SOFTWARE FOR ANALYSIS OF THREE DIMENSIONAL STATISTICAL TOLERANCE PROPAGATION IN ASSEMBLIES USING CLOSED-FORM MATRIX TRANSFORMS

by

Marek J. Jastrzebski
B. S. Mech. Eng., University of Michigan
(1988)

Submitted to the Department of Mechanical Engineering in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
January 1991

© Marek J. Jastrzebski, 1991
The author hereby grants to M.I.T. and the Charles Stark Draper Laboratory, Inc. permission to reproduce and to distribute copies of this thesis document in whole or in part.

Signature of Author ____________________________
Department of Mechanical Engineering
January 29, 1991

Certified by ____________________________ Daniel E. Whitney, PhD
Thesis Supervisor

Accepted by ____________________________ Ain A. Sonin, Chairman
Departmental Graduate Committee

MIT LIBRARIES
SEP 3 1991
BARKER

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
APR 26 1991
LIBRARIES
DISCLAIMER

This research is supported by the National Science Foundation. It represents the opinions of the author only, and is offered for the stimulation and exchange of ideas. It does not represent the policies of the National Science Foundation, the Charles Stark Draper Laboratory, or the Massachusetts Institute of Technology.
SOFTWARE FOR ANALYSIS OF THREE DIMENSIONAL STATISTICAL TOLERANCE PROPAGATION IN ASSEMBLIES USING CLOSED-FORM MATRIX TRANSFORMS

by

Marek J. Jastrzebski

Submitted to the Department of Mechanical Engineering on January 29, 1991 in partial fulfillment of the requirements for the Degree of Master of Science at the Massachusetts Institute of Technology

ABSTRACT

If a mechanical part is to be a component of an assembly, the dimensions and tolerances of the part have to be specified and adhered to in the fabrication process. Because of the inherent dimensional uncertainty, the tolerances accumulate in the assemblies; which accumulation may have negative consequences on the final product properties. The tolerance stack-up in assemblies may prevent successful mating of component sub-assemblies; or it may result in the unacceptable final tolerance range of the completed assembly. A design tool which helps estimate probability of successful assembly operation, or helps predict the final tolerance variability could be of great benefit at the assembly design stage.

In the spirit of the concurrent engineering principle, the design process should address (among other issues) the realistic capabilities and constraints of a manufacturing environment. The author hopes that this thesis can alleviate some of the problems that a designer could encounter in the process of tolerance propagation analysis in assemblies, and help him/her make conscious decisions regarding tolerance allocations.

In this work, I present a new method of statistical tolerance propagation analysis, which is based on a closed-form matrix transform solution. In the past, tolerance analysis has been performed with the help of Monte Carlo simulation. Because the Monte Carlo simulation is based on a random point generation routine, it is computationally expensive. The new method deals with continuous statistical variables in three dimensions. The computing time of the new method is comparable to the time of one iteration performed in Monte Carlo simulation.

Thesis Supervisor: Dr. Daniel E. Whitney, Lecturer, Mechanical Engineering

3
ACKNOWLEDGMENTS

This work would have never been possible if it were not for my thesis advisor Dr. Daniel E. Whitney. I would like to thank him for his educational, academic and motivational support. Working under his supervision was a great experience.

Mr. Steven Luby, Dr. Thomas De Fazio, Mr. Richard Gustavson, Draper Laboratory staff members, were always willing to help in my research, sharing their experience and wisdom; for which I am very grateful. Mr. Richard Metzinger was very helpful in solving some mysterious software problems that I occasionally ran into at the Laboratory. My appreciation goes to Ms. Ava Seekins for helping with any remaining problems and questions that I had.

Words of thanks go to the fellow students: Dan Baldwin, Russ Whipple for helping me to get started at the Draper Laboratory.

My wife, Elzbieta, was always patient and understanding when I often had to prolong my working hours at the Laboratory. Thanks, MiSiu (and I apologize for not washing the dishes).

This research was founded by the National Science Foundation under Grant No. DMC 87-15800.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>3</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>4</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>5</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>9</td>
</tr>
<tr>
<td>1.0 INTRODUCTION</td>
<td>11</td>
</tr>
<tr>
<td>1.1 Past work</td>
<td>12</td>
</tr>
<tr>
<td>1.2 Thesis objective</td>
<td>13</td>
</tr>
<tr>
<td>1.3 Current practices and proposed new approach to tolerance analysis</td>
<td>16</td>
</tr>
<tr>
<td>1.4 Motivation for derivation of analytical solution</td>
<td>16</td>
</tr>
<tr>
<td>1.5 Thesis organization</td>
<td>17</td>
</tr>
<tr>
<td>1.5.1 Derivation of analytical matrix transform method (Chapter 2)</td>
<td>17</td>
</tr>
<tr>
<td>1.5.2 Review of the current dimensioning and tolerancing practices (Chapter 3)</td>
<td>18</td>
</tr>
<tr>
<td>1.5.3 Description of the program major functions (Chapter 4)</td>
<td>18</td>
</tr>
<tr>
<td>1.5.4 Case study (Chapter 5)</td>
<td>18</td>
</tr>
<tr>
<td>1.5.5 Recommendation for future research and conclusions (Chapter 7)</td>
<td>19</td>
</tr>
<tr>
<td>2.0 DERIVATION OF ANALYTICAL MATRIX TRANSFORM METHOD</td>
<td>20</td>
</tr>
<tr>
<td>2.1 Derivation outline</td>
<td>20</td>
</tr>
</tbody>
</table>
2.2 Assumptions
2.3 Derivation summary
  2.3.1 Error transformation between $t-1$ and $t$th frame
  2.3.2 Total error transformation for a chain of $N$ frames
  2.3.3 Covariance matrices of the tri-variable normal
distribution for the dp and $\delta$ vectors
  2.3.4 Probability density ellipsoid
2.4 Approximation of a clearance between parts with
  uniform distribution
3.0 REVIEW OF THE CURRENT DIMENSIONING AND
TOLERANCING PRACTICES
  3.1 The issue of over- and under-constrained tolerance
  chains
  3.2 The analytical tolerance propagation method and the
  current ANSI Y14.5M-1982 standards
  3.3 How the matrix propagation method relates to the ANSI
  Y14.5M-1982 standards
    3.3.1 Position tolerance $\Phi$
    3.3.2 Perpendicularity tolerance $\perp$
    3.3.3 Angularity tolerance $\angle$
    3.3.4 Parallelism tolerance $\parallel$
4.0 DESCRIPTION OF THE PROGRAM MAJOR FUNCTIONS
  4.1 Setting the default values (default_data)
  4.2 User input data (get_user_data)
  4.3 The nominal position of the propagated frame
        (optimal_position)
4.4 Assign the values to the variance matrices (V_matrices) 51
4.5 Computation of the R, p, m matrices
(all_Rs_ps_and_ms) 52
4.6 Computation of the W matrices (W_matrices) 52
4.7 Covariance matrix of the position vector (Vp_matrix) 52
4.8 Covariance matrix of the rotation vector (Vdel_matrix) 53
4.9 Eigenvalues of the position covariance matrix
(eigenvalues) 53
4.10 Approximate convolution function
(convolve_uniform_and_normal) 53
4.11 Two dimensional ellipsoid case (ellipse_2D_case) 54
4.12 Three dimensional ellipsoid case (inverse,
base_orientation, ellipse_coefficients) 55
4.13 Graphical display of the results
(draw_ellipses_and_base) 57
4.14 Monte Carlo simulation (monte_carlo) 58

5.0 CASE STUDY 59
5.1 How the transformation frames are assigned 61
5.2 The frames coordinates 66
5.3 Interpretation of the results 73
5.4 Practical conclusions for the first solution method 76
5.5 Alternative formulation of the same problem 77
5.6 Practical conclusions for the second solution method 78

6.0 RECOMMENDATIONS FOR FUTURE RESEARCH 83
6.1 Data representation 83
6.2 Influence of tolerance stack-up on assembly sequence
choice and vice versa

6.3 Stability issues related to tolerance accumulation

6.4 Notes on predicting different probability density functions for tolerance distributions

6.5 Solid modeler application

7.0 CONCLUSIONS

8.0 REFERENCES

9.0 APPENDICES

Appendix A. Derivation of m vectors, for a simplified case

Appendix B. Choosing appropriate probability distribution model for clearances and their convolutions

Appendix C. Complete listing of the software code
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Six degrees of freedom associated with a three dimensional body</td>
<td>12</td>
</tr>
<tr>
<td>1-2</td>
<td>Free body transformation in space</td>
<td>14</td>
</tr>
<tr>
<td>1-3</td>
<td>Chain of transformations through a three-part assembly</td>
<td>15</td>
</tr>
<tr>
<td>2-1</td>
<td>The effect of small error transformation $dA_i$ on the $A_i$ matrix</td>
<td>26</td>
</tr>
<tr>
<td>2-2</td>
<td>It is equally likely to find the peg anywhere within the clearance range</td>
<td>36</td>
</tr>
<tr>
<td>3-1</td>
<td>Exactly dimensioned and exactly tolerated scheme</td>
<td>38</td>
</tr>
<tr>
<td>3-2</td>
<td>Unreachable features</td>
<td>39</td>
</tr>
<tr>
<td>3-3</td>
<td>Over-dimensional and over-toleranced scheme</td>
<td>39</td>
</tr>
<tr>
<td>3-4</td>
<td>The angularity tolerance controls both flatness and the angular deviation from nominal</td>
<td>43</td>
</tr>
<tr>
<td>3-5</td>
<td>The position tolerance of the hole axis is specified as cylindrical zone</td>
<td>45</td>
</tr>
<tr>
<td>3-6</td>
<td>The perpendicularly tolerance specifies that the shaft's axis has to lie entirely within the cylindrical zone</td>
<td>46</td>
</tr>
<tr>
<td>3-7</td>
<td>The upper surface of the block has to lie between two parallel planes</td>
<td>47</td>
</tr>
<tr>
<td>4-1</td>
<td>The graphical representation of the angle $\beta$</td>
<td>56</td>
</tr>
<tr>
<td>5-1</td>
<td>The precision assembly case study</td>
<td>60</td>
</tr>
<tr>
<td>5-2</td>
<td>End cover with the significant dimensions and tolerances</td>
<td>61</td>
</tr>
<tr>
<td>5-3</td>
<td>Core with the significant dimensions and tolerances</td>
<td>62</td>
</tr>
</tbody>
</table>
5-4 Housing with the significant dimensions and tolerances 63
5-5 The chain of five transformations through the end cover and the stem 64
5-6 The chain of three transformations through the end cover and the housing 65
5-7 Contours of the ellipsoid of density for the first chain of transformations 67
5-8 Monte Carlo simulation for the first chain of transformations 68
5-9 Contours of the ellipsoid of density for the second chain of transformations 69
5-10 Monte Carlo simulation for the second chain of transformations 70
5-11 Two overlapping ellipsoids of density represent the position uncertainty of the frames F_{15} and F_{23} 76
5-12 Contours of the ellipsoid of density for the third chain of transformations 80
5-13 Monte Carlo simulation for the third chain of transformations 81
5-14 Contours of the ellipsoid of density represent the position uncertainty of the frame F_{38} in the horizontal plane 82
6-1 The effect of gravitation on the tolerance stack-up 87
1.0 INTRODUCTION

Technical drawings of mechanical parts play crucial role to the engineering profession. The ability to interpret these drawings unambiguously ensures manufacture, interchangeability, subcontracting, and standardizing. Tolerance specifications are inseparable component of an engineering drawing nowadays. In the year 1920, drawings with dimensions were common; however, the tolerance specifications were not. Only in the 1940s, in England, was the currently used method of "true position tolerancing" developed (Nevins, Whitney, et al. [23]). The science of dimensioning and tolerancing is still under development.

The objective of this thesis is to present the development of the analytical (closed-form) matrix model for tolerance propagation analysis in three dimensional assemblies. The developed matrix model is generic and accounts for nonlinearities during transformations of bodies in space.

Kinematic position of a body in three dimensional space is fully described by six degrees of freedom: three positional coordinates \((X, Y, Z)\) and three rotational coordinates \((\theta_x, \theta_y, \theta_z)\), figure 1-1.

There are ranges of uncertainty associated with the six degrees of freedom. It is assumed that the uncertainties are normally distributed around the nominal value of a position or an angle. The exception of uniform distribution is allowed in the \(X\) and \(Y\) directions to account for clearances between two parts (more detailed discussion will follow).
1.1 Past work

The majority of the past research in the area of tolerancing is based on the Monte-Carlo simulation. The approach is to generate a large number of discrete assembly instances, see Turner [29]. The algorithm randomly selects a point within the in-tolerance region with the prescribed statistical distribution. The corresponding assembly instance is simulated and the statistics are collected. This procedure is continued until a given confidence level is achieved to make conclusions about the resultant probability density distribution.

The recent research has been focused on the development of solid modeling applications to issues related to tolerancing. For reference, see Turner [29] and [30], Pandit and Starkey [24], Roy, et al.[26]. Solid modeler application is the logical and necessary next step in overcoming the current representation problems of the real world three dimensional parts and assemblies (which traditionally have been represented in two dimensions in the past). The majority
of the methods known to us use the Monte-Carlo simulation to generate the discrete, random instances of part or assembly. The problem with the Monte-Carlo algorithm, however, is its inherent slowness, especially when coupled with a solid modeler.

1.2 Thesis objective

This work intends to set the framework for fast, closed-form matrix solution with continuously distributed random variables; which could eliminate the need for computationally expensive random number iterations. The inspiration for this research came from the work done by Veitschegger and Wu [31], in which a mathematical model for evaluation of robot arm accuracy due to kinematic errors is developed. In the case of a robot manipulator, its position and orientation can be fully described by means of five degrees of freedom. Furthermore, the Denavit - Hartenberg notation [9] is conveniently applicable to a robot arm.

When considering the case of a free body transformation in space, six degrees of freedom (three translations and three rotations) are necessary to fully describe its transformations in space, figure 1-2. Any transformation in space can be represented by a series of four homogeneous 4x4 transformation matrices: linear translation matrix and rotation matrices around X, Y, Z axes (see section 2.2). The Denavit - Hartenberg notation [9] relates to transformations with five degrees of freedom; therefore, it is not applicable to the case of a free body transformation.
Figure 1-2. Free body transformation in space. $F_0$ is the reference frame, $F_1$ is the transformed frame. In general case, three translations and three rotations describe a transformation in space.

The general idea of the free body transformation in the three dimensional space can be extended to tolerance propagation in assemblies. We can assign a coordinate frame to each part, or to a feature on a part. Then, these frames can be viewed as propagated in space by the specified nominal position coordinates and orientation angles with respect to an assumed base reference frame, see figure 1-3.

Specified tolerances on a part or its features can be interpreted as uncertainties of position and orientation. Therefore, knowing the
Figure 1-3. Chain of transformations through a three-part assembly. $F_0$ is the reference frame, attached to an arbitrary point on the first part. The dashed line connects the assigned transformed frames.

Uncertainties associated with the nominal position and orientation of each propagated frame, and the statistical distribution of the uncertainties; the uncertainty of the position and the orientation of the last propagated frame can be obtained. This uncertainty is the final (accumulated) tolerance distribution of the specified point in the assembly.
1.3 Current practices and proposed new approach to tolerance propagation analysis

The current methods for tolerance propagation analysis are based either on the Monte-Carlo simulation or on the assumption of "worst-case" tolerance distributions. Both approaches have serious drawbacks. The Monte-Carlo method, although reliable, is computationally intensive. The "worst-case" method invariably leads to tighter than necessary part tolerances and more expensive manufacturing. The Selected Panel on Research Opportunities in Mechanical Tolerancing during the International Workshop held in 1988 in Orlando, Florida [28] recommended that vector-based tolerance models be developed, non-linearities addressed, and the computational efficiency improved. This work is an attempt to address these issues and to provide a solution.

Recently, I have developed a mathematical model for three dimensional vector tolerancing. Based on the assumptions listed in the section 2.4, the model helps estimate the probability of finding a point within a specified region. The model is capable of propagating specified linear and angular tolerances through a series of given transformations in space. Since the angular tolerances can significantly affect the actual position of a point in space, the influence of angular tolerances has been taken into account.

1.4 Motivation for derivation of analytical solution

Three dimensional tolerance distribution and tolerance
propagation are of extreme importance in the process of parts design and multi-part assemblies design. Production of tightly tolerated parts is expensive and not always necessary. In the engineering and scientific community much work has been done on the subject of tolerance allocation. However, it should be obvious that a designer should have an effective analytical tool if he/she could perform tolerance synthesis (allocation) effectively. Such a tool should be able to statistically estimate the resulting tolerance distribution at any point in a chain of interrelated parts in an assembly. The effect of angular tolerances (errors) on linear position tolerances is important and should be taken into account.

A mathematical model able to predict tolerance propagation in multi-part assemblies should be of great benefit to engineering and scientific community. It can improve our perception in the research on optimal tolerance allocation. By changing desired tolerance ranges, sensitivity analysis should be easy to perform. The model could be helpful if an automated assembly method is considered, e.g. for assembly sequence selection or equipment choice. The model can also be applied to investigate robotic arm errors and to investigate methods for the errors reduction.

1.5 Thesis organization

1.5.1 Derivation of analytical matrix transform method (Chapter 2).

I describe the main flow of derivation that led to the development
of the position and orientation variance matrices for the \( N \)th frame in
the chain of transformations in space. The position or orientation of
the frame in space is represented as a probability density ellipsoid.
Simplified method of convolving normal and uniform distributions is
presented.

1.5.2 Review of the current dimensioning and tolerancing practices
(Chapter 3)

I discuss some of the issues related to the science of
dimensioning and tolerancing in three dimensional space. Over- and
under constraint issues are addressed. The ANSI Y14.5M-1982
standards are discussed in the context of their applicability to the
closed form tolerance analysis method.

1.5.3 Description of the program major functions (Chapter 4)

Here, I briefly describe the computer program. All the major
functions are listed and described in separate sections. The code was
written in C programming language on Sun 3/60 minicomputer.

1.5.4 Case study (Chapter 5)

An application of the program is presented in this case study of a
three-part precision assembly. Two different ways of setting up the
transformation chains are shown, and the results are analyzed.
1.5.5 Recommendation for future research (Chapter 6) and conclusions (Chapter 7)

I elaborate on the possible directions for the research continuation in the area. I suggest that data base be developed to improve the effectiveness of solving tolerance analysis problems. Influence of assembly sequence on tolerance accumulation (and vice versa) should be investigated. I recommend continuing research in the area of different tolerance distributions (other than normal or uniform). Solid modeler should be coupled with the tolerance analysis software.
2.0 DERIVATION OF ANALYTICAL MATRIX TRANSFORM METHOD

This chapter presents the summary of the derivation that led to the closed-form analytical solution to the problem of tolerance propagation in three dimensions. It is assumed that the reader possesses the basic knowledge of linear algebra, statistics and differential calculus. All the matrices and vectors in the following derivation are shown in abbreviated symbolic form (their expanded forms, with some simplifying assumptions, are presented in the Appendix A).

2.1 Derivation outline

In this work, tri-variable normal probability density distribution of tolerances has been assumed. However, the option of uniform distributions exists for X and Y linear directions to account for clearances between parts. Subsequently, I will elaborate on the mathematical derivation of the density function. The geometric interpretation of the resulting distribution of a point position in space is an ellipsoid with the highest density at the nominal position of the point. Due to the angular errors, the axes of the propagated ellipsoid in space might not be parallel to the directions of the base coordinate frame axes. However, all the information necessary to describe the ellipsoid in space is contained in the 3x3 covariance matrix of the resulting tri-variable probability density function. Therefore, after the covariance matrix of the function is determined, the matrix
eigenvalues can be computed. The square roots of the obtained three eigenvalues are standard deviations in the principal directions of the ellipsoid. The orientation of the probability density ellipsoid in space is determined by the eigenvectors of the covariance matrix. The eigenvectors are aligned with the principal axes of the ellipsoid.

Next, the assumptions and the summary of the derivation is presented.

2.2 Assumptions

Because of the inherent complexity related to the issues of three dimensional tolerancing, it was necessary to make simplifying assumptions.

1. The propagation of tolerances in 3-D space is treated as propagation of errors during transformations of Cartesian frames. Cartesian frames are attached to a specified point on the surface of a part. There exist 6 degrees of freedom for dimensional transformations of frames or their small errors. The capital letters $X, Y, Z$ signify linear translations; and the lower case letters $x, y, z$ represent rotation angles around respective axes, see figure 1-1.

2. The order of rotation is important and consistency must be maintained through the process of $N$ transformations. In this work, the following order is assumed and maintained.
\[ T = T_{\text{Trans}} T_{\text{RotX}} T_{\text{RotY}} T_{\text{RotZ}} \]

where \( T \) is a general form of a transformation matrix for transformation from one point in Cartesian space to the next. The transformation matrices are shown below for reference.

\[
T_{\text{Trans}} = \begin{pmatrix}
1 & 0 & 0 & X \\
0 & 1 & 0 & Y \\
0 & 0 & 1 & Z \\
0 & 0 & 0 & 1
\end{pmatrix},
\]

where the capital letters \( X, Y, Z \) signify the linear displacement vector coordinates.

\[
T_{\text{RotX}} = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & \cos[x] & -\sin[x] & 0 \\
0 & \sin[x] & \cos[x] & 0 \\
0 & 0 & 0 & 1
\end{pmatrix},
\]

The subscript RotX means that rotation is performed around the X axis. The lower case letter \( x \) signifies the rotation angle around the X axis.

\[
T_{\text{RotY}} = \begin{pmatrix}
\cos[y] & 0 & \sin[y] & 0 \\
0 & 1 & 0 & 0 \\
-\sin[y] & 0 & \cos[y] & 0 \\
0 & 0 & 0 & 1
\end{pmatrix},
\]

The subscript RotY means that rotation is performed around the Y axis. The lower case letter \( y \) signifies the rotation angle around
the Y axis

\[
T_{RotZ} = \begin{pmatrix}
\cos[z] & -\sin[z] & 0 & 0 \\
\sin[z] & \cos[z] & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

The subscript RotZ means that rotation is performed around the Z axis. The lower case letter \( z \) signifies the rotation angle around the Z axis.

3. I assume normal trivariable (Gaussian) distribution of error. Propagation of normal density ellipsoid over N frames results in the output normal density ellipsoid (possibly of changed size, density and orientation in space).

4. The specified, design tolerance range for any single dimensional quantity (length or angle) is equivalent to 6\( \sigma \) (\( \sigma \) = standard deviation). Therefore, any specified dimension \( a \) with specified tolerance range \( \Delta a \) (\( \Delta a = 3\sigma \)) will be included in the interval (\( a - \Delta a, a + \Delta a \)) with the probability of approximately 99.7% (in one dimensional space).

5. The probabilistic events are independent. Any single transformation of any frame does not, in any way, affect transformation of the next frame (within their prescribed variances limits).
2.3 Derivation summary

This section shows only the main line of the derivation. The matrices and the vectors are not shown in their expanded form. For illustration, the vectors are derived in the Appendix A, with some simplifying assumptions.

The intermediate objective of this derivation is the development of the tri-variable normal density function for the Nth transformation frame. The ultimate objective is the description of the location, the orientation and the axes' lengths of the probability density ellipsoid. The density function argument is the 3x1 differential translation vector dp. The function describes the distribution of the Nth frame origin position, which is the last frame in the chain of transformations, in the three dimensional space:

\[ f(dp) = (2\pi)^{-\frac{3}{2}} |V_p|^{-\frac{1}{2}} \exp\left(-0.5 [(dp)^\top V_p^{-1}(dp)]\right). \] (1)

\( V_p \) is a 3x3 covariance matrix of the dp vector and the symbol \( \top \) means vector transpose. The eigenvalues of the \( V_p \) matrix are variances in the principal directions of the probability density ellipsoid. The directions of the eigenvectors of the matrix determine the ellipsoid orientation in space.

2.3.1 Error transformation between i-1 and ith frame

The \( i \) index denotes a frame number in a chain of \( N \) transformations. A transformation between \( i-1 \) and the \( i \)th frame is
defined as $A_i$. Since the order of transformations is important, we assume that the translation occurs first, followed by the rotations with respect to X, Y, Z axes respectively. Therefore,

$$A_i = (A_i)_{\text{Transl}}(A_i)_{\text{Rot}},$$

where

$$(A_i)_{\text{Rot}} = (A_i)_{\text{Rot}X}(A_i)_{\text{Rot}Y}(A_i)_{\text{Rot}Z},$$

and the $A_i$ transform matrices are:

$$(A_i)_{\text{Transl}} = \begin{bmatrix} 1 & 0 & 0 & X_i \\ 0 & 1 & 0 & Y_i \\ 0 & 0 & 1 & Z_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$(A_i)_{\text{Rot}X} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos[x_i] & -\sin[x_i] & 0 \\ 0 & \sin[x_i] & \cos[x_i] & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$(A_i)_{\text{Rot}Y} = \begin{bmatrix} \cos[y_i] & 0 & \sin[y_i] & 0 \\ 0 & 1 & 0 & 0 \\ -\sin[y_i] & 0 & \cos[y_i] & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$(A_i)_{\text{Rot}Z} = \begin{bmatrix} \cos[z_i] & -\sin[z_i] & 0 & 0 \\ \sin[z_i] & \cos[z_i] & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where $X_i$, $Y_i$, $Z_i$ are translations in the respective directions; and $x_i$, $y_i$, $z_i$ are rotations around respective axes. The $i$ index refers to the $i$th
frame transformation.

Therefore, the general form of the $A_i$ transform can be written in the following simplified homogeneous matrix form:

$$A_i = \begin{pmatrix} R_i & p_i \\ 0 & 1 \end{pmatrix},$$

(2)

where $R_i$ is a 3x3 rotation matrix, and $p_i$ is a 3x1 translation vector, both taken with respect to the $i$-1 frame.

The actual position of the $N$th frame is determined by the transform matrix $T_N^c$:

$$T_N^c = T_N + dT_N,$$

(3)

where $T_N$ is the nominal transform matrix (if no errors were present). $dT_N$ represents the total differential change transform matrix of the $N$th frame position and orientation. The $dT_N$ matrix can be written in the following form:

$$dT_N = (\delta T)T_N,$$

where $\delta T$ can be defined as the differential error matrix of the total transformation chain. For any transformation between $i$ and $i$-1 frame, it is true that (figure 2-1):

$$A_i + dA_i = (I + \delta A_i)A_i.$$

Figure 2-1. The effect of small error transformation $dA_i$ on the $A_i$ matrix.

$\delta A_i$ is defined as the 4x4 homogeneous error matrix transform with
respect to the \(t-1\) frame. And therefore, the differential change matrix 
\(dA_t\) (4x4) for the \(i\)th frame is:

\[
dA_i = (\delta A_i)A_i. \tag{4}
\]

Because \(\delta A_i\) transform matrix describes an error between two frames positions and orientations, it is important to find its general form. \(\delta A_i\) can be represented in the following form of a homogeneous matrix:

\[
\delta A_i = \begin{pmatrix}
\delta R_i & d_i \\
0 & 1
\end{pmatrix},
\]

where \(\delta R_i\) is the 3x3 error rotation transform matrix with respect to the \(t-1\) transformation. Similarly, \(d_i\) is the 3x1 error translation vector. The following flow of the derivation in this section aims at finding the general form of \(d_i\) error translation vector, and the real eigenvector of the \(\delta R_i\) error rotation transform matrix (defined as \(\delta_i\)).

The equation (3) can be rewritten as

\[
T_N + dT_N = (A_1 + dA_1)(A_2 + dA_2)...(A_N + dA_N),
\]

or

\[
T_N + dT_N = \prod_{i=1}^{N} (A_i + dA_i)
\]

The differential change matrix \(dA_t\) may be obtained by differentiating the matrix \(A_t\) with respect to the three scalar translations \((X_t, Y_t, Z_t)\), and the three scalar rotations \((x_i, y_i, z_i)\).

\[
dA_i = \left[ \frac{\partial A_i}{\partial X_t} \right] dX_i + \left[ \frac{\partial A_i}{\partial Y_t} \right] dY_i + \left[ \frac{\partial A_i}{\partial Z_t} \right] dZ_i + \left[ \frac{\partial A_i}{\partial x_i} \right] dx_i + \left[ \frac{\partial A_i}{\partial y_i} \right] dy_i + \left[ \frac{\partial A_i}{\partial z_i} \right] dz_i.
\]

Let us rewrite the above equation in a different form:

\[
dA_i = ([D_i]_{\text{Trans}X} dX_i + [D_i]_{\text{Trans}Y} dY_i + [D_i]_{\text{Trans}Z} dZ_i + [D_i]_{\text{Rot}X} dx_i + [D_i]_{\text{Rot}Y} dy_i + [D_i]_{\text{Rot}Z} dz_i) A_i
\]

\[
\tag{5}
\]
The \((D_i)_{\text{Trans}X}\) 4x4 matrix is defined as:

\[
(D_i)_{\text{Trans}X} = \left( \frac{\partial}{\partial X_i} A_i \right) A_i^{-1}
\]

and the remaining five 4x4 \(D_i\) matrices are computed analogously.

