NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer’s or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.
Vectors and Rotations in 3-Dimensions: Vector Algebra for the C++ Programmer

by Richard Saucier
Survivability/Lethality Analysis Directorate, ARL

Approved for public release; distribution is unlimited.
This report describes 2 C++ classes: a Vector class for performing vector algebra in 3-dimensional space (3D) and a Rotation class for performing rotations of vectors in 3D. Each class is self-contained in a single header file (Vector.h and Rotation.h) so that a C++ programmer only has to include the header file to make use of the code. Examples and reference sheets are provided to serve as guidance in using the classes.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vi</td>
</tr>
<tr>
<td>List of Listings</td>
<td>vii</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Vector Class</td>
<td>3</td>
</tr>
<tr>
<td>3. Rotation Class</td>
<td>5</td>
</tr>
<tr>
<td>4. Example Applications</td>
<td>7</td>
</tr>
<tr>
<td>4.1 Impact Location on a Target Plate</td>
<td>7</td>
</tr>
<tr>
<td>4.2 Computing the Rotation between 2 Orientations</td>
<td>11</td>
</tr>
<tr>
<td>4.3 Deflected Spall Cone from an Oblique Shot</td>
<td>14</td>
</tr>
<tr>
<td>5. Conclusion</td>
<td>19</td>
</tr>
<tr>
<td>6. References</td>
<td>20</td>
</tr>
<tr>
<td>Appendix A. Vector Class</td>
<td>21</td>
</tr>
<tr>
<td>Appendix B. Rotation Class</td>
<td>29</td>
</tr>
<tr>
<td>Appendix C. Quaternion Algebra and Vector Rotations</td>
<td>45</td>
</tr>
<tr>
<td>Appendix D. Fundamental Theorem of Rotation Sequences</td>
<td>49</td>
</tr>
<tr>
<td>Appendix E. Factoring a Rotation into a Rotation Sequence</td>
<td>55</td>
</tr>
<tr>
<td>Appendix F. Conversion between Quaternion and Rotation Matrix</td>
<td>69</td>
</tr>
<tr>
<td>Appendix G. Slerp (Spherical Linear Interpolation)</td>
<td>73</td>
</tr>
<tr>
<td>Appendix H. Exact Solution to the Absolute Orientation Problem</td>
<td>81</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Fig.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initial orientation of the target plate in a right-handed Cartesian coordinate system</td>
</tr>
<tr>
<td>2</td>
<td>Description of pitch, yaw, and roll</td>
</tr>
<tr>
<td>3</td>
<td>Geometry for oblique shots</td>
</tr>
<tr>
<td>4</td>
<td>Plots of the 30° (half-angle) spall cone on the witness plate for a range of deflections</td>
</tr>
<tr>
<td>G-1</td>
<td>Spherical linear interpolation over the unit sphere</td>
</tr>
</tbody>
</table>
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-1</td>
<td>Vector: A C++ Class for 3D Vector Algebra—Reference Sheet</td>
<td>27</td>
</tr>
<tr>
<td>B-1</td>
<td>Rotation: A C++ Class for 3D Rotations—Reference Sheet</td>
<td>42</td>
</tr>
<tr>
<td>E-1</td>
<td>Factorization into (z)-(y)-(x) (aerospace) rotation sequence, consisting of yaw about the (z)-axis, pitch about the (y)-axis and roll about the (x)-axis</td>
<td>55</td>
</tr>
<tr>
<td>E-2</td>
<td>Factorization into (x)-(y)-(z) (FATEPEN) rotation sequence, consisting of pitch about the (x)-axis, yaw about the (y)-axis and roll about the (z)-axis</td>
<td>55</td>
</tr>
<tr>
<td>E-3</td>
<td>Factorization into (y)-(x)-(z) rotation sequence</td>
<td>56</td>
</tr>
<tr>
<td>E-4</td>
<td>Factorization into (z)-(x)-(y) rotation sequence</td>
<td>56</td>
</tr>
<tr>
<td>E-5</td>
<td>Factorization into (x)-(z)-(y) rotation sequence</td>
<td>57</td>
</tr>
<tr>
<td>E-6</td>
<td>Factorization into (y)-(z)-(x) rotation sequence</td>
<td>57</td>
</tr>
<tr>
<td>E-7</td>
<td>Factorization into (z)-(y)-(x) rotation sequence</td>
<td>60</td>
</tr>
<tr>
<td>E-8</td>
<td>Factorization into (z)-(x)-(z) rotation sequence</td>
<td>60</td>
</tr>
<tr>
<td>E-9</td>
<td>Factorization into (y)-(z)-(y) rotation sequence</td>
<td>61</td>
</tr>
<tr>
<td>E-10</td>
<td>Factorization into (y)-(x)-(y) rotation sequence</td>
<td>61</td>
</tr>
<tr>
<td>E-11</td>
<td>Factorization into (x)-(y)-(x) rotation sequence</td>
<td>61</td>
</tr>
<tr>
<td>E-12</td>
<td>Factorization into (x)-(z)-(x) rotation sequence</td>
<td>62</td>
</tr>
</tbody>
</table>
## List of Listings

