
      temp_matrix_1[i][j] = 0.0;
      temp_matrix_2[i][j] = 0.0;
      temp_matrix_3[i][j] = 0.0;
      temp_matrix_4[i][j] = 0.0;
      temp_matrix_5[i][j] = 0.0;
    }
  }
}

/* Read in the data from the form. Use the atof command since the

```

```

data is stored in the form as ascii character strings. Store the
data to the proper arrays. Data being read in is in euler
form. Check whether the flex_feed radio button has been
pressed. If it has been pressed then set the table_rel array to
0. If it hasn't been pressed then read in the table_rel data from
the form. The names "qc_x, qc_y, ..." are names of input boxes on
the main window and are defined in the file
tool_offset_calculator.c */

qc[0] = atof(fl_get_input(form->qc_x));
qc[1] = atof(fl_get_input(form->qc_y));
qc[2] = atof(fl_get_input(form->qc_z));
qc[3] = atof(fl_get_input(form->qc_yaw));
qc[4] = atof(fl_get_input(form->qc_pitch));
qc[5] = atof(fl_get_input(form->qc_roll));

table[0] = atof(fl_get_input(form->table_frame_x));
table[1] = atof(fl_get_input(form->table_frame_y));
table[2] = atof(fl_get_input(form->table_frame_z));
table[3] = atof(fl_get_input(form->table_frame_yaw));
table[4] = atof(fl_get_input(form->table_frame_pitch));
table[5] = atof(fl_get_input(form->table_frame_roll));

flange[0] = atof(fl_get_input(form->flange_x));
flange[1] = atof(fl_get_input(form->flange_y));
flange[2] = atof(fl_get_input(form->flange_z));
flange[3] = atof(fl_get_input(form->flange_yaw));
flange[4] = atof(fl_get_input(form->flange_pitch));
flange[5] = atof(fl_get_input(form->flange_roll));

if(flex_feed) {
    table_rel[0] = atof(fl_get_input(form->table_relative_x));
    table_rel[1] = atof(fl_get_input(form->table_relative_y));
    table_rel[2] = atof(fl_get_input(form->table_relative_z));
    table_rel[3] = atof(fl_get_input(form->table_relative_yaw));
    table_rel[4] = atof(fl_get_input(form->table_relative_pitch));
    table_rel[5] = atof(fl_get_input(form->table_relative_roll));
}
else {
    table_rel[0] = 0.0;
    table_rel[1] = 0.0;
    table_rel[2] = 0.0;
    table_rel[3] = 0.0;
    table_rel[4] = 0.0;
    table_rel[5] = 0.0;
}

/* Convert all the angles to radians. Only the last three terms of
each array are angle terms. The first three terms are
translational terms. */

for(i=3;i<6;i++) {
    qc[i] *= PI/180.0;
    table[i] *= PI/180.0;
    flange[i] *= PI/180.0;
    table_rel[i] *= PI/180.0;
}

/* Compute the transform matrices. This section of the code converts
the euler form offsets to matrix form. This is done by first
creating a translational matrix and three rotational matrices
(one for each euler angle) and then multiplying the matrices
together. This was easier to code than a single command to go
straight to the complete transform matrix since each rotational
matrix is very basic and a function to multiply matrices is
available. The matrix is constructed according to the following
equation ([[] denotes a 4x4 matrix).

[complete transform] = [translation] X [yaw rotation] X
                       [pitch rotation] X [roll rotation]
*/

/* Computer the quick connector matrix transform */
make_trans_matrix(temp_matrix_1,qc[0],qc[1],qc[2]);
make_z_rot_matrix(temp_matrix_2,qc[3]);
make_y_rot_matrix(temp_matrix_3,qc[4]);
make_z_rot_matrix(temp_matrix_4,qc[5]);
matrix_mulp(temp_matrix_1,temp_matrix_2,temp_matrix_5);
matrix_mulp(temp_matrix_5,temp_matrix_3,temp_matrix_1);
matrix_mulp(temp_matrix_1,temp_matrix_4,matrix_qc);

```