Comparing (4) and (5) we can write the \(\delta A_i\) transform in the following form:

\[
\delta A_i = (D_i)_{\text{Trans}X} dX_i + (D_i)_{\text{Trans}Y} dY_i + (D_i)_{\text{Trans}Z} dZ_i + (D_i)_{\text{Rot}X} dX_i + (D_i)_{\text{Rot}Y} dY_i + (D_i)_{\text{Rot}Z} dZ_i
\]

To find the general form of the error transform matrix \(\delta A_i\), first we must derive the inverse of matrix \(A_i\) and the partial derivatives of \(A_i\) with respect to \(X_i, Y_i, Z_i, x_i, y_i, z_i\). These tasks require a rather substantial amount of patience, and the detailed derivation is not shown on these pages. The general forms of \(A_i\) inverse and the six \(D_i\) matrices (with simplifying assumptions) are shown in the Appendix A.

The 3x1 \(d_i\) vector is the last column of the \(\delta A_i\) matrix (excluding the corner element 1). The differential error notations \((d)\) can be replaced by the actual (small) errors \((\Delta)\) occurring during linear or angular transformations. Therefore, \(d_i\) can be written in the following convenient form, with contributing errors isolated:

\[
d_i = m_{1i} \Delta X_i + m_{2i} \Delta Y_i + m_{3i} \Delta Z_i + m_{4i} \Delta x_i + m_{5i} \Delta y_i + m_{6i} \Delta z_i.
\]

\(m_{1i}, m_{2i}, m_{3i}, m_{4i}, m_{5i}, m_{6i}\) are all 3x1 vectors contributing to the transformation translation errors occurring between the \(i\)-1 and the \(i\)th frames (their expanded forms are shown in the Appendix A). Notice that not only linear errors \(\Delta X_i, \Delta Y_i, \Delta Z_i\), but also the angular errors \(\Delta x_i, \Delta y_i, \Delta z_i\), contribute to the final position error of the \(i\)th frame.

The 3x1 eigenvector of the \(\delta R_i\) matrix can be found and denoted as
\( \delta_i \): The error rotation vector \( \delta_i \) can be written in the following form:
\[
\delta_i = m_{7i} \Delta x_i + m_{8i} \Delta y_i + m_{9i} \Delta z_i
\]
The \( m_{7i}, m_{8i}, m_{9i} \) are 3x1 vectors contributing to the angular error transformation occurring between the \( i-1 \) and the \( i \)th frame. Notice, however, that only the angular errors \( (\Delta x_i, \Delta y_i, \Delta z_i) \) affect the \( \delta_i \) vector.

### 2.3.2 Total error transformation for a chain of \( N \) frames

It can be shown (Veitschegger, Wu [31]) that the combined vectors \( d \) and \( \delta \) associated with the total transformation of \( N \) frames, can be computed as follows:

\[
d = \sum_{i=1}^{N} (R_{i-1} \delta_i + p_{i-1} \times [R_{i-1} \delta_i])
\]

\( \delta = \sum_{i=1}^{N} R_{i-1} \delta_i \) \hspace{1cm} (6)

where \( R_{i-1} \) is 3x3 rotation matrix and \( p_{i-1} \) is 3x1 translation vector, both obtained from the \( A_{i-1} \) transform (2).

The \( d \) and \( \delta \) vectors can be represented in the following matrix form:

\[
\begin{pmatrix}
\mathbf{d} \\
\mathbf{\delta}
\end{pmatrix} = 
\begin{pmatrix}
W_1 \\
0
\end{pmatrix} \Delta X + 
\begin{pmatrix}
W_2 \\
0
\end{pmatrix} \Delta Y + 
\begin{pmatrix}
W_3 \\
0
\end{pmatrix} \Delta Z + 
\begin{pmatrix}
W_4 \\
W_5
\end{pmatrix} \Delta x + 
\begin{pmatrix}
W_6 \\
W_7
\end{pmatrix} \Delta y + 
\begin{pmatrix}
W_8 \\
W_9
\end{pmatrix} \Delta z
\]

\( \Delta X = (\Delta X_1, \ldots, \Delta X_N)^t \)
\( \Delta Y = (\Delta Y_1, \ldots, \Delta Y_N)^t \)
\( \Delta Z = (\Delta Z_1, \ldots, \Delta Z_N)^t \)
\( \Delta x = (\Delta x_1, \ldots, \Delta x_N)^t \)
\( \Delta y = (\Delta y_1, \ldots, \Delta y_N)^t \)
\[ \Delta z = (\Delta z_1, \ldots, \Delta z_N)^+ \]

and \( ^+ \) means vector transpose.

The \( W \) vectors are all \( 3 \times N \) vectors and (from (6), (7) and (8)) their \( i \)th columns can be written:

\[
\begin{align*}
W_{1i} &= R_i - 1m_{1i} \\
W_{2i} &= R_i - 1m_{2i} \\
W_{3i} &= R_i - 1m_{3i} \\
W_{4i} &= R_i - 1m_{4i} + p_i - 1 \times (R_i - 1m_{7i}) \\
W_{5i} &= R_i - 1m_{7i} \\
W_{6i} &= R_i - 1m_{5i} + p_i - 1 \times (R_i - 1m_{8i}) \\
W_{7i} &= R_i - 1m_{8i} \\
W_{8i} &= R_i - 1m_{6i} + p_i - 1 \times (R_i - 1m_{9i}) \\
W_{9i} &= R_i - 1m_{9i}.
\end{align*}
\]

Because the argument of the probability density function (1) is the \( 3 \times 1 \) differential translation vector \( dp \), we should find its general form.

The actual (correct) position of the last \( N \)th frame is described by the \( 3 \times 1 \) vector \( pt_N^c \):

\[ pt_N^c = pt_N + dp, \]

where the \( pt_N \) vector describes the nominal position (no errors present) of the \( N \)th frame. The vector \( dp \) can be written as (Veitschegger, Wu [31]):

\[ dp = \delta \times pt_N + d, \quad (9) \]

where \( d \) is obtained from (6) and \( \delta \) from (7). The above equation shows clearly that the total error of any chain of transformations in space depends both on linear and angular errors (tolerances).

By substituting (8) into (9), the \( dp \) vector can be rewritten in the
form:
\[ dp = W_1 \Delta X + W_2 \Delta Y + W_3 \Delta Z + W_{10} \Delta x + W_{11} \Delta y + W_{12} \Delta z, \]
where again \( W_{10}, \ W_{11}, \ W_{12} \) are all 3xN matrices, and their \( i \)th columns are:
\[ W_{10i} = W_{5i} \times pt_N + W_{4i} \]
\[ W_{11i} = W_{7i} \times pt_N + W_{6i} \]
\[ W_{12i} = W_{9i} \times pt_N + W_{8i}. \]

2.3.3 Covariance matrices of the trivariable normal distribution for the dp and \( \delta \) vectors

\( \Delta X, \ \Delta Y, \ \Delta Z, \ \Delta x, \ \Delta y, \ \Delta z \) are Nx1 vectors of designer specified tolerance ranges. For this paper, it has been assumed that \( \Delta X, \ \Delta Y, \ \Delta Z, \ \Delta x, \ \Delta y, \ \Delta z \) are six independent, N-variables with zero means and normal distribution with the following properties:

- \( V_X = \) variance of \( \Delta X = \) a NxN diagonal matrix with components (\( \text{var}_{X1}, \ldots, \text{var}_{XN} \)), where \( \text{var}_{X1} = \) variance of \( \Delta X_1 \)
- \( V_Y = \) variance of \( \Delta Y = \) a NxN diagonal matrix with components (\( \text{var}_{Y1}, \ldots, \text{var}_{YN} \)), where \( \text{var}_{Y1} = \) variance of \( \Delta Y_1 \)
- \( V_Z = \) variance of \( \Delta Z = \) a NxN diagonal matrix with components (\( \text{var}_{Z1}, \ldots, \text{var}_{ZN} \)), where \( \text{var}_{Z1} = \) variance of \( \Delta Z_1 \)
- \( V_x = \) variance of \( \Delta x = \) a NxN diagonal matrix with components (\( \text{var}_{x1}, \ldots, \text{var}_{xN} \)), where \( \text{var}_{x1} = \) variance of \( \Delta x_1 \)
- \( V_y = \) variance of \( \Delta y = \) a NxN diagonal matrix with components (\( \text{var}_{y1}, \ldots, \text{var}_{yN} \)), where \( \text{var}_{y1} = \) variance of \( \Delta y_1 \)
- \( V_z = \) variance of \( \Delta z = \) a NxN diagonal matrix with components

31
\((\text{var}_{z1}, \ldots, \text{var}_{zN})\), where \(\text{var}_{z1} = \text{variance of } \Delta z_{\ell}\)

The 3x3 covariance matrix \(V_p\) of the dp differential translation vector can be computed from the following formula:

\[
V_p = W_1 V_x W_1^\dagger + W_2 V_y W_2^\dagger + W_3 V_z W_3^\dagger + W_{10} V_x W_{10}^\dagger + W_{11} V_y W_{11}^\dagger + W_{12} V_z W_{12}^\dagger
\]

Similarly, the expression for the 3x3 covariance matrix of the total error rotation vector \(\delta\) is:

\[
V_\delta = W_5 V_x W_5^\dagger + W_7 V_y W_7^\dagger + W_9 V_z W_9^\dagger
\]

2.3.4 Probability density ellipsoid

Knowing the \(V_p\) covariance matrix, we obtain the ellipsoid of probability density in space, which is fully described by the \(V_p\) matrix:

\[
\begin{pmatrix}
X \\
Y \\
Z
\end{pmatrix}^\dagger V_p^{-1} \begin{pmatrix}
X \\
Y \\
Z
\end{pmatrix} = \ell^2
\]

There is an infinite number of orientations in space that the ellipsoid can assume. We would like to transform it, so that we can use the ellipsoid axes as the coordinate axes. In other words, we need to diagonalize the matrix \(V_p\). The diagonalization can be done by finding the eigenvalues of the matrix. The characteristic equation is:

\[
|V_p - \lambda I| = 0,
\]

where \(\lambda\) means an eigenvalue of \(V_p\), and \(I\) is a 3x3 identity matrix. The three eigenvalues, obtained from the characteristic equation, are equal to variances in the principal directions of the ellipsoid:

\[
\lambda_1 = \sigma_1^2 \\
\lambda_2 = \sigma_2^2
\]
\[ \lambda_3 = \sigma_3^2, \]

where \( \sigma_1, \sigma_2, \sigma_3 \) are standard deviations in the principal directions. Therefore, after transforming the orientation of the principal axes of the ellipsoid to the position of the base coordinate frame, we obtain the following equation:

\[ \frac{y_1^2}{\sigma_1^2} + \frac{y_2^2}{\sigma_2^2} + \frac{y_3^2}{\sigma_3^2} = l^2, \]

where \( l \) is a constant.

By another transformation,

\[ z_i = \frac{y_i}{\sigma}, \]

we transform the equation of the ellipsoid to a sphere with a radius \( r = l \)

\[ z_1^2 + z_2^2 + z_3^2 = r^2, \]

The probability of finding \( z \) inside the hypersphere is (Bryson, Ho [4]):

\[ \int \int \int \frac{1}{(2\pi)^{\frac{3}{2}}} \exp\left( -\frac{1}{2} \left[ z_1^2 + z_2^2 + z_3^2 \right] \right) dz_1 dz_2 dz_3, \]

where the integration is performed over the volume of the hypersphere. Therefore, the probability of finding a point inside the ellipsoid:

\[ \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}^\dagger V_p^{-1} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = l^2, \]

is the same as the probability of finding it within the hypersphere of radius \( r = l \).
\[
\frac{1}{3} \sqrt{\frac{2}{\pi}} \left( \int_0^l \exp \left[ -\frac{1}{2} r^2 \right] f[r] \, dr \right) \\
\]

or, in the solved form (Bryson, Ho[4]):

\[
\sqrt{\frac{2}{\pi}} \left( \int_0^l \exp \left[ -\frac{1}{2} r^2 \right] r^2 \, dr \right) = \text{erf} \left( \frac{l}{\sqrt{2}} \right) - \sqrt{\frac{2}{\pi}} \exp \left( -\frac{1}{2} l^2 \right),
\]

where \( \text{erf} \) signifies the error function and is defined by the following equation:

\[
\frac{1}{2} \text{erf} \left( \frac{x}{\sqrt{2}} \right) = \int_0^x \frac{1}{\sqrt{2\pi}} \exp \left( -\frac{1}{2} t^2 \right) dt
\]

In some cases, the three dimensional ellipsoid can degenerate to two dimensional ellipse. This occurs, for example, when the Z is the axis of assembly, and the tolerances are specified in X and Y linear directions only, and the angular tolerances are zero. It is useful to tabularize the values of probabilities for \( l = 1, 2, 3 \); for \( n = 1, 2, 3 \) dimensions. These values are often called the one-, two-, or three-sigma probabilities:

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>Number of ( \sigma )'s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( l = 1 )</td>
</tr>
<tr>
<td>( n = 1 )</td>
<td>0.683</td>
</tr>
<tr>
<td>( n = 2 )</td>
<td>0.394</td>
</tr>
<tr>
<td>( n = 3 )</td>
<td>0.200</td>
</tr>
</tbody>
</table>
2.4 Approximation of a clearance between parts with uniform distribution

As it often happens during assembly, the part (e.g., a peg) being inserted into a hole is of smaller dimensions (e.g., a peg diameter) than the hole. In the mating part theory, this instance is called the clearance fit. The other two insertion cases are: interference fit (the peg has larger diameter than the hole), and the transition fit (both the peg and the hole are of the same diameters). The uncertainty of location for the interference and the transition fits can be assumed to be normally distributed.

However, the uncertainty of the location of a peg within a hole cannot be reasonably approximated by a normal distribution. The insertion is often accomplished with a physical contact occurring between the two parts, irrespective of the automatic or manual choice for assembly method. Therefore, the safest and most conservative assumption is the uniform distribution for clearances between pegs and holes, figure 2-2.

By the Central Limit Theorem, the convolution of many (at least four) uniform distributions with the same means and variances is well approximated by the normal distribution with the variance equal to the sum of the component distributions' variances, see Montgomery [22]. The problem arises when the number of distributions is less than four and the variances are not the same. Obviously, this is the situation often encountered in assembly practices.

It is not worthwhile to develop a computer routine which would
Figure 2-2. It is equally likely to find the peg anywhere within the clearance range $\frac{D-d}{2}$.

set up the exact convolution integrals, and numerically integrate them for unlimited number of superimposed functions. At least, the effort involved in the development of such an accurate routine is not justified, bearing in mind that none of the probability density functions are known exactly. Also, the routine could slow down the program run, thwarting our primary objective of the algorithm efficiency and speed.

To accommodate the assumption of the uniform distribution being the best approximation for clearances, a series of tests was performed
to investigate the behavior of convolved functions.

First, the exact convolution integrations of up to four uniform density functions with varying ratios of their standard deviations was performed, and the resulting convolved functions obtained. Then, the convolution integration for uniform plus truncated (the total range = 6σ) normal distributions with varying ratios of σ's was performed, and their exact convolved functions obtained. The specific cases and the resultant graphs are shown in the Appendix B.

Next, simulation runs were performed with discrete points being distributed according to prescribed parameters. The randomly generated points were added up to simulate some convolution cases, as with the continuous functions before. To check if the distributions of the added discrete points can be confidently approximated by the normal or uniform distributions, the $\chi^2$ goodness-of-fit tests were performed for different cases. The results are summarized in two tables in the Appendix B. The $\chi^2$ test was run at $\alpha = 0.05$ significance level. For the approximation to $\chi^2$ distribution to be accurate, the number of the discrete points added together had to be at least 610.

The results of the test show that it is reasonably safe to approximate any superposition of uniform plus normal distributions with the resultant distribution either uniform or normal. The variance of the resultant distribution is always the sum of the component two distributions' variances. Based on the series of $\chi^2$ tests summarized in the Appendix B, a "threshold" value was determined to decide whether the uniform or normal distribution approximation is better for the convolved function.
DISCLAIMER OF QUALITY

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort to provide you with the best copy available. If you are dissatisfied with this product and find it unusable, please contact Document Services as soon as possible.

Thank you.

PAGE 38 IS MISSING FROM THE ORIGINAL ARCHIVAL THESIS
Figure 3-2. Unreachable features.

Figure 3-3. Over-dimensioned and over-toleranced scheme.

When formulating an analysis problem of tolerance stack-up in assemblies for the matrix based tolerance propagation method, only the scheme in the figure 3-1 represents the proper formulation of the problem. The scheme in the figure 3-2 does not provide enough information to set up a continuous chain of transformations. The scheme in the figure 3-3 provides more information than needed. There are two possible transformation chains ($D_1 + D_2 + D_3$ or $D_1 + D_4$), and it is unclear which is the right one.
The designer should always check the design for any under- or over-constraints. The verification is a relatively simple task for one- or two-dimensional drawings. However, the interactions between different dimensions and tolerances become substantially more complex in three dimensions. These are the problems of design validity and the issues, although important, are not addressed by in work. For additional reading on this subject refer to N. S. Bernstein and K. Preiss [3]. It is important that, in the future, unambiguous and complete methods of assigning dimensions and tolerances to three dimensional objects in a systematic way be developed.

3.2 The analytical tolerance propagation method and the current
ANSI Y14.5M-1982 standards

According to Lowell W. Foster [12], ANSI Y14.5M-1982 is the authoritative document governing the use of geometric dimensioning and tolerancing in the United States. There were three primary objectives in the development of the standard:

1. To provide a single standard for practices in the United States.
2. To update the existing practices in keeping with technological advances and extend the principles into new areas of application.
3. To establish a single basis and "voice" for the United States in the interest of international trade, in keeping with the United States' desire to be more active, gain influence, and
pursue a more extensive exchange of ideas with other nations in the area of international standards development.

All of these objectives have been achieved to some degree through development and implementation of the ANSI Y14.5M-1982 standards. However, in author's opinion, the ANSI standards have a few flaws and inconsistencies which may discourage their use in the industrial and the academic environment.

One obvious flaw is its limited capability of interpreting three dimensional objects. Although the solid modeler technology and other computer aided packages help the designer to visualize and understand three dimensional aspects of parts and assemblies, no clearly defined rules help the designer to systematically assign dimensions and tolerances to three dimensional objects. The current standards apply to two dimensional drawings, and their meaning have to be translated into three dimensional environment.

The ANSI standards have been developed based on measurement methods and not on a mathematical model. Therefore, they are well suited for quality controllers and not very well for more rigorous mathematical analysis; for example, of the tolerance stack-up in assemblies. The different tolerance characteristics are often redundant. To cite the three examples from [12]:

1) Parallelism can include flatness or straightness.
2) Runout can include circularity, straightness or cylindricity.
3) Cylindricity tolerance simultaneously controls circularity, straightness, and parallelism of elements of a cylindrical surface.
Another problem of ANSI standards is the lack of explicit representation for angular (or orientation) tolerance. Although the angular tolerance is often the culprit of significant position errors accumulation, it is not directly addressed by the ANSI definitions of orientation tolerances. Perpendicularity tolerance, for example, is defined by different types of tolerance zones. Depending on the application, there are the following types of zones defining perpendicularity: zone between two parallel planes, cylindrical zone or zone between two parallel lines. Similarly, angularity is defined by the distance between two parallel planes, and not by the angle deviation from nominal, as its name implies. Parallelism is defined by the same types of tolerance zones as the perpendicularity.

Consider the example of angularity, as in the figure 3-4. The entire surface must fall within 0.005 tolerance zone to be acceptable. However, the maximum angular deviation of the surface orientation from the nominal 45° can be determined indirectly knowing the nominal length of the feature $a$. Then, the maximum angular tolerance $\theta = \arctan\left(\frac{0.005}{a}\right)$. However, the angularity tolerance also controls the flatness of the surface to 0.005. This is another example of redundancy that could be avoided if the angularity tolerance were defined only by the angular deviation from nominal specified angle.

3.3 How the matrix propagation method relates to the ANSI Y14.5M-1982 standards

The analytical method of Cartesian frames propagations through
Figure 3-4. The angularity tolerance controls both flatness and the angular deviation from nominal. Any point on the surface must lie between the two parallel planes separated by the distance of 0.005.

the three dimensional space, was developed based on analytical assumptions (listed in section 2.2). The conceptual representation of a feature on a part (its position and orientation) by means of a Cartesian frame with six degrees of freedom is unambiguous from the mathematical point of view. The concept's acceptance has not yet been tested in the industrial and research environment. It should prove to be useful in analysis of tolerance propagation through complex three dimensional assemblies (the method was designed for this purpose). Its usefulness may be limited, however, when applied to complete representation of a single part dimensions and tolerances.

In order to be able to formulate problems of tolerance propagation through assemblies, some attempt should be made on reviewing compatibility questions between the ANSI Y14.5M standards and the
three dimensional Cartesian frame representation. It is beyond the scope of this thesis to review all the possible variations of tolerance characteristics specified in the ANSI Y14.5M standards, and to compare them all with the method presented in this thesis. In the following discussion, I describe four simple examples and re-formulate the four ANSI tolerance characteristics (position, perpendicularity, angularity, parallelism) in an alternate way, suitable for Cartesian frame assignments.

3.3.1 Position tolerance

Position is a term used to describe the perfect (exact) location of a point, line, or plane of a feature in relationship with a datum reference or other frame. A position tolerance is the total permissible variation in the location of a feature about its exact true position.

Representation of position tolerance by means of Cartesian frame position uncertainty is relatively straightforward. As an example, consider the figure 3-5 representing a hole with its axial position constrained to a cylindrical zone of radius .005.

Only the $\Delta X = .005$ and $\Delta Y = .005$ position tolerances of the frame has to be specified to represent the tolerance zone in the matrix form. By specifying the X and Y linear tolerances, the program automatically transforms the frame location uncertainty into the circular zone with $R = 0.005$; and not to the square tolerance zone which is not a recommended practice by ANSI standards.

44
Figure 3-5. The position tolerance of the hole axis is specified as a cylindrical zone with the radius 0.005.

3.3.2 Perpendicularity tolerance

Perpendicularity is the condition of a surface, median plane, or axis which is at exactly 90° to a datum plane or axis. Depending on a feature being tolerated, the tolerance zone is defined by:

1) Two parallel planes perpendicular to a datum plane.
2) Two parallel planes perpendicular to a datum axis.
3) Cylinder perpendicular to a datum plane.
4) Two parallel straight lines perpendicular two a datum plane or a datum axis.

As an example, consider the figure 3-6. The tolerance zone is the cylindrical region perpendicular to the datum plane A. Knowing the length of the shaft 'a', we can calculate the maximum allowable angular...
Figure 3-6. The perpendicularity tolerance specifies that the shaft's axis has to lie entirely within the cylindrical zone with the radius 0.005, at the maximum material condition, with respect to the datum plane A.

tolerance: $\Delta \theta_{\text{max}} = \arctan\left(\frac{0.005}{\text{a}}\right)$. Let us position the coordinate frame such that the Z axis is aligned with the shaft's axis. The angular tolerance of orientation deviation from nominal values is described by the small rotation angles $\Delta \theta_x$ and $\Delta \theta_y$. For small angles, the order of rotations is not significant, and it can be assumed that $\Delta \theta_{\text{max}} = \Delta \theta_x = \Delta \theta_y$.

3.3.3 Angularity tolerance

Angularity is the condition of a surface, axis, or median plane which is at the specified angle (other than 90°) from a datum plane or axis. Angularity tolerance is the distance between two parallel planes, inclined at the specified angle to a datum plane or axis, within which the tolerated surface, axis or a median must lie.
The angularity example has been shown in the figure 3-4. As in the previous perpendicularly case, the maximum angular tolerance can be computed \( \Delta \theta_{\text{max}} = \arctan \left( \frac{0.005}{a} \right) \). This angular tolerance can also be represented as two small angular rotations around the X, and Y axes. Note that the frame assignment to the surface of a feature (its position and orientation) depends on the tolerance chain that we would like to analyze.

3.3.4 Parallelism tolerance

Parallelism is the condition of a surface or axis which is equidistant at all points from a datum plane or axis. Depending on a case, a parallelism tolerance is specified by:

1) two planes or lines parallel to a datum plane (or axis),
2) a cylinder parallel to a datum axis.

Consider the block in the figure 3-7.

![Figure 3-7](image)

Figure 3-7. The upper surface of the block has to lie between two planes separated by the distance of 0.002 and parallel to the datum plane A.
Because the toleranced surface must lie between the two parallel planes separated by the distance 0.002 and parallel to the datum plane, once again the maximum angular error of orientation can be computed as $\Delta \theta_{\text{max}} = \arctan\left(\frac{0.002}{a}\right)$. This tolerance can be interpreted as small angular uncertainty ($\theta_x$ and $\theta_y$) of the Cartesian frame orientation which is attached to the upper surface of the block.
4.0 DESCRIPTION OF THE PROGRAM MAJOR FUNCTIONS

There are fifteen major functions in the main structure of the program. These functions are described briefly in this chapter. The names of the functions (as used in the program) are listed in the title of each section in parentheses. The complete listing of the code can be found in the Appendix C. The program has been written in C computer language and implemented on Sun 3/60 minicomputer.

4.1 Setting the default values (default_data)

The first function called by the main program is the initialization routine. Three default transformations are initialized.

The first query displayed by the program is: "Would you like to display the demonstration run?". If the user answers 'y', the default transformations with default tolerances set in the initialization routine will be used in the first program run.

4.2 User input data (get_user_data)

The user can input his/her own data. This can be done immediately after he/she answers 'n' to the default data run query (above). Another option the user has is to display the default run, and then, from the menu that follows, choose to change the existing data. The input routine calls upon other functions that check for possible user's input errors. The newly entered values can be saved into an existing file, or into a new file created by the user. The data can be
later retrieved from the file for re-display, modification, or adding additional transformations.

4.3 The nominal position of the propagated frame (optimal position)

The nominal position (no errors present) of the last frame in the chain of transformations is computed in the function "optimal_position". The nominal position vector is obtained by multiplying the 4x4 homogeneous transformation matrices \( T_i \) in the specified order.

\[
T_N = \prod_{i=1}^{N} T_i
\]

\[
T_i = (T_i)_{\text{Transl}}(T_i)_{\text{RotX}}(T_i)_{\text{RotY}}(T_i)_{\text{RotZ}}.
\]

\( T_N \) is the final transformation matrix (in the chain of \( N \) transformations). The \( i \) index denotes the transformation number. Transl means a translation transformation. RotX, RotY, RotZ are the indices specifying the rotation transformations around the X, Y, Z axes respectively. The transformation matrices are of the following form:

\[
(T_i)_{\text{Transl}} = \begin{pmatrix}
1 & 0 & 0 & X_i \\
0 & 1 & 0 & Y_i \\
0 & 0 & 1 & Z_i \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
(T_i)_{\text{RotX}} = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & \cos[x_i] & -\sin[x_i] & 0 \\
0 & \sin[x_i] & \cos[x_i] & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
\[ (T_i)_{\text{Rot}Y} = \begin{pmatrix}
\cos[y_i] & 0 & \sin[y_i] & 0 \\
0 & 1 & 0 & 0 \\
-sin[y_i] & 0 & \cos[y_i] & 0 \\
0 & 0 & 0 & 1
\end{pmatrix} \]

\[ (T_i)_{\text{Rot}Z} = \begin{pmatrix}
\cos[z_i] & -\sin[z_i] & 0 & 0 \\
\sin[z_i] & \cos[z_i] & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix} \]

The nominal position vector (3x1) is extracted from the last column of the final transformation matrix \( T_N \) (4x4), with its three elements equal to: \( T_N(1,4) \), \( T_N(2,4) \), \( T_N(3,4) \).

### 4.4 Assign the values to the variance matrices (V_matrices)

By definition, the variance matrices \( V \) are NxN diagonal matrices with the diagonal elements equal to the variances in the specified linear or angular directions. Therefore, there are six variance matrices: \( V_X \), \( V_Y \), \( V_Z \), \( V_x \), \( V_y \), \( V_z \). The first three refer to the translations in the \( X \), \( Y \), \( Z \) directions and the other three to the rotations around the \( X \), \( Y \), \( Z \) axes, respectively.

Because the normal distributions are assumed to be truncated \((\pm3\sigma)\), the variance in any direction for the particular frame is \((\frac{\text{tolerance}}{3})^2\). If the assumed distribution is uniform, the variance is \((k\times\text{tolerance})^2\), and \( k = \frac{2}{\sqrt{12}} \).
4.5 Computation of the $R$, $p$, $m$ matrices (all Rs_ps_and_ms)

For the derivation of the $m$ matrices and their final general form, the reader is referred to the section 2.3.1, and the Appendix A. The matrices are computed separately for each specified frame number, and stored in arrays for quick retrieval with no need for re-computation. The $R$ rotation matrices are stored in three dimensional arrays. The remaining $p$, $m_1$, $m_2$, $m_3$, $m_4$, $m_5$, $m_6$, $m_7$, $m_8$, $m_9$ vectors are stored in two dimensional arrays. The last dimension index (third or second) is reserved for the frame number.

