JSC-61777-models/utils/orientation

JSC Engineering Orbital Dynamics Orientation Model

Simulation and Graphics Branch (ER7)
Software, Robotics, and Simulation Division
Engineering Directorate

Package Release JEOD v5.0

Document Revision 2.0
July 2022

National Aeronautics and Space Administration
Lyndon B. Johnson Space Center
Houston, Texas

JSC-61777-models/utils/orientation

JSC Engineering Orbital Dynamics Orientation Model

Document Revision 2.0
July 2022

David Hammen

Simulation and Graphics Branch (ER7)
Software, Robotics, and Simulation Division
Engineering Directorate

National Aeronautics and Space Administration
Lyndon B. Johnson Space Center
Houston, Texas

Executive Summary

The Orientation Model forms a component of the utilities suite of models within JEOD v5.0. It is located at models/utils/orientation.

This document describes the implementation of the Orientation Model including the model requirements, specifications, mathematical theory, and architecture. A user guide is also provided to assist with implementing the model in Trick simulations.

Rotations in JEOD

There are many ways to represent the orientation of one three dimensional object with respect to another. The representation schemes supported in JEOD are

- Special orthogonal matrices, or proper rotation matrices. A proper rotation matrix is an orthogonal matrix whose determinant is one. The Matrix3x3 class in the *Mathematical Functions Model* [8] provides methods for manipulating 3x3 matrices and for using them to transform 3-vectors. Note that JEOD uses transformation matrices rather than rotation matrices.
- Unit quaternions. The unit quaternions are the subject of the *Quaternion Model* [1]. The Quaternion class defined in that model provides methods that relate to using unit quaternions to represent rotations and transformations. Note that JEOD uses left transformations unit quaternions exclusively.
- Eigen axis, or single axis, rotations. Any sequence of rotations in three dimensional space is equivalent to a pure rotation about a single axis. JEOD represents eigen rotations in terms of a rotation angle (a scalar) and a unit vector that defines the direction of the rotation via the right-hand rule.
- Euler rotations. An Euler rotation sequence is a sequence of three rotations about a set of principal axes. There are twelve possible conventions for these rotations. JEOD implements all twelve conventions.

Figure 1 portrays these four representation schemes and the direct conversions between them provided by JEOD v5.0. The dotted lines represent conversions implemented by the Quaternion Model while the solid lines represent conversions implemented in this model.

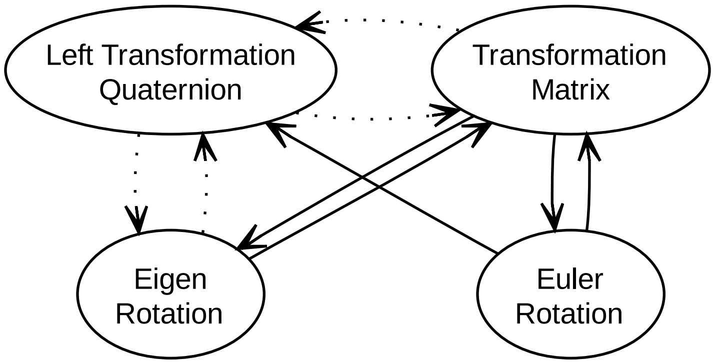

Figure 1: Orientations in JEOD

Model Architecture

This model exists for two reasons. One reason is to provide those conversions represented as solid lines in figure 1. The other reason is to provide an input mechanism by which a user can specify a rotation or transformation using any of the representation schemes described above. The Orientation class provides both of these capabilities.

The class provides a static method for each of the conversions represented as a solid line in figure 1. Note that the graph in figure 1 is not fully connected. While it is possible to convert between any two representation schemes, some conversions require the use of an intermediate representation scheme. For example, converting a left transformation quaternion to an Euler rotation sequence requires going through a transformation matrix as an intermediate step.

The Orientation class is also instantiable. An Orientation object contains data members that correspond to each of the four representation schemes supported by JEOD and contains auxiliary data members that indicate which representation scheme was input by the user. Other JEOD models can query the Orientation object to express the user-input data in any of the four supported representations.

Interactions With Other Models

Four of the conversions in figure 1, the conversions between quaternions and matrices and the conversions between quaternions and eigen rotations, are represented as dotted lines. The Quaternion class in the Quaternion Model provides the methods that implement these conversions.

Contents

Chapter 1

Introduction

1.1 Model Description

This documentation describes the design and testing of the routines in the JSC Engineering Orbital Dynamics (JEOD) Orientation Model. These routines are derived from well-known conversions between the various attitude representations (i.e., quaterion to matrix, etc.).

Included in this documentation are verification and validation cases that describe tests done on the algorithms to verify that they are working correctly and computing the correct values for given input data. There is also a User Guide which describes how to incorporate the above mentioned routines as part of a Trick simulation.

The parent document to this model document is the JEOD Top Level Document [3].

1.2 Documentation History

Author	**Date**	**Revision**	**Description**
Blair Thompson	November, 2009	1.0	Initial Version
David Hammen	October, 2010	2.0	JEOD 2.1

1.3 Document Organization

This document is formatted in accordance with the NASA Software Engineering Requirements Standard [4] and is organized into the following chapters:

Chapter 1: Introduction - This introduction contains three sections: description of model, document history, and organization. The first section provides the introduction to the Orientation Model and its reason for existence. It contains a brief description of the interconnections with other models, and references to any supporting documents. It also lists the document that is parent to this one. The second section displays the history of this document which includes author, date, and reason for each revision. The final section contains a description of the how the document is organized.

Chapter 2: Product Requirements - Describes requirements for the Orientation Model.

Chapter 3: Product Specification - Describes the underlying theory, architecture, and design of the Orientation Model in detail. It is organized in three sections: Conceptual Design, Mathematical Formulations, and Detailed Design.

Chapter 4: User Guide - Describes how to use the Orientation Model in a Trick simulation. It is broken into three sections to represent the JEOD defined user types: Analysts or users of simulations (Analysis), Integrators or developers of simulations (Integration), and Model Extenders (Extension).

Chapter 5: Verification and Validation - Contains Orientation Model verification and validation procedures and results.