```

/* Compute the table frame origin location matrix transform */
make_trans_matrix(temp_matrix_1,table[0],table[1],table[2]);
make_z_rot_matrix(temp_matrix_2,table[3]);
make_y_rot_matrix(temp_matrix_3,table[4]);
make_z_rot_matrix(temp_matrix_4,table[5]);
matrix_mulp(temp_matrix_1,temp_matrix_2,temp_matrix_5);
matrix_mulp(temp_matrix_5,temp_matrix_3,temp_matrix_1);
matrix_mulp(temp_matrix_1,temp_matrix_4,matrix_table);

/* Compute the robot flange location matrix transform */
make_trans_matrix(temp_matrix_1,flange[0],flange[1],flange[2]);
make_z_rot_matrix(temp_matrix_2,flange[3]);
make_y_rot_matrix(temp_matrix_3,flange[4]);
make_z_rot_matrix(temp_matrix_4,flange[5]);
matrix_mulp(temp_matrix_1,temp_matrix_2,temp_matrix_5);
matrix_mulp(temp_matrix_5,temp_matrix_3,temp_matrix_1);
matrix_mulp(temp_matrix_1,temp_matrix_4,matrix_flange);

/* compute the table relative matrix transform */
make_trans_matrix(temp_matrix_1,table_rel[0],table_rel[1],
table_rel[2]);
make_z_rot_matrix(temp_matrix_2,table_rel[3]);
make_y_rot_matrix(temp_matrix_3,table_rel[4]);
make_z_rot_matrix(temp_matrix_4,table_rel[5]);
matrix_mulp(temp_matrix_1,temp_matrix_2,temp_matrix_5);
matrix_mulp(temp_matrix_5,temp_matrix_3,temp_matrix_1);
matrix_mulp(temp_matrix_1,temp_matrix_4,matrix_table_rel);

/* Now its time to determine the resulting quick connector
offset. The offset is determined by solving the following matrix
equation:

[resulting offset] = Inverse[Quick Connect] X
                    Inverse[Flange Location] X [Table Location] X
                    [Table Relative Location]

This is accomplished in two steps. First determine the inverse of
the quick connector and flange transforms then multiply out the
equation. The final action to perform is then to switch the order
of the transform. A general homogeneous coordinate transform does
translation then rotation. In this case, it is advantageous to
specify the offset as a rotation then a translation. In this way,
it is possible to easily "tweek" the offset by hand while the
system is running to improve it. This is done by pre multiplying
the final offset in matrix form by an transform composed of just
the rotational part of the resulting matrix. NOTE: the final
resulting offset matrix is the same this only affects the
translational offsets are reported in euler form.
*/

/* Take the inverse of the quick connect and flange transforms */
homo_matrix_invert(matrix_gc,temp_matrix_1);
homo_matrix_invert(matrix_flange,temp_matrix_2);

/* Multiply of the matrix equation */
matrix_mulp(matrix_table,matrix_table_rel,temp_matrix_3);
matrix_mulp(temp_matrix_2,temp_matrix_3,temp_matrix_4);
matrix_mulp(temp_matrix_1,temp_matrix_4,matrix_out);

/* Multiply the resulting matrix by the inverse of its rotation
part */
homo_matrix_invert(matrix_out,temp_matrix_1);
for(i=0;i<3;i++) temp_matrix_1[i][3] = 0.0;
matrix_mulp(temp_matrix_1,matrix_out,temp_matrix_2);

/* Determine the euler form of the resulting transform given the
matrix form. */
euler_parms_find(matrix_out,euler_form);

/* Convert the angles back to degrees */
euler_form[3] *= 180.0/PI;
euler_form[4] *= 180.0/PI;
euler_form[5] *= 180.0/PI;

/* Convert the numerical data back to character strings for posting
in the form */