4.6 Computation of the $W$ matrices (W_matrices)

For the derivation of the $W$ matrices, the reader is referred to the section 2.3.2. There are 12 $W$ matrices of size $3 \times N$, where $N$ is the total number of frames.

4.7 Covariance matrix of the position vector ($V_p$ matrix)

After the prerequisite $W$ and variance $V$ matrices have been computed, the program is ready to compute the covariance matrix ($V_p$) for the $N$th frame in the chain position uncertainty. The formula to compute the covariance matrix is:

$$V_p = W_1 V_x W_1^\dagger + W_2 V_y W_2^\dagger + W_3 V_y W_3^\dagger + W_{10} V_x W_{10}^\dagger + W_{11} V_y W_{11}^\dagger + W_{12} V_z W_{12}^\dagger$$

where $\dagger$ means matrix transpose. The resultant $V_p$ matrix is of size $3 \times 3$. 

52
4.8 Covariance matrix of the rotation vector (Vdel_matrix)

The next step is the computation of the 3x3 covariance matrix 
(V\delta) for the orientation uncertainty of the final frame in the chain of N 
transformations. The matrix is computed from the following formula:

\[ V_\delta = W_5 V_x W_5^\dagger + W_7 V_y W_7^\dagger + W_9 V_z W_9^\dagger \]

4.9 Eigenvalues of the position covariance matrix (eigenvalues)

To find the eigenvalues and eigenvectors of the covariance 
matrices, the Jacobi algorithm is used. Readers interested in this 
algorithm should refer to Hornbeck, Jacobi [16].

The eigenvalues of the Vp covariance matrix are variances in the 
principal directions of the probability density ellipsoid. Standard 
deviations in each of the three principal directions are the square 
roots of the variances. The full lengths of the three ellipsoid's 
principal axes are equal to six times the standard deviation in the 
principal directions.

4.10 Approximate convolution function

(convolve_uniform_and_normal)

If one of two input distributions is normal and the other one is 
uniform, or both distributions are uniform; the program performs a 
simplified convolution operation. Depending on the ratio of \sigma_{uniform}
to $\sigma_{\text{normal}}$, or $\sigma_{\text{uniform1}}$ to $\sigma_{\text{uniform2}}$; either normal or uniform distribution is chosen as the output convolved distribution. The resulting variance is always equal to the sum of the component distributions' variances. The $\chi^2$ tests showed that in majority of cases such approximation is justified. For details see section 2.4 and the Appendix B.

4.11 Two dimensional ellipsoid case (ellipse_2D_case)

The program checks if the probability density ellipsoid of the last frame position is degenerated to two dimensions. This occurs when the user specified tolerances only in the X and Y directions. Additionally, the rotations and their angular tolerances are specified to be zero. If all these conditions are satisfied, the calculated variance in the Z direction of the last frame is zero. For the two dimensional probability density ellipse, the calculations of its parameters are simple. Knowing the eigenvectors of the position covariance matrix, the rotation angle between the principal axes of the ellipse and the base reference frame can be easily obtained. The lengths of the principal half-axes of the probability density ellipse are determined by the remaining two (non-zero) standard deviations $\sigma_1$, $\sigma_2$:

$$\frac{X}{\sigma_1^2} + \frac{Y}{\sigma_2^2} = l^2,$$

where $l$ is a constant. For $l = 1, 2, 3$; the probabilities of finding the point within these ellipses are 0.394, 0.865, 0.989 respectively. Bryson, Ho [4].
4.12 Three dimensional ellipsoid case (inverse, base_orientation, ellipse_coefficients)

If all three variances in the principal directions of the ellipsoid are non-zero, the probability density ellipsoid of the final frame position is described by the equation:

\[
\begin{pmatrix}
X \\
Y \\
Z
\end{pmatrix}^\dagger \cdot V_p^{-1} \cdot 
\begin{pmatrix}
X \\
Y \\
Z
\end{pmatrix} = \ell^2
\]

where \( \ell \) is constant.

Therefore, the inverse of the covariance matrix has to be found first. This is done in the "inverse" function which is based on the Gauss-Jordan elimination algorithm. For reference, see Chapra, Canale [6].

The numerical results of the standard deviations in the principal directions of the ellipsoid are displayed on the screen. Also, a graphical representation of the ellipsoid was desired. Currently, the program does not make use of any solid modeler application. The display shows a two dimensional cross-section of the ellipsoid with the plane parallel to the XY reference plane and passing through the ellipsoid center. The graphical interface was designed using the SunCGI (Computer Graphics Interface) [28] application. The ellipsoid contours for \( \ell = 1, 2, 3 \) are shown on the screen. The probabilities of finding the point within these three concentric ellipsoids is 0.200, 0.739, 0.971 for the respective \( \ell = 1, 2, 3 \) multiples of \( \sigma \).

Let \( \beta \) be the angle of rotation between the ellipsoid's principal axes projections on the plane parallel to the reference plane, and the
$X$, $Y$ reference frame axes, see figure 4-1 below.

![Diagram of X, Y reference frame axes]

Figure 4-1. The graphical representation of the angle $\beta$.

Then, after the inverse of the $V_p$ matrix has been obtained, the $\beta$ angle can be found from:

$$
\beta = 0.5 \arctan \left( 2 \frac{[V_p^{-1}]_{[1,2]}}{[V_p^{-1}]_{[1,1]} - [V_p^{-1}]_{[2,2]}} \right)
$$

where the two indices in parentheses denote the row and the column of the $V_p^{-1}$ matrix. Note that the $V_p$ covariance matrix (and therefore $V_p^{-1}$, as well) is symmetric.

The $\beta$ angle is computed in the "base_orientation" function.

If the ellipsoid's contours in the $XY$ plane are described by the
following equation:
\[ a_1 x^2 + a_2 y^2 = l^2, \]
then the coefficients \( a_1 \) and \( a_2 \) can be found from the formulas:
\[ a_1 = (\cos[\beta])^2 (V_p^{-1})_{1,1} + (\sin[\beta])^2 (V_p^{-1})_{2,2} + \sin(2 \beta)(V_p^{-1})_{1,2} \]
\[ a_2 = (\cos[\beta])^2 (V_p^{-1})_{2,2} + (\sin[\beta])^2 (V_p^{-1})_{1,1} + \sin(2 \beta)(V_p^{-1})_{1,2}. \]

Note that, in a general case, the ellipsoid is not aligned with the base reference frame. Therefore, usually
\[ a_1 \neq \frac{1}{\sigma_1^2}, \quad a_2 \neq \frac{1}{\sigma_2^2}, \]
where \( \sigma_1 \) and \( \sigma_2 \) are the standard deviations in the principal directions.

The \( a_1 \) and \( a_2 \) coefficients are computed in the "ellipse_coefficients" function.

4.13 Graphical display of the results (draw_ellipses_and_base)

The graphical display was designed using the SunCGI [28] application. The function "draw_ellipses_and_base" displays three concentric ellipses. The three ellipses represent the contours of the ellipsoids:
\[
\begin{pmatrix}
X \\
Y \\
Z
\end{pmatrix}^\dagger V_p^{-1} \begin{pmatrix}
X \\
Y \\
Z
\end{pmatrix} = l^2
\]
for \( l = 1, 2, 3 \); in the plane parallel to the base reference frame XY plane. The additional information displayed in the graphics window is the length of principal axes for \( l = 1, 2, 3 \) ellipsoids' contours, in the
plane parallel to XY. Also, the orientation of the X, Y base reference axes is shown with respect to the ellipses' two principal axes orientation.

The function "draw_ellipses_and_base" utilizes substantial amount of features and functions from the SunCGI application. The reader interested in the particular details should refer to the code listing (Appendix C) and to the SunCGI Reference Manual [28].

4.14 Monte Carlo simulation (monte_carlo)

The user may opt to generate 1000 random assembly instances through the previously inputted chain of transformations. The perturbations around the nominal value are the result of the user inputted tolerances and the tolerances' distribution types. The initial reason for the Monte Carlo simulation inclusion in the program was to verify the results obtained from the analytical method. Therefore, the Monte Carlo method lets the user confirm visually the validity of previously obtained ellipsoids' contours. The randomly generated points are projected on the plane parallel to the reference frame XY plane. Note that because of the general three dimensionality of the ellipsoid and the random points locations, the comparison of both methods is best performed for two dimensional cases.
5.0 CASE STUDY

As the case study example, a three-part precision-instrument assembly was chosen. The case is also discussed in Nevins, Whitney, et al [24].

The instrument consists of the end cover, the core and the housing, as shown in figure 5-1. These parts are circularly symmetric about the center line and, are to be soldered together at the places shown in the figure. The whole assembly process is performed automatically, at elevated temperature, and in high vacuum environment. The alignment of the core with both the housing and the end cover must be extremely precise.

There are the following alternatives of assembly operations. The core may be installed first in the housing or first in the end cover. Soldering of the component parts may be done joint-by-joint as soon as surfaces to be soldered are brought into proximity, or all joints may be soldered simultaneously after assembly. For the simultaneous soldering operation, a highly sensitive electrical measurement is needed to perform alignment adjustment. The other option is to perform the soldering operations one at a time. In the following analysis, I investigate the case when the core and the end cover are soldered first.

Dimensions and tolerances for the parts have been established as shown in the figures 5-2, 5-3 and 5-4; and the designer would like to know if the choice of tolerances permits successful assembly.

The tolerance propagation through the assembly can be
Figure 5-1. The precision assembly case study.

represented as the propagation of frames and their origin location uncertainty. The chain of five transformations through the end cover and the core assembly is shown in figure 5-5. The chain of three transformations through the end cover and the housing assembly is
Figure 5-2. End cover with the significant dimensions and tolerances shown in figure 5-6. The frames are designated with the letter F and two indices. The first index is equal to 1 for the transformation chain through the end cover and core assembly; and 2 for the transformation chain through the end cover and housing assembly. The second index designates the sequence of transformed frames, beginning with the 0 base reference frame. The frame F_{10} is coincident with the frame F_{20}.

In section 5.5, it is shown how the tolerance analysis problem can be performed by setting up one continuous chain through the assembly instead of two separate ones.

5.1 How the transformation frames are assigned

There is no restriction on the assignment of frames in the three dimensional space, as long as the position and the location tolerance of the nth frame can be unambiguously expressed in terms of the (n-1)th frame. As an example, in the following figures 5-5 and 5-6, the
Figure 5-3. Core with the significant dimensions and tolerances
frames are attached to mating surfaces (e.g. F₂₀, F₁₀, F₁₁), or positioned at the core axis (e.g. F₁₄), but they could equally well be positioned within a clearance gap should the position be justified by the design dimensional specifications.

Figure 5-4. Housing with the significant dimensions and tolerances.
Figure 5-5. The chain of five transformations through the end cover and the stem. Only the Y, Z axes are shown; the X axes point toward the reader.
Figure 5-6. The chain of three transformations through the end cover and the housing. Only the Y, Z axes are shown; the X axes point toward the reader.

We would like to investigate any possible interference between the housing top opening and the assembly consisting of the end cover and core. Therefore, we need to begin the two chains of
transformations at the same base reference frame position: $F_{10} = F_{20}$. Also, the last frames in both chains should be at the same position or close to each other. In this case, $F_{15}$ is at the upper stem mating surface at the level of the hole edge (figure 5-5), and the $F_{23}$ frame (figure 5-6) is affixed to the housing opening edge at the same $X$, and $Z$ coordinate as $F_{15}$ with respect to the base frame. The nominal distance along the $Y$ axis between the two frames is equal to the nominal clearance (0.02) between the core stem and the hole.

For this case study, it is important to observe that the problem is formulated in two dimensional plane $YZ$. The frames are assigned in such manner that the transformations occur in $Y$ and $Z$ direction only. In this case, there is no need for any transformations in $X$ direction; although, it is always possible to set them up, if needed for the purpose of analysis. Moreover, the assembly in this case is symmetric. Therefore, identical chains of transformations could have been specified in the $XZ$ plane, with no transformations in $Y$ direction. In fact, the symmetry of the product indicates that this one calculation is representative of all other planes passing through the symmetry axis. Our ultimate objective is to verify if there exists a possible interference between the frames $F_{15}$ and $F_{23}$ along the horizontal direction, in this case $Y$ (figures 5-5 and 5-6).

5.2 The frames coordinates

All the coordinates of the propagated frames are listed in figures 5-7 and 5-9 (screen dumps from the program run on the Sun minicomputer).
Figure 5-7. Contours of the ellipsoid of density for the first chain of transformations. Coordinates of the five transformations and the tolerances are shown in the 'CONSOLE' window.
Figure 5.9: Contours of the ellipsoid of density for the second chain of transformations are displayed. The nominal position of the last frame, and standard deviations are printed out.

To continue, press the right mouse button anywhere in the graphics area...
Figure 5.10. Monte Carlo simulation for the second chain of transformations. The nominal position of the last frame, and standard deviations are printed out.
As noted before, both chains start at the same reference frame. The transformation between the frames $F_{10}$ and $F_{11}$ is performed along the Y axis by the distance of $29.3 - \frac{12.015}{2} = 23.2925$, after which the frame $F_{11}$ is positioned at the surface of the end cover hole. Both Z and X translation coordinates are 0. The hole radius tolerance is ±0.015, and it is assumed to be normally distributed. However, for this transformation, the tolerance is related to Y translation only (the translation occurs along the radius, and is affected only by the radius tolerance). All the rotations and their tolerances are 0.

Next, we would like to account for the clearance between the end cover opening and the core lower stem\(^1\). The nominal clearance between the two is $\frac{12.015 - 11.985}{2} = 0.015$. Therefore, this is the amount of the next translation in Y direction between the frames $F_{11}$ and $F_{12}$. The $F_{12}$ frame position tolerance (±0.015) now affects both the X and Y directions, and it is uniformly distributed. All the other coordinates and tolerances are 0.

We move next to the $F_{13}$ frame positioned at the nominal assembly axis, by the translation of $\frac{11.985}{2} = 5.9925$. The tolerance of the stem radius (±0.015) affects the translation only in the Y direction, and it is assumed to be normally distributed.

The next transformation from $F_{13}$ to $F_{14}$ by $Z = 72.2$ has to account for the axial runout of the stem (maximum 0.001) and the end cover off-axis error (maximum 0.0112), both errors are assumed to be normally distributed. Any angular errors of stem position with respect

---

\(^1\) Because of the clearance, there is some wobble between the core and the end cover. The wobble may result in some angular error of the core axis, which in turn affects the lateral error at the top of the core. However, this error is neglected in the analysis.
to the instrument axis are neglected here.

The last frame (F₁₅) is positioned at the stem top mating surface, at the height (Z) where the top edge of the housing opening is at its nominal position.

The second chain of transformations starts at the same reference frame (F₂₀) as the first chain (F₁₀), and we move directly to the axis of the instrument at F₂₁. The translation is in Y direction only and there is no tolerances specified for any linear translation. However, the specifications call for the compounded angular tolerances of both the end-plate (0.0014 rad) and the housing (0.0023 rad). The angular data has to be input in degrees. Therefore, the angular errors in figure 5-9, transformation #1 are (0.0014+0.0023)\frac{180}{\pi} = 0.21199 degrees, and assumed to be normally distributed.

The next transformations to F₂₂ by Z = 72.2 takes into account the off-axial error of the housing (±0.1337), normally distributed. The error affects the translation Z = 72.2 in the horizontal plane. Therefore, it is specified in both X and Y directions. No other angular errors, or linear tolerances are specified.

The last translation to F₂₃ by Y = \frac{12.02}{2} gets us very close to the F₁₅ frame. The nominal distance between the two frames F₂₃ and F₁₅ is the nominal clearance \frac{12.02 - 11.98}{2} = 0.02.

The nominal positions of the final propagated frames for the two chains of transformations are listed in the program screen dumps in the figures 5-8 and 5-9.
5.3 Interpretation of the results

Four programs runs have been performed and the screen dumps of their results are included on the following pages: figures 5-7, 5-8, 5-9, 5-10.

After the user data input is finished, or the transformations data is retrieved from an existing file; the program displays the total number of transformations, all the translations with associated tolerances, and all the angular rotations with their tolerances (in degrees). The tolerances are assumed to be normally distributed, except for the X and Y directions. In this case, the user has the option of choosing the uniform distribution to account for a clearance between two mating parts, which were assembled along the Z axis. In our example, the uniform distribution was chosen for the frame $F_{12}$ location uncertainty, which accounted for the clearance between the end plate and the lower stem.

The three concentric ellipses, displayed on the screen, represent the contours of $1\sigma$, $2\sigma$ and $3\sigma$ ellipsoids of density intersected with a plane parallel to XY base reference plane and passing through the center of the ellipsoid. It should be noted that, although the product is symmetric, the ellipsoid contours are not circular. This happens because the Y translations in the chains occur along the tolerated radii where no tolerances on X directions are specified.

The nominal position of the ellipsoid center is computed and displayed on the screen. The probability of finding a point inside the inner most ellipsoid (Bryson, Ho [4]) is 0.200, inside the middle one is 0.739, and inside the outer most ellipsoid is 0.971. For the case of a
two dimensional ellipse, the same probabilities are respectively: 0.394, 0.865, and 0.989.

In our example, in the first chain of five transformations, the uniform distribution in X direction is dominating, as seen in figure 5-7. The program accepts uniform distributions for tolerances in X and Y directions, only; which helps to account for clearances between parts. A simplified convolution operation is performed within the program for two uniform, or one normal plus one uniform distributions. The resulting convolved distribution can be approximated either by uniform or normal probability density functions with adjusted variances (for details see the Appendix B).

The actual orientation of the ellipsoid in space is unrestricted. The X, Y axes displayed in the figures 5-7 through 5-10 represent the X, Y axes of the base reference frames F10, F20. In this case, they coincide with the projections of the ellipsoids principal axes on the planes of intersection. It is not always the case, however, that the axes coincide.

The linear and the angular standard deviations from nominal positions are computed and displayed on the screen. Some interesting observations can be made based on the ellipsoids contours and the values of standard deviations in our case study example. Notice that, in figure 5-8, the two linear standard deviations along the ellipsoid principal axes are non-zero (0.0166 and 0.0136). However, the third linear standard deviation is 0.0000. Therefore, we have a special case of two dimensional probability density distribution. Because the ellipse of density lies in the plane parallel to the XY base
reference plane, its view is displayed in the actual proportions.

For the second chain of the three transformations (figure 5-9), the standard deviations in two principal directions are approximately equal. However, the principal axis of the ellipse in Y direction is considerably foreshortened. Therefore, we conclude that the principal axes of the ellipsoid do not coincide with the base frame directions.

Finally, the Monte Carlo simulation is performed for the two tolerance propagation chains, figures 5-8 and 5-10. The actual 1000 random points are suspended in the three dimensional space around the nominal position point. Here, the projections of the points on the plane parallel to the base reference frame are shown. The display seems to confirm our previous observations. For the first case of five transformations and the resultant two dimensional probability density distribution, figures 5-7 and 5-8, almost all the randomly generated points fall inside the outer most ellipse. Also, the roughly uniform density gradient along the X axis can be visually verified. For the second case of the three transformations and the resultant three dimensional probability density distribution, figures 5-9 and 5-10, some of the points' projections fall significantly outside the foreshortened contour of the outermost ellipse in the Y direction. This confirms our previous observation that the ellipsoid in this case is not aligned with the base reference frame.
5.4 Practical conclusions for the first solution method

Figure 5-11 shows the two contours of the probability density ellipsoids of the frames $F_{15}$ and $F_{23}$ positions.

Figure 5-11 Two overlapping ellipsoids of density represent the position uncertainty of the frames $F_{15}$ and $F_{23}$. The nominal clearance between the two frames is 0.02. The figure shows that there is a substantial probability of interference between the two frames.
The ellipsoids' contours are shown in their actual proportions with their centers separated by the distance of the nominal clearance (0.02) between them. The ellipsoids of the angular error between the frames $F_{15}$ and $F_{23}$ are not shown here. According to the specified tolerances, the angular errors of the frame $F_{15}$ are 0. The angular standard deviations of the $F_{23}$ frame are 0.0707 degrees around the X, Y axes, and 0 around the Z axis.

A quick inspection of the figure shows that the tolerance design in many instances precludes this assembly successful operation. The final tolerance accumulation in the end cover and core assembly is quite satisfactory (frame $F_{15}$). Relatively few assemblies would result in tolerance stack-up exceeding the nominal clearance. However, the tolerance accumulation at the top opening of the housing and the end cover assembly (frame $F_{23}$) is poor. The large variability of location, resulting from combination of angular tolerances and the significant off-axial error of the housing, is likely to create interferences during the insertion, and to prevent significant number of assembly operations from completion.

5.5 Alternative formulation of the same problem

The two chains of transformations can be combined into one. The base reference frame can be established at the top of the core shaft. Therefore, the third chain of transformations starts where the first chain of transformations ends: $F_{30} = F_{15}$. The subsequent transformations follow the first chain of transformations (figure 5-5).
however, in the negative directions of Y and Z translations. Referring to the figure 5-5: $F_{31} = F_{14}$, $F_{32} = F_{13}$, $F_{33} = F_{12}$, $F_{34} = F_{11}$, $F_{35} = F_{10}$.

From this point on, the third chain is identical to the second chain of transformations. Referring to the figure 5-6: $F_{36} = F_{21}$, $F_{37} = F_{22}$, $F_{38} = F_{23}$.

The nominal distance between the base reference frame ($F_{30}$) and the last frame in the chain ($F_{38}$) is equal to the nominal clearance between the upper shaft and the top opening. The nominal clearance is $\frac{12.02 - 11.98}{2} = 0.02$.

The results of the program run are shown in the figures 5-12 and 5-13. Figure 5-12 illustrates the contours of the ellipsoid in the horizontal plane, and lists the first 7 transformations. Figure 5-13 shows the same ellipsoid's contours with added points' projections from the Monte Carlo simulation. Also, the coordinates of the 8th transformation are displayed. The resultant nominal position of the final frame, linear standard deviations in the principal axes and angular standard deviations are computed and printed out.

5.6 Practical conclusions for the second solution method

The conclusion for this tolerance chain set-up remains the same: There is little chance that the assembly operation will be successful with the currently specified dimensions and tolerances, as the figure 5-14 illustrates. Setting up one continuous chain of transformations simplifies the final interpretation of the results. There is no need to re-draw the overlapping ellipses, as it was done in the figure 5-11.
The final result displayed in the figures 5-12 and 5-13 apparently hides the information of tolerance contributions from the sub-assembly of the end cover plus housing, versus the sub-assembly of core plus housing. However, by choosing '2' from the "menu", we can re-display ellipsoid's contours and obtain standard deviations for any frame in the transformation chain. Therefore, the distribution of the frame $F_{35}$ will be displayed the same as the distribution of the frame $F_{15}$. 
Figure 5.12. Contours of the ellipsoid of density for the third chain of transformations. The coordinates and tolerances for the first seven transformations are displayed.
Figure 5.13. Monte Carlo simulation for the third chain of transformations. The coordinates and tolerances for the nominal position of the last frame, and standard deviations are printed out.
Figure 5-14 Contours of the ellipsoid of density represent the position uncertainty of the frame F_{38} in the horizontal plane. The target hole clearance is 0.02. The figure shows that there is little chance of successful assembly operation completion for this case.
6.0 RECOMMENDATIONS FOR FUTURE RESEARCH

The software for closed-form tolerance analysis enables the designer to estimate the statistical probability of the assembly task completion. Obviously, it was not feasible to address all the issues related to statistical tolerance analysis in three dimensions. In this chapter, I outline the research directions, which in my opinion should be given a prior consideration in continuing research.

6.1 Data representation

The current software allows the user to enter new data, modify, or save it into a file. No attempt has been made to design a flexible data structure that could organize the information on geometric properties of objects, and facilitate efficient retrieval of the data for further processing. The author acknowledges the view of N. S. Bernstein and K Preiss [3] that the data representation is the key to further progress in the field of tolerance analysis.

The development of a data base for this software was intentionally postponed. The primary goal was to investigate if the closed form probabilistic solution to the three dimensional tolerance propagation problem is achievable. The next goal was to verify the results of the derived solution and to assess its computational efficiency. The comparison was made against the Monte Carlo simulation method, and rendered very promising results.

It is the author's opinion that the data representation
development should be the next research goal. The tolerance analysis should be made more effective, flexible and user-friendly as the result.

Currently, the user is responsible for setting the tolerance chain propagation through the assembly. The user must supply all the dimensional data (all the translations and rotations with their respective tolerances between the frames). Also, the frames nominal locations must be supplied by the user. Provided that an efficient database, with well organized spatial information, is designed and implemented, these tasks could be automated within the program. The quotation from J. Martin (as quoted in [3]) is applicable here:

"The overall logical structure of the data will serve diverse applications in the future. Because its uses cannot be fully foreseen, it should be designed so as to represent the true properties of the data. In other words, the groupings of data items and the association between data items should represent the inherent structure of the data."

The following issues should be addressed when designing the data structure:

1) The database should provide all the information about the three dimensional aspects of the parts. For that purpose, the ANSI Y14.5M-1982 practices should be investigated, and their applicability to spatial representation of dimensional and tolerance data scrutinized.

2) Flow of information to and from the data base should be efficient. The data structure should be dynamic; that is, the user should have the option of augmenting the data base with
new, or remove redundant information.

3) Some mechanism should be implemented that could detect the over- or under- dimensioned and tolerated schemes. Every object should be dimensioned and tolerated unambiguously and reachable through only one path in space.

These are some of the issues that I would recommend to pay particular attention to, when developing the data base. There has been some work done in the area of spatial data representation, the reader interested in this subject is referred to Bernstein [3], Roy [27].

6.2 Influence of tolerance stack-up on assembly sequence choice and vice versa

The tolerance built-up can prevent some assembly sequences from completion. In other words, the probability that two parts or sub-assemblies will mate may be too low. This situation may occur even when all the parts are within their tolerance ranges. Sometimes, it may be worthwhile to re-consider other assembly sequences and/or redesign tolerances on the component parts. The tolerance analysis software could be of great help when deciding upon the optimal sequence choice.

Assembly sequence selection can affect the final tolerance of an assembly. Depending on the assembly sequence, the resting and grasping surfaces may be different, see Nevins, Whitney, et al. [24]. This fact may result in the final tolerance built-up falling out of the specified range for the final assembly. Similarly, some assembly
sequences may not be feasible (or unlikely to succeed).

The relationship between the assembly sequence and tolerance stack-up should be researched. For that purpose, the closed form tolerance analysis package could be interfaced with the assembly sequence generation program developed by Lui [21], Abell [1], Baldwin [2] and Whipple [34], at C. S. Draper Laboratory, Inc. Combined assembly sequence generation and tolerance analysis coupled with a flexible and efficient dimensional data base could become a very powerful design tool, to be used at very early product and process design stages. Some of the assembly sequences can be pruned early from the assembly sequence tree. Suggestions can be made to the user, which tolerances are most responsible for the tolerance built-up. Sensitivity analysis could be performed quickly, with pinpointing the percentage contribution of each part's tolerance in the particular assembly sequence total tolerance stack-up. Also, the effects of any fixture tolerances and the robot gripper errors should be accounted for.

6.3 Stability issues related to tolerance accumulation

When the assembly axis direction is not aligned with the gravitational forces, the total assembly tolerance can significantly increase. In this case, both angular errors and linear errors usually increase, which may result in the worst-case tolerance accumulation for the final assembly, figure 6.1. Non-vertical insertion assembly operation is quite common in practice. It may occur, for example, in
Figure 6-1. The effect of gravitation on the tolerance stack-up. The parts are positioned at their worst case clearance limits. The clearances are exaggerated to illustrate the point.

the case of an automatic transmission assembly, with both vertical and horizontal assembly axes. It is often not feasible to turn the whole transmission case to ensure vertical direction for the secondary axis insertion. Some insertion operations have to be performed horizontally.

The stability aspect of assemblies on tolerance built-up should be investigated. If the effects of gravity are not accounted for in assembly operation planning, many tolerance designs and assembly sequences could result in failure.
6.4 Notes on predicting different probability density functions for
tolerance distributions

The closed-form tolerance analysis method allows the user the limited choice of normal distributions for all tolerances associated with the six degrees of freedom, and uniform distributions for tolerances in the X and Y linear directions (which can account for clearance fits). Research has been done in the area of different tolerance probability density distributions, see Chase and Greenwood [7], Bjorke [5].

The author believes that much research is still needed to be able to choose the best statistical distribution model for a particular tolerance range in question. There are considerable obstacles to surmount before the best distribution model can be decided upon:

1) It is often impossible to claim any distribution type, before detailed information about the manufacturing process used to make a part is known. Even then, the relationship between process parameters and the tolerances are difficult to model.

2) Having obtained the information necessary to model the distribution, there is still a significant chance that the distribution may change. This may happen as a result of tool wear, machine operator change, variations in material, changes in machine settings, and many other factors.