Chapter 2

Product Requirements

Requirement Orientation_1: Project Requirements

Requirement:
This model shall meet the JEOD project requirements specified in the JEOD top-level document.

Rationale:
This is a project-wide requirement.

Verification:
Inspection

Requirement Orientation_2: Supported Representations

Requirement:
The Orientation Model shall provide the ability to represent an orientation in any of the following forms:

2.1 3x3 transformation matrix.

2.2 Left unit transformation quaternion.

2.3 Eigen rotation.

2.4 Euler angles.

Rationale:
There are many different but equivalent ways of representing the orientation of an object. Some are more intuitive than others while some are more amenable to analysis. An external source may specify orientation in any one of these forms (and others as well). For these reasons, JEOD must be able to model the orientation of an object in a number of representation schemes.

Verification:

Inspection, Test

Requirement Orientation_3: Data Access

Requirement:

3.1 Member Access. The Orientation Model shall provide public access to its representations of an orientation.

3.2 Data Consistency. The Orientation Model shall provide mechanisms to ensure that the member data for some representation of an orientation is consistent with input data.

3.3 Programmatic Assignment. The Orientation Model shall provide functional mechanisms to initialize or update an orientation.

3.4 Programmatic Access. The Orientation Model shall provide functional mechanisms to access representations of an orientation.

Rationale:

Public access is provided for backward compatibility. The model would fail to satisfy the requirement to represent multiple representation schemes without the consistency requirement. The latter two sub-requirements support a more modern, object-oriented view.

Verification:

Inspection, Test

Requirement Orientation_4: Euler Angles

Requirement:

With regard to Euler angles,

4.1 Supported Sequences. The Orientation Model shall support all twelve of the Euler rotation sequences.

4.2 Sequence Identifiers. The numerical values of the identifiers used in the Orientation Model to identify the six aerodynamics angles shall be equal to the corresponding identifiers used in Trick.

Rationale:

JEOD 2.0 used the Trick math library Euler angle identifiers and only supported the six aerodynamics Euler angles supported by Trick. The project requirement to make JEOD 2.1 Trick-independent means that the identifiers must now be defined by JEOD. The requirement to use the same numerical values as in Trick makes these new identifiers backwards compatible with JEOD 2.0. The requirement to be Trick-independent also provided the opportunity to support all twelve Euler sequences.

Verification:
Inspection, Test

Requirement Orientation_5: Conversions between Representation Schemes

Requirement:
The Orientation Model shall provide the following conversion schemes:

5.1 Eigen rotations to transformation matrices.

5.2 Transformation matrices to eigen rotations.

5.3 Euler angles to transformation matrices.

5.4 Transformation matrices to Euler angles.

5.5 Euler angles to quaternions.

Rationale:
JEOD 2.0 used Trick math library functions to perform the first four of the specified conversions. The project requirement to make JEOD 2.1 Trick-independent means that these conversion methods must now be implemented in JEOD. The final conversion makes it possible to directly compute a quaternionic representation from all supported representations.

Verification:
Inspection, Test

Chapter 3

Product Specification

There are many ways to represent the orientation of one three dimensional object with respect to another. The primary purpose of this model is to convert between the representation schemes used elsewhere and the schemes used internally by JEOD. Figure 3.1 portrays the four representation schemes supported by JEOD. Each of the connections between the representation schemes in the figure designates a specific method that converts from one scheme to another. The dotted lines are the responsibility of the Quaternion Model. The solid lines are the responsibility of this model.

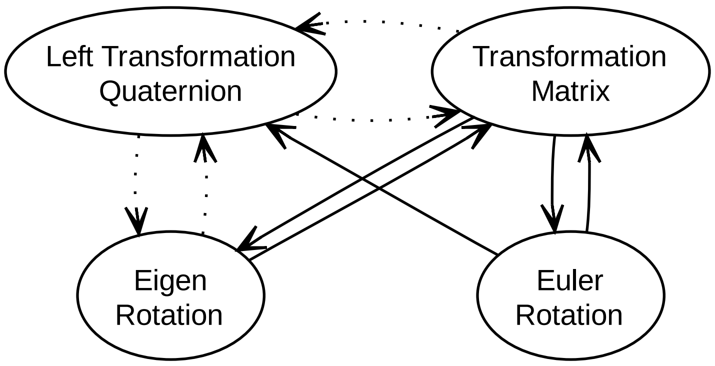

Figure 3.1: Orientations in JEOD

The representation schemes supported in JEOD are

- Special orthogonal matrices, or proper transformation matrices. A proper transformation matrix is an orthogonal matrix whose determinant is one. The Matrix3x3 class in the *Mathematical Functions Model* [8] provides methods for manipulating 3x3 matrices and for using them to transform 3-vectors. Note that JEOD uses transformation matrices rather than rotation matrices.

- Left transformation unit quaternions. The unit quaternions are the subject of the *Quaternion Model* [1]. The Quaternion class defined in that model provides methods that relate to using unit quaternions to represent rotations and transformations. Note that JEOD uses left transformation unit quaternions rather than right quaternions.

- Eigen axis, or single axis, rotations. Any sequence of rotations in three dimensional space is equivalent to a pure rotation about a single axis. JEOD represents eigen rotations in terms of a rotation angle (a scalar) and a unit vector that defines the direction of the rotation via the right-hand rule.

- Euler angles. An Euler rotation sequence is a sequence of three rotations about a set of principal axes. There are twelve possible conventions for these rotations. JEOD implements all twelve conventions.

3.1 Conceptual Design

The Orientation Model defines one primary class, the Orientation class. The model also defines an auxiliary class, the OrientationMessages class, which member functions of the Orientation class use for error reporting.

3.1.1 Static Conversion Methods

The Orientation class defines static methods which can be used as ordinary functions to convert between various representations of an orientation. Each solid line in figure 3.1 represents one of these static conversion methods provided by the Orientation class.

The graph depicted in the figure is not fully connected. Direct conversions do exist from all supported representation schemes to both quaternions and transformation matrices. This is intentional. Other JEOD models use quaternions and transformation matrices to represent orientations. It is best to have a direct conversion from all representation schemes to each of the primary schemes used with JEOD. On the other hand, the eigen rotation and Euler angle schemes are only partially connected in the figure. This too is intentional. There is, for example, no driving reason to provide a direction conversion from eigen rotations to Euler angles. The conversion method would be quite complex and the conversion can still be made by using a transformation matrix as an intermediate step.