/* Construct the euler angle form character array. Since this is

```

```

reported as first a rotation then a translation, use the
translation values determined by multiplying the output matrix by
its inverse rotation as explained above. The first three terms of
the array are the rotational part, the second three terms are the
translational part.*/

for(i=0;i<3;i++) {
    sprintf(euler_form_char[i],"%.3f",temp_matrix_2[i][3]);
}
for(i=3;i<6;i++) {
    sprintf(euler_form_char[i],"%.3f",euler_form[i]);
}

/* Construct the V+ transform output text strings. V+ uses a canned
transform function of the form
"TRANS(x,y,z,yaw,pitch,roll)". Since the rotation is desired
before the translation, two such functions are called in series
in a V+ program. Therefore report the two functions as they
should appear in a program. The first function has only the
rotational part, the other three terms are set to 0. The second
function has only the translation part, the last three terms are
set to 0. Use the strcat funtion to build up the output
strings. */

strcpy(v_plus[0],"TRANS(0.000,0.000,0.000,");
strcpy(v_plus[1],"TRANS(");
for(i=0;i<3;i++) {
    strcat(v_plus[0],euler_form_char[i+3]);
    if(i<2) strcat(v_plus[0],",");
    strcat(v_plus[1],euler_form_char[i]);
    if(i<2) strcat(v_plus[1],",");
}
strcat(v_plus[0],"):");
strcat(v_plus[1],"",0.000,0.000,0.000)");

/* Construct the matrix form character array. This is simply the
matrix determined from the original matrix equation. There is no
reason to separate the translation and rotation parts of the
transform since a transform in matrix form is difficult to
"tweek" by hand. */
for(i=0;i<4;i++) {
    for(j=0;j<4;j++) {
        sprintf(matrix_out_char[i][j],"%.3f",matrix_out[i][j]);
    }
}

/* Write the data out to the form. This section writes all the data
determined by the above procedures back out to the program's
display. As in the input section, the names refer to label
location on the program's window and are defined in the
tool_offset_calculator.c file. */

/* This section fills out the matrix output */
fl_set_object_label(form->offset_matrix_11,matrix_out_char[0][0]);
fl_set_object_label(form->offset_matrix_12,matrix_out_char[0][1]);
fl_set_object_label(form->offset_matrix_13,matrix_out_char[0][2]);
fl_set_object_label(form->offset_matrix_14,matrix_out_char[0][3]);
fl_set_object_label(form->offset_matrix_21,matrix_out_char[1][0]);
fl_set_object_label(form->offset_matrix_22,matrix_out_char[1][1]);
fl_set_object_label(form->offset_matrix_23,matrix_out_char[1][2]);
fl_set_object_label(form->offset_matrix_24,matrix_out_char[1][3]);
fl_set_object_label(form->offset_matrix_31,matrix_out_char[2][0]);
fl_set_object_label(form->offset_matrix_32,matrix_out_char[2][1]);
fl_set_object_label(form->offset_matrix_33,matrix_out_char[2][2]);
fl_set_object_label(form->offset_matrix_34,matrix_out_char[2][3]);
fl_set_object_label(form->offset_matrix_41,matrix_out_char[3][0]);
fl_set_object_label(form->offset_matrix_42,matrix_out_char[3][1]);
fl_set_object_label(form->offset_matrix_43,matrix_out_char[3][2]);
fl_set_object_label(form->offset_matrix_44,matrix_out_char[3][3]);

/* This section fills out the V+ transform output */
fl_set_object_label(form->offset_vplus_1,v_plus[0]);
fl_set_object_label(form->offset_vplus_2,v_plus[1]);

/* This section fills out the euler form output */
fl_set_object_label(form->offset_yaw,euler_form_char[3]);
fl_set_object_label(form->offset_pitch,euler_form_char[4]);
fl_set_object_label(form->offset_roll,euler_form_char[5]);
fl_set_object_label(form->offset_x,euler_form_char[0]);

```

```

fl_set_object_label(form->offset_y,euler_form_char[1]);
fl_set_object_label(form->offset_z,euler_form_char[2]);