3) Very often information about the process is not available a priori and historic data is needed to make any estimations. This is a serious and costly drawback, since the tolerance
analysis cannot be done before the actual production starts.

4) The author acknowledges the significance of the mean shift, as stressed by Chase and Greenwood [7]. However, it is questionable that the shift is a stable and predictable phenomena, and remains fixed at a certain point within the manufacturing tolerance range over the time. If the mean shift does drift within the tolerance range, the questions remain: How quickly can the drift occur? How do we determine the next position of the mean shift? How should we respond to the drift in the mean shift?

The issues stated above should be further investigated. However, it is the author's opinion that the effects of distribution models are sometimes over-emphasized. On the other hand, for example, the effects of angular tolerances on the final assembly tolerance are of great importance, and have not been given so much attention. The effects may be significant even for relatively "tight" tolerances, when the angular errors act on a long arm. This may occur even for very tight tolerances, when the angular tolerance error acts on a long arm. For detailed discussion, the reader is referred to Nevins and Whitney, et al [24], where the case study of axle shaft insertion is discussed.

The effort required to analyze or predict the probability density function for the tolerance distribution might not be warranted by the benefits obtained from that information. It is the author's opinion that the normal or uniform distribution assumption can be considered "the best guess", for the lack of better intelligence. Any other assumptions about the process distributions can only make the same claim.
6.5 Solid modeler application

The "real world" parts are three dimensional, and the interactions between the parts' dimensions and tolerances occur in three dimensional space. Therefore, efforts should be made to integrate the currently developed tolerance analysis software and the assembly sequence generating program into solid modeler environment. The solid modeler implementation can help the user visualize the three dimensional objects. Also, it could improve the capacity for computing the probability of interferences during the assembly process.

The closed-form tolerance analysis method is significantly more efficient than the currently used Monte Carlo simulation method. In fact, the time it takes to estimate the probability distribution using the closed-form matrix method is approximately equal to the time it takes to perform one iteration using the Monte Carlo simulation. The computational efficiency of the newly developed tolerance propagation analysis package should provide the incentive for its use in the solid modeler environment.
7.0 CONCLUSIONS

The tolerance analysis method, described in this thesis, is suitable for analysis of final position and orientation uncertainties of any frame in the chain of N transformations in space. The method is general enough to allow for statistical analysis of errors propagation through any chain that can be described by means of six degrees of freedom with their associated errors (for example, a robot gripper).

This approach provides an alternative to tolerance analysis based on the Monte Carlo simulation. The algorithm is much faster than the Monte Carlo simulation. The estimated time of finding the parameters of probability density ellipsoid for a tolerance chain is approximately equal to the time of one iteration in the Monte Carlo method for the same chain. The efficiency of the program may prove to be a promising factor for its possible future implementation on personal computers.

The tolerance stack-up analysis could be performed more efficiently if the closed-form tolerance propagation algorithm were combined with a solid modeler. In the past, Monte Carlo method has been use together with solid modeler for that purpose. The major advantage of the recently developed algorithm is its computational efficiency.

The only limitation that the current algorithm has is its inability to accommodate tolerance probability distributions other than normal (for all six degrees of freedom), and uniform (for X and Y linear translations). This limitation is due to the necessity of performing
convolution operations within the program for different types of distributions. This operation may be difficult to solve numerically and computationally intensive. The convolutions of uniform and normal distributions, however, have been solved by approximation (as explained in Appendix B).
REFERENCES


APPENDIX A. Derivation of $m$ vectors, for a simplified case

In this appendix the expanded forms\(^1\) of the $m$ vectors are derived for a simplified case. The derivation flow that leads to the $m$ vectors general form is repeated here (it was summarized in the section 2.3.1).

Any transformation between two frames $T$ in three dimensions can be described by translation transformation and three rotation transformations. The order of transformations is important:

$$ T = T_{\text{Transl}} T_{\text{RotX}} T_{\text{RotY}} T_{\text{RotZ}} $$

In a chain of $N$ transformations, $A_i$ is a transformation between $i$th and $i-1$ frame. As noted before, the order of transformations must be consistently maintained: first, translation transformation, followed by the rotation transformation:

$$ A_i = (A_i)_{\text{Transl}} (A_i)_{\text{Rot}}. $$

The rotation order is the following:

$$ (A_i)_{\text{Rot}} = (A_i)_{\text{RotX}} (A_i)_{\text{RotY}} (A_i)_{\text{RotZ}}. $$

The component homogeneous transform matrices are:

$$ (A_i)_{\text{Transl}} = \begin{bmatrix} 1 & 0 & 0 & X_i \\ 0 & 1 & 0 & Y_i \\ 0 & 0 & 1 & Z_i \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

---

\(^1\) The elements of some vectors and matrices are of considerable size. To fit on the page, their columns have been compressed and written one above another.
\[
\begin{align*}
(A_i)_{\text{Rot}X} &= \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & \cos[x_i] & -\sin[x_i] & 0 \\
0 & \sin[x_i] & \cos[x_i] & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix} \\
(A_i)_{\text{Rot}Y} &= \begin{pmatrix}
\cos[y_i] & 0 & \sin[y_i] & 0 \\
0 & 1 & 0 & 0 \\
-\sin[y_i] & 0 & \cos[y_i] & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix} \\
(A_i)_{\text{Rot}Z} &= \begin{pmatrix}
\cos[z_i] & -\sin[z_i] & 0 & 0 \\
\sin[z_i] & \cos[z_i] & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}.
\end{align*}
\]

Then, after the multiplication of the four 4x4 matrices, the general form of transform matrix \( A_i \) is:

\[
A_i = \begin{pmatrix}
\cos[y_i]\cos[z_i] & -\cos[y_i]\sin[z_i] & 0 & 0 \\
\cos[z_i]\sin[x_i]\sin[y_i] + \cos[x_i]\sin[z_i] & \cos[x_i]\cos[z_i] - \sin[x_i]\sin[y_i]\sin[z_i] & 0 & 0 \\
-\cos[x_i]\cos[z_i]\sin[y_i] + \sin[x_i]\sin[z_i] & \cos[z_i]\sin[x_i] + \cos[x_i]\sin[y_i]\sin[z_i] & 0 & 0 \\
0 & 0 & 1 & 0 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
\sin[y_i] \\
-\cos[y_i]\sin[x_i] \\
\cos[x_i]\cos[y_i] \\
0 \\
\end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}
\]

The \( A_i \) matrix consists of 3x3 rotation transform matrix \( R_i \) and translation vector \( p_i \):

\[
A_i = \begin{pmatrix}
R_i \\
p_i \\
0 \\
1 \\
\end{pmatrix}.
A transformation with a differential error superimposed on it can be modeled as:

\[ A_i + dA_i = (I + \delta A_i)A_i \] (see figure 2-1).

The above equation is equivalent to:

\[ dA_i = (\delta A_i)A_i. \]

Homogeneous error transform matrix \( \delta A_i \) consists of 3x3 error rotation transform matrix \( \delta R_i \) and error translation vector \( d_i \)

\[ \delta A_i = \begin{pmatrix} \delta R_i & d_i \\ 0 & 1 \end{pmatrix}. \]

Therefore, the differential change matrix \( dA_i \) will be fully described if we find the \( d_i \) error translation vector, and the error rotation vector \( \delta_i \). The \( \delta_i \) is the eigenvector of the error rotation transform matrix \( \delta R_i \). Let us first find the \( \delta A_i \) matrix.

Differential change matrix \( dA_i \) can be obtained by partial differentiation of \( A_i \) matrix with respect to scalars \( X_i, Y_i, Z_i, x_i, y_i, z_i \).

\[ dA_i = \left( \frac{\partial}{\partial X_i} A_i \right) dX_i + \left( \frac{\partial}{\partial Y_i} A_i \right) dY_i + \left( \frac{\partial}{\partial Z_i} A_i \right) dZ_i + \left( \frac{\partial}{\partial x_i} A_i \right) dx_i + \left( \frac{\partial}{\partial y_i} A_i \right) dy_i + \left( \frac{\partial}{\partial z_i} A_i \right) dz_i \]

The above equation can be rewritten in different form:

\[ dA_i = (D_i)_{\text{Trans}X} dX_i + (D_i)_{\text{Trans}Y} dY_i + (D_i)_{\text{Trans}Z} dZ_i + (D_i)_{\text{Rot}X} dx_i + (D_i)_{\text{Rot}Y} dy_i + (D_i)_{\text{Rot}Z} dz_i \]

We can find the \( \delta A_i \) matrix:

\[ \delta A_i = (D_i)_{\text{Trans}X} dX_i + (D_i)_{\text{Trans}Y} dY_i + (D_i)_{\text{Trans}Z} dZ_i + (D_i)_{\text{Rot}X} dx_i + (D_i)_{\text{Rot}Y} dy_i + (D_i)_{\text{Rot}Z} dz_i \]

The \( D_i \) matrices are computed as follows:

\[ (D_i)_{\text{Trans}X} = \left( \frac{\partial}{\partial X_i} A_i \right) A_i^{-1} \]

\[ (D_i)_{\text{Trans}Y} = \left( \frac{\partial}{\partial Y_i} A_i \right) A_i^{-1} \]
\[(D_i)_{\text{TransZ}} = \left( \frac{\partial}{\partial Z_i} A_i \right) A_i^{-1}\]
\[(D_i)_{\text{RotX}} = \left( \frac{\partial}{\partial x_i} A_i \right) A_i^{-1}\]
\[(D_i)_{\text{RotY}} = \left( \frac{\partial}{\partial y_i} A_i \right) A_i^{-1}\]
\[(D_i)_{\text{RotZ}} = \left( \frac{\partial}{\partial z_i} A_i \right) A_i^{-1}\]

At this point, we would make a simplifying assumption, stating that all of the partial derivatives are computed with the angles equal to 0. Therefore, \(x_i = 0, y_i = 0, z_i = 0\). Let us find the partial derivatives of \(A_i\) matrix first, and then the inverse of \(A_i\) matrix:

\[\frac{\partial}{\partial x_i} A_i = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}\]

\[\frac{\partial}{\partial y_i} A_i = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}\]

\[\frac{\partial}{\partial z_i} A_i = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}\]

\[\frac{\partial}{\partial x_i} A_i = \begin{pmatrix} 0 & 0 & 0 & 0 \\ \cos[z_i] \sin[y_i] & -\sin[y_i] \sin[z_i] & -\cos[y_i] & 0 \\ \sin[z_i] & \cos[z_i] & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}\]

\[\frac{\partial}{\partial y_i} A_i = \begin{pmatrix} 0 & 0 & 1 & 0 \\ \cos[z_i] \sin[y_i] & -\sin[y_i] \sin[z_i] & 0 & 0 \\ -\cos[y_i] \cos[z_i] & \cos[y_i] \sin[z_i] & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}\]
\[
\frac{\partial}{\partial z_i} A_i = \begin{pmatrix}
0 & -\cos[y_i] & 0 & 0 \\
\cos[x_i] & -\sin[x_i] \sin[y_i] & 0 & 0 \\
\sin[x_i] & \cos[x_i] \sin[y_i] & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

\[
A_i^{-1} = \begin{pmatrix}
\cos[y_i] \cos[z_i] & -\cos[y_i] \sin[z_i] & \sin[y_i] & 0 \\
-\cos[y_i] \sin[z_i] & \cos[y_i] \cos[z_i] & \sin[z_i] & 0 \\
\sin[y_i] & \sin[z_i] & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
\cos[z_i] \sin[x_i] \sin[y_i] + \cos[x_i] \sin[z_i] & -\cos[x_i] \cos[z_i] \sin[y_i] + \sin[x_i] \sin[z_i] & \cos[z_i] \sin[x_i] + \cos[x_i] \sin[y_i] \sin[z_i] & \cos[x_i] \cos[y_i] \\
\cos[x_i] \cos[z_i] - \sin[x_i] \sin[y_i] \sin[z_i] & -\cos[y_i] \sin[x_i] & \cos[z_i] \sin[y_i] \sin[z_i] & 0 \\
\sin[y_i] & \sin[z_i] & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\]

Now we are ready to find the \( D_i \) matrices\(^2\):

\[
(D_i)_{\text{trans}X} = \begin{pmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

\[
(D_i)_{\text{trans}Y} = \begin{pmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

\[
(D_i)_{\text{trans}Z} = \begin{pmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

\(^2\) For clarity and compactness of notation, the \( i \) indices have been dropped here.
\[
(D_i)_{\text{RotX}} = \begin{pmatrix}
0 & 0 & 0 & 0 \\
0 & \sin[x] & -\cos[x] & Z \cos[x] - Y \sin[x] \\
0 & \cos[x] & \sin[x] & -Y \cos[x] - Z \sin[x] \\
0 & 0 & 0 & 0
\end{pmatrix}
\]
\[
(D_i)_{\text{RotY}} = \begin{pmatrix}
\sin[y] & -\cos[y] \sin[x] & \cos[x] \cos[y] & -X \sin[y] + Y \cos[y] \sin[x] - Z \cos[x] \cos[y] \\
\cos[y] \sin[x] & [\sin(x)]^2 \sin[y] & -\cos[x] \sin[y] & -X \cos[y] \sin[x] - Y [\sin(x)]^2 \sin[y] + Z \cos[x] \sin[x] \sin[y] \\
-\cos[x] \cos[y] & -\cos[x] \sin[x] \sin[y] & [\cos(x)]^2 \sin[y] & X \cos[x] \cos[y] + Y \cos[x] \sin[x] \sin[y] - Z [\cos(x)]^2 \sin[y] \\
0 & 0 & 0 & 0
\end{pmatrix}
\]
\[
(D_i)_{\text{RotZ}} = \begin{pmatrix}
[\cos(y)]^2 \sin[z] & & & \\
\cos[x] \cos[y] \cos[z] + \cos[y] \sin[x] \sin[y] \sin[z] & & & \\
\cos[y] \cos[z] \sin[x] - \cos[x] \cos[y] \sin[y] \sin[z] & & & \\
0 & & & \\
-\cos[x] \cos[y] \cos[z] + \cos[y] \sin[x] \sin[y] \sin[z] & & & \\
-\cos[x] \sin[y] + \cos[x] \sin[x] \sin[z] - \cos[x] \sin[x] \sin[y] \sin[z] & & & \\
[\sin(x)]^2 \sin[z] + [\cos(x)]^2 [\sin(y)]^2 \sin[z] & & & \\
0 & & & \\
-X [\cos(y)]^2 \sin[z] + [\cos(x) \cos(z) - \sin(x) \sin(y) \sin(z)] \sin[y] + [\cos(z) \sin(x) + \cos(x) \sin(y) \sin(z)] \cos[y] & & & \\
-\cos(x) \cos(z) + \sin(x) \sin(y) \sin(z) \cos[y] - [\cos(z)]^2 + [\sin(x)]^2 [\sin(y)]^2 \sin[z] + \cos(z) \sin(x) \sin(y) \sin(z) + \cos(x) \sin(x) \sin(y) \sin(z)] \cos[y] & & & \\
[\cos(x) \sin(x) + \cos(y) \sin(z)] \cos[y] + [-\cos(x) \sin(y) - \cos(x) \sin(z)] \sin[z] + \cos(x) \sin(x) \sin(z) \sin(y) \sin(z)] \cos[y] & & & \\
0 & & & \\
\end{pmatrix}
\]
So the $\delta A_i$ matrix is found from the equation:

$$\delta A_i = (D_i)_{\text{Trans}X}dX_i + (D_i)_{\text{Trans}Y}dY_i + (D_i)_{\text{Trans}Z}dZ_i + (D_i)_{\text{Rot}X}dX_i + (D_i)_{\text{Rot}Y}dY_i + (D_i)_{\text{Rot}Z}dZ_i$$

The differential displacement and rotations $dX_i, dY_i, dZ_i, dx_i, dy_i, dz_i$ can be replaced by small actual errors (tolerances) $\Delta X_i, \Delta Y_i, \Delta Z_i, \Delta x_i, \Delta y_i, \Delta z_i$ to obtain the $\delta A_i$ matrix:

$$\delta A_i = \begin{pmatrix}
\sin[y] \Delta y + [\cos(y)]^2 \sin[z] \Delta z \\
\cos[y] \sin[x] \Delta y + [\cos(x) \cos(y) \cos(z) + \cos(y) \sin(x) \sin(y) \sin(z)] \Delta z \\
-\cos[x] \cos[y] \Delta y + [\cos(y) \cos(z) \sin(x) - \cos(x) \cos(y) \sin(y) \sin(z)] \Delta z \\
0
\end{pmatrix}
$$

$$-\cos[y] \sin[x] \Delta y + [-\cos(x) \cos(y) \cos(z) + \cos(y) \sin(x) \sin(y) \sin(z)] \Delta z$$

$$\sin[x] \Delta x + [\sin(x)]^2 \sin[y] \Delta y + [(\cos(x))^2 \sin(z) + (\sin(x))^2 (\sin(y))^2 \sin(z)] \Delta z$$

$$\cos[x] \Delta x - \cos[x] \sin[x] \sin[y] \Delta y + [\cos(z) \sin(y) + \cos(x) (\cos(y))^2 \sin(x) \sin(z)] \Delta z$$

$$0$$

$$\cos[x] \cos[y] \Delta y + [-\cos(y) \cos(z) \sin(x) - \cos(x) \cos(y) \sin(y) \sin(z)] \Delta z$$

$$-\cos[x] \Delta x - \cos[x] \sin[x] \sin[y] \Delta y + [-\cos(z) \sin(y) + \cos(x) \sin(x) \sin(z) - \cos(x) \sin(x) \sin(y)] \Delta z$$

$$\sin[x] \Delta x + [\cos(x)]^2 \sin[y] \Delta y + [(\sin(x))^2 \sin(z) + (\cos(x))^2 (\sin(y))^2 \sin(z)] \Delta z$$

$$0$$

Because the $dA_i$ matrix is of the form:

$$\delta A_i = \begin{pmatrix}
\delta R_i & d_i \\
0 & 1
\end{pmatrix}$$

103
the 3x1 $d_i$ vector is the last column of the $dA_i$ matrix. For convenience, it can be decomposed as the sum of contributing vectors, each associated with the small errors $\Delta X_i$, $\Delta Y_i$, $\Delta Z_i$, $\Delta x_i$, $\Delta y_i$, $\Delta z_i$:

$$d_i = \begin{pmatrix} \Delta X_i \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \Delta Y_i + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \Delta Z_i + \begin{pmatrix} Z_i \cos[x_i] - Y_i \sin[x_i] \\ -Y_i \cos[x_i] - Z_i \sin[x_i] \\ -X_i \cos[y_i] + Z_i \cos[x_i] \sin[x_i] \sin[y_i] - Y_i [\sin(x_i)]^2 \sin[y_i] \\ X_i \cos[x_i] \cos[y_i] - Z_i [\cos(x_i)]^2 \sin[y_i] + Y_i \cos[x_i] \sin[x_i] \sin[y_i] \end{pmatrix} \Delta x_i +$$

Let us name the six contributing vectors as $m$, so that:

$$d_i = m_{1i} \Delta X_i + m_{2i} \Delta Y_i + m_{3i} \Delta Z_i + m_{4i} \Delta x_i + m_{5i} \Delta y_i + m_{6i} \Delta z_i,$$

and

$$m_{1i} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix},$$

$$m_{2i} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix},$$

$$m_{3i} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix},$$

104
\[ m_{4i} = \begin{pmatrix} 0 \\ Z_i \cos[x_i] - Y_i \sin[x_i] \\ -Y_i \cos[x_i] - Z_i \sin[x_i] \end{pmatrix} \]

\[ m_{5i} = \begin{pmatrix} -Z_i \cos[x_i] \cos[y_i] + Y_i \cos[y_i] \sin[x_i] - X_i \sin[y_i] \\ -X_i \cos[y_i] \sin[x_i] + Z_i \cos[x_i] \sin[x_i] \sin[y_i] - Y_i \sin(x_i)]^2 \sin[y_i] \\ X_i \cos[x_i] \cos[y_i] - Z_i \sin(x_i)]^2 \sin[y_i] + Y_i \cos[x_i] \sin[x_i] \sin[y_i] \end{pmatrix} \]

\[ m_{6i} = \begin{pmatrix} \cos(x_i) \cos(y_i) - \sin(x_i) \sin(y_i) \sin(z_i)]^2 \sin[x_i] + [\cos(x_i) \sin(x_i) + \cos(x_i) \sin(y_i) \sin(z_i)] Z_i \cos[y_i] - X_i \sin[y_i] \\ [\cos(x_i) \sin(x_i) - \cos(x_i) \sin(y_i) \sin(z_i)] \cos(y_i) + \cos(x_i) \sin(x_i) \sin(y_i) \sin(z_i)] X_i \cos[y_i] - \sin(x_i)]^2 \sin[y_i] \\ (\cos(x_i) \sin(z_i) - \cos(x_i) \sin(y_i) \sin(z_i)) X_i \cos(y_i) - \sin(x_i)]^2 \sin[y_i] + (\cos(x_i) \sin(x_i) + \cos(x_i) \sin(y_i) \sin(z_i)) X_i \cos[y_i] - \sin(x_i)]^2 \sin[y_i] \\ (-\cos(x_i) \sin(x_i) - \cos(x_i) \sin(y_i) \sin(z_i)) X_i \cos(y_i) - \sin(x_i)]^2 \sin[y_i] + (-\cos(x_i) \sin(x_i) + \cos(x_i) \sin(y_i) \sin(z_i)) X_i \cos[y_i] - \sin(x_i)]^2 \sin[y_i] \end{pmatrix} \]

The \( \delta_i \) error rotation vector is the real eigenvector of \( \delta R_i \) 3x3 matrix. After the contributing vectors are isolated, \( \delta_i \) can be written as:

\[ \delta = m_{7i} \Delta x_i + m_{8i} \Delta y_i + m_{9i} \Delta z_i , \]

where

\[ m_{7i} = \begin{pmatrix} \cos[x_i] \\ 0 \\ 0 \end{pmatrix} \]

\[ m_{8i} = \begin{pmatrix} \cos[x_i] \cos[y_i] \\ \cos[y_i] \sin[x_i] \end{pmatrix} \]

\[ m_{9i} = \begin{pmatrix} \cos[z_i] \sin[y_i] \\ -\cos[y_i] \cos[z_i] \sin[x_i] \\ \cos[x_i] \cos[y_i] \cos[z_i] \end{pmatrix} \]
APPENDIX B. Choosing appropriate probability distribution model for clearances and their convolutions

Nine different cases have been analyzed, to find the exact forms of the convolved probability density functions. First, I analyzed the convolutions of two uniform distributions, then one uniform and one normal distribution. Each time, the mean of all the distribution functions was equal zero. To simplify, the resultant convolved distribution function is shown only for the negative portion of the real number axis. It is always symmetric with respect to the \( Y = 0 \) axis. The sum of the two continuous variables \( x \) and \( y \) is \( u \).

\[
    u = x + y, 
\]

and the convolved function is called \( P(u) \).

Next consider the following cases:

1) Two uniform distribution functions. One has twice the range of the other.

\[
\begin{align*}
    f(x) &= 0.1 & -5 \leq (x \leq 5) \\
    g(y) &= 0.05 & -10 \leq (y \leq 10).
\end{align*}
\]

The convolved function is of the form:

\[
\begin{align*}
    P(u) &= 0.005u + 0.075 & -15 \leq (u \leq -5) \\
    P(u) &= 0.05 & -5 \leq (u \leq 0)
\end{align*}
\]

The bold line represent the plot of \( P(u) \). The fine line is the normal
distribution with the variance being equal to the sum of variances of the two uniform distributions.

\[ h(x) = 0.125 \quad -4 \leq (x \leq 4) \]

Now \( P(u) \) represents the convolution of the functions \( f, g, \) and \( h. \)

\[
\begin{align*}
P(u) &= 0.0003125u^2 + 0.011875u + 0.11281, & -19 \leq (u \leq -11) \\
P(u) &= 0.005u + 0.075, & -11 \leq (u \leq -9) \\
P(u) &= -0.0003125u^2 - 0.000625u + 0.049688, & -9 \leq (u \leq -1) \\
P(u) &= 0.05, & -1 \leq (u \leq 0).
\end{align*}
\]

As before, the bold line represents the plot of \( P(u) \). The fine line is the normal distribution with the variance being equal to the sum of variances of all three uniform distributions.
3) Another uniform distribution function \( p(x) \) is added to the resultant distribution function \( F(u) \). This time the range of \( p(x) \) is 4.

\[
p(x) = 0.25 \quad \quad -2 \leq (x \leq 2)
\]

Now \( P(u) \) represents the convolution of the functions \( f, g, h, \) and \( p \).

\[
P(u) = 2.6042 \times 10^{-5} u^3 + 0.0016406 u^2 + 0.034453 u + 0.24117
\]

\[\quad -21 \leq (u \leq -17)\]

\[
P(u) = 0.0003125 u^2 + 0.011875 u + 0.11323
\]

\[\quad -17 \leq (u \leq -13)\]

\[
P(u) = -2.6042 \times 10^{-5} u^3 - 0.00070312 u^2 - 0.0013281 u + 0.056016
\]

\[\quad -13 \leq (u \leq -11)\]

\[
P(u) = -5.2083 \times 10^{-5} u^3 - 0.0015625 u^2 - 0.010781 u + 0.021354
\]

\[\quad -11 \leq (u \leq -9)\]
\[ P(u) = -2.6042 \times 10^{-5} u^3 - 0.00085937 u^2 - 0.0044531 u + 0.040339 \quad -9 \leq (u \leq -7) \]

\[ P(u) = -0.0003125 u^2 - 0.000625 u + 0.049271 \quad -7 \leq (u \leq -3) \]

\[ P(u) = 2.6042 \times 10^{-5} u^3 - 7.8125 \times 10^{-5} u^2 + 7.8125 \times 10^{-5} u + 0.049974 \quad -3 \leq (u \leq 0) \]

The bold line represents the plot of \( P(u) \). The fine line is the normal distribution with the variance being equal to the sum of variances of all four uniform distributions. As can be seen below, the convolved function approaches the normal distribution function when the number of overlapping uniform distributions increases. However, the number of segments and the complexity of the integration process also increases.
Now let's investigate a few cases of uniform plus normal distribution functions convolutions.

4) Both normal and uniform distribution functions have the same variances.

The uniform function has the range of 10.

\[ f(x) = 0.1 \quad \text{for} \quad -5 \leq x \leq 5 \]

Its variance is:

\[ \sigma^2 = \int_{-5}^{5} (0.1x^2) \, dx \]

and the standard deviation

\[ \sigma = 2.88675. \]

Therefore, the normal distribution function is of the form:

\[ g(x) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{x^2}{2\sigma^2}\right) \]

Here, the closed form solution \( P(u) \) of the convolved function cannot be found, but can be represented in the integral form.

\[
P(u) = 0.1 \left( \int_{-8.66025}^{u+5} \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{x^2}{2\sigma^2}\right) \, dx \right) 
\]

\[ -13.6603 \leq u \leq -3.66025 \]

\[
P(u) = 0.1 \left( \int_{u-5}^{u+5} \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{x^2}{2\sigma^2}\right) \, dx \right) 
\]

\[ -3.66025 \leq u \leq 0 \]

The bold line represents the plot of the convolved function \( P(u) \). The
fine line is the normal distribution with the variance being equal to the sum of variances of the uniform and the original normal distributions.

5) Now, consider the case when the uniform distribution standard deviation is 10 times the standard deviation of the normal distribution.

\[ f(x) = 0.01 \quad -50 \leq (x \leq 50) \]

\[ g(x) = \frac{1}{\sqrt{2\pi \sigma}} \exp\left( -\frac{x^2}{2\sigma^2} \right) \]

\[ \sigma = 2.88675. \]

And the convolved function \( P(u) \):

\[ P(u) = \int_{-3\sigma}^{u+50} \left( 0.01 \left[ \frac{1}{\sqrt{2\pi \sigma}} \exp\left( -\frac{x^2}{2\sigma^2} \right) \right] \right) dx \]

\[ -58.6603 \leq (u \leq -41.3397) \]
\[ P(u) = \int_{-3\sigma}^{3\sigma} \left( 0.01 \left[ \frac{1}{\sqrt{2\pi}\sigma} \exp\left( -\frac{x^2}{2\sigma^2} \right) \right] \right) \, dx \]

\[ P(u) = 0.009973 \quad -41.3397 \leq (u \leq 0) \]

6) Next case: a very small range for the uniform distribution. Standard deviation of the uniform distribution is 10 times smaller than the standard deviation of the normal distribution.

\[ f(x) = 1 \quad -0.5 \leq (x \leq 0.5) \]

The normal distribution function \( g(x) \) is the same as in the two previous cases.

\[ P(u) = \int_{-8.6603}^{u + 0.5} \left( 1 \left[ \frac{1}{\sqrt{2\pi}\sigma} \exp\left( -\frac{x^2}{2\sigma^2} \right) \right] \right) \, dx \]

\[ -9.1603 \leq (u \leq -8.1603) \]
\[ P(u) = \int_{u-0.5}^{u+0.5} \left( \frac{1}{\sqrt{2\pi}\sigma} \exp\left( -\frac{x^2}{2\sigma^2} \right) \right) dx \] 

\[ -8.1603 \leq u \leq 0 \]

Note: The lack of smoothness of the convolved (bold) curve is due to the fact that the actual \( P(u) \) function was approximated by linear segments. If the segments were smaller, the lack of smoothness would be less noticeable.