3.1.2 Orientation Objects

The Orientation class is also instantiable. An Orientation object contains data members that correspond to each of the four representation schemes supported by JEOD. The object also contains auxiliary data members that indicate the representation scheme that was input by the user and that indicate which of the primary data members have been computed from the user input data.

The class provides several non-static member functions that operate on Orientation objects. Each method belongs to one the following groups:

- Constructors. The class defines a default constructor and also defines four non-default constructors, one per supported representation scheme. One way to initialize an Orientation object is to use one of the non-default constructors to create the Orientation object.

- Destructor. The Orientation destructor does nothing; orientation objects do not allocate resources. The destructor exists so that non-JEOD users can extend the class. The Orientation destructor is a virtual destructor.
- Setters and getters. The model provides setters and getters for each of the four supported representation schemes, a setter and getter to set/retrieve the Euler sequence, and a method to clear (invalidate) the Euler sequence.
- Compute methods. The member data that describe the supported representation schemes are publicly visible. This means outside users can modify the object and potentially render it internally inconsistent. The compute methods provide the means to make an Orientation object consistent with a specified representation scheme.
- Methods provided by c++. Orientation objects do not allocate resources and are not pointers. This means that the copy constructor and assignment operator provided by default by c++ will work properly.
- Internal methods. The class defines several protected methods that are not available to the outside callers. These member functions are protected rather than private, so they are accessible to classes that inherit from the Orientation class.

3.2 Mathematical Formulations

The mathematics of representing orientations in three space are described in many references, including Schaub and Junkins[6]. This section focuses on the mathematics that underlies the connections depicted as solid lines in figure 3.1. The connections depicted as dotted lines, the bidirectional conversions between quaternions and matrices and between quaternions and eigen rotations, are handled by the *Quaternion Model* [1]. Readers interested in the mathematical formulations of those conversions should refer to the documentation for that model.

3.2.1 Eigen Rotations to Matrices

Given a pair of reference frames A and B, with frame B rotated by an angle ϕ about an axis $\hat{\boldsymbol{u}}$ with respect to frame A, the i,j element of the transformation matrix $\boldsymbol{T}$ from frame A to B is given by

$$T_{ij} = \cos\phi\,\delta_{ij} + (1-\cos\phi)\,\hat{u}_i\hat{u}_j + \epsilon_{ijk}\sin\phi\,\hat{u}_k \tag{3.1}$$

where

- Each of i and j are one of 0, 1, or 2,
- $k \equiv i + j \bmod 3$,
- δ_{ij} is the Kronecker delta,

  $$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

- ϵ_{ijk} is the Levi-Civita symbol taken with respect to (0,1,2),

$$\epsilon_{ijk} = \begin{cases} 1 & \text{if } (i,j,k) \text{ is an even permutation of } (0,1,2) \\ -1 & \text{if } (i,j,k) \text{ is an odd permutation of } (0,1,2) \\ 0 & \text{if } i=j,\ i=k, \text{ or } j=k \end{cases}$$

The Levi-Civita symbol vanishes for the diagonal elements while the Kronecker delta vanishes for the off-diagonal elements. Thus the diagonal elements are given by

$$T_{ii} = \cos\phi + (1-\cos\phi)\,\hat{u}_i^2 \tag{3.2}$$
$$T_{ij} = (1-\cos\phi)\,\hat{u}_i\hat{u}_j \pm \sin\phi\,\hat{u}_k,\ \ j \neq i \tag{3.3}$$

Equation (3.3) is additive (the $\pm$ becomes $+$) if (i,j,k) is an even permutation of $(0,1,2)$. The equation is subtractive for an odd permutation of $(0,1,2)$.

3.2.2 Matrices to Eigen Rotations

The trace of a transformation matrix immediately yields one expression for the rotation angle. From equation (3.1), the trace of a transformation matrix is

$$\operatorname{trace}\boldsymbol{T} = 2\cos\phi + 1 \tag{3.4}$$

Solving for $\cos\phi$,

$$\cos\phi = \frac{\operatorname{trace}\boldsymbol{T} - 1}{2} \tag{3.5}$$

The above says nothing about the rotation axis, however. Each diagonal element does contain information about the corresponding member of the eigen rotation axis:

$$u_i = \pm\sqrt{\frac{T_{ii} - \cos\phi}{1-\cos\phi}} \tag{3.6}$$

That the sign is unknown (the $\pm$ factor) presents a challenge with respect to the utility of the above expression.

An alternative expression that contains both the rotation angle and rotation axis is to look at the symmetric difference of the transformation matrix. This yields the difference vector $\boldsymbol{d}$,

$$d_k \equiv T_{ij} - T_{ji} = 2\sin\phi\,\hat{u}_k \tag{3.7}$$

where

$$i = (k+1) \bmod 3$$
$$j = (i+1) \bmod 3$$

Based on this skew symmetric difference vector,

$$\sin\phi = \frac{||\boldsymbol{d}||}{2} \tag{3.8}$$

$$\hat{\boldsymbol{u}} = \frac{\boldsymbol{d}}{||\boldsymbol{d}||} \tag{3.9}$$

Note that equation (3.8) will yield a rotation angle between 0 and 90 degrees. Special processing will be needed when the rotation angle is between 90 and 180 degrees.

One final expression is based on the symmetric sum $\boldsymbol{T}+\boldsymbol{T}^\top$. Given the i^{th} element of the rotation axis unit vector $\hat{u}_i$, this sum yields expressions for the other two elements of the rotation axis unit vector:

$$s_j \equiv T_{ij} + T_{ji} = 2(1-\cos\phi)\,\hat{u}_i\hat{u}_j,\ j \neq i \tag{3.10}$$

and thus

$$\hat{u}_j = \frac{s_j}{2(1-\cos\phi)\,\hat{u}_i} \tag{3.11}$$

Summarizing the above, two equations address the rotation angle ϕ (equations (3.5) and (3.8)) while three address the rotation axis $\hat{\boldsymbol{u}}$ (equations (3.6), (3.9), and (3.11)). Which is the best to use depends on the rotation angle. There are four cases to consider.