<table>
<thead>
<tr>
<th>Listing</th>
<th>File</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing 1</td>
<td>vtest.cpp</td>
<td>3</td>
</tr>
<tr>
<td>Listing 2</td>
<td>rtest.cpp</td>
<td>5</td>
</tr>
<tr>
<td>Listing 3</td>
<td>geometry.cpp</td>
<td>10</td>
</tr>
<tr>
<td>Listing 4</td>
<td>r2.cpp</td>
<td>11</td>
</tr>
<tr>
<td>Listing 5</td>
<td>oblique.cpp</td>
<td>17</td>
</tr>
<tr>
<td>Listing A-1</td>
<td>Vector.h</td>
<td>22</td>
</tr>
<tr>
<td>Listing B-1</td>
<td>Rotation.h</td>
<td>30</td>
</tr>
<tr>
<td>Listing D-1</td>
<td>order.cpp</td>
<td>51</td>
</tr>
<tr>
<td>Listing E-1</td>
<td>factor.cpp</td>
<td>65</td>
</tr>
<tr>
<td>Listing F-1</td>
<td>convert.cpp</td>
<td>71</td>
</tr>
<tr>
<td>Listing G-1</td>
<td>slerp.cpp</td>
<td>76</td>
</tr>
<tr>
<td>Listing G-2</td>
<td>fast_slerp.cpp</td>
<td>78</td>
</tr>
<tr>
<td>Listing H-1</td>
<td>ao.cpp</td>
<td>84</td>
</tr>
</tbody>
</table>
1. Introduction

This report describes 2 C++ classes: a Vector class for performing vector algebra in 3-dimensional space (3D) and a Rotation class for performing rotations of vectors in 3D. These classes give the programmer the ability to use vectors and rotation operators in 3D as if they were native types in the C++ language. Thus, the code

```cpp
Vector c = a + b; // addition of two vectors
```

performs vector addition, accounting for both magnitude and direction of the vectors to satisfy the parallelogram law of vector addition in exactly the same way as the vector algebra expression \( c = a + b \). We also take advantage of the operator overloading capabilities of C++ so that operations can be written in a more natural style, similar to that of vector algebra. Thus, the code

```cpp
Vector c = a * b; // dot product of two vectors
```

expresses the scalar dot product \( c = a \cdot b \),

```cpp
Vector c = a ^ b; // cross product of two vectors
```

expresses the vector cross product \( c = a \times b \), and

```cpp
Vector c = R * a; // rotation of a vector
```

expresses a rotation of the vector \( a \) by the rotation operator \( R \) to give a vector \( c \). A reference sheet for each class is made available in Appendix A and Appendix B.

Rotations only require an *axis* and an *angle* of rotation—which is how they are stored—and may be specified in a number of convenient ways. We also provide methods for converting from the internal representation to the equivalent quaternion and rotation matrix representation. Quaternion algebra is summarized in Appendix C, which then provides a coordinate-free formula for the rotation of a vector.