And let's check some intermediate cases for the uniform distribution standard deviation between 10 to 1 times \( \sigma \) of the original normal distribution.

7) Uniform distribution \( \sigma \) is 5 times \( \sigma \) of the normal distribution.

The standard deviation of the uniform distribution remains the same as in the previous three cases:

\[ \sigma = 2.88675. \]
The uniform distribution is
\[ f(x) = 0.02 \quad -25 \leq (x \leq 25) \]

The convolved function:
\[
P(u) = \int_{-3 \sigma}^{u + 25} 0.02 \frac{1}{\sqrt{2 \pi} \sigma} \exp \left[ -\frac{x^2}{2 \sigma^2} \right] dx
\]
\[-33.6603 \leq (u \leq -16.3397)\]
\[
P(u) = 0.019946 \quad -16.3397 \leq (u \leq 0)
\]

8) Next, the standard deviation of the uniform distribution is twice the standard deviation of the normal distribution.

\[ f(x) = 0.05 \quad -10 \leq (x \leq 10) \]
The convolved function:

\[ P(u) = \int_{-3\sigma}^{u+10} 0.05 \left[ \frac{1}{\sqrt{2\pi\sigma}} \exp\left\{ -\frac{x^2}{2\sigma^2} \right\} \right] dx \]

\[ -18.6603 \leq (u \leq -1.3397) \]

\[ P(u) = \int_{-3\sigma}^{3\sigma} 0.05 \left[ \frac{1}{\sqrt{2\pi\sigma}} \exp\left\{ -\frac{x^2}{2\sigma^2} \right\} \right] dx \]

\[ -1.3397 \leq (u \leq 0) \]

9) The standard deviation of the uniform distribution is three times the size of standard deviation of the normal distribution.

\[ f(x) = 0.0333333 \quad -15 \leq (x \leq 15) \]

The convolved function:

\[ P(u) = \int_{-3\sigma}^{u+15} 0.0333333 \left[ \frac{1}{\sqrt{2\pi\sigma}} \exp\left\{ -\frac{x^2}{2\sigma^2} \right\} \right] dx \]

\[ -23.6603 \leq (u \leq -6.3397) \]
\[ P(u) = \int_{-\frac{3\sigma}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right)}^{\frac{3\sigma}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2}\right)} dx \]

\[-6.3397 \leq (u \leq 0)\]

**Testing the goodness of fit.**

The cases presented above show that some of the convolved functions may be well approximated by normal distributions, and some by uniform. To investigate if these approximations are justified, I performed the \( \chi^2 \) goodness-of-fit test.


Let \((X_1, X_2, \ldots, X_k)\) be a multinomial random variable with parameters
\( n, p_1, p_2, \ldots, p_k \). Then:

(a) The random variable

\[
C = \sum_{i=1}^{k} \frac{(X_i - np_i)^2}{np_i}
\]

has approximately a \( \chi^2 \) distribution with \( k-1 \) degrees of freedom.

(For the approximation to be adequate, the \( k \) classes should be defined so that \( np_i \) is greater than or equal to 5, for all \( i \).)

(b) At the \( \alpha \) level of significance, \( H_0: p_1 = p_{10}, \ldots, p_k = p_{k0} \) is rejected in favor of \( H_1: \) at least one \( p_i \) not equal \( p_{10} \) if

\[
c \geq (\chi^2)(1-\alpha, k-1),
\]

where

\[
c = \sum_{i=1}^{k} \frac{(X_i - np_{10})^2}{np_{10}}.
\]

I wrote a simple program that generated normally or uniformly distributed random numbers with desired parameters of the two component distributions. These discrete points were added together and then tested for goodness-of-fit to either Gaussian or uniform distributions with variances equal to the sum of the component variances.

To assure that the approximation to \( \chi^2 \) was good enough, I had to satisfy the condition that \( np_i \) is greater than or equal to 5, for all \( i \).

Both normalized, the uniform and Gaussian distributions were divided into \( k = 10 \) classes. The \( p_{10} \) probability reached the lowest value at the extreme tails of the normal distribution (\( p_{10} = 0.0082 \), for \( i = 0 \) and \( i = \) 0082).
10). Therefore, the number of randomly generated discrete points had to be greater than or equal to 610.

At the significance level $\alpha = 0.05$, the hypothesis that the fit were good enough could be accepted if $c < 16.9$.

The results obtained for different distributions and different ranges are summarized in the two tables on the following pages.

**Explanation to the tables**

The first table shows the $\chi^2$ goodness-of-fit test results to the uniform and normal distribution when *both* of the initial distributions are uniform.

The second table shows the $\chi^2$ goodness-of-fit test results to the uniform and normal distribution when one of the initial distributions is normal and the other uniform.

In the first column, the value of standard deviation $\sigma$ is exactly the same as in the previously tested continuous functions' convolutions:

$$\sigma = 2.88675.$$  

In the next column ("$\chi^2$ TEST"), the $c$ value obtained from running the program is listed for the goodness-of-fit match to the normal
distribution. The fit is "good" at $\alpha = 0.05$ if $c < 16.9$.

The third column shows the range over which the test was performed for $H_0$: fit to the Gaussian distribution hypothesis. $\sigma_{\text{con}}$ is the standard deviation of the convolved function.

In the fourth column ("$\chi^2$ TEST"), the $c$ value obtained from running the program is listed for the goodness-of-fit match to the uniform distribution. The fit is "good" at $\alpha = 0.05$ if $c > 16.9$.

The fifth column shows the range over which the test was performed with $H_0$: the data fits to the uniform distribution function. $R$ is a constant defined as the ratio:

$$R = \frac{\text{Range of the uniform distribution}}{2\sigma \text{ of the same distribution}}$$

$R = 1.732051$.

The "PREFERRED DISTRIBUTION" column lists which approximation is better: normal or uniform, based on the smaller $c$ value.

The "GOODNESS OF FIT" column shows if the fit of the data was acceptable (i.e., $c < 16.9$ at $\alpha = 0.05$).
<table>
<thead>
<tr>
<th>Yes</th>
<th>Uniform</th>
<th>100.1</th>
<th>3.5</th>
<th>11.1</th>
<th>129.7</th>
<th>100.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Uniform</td>
<td>51.0</td>
<td>12.3</td>
<td>88.3</td>
<td>88.7</td>
<td>50.6</td>
</tr>
<tr>
<td>Yes</td>
<td>Uniform</td>
<td>31.6</td>
<td>10.7</td>
<td>54.8</td>
<td>61.8</td>
<td>30.0</td>
</tr>
<tr>
<td>Yes</td>
<td>Uniform</td>
<td>26.0</td>
<td>10.2</td>
<td>45.0</td>
<td>47.6</td>
<td>24.0</td>
</tr>
<tr>
<td>No</td>
<td>Uniform</td>
<td>18.6</td>
<td>43.4</td>
<td>43.4</td>
<td>33.5</td>
<td>23.0</td>
</tr>
<tr>
<td>Yes</td>
<td>Uniform</td>
<td>24.2</td>
<td>41.9</td>
<td>41.9</td>
<td>39.6</td>
<td>22.9</td>
</tr>
<tr>
<td>No</td>
<td>Uniform</td>
<td>23.3</td>
<td>20.8</td>
<td>40.3</td>
<td>27.1</td>
<td>16.0</td>
</tr>
<tr>
<td>No</td>
<td>Uniform</td>
<td>22.4</td>
<td>38.7</td>
<td>38.7</td>
<td>34.6</td>
<td>26.0</td>
</tr>
<tr>
<td>No</td>
<td>Normal</td>
<td>25.5</td>
<td>37.2</td>
<td>22.4</td>
<td>9.9</td>
<td>1.86</td>
</tr>
<tr>
<td>Yes</td>
<td>Normal</td>
<td>21.9</td>
<td>35.7</td>
<td>26.7</td>
<td>1.76</td>
<td>1.76</td>
</tr>
<tr>
<td>No</td>
<td>Normal</td>
<td>18.9</td>
<td>32.7</td>
<td>17.0</td>
<td>1.65</td>
<td>1.65</td>
</tr>
<tr>
<td>Yes</td>
<td>Normal</td>
<td>18.0</td>
<td>31.2</td>
<td>9.5</td>
<td>1.50</td>
<td>1.50</td>
</tr>
<tr>
<td>Yes</td>
<td>Normal</td>
<td>14.1</td>
<td>24.5</td>
<td>6.1</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Yes</td>
<td>Normal</td>
<td>10.0</td>
<td>27.2</td>
<td>8.4</td>
<td>0.15</td>
<td>0.15</td>
</tr>
</tbody>
</table>

**Legend:**
- **Range:** The range of the distribution.
- **$\chi^2$ Test:** The $\chi^2$ test statistic.
- **Goodness of Fit:** Indicates the goodness of fit for the distribution.
- **Preferred Distribution:** The preferred distribution for the given conditions.
- **Standard Deviation:** The standard deviation of the distribution.

**Note:** The table shows the results for different conditions, with the rightmost column indicating the distribution that best fits the data (in multiples of the other).
<table>
<thead>
<tr>
<th>GOODNESS OF FIT</th>
<th>DISTRIBUTION</th>
<th>UNIFORM _ APPROX</th>
<th>GAUSSIAN _ APPROX</th>
<th>DISTRIBUTION OF CONVOLVED</th>
<th>DISTRIBUTION OF CONVOLVED</th>
<th>STANDARD DEVIATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFERENCE</td>
<td>UNIFORM</td>
<td>3.1.6</td>
<td>10.1</td>
<td>8.4</td>
<td>4.0</td>
<td>3.0</td>
</tr>
<tr>
<td>PREFERENCE</td>
<td>UNIFORM</td>
<td>26.9</td>
<td>1.3</td>
<td>4.0</td>
<td>15.2</td>
<td>4.0</td>
</tr>
<tr>
<td>PREFERENCE</td>
<td>UNIFORM</td>
<td>26.0</td>
<td>1.3</td>
<td>9.0</td>
<td>15.2</td>
<td>4.0</td>
</tr>
<tr>
<td>NO</td>
<td>UNIFORM</td>
<td>2.2.9</td>
<td>21.5</td>
<td>1.2</td>
<td>9.7</td>
<td>21.7</td>
</tr>
<tr>
<td>NO</td>
<td>UNIFORM</td>
<td>2.2.0</td>
<td>2.2.4</td>
<td>22.7</td>
<td>3.8</td>
<td>1.9</td>
</tr>
<tr>
<td>NO</td>
<td>UNIFORM</td>
<td>2.0.9</td>
<td>18.5</td>
<td>1.2</td>
<td>1.7</td>
<td>1.0</td>
</tr>
<tr>
<td>NO</td>
<td>NORMAL</td>
<td>9.6.9</td>
<td>20.6</td>
<td>1.2</td>
<td>9.7</td>
<td>1.0</td>
</tr>
<tr>
<td>NO</td>
<td>NORMAL</td>
<td>1.6.9</td>
<td>1.9</td>
<td>7.2</td>
<td>3.9</td>
<td>1.0</td>
</tr>
<tr>
<td>NO</td>
<td>NORMAL</td>
<td>1.9.0</td>
<td>1.8</td>
<td>7.0</td>
<td>3.7</td>
<td>1.0</td>
</tr>
<tr>
<td>NO</td>
<td>NORMAL</td>
<td>1.4.9</td>
<td>1.7</td>
<td>0.7</td>
<td>2.9</td>
<td>1.0</td>
</tr>
<tr>
<td>NO</td>
<td>NORMAL</td>
<td>1.3.9</td>
<td>1.5</td>
<td>7.0</td>
<td>3.7</td>
<td>1.0</td>
</tr>
<tr>
<td>NO</td>
<td>NORMAL</td>
<td>1.2.9</td>
<td>1.6</td>
<td>0.7</td>
<td>2.7</td>
<td>1.0</td>
</tr>
<tr>
<td>NO</td>
<td>NORMAL</td>
<td>1.5.9</td>
<td>1.4</td>
<td>7</td>
<td>3.6</td>
<td>1.0</td>
</tr>
<tr>
<td>NO</td>
<td>NORMAL</td>
<td>1.3</td>
<td>0.7</td>
<td>2.4</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>NO</td>
<td>NORMAL</td>
<td>1.1</td>
<td>0.7</td>
<td>2.4</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conclusions

(a) Two uniform distributions. From the test results, it can be seen that we can safely assume that the resultant distribution is normal if the standard deviations of the two initial distributions are almost equal to each other. If one of the component distributions is significantly larger than the other, than the resultant uniform distribution is a good assumption. There is a transition region, approximately between $1.8\sigma$ and $2.3\sigma$, where neither uniform nor normal distribution fits the data well. However, it can always be said that one distribution fits better than the other in that region.

(b) Uniform distribution plus normal distribution. For the uniform distribution with smaller, equal or slightly greater standard deviation than the initial normal distribution standard deviation; the resultant normal distribution is a good bet. The transition region also occurs, approximately between $1.8\sigma$ and $2.3\sigma$, where neither uniform nor normal distribution fits the data well. As before, we can make an assumption that one of the distributions is a better approximation, however. For the uniform distribution dominating over the normal, we are on the safe side assuming that the resultant data is uniformly distributed.

Any approximations are radically improving when there are more than two convolving distributions. Also, the region of "uncertainty-of-fit" is
relatively small, and the chance that the ratio of the two convolving
distributions falls in between is also small. Moreover, because the
distributions that we start with may be also quite crude assumptions to
real life situations, we shouldn't be too particular about this small
inaccuracy.

Concluding, it would be justified to decide on some "threshold" value,
and simply choose between the normal or uniform distribution for that
case, with appropriate convolved function variance as the sum of
contributing distributions' variances.
APPENDIX C. Complete listing of the software code.

The program was written in C programming language on Sun 3/60 minicomputer.

```c
#include <stdio.h>
#include <cgiedefs.h>
#include <math.h>
#define PI 3.141592653589793238
#define MAX_FRAME_NUM 100  /* Max. number of propagated frames that the program accepts */
#define MAX_TRIALS_NUM 1000 /* Max. number of random points generated in the Monte Carlo function */

#define TEN_MIN (600 * 1000 * 1000)
#define LID_LOC 1
#define MOUSE_BUTTON_3 4
#define YES 1
#define NO 0
#define UNI 0
#define NRM 1
#define RATIO 0.5773502691896257646
#define QUIT 0
#define CHANGE_DATA 1
#define ANOTHER_FRAME 2
#define MONTE_CARLO 3
#define FILE_DATA 4

/* ****************************************** *
/* The following function provides defaults for input dimensional data, and other info, normally provided by the user */
/* ****************************************** */
void default_data(X, delta_X, Y, delta_Y, Z, delta_Z, x, delta_x, y, delta_y, z, delta_z, num_of_frames, frame_num)
{ double X[MAX_FRAME_NUM], delta_X[MAX_FRAME_NUM][2],
   Y[MAX_FRAME_NUM],
   delta_Y[MAX_FRAME_NUM][2], Z[MAX_FRAME_NUM],
   delta_Z[MAX_FRAME_NUM], x[MAX_FRAME_NUM],
   delta_x[MAX_FRAME_NUM], y[MAX_FRAME_NUM],
   delta_y[MAX_FRAME_NUM], z[MAX_FRAME_NUM],
   delta_z[MAX_FRAME_NUM];
  int *num_of_frames, *frame_num;
  { X[0] = 1.0; /* capital letters X, Y, Z signify linear translations of the ith frame, max. number of frames is defined by MAX_FRAME_NUM */
    X[1] = 5.0;
    ... /* rest of the function */
    return 0; /* return value */
  }
}
```
delta_X[0][NRM] = 0.05;  /* variables preceded with the word 
"delta_" are acceptable, one sided 
tolerances from nominal. For that 
dimension: delta_ = 3*sigma */

delta_X[0][UNI] = 0.0;
delta_X[1][NRM] = 0.0;
delta_X[1][UNI] = 0.4;
Y[0] = 1.0;
Y[1] = 2.0;
delta_Y[0][NRM] = 0.05;
delta_Y[0][UNI] = 0.0;
delta_Y[1][NRM] = 0.1;
delta_Y[1][UNI] = 0.0;
Z[0] = 1.0;
Z[1] = 6.0;
delta_Z[0] = 0.03;
delta_Z[1] = 0.1;
x[0] = 0.0 * PI / 180.0;
x[1] = 0.0;
delta_x[0] = 0.5 * PI / 180.0; /* the angles are input in 
degrees, and then converted to 
radians */

delta_x[1] = 0.0 * PI / 180.0;
y[0] = 0.0 * PI / 180.0;
y[1] = 0.0;
delta_y[0] = 0.5 * PI / 180.0;
delta_y[1] = 0.0 * PI / 180.0;
z[0] = 0.0 * PI / 180.0;
z[1] = 0.0;
delta_z[0] = 0.5 * PI / 180.0;
delta_z[1] = 0.0 * PI / 180.0;
x[2] = 35.0 * PI / 180.0;
delta_x[2] = 1.0 * PI / 180.0;
y[2] = 78.0 * PI / 180.0;
delta_y[2] = 1.0 * PI / 180.0;
z[2] = 21.0 * PI / 180.0;
delta_z[2] = 1.0 * PI / 180.0;
x[2] = 11.2;
delta_X[2][NRM] = 0.05;
delta_X[2][UNI] = 0.0;
Y[2] = 0.07;
delta_Y[2][NRM] = 0.0;
delta_Y[2][UNI] = 0.1;
Z[2] = 19.9;
delta_Z[2] = 0.1;
*num_of_frames = 2;
/* the total number of frames over 
which the propagation occurs */
/* a particular frame number being 
considered */

*frame_num = 2;
}

/*****************************/
/* Check for errors in user inputted decimal integers. */
/* ........................................................................... */

int check_error_dec()
{
  int dec_number;
  char buf[100];

  while ( scanf("%d", &dec_number) != 1 )
    {
    printf("Invalid entry. Please try again...\n");
    scanf("%[^\n]\n", buf);
  }
  return ( dec_number );
}

/* ........................................................................... */
/* Check for errors in user inputted floating point numbers. */
/* ........................................................................... */

double check_error_double()
{
  double double_number;
  char buf[100];

  while ( scanf("%lf", &double_number) != 1 )
    {
    printf("Invalid entry. Please try again...\n");
    scanf("%[^\n]\n", buf);
  }
  return ( double_number );
}

/* ........................................................................... */
/* Check for errors in user inputted strings. */
/* ........................................................................... */

void check_error_string(str)
char str[20];
{
  char buf[100];

  while ( scanf("%s", str) != 1 )
    {
    printf("Invalid entry. Please try again...\n");
    scanf("%[^\n]\n", buf);
  }
}
/* ****************************************** */
/* Get the data from the user. Change the existing defaults, 
  add new frames, save data to a file. */
/* ****************************************** */

void get_user_data(X, delta_X, Y, delta_Y, Z, delta_Z, x,
  delta_x, y, delta_y, z, delta_z, num_of_frames,
  frame_num, demo)
{
  int i, yes_or_no(), answer = YES, trans = *frame_num +
    1, type, j, dec_number;
  double del_X[MAX_FRAME_NUM], del_Y[MAX_FRAME_NUM];
  int distr_type_X[MAX_FRAME_NUM],
    distr_type_Y[MAX_FRAME_NUM];
  int check_error_dec(), no_previous_error = 1;
  double check_error_double();
  FILE *file_ptr;
  char filename[20];
  void check_error_string();

  if (*demo == YES)
  {
    printf("\nTHE TOTAL NUMBER OF TRANSFORMATIONS IS %d\n\n",
      *num_of_frames + 1);
    for (i = 0; i <= *num_of_frames; i++)
      {
        printf("\nTRANSFORMATION # %d\n", i+1);
        printf("Translation: X = \%.3f, Y = \%.3f, Z = \n",X[i],Y[i],Z[i]);
        distr_type_X[i] = NRM;
        distr_type_Y[i] = NRM;
        for (type = UNI; type <= NRM; type++)
          {
            if(type == UNI \\&\
            delta_X[i][UNI] > 0.000003)
              {
                del_X[i] = delta_X[i][type];
                distr_type_X[i] = UNI;
              }
            else if (delta_X[i][type] > 0.000003)
              del_X[i] = delta_X[i][type];
            if(type == UNI \\&\
            delta_Y[i][UNI] > 0.000003)
              {
                del_Y[i] = delta_Y[i][type];
                distr_type_Y[i] = UNI;
              }
            else if (delta_Y[i][type] > 0.000003)
  127
\text{del}_Y[i] = \text{delta}_Y[i][\text{type}];
}

\text{printf}("Tolerance (+/-): X = %8.5f, Y = %8.5f, Z = %8.5f\n", \text{del}_X[i], \text{del}_Y[i], \text{delta}_Z[i]);
\text{printf}("Rotation (deg): x = %8.4f, y = %8.4f, z = %8.4f\n", \text{x[i]}*180.0/\text{PI}, \text{y[i]}*180.0/\text{PI}, \text{z[i]}*180.0/\text{PI});
\text{printf}("Angular toler.: x = %8.5f, y = %8.5f, z = %8.5f\n", \text{delta}_x[i]*180.0/\text{PI}, \text{delta}_y[i]*180.0/\text{PI},
\text{delta}_z[i]*180.0/\text{PI});
\text{if} (\text{distr\_type}\_X[i] != \text{NRM} && \text{distr\_type}\_Y[i] != \text{NRM})
\text{printf}("All tolerance distributions are normal except for X and Y, \text{which are uniform.}\n\n")
\text{else if} (\text{distr\_type}\_X[i] != \text{NRM})
\text{printf}("All tolerance distributions are normal except for X, \text{which is uniform.}\n\n")
\text{else if} (\text{distr\_type}\_Y[i] != \text{NRM})
\text{printf}("All tolerance distributions are normal except for Y, \text{which is uniform.}\n\n")
\text{else}
\text{printf}("All tolerance distributions are normal\n\n")
\text{printf}("Do you want to change any of the existing data?\n")
\text{answer} = \text{yes\_or\_no}();

\text{while} (\text{answer} == \text{YES})
{\n\text{if} (\text{no\_previous\_error})
  \text{do}
  \text{printf}("Please enter the transformation number you wish to change\n(between 1 and %d)\n", (\text{num\_of\_frames} + 1));
  \text{trans} = \text{check\_error\_dec}();
  \text{while} (\text{trans} < 1 || \text{trans} > (\text{num\_of\_frames} + 1));
\text{no\_previous\_error} = 1;
\text{printf}("The translation in X direction is %8.3f.\n", \text{X[trans-1]});
\text{printf}("Would you like to change it? (y/n)\n")
\text{answer} = \text{yes\_or\_no}();
\text{if} (\text{answer} == \text{YES})
  \text{printf}("Enter new value:\n");
  \text{X[trans-1]} = \text{check\_error\_double}();
\}

\text{printf}("The specified tolerance of X is (+/-)%8.5f,\n", \text{del}_X[\text{trans-1}]);\n\text{if} (\text{distr\_type}\_X[\text{trans-1}] == \text{NRM})

128
printf("and it is normally distributed.\n");
else
  printf("and it is uniformly distributed.\n");
printf("Would you like to change it? (y/n)\n");
answer = yes_or_no();
if (answer == YES)
{
  printf("Enter new value:\n");
  del_X[trans-1] = check_error_double();
  if (del_X[trans-1] != -0.0)
    {
      printf("Is the tolerance in X direction
normally or uniformly distributed?\nEnter 'n' for normally,
or 'u' for uniformly...\n");
      answer = is_uniform();
      if (answer == YES)
        distr_type_X[trans-1] = UNI;
      else
        distr_type_X[trans-1] = NRM;
    }
else
  distr_type_X[trans-1] = NRM;
}
printf("The translation in Y direction is \%8.3f.\n", Y[trans-1]);
printf("Would you like to change it? (y/n)\n");
answer = yes_or_no();
if (answer == YES)
{
  printf("Enter new value:\n");
  Y[trans-1] = check_error_double();
}
printf("The specified tolerance of Y is (+/-)\%8.5f,\n", del_Y[trans-1]);
if (distr_type_Y[trans-1] == NRM)
  printf("and it is normally distributed.\n");
else
  printf("and it is uniformly distributed.\n");
printf("Would you like to change it? (y/n)\n");
answer = yes_or_no();
if (answer == YES)
{
  printf("Enter new value:\n");
  del_Y[trans-1] = check_error_double();
  if (del_Y[trans-1] != -0.0)
    {
      printf("Is the tolerance in Y direction
normally or uniformly distributed?\nEnter 'n' for normally,
or 'u' for uniformly...\n");
      answer = is_uniform();
      if (answer == YES)
distr_type_Y[trans-1] = UNI;
else
distr_type_Y[trans-1] = NRM;
}
else
distr_type_Y[trans-1] = NRM;
}

printf("The translation in Z direction is %8.3f.\n", Z[trans-1]);
printf("Would you like to change it? (y/n)\n");
answer = yes_or_no();
if (answer == YES)
{
    printf("Enter new value:\n");
    Z[trans-1] = check_error_double();
}

printf("The specified tolerance of Z is (+/-)%8.5f.\n", delta_Z[trans-1]);
printf("Would you like to change it? (y/n)\n");
answer = yes_or_no();
if (answer == YES)
{
    printf("Enter new value:\n");
    delta_Z[trans-1] = check_error_double();
}

printf("The rotation angle around the x axis is %8.4f (deg).\n", x[trans-1]*180.0/PI);
printf("Would you like to change it? (y/n)\n");
answer = yes_or_no();
if (answer == YES)
{
    printf("Enter new value (in degrees):\n");
    x[trans-1] = check_error_double();
    x[trans-1] = x[trans-1]*PI/180.0;
}

printf("Angular tolerance with respect to x axis is %8.5f (deg)\n", delta_x[trans-1]*180.0/PI);
printf("Would you like to change it? (y/n)\n");
answer = yes_or_no();
if (answer == YES)
{
    printf("Enter new value (in degrees):\n");
    delta_x[trans-1] = check_error_double();
    delta_x[trans-1] = delta_x[trans-1]*PI/180.0;
}

printf("The rotation angle around the y axis is %8.4f (deg)\n", y[trans-1]*180.0/PI);
printf("Would you like to change it? (y/n)\n");

130
answer = yes_or_no();
if (answer == YES)
{
    printf("Enter new value (in degrees): \n");
    y[trans-1] = check_error_double();
    y[trans-1] = y[trans-1]*PI/180.0;
}

printf("Angular tolerance with respect to y axis is %8.5f (deg)\n", delta_y[trans-1]*180.0/PI);
printf("Would you like to change it? (y/n)\n");
answer = yes_or_no();
if (answer == YES)
{
    printf("Enter new value (in degrees): \n");
    delta_y[trans-1] = check_error_double();
    delta_y[trans-1] = delta_y[trans-1]*PI/180.0;
}

printf("The rotation angle around the z axis is %8.4f (deg).\n", z[trans-1]*180.0/PI);
printf("Would you like to change it? (y/n)\n");
answer = yes_or_no();
if (answer == YES)
{
    printf("Enter new value (in degrees): \n");
    z[trans-1] = check_error_double();
    z[trans-1] = z[trans-1]*PI/180.0;
}

printf("Angular tolerance with respect to z axis is %8.5f (deg)\n", delta_z[trans-1]*180.0/PI);
printf("Would you like to change it? (y/n)\n");
answer = yes_or_no();
if (answer == YES)
{
    printf("Enter new value (in degrees): \n");
    delta_z[trans-1] = check_error_double();
    delta_z[trans-1] = delta_z[trans-1]*PI/180.0;
}

printf("\nThe data you entered for the transformation
#d IS: \n", trans);
printf("Translation: \n"," \n","X = %8.3f, Y = %8.3f, Z = %8.3f\n", X[trans-1], Y[trans-1], Z[trans-1]);
printf("Tolerance (+/-) :X = %8.5f, Y = %8.5f, Z = %8.5f\n", del_X[trans-1], del_Y[trans-1], delta_Z[trans-1]);
printf("Rotation (deg) : \n"," \n","x = %8.4f, y = %8.4f, z = %8.4f\n", x[trans-1]*180.0/PI, y[trans-1]*180.0/PI, z[trans-1]*180.0/PI);
printf("Angular toler.: x = %8.5f, y = %8.5f, z = %8.5f\n", delta_x[trans-1]*180.0/PI, delta_y[trans-1]*180.0/PI, delta_z[trans-1]*180.0/PI);
if ( distr_type_X[trans-1] != NRM &&
    distr_type_Y[trans-1] != NRM )
    printf("All tolerance distributions are normal
except for X and Y, which are uniform."
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
will not be\naccurate; please change the units (but remember
to be\nconsistent throughout the whole data entry).\n
if (no_previous_error)
{
    printf("Do you want to make any other changes to
present data?\n");
    answer = yes_or_no();
}

printf("Would you like to add an additional
transformation?\n");
answer = yes_or_no();

 /* The whole thing is done if(*demo == YES) */

if (*demo == NO)
{
    printf("Please enter the sequence of your
transformations.\n");
    *num_of_frames = -1;
}

for( trans = (*num_of_frames) + 1; answer == YES; trans++)
{
    do
    {
        if (no_previous_error)
        {
            printf("THE NEXT TRANSFORMATION NUMBER IS
\n\n", 
                   trans+1);
            no_previous_error = 1;
        }
        printf("Enter the translation in X direction.\n");
        X[trans] = check_error_double();
        printf("Enter the tolerance on X translation.\n");
        printf("It can be normally or uniformly
distributed.\n");
        printf("Enter the value of X tolerance:\n");
        del_X[trans] = check_error_double();
        if (del_X[trans] != 0.0)
        {
            printf("Enter 'n' for normal, or 'u' for
uniform distribution...\n");
            answer = is_uniform();
            if (answer == YES)
                distr_type_X[trans] = UNI;
            else
                distr_type_X[trans] = NRM;
        }
        else
            distr_type_X[trans] = NRM;
        printf("Enter the translation in Y direction.\n");
        Y[trans] = check_error_double();
        printf("Enter the tolerance on Y translation.\n");
        printf("It can be normally or uniformly
distributed.\n");
    }
}


distr_type_Y = NRM;
printf("Enter the translation in Z direction.\n");
Z[trans] = check_error_double();
printf("Enter the tolerance on Z translation.\n");
printf("It can be normally or uniformly
distributed.\n");
printf("Enter the value of Y tolerance:\n");
del_Y[trans] = check_error_double();
if (del_Y[trans] != 0.0)
{  
    printf("Enter 'n' for normal, or 'u' for uniform distribution...\n");  
    answer = is_uniform();  
    if (answer == YES)
        distr_type_Y[trans] = UNI;
    else
        distr_type_Y[trans] = NRM;
}
else
    distr_type_Y[trans] = NRM;
printf("Enter the translation in Z direction.\n");
Z[trans] = check_error_double();
printf("Enter the tolerance on Z translation.\n");
delta_Z[trans] = check_error_double();
printf("Enter the rotation angle about the x axis (in degrees).\n");
x[trans] = check_error_double();
printf("Enter the angular tolerance about the x axis (in degrees).\n");
delta_x[trans] = check_error_double();
printf("Enter the rotation angle about the y axis (in degrees).\n");
y[trans] = check_error_double();
printf("Enter the angular tolerance about the y axis (in degrees).\n");
delta_y[trans] = check_error_double();
printf("Enter the rotation angle about the z axis (in degrees).\n");
z[trans] = check_error_double();
printf("Enter the angular tolerance about the z axis (in degrees).\n");
delta_z[trans] = check_error_double();
printf("\nYou have entered the following data.\n");
printf("\nTRANSFORMATION #: %d\n", trans+1);
printf("Translation:  \n  X = %8.3f, Y = %8.3f, Z = %8.3f\n", X[trans], Y[trans], Z[trans]);
printf("Tolerance (+/-):X = %8.5f, Y = %8.5f, Z = %8.5f\n", del_X[trans], del_Y[trans], delta_Z[trans]);
printf("Rotation (deg):  x = %8.4f, y = %8.4f, z = %8.4f\n", x[trans], y[trans], z[trans]);
printf("Angular toler.: x = %8.5f, y = %8.5f, z = %8.5f\n", delta_x[trans], delta_y[trans], delta_z[trans]);
if ( distr_type_X[trans] != NRM &&
    distr_type_Y[trans] != NRM)
    printf("All tolerance distributions are normal except for X and Y, which are uniform.\n")
else if ( distr_type_X[trans] != NRM )
    printf("All tolerance distributions are normal

134
except for $X$, which is uniform.

```
else if (distr_type_Y[trans] != NRM)
    printf("All tolerance distributions are normal
except for $Y$, which is uniform.