Null rotations. All of the expressions for the rotation axis become indeterminate in the case of an identity matrix. This case can be easily identified by looking for $||\boldsymbol{d}|| = 0$ (equation (3.7)) and $\cos\phi = 1$ (equation (3.5)). The rotation axis truly is indeterminate in this case. The resolution to this indeterminacy as implemented in the model code is to arbitrarily say that the rotation is about the x-hat axis.

Small rotations. Small rotations are evidenced by $cos\phi > \sin\phi$, with $\cos\phi$ and $\sin\phi$ given by equations (3.5) and (3.8). This is the region $\phi \in (0, \pi/4)$. The inverse cosine in this region is less accurate than is the inverse sine, making equation (3.8) the better choice for determining the rotation angle. That $\cos\phi$ is somewhat close to one in this interval means that equations (3.6) and (3.11) are suspect in this region. Equation (3.9) is the only viable choice for determining the rotation axis in this region.

Large rotations. Large rotations are evidenced by $-\cos\phi > \sin\phi$. This is the region $\phi \in (3\pi/4, -\pi]$. The inverse cosine in this region is again less accurate than is the inverse sine, once again making equation (3.8) the better choice for determining the rotation angle. In this region it is equation (3.9) that becomes suspect for determining the rotation axis. A combination of equations (3.6) and (3.11) are needed. Equation (3.6) will suffer precision loss when the symmetric sum is close to zero while equation (3.11) cannot be used for all three components. The solution is to use equation (3.6) for the one component corresponding to the largest diagonal element of the matrix and to use equation (3.11) for the other two components.

Two problems remain in this region. Equation (3.6) has a sign uncertainty. Except for rotations very close to 180 degrees, equation (3.6) still has enough precision to enable its use to resolve the sign uncertainty. For rotations of exactly 180 degrees the choice of axis direction is arbitrary. For rotations very close to 180 degrees getting the sign wrong corresponds to a very small rotation error. Thus the use of equation (3.9) to determine the sign in equation (3.6) works just fine. The other problem is that equation (3.8) if applied directly would yield a small rather than large rotation angle. This is easily rectified by correcting for the quadrant in which the angle is known to lie.

Intermediate rotations. The remaining cases, $\phi \in [\pi/4, 3\pi/4]$, represents the region where $\sin\phi \geq |\cos\phi|$. In this region it is equation (3.5) that is more accurate than equation (3.8). Equation (3.5) is used in this region to determine the rotation angle. This region is far from the problematic small and large rotation regions, making the choice of the technique for determining the rotational axis less pressing. Equation (3.9) is used in this to region to determine the rotation because of its simplicity.

3.2.3 Euler Angles to Quaternions

Converting an Euler rotation sequence to a left transformation quaternion is a very simple process.

An Euler rotation sequence is a sequence of individual rotations, with each subsequent rotation in the sequence representing a rotation about the already rotated axes. Since left transformation quaternions chain right to left, the equation for the left transformation quaternion that results from an Euler rotation sequence (θ_0 about $\hat{\boldsymbol{u}}_0$, θ_1 about $\hat{\boldsymbol{u}}_1$, θ_2 about $\hat{\boldsymbol{u}}_2$) is

$$\mathcal{Q}_{\text{seq}} = \mathcal{Q}(\theta_2; \hat{\boldsymbol{u}}_2)\mathcal{Q}(\theta_1; \hat{\boldsymbol{u}}_1)\mathcal{Q}(\theta_0; \hat{\boldsymbol{u}}_0) \tag{3.12}$$

where

$$\mathcal{Q}(\theta_i; \hat{\boldsymbol{u}}_i) = \begin{bmatrix} \cos\theta_i \\ -\sin\theta_i\,\hat{\boldsymbol{u}}_i \end{bmatrix} \tag{3.13}$$

Since each element of an Euler sequence is about one of the three principal axes, forming the unit vectors in equation (3.13) is a trivial matter. These are the canonical unit vectors.

3.2.4 Euler Angles to Matrices

The process of converting an Euler rotation sequence to a transformation matrix is similar to that used for converting an Euler rotation sequence to a left transformation quaternion. Transformation matrices also chain right to left. Thus the equation for the transformation matrix that results from an Euler rotation sequence is

$$\boldsymbol{T}_{\text{seq}} = \boldsymbol{T}(\theta_2; \hat{\boldsymbol{u}}_2)\boldsymbol{T}(\theta_1; \hat{\boldsymbol{u}}_1)\boldsymbol{T}(\theta_0; \hat{\boldsymbol{u}}_0) \tag{3.14}$$

The individual matrices $\boldsymbol{T}_{\theta_i}$ in the above are given by

$$\boldsymbol{T}(\phi;\hat{\boldsymbol{x}}) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix} \tag{3.15}$$

$$\boldsymbol{T}(\theta;\hat{\boldsymbol{y}}) = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \tag{3.16}$$

$$\boldsymbol{T}(\psi;\hat{\boldsymbol{z}}) = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.17}$$

3.2.5 Matrices to Euler Angles

A transformation matrix constructed from an XYZ Euler sequence that involves a rotation of ϕ about the X axis, a rotation of θ about the rotated Y axis, and a rotation of ψ about the rotated Z axis is of the form

$$T_{XYZ} = \begin{bmatrix} \cos\psi\cos\theta & \cdots & \cdots \\ -\sin\psi\cos\theta & \cdots & \cdots \\ \sin\theta & -\cos\theta\sin\phi & \cos\theta\cos\phi \end{bmatrix}$$

Note that $T_{2,0} = \sin\theta$: This element of the matrix depends on θ only. The other two elements of the leftmost column are simple terms that depend on θ and ψ only, and the other two elements of the bottommost row are simple terms that depend on θ and ϕ only. Those five elements are the key to extracting an XYZ Euler sequence from a typical (non-gimbal locked) transformation matrix. The same principle applies to all twelve of the Euler sequences: Five key elements contain all of the information needed to extract the desired sequence. The location and form of those key elements of course depends on the sequence.