It is also useful to describe rotations as a sequence of 3 standard rotations (Euler angles or yaw, pitch, and roll), and Appendix D shows that a rotation sequence about body axes is equivalent to the same rotation sequence applied in reverse order about fixed axes. There are a total of 12 rotation sequences that can be used to describe the orientation of a vector. In Appendix E we provide formulas\(^1\) for factoring an arbitrary rotation into each of these rotation sequences.

Rotations are commonly described with rotation matrices. Appendix F provides formulas and source code for converting between our descriptions of rotations, the quaternion representation, and the rotation matrix.
Quaternions are also very convenient and efficient for describing smooth rotations between 2 different orientations. Appendix G provides a derivation of the spherical linear interpolation (Slerp) formula for this purpose. We also provide a formula and coding for fast incremental Slerp.

Sometimes we need to relate 2 different orientations and find the rotation that will transform from one to the other. This is called the absolute orientation problem and Appendix H provides an exact solution to this problem.

The Rotation and Vector classes provide C++ support for all these operations. No libraries are required and there is nothing to build; one merely needs to include the header file to make use of the class. (The Rotation class includes the Vector class, so one only needs to include Rotation.h to also make use of the Vector class.)

We also provide examples of how these classes can be used to solve real problems.
2. Vector Class

The source code for the Vector class is completely self-contained in the header file Vector.h, which is listed and described in Appendix A. The program in Listing 1 provides some examples of how one might use the Vector class.

Listing 1. vtest.cpp

```cpp
// vtest.cpp: simple program to demonstrate basic usage of Vector class

#include "Vector.h" // only need to include this header file
#include <iostream>
#include <cstdlib>
using namespace va; // vector algebra namespace

int main( void ) {

    // let u be a unit vector that has equal components along all 3 axes
    Vector u = normalize( Vector(1., 1., 1.) );

    // output the vector
    std::cout << "u = " << u << std::endl;

    // show that the magnitude is 1
    std::cout << "magnitude = " << u.r() << std::endl;

    // output the polar angle in degrees
    std::cout << "polar angle (deg) = " << deg( u.theta() ) << std::endl;

    // output the azimuthal angle in degrees
    std::cout << "azimuthal angle (deg) = " << deg( u.phi() ) << std::endl;

    // output the direction cosines
    std::cout << "direction cosines = " << u.dircos( X ) << " " << u.dircos( Y ) << " " << u.dircos( Z ) << std::endl;

    // let ihat, jhat, khat be unit vectors along x-axis, y-axis, and z-axis, respectively
    Vector ihat( 1., 0., 0. ), jhat( 0., 1., 0. ), khat( 0., 0., 1. );

    // output the vectors
    std::cout << "ihat = " << ihat << std::endl;
    std::cout << "jhat = " << jhat << std::endl;
    std::cout << "khat = " << khat << std::endl;

    // rotate ihat, jhat, khat about u by 120 degrees
    ihat.rotate( u, rad( 120. ) );
    jhat.rotate( u, rad( 120. ) );
    khat.rotate( u, rad( 120. ) );

    // output the rotated vectors
    std::cout << "after 120 deg rotation about u:
    ihat is now = " << ihat << std::endl;
    std::cout << "jhat is now = " << jhat << std::endl;
    std::cout << "khat is now = " << khat << std::endl;

    // define two vectors, a and b
    Vector a( 2., 1., -1. ), b( 3., -4., 1. );
    std::cout << "a = " << a << std::endl;
    std::cout << "b = " << b << std::endl;
    std::cout << "a + b = " << a + b << std::endl;
    std::cout << "a - b = " << a - b << std::endl;

    // compute and output the dot product
    double s = a * b;
    std::cout << "dot product, a * b = " << s << std::endl;

    // compute and output the cross product
    Vector c = a ^ b;
    std::cout << "cross product, a ^ b = " << c << std::endl;

    // output the angle (deg) between a and b
    std::cout << "angle between a and b (deg) = " << deg( angle( a, b ) ) << std::endl;

    // compute and output the projection of a along b
    std::cout << "proj( a, b ) = " << proj( a, b ) << std::endl;

    // rotate a and b 120 deg about u
    a.rotate( u, rad( 120. ) );
    b.rotate( u, rad( 120. ) );
    std::cout << "after rotating a and b 120 deg about u: " << a