```
else
    printf("All tolerance distributions are

```
\texttt{x[trans] = x[trans]*PI/180.0;}
\texttt{delta_x[trans] = delta_x[trans]*PI/180.0;}
\texttt{y[trans] = y[trans]*PI/180.0;}
\texttt{delta_y[trans] = delta_y[trans]*PI/180.0;}
\texttt{z[trans] = z[trans]*PI/180.0;}
\texttt{delta_z[trans] = delta_z[trans]*PI/180.0;}
\texttt{z[trans] == 0.0)}
```

```
    printf("\n\nWARNING: For the program to work
properly, there must be a transformation between two
frames. Therefore, at least one coordinate of linear or
angular transformation ($X$, $Y$, $Z$, $x$, $y$, $z$) must be non-zero (it
may be very small). Please re-enter the data for that
frame.\n\n"");

```answer = NO;
no_previous_error = 0;
```

```
if ( (distr_type_X[trans] == NRM &&
del_X[trans] != 0 &&
( del_X[trans] > 30.0 ||
del_X[trans] < 0.000003 ) ) ||
( distr_type_Y[trans] == NRM &&
del_Y[trans] != 0 &&
( del_Y[trans] > 30.0 ||
del_Y[trans] < 0.000003 ) ) ||
( delta_Z[trans] != 0.0 &&
( delta_Z[trans] > 30.0 ||
delta_Z[trans] < 0.000003 ) ) ||
( delta_x[trans] != 0.0 &&
( delta_x[trans] > 30.0 ||
delta_x[trans] < 0.000003 ) ) ||
( delta_y[trans] != 0.0 &&
( delta_y[trans] > 30.0 ||
delta_y[trans] < 0.000003 ) ) ||
( delta_z[trans] != 0.0 &&
( delta_z[trans] > 30.0 ||
delta_z[trans] < 0.000003 ) )
```

```
    printf("\n\nCAUTION: The resolution of the
program is 0.000003 for the smallest acceptable tolerance and
30.0 for the largest. This is in the current units for
linear variables, and in radians for angular variables. The
restriction does not apply to uniformly
distributed tolerances. The 0.0 tolerance is
acceptable. If a tolerance $<$ 0.000003 is detected, it will
be set to 0.\n\n```

```
If you entered tolerance $>$ 30.0, the results
```

```
135
```

will not be accurate; please change the units (but remember
to be\nconsistent throughout the whole data entry).

    if (no_previous_error)
    {
        printf("Is this information correct?\n");
        answer = yes_or_no();
    }

} while(answer == NO);

if (trans == MAX_FRAME_NUM - 1)
{
    printf("\nYou have just entered the maximum number
(%d) of transformations that the program allows.\n", MAX_FRAME_NUM);
    break;
}

printf("\nWould you like to add an additional
transformation?\n");
    answer = yes_or_no();
}

trans = trans-1;
for(j = 0; j <= trans; j++)
    for (type = UNI; type <= NRM; type++)
    {
        delta_X[j][type] = 0.0;
        delta_Y[j][type] = 0.0;
    }

for(j = 0; j <= trans; j++)
{
    delta_X[j][ distr_type_X[j] ] = del_X[j];
    delta_Y[j][ distr_type_Y[j] ] = del_Y[j];
}

*frame_num = trans;
*num_of_frames = trans;
for (i = 0; i <= *num_of_frames; i++)
    {
    /* If tolerances are very
    small, set them to 0.
    3e-6 is the smallest
    number that can be
    generated by the Monte-Carlo function */

    if(delta_X[i][NRM] < 0.0000003)
        delta_X[i][NRM] = 0.0;
    if(delta_Y[i][NRM] < 0.0000003)
        delta_Y[i][NRM] = 0.0;
    if(delta_Z[i] < 0.0000003)
        delta_Z[i] = 0.0;
    if(delta_x[i] < 0.0000003)
        delta_x[i] = 0.0;
    if(delta_y[i] < 0.0000003)
        delta_y[i] = 0.0;
    if(delta_z[i] < 0.0000003)
        delta_z[i] = 0.0;
printf("Would you like to save your data to a new file for future use?\n");
answer = yes_or_no();
if (answer == YES)
{
    printf("Please enter the name of the new file.\n");
    printf("CAUTION: If you use the name of an already existing file, the old version will be erased.\n");
    check_error_string(filename);
    file_ptr = fopen(filename, "w");
    fprintf(file_ptr, "%d\n", trans);
    for (i = 0; i <= *num_of_frames; i++)
    {
        fprintf(file_ptr, "%17.11lf\n", X[i]);
        for (type = UNI; type <= NRM; type++)
            fprintf(file_ptr, "%17.13lf\n",
                    delta_X[i][type]);
        fprintf(file_ptr, "%17.11lf\n", Y[i]);
        for (type = UNI; type <= NRM; type++)
            fprintf(file_ptr, "%17.13lf\n",
                    delta_Y[i][type]);
        fprintf(file_ptr, "%17.11lf\n", Z[i]);
        fprintf(file_ptr, "%17.13lf\n", delta_Z[i]);
        fprintf(file_ptr, "%17.11lf\n", x[i]);
        fprintf(file_ptr, "%17.13lf\n", delta_x[i]);
        fprintf(file_ptr, "%17.11lf\n", y[i]);
        fprintf(file_ptr, "%17.13lf\n", delta_y[i]);
        fprintf(file_ptr, "%17.11lf\n", z[i]);
        fprintf(file_ptr, "%17.13lf\n", delta_z[i]);
    }
    fclose(file_ptr);
    printf("The file has been saved under the name: %s\n",filename);
    printf("Keep the name handy for future file retrieval.\n");
}
*demo = YES;
}
int z[MAX_FRAME_NUM], delta_z[MAX_FRAME_NUM];

*num_of_frames, *frame_num;

FILE *file_ptr;
char filename[20];
void check_error_string();
int i, type, temp, answer = 1;
double scX[MAX_FRAME_NUM], scdelta_X[MAX_FRAME_NUM][2],
scY[MAX_FRAME_NUM], scdelta_Y[MAX_FRAME_NUM][2],
scZ[MAX_FRAME_NUM], scdelta_Z[MAX_FRAME_NUM],
scx[MAX_FRAME_NUM], scdelta_x[MAX_FRAME_NUM],
scy[MAX_FRAME_NUM], scdelta_y[MAX_FRAME_NUM],
scz[MAX_FRAME_NUM], scdelta_z[MAX_FRAME_NUM];

do {
    printf("Please enter the name of the existing file...\n");
    check_error_string(filename);
    if ((file_ptr = fopen(filename, "r")) != NULL) {
        fscanf(file_ptr, "%d\n", &temp);
        *num_of_frames = temp;
        *frame_num = temp;
        for (i = 0; i <= *num_of_frames; i++) {
            fscanf(file_ptr, "%lf\n", &scX[i]);
            X[i] = scX[i];
            for (type = UNI; type <= NRM; type++) {
                fscanf(file_ptr, "%lf\n",
                        &scdelta_X[i][type]);
                delta_X[i][type] = scdelta_X[i][type];
            }
            fscanf(file_ptr, "%lf\n", &scY[i]);
            Y[i] = scY[i];
            for (type = UNI; type <= NRM; type++) {
                fscanf(file_ptr, "%lf\n",
                        &scdelta_Y[i][type]);
                delta_Y[i][type] = scdelta_Y[i][type];
            }
            fscanf(file_ptr, "%lf\n", &scZ[i]);
            Z[i] = scZ[i];
            fscanf(file_ptr, "%lf\n", &scdelta_Z[i]);
            delta_Z[i] = scdelta_Z[i];
            fscanf(file_ptr, "%lf\n", &scx[i]);
            x[i] = scx[i];
            fscanf(file_ptr, "%lf\n", &scdelta_x[i]);
            delta_x[i] = scdelta_x[i];
            fscanf(file_ptr, "%lf\n", &scy[i]);
            y[i] = scy[i];
            fscanf(file_ptr, "%lf\n", &scdelta_y[i]);
        }
    }
}

138
delta_y[i] = scdelta_y[i];
fscanf(file_ptr,"%lf\n", &scz[i]);
z[i] = scz[i];
fscanf(file_ptr,"%lf\n", &scdelta_z[i]);
delta_z[i] = scdelta_z[i];
}
fclose(file_ptr);
answer = 0;
}
else
{
printf("The file 's' does not exist\n", filename);
printf("Press 0 to exit, any number to continue.\n");
answer = check_error_dec();
}
} while (answer != 0);


int is_uniform()
{
char ch;
while( ((ch = getchar()) != 'n') && (ch != 'u') )
    if (ch != '\n')
        printf("Please enter 'n' or 'u'...\n");
if ( ch == 'u')
    return(YES);
else
    return(NO);
}

int yes_or_no()
{
char ch;
while( ((ch = getchar()) != 'y') && (ch != 'n') )
    if (ch != '\n')
        printf("Please enter 'y' or 'n'...\n");
if ( ch == 'y')
    return(YES);
else
  return(NO);
}

;/*                        */
/* User chooses to display another frame. Check if the */
/* desired frame number is valid */
/*                        */

int get_another_frame( num_of_frames )
int num_of Frames;
{
  int frame, frame_num;
  printf("\nTHE TOTAL NUMBER OF FRAMES ENTERED IS %d\n", 
  num_of_frames + 1);
  do
  {
      printf("Please enter the frame number between 1 and 
    %d\n", num_of_frames + 1);
      frame = check_error_dec();
      while ( frame < 1 | | frame > num_of_frames + 1 );
  frame_num = frame - 1;
  return( frame_num );
  }

;/*                        */
/* Display the menu and get a number for a validity check. */
/*                        */

void menu( choice )
int *choice;
{
  int what_you_want, wrong_choice(), check_error_dec();
  do
  {
    printf("\nYOU CAN DO THE FOLLOWING, BY ENTERING THE 
    NUMBER...\n") ;
    printf("Change the existing data, or add another frame 
    ----> 1\n");
    printf("Display the distribution for another frame ----
    ----> 2\n");
    printf("Start the Monte-Carlo simulation ---------------
    ----> 3\n");
    printf("Get data from an existing file -----------------
    ----> 4\n");
    printf("Exit -------------------------------
    ----> 0\n");
    what_you_want = check_error_dec();
    } while ( wrong_choice( what_you_want ) == YES );

140
*choice = what_you_want ;
}

/* *************************************************************************/
/* Check if the number entered in the menu is acceptable */
/* *************************************************************************/

int wrong_choice( what_you_want )
int what_you_want;
{
if( what_you_want < 0 || what_you_want > 4 || what_you_want == EOF)
    return (YES);
else
    return (NO);
}

/* *************************************************************************/
/* Find partial derivatives of the transform matrix Ai with respect to X, Y, Z, x, y, z coordinates, for the particular frame number 'n' */
/* *************************************************************************/

void derivatives(x, y, z, n, derAX, derAY, derAZ, derAx, derAy, derAz)
double x[MAX_FRAME_NUM], y[MAX_FRAME_NUM], z[MAX_FRAME_NUM], 
derAX[4][4][MAX_FRAME_NUM], derAY[4][4][MAX_FRAME_NUM], 
derAZ[4][4][MAX_FRAME_NUM], derAx[4][4][MAX_FRAME_NUM], 
derAy[4][4][MAX_FRAME_NUM], derAz[4][4][MAX_FRAME_NUM];

int n;
{
    int i, j;

    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            { 
                derAX[i][j][n] = 0.0;
                derAY[i][j][n] = 0.0;
                derAZ[i][j][n] = 0.0;
            }
    derAX[0][3][n] = 1.0;
    derAY[1][3][n] = 1.0;
    derAZ[2][3][n] = 1.0;

    for (j = 0; j < 4; j++)
        { 
            derAx[3][j][n] = 0.0;
            derAy[3][j][n] = 0.0;
            derAz[3][j][n] = 0.0;
            derAx[j][3][n] = 0.0;
        }
```
derAy[j][3][n] = 0.0;
derAz[j][3][n] = 0.0;
}
for (j = 0; j < 3; j++)
    derAx[0][j][n] = 0.0;
derAx[1][0][n] = cos(x[n]) * cos(z[n]) * sin(y[n]) -
    sin(x[n]) * sin(z[n]);
derAx[1][1][n] = -cos(z[n]) * sin(x[n]) -
    cos(x[n]) * sin(y[n]) * sin(z[n]);
derAx[1][2][n] = -cos(x[n]) * cos(y[n]);
derAx[2][0][n] = cos(z[n]) * sin(x[n]) * sin(y[n]) +
    cos(x[n]) * sin(z[n]);
derAx[2][1][n] = cos(x[n]) * cos(z[n]) -
    sin(x[n]) * sin(y[n]) * sin(z[n]);
derAx[2][2][n] = -cos(y[n]) * sin(x[n]);
derAy[0][0][n] = -cos(z[n]) * sin(y[n]);
derAy[0][1][n] = sin(y[n]) * sin(z[n]);
derAy[0][2][n] = cos(y[n]);
derAy[1][0][n] = cos(y[n]) * cos(z[n]) * sin(x[n]);
derAy[1][1][n] = -cos(y[n]) * sin(x[n]) * sin(z[n]);
derAy[1][2][n] = sin(x[n]) * sin(y[n]);
derAy[2][0][n] = -cos(x[n]) * cos(y[n]) * cos(z[n]);
derAy[2][1][n] = cos(x[n]) * cos(y[n]) * sin(z[n]);
derAy[2][2][n] = -cos(x[n]) * sin(y[n]);
for (i = 0; i < 3; i++)
    derAz[i][2][n] = 0.0;
derAz[0][0][n] = -cos(y[n]) * sin(z[n]);
derAz[0][1][n] = -cos(y[n]) * cos(z[n]);
derAz[1][0][n] = cos(x[n]) * cos(z[n]) -
    sin(x[n]) * sin(y[n]) * sin(z[n]);
derAz[1][1][n] = -cos(z[n]) * sin(x[n]) * sin(y[n]) -
    cos(x[n]) * sin(z[n]);
derAz[2][0][n] = cos(z[n]) * sin(x[n]) +
    cos(x[n]) * sin(y[n]) * sin(z[n]);
derAz[2][1][n] = cos(x[n]) * cos(z[n]) * sin(y[n]) -
    sin(x[n]) * sin(z[n]);
}

/* *********************************************** */
/* Inverse of the Ai matrix was found in the closed form. */
/* *********************************************** */

void Ai_inverse(X, Y, Z, x, y, z, i, AinvtId)
    double X[MAX_FRAME_NUM], Y[MAX_FRAME_NUM], Z[MAX_FRAME_NUM],
    x[MAX_FRAME_NUM], y[MAX_FRAME_NUM], z[MAX_FRAME_NUM],
    AinvtId[4][4][MAX_FRAME_NUM];
int i;
{
    AinvtId[0][0][i] = cos(y[i]) * cos(z[i]);
```
AinvrtD[0][1][i] = cos(z[i])*sin(x[i])*sin(y[i]) +
    cos(x[i])*sin(z[i]);
AinvrtD[0][2][i] = -cos(x[i])*cos(z[i])*sin(y[i]) +
    sin(x[i])*sin(z[i]);
AinvrtD[0][3][i] = (- cos(z[i])*sin(x[i])*sin(y[i]) +
    cos(x[i])*sin(z[i]))*y[i] + ( cos(x[i])*cos(z[i])*sin(y[i]) -
    sin(x[i])*sin(z[i]))*z[i] - X[i]*cos(y[i])*cos(z[i]);
AinvrtD[1][0][i] = -cos(y[i])*sin(z[i]);
AinvrtD[1][1][i] = cos(x[i])*cos(z[i]) -
    sin(x[i])*sin(y[i])*sin(z[i]);
AinvrtD[1][2][i] = cos(z[i])*sin(x[i]) +
    cos(x[i])*sin(y[i])*sin(z[i]);
AinvrtD[1][3][i] = ( -cos(x[i])*cos(z[i]) +
    sin(x[i])*sin(y[i])*sin(z[i]))*y[i] - ( cos(z[i])*sin(x[i]) +
    cos(x[i])*sin(y[i])*sin(z[i]))*z[i] +
    X[i]*cos(y[i])*sin(z[i]);
AinvrtD[2][0][i] = sin(y[i]);
AinvrtD[2][1][i] = -cos(y[i])*sin(x[i]);
AinvrtD[2][2][i] = cos(x[i])*cos(y[i]);
AinvrtD[2][3][i] = -z[i]*cos(x[i])*cos(y[i]) +
    y[i]*cos(y[i])*sin(x[i]) - X[i]*sin(y[i]);
AinvrtD[3][0][i] = 0.0;
AinvrtD[3][1][i] = 0.0;
AinvrtD[3][2][i] = 0.0;
AinvrtD[3][3][i] = 1.0;
}

/* * Find all six D matrices */
/* ****************************************************** */
void D_matrices(derAX, derAY, derAZ, derAx, derAy, derAz, n, 
    AinvrtD, DX, DY, DZ, Dx, Dy, Dz)
double derAX[4][4][MAX_FRAME_NUM],
    derAY[4][4][MAX_FRAME_NUM], derAZ[4][4][MAX_FRAME_NUM],
    derAx[4][4][MAX_FRAME_NUM], derAy[4][4][MAX_FRAME_NUM],
    derAz[4][4][MAX_FRAME_NUM],
    AinvrtD[4][4][MAX_FRAME_NUM], DX[4][4][MAX_FRAME_NUM],
    DY[4][4][MAX_FRAME_NUM], DZ[4][4][MAX_FRAME_NUM],
    Dx[4][4][MAX_FRAME_NUM], Dy[4][4][MAX_FRAME_NUM],
    Dz[4][4][MAX_FRAME_NUM];
int n;

int i, j;
void multiply();

for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++)
        {
            temp2[i][j] = AinvrtD[i][j][n];
            /* for the the purpose of the 'multiply'
fcn, which does not work with three dimensional matrices */
  temp1[i][j] = derAX[i][j][n];
}
multiply(temp1, temp2, temp_out);
for (i = 0; i < 4; i++)
  for (j = 0; j < 4; j++)
    DX[i][j][n] = temp_out[i][j];

for (i = 0; i < 4; i++)
  for (j = 0; j < 4; j++)
    temp1[i][j] = derAY[i][j][n];
multiply(temp1, temp2, temp_out);
for (i = 0; i < 4; i++)
  for (j = 0; j < 4; j++)
    DZ[i][j][n] = temp_out[i][j];

for (i = 0; i < 4; i++)
  for (j = 0; j < 4; j++)
    temp1[i][j] = derAZ[i][j][n];
multiply(temp1, temp2, temp_out);
for (i = 0; i < 4; i++)
  for (j = 0; j < 4; j++)
    DZ[i][j][n] = temp_out[i][j];

for (i = 0; i < 4; i++)
  for (j = 0; j < 4; j++)
    temp1[i][j] = derAx[i][j][n];
multiply(temp1, temp2, temp_out);
for (i = 0; i < 4; i++)
  for (j = 0; j < 4; j++)
    Dx[i][j][n] = temp_out[i][j];

for (i = 0; i < 4; i++)
  for (j = 0; j < 4; j++)
    temp1[i][j] = derAy[i][j][n];
multiply(temp1, temp2, temp_out);
for (i = 0; i < 4; i++)
  for (j = 0; j < 4; j++)
    Dy[i][j][n] = temp_out[i][j];

for (i = 0; i < 4; i++)
  for (j = 0; j < 4; j++)
    temp1[i][j] = derAz[i][j][n];
multiply(temp1, temp2, temp_out);
for (i = 0; i < 4; i++)
  for (j = 0; j < 4; j++)
    Dz[i][j][n] = temp_out[i][j];

/* ***************************************** */
/* The function computes the m vectors for ith frames, as an
void m_vectors( DX, DY, DZ, Dx, Dy, Dz, i, m1, n2, m3, m4, m5, m6, m7, m8, m9 )

double DX[4][4][MAX_FRAME_NUM], DY[4][4][MAX_FRAME_NUM],
DZ[4][4][MAX_FRAME_NUM], Dx[4][4][MAX_FRAME_NUM],
Dy[4][4][MAX_FRAME_NUM], Dz[4][4][MAX_FRAME_NUM],
m1[3][MAX_FRAME_NUM], m2[3][MAX_FRAME_NUM],
m3[3][MAX_FRAME_NUM], m4[3][MAX_FRAME_NUM],
m5[3][MAX_FRAME_NUM], m6[3][MAX_FRAME_NUM],
m7[3][MAX_FRAME_NUM], m8[3][MAX_FRAME_NUM],
m9[3][MAX_FRAME_NUM];

int i;
{
    m1[0][i] = DX[0][3][i];
    m1[1][i] = DX[1][3][i];
    m1[2][i] = DX[2][3][i];

    m2[0][i] = DY[0][3][i];
    m2[1][i] = DY[1][3][i];
    m2[2][i] = DY[2][3][i];

    m3[0][i] = DZ[0][3][i];
    m3[1][i] = DZ[1][3][i];
    m3[2][i] = DZ[2][3][i];

    m4[0][i] = Dx[0][3][i];
    m4[1][i] = Dx[1][3][i];
    m4[2][i] = Dx[2][3][i];

    m5[0][i] = Dy[0][3][i];
    m5[1][i] = Dy[1][3][i];
    m5[2][i] = Dy[2][3][i];

    m6[0][i] = Dz[0][3][i];
    m6[1][i] = Dz[1][3][i];
    m6[2][i] = Dz[2][3][i];

    m7[0][i] = .5 * (Dx[2][1][i] - Dx[1][2][i]);
    m7[1][i] = .5 * (Dx[0][2][i] - Dx[2][0][i]);
    m7[2][i] = .5 * (Dx[1][0][i] - Dx[0][1][i]);

    m8[0][i] = .5 * (Dy[2][1][i] - Dy[1][2][i]);
    m8[1][i] = .5 * (Dy[0][2][i] - Dy[2][0][i]);
    m8[2][i] = .5 * (Dy[1][0][i] - Dy[0][1][i]);

    m9[0][i] = .5 * (Dz[2][1][i] - Dz[1][2][i]);
    m9[1][i] = .5 * (Dz[0][2][i] - Dz[2][0][i]);
    m9[2][i] = .5 * (Dz[1][0][i] - Dz[0][1][i]);
}
/ * ****************************************************************** */
/* The following function computes the elements of the 3x3 rotation matrix R for ith transformation */
/* ****************************************************************** */

void R_matrix( x, y, z, R, i)
{ double x[MAX_FRAME_NUM], y[MAX_FRAME_NUM], z[MAX_FRAME_NUM], R[3][3][MAX_FRAME_NUM];
  int i; /* i is the particular frame number */
  
  R[0][0][i] = cos(y[i])*cos(z[i]);
  R[0][1][i] = -cos(y[i])*sin(z[i]);
  R[0][2][i] = sin(y[i]);
  R[1][0][i] = cos(z[i])*sin(x[i])*sin(y[i]) +
         cos(x[i])*sin(z[i]);
  R[1][1][i] = cos(x[i])*cos(z[i]) -
         sin(x[i])*sin(y[i])*sin(z[i]);
  R[1][2][i] = -cos(y[i])*sin(x[i]);
  R[2][0][i] = -cos(x[i])*cos(z[i])*sin(y[i]) +
         cos(x[i])*sin(z[i]);
  R[2][1][i] = cos(z[i])*sin(x[i]) +
         cos(x[i])*sin(y[i])*sin(z[i]);
  R[2][2][i] = cos(x[i])*cos(y[i]);
}

void p_vector(X, Y, Z, p, i)
{ double X[MAX_FRAME_NUM], Y[MAX_FRAME_NUM], Z[MAX_FRAME_NUM], p[3][MAX_FRAME_NUM];
  int i; /* i is the frame number */
  
  p[0][i] = X[i];
  p[1][i] = Y[i];
  p[2][i] = Z[i];
}

/* 4 x 4 matrix multiplication */
/* ****************************************************************** */

void multiply( matrix1, matrix2, matrix_out )
{ double matrix1[4][4], matrix2[4][4], matrix_out[4][4];
  
  double sum = 0.0;
  int i, j, k;

  146
for (i = 0; i < 4; i++)
{
    for (k = 0; k < 4; k++)
    {
        for (j = 0; j < 4; j++)
        {
            sum += (matrix1[i][j]) * (matrix2[j][k]);
        }
        matrix_out[i][k] = sum;
        sum = 0.0;
    }
}