A problem arises in the above when $\cos\theta$ is zero, or nearly so. This siutation is called "gimbal lock". The four elements used to determine ϕ and ψ will be nearly zero in this situation. The elements marked as $\cdots$ in the above matrix take on a simpler form in this case. Once again looking at the matrix generated from an XYZ Euler sequence, when $\theta = \pi/2$ the matrix becomes

$$T_{XYZ} = \begin{bmatrix} 0 & \sin(\phi+\psi) & -\cos(\phi+\psi) \\ 0 & \cos(\phi+\psi) & \sin(\phi+\psi) \\ 1 & 0 & 0 \end{bmatrix}$$

In this case there is no way to determine both ϕ and ψ; all that can be determined is their sum. One way to overcome this problem is to arbitrarily set one of those angles to an arbitrary value such as zero. That is the approach used in JEOD. This arbitrary setting enables an XYZ Euler sequence to be extracted from the matrix even in the case of gimbal lock. The same principle once again applies to all twelve Euler sequences.

In summary, for a transformation matrix corresponding to an XYZ sequence,

- One element of the matrix, the [2][0] element, specifies θ.

- The [1][0] and [0][0] elements of the matrix specify ψ when gimbal lock is not present.
- The [1][1] and [2][2] elements of the matrix specify ϕ when gimbal lock is not present.
- The [1][2] and [1][1] elements of the matrix specify ϕ in the case of gimbal lock; ψ is arbitrarily set to zero in this case.

One approach to providing the required capability to convert a matrix to Euler angles would be to write twelve separate conversion methods, one per rotation sequence. This is not the approach taken in JEOD. Extending the above analysis to the remaining eleven sequences provides the essential information needed to extract the Euler angles from a transformation matrix for any of the twelve Euler sequences. The implementation embeds this information in a data structure array. Each element of the array represents one of the twelve sequences and contains

- The indices of the axes about which the rotations are performed. For example, an XYZ sequence entails a rotation about X, then about Y, and then about Z. This is a 0-1-2 sequence. A ZXZ seqence is a 2-0-2 sequence.
- Alternates for the initial and final elements of the sequence. For the aerodynamic sequences (XYZ, ZXY, $\cdots$), these alternates are just the initial and final elements of the sequence. For the astronomical sequences (XYX, XZX, $\cdots$), these alternates are the axis that is not involved in the sequence (e.g., Y or 1 for a ZXZ sequence).
- A boolean that indicates whether the aerodynamic sequence formed by the first two elements of the sequence followed by the alternate final element is an even (true) or odd (false) permutation of XYZ.
- A boolean that indicates whether the sequence is an aerodynamics sequence (true) or an astronomical sequence (false).

The resulting array is

Sequence	Indices	Alt x	Alt z	Is even	Is aero
XYZ	$0, 1, 2$	0	2	true	true
XZY	$0, 2, 1$	0	1	false	true
YZX	$1, 2, 0$	1	0	true	true
YXZ	$1, 0, 2$	1	2	false	true
ZXY	$2, 0, 1$	2	1	true	true
ZYX	$2, 1, 0$	2	0	false	true
XYX	$0, 1, 0$	2	2	true	false
XZX	$0, 2, 0$	1	1	false	false
YZY	$1, 2, 1$	0	0	true	false
YXY	$1, 0, 1$	2	2	false	false
ZXZ	$2, 0, 2$	1	1	true	false
ZYZ	$2, 1, 2$	0	0	false	false

3.3 Detailed Design

The details of the design of the Orientation Model can be found in the JEOD Orientation Model Reference Manual [5].

3.4 Inventory

All Orientation Model files are located in the directory `${JEOD_HOME}/models/utils/orientation`. Relative to this directory,

- Header and source files are located in the model `include` and `src` subdirectories. Table 3.1 lists the configuration-managed files in these directories.
- Verification files are located in the model `verif` subdirectory. See table 3.3 for a listing of the configuration-managed files in this directory.
- Documentation files are located in the model `docs` subdirectory. See table 3.2 for a listing of the configuration-managed files in this directory.

Table 3.1: Source Files

File Name
include/orientation.hh
include/orientation_messages.hh
src/eigen_rotation.cc
src/euler_angles.cc
src/orientation.cc
src/orientation_messages.cc

Table 3.2: Documentation Files

File Name
docs/orientation.pdf
docs/refman.pdf
docs/tex/change_history.tex
docs/tex/intro.tex
docs/tex/ivv.tex
docs/tex/ivv_inspect.tex
docs/tex/ivv_test_eigen.tex
docs/tex/ivv_test_euler.tex

Continued on next page

Table 3.2: Documentation Files (continued from previous page)

File Name
docs/tex/ivv_test_instance.tex
docs/tex/ivv_trace.tex
docs/tex/makefile
docs/tex/model_name.mk
docs/tex/orientation.bib
docs/tex/orientation.sty
docs/tex/orientation.tex
docs/tex/overrides.mk
docs/tex/representations.dot
docs/tex/representations.pdf
docs/tex/reqt.tex
docs/tex/spec.tex
docs/tex/summary.tex
docs/tex/user.tex

Table 3.3: Verification Files

File Name
verif/unit_tests/eigen_rotation/CMakeLists.txt
verif/unit_tests/eigen_rotation/main.cc
verif/unit_tests/eigen_rotation/makefile
verif/unit_tests/euler_angles/CMakeLists.txt
verif/unit_tests/euler_angles/main.cc
verif/unit_tests/euler_angles/makefile
verif/unit_tests/instance/CMakeLists.txt
verif/unit_tests/instance/main.cc
verif/unit_tests/instance/makefile

Chapter 4

User Guide

This chapter describes how to use the Orientation Model from the perspective of a simulation user, a simulation developer, and a model developer.

4.1 Analysis

One use of the Orientation Model is to embed an Orientation object as a data member in a class used to initialize some object. For example, the *Mass Body Model* [7], which uses the Orientation Model to enable a user to specify the orientation of a MassBody object's body reference frame with respect to the MassBody object's structural frame. Another example is in the *Body Action Model* [2]. This model uses the Orientation Model to enable a user to specify the initial orientation of a DynBody object with respect to some other reference frame. The sample code below focus on this latter usage.