/* End of the calculate callback */
}

/* The following routines are used to construct the rotation and
translation matrices, determine the euler parameters given a
transform matrix, multiply transform matrices, and invert a
transform matrix. In all the routines, the full matrix is always
specified even if it contains mostly 0's. This is done to ensure
that errors will not creep into the problem from old data if the
program is ran multiple time without being restarted. Since the
matrices are passed by reference, there is no need to explicitly
return anything. */

/* This routine constructs a 4x4 rotation matrix about the Z axis
given an input angle. The translational part of the matrix is set
to 0 since this is only a rotation matrix. The input is a single
angle, in radians, which specifies the rotation. The output is a
4x4 matrix. */

void make_z_rot_matrix(double output[4][4],double input)
{
output[0][0]=cos(input);
output[0][1]=(-sin(input));
output[0][2]=0.0;
output[0][3]=0.0;
output[1][0]=sin(input);
output[1][1]=cos(input);
output[1][2]=0.0;
output[1][3]=0.0;
output[2][0]=0.0;
output[2][1]=0.0;
output[2][2]=1.0;
output[2][3]=0.0;
output[3][0]=0.0;
output[3][1]=0.0;
output[3][2]=0.0;
output[3][3]=1.0;
}

/* This routine constructs a 4x4 rotation matrix about the Y axis
given an input angle. The translational part of the matrix is set
to 0 since this is only a rotation matrix. The input is a single
angle, in radians, which specifies the rotation. The output is a
4x4 matrix. */

void make_y_rot_matrix(double output[4][4],double input)
{
output[0][0]=cos(input);
output[0][1]=0.0;
output[0][2]=sin(input);
output[0][3]=0.0;
output[1][0]=0.0;
output[1][1]=1.0;
output[1][2]=0.0;
output[1][3]=0.0;
output[2][0]=(-sin(input));
output[2][1]=0.0;
output[2][2]=cos(input);
output[2][3]=0.0;
output[3][0]=0.0;
output[3][1]=0.0;
output[3][2]=0.0;
output[3][3]=1.0;
}

/* This routine constructs a 4x4 translation matrix. The rotational
part of the matrix is set to the identity matrix since this is only
a translation matrix. The input is three translational
quantities. The output is a 4x4 matrix. */

```

```

void make_trans_matrix(double output[4][4],double x,
                      double y, double z)
{
output[0][0]=1.0;
output[0][1]=0.0;
output[0][2]=0.0;
output[0][3]=x;
output[1][0]=0.0;
output[1][1]=1.0;
output[1][2]=0.0;
output[1][3]=y;
output[2][0]=0.0;
output[2][1]=0.0;
output[2][2]=1.0;
output[2][3]=z;
output[3][0]=0.0;
output[3][1]=0.0;
output[3][2]=0.0;
output[3][3]=1.0;
}

/* This routine multiplies two matrices together. The matrices are
hardcoded to be of size 4x4 since that homogeneous transforms are
always 4x4 matrices. */

void matrix_mulp(double input_1[4][4],double input_2[4][4],
                double output[4][4])
{
/* Define the variables.
sum: temporary variable to hold the intermediate calculations
ctr1,ctr2,ctr3: integer counter variables
*/
double sum;
int ctr1,ctr2,ctr3;

/* Standard loop to multiply two matrices */
for(ctr1=0;ctr1<4;ctr1++) {
for(ctr2=0;ctr2<4;ctr2++) {
sum=0.0;
for(ctr3=0;ctr3<4;ctr3++) {
sum+=input_1[ctr1][ctr3]*input_2[ctr3][ctr2];
}
output[ctr1][ctr2] = sum;
}
}
}

/* This routine determines the inverse of a 4x4 homogeneous transform
matrix. Homogeneous transforms have a very simple inverse and it is
easier to program this way rather than use a general matrix
inversion routine. */

void homo_matrix_invert(double input[4][4],double output[4][4])
{
/* Define the variables
sum: temporary variable to hold the intermediate calculations
ctr1,ctr2: integer counter variables
*/

double sum;
int ctr1,ctr2;