/* *********************************************** */
/* This function finds the nominal position of the point */
/* after undergoing N transformations (determined by */
/* the variable "frame_num") */
/* *********************************************** */

void ptN_vector(X, Y, Z, x, y, z, delta_x, delta_y, delta_z, 
ptN, i)
double X[MAX_FRAME_NUM], Y[MAX_FRAME_NUM], Z[MAX_FRAME_NUM], 
    x[MAX_FRAME_NUM], y[MAX_FRAME_NUM], z[MAX_FRAME_NUM], 
    delta_x[MAX_FRAME_NUM], delta_y[MAX_FRAME_NUM], 
    delta_z[MAX_FRAME_NUM], ptN[3];
int i;    /* i is the Nth frame number */
{
    int j, k, count;
    double temp[4][4], next_temp[4][4], out_temp[4][4], 
        /* these are temporary matrices */
        introduced for the purpose of using 
        the multiplication subroutine */
    R[3][3][MAX_FRAME_NUM], p[3][MAX_FRAME_NUM];
    /* rotation matrix, and translation vector */
    void multiply(), R_matrix(), p_vector();
    /* functions to be utilized here */
    double Xtemp[MAX_FRAME_NUM], Ytemp[MAX_FRAME_NUM], 
        Ztemp[MAX_FRAME_NUM];
    int starting_point;    /* this is where the angular errors */
        /* are beginning to be effective */
    starting_point = i+1;

    for (count = 0; count <= i; count++)
    {
        if (delta_x[count] > 0.000003 || delta_y[count] > 
            0.000003 || delta_z[count] > 0.000003) 
            starting_point = count;

        if (starting_point != i+1)
        {
            for (j = 0; j < 4; j++)
            {
                
            
        


for (k = 0; k < 4; k++)
{
    if (j == k)
        temp[j][k] = 1.0;  /* within these two loops
        the first temp matrix is
        initialized */
    else
        temp[j][k] = 0.0;
}

for (count = starting_point; count <= i; count++)
{
    Xtemp[count] = fabs(X[count]);
    Ytemp[count] = fabs(Y[count]);
    Ztemp[count] = fabs(Z[count]);
}

for (count = starting_point; count <= i; count++)
    /*Here, the propagation of nominally
    positioned matrices is performed, "count"
    is the last frame number for the
    propagated frames' system */
{
    R_matrix(x, y, z, R, count);
    p_vector(Xtemp, Ytemp, Ztemp, p, count);
    for (j = 0; j < 3; j++)
        for (k = 0; k < 3; k++)
            next_temp[j][k] = R[j][k][count];
    for (j = 0; j < 3; j++)
        next_temp[j][3] = p[j][count];
    for (j = 0; j < 3; j++)
        next_temp[3][j] = 0.0;
    next_temp[3][3] = 1.0;
    multiply(temp, next_temp, out_temp);
    /* the multiplication fcn is called */
    for (j = 0; j < 4; j++)
        for (k = 0; k < 4; k++)
            temp[j][k] = out_temp[j][k];

    for (j = 0; j < 3; j++)
        ptN[j] = out_temp[j][3];
    /* and the ptN vector is the last 3x1 column
    of the resulting temporary output matrix */
}
else
{
    ptN[0] = 0.0;
    ptN[1] = 0.0;
    ptN[2] = 0.0;
}

*/  ****************************************************************** */
/* The function multiplies a 3x3 matrix times 3x1 vector, and returns "vector_out" */
/* ****************************************************************************** */

void mult_matr_vect( matrix, vector_in, vector_out)
double matrix[3][3], vector_in[3], vector_out[3];
{
    int i, j;
    for (i = 0; i < 3; i++)
        vector_out[i] = 0.0;    /* first initialize... */
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            vector_out[i] += matrix[i][j] * vector_in[j];
}

/* ****************************************************************************** */
/* The function to compute the cross product of two 3x1 vectors */
/* ****************************************************************************** */

void cross_prod( vect1, vect2, vect_out )
double vect1[3], vect2[3], vect_out[3];
{
    vect_out[0] = vect1[1]*vect2[2] - vect2[1]*vect1[2];
    vect_out[1] = -vect1[0]*vect2[2] + vect2[0]*vect1[2];
    vect_out[2] = vect1[0]*vect2[1] - vect2[0]*vect1[1];
}

/* ****************************************************************************** */
/* This function computes R matrices, p and m vectors for each frame and stores them in appropriate arrays. "count" variable represents the total number of frames */
/* ****************************************************************************** */

void all_Rs_p and_ms(X, Y, Z, x, y, z, count, R, p, m1, m2,
m3, m4, m5, m6, m7, m8, m9)
double X[MAX_FRAME_NUM], Y[MAX_FRAME_NUM], Z[MAX_FRAME_NUM],
x[MAX_FRAME_NUM], y[MAX_FRAME_NUM], z[MAX_FRAME_NUM],
R[3][3][MAX_FRAME_NUM], p[3][MAX_FRAME_NUM],
m1[3][MAX_FRAME_NUM], m2[3][MAX_FRAME_NUM],
m3[3][MAX_FRAME_NUM], m4[3][MAX_FRAME_NUM],
m5[3][MAX_FRAME_NUM], m6[3][MAX_FRAME_NUM],
m7[3][MAX_FRAME_NUM], m8[3][MAX_FRAME_NUM],
m9[3][MAX_FRAME_NUM];

int count;    /* the total number of frames */
{
    int i;     /* loop counter */
    void R_matrix(), m_vectors(), p_vector(), derivatives(),
        AI_inverse(), D_matrices();
    /* the functions to be called */
    double derAX[4][4][MAX_FRAME_NUM],

149
for (i = 0; i <= count; i++)
{
R_matrix( x, y, z, R, i);
p_vector(X, Y, Z, p, i);
derivatives(x, y, z, i, derAX, derAY, derAZ, derAx, derAy, derAz);
Ai_inverse(X, Y, Z, x, y, z, i, Ainvrtd);
Ds_matrices(derAX, derAY, derAZ, derAx, derAy, derAz, i,
Ainvrtd, DX, DY, DZ, Dx, Dy, Dz);
m_vectors(DX, DX, DZ, Dx, Dy, Dz, i, m1, m2, m3, m4,
m5, m6, m7, m8, m9);
}

void all_Rs_ps_and_ms(X, Y, Z, x, y, z, count, R, p, m1, m2,
m3, m4, m5, m6, m7, m8, m9)
double X[MAX_FRAME_NUM], Y[MAX_FRAME_NUM], Z[MAX_FRAME_NUM],
x[MAX_FRAME_NUM], y[MAX_FRAME_NUM], z[MAX_FRAME_NUM],
R[3][3][MAX_FRAME_NUM], p[3][MAX_FRAME_NUM],
m1[3][MAX_FRAME_NUM], m2[3][MAX_FRAME_NUM],
m3[3][MAX_FRAME_NUM], m4[3][MAX_FRAME_NUM],
m5[3][MAX_FRAME_NUM], m6[3][MAX_FRAME_NUM],
m7[3][MAX_FRAME_NUM], m8[3][MAX_FRAME_NUM],
m9[3][MAX_FRAME_NUM];
int count;   /* the total number of frames */
{
int i;           /* loop counter */
void R_matrix(), m_vectors(), p_vector();
/* the functions to be called herein*/
for (i = 0; i <= count; i++)
{
R_matrix(x, y, z, R, i);
p_vector(X, Y, Z, p, i);
mI[0][i] = 1.0;
    /* m1, m2 and m3 vectors are constants */
m1[1][i] = 0.0;
m1[2][i] = 0.0;
m2[0][i] = 0.0;
m2[1][i] = 1.0;
m2[2][i] = 0.0;
m3[0][i] = 0.0;
m3[1][i] = 0.0;
m3[2][i] = 1.0;
m_vectors(X, Y, Z, x, y, z, m4, m5, m6, m7, m8, m9, i);
}

/* ******************************************** */
/* The function adds two 3x1 vectors */
/* ******************************************** */

void add_vectors(vector1, vector2, vector_out)
double vector1[3], vector2[3], vector_out[3];
{
int i;
for (i = 0; i < 3; i++)
        vector_out[i] = vector1[i] + vector2[i];
}

/* ******************************************** */
/* Compute the 12 W matrices for all the frames in the */
/* propagated chain */
/* ******************************************** */

void W_matrices(R, p, ptN, m1, m2, m3, m4, m5, m6, m7, m8,
        m9, count, W1, W2, W3, W4, W5, W6, W7, W8, W9, W10, W11,
        W12)
double R[3][3][MAX_FRAME_NUM], p[3][MAX_FRAME_NUM], ptN[3],
        m1[3][MAX_FRAME_NUM], m2[3][MAX_FRAME_NUM],
        m3[3][MAX_FRAME_NUM], m4[3][MAX_FRAME_NUM],
        m5[3][MAX_FRAME_NUM], m6[3][MAX_FRAME_NUM],
        m7[3][MAX_FRAME_NUM], m8[3][MAX_FRAME_NUM],
        m9[3][MAX_FRAME_NUM], W1[3][MAX_FRAME_NUM],
        W2[3][MAX_FRAME_NUM], W3[3][MAX_FRAME_NUM],
        W4[3][MAX_FRAME_NUM], W5[3][MAX_FRAME_NUM],
        W6[3][MAX_FRAME_NUM], W7[3][MAX_FRAME_NUM],
        W8[3][MAX_FRAME_NUM], W9[3][MAX_FRAME_NUM],
        W10[3][MAX_FRAME_NUM], W11[3][MAX_FRAME_NUM],
        W12[3][MAX_FRAME_NUM];
int count;
{
int frame, i, j;        /* indices */
double R_temp[3][3], m_temp[3], vect_out[3], p_temp[3],
        W_temp[3], crossed_vect[3], W_temp_added[3];
        /* temporary arrays */
void mult_matr_vect(), cross_prod(), add_vectors();
for (frame = 0; frame <= count; frame++)
{

151
/* because the functions mult_matr_vect(), cross_prod(),
add_vectors() take only 2 or 3 dimensional matrices, we
have to initialize all required temporary matrices.
First, discarding the third frame index */
for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
        R_temp[i][j] = R[i][j][frame];
for (i = 0; i < 3; i++)
    m_temp[i] = m1[i][frame];
mult_matr_vect( R_temp , m_temp, vect_out);
for (i = 0; i < 3; i++)
    W1[i][frame] = vect_out[i]; /* put them back to
the output array*/
for (i = 0; i < 3; i++)
    m temp [i] = m2[i][frame];
mult_matr_vect( R_temp , m_temp, vect_out);
for (i = 0; i < 3; i++)
    W2[i][frame] = vect_out[i];
for (i = 0; i < 3; i++)
    m_temp[i] = m3[i][frame];
mult_matr_vect( R_temp , m_temp, vect_out);
for (i = 0; i < 3; i++)
    W3[i][frame] = vect_out[i];
for (i = 0; i < 3; i++)
    m_temp[i] = m7[i][frame];
mult_matr_vect( R_temp , m_temp, vect_out);
for (i = 0; i < 3; i++)
    W5[i][frame] = vect_out[i];
for (i = 0; i < 3; i++)
    m_temp[i] = m8[i][frame];
mult_matr_vect( R_temp , m_temp, vect_out);
for (i = 0; i < 3; i++)
    W7[i][frame] = vect_out[i];
for (i = 0; i < 3; i++)
    m_temp[i] = m9[i][frame];
mult_matr_vect( R_temp , m_temp, vect_out);
for (i = 0; i < 3; i++)
    W9[i][frame] = vect_out[i];
for (i = 0; i < 3; i++)
    m_temp[i] = m4[i][frame];
mult_matr_vect( R_temp , m_temp, vect_out);
for (i = 0; i < 3; i++)
    p_temp[i] = p[i][frame];
for (i = 0; i < 3; i++)
    W_temp[i] = W5[i][frame];
cross_prod( p_temp, W_temp, crossed_vect);
add_vectors( vect_out, crossed_vect, W_temp_added);
for (i = 0; i < 3; i++)
    W4[i][frame] = W_temp_added[i];
cross_prod( W_temp, pTN, crossed_vect);
add_vectors( crossed_vect, W_temp_added, W_temp);
for (i = 0; i < 3; i++)
    W10[i][frame] = W_temp[i];
for (i = 0; i < 3; i++)
    m_temp[i] = m5[i][frame];
mult_matr_vect( R_temp, m_temp, vect_out);
for (i = 0; i < 3; i++)
    W_temp[i] = W7[i][frame];
cross_prod( p_temp, W_temp, crossed_vect);
add_vectors( vect_out, crossed_vect, W_temp_added);
for (i = 0; i < 3; i++)
    W6[i][frame] = W_temp_added[i];
cross_prod( W_temp, ptN, crossed_vect);
add_vectors( crossed_vect, W_temp_added, W_temp);
for (i = 0; i < 3; i++)
    W11[i][frame] = W_temp[i];
for (i = 0; i < 3; i++)
    m_temp[i] = m6[i][frame];
mult_matr_vect( R_temp, m_temp, vect_out);
for (i = 0; i < 3; i++)
    W_temp[i] = W9[i][frame];
cross_prod( p_temp, W_temp, crossed_vect);
add_vectors( vect_out, crossed_vect, W_temp_added);
for (i = 0; i < 3; i++)
    W8[i][frame] = W_temp_added[i];
cross_prod( W_temp, ptN, crossed_vect);
add_vectors( crossed_vect, W_temp_added, W_temp);
for (i = 0; i < 3; i++)
    W12[i][frame] = W_temp[i];
}

/* ******************************************************* */
/* The function takes the transpose of the 3xN W matrices */
/* before the Vp covariance matrix can be computed */
/* ******************************************************* */

void transpose(matrix_in, frame_num, matrix_out)
    double matrix_in[3][MAX_FRAME_NUM],
    matrix_out[MAX_FRAME_NUM][3];
    int frame_num;
/* the number of columns (N) is equal to the "frame_num" */
{
    int i, j;
    for (i = 0; i < 3; i++)
        for (j = 0; j <= frame_num; j++)
            matrix_out[j][i] = matrix_in[i][j];
}

/* ******************************************************* */
/* This function assigns the previously input tolerance */
/* ranges to 6 NxN diagonal V matrices. The diagonal elements */
/* are the variances in the appropriate frame and for */

153
appropriate X, Y, Z, x, y, z variables/
/* ****************************************** */

void V_matrices(delta_X, delta_Y, delta_Z, delta_x, delta_y,
    delta_z, frame_num, VX, VY, VZ, Vx, Vy, Vz)
double delta_X[MAX_FRAME_NUM][2], delta_Y[MAX_FRAME_NUM][2],
    delta_Z[MAX_FRAME_NUM], delta_x[MAX_FRAME_NUM],
    delta_y[MAX_FRAME_NUM], delta_z[MAX_FRAME_NUM],
    VX[MAX_FRAME_NUM][MAX_FRAME_NUM],
    VY[MAX_FRAME_NUM][MAX_FRAME_NUM],
    VZ[MAX_FRAME_NUM][MAX_FRAME_NUM],
    Vx[MAX_FRAME_NUM],
    Vy[MAX_FRAME_NUM],
    Vz[MAX_FRAME_NUM][MAX_FRAME_NUM],

int    frame_num;
{
    int i, j, type;
    for (i = 0; i <= frame_num; i++)
        for (j = 0; j <= frame_num; j++)
            {
                VX[i][j] = 0.0;
                VY[i][j] = 0.0;
                VZ[i][j] = 0.0;
                Vx[i][j] = 0.0;
                Vy[i][j] = 0.0;
                Vz[i][j] = 0.0;
            }
    for (i = 0; i <= frame_num; i++)
        {
            /* variances */
            for (type = UNI; type <= NRM; type++)
                {
                    if(type == UNI && delta_X[i][UNI] > 0.000003)
                        VX[i][i] = (RATIO*delta_X[i][UNI])*%
                                    (RATIO*delta_X[i][UNI]);
                    if(type == UNI && delta_Y[i][UNI] > 0.000003)
                        VY[i][i] = (RATIO*delta_Y[i][UNI])*%
                                    (RATIO*delta_Y[i][UNI]);
                    if(type == NRM && delta_X[i][NRM] > 0.000003)
                        VX[i][i] = (delta_X[i][NRM]/3)*%
                                    (delta_X[i][NRM]/3);
                    if(type == NRM && delta_Y[i][NRM] > 0.000003)
                        VY[i][i] = (delta_Y[i][NRM]/3)*%
                                    (delta_Y[i][NRM]/3);
                    VZ[i][i] = (delta_Z[i][3]/3)*(delta_Z[i][3]/3);
                    Vx[i][i] = (delta_x[i][3]/3)*(delta_x[i][3]/3);
                    Vy[i][i] = (delta_y[i][3]/3)*(delta_y[i][3]/3);
                    Vz[i][i] = (delta_z[i][3]/3)*(delta_z[i][3]/3);
                }
}
/* The function multiplies 3 by N matrices with N by N matrices, necessary operation for obtaining the covariance matrix Vp */

void mult_3xN_x_NxN(mat3Nin, matNN, N, mat3Nout)
double mat3Nin[3][MAX_FRAME_NUM],
    matNN[MAX_FRAME_NUM][MAX_FRAME_NUM],
    mat3Nout[3][MAX_FRAME_NUM];

int N;
{
int i, j, k;
double sum = 0.0;
for (i = 0; i < 3; i++)
    for (j = 0; j <= N; j++)
    {
        for (k = 0; k <= N; k++)
            sum += mat3Nin[i][k] * matNN[k][j];
        mat3Nout[i][j] = sum;
        sum = 0.0;
    }
}

/* The function multiplies 3 by N matrices with N by 3 matrices, necessary operation for obtaining the covariance matrix Vp */

void mult_3xN_x_Nx3(mat3N, matN3, N, mat33)
double mat3N[3][MAX_FRAME_NUM], matN3[MAX_FRAME_NUM][3],
    mat33[3][3];
int N;
{
int i, j, k;
double sum = 0.0;
for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
    {
        for (k = 0; k <= N; k++)
            sum += mat3N[i][k] * matN3[k][j];
        mat33[i][j] = sum;
        sum = 0.0;
    }
}

/* The function to add two 3x3 matrices */

/* */
void matrix_add(matrix1, matrix2, matrix_out)
double matrix1[3][3], matrix2[3][3], matrix_out[3][3];
{
    int i, j;
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            matrix_out[i][j] = matrix1[i][j] + matrix2[i][j];
}

/* ********************************************************** */
/* The function computes the covariance matrix of the */
/* positional dp vector for the given frame number and */
/* transformation chain */
/* ********************************************************** */

void Vp_matrix( VX, VY, VZ, Vx, Vy, Vz, W1, W2, W3, W10, W11, W12, N, Vp )
double VX[MAX_FRAME_NUM][MAX_FRAME_NUM],
    VY[MAX_FRAME_NUM][MAX_FRAME_NUM],
    VZ[MAX_FRAME_NUM][MAX_FRAME_NUM],
    Vx[MAX_FRAME_NUM][MAX_FRAME_NUM],
    Vy[MAX_FRAME_NUM][MAX_FRAME_NUM],
    Vz[MAX_FRAME_NUM][MAX_FRAME_NUM], W1[3][MAX_FRAME_NUM],
    W2[3][MAX_FRAME_NUM], W3[3][MAX_FRAME_NUM],
    W10[3][MAX_FRAME_NUM], W11[3][MAX_FRAME_NUM],
    W12[3][MAX_FRAME_NUM], Vp[3][3];

int N;
{
    double temp3xN[3][MAX_FRAME_NUM], tempNx3[MAX_FRAME_NUM][3],
        temp3x3[3][3];
    void mult_3xN_x_NxN(), transpose(), mult_3xN_x_Nx3(),
        matrix_add();
    /* the above lines list the functions called from here */
    mult_3xN_x_NxN(W1, VX, N, temp3xN);
    transpose(W1, N, tempNx3);               
    mult_3xN_x_Nx3(temp3xN, tempNx3, N, Vp);         
    mult_3xN_x_NxN(W2, VY, N, temp3xN);
    transpose(W2, N, tempNx3);               
    mult_3xN_x_Nx3(temp3xN, tempNx3, N, temp3x3);
    matrix_add(Vp, temp3x3, Vp);
    mult_3xN_x_NxN(W3, VZ, N, temp3xN);    
    transpose(W3, N, tempNx3);               
    mult_3xN_x_Nx3(temp3xN, tempNx3, N, temp3x3);
    matrix_add(Vp, temp3x3, Vp);
    mult_3xN_x_NxN(W10, Vx, N, temp3xN);  
    transpose(W10, N, tempNx3);               
    mult_3xN_x_Nx3(temp3xN, tempNx3, N, temp3x3);
    matrix_add(Vp, temp3x3, Vp);
    mult_3xN_x_NxN(W11, Vy, N, temp3xN);  
    transpose(W11, N, tempNx3);               
    mult_3xN_x_Nx3(temp3xN, tempNx3, N, temp3x3);
matrix_add(Vp, temp3x3, Vp);
mult_3xN_x_NxN(W12, Vz, N, temp3xN);
transpose(W12, N, tempNx3);
mult_3xN_x_Nx3(temp3xN, tempNx3, N, temp3x3);
matrix_add(Vp, temp3x3, Vp);
}

/* ********************************************** */
/* The function computes the covariance matrix of the 
rotation del vector for the given frame number and 
transformation chain */
/* ********************************************** */

void Vdel_matrix( Vx, Vy, Vz, W5, W7, W9, N, Vdel)
double Vx[MAX_FRAME_NUM][MAX_FRAME_NUM],
Vy[MAX_FRAME_NUM][MAX_FRAME_NUM],
Vz[MAX_FRAME_NUM][MAX_FRAME_NUM], W5[3][MAX_FRAME_NUM],
W7[3][MAX_FRAME_NUM], W9[3][MAX_FRAME_NUM], Vdel[3][3];
int N;
{
double temp3xN[3][MAX_FRAME_NUM], tempNx3[MAX_FRAME_NUM][3],
temp3x3[3][3];
void mult_3xN_x_NxN(), transpose(), mult_3xN_x_Nx3(),
matrix_add();
mult_3xN_x_NxN(W5, Vx, N, temp3xN);
transpose(W5, N, tempNx3);
mult_3xN_x_Nx3(temp3xN, tempNx3, N, Vdel);

mult_3xN_x_NxN(W7, Vy, N, temp3xN);
transpose(W7, N, tempNx3);
mult_3xN_x_Nx3(temp3xN, tempNx3, N, temp3x3);
matrix_add(Vdel, temp3x3, Vdel);

mult_3xN_x_NxN(W9, Vz, N, temp3xN);
transpose(W9, N, tempNx3);
mult_3xN_x_Nx3(temp3xN, tempNx3, N, temp3x3);
matrix_add(Vdel, temp3x3, Vdel);
}

/* ********************************************** */
/* The following is the function to compute the eigenvectors 
and eigenvalues of the 3x3 covariance matrix. The Jacobi 
algorithm is described in Hornbeck, Robert W., "Numerical 
Methods", p.240 */
/* ********************************************** */

#define EPSILON 1.0e-6
#define MAX_SIZE 4

void eigenvalues(matrix_in, n, eigenval, R)
double matrix_in[3][3], eigenval[MAX_SIZE],
    R[MAX_SIZE][MAX_SIZE];
int n;
{
    int i, j, l, m, p, q;
    double mu, muf, v0 = 0.0, a[MAX_SIZE], d[MAX_SIZE],
        H[MAX_SIZE][MAX_SIZE], e[MAX_SIZE], f, alpha, beta, c, s;
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            H[i+1][j+1] = matrix_in[i][j];
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            if (i != j)
                v0 += H[i][j] * H[i][j];
    mu = sqrt(v0)/n;
    muf = EPSILON * mu;
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
        {
            if (i == j)
                R[i][j] = 1.0;
            else
                R[i][j] = 0.0;
        }
    for (; mu > muf; mu /= n)
        for (l = 1; l != n; l++)
            for (m = l+1; m <= n; m++)
                if (fabs(H[l][m]) >= mu && H[l][l] != H[m][m])
                {
                    p = l;
                    q = m;
                    for (i = 1; i != n+1; i++)
                    {
                        a[i] = H[p][i];
                        d[i] = H[i][p];
                        e[i] = R[i][p];
                    }
                    f = H[p][p];
                    alpha = 0.5 * (H[p][p] - H[q][q]);
                    beta = sqrt(H[p][q]*H[p][q] +
                                alpha*alpha);
                    c = sqrt(0.5 + fabs(alpha)/(2*beta));
                    s = -alpha*H[p][q]/(2*beta*fabs(alpha)*c);
                    for (j = 1; j != n+1; j++)
                    {
                        if (j != p && j != q)
                        {
                            H[p][j] = c*H[p][j] -
                                s*H[q][j];
                            H[q][j] = s*a[j] + c*H[q][j];
                        }
                    }
                }
```c
for (i=1; i <= n; i++)
    eigenval[i] = H[i][i];

/* ========================================================================= */
/* The function prints out on the screen the standard deviations in the principal directions of the ellipsoid. Adjustments are made for uniform distributions */
/* ========================================================================= */

void print_std_dev(eigenval, dist_typeX, dist_typeY)
    double eigenval[4];
    int dist_typeX, dist_typeY;
{
    printf("Linear standard deviations along ellipsoid principal axes: \
    if (dist_typeX == UNI)
        printf( "\%6.4f\n", RATIO*3.0*sqrt(eigenval[1]) );
    else
        printf( "\%6.4f\n", sqrt(eigenval[1]) );
    if (dist_typeY == UNI)
        printf( "\%6.4f\n", RATIO*3.0*sqrt(eigenval[2]) );
    else
        printf( "\%6.4f\n", sqrt(eigenval[2]) );
    printf( "\%6.4f\n", sqrt(eigenval[3]) );
}
```

```c
/* ========================================================================= */
/* The function below computes the inverse of a 3x3 matrix (here Vp). The algorithm is the well-known Gauss - Jordan elimination. The flow chart can be found in Chapra, S. and Canale, R.P., "Numerical Methods for Engineers", */
/* ========================================================================= */

void inverse(mat_in, mat_out)
    double mat_in[3][3], mat_out[3][3];
```
{ int i, j, k;
    double a[3][6], temp;
    for (i = 0; i < 3; i++)
    {
        /* initialize the augmented matrix */
        for (j = 0; j < 3; j++)
            a[i][j] = mat_in[i][j];
        for (j = 3; j < 6; j++)
            if (i + 3 == j)
                a[i][j] = 1.0;
            else
                a[i][j] = 0.0;
    }
    for (k = 0; k < 3; k++)
    {
        /* perform the Gauss - Jordan elimination */
        temp = a[k][k];
        for (j = 0; j < 6; j++)
            a[k][j] = a[k][j] / temp;
        for (i = 0; i < 3; i++)
            if (i != k)
            {
                temp = a[i][k];
                for (j = 0; j < 6; j++)
                    a[i][j] = a[i][j] - temp * a[k][j];
            }
    }
    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 3; j++)
            mat_out[i][j] = a[i][j + 3];
    }
}

/* *************************************************************** */
/* The function computes the coefficients of the two */
/* perpendicular lines - XY base - with respect to the principal */
/* axes of the ellipsoid */
/* *************************************************************** */

void base_orientation(inverted_mat, a1, a2)
    double inverted_mat[3][3], *a1, *a2;
    { double theta; /* "theta" is the angle of rotation of XY */
        /* base coordinates and principal ellipsoid */
        /* axes in the XY plane with respect to each */
        /* other */
        theta = 0.5*atan( 2*inverted_mat[0][1]/(inverted_mat[0][0] -
            inverted_mat[1][1]) );
        *a1 = tan(theta);
        *a2 = tan(theta + PI/2);
    }
/* ***************************************************************************/
/* The function computes coefficients of the contours of the constant probability ellipses in the XY plane. The form of the ellipse eqn is a1*X*X + a2*Y*Y = 1*1 */
/* ***************************************************************************/

void ellipse_coefficients(inverted_mat, a1, a2)
double inverted_mat[3][3], *a1, *a2;
{
  double theta; /* "theta" is the angle of rotation of XY base coordinates and principal ellipsoid axes in the XY plane with respect to each other */

  theta = 0.5*atan( 2*inverted_mat[0][1]/(inverted_mat[0][0]-inverted_mat[0][0]) );
  *a1 = cos(theta)*cos(theta)*inverted_mat[0][0] + inverted_mat[1][1]*sin(theta)*sin(theta) -
       sin(2.0*theta)*inverted_mat[0][1];
  *a2 = cos(theta)*cos(theta)*inverted_mat[1][1] +
       inverted_mat[0][0]*sin(theta)*sin(theta) -
       sin(2.0*theta)*inverted_mat[0][1];
}

/* ***************************************************************************/
/* In case that the 2D ellipsoid case is detected, the ellipses are drawn from previously computed eigenvalues of Vp matrix. */
/* ***************************************************************************/

void ellipse_2D_case(eigenval, eigenvect, a1, a2, e1, e2)
double eigenval[4], eigenvect[4][4], *a1, *a2, *e1, *e2;
{
  double theta;
  if (eigenvect[1][1] != 0.0)
    theta = atan( eigenvect[2][1] / eigenvect[1][1] );
  else
  {
    /* This will hopefully never happen */
    printf("\nBUG! Division by 0 in the ellipse_2D_case()
function\n")
    function

    exit();
  }
  *a1 = tan(theta);
  *a2 = tan(theta + PI/2);
  *e1 = 1.0 / eigenval[1];
  *e2 = 1.0 / eigenval[2];
}
void convolve_uniform_and_normal(delta_X, delta_Y, frame_num, dist_typeX, dist_typeY)
double delta_X[MAX_FRAME_NUM][2], delta_Y[MAX_FRAME_NUM][2];
int *dist_typeX, *dist_typeY, frame_num;
{
    int i;
    double var_X_uni = 0.0, var_Y_uni = 0.0;
    double var_X_nrm = 0.0, var_Y_nrm = 0.0;
    double tempX[MAX_FRAME_NUM], something, tempY[MAX_FRAME_NUM],
    var_X_conv_uni = 0.0, tempXsum = 0.0, var_Y_conv_uni =
    0.0, tempYsum = 0.0;

    for (i = 0; i <= frame_num; i++)
    {
        var_X_nrm += (delta_X[i][NRM] / 3) * (delta_X[i][NRM] / 3);
        var_Y_nrm += (delta_Y[i][NRM] / 3) * (delta_Y[i][NRM] / 3);
        /* the arrays "tempX(Y)" hold the nonzero values of the
          uniform distributions array */
        tempX[i] = delta_X[i][UNI];
        tempY[i] = delta_Y[i][UNI];
    }