To specify an orientation via the Orientation Model, the user must set the Orientation object's `data_source` data member to identify which of the four supported representations is being used and must populate the data member(s) appropriate to that selected representation. To specify an orientation via a

- Transformation matrix:
  - Set the `data_source` data member to `Orientation::InputMatrix` and
  - Populate the `trans` data member with the desired matrix.
- Left transformation quaternion:
  - Set the `data_source` data member to `Orientation::InputQuaternion` and
  - Populate the `quat` data member with the desired quaternion.
- Eigen rotation:
  - Set the `data_source` data member to `Orientation::InputEigenRotation` and
  - Populate the `eigen_angle` and `eigen_axis` data members with the desired rotation angle about the desired rotation axis.

- Euler rotation sequence:
  - Set the `data_source` data member to `Orientation::InputEulerRotation` and
  - Populate the `euler_sequence` and `euler_angles` data members with the desired Euler rotation sequence about the desired rotation axis.

Examples of using a transformation matrix and an Euler sequence are shown below.

Transformation matrix:

```
VEH_OBJ.rot_init.orientation.data_source = Orientation::InputMatrix;
VEH_OBJ.rot_init.orientation.trans[0][0] =
    0.1412307175854331, -0.9892782736897275,  0.03718039289432346;
VEH_OBJ.rot_init.orientation.trans[1][0] =
   -0.7726919750981249, -0.1336331223242702, -0.6205556383087864;
VEH_OBJ.rot_init.orientation.trans[2][0] =
    0.6188707425862546,  0.0589125268795973, -0.7832804849780176;
```

Euler sequence:

```
VEH_OBJ.lvlh_init.orientation.data_source    = Orientation::InputEulerRotation;
VEH_OBJ.lvlh_init.orientation.euler_sequence = Orientation::Roll_Pitch_Yaw;
VEH_OBJ.lvlh_init.orientation.euler_angles[0] {d} = 0.0, 85.0, 1.0;
```

4.2 Integration

The Orientation Model is not intended to be used directly at the S_define level. Orientation objects are almost inevitably embedded as data members of some other class. That said, this does not mean that a simulation integrator has no responsibility with respect to the Orientation Model. For example, the standard set of input files for a multi-body simulation typical specify how the bodies in the simulation attach to one another. These attachments must be physically correct, and ensuring this often is an integration-level task.

4.3 Embedding

As mentioned above, various JEOD models embed an Orientation object as a data member of some class. External model developers are free to do so as well. The user populates the Orientation object; methods of the containing class query the Orientation object to extract the desired orientation data from the object. Model developers who wish to use the model in this way should read the Orientation API.

4.4 Extension

The Orientation class is extensible. The VectorOrientation class defined in the Integration Model verification code is an example of such an extension.

Chapter 5

Verification and Validation

5.1 Inspection

Inspection Orientation_1: Top-level Inspection

This document structure, the code, and associated files have been inspected, and together satisfy requirement Orientation_1.

Inspection Orientation_2: Representations

The model design hinges on a single class, the Orientation class. This class defines two enumerations, one of which identifies each of the required representation schemes. Public member data exist to represent each required representation scheme. Non-static member functions exist to

- Ensure that some desired representation is consistent with input data,
- Set the orientation object to a specified value in any of the supported representation schemes (setters),
- Retrieve the equivalent value of an orientation in any of the supported representation schemes (getters).

By inspection, the Orientation Model satisfies requirements Orientation_2 and Orientation_3.

Inspection Orientation_3: Euler Angles

The Orientation class also defines an enumeration that identifies all twelve Euler angle sequences. The six aerodynamics sequences defined in the latter are numerically identical to the corresponding values in the Trick implementation. Each of the three static conversion methods that pertain to Euler angles handle all twelve sequences.

By inspection, the Orientation Model satisfies requirement Orientation_4.

Inspection Orientation_4: Mathematical Formulation

The implementations of the static conversion methods implement the corresponding algorithms as described in section 3.2.

By inspection, the Orientation Model satisfies requirement Orientation_5.

5.2 Verification

This section describes various tests conducted to verify and validate that the Orientation Model satisfies the requirements levied against it. The tests described in this section are located in the JEOD directory `models/utils/orientation/verif/unit_tests`. As the location suggests, all of the tests used to verify this model are unit tests and follow the JEOD unit test framework. Each subdirectory of the `unit\_tests\verb` directory is a unit test, comprising source files that define the test and a makefile with targets “build” and “run” that run the test.

Test Orientation_1: Eigen Rotation Test

Background The purpose of this test is to determine whether the two static methods that convert eigen rotations to and from transformation matrix are implemented correctly. The transformation from an eigen rotation to a matrix is straight line code. This method involves no branching and has little if any numerical sensitivity.

The true challenge is to verify the correctness of the conversion from transformation matrices to eigen rotations. The underlying algorithm splits rotations into four main groups: Null rotations, small rotations (up to 45 degrees), intermediate rotations (45 to 135 degrees), and large rotations (135 to 180 degrees). The test needs to ensure each case is well-covered, with special attention to rotations very close to 0 and 180 degrees. The algorithm shouldn't be sensitive to which of x, y, or z is the dominant axis, but that selection is the subject of a branch. This needs to be well-covered as well. The test achieves this coverage by overkill.

Test description The test generates 340 unit vectors spread around the unit sphere. A set of 19 eigen rotations are performed for each unit vector. Nine of these rotations are from 0 to 180 degrees in steps of 22.5 degrees. The remaining ten involve rotations very close to 0 and 180 degrees.

Each individual test proceeds by

1. Constructing a left transformation quaternion from the eigen rotation and then constructing a transformation from the quaternion. This provides a means of testing the correctness of the eigen rotation to matrix computation.
2. Constructing a transformation matrix from the eigen rotation using the Orientation model.
3. Comparing the results from steps 1 and 2. The two matrices should be very close to one another. The threshold used is 10^{-15} radians, or numerical error.
4. Computing an eigen rotation from the matrix constructed in step 1. This should reproduce the original rotation angle and unit vector. The combined angular error in the two should once again represent numerical error, or 10^{-15} radians.

Test directory `verif/unit_tests/eigen_rotation`
This is a standard unit test directory with two configuration managed items, main.cc and makefile. Simply type tt make to build the test article and run the test in default mode. This default mode summarizes the results of the test. To see individual output, issue the command `./test_program -verbose` after have made the test article.