/* First invert the 3x3 rotational part of the matrix */
for(ctr1=0;ctr1<3;ctr1++) {
for(ctr2=0;ctr2<3;ctr2++) {
output[ctr1][ctr2]=input[ctr2][ctr1];
}
}

/* Next determine the translational part using the
Transpose[rotational part] X [translational part] formula. This
is a standard 3x3 X 3x1 matrix multiplication. */

```

```

for(ctr1=0;ctr1<3;ctr1++) {
    sum=0.0;
    for(ctr2=0;ctr2<3;ctr2++) {
        sum+=(-input[ctr2][3])*input[ctr2][ctr1];
    }
    output[ctr1][3] = sum;
}

/* Finally set the bottom row to 0,0,0,1 */
for(ctr1=0;ctr1<3;ctr1++) {
    output[3][ctr1]=0.0;
}
output[3][3]=1.0;
}

```

/\* This routine determines the euler form given a homogeneous transform matrix. This is by far the ugliest code in the program. Determining euler angles for a given matrix is not a trivial or easy thing to do. It is compounded by singularities in the euler angle set near pitch angles of  $n\pi$ , by numerical inaccuracies in the transcendental and inverse transcendental functions, uncertainty in the correct sign of the results and by the use of inverse trig functions. It is also difficult because there are usually multiple sets of euler angles that can arrive at the same final rotated state. While all the sets of euler angles which arrive at a given final state are equally valid, they sometimes don't appear as expected which can cause the user to doubt the validity of the output.

The translational part of the offset simply appears in the 4th column, 1st three rows of the matrix. It can be easily read off. Determining the angles is where the problems begin. Since the 3-2-3 euler angle set is used, the general form of the rotation part of the 4x4 matrix can be examined. It can be seen that the 3rd row 3rd column position has the sole term  $\text{Cos}(\text{pitch})$ . From this term we can determine the pitch angle. However, since the arccos function is used, only a positive angle is returned. We cannot recover the sign information. It can then be seen that the 1st row 3rd column is equal to  $\text{Cos}(\text{yaw}) \times \text{Sin}(\text{pitch})$ . Using the previously determined pitch, the yaw can be determined. Again, however, no sign information is retrievable. Finally, upon examination, the 3rd row 1st column is equal to  $-(\text{Cos}(\text{roll}) \times \text{Sin}(\text{pitch}))$ . This can be used to determine the roll, again with no sign. This leaves us with sixteen sets of three angles from which we must choose the correct one. The following procedure is used to determine the correct angles.

First, check the pitch angle. If it is less than 0.1 then assume that the pitch is equal to 0. Next assume all the yaw + roll angle is lumped into the yaw. Examining the matrix shows that the 1st row 1st column value is equal to  $\text{Cos}(\text{yaw})$  after substituting  $\text{pitch}=\text{roll}=0$ . Use this to determine the yaw. Then use the 2nd row 1st column matrix value to test for the correct sign. If the incorrect sign was found, switch the sign of the yaw and return.

If the pitch angle was found to be greater than 0.1 degrees then solve for all the angles. First determine the pitch angle and then determine candidate yaw and roll angles. Next using the 1:3, 2:3, 3:1, and 3:2 matrix values (where 1:3 is first row 3rd column) test for the correctness of the signs of the angles. If an error is found, change the sign of one of the angles and try again. If none of the eight combinations are correct, change the sign of the original pitch angle, recompute the yaw and roll angles and go through the trial procedure again. Finally, before exiting, check to be sure that a 180, -pitch, -180 condition wasn't found (if it was correct it).

Several techniques are used throughout this routine to try and counter the problems discussed in the first paragraph. They are discussed below.

Signularities: Since we are using the 3-2-3 euler angle set, there is a problem when the pitch (middle rotation) goes to multiples of  $n\pi$ . Since most of the offsets have no pitch (ie, the second rotation is 0) this is constantly a problem. However, since all that is desired is the final offset, it is possible to lump all the

z rotation into either the yaw or roll. In this case, the rotation is moved into the yaw. A check must be made to determine if this is the case and an alternate section of code is used to determine the correct yaw. To combat problems of numerical inaccuracies in the trig functions, if the total pitch angle for the matrix is determined to be less than 0.1 degrees, it is set to 0. This is not a problem since this is a real physical system and a pitch angle of less than a tenth of a degree would not make any difference to the operation of the system.

Multiple sets of euler angles: Generally there is nothing that can be done to correct for recovering various sets of euler angles. There is no mathematical case for one set of angles being "more correct" than any other assuming that the final rotated state is the same. The one case which is dealt with is the case of only pitch and no yaw or roll. That is, the yaw=0, pitch, roll=0 case which can lead to the recovered euler angle set of yaw=180, -pitch, roll=-180. In this situation, the initial guess for the pitch had an incorrect sign, but because a yaw of 180 degrees points the y axis in the opposite direction, a negative pitch angle followed by a roll of -180 degrees will achieve the same final orientation. However, it is obvious that this is not the intended angle set to return. To fix this, a check is made to determine if the yaw and roll angles are 180 and -180 degrees respectively. If this is the case, then the yaw and roll angles are set to 0 and the sign of the pitch angle is switched.

Numerical inaccuracies: Numerical inaccuracies occur because of the use of transcendental and inverse transcendental functions. When these functions are evaluated over certain ranges, the output is not well behaved. Another problem is the range of input values to the functions. For example, if an input to the arccos function is a bit over +1 or a bit under -1 a NAN output will result. Since a lot of matrix manipulation occurs in determining the final transform matrix, this is a possibility, especially when some of the initial angles of the input offsets are 0. Another example is examining two values which should be equal as a test. Because of the ill-behaved inverse functions, the two numbers might not be exactly equal and the test will fail even though it should be true. To fix this problem several steps are taken. First, to fix the greater than 1 error, a test is made before every arccos call to ensure the input value is between -1 and 1. If it is outside this range, it is set to -1 or 1 depending on whether it was greater than 1 or less than -1. To fix the equality problem the two numbers are subtracted and compared to a small value. If the result is less than the value, then equality is assumed.

The program takes as an input a 4x4 transform matrix. It returns a 6 element array which contains the x,y,z,yaw,pitch,roll values for the given matrix.