    /* check if we are in the region of preferred uniform
     distribution or preferred normal distribution and compute the
     new variance of the convolved distribution*/
    for(i = 0; i <= frame_num; i++)
    {
        if( (var_X_nrm > 9.0e-12 ) && (something =
            RATIO*tempX[i]/sqrt(var_X_nrm) ) < 1.92 )
        {
            *dist_typeX = NRM;
            var_X_nrm = var_X_nrm + (RATIO*tempX[i]) *
            (RATIO*tempX[i]);
        }
        else if (var_X_nrm > 9.0e-12)
        {
            *dist_typeX = UNI;
            var_X_conv_uni = (RATIO*tempX[i])*(RATIO*tempX[i])
            + var_X_nrm;
            var_X_nrm = 0.0;
            tempXsum = sqrt(var_X_conv_uni)/RATIO;
        }
        else if ( tempXsum/tempX[i] < 0.54 ||
            tempXsum/tempX[i] > 1.86)

{
  *dist_typeX = UNI;
  var_X_conv_uni = (RATIO*tempXsum) *
      (RATIO*tempXsum) + (RATIO*tempX[i]) * (RATIO*tempX[i]);
  tempXsum = sqrt(var_X_conv_uni)/RATIO;
}
else
{
  *dist_typeX = NRM;
  var_X_nrm = (RATIO*tempXsum)*
      (RATIO*tempXsum) + (RATIO*tempX[i]) * (RATIO*tempX[i]);
}

if( (var_Y_nrm > 9.0e-12) && ( something = RATIO*tempY[i]/sqrt(var_Y_nrm) ) < 1.92 )
{
  *dist_typeY = NRM;
  var_Y_nrm = var_Y_nrm + (RATIO*tempY[i]) * (RATIO*tempY[i]);
}
else if (var_Y_nrm > 9.0e-12)
{
  *dist_typeY = UNI;
  var_Y_conv_uni = (RATIO*tempY[i])*(RATIO*tempY[i]) + var_Y_nrm;
  var_Y_nrm = 0.0;
  tempYsum = sqrt(var_Y_conv_uni)/RATIO;
}
else if ( tempYsum/tempY[i] < 0.54 || tempYsum/tempY[i] > 1.86)
{
  *dist_typeY = UNI;
  var_Y_conv_uni = (RATIO*tempYsum)*
      (RATIO*tempYsum) + (RATIO*tempY[i]) *
      (RATIO*tempY[i]);
  tempYsum = sqrt(var_Y_conv_uni)/RATIO;
}
else
{
  *dist_typeY = NRM;
  var_Y_nrm = (RATIO*tempYsum)*
      (RATIO*tempYsum) + (RATIO*tempY[i]) *
      (RATIO*tempY[i]);
}
}

/* ************************************************** */
/* The following fcn uses the Sun C 2 application to display ellipses on the screen. Refer to user manual for information on the functions used here. The displayed ellipses are */

163
cross-sections of the ellipsoid with a plane passing through
the ellipsoid's center and parallel to the XY base reference
frame. */
/
#define GREEN 3
#define RED 1
#define WHITE 7
#define BLACK 0

void draw_ellipses_and_base( coeff1, coeff2, tan1, tan2,
   dist_typeX, dist_typeY, sc_fact )
double coeff1, coeff2, tan1, tan2, *sc_fact;
int dist_typeX, dist_typeY;
{
   Cvwsurf device;
   Cint name,
   trigger;
   Ccoor dv1, dv2, lower, upper, center, list[2], pl,
   p2;
   Ccoorlist points;
   static Ccoor ipt = { 16384, 16384 };
   Cawresult valid;
   static Cintrep ivalue = &ipt;
   double maj_axis, min_axis, maxX, maxY;
   int a, b, l, i;
   char valueX[10], valueY[10];
   char *toolposition = "0, 0, 680, 680, 0, 0, 680, 680, 0";

NORMAL_VWSURF(device, PIXWINDD);
device.flags = VWSURF_NEWFLG;
device.ptr = &toolposition;

fprintf(stderr, "\nTo continue, press the right mouse button
anywhere in the graphics area...\n");

open_cgi();
open_vws(&name, &device);
dv1.x = dv1.y = 0;
dv2.x = dv2.y = 1000;
vdc_extent(&dv1, &dv2);
center.x = center.y = 500;
initialize_lid( IC_LOCATOR, LID_LOC, &ivalue);
associate( MOUSE_BUTTON_3, IC_LOCATOR, LID_LOC);
maj_axis = 3.0 / sqrt(coeff1);
min_axis = 3.0 / sqrt(coeff2);
if ( maj_axis > min_axis )
   *sc_fact = 450.0 / maj_axis;
else
   *sc_fact = 450.0 / min_axis;
line_type(DOTTED);
line_color(WHITE);
list[0].y = list[1].y = 500;
list[0].x = 10;
list[1].x = 990;
points.ptlist = list;
points.n = 2;
polyline( &points);
list[0].x = list[1].x = 500;
list[0].y = 10;
list[1].y = 990;
points.ptlist = list;
points.n = 2;
polyline( &points);
perimeter_color(WHITE);
for ( l = 1;  l <= 3;  l++)
{
    maxX = *sc_fact * 3 / sqrt(coef1);
    maxY = *sc_fact * 3 / sqrt(coef2);
    if(dist_typeX == UNI)
        a = *sc_fact * 3 / sqrt(coef1);
    else
        a = *sc_fact * 1 / sqrt(coef1);
    if(dist_typeY == UNI)
        b = *sc_fact * 3 / sqrt(coef2);
    else
        b = *sc_fact * 1 / sqrt(coef2);
    ellipse( &center, a, b);
    list[0].x = 10;
    list[1].x = 990;
    list[0].y = list[1].y = 500 - (int)( 1/3.0 * (*sc_fact)*min_axis );
    points.ptlist = list;
    points.n = 2;
polyline( &points);
    list[0].y = list[1].y = 500 + (int)( 1/3.0 * (*sc_fact)*min_axis );
    points.ptlist = list;
    points.n = 2;
polyline( &points);
    list[0].y = list[1].x = list[0].x = 500 - (int)( 1/3.0 * (*sc_fact)*maj_axis );
    points.ptlist = list;
    points.n = 2;
polyline( &points);
    list[0].x = list[1].x = 500 + (int)( 1/3.0 * (*sc_fact)*maj_axis );
    points.ptlist = list;
    points.n = 2;
polyline( &points);
line_type(SOLID);
line_color(WHITE);
list[0].x = 10;               /* the first line (X-axis) */
list[0].y = (int)( tan1*(-490.) + 500. );
list[1].x = 990;
list[1].y = (int)( 980.*tan1+list[0].y );
if ( list[0].y < 10 )
{
    list[0].y = 10;
    list[0].x = (int)( (-490)/tan1+500. );
    list[1].x = (int)( list[0].x +980./tan1);
    list[1].y = 990;
}
points.ptlist = list;
points.n = 2;
polyline( &points);
pl.x = 975;
pl.y = (int)( 500 + (pl.x - 500)*tan1 );
text_color(WHITE);
text(&pl, "X");

list[0].x = 10;               /* second line (Y-axis) */
list[0].y = (int)( tan2*(-490.) + 500. );
list[1].x = 990;
list[1].y = (int)( 980.*tan2+list[0].y );
if ( list[0].y < 10 )
{
    list[0].y = 10;
    list[0].x = (int)( (-490)/tan2+500. );
    list[1].x = (int)( list[0].x +980./tan2);
    list[1].y = 990;
}
points.ptlist = list;
points.n = 2;
polyline( &points);
p2.y = 975;
p2.x = (int)( (p2.y - 500)/tan2 + 500 );
text(&p2, "Y");

for ( i = 0; i <= 6; ++i )
{
    sprintf(valueX, "%4.2f", (-3 + i)/3.0 * maj_axis );
    sprintf(valueY, "%4.2f", (-3 + i)/3.0 * min_axis );
    pl.x = 500 + (int)((-3 + i)/3.0 * (*sc_fact)*maj_axis );
    pl.y = 500;
p2.x = 500;
p2.y = 500 + (int)((-3 + i)/3.0 * (*sc_fact)*min_axis );
text(&pl, valueX );
text(&p2, valueY );
}

/* Continue untiill interrupted by the pressing of the third mouse button by the user. */
do
    request_input(IC_LOCATOR, LID_LOC, TEN_MIN,
        &valid, &iValue, &trigger);
} while (valid == VALID_DATA && trigger !=
        MOUSE_BUTTON_3);

dissociate(MOUSE_BUTTON_3, IC_LOCATOR, LID_LOC);
release_input_device(IC_LOCATOR, LID_LOC);

/* *************************************************************/
/* The Monte Carlo function begins here */
/* *************************************************************/

#define MAX_NUM 100
#define WHITE 7
#define BLUE 5
#define CYAN 4

void monte_carlo(X, delta_X, Y, delta_Y, Z, delta_Z, x,
        delta_x, y, delta_y, z, delta_z, num_of_frames,
        frame_num, sc_fact)
    double X[MAX_FRAME_NUM], delta_X[MAX_FRAME_NUM][2],
            Y[MAX_FRAME_NUM], delta_Y[MAX_FRAME_NUM][2],
            Z[MAX_FRAME_NUM], delta_Z[MAX_FRAME_NUM],
            x[MAX_FRAME_NUM], delta_x[MAX_FRAME_NUM],
            y[MAX_FRAME_NUM], delta_y[MAX_FRAME_NUM],
            z[MAX_FRAME_NUM], delta_z[MAX_FRAME_NUM];
    double sc_fact;
    int num_of_frames, frame_num;
{
    Ccoorlist points;
    Ccoor list[1];
    Cint trigger;
    static Ccoor ipt = { 16384, 16384 };  
    Cawresult valid;
    static Cinrep iValue = &ipt;
    double normal_num();
    int count, i, j, point_num, type;
    double drand48(), sum, rand_num[MAX_NUM], std_dev,
            mean;
    double des_std_dev;
    double transformed_matrix[4][4],
            propagated_frame[4][4][MAX_FRAME_NUM];
    void default_data(), propagate_transform(),
            optimal_position();
    double find_probability();
    double X_optim_posit, Y_optim_posit, Z_optim_posit;

    printf("\nThe max number of random points to be generated is
        %d.\n", MAX_TRIALS NUM);
    printf("To Interrupt, press the right mouse button anywhere

in the graphics area...\ninitialize_lid( IC_LOCATOR, LID_LOC, &ivalue);
associate(MOUSE_BUTTON_3, IC_LOCATOR, LID_LOC);
opimal_position(X, Y, Z, x, y, z, frame_num,
    propagated_frame, &X_optim_posit,
    &Y_optim_posit, &Z_optim_posit);

trigger = 0;
for ( i = 0; i <= MAX_TRIALS_NUM && trigger !=
    MOUSE_BUTTON_3; i++)
{
    propagate_transform(X, delta_X, Y, delta_Y, Z, delta_Z,
        x, delta_x, y, delta_y, z, delta_z,
        transformed_matrix, frame_num, propagated_frame);
    list[0].x = 500 + (int)(sc_fact*
        (X_optim_posit - propagated_frame[0][3][frame_num])
    list[0].y = 500 + (int)(sc_fact*
        (Y_optim_posit - propagated_frame[1][3][frame_num])
    points.ptlist = list;
    points.n = 1;
    marker_size_specification_mode(Absolute);
    marker_type( CIRCLE);
    marker_size(2.0);
    marker_color(WHITE);
    polymarker( &points);
    request_input(IC_LOCATOR, LID_LOC, 1, &valid, &ivalue, &trigger);
}

printf("\nSimulation aborted.\n"); 
dissociate(MOUSE_BUTTON_3, IC_LOCATOR, LID_LOC); 
release_input_device(IC_LOCATOR, LID_LOC);

/
/* The function returns random numbers uniformly distributed. */
/* The range of uniform distribution is specified by the 'range' variable. */
/* */

double uniform_num(range)
{ 
    double drand48();

    return ( 2.0 * range * (drand48() - 0.5) );
}
#define UNIFORM_STD_DEV 2.8867513459481287
    /* this is the value of standard deviation for the
       uniform distribution between -5.0 and +5.0 */

double normal_num(normal_std_dev)
    double normal_std_dev;
    {
        double drand48(), sum, a, b, c;
        int sample_size, i;
        sum = 0.0;
        if (normal_std_dev < 0.000001)
            normal_std_dev = 0.0;
        if (normal_std_dev != 0.0)
            {
                if (normal_std_dev >= 1.0)
                    {
                        a = 10.0;
                        b = 100.0;
                        c = 50.0;
                    }
                if (normal_std_dev >= 0.1)
                    {
                        a = 1.0;
                        b = 10.0;
                        c = 5.0;
                    }
                else if (normal_std_dev >= 0.01)
                    {
                        a = 0.1;
                        b = 1.0;
                        c = 0.5;
                    }
                else if (normal_std_dev >= 0.001)
                    {
                        a = 0.01;
                        b = 0.1;
                        c = 0.05;
                    }
                else if (normal_std_dev >= 0.0001)
                    {
                        a = 0.001;
                        b = 0.01;
                        c = 0.005;
                    }
                else if (normal_std_dev >= 0.00001)
                    {
                        a = 0.0001;
                        b = 0.001;
                        c = 0.0005;
                    }
                else if (normal_std_dev >= 0.000001)
                    {
                        a = 0.00001;
                    }
                a = 0.00001;
b = 0.0001;
c = 0.00005;
}
else /* This will never happen. (I hope.) */
{
    printf("\n\nBug in the normal_num() function. The required std deviation too small to generate a number.\n\nSorry!\n\n");
    exit();
}
sample_size = (a * UNIFORM_STD_DEV / normal_std_dev) *
    (a * UNIFORM_STD_DEV / normal_std_dev);
/* sample_size is calculated to be used in the central limit theorem to approximate the
  normal distribution */
for (i = 0; i <= sample_size; i++)
    sum += b * drand48() - c;
return (sum / sample_size);
/* the average of appropriate sample size gives the
  normal distribution (with standard deviation determined by the sample size) */
}
else
    return (0.0);


توقف ملخص المادة

/* Add the perturbation resulting from errors to nominal values. Find the transformation matrix for a particular frame number 'i'. */
/* ****************************************************************************** */
#define PI 3.141592653589793238

void transform(X, delta_X, Y, delta_Y, Z, delta_Z, x, delta_x, y, delta_y, z, delta_z, output_matrix, i)
    double X[MAX_FRAME_NUM], delta_X[MAX_FRAME_NUM][2],
            Y[MAX_FRAME_NUM], delta_Y[MAX_FRAME_NUM][2],
            Z[MAX_FRAME_NUM], delta_Z[MAX_FRAME_NUM],
            x[MAX_FRAME_NUM], delta_x[MAX_FRAME_NUM],
            y[MAX_FRAME_NUM], delta_y[MAX_FRAME_NUM],
            z[MAX_FRAME_NUM], delta_z[MAX_FRAME_NUM];
    double output_matrix[4][4][MAX_FRAME_NUM];
    int i; /* frame number */
{
    double tempX[MAX_FRAME_NUM], tempY[MAX_FRAME_NUM],
            tempZ[MAX_FRAME_NUM], tempx[MAX_FRAME_NUM],
            tempy[MAX_FRAME_NUM], tempz[MAX_FRAME_NUM];
    if (delta_X[i][NRM] != 0.0)
        tempX[i] = X[i] + normal_num(deltax[i][NRM] / 3);
    else if (delta_X[i][UNI] != 0.0)
tempX[i] = X[i] + uniform_num(delta_X[i][UNI]);
else
    tempX[i] = X[i];
if (delta_Y[i][NRM] != 0.0)
    tempY[i] = Y[i] + normal_num(delta_Y[i][NRM]/3);
else if (delta_Y[i][UNI] != 0.0)
    tempY[i] = Y[i] + uniform_num(delta_Y[i][UNI]);
else
    tempY[i] = Y[i];
tempZ[i] = Z[i] + normal_num(delta_Z[i]/3);
tempX[i] = X[i] + normal_num(delta_x[i]/3);
tempY[i] = Y[i] + normal_num(delta_y[i]/3);
tempZ[i] = Z[i] + normal_num(delta_z[i]/3);
output_matrix[0][0][i] = cos(tempy[i])*cos(tempz[i]);
output_matrix[0][1][i] = -cos(tempy[i])*sin(tempz[i]);
output_matrix[0][2][i] = sin(tempy[i]);
output_matrix[0][3][i] = tempX[i];
output_matrix[1][0][i] = cos(tempz[i])*sin(tempx[i])*sin(tempy[i]) + cos(tempx[i])*sin(tempy[i])*sin(tempz[i]);
output_matrix[1][1][i] = cos(tempx[i])*cos(tempz[i]) - sin(tempx[i])*sin(tempz[i]);
output_matrix[1][2][i] = -cos(tempy[i])*sin(tempx[i]);
output_matrix[1][3][i] = tempY[i];
output_matrix[2][0][i] = -cos(tempy[i])*cos(tempz[i])*sin(tempy[i]) + sin(tempy[i])*sin(tempz[i])*sin(tempy[i]);
output_matrix[2][1][i] = cos(tempz[i])*sin(tempy[i]) + cos(tempx[i])*sin(tempy[i])*sin(tempz[i]);
output_matrix[2][2][i] = cos(tempx[i])*cos(tempz[i]);
output_matrix[2][3][i] = tempZ[i];
output_matrix[3][0][i] = 0.0;
output_matrix[3][1][i] = 0.0;
output_matrix[3][2][i] = 0.0;
output_matrix[3][3][i] = 1.0;

/* **************************************************************************/ /* Perform the multiplication of matrices to propagate all the perturbations through the chain of transformations. */
/* **************************************************************************/
void mult_perturb (X, delta_X, Y, delta_Y, Z, delta_Z, x, delta_x, y, delta_y, z, delta_z, transformed_matrix, frame_num, propagated_frame )
double X[MAX_FRAME_NUM], delta_X[MAX_FRAME_NUM][2],
      Y[MAX_FRAME_NUM], delta_Y[MAX_FRAME_NUM][2],
      Z[MAX_FRAME_NUM], delta_Z[MAX_FRAME_NUM],
      x[MAX_FRAME_NUM], delta_x[MAX_FRAME_NUM],
      y[MAX_FRAME_NUM], delta_y[MAX_FRAME_NUM],
      z[MAX_FRAME_NUM], delta_z[MAX_FRAME_NUM];
double transformed_matrix[4][4][MAX_FRAME_NUM],
      propagated_frame[4][4][MAX_FRAME_NUM];
int  frame_num;
{  
    int count, i, j;
    double temp[4][4], next_temp[4][4], out_temp[4][4];
    void multiply();
    for (i = 0; i < 4; i++)
    {
        for (j = 0; j < 4; j++)
        {
            if (i == j)
            {
                temp[i][j] = 1.0;
            }
            else
            {
                temp[i][j] = 0.0;
            }
        }
    }
    for (count = 0; count <= frame_num; count++)
    {
        transform(X, delta_X, Y, delta_Y, Z, delta_Z, x, 
                  delta_x, y, delta_y, z, delta_z, 
                  transformed_matrix, count);
        for (i = 0; i < 4; i++)
        {
            for (j = 0; j < 4; j++)
            {
                next_temp[i][j] = transformed_matrix[i][j][count];
            }
        }
        multiply(temp, next_temp, out_temp);
        for (i = 0; i < 4; i++)
        {
            for (j = 0; j < 4; j++)
            {
                propagated_frame[i][j][count] = out_temp[i][j];
                temp[i][j] = out_temp[i][j];
            }
        }
    }
}

/* *****************************************/
/* Here the nominal position of the propagated frame is found */
/* (if no errors were present). */
/* *****************************************/

void optimal_position(X, Y, Z, x, y, z, frame_num,
        propagated_frame, X_optim_posit, Y_optim_posit, 
        Z_optim_posit)
    double X[MAX_FRAME_NUM], Y[MAX_FRAME_NUM], Z[MAX_FRAME_NUM],
        x[MAX_FRAME_NUM], y[MAX_FRAME_NUM], z[MAX_FRAME_NUM],
        propagated_frame[4][4][MAX_FRAME_NUM], *X_optim_posit,
*Y_optim_posit, *Z_optim_posit;
int frame_num;

{
int i, type;
double delta_X0[MAX_FRAME_NUM][2],
delta_Y0[MAX_FRAME_NUM][2], delta_Z0[MAX_FRAME_NUM],
delta_x0[MAX_FRAME_NUM], delta_y0[MAX_FRAME_NUM],
delta_z0[MAX_FRAME_NUM],
transformed_matrix[4][4][MAX_FRAME_NUM]

printf("Stand by...\n");
for (i = 0; i <= frame_num; i++)
{
for(type = UNI; type <= NRM; type++)
{
delta_X0[i][type] = 0.0;
delta_Y0[i][type] = 0.0;
}
delta_Z0[i] = 0.0;
delta_x0[i] = 0.0;
delta_y0[i] = 0.0;
delta_z0[i] = 0.0;
}
propagate_transform(X, delta_X0, Y, delta_Y0, Z, delta_Z0, x,
delta_x0, y, delta_y0, z, delta_z0, transformed_matrix,
frame_num, propagated_frame);
*X_optim_posit = propagated_frame[0][3][frame_num];
*Y_optim_posit = propagated_frame[1][3][frame_num];
*Z_optim_posit = propagated_frame[2][3][frame_num];
}

/* ****************************************************** */
/* MAIN PROGRAM */
/* ****************************************************** */

main()
{
Cint name;
int frame_num, num_of_frames, dist_typeX, dist_typeY;
int choice = 10, get_another_frame();
double X[MAX_FRAME_NUM], delta_X[MAX_FRAME_NUM][2],
Y[MAX_FRAME_NUM], delta_Y[MAX_FRAME_NUM][2],
Z[MAX_FRAME_NUM], delta_Z[MAX_FRAME_NUM],
x[MAX_FRAME_NUM], delta_x[MAX_FRAME_NUM],
y[MAX_FRAME_NUM], delta_y[MAX_FRAME_NUM],
z[MAX_FRAME_NUM], delta_z[MAX_FRAME_NUM],
m1[3][MAX_FRAME_NUM], m2[3][MAX_FRAME_NUM],
m3[3][MAX_FRAME_NUM], m4[3][MAX_FRAME_NUM],
m5[3][MAX_FRAME_NUM], m6[3][MAX_FRAME_NUM],
m7[3][MAX_FRAME_NUM], m8[3][MAX_FRAME_NUM],
m9[3][MAX_FRAME_NUM], R[3][3][MAX_FRAME_NUM],
p[3][MAX_FRAME_NUM], ptN[3], W1[3][MAX_FRAME_NUM],
W2[3][MAX_FRAME_NUM], W3[3][MAX_FRAME_NUM],
W4[3][MAX_FRAME_NUM], W5[3][MAX_FRAME_NUM],
W6[3][MAX_FRAME_NUM], W7[3][MAX_FRAME_NUM],
W8[3][MAX_FRAME_NUM], W9[3][MAX_FRAME_NUM],
W10[3][MAX_FRAME_NUM], W11[3][MAX_FRAME_NUM],
W12[3][MAX_FRAME_NUM],
VX[MAX_FRAME_NUM][MAX_FRAME_NUM],
VY[MAX_FRAME_NUM][MAX_FRAME_NUM],
VZ[MAX_FRAME_NUM][MAX_FRAME_NUM],
Vx[MAX_FRAME_NUM], Vy[MAX_FRAME_NUM],
Vz[MAX_FRAME_NUM],
ingverted_mat[3][3], Vp[3][3], eigenval[4],
eigenvect[4][4], sc_fact;
double propagated_frame[4][4][MAX_FRAME_NUM], X_optim_posit,
Y_optim_posit, Z_optim_posit, Vdel[3][3];
void default_data(), ptN_vector(), all_Rs_ps_and_ms(),
W_matrices(), V_matrices(), mult_3xN_x_NxN(),
mult_3xN_x_Nx3(), matrix_add(), Vp_matrix(),
eigenvalues(), inverse(), base_orientation(),
ellipse_coefficients(), draw_ellipses_and_base(),
monte_carlo(), get_user_data(),
convolve_uniform_and_normal(), menu(), Vdel_matrix(),
get_data_from_file(), ellipse_2D_case(),
print_std_dev();

double a1, a2, e1, e2;
int j, i, demo, yes_or_no();

default_data(X, delta_X, Y, delta_Y, Z, delta_z, x, delta_x,
y, delta_y, z, delta_z, &num_of_frames, &frame_num,
&demo);
printf("Would you like to display the demonstration run?\n");
demo = yes_or_no();
get_user_data(X, delta_X, Y, delta_Y, Z, delta_z, x, delta_x,
y, delta_y, z, delta_z, &num_of_frames, &frame_num,
&demo);
do{
  if (choice != MONTE_CARLO )
  {
    if (choice == CHANGE_DATA)
    {
      close_vws(name);
      close_cgi();
      demo = YES;
      get_user_data(X, delta_X, Y, delta_Y, Z,
                   delta_z, x, delta_x, y, delta_y, z,
                   delta_z, &num_of_frames, &frame_num,
                   &demo);
    }
  }
  if (choice == ANOTHER_FRAME)
close_vws(name);
close_cgi();
frame_num = get_another_frame(num_of_frames);
}

if (choice == FILE_DATA )
{
close_vws(name);
close_cgi();
get_data_from_file(X, delta_X, Y, delta_Y, Z,
    delta_Z, x, delta_x, y, delta_y, z,
    delta_z, &num_of_frames, &frame_num);
}

optimal_position(X, Y, Z, x, y, z, frame_num,
    propagated_frame, &X_optim_posit,
    &Y_optim_posit, &Z_optim_posit);
printf("\nThe nominal position of the propagated
frame #\%d is:\n", frame_num + 1);
printf("X = %8.4f, Y = %8.4f, Z = %8.4f \n\n",
    X_optim_posit, Y_optim_posit, Z_optim_posit);
V_matrices(delta_X, delta_Y, delta_Z, delta_x,
    delta_y, delta_z, frame_num, VX, VY, VZ, Vx,
    Vy, Vz);
ptN_vector(X, Y, Z, x, y, z, ptN, frame_num);
all_Rs_ps_and_ms(X, Y, Z, x, y, z, frame_num, R, P,
    m1, m2, m3, m4, m5, m6, m7, m8, m9);
W_matrices(R, P, ptN, m1, m2, m3, m4, m5, m6, m7,
    m8, m9, frame_num, W1, W2, W3, W4, W5, W6, W7,
    W8, W9, W10, W11, W12);
Vp_matrix( VX, VY, VZ, Vx, Vy, Vz, W1, W2, W3, W4,
    W5, W6, W7, W8, W9, W10, W11, W12, frame_num, Vp);
Vdel_matrix( VX, VY, VZ, W5, W7, W9, frame_num,
    Vdel);
eigenvalues(Vp, 3, eigenval, eigenvect);
convolve_uniform_and_normal(delta_X, delta_Y,
    frame_num, &dist_typeX, &dist_typeY);
print_std_dev(eigenval, dist_typeX, dist_typeY);
if ( eigenval[3] < 9.0e-12 )
{
    ellipse_2D_case(eigenval, eigenvect, &a1, &a2,
    &e1, &e2);
}
else        /* the ellipsoid is in 3-D space */
{
    inverse(Vp, inverted_mat);
    base_orientation(inverted_mat, &a1, &a2);
    ellipse_coefficients(inverted_mat, &e1, &e2);
}
printf("\nAngular standard deviations around
principal axes (deg):\n");
eigenvalues(Vdel, 3, eigenval, eigenvect);
printf("%6.4f\n", sqrt(eigenval[1])*180.0/PI );
printf("%6.4f\n", sqrt(eigenval[2])*180.0/PI );
printf("%6.4f\n", sqrt(eigenval[3])*180.0/PI );
draw_ellipses_and_base( e1, e2, a1, a2, dist_typeX,
    dist_typeY, &sc_fact );
}
else
    monte_carlo(X, delta_X, Y, delta_Y, Z, delta_Z, x,
        delta_x, y, delta_y, z, delta_z,
        num_of_frames, frame_num, sc_fact);
    menu( &choice );
} while(choice != QUIT);

close_vws(name);
close_cgi();
}