Success criteria As mentioned above, each of the 6460 tests must pass both the matrix and eigen tests to within numerical error. An individual test passes only if both subtests pass. All 340*19=6460 tests must pass for the test to pass as a whole.

Test results The test passes.

Relevant requirements This test demonstrates the satisfaction of requirement Orientation_5 sub-requirements 5.1 to 5.2 and partially demonstrates the satisfaction of requirement Orientation_2.

Test Orientation_2: Euler Angles Test

Background The purpose of this test is to determine whether the three static methods that convert to and from Euler angles are implemented correctly. The transformations from an Euler angle sequence to a matrix or a quaternion are fairly simple code. These methods involve little branching and have little if any numerical sensitivity.

As with the eigen rotation test, the true challenge is to verify the correctness of the conversion from transformation matrices to Euler angles. The algorithm is rather complex, with an extended cyclomatic complexity of 14. Making matters worse, each of the twelve Euler rotation sequences potentially has a problem with gimbal lock. Matrices that correspond to gimbal lock / near-gimbal lock conditions need to be tested carefully.

Test description This test centers on the middle angle θ of the Euler rotation sequence. This is the element that determines whether gimbal lock is present. The test covers three intervals for this angle: Angles well-removed from gimbal lock positions, pure gimbal lock, and near-gimbal lock.

For each θ value, the test sweeps both the first angle ϕ and last angle ψ over a range of 360 degrees. Each of the twelve Euler sequences is tested for each (ϕ, θ, ψ) triple. Each test involves

1. Using the model conversions from Euler angles to transformation quaternions and to transformation matrices to compute the quaternion and transformation matrix that corresponds to the given Euler sequence and angles.
2. Using the model conversion from transformation matrix to Euler angles to recompute the Euler angles from the transformation matrix step one above.
3. For aerodynamics sequences only, computing the transformation matrix using the Trick math library and comparing the result to the matrix computed in step one.
4. Computing the error between the quaternion and matrix computed in step one.
5. Computing a quaternion from the Euler angles computed in step two and comparing that to the quaternion computed in step one.

Test directory `verif/unit_tests/euler_angles`
This is a standard unit test directory with two configuration managed items, main.cc and makefile. Simply type `make` to build the test article and run the test in default mode. This default mode summarizes the results of the test. To see individual output, issue the command `./test_program -verbose` after having made the test article.

Success criteria A common threshold of 10^{-15} radians angular error (numerical error) is used for all tests. Over 1.5 million cases are evaluated during the course of the test, and each one of these must pass for the test to pass as a whole.

Test results The test passes.

Relevant requirements This test demonstrates the satisfaction of requirement Orientation_5 sub-requirements 5.3 to 5.5 and partially demonstrates the satisfaction of requirements Orientation_2 and Orientation_4.

Test Orientation_3: Instance Test

Background The two unit tests described above address the static conversion methods provided by the model. These conversion methods are secondary to the primary purpose of the model, which is to serve as an input mechanism by which simulation users can specify an orientation in any of the supported forms. The model accomplishes this by making the Orientation class instantiable. The purpose of this test is to test the completeness and correctness of the treatment of Orientation objects.

The model provides three different schemes for setting an Orientation object: By construction, through setters, and by direct assignment to member data. The model provides two schemes for accessing an Orientation object: Through getters and by direct access to member data. The direct access to member data means that the model must provide means to ensure that an accessible representation is consistent with input. This leads to yet another set of member functions that compute some desired product.

This test tests various combinations of assignment techniques paired with access techniques, with each of the twelve different Euler sequences counted as a separate technique. While the two previous unit tests used numerical overkill to demonstrate correctness, this test uses a combinatoric overkill approach.

Test description This test assumes that all of the direct conversions depicted in figure 3.1 operate correctly. Ignoring the multiplicity of representation schemes, this test uses but one orientation as the basis for testing, a rotation of 120 degrees about the line $x = y = z$. As this orientation is well-removed from all gimbal lock positions, the errors that result from converting from one form to another should be small numerical errors.

The following pairings of assignment and access techniques are tested:

- Setters and getters.
- Constructors and getters.
- Direct assignments and direct access.

Test directory `verif/unit_tests/instance`
This is a standard unit test directory with two configuration managed items, main.cc and makefile. Simply type `make` to build the test article and run the test in default mode. This default mode summarizes the results of the test. To see individual output, issue the command `./test_program -verbose` after having made the test article.

Success criteria Having a known correct response means that the error in any retrieved representation can be readily calculated. The test passes if each of the retrieved representations is within some threshold of the known value. The selected orientation is well-removed from all gimbal lock positions. The errors that result from converting from any one form to another should be very small numerical errors. The test uses a threshold of 10^{-15} radians to detect errors.

Test results The test exhaustively tests all combinations of inputs and outputs. All outputs are numerically close to the expected correct value. The test passes.

Relevant requirements This test demonstrates the satisfaction of requirements Orientation_2 and Orientation_3 and partially demonstrates the satisfaction of requirement Orientation_4.

5.3 Metrics

5.3.1 Code Metrics

Table 5.1 presents coarse metrics on the source files that comprise the model.

Table 5.1: Coarse Metrics

	Number of Lines			
File Name	**Blank**	**Comment**	**Code**	**Total**
include/orientation.hh	80	119	140	339
include/orientation_messages.hh	28	47	17	92
src/eigen_rotation.cc	36	188	90	314
src/euler_angles.cc	58	201	176	435
src/orientation.cc	183	304	447	934
src/orientation_messages.cc	20	27	10	57
Total	405	886	880	2171

Table 5.2 presents the extended cyclomatic complexity (ECC) of the methods defined in the model.