```

*/
void euler_parms_find(double input[4][4], double output[6])
{
    /* Declare some integer counter variables */
    int h,i,j,k;

    /* Set the translational parts of the matrix to the output array */
    output[0] = input[0][3];
    output[1] = input[1][3];
    output[2] = input[2][3];

    /* Determine the initial pitch angle. Be sure to check that the
    input value to the acos function is between -1 and 1. The check
    will be explained here but will appear a lot in this routine. If
    the input is less than -1 or greater than 1 then check if the
    input is greater than 1. If so, then the arccos should be set to
    0 (no need to actually call the acos function). Otherwise set the
    output equal to PI. If the input was between -1 and 1 then
    compute the acos */
    if(-1.0>input[2][2]||input[2][2]>1.0) {
        if(input[2][2]>1.0) {output[4]=0.0;}
        else {output[4] = PI;}
    }
    else{output[4] = acos(input[2][2]);}

    /* If the pitch angle is less than 0.1 degrees then set it equal to
    0. Now we need to do the special case calculations as discussed
    above. */

```

```

if(output[4] <= (0.1)*PI/180.0) {
    /* Set the pitch and roll to 0 */
    output[4] = 0.0;
    output[5] = 0.0;

    /* Determine the yaw. Make sure the input is between -1 and 1 */
    if(-1.0>input[0][0]||input[0][0]>1.0) {
        if(input[0][0]>1.0) {output[3]=0.0;}
        else {output[3] = PI;}}
    else{output[3] = acos(input[0][0]);}

    /* Check the sign of the yaw. If it is correct, fine. Otherwise
    change the sign of the angle. */
    if(sin(output[3])-input[1][0]>0.01){
        output[3] *= -1.0;
    }

    /* The pitch was not less than 0.1 degree so determine all the
    angles. */
} else {

    /* Determine the yaw angle. Note here that the quantity which goes
    into the acos function is more complex. It still must be
    checked for the -1 to 1 range. */
    if(-1.0>input[0][2]/sin(output[4])||
        input[0][2]/sin(output[4])>1.0) {
        if(input[0][2]/sin(output[4])>1.0) {
            output[3]=0.0;}
        else
            {output[3]=PI;}}
    else
        {output[3]=acos(input[0][2]/sin(output[4]));}

    /* Determine the roll angle */
    if(-1.0>input[2][0]/sin(output[4])||
        input[2][0]/sin(output[4])>1.0) {
        if(input[2][0]/sin(output[4])>1.0)
            {output[5]=PI;}
        else
            {output[5]=0.0;}}
    else
        {output[5]=acos(-input[2][0]/sin(output[4]));}

    /* Just a check to set any really small angle to 0. */
    if(fabs(output[3]) < 0.000001) output[3] = 0.0;
    if(fabs(output[5]) < 0.000001) output[5] = 0.0;

    /* This loop checks for the correctness of the angles. It alters
    signs and rechecks until a valid set of angles are found. When
    that occurs use a goto statement to get out of all the
    loops. The three inner loops check for the sign of the
    angles. If all eight possible sign combinations fail, the
    outermost loop changes the sign of the pitch, recomputes a yaw
    and roll and tries again. The if check at the center of the
    loop checks the possible angles vs. the matrix values. */
    for(h=0;h<2;h++){
        for(i=0;i<2;i++){
            for(j=0;j<2;j++){
                for(k=0;k<2;k++){
                    if((fabs(cos(output[3])*sin(output[4])
                        -input[0][2])<0.01)&&
                        (fabs(sin(output[3])*sin(output[4])
                        -input[1][2])<0.01)&&
                        (fabs(-(cos(output[5])*sin(output[4]))
                        -input[2][0])<0.01)&&
                        (fabs(sin(output[5])*sin(output[4])
                        -input[2][1])<0.01)) {
                        /* Found a set. Get out of here. */
                        goto end_of_angle_find_loop;
                    }
                    /* Change the roll and try again */
                    output[5] *= -1.0;
                    /* Change the pitch and try again */
                    output[4] *= -1.0;
                    /* Change the yaw and try again */
                    output[3] *= -1.0;
                }
            }
        }
    }

    /*Nothing worked. Change the sign of the pitch, recompute the
    yaw and roll and try again. */
}

```

```

output[4] *= -1.0;

/* Find a new yaw angle */
if(-1.0>input[0][2]/sin(output[4])||
   input[0][2]/sin(output[4])>1.0) {
    if(input[0][2]/sin(output[4])>1.0) {
        {output[3]=0.0;}
    }
    else
        {output[3]=PI;}
}
else
    {output[3]=acos(input[0][2]/sin(output[4]));}

/* Find a new pitch angle */
if(-1.0>input[2][0]/sin(output[4])||
   input[2][0]/sin(output[4])>1.0) {
    if(input[2][0]/sin(output[4])>1.0) {
        {output[5]=0.0;}
    }
    else
        {output[5]=PI;}
}
else
    {output[5]=acos(-input[2][0]/sin(output[4]));}
if(fabs(output[3]) < 0.000001) output[3] = 0.0;
if(fabs(output[5]) < 0.000001) output[5] = 0.0;
}

/* If we get here we must be done */
end_of_angle_find_loop:

}

/* Finally check for the 180, -pitch, -180 condition and correct
before exiting. */
if(output[3]==output[5]) {
    if((fabs(fabs(output[3])-PI)<0.000001)) {
        output[3] = 0.0;
        output[5] = 0.0;
        output[4] = -output[4];
    }
}

/* All done */
}

```