Table 5.2: Cyclomatic Complexity

Method	**File**	**Line**	**ECC**
jeod::Orientation::compute_quaternion_from_euler_angles (void)	include/orientation.hh	261	1
jeod::Orientation::compute_matrix_from_euler_angles (void)	include/orientation.hh	275	1
jeod::Orientation::compute_euler_angles_from_matrix (void)	include/orientation.hh	289	1
jeod::Orientation::compute_matrix_from_eigen_rotation (void)	include/orientation.hh	303	1
jeod::Orientation::compute_eigen_rotation_from_matrix (void)	include/orientation.hh	317	1

Continued on next page

Table 5.2: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::Orientation::compute_matrix_from_eigen_rotation (double eigen_angle, const double eigen_axis[3], double trans[3][3])	src/eigen_rotation.cc	58	1
jeod::Orientation::compute_eigen_rotation_from_matrix (const double trans[3][3], double * eigen_angle, double eigen_axis[3])	src/eigen_rotation.cc	129	10
jeod::Orientation::compute_quaternion_from_euler_angles (EulerSequence euler_sequence, const double euler_angles[3], Quaternion & quat)	src/euler_angles.cc	114	2
jeod::Orientation::compute_matrix_from_euler_angles (EulerSequence euler_sequence, const double euler_angles[3], double trans[3][3])	src/euler_angles.cc	148	5
jeod::Orientation::compute_euler_angles_from_matrix (const double trans[3][3], EulerSequence euler_sequence, double euler_angles[3])	src/euler_angles.cc	203	13
jeod::Orientation::Orientation (void)	src/orientation.cc	62	1
jeod::Orientation::Orientation (const double trans_in[3][3])	src/orientation.cc	90	1
jeod::Orientation::Orientation (const Quaternion & quat_in)	src/orientation.cc	116	1
jeod::Orientation::Orientation (double eigen_angle_in, const double eigen_axis_in[3])	src/orientation.cc	141	1

Continued on next page

Table 5.2: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::Orientation::Orientation (EulerSequence sequence_in, const double angles_in[3])	src/orientation.cc	169	1
jeod::Orientation::~Orientation (void)	src/orientation.cc	197	1
jeod::Orientation::reset (void)	src/orientation.cc	220	1
jeod::Orientation::mark_input_as_available (void)	src/orientation.cc	238	6
jeod::Orientation::compute_transform (void)	src/orientation.cc	316	7
jeod::Orientation::compute_quaternion (void)	src/orientation.cc	369	7
jeod::Orientation::compute_eigen_rotation (void)	src/orientation.cc	421	8
jeod::Orientation::compute_euler_angles (void)	src/orientation.cc	477	11
jeod::Orientation::compute_all_products (void)	src/orientation.cc	548	1
jeod::Orientation::compute_transformation_and_quaternion (void)	src/orientation.cc	565	1
jeod::Orientation::get_transform (double trans_out[3][3])	src/orientation.cc	595	2
jeod::Orientation::get_quaternion (Quaternion & quat_out)	src/orientation.cc	618	2
jeod::Orientation::get_eigen_rotation (double * eigen_angle_out, double eigen_axis_out[3])	src/orientation.cc	641	2
jeod::Orientation::get_euler_angles (EulerSequence * sequence, double angles[3])	src/orientation.cc	667	2
jeod::Orientation::get_euler_angles (double angles[3])	src/orientation.cc	693	2
jeod::Orientation::get_euler_sequence (void)	src/orientation.cc	716	1

Continued on next page

Table 5.2: Cyclomatic Complexity (continued)

Method	**File**	**Line**	**ECC**
jeod::Orientation::set_transform (const double trans_in[3][3])	src/orientation.cc	741	1
jeod::Orientation::set_quaternion (const Quaternion & quat_in)	src/orientation.cc	759	1
jeod::Orientation::set_eigen_rotation (double eigen_angle_in, const double eigen_axis_in[3])	src/orientation.cc	777	1
jeod::Orientation::set_euler_angles (EulerSequence sequence, const double angles[3])	src/orientation.cc	798	3
jeod::Orientation::set_euler_angles (const double angles[3])	src/orientation.cc	831	3
jeod::Orientation::set_euler_sequence (EulerSequence sequence)	src/orientation.cc	867	4
jeod::Orientation::clear_euler_sequence (void)	src/orientation.cc	900	4

5.4 Requirements Traceability

Table 5.3 summarizes the inspections and tests that demonstrate the satisfaction of the requirements levied on the model.

Table 5.3: Requirements Traceability

Requirement	**Inspection or test**
Orientation_1 Project Requirements	Insp. Orientation_1 Top-level Inspection
Orientation_2 Supported Representations	Insp. Orientation_2 Representations Test Orientation_1 Eigen Rotation Test Test Orientation_2 Euler Angles Test Test Orientation_3 Instance Test
Orientation_3 Data Access	Insp. Orientation_2 Representations Test Orientation_3 Instance Test
Orientation_4 Euler Angles	Insp. Orientation_3 Design Inspection Test Orientation_2 Euler Angles Test Test Orientation_3 Instance Test
Orientation_5 Conversions	Insp. Orientation_4 Mathematical Formulation Test Orientation_1 Eigen Rotation Test Test Orientation_2 Euler Angles Test

Bibliography

[1] Hammen, D. Quaternion Model. Technical Report JSC-61777-utils/quaternion, NASA, Johnson Space Center, Houston, Texas, July 2022.

[2] Hammen, D. Body Action Model. Technical Report JSC-61777-dynamics/body_action, NASA, Johnson Space Center, Houston, Texas, July 2022.

[3] Jackson, A., Thebeau, C. JSC Engineering Orbital Dynamics. Technical Report JSC-61777-docs, NASA, Johnson Space Center, Houston, Texas, July 2022.

[4] NASA. NASA Software Engineering Requirements. Technical Report NPR-7150.2, NASA, NASA Headquarters, Washington, D.C., September 2004.

[5] National Aeronautics and Space Administration, Johnson Space Center, Software, Robotics & Simulation Division, Simulation and Graphics Branch, 2101 NASA Parkway, Houston, Texas, 77058. *JEOD Orientation Model Reference Manual*, July 2022.

[6] Schaub, H. and Junkins, J. *Analytical Mechanics of Space Systems*. American Institute of Aeronautics and Astronautics, Inc., Reston, Virginia, 2003.

[7] Thebeau, C. Mass Body Model. Technical Report JSC-61777-dynamics/mass, NASA, Johnson Space Center, Houston, Texas, July 2022.

[8] Thompson, B. Mathematical Functions Model. Technical Report JSC-61777-utils/math, NASA, Johnson Space Center, Houston, Texas, July 2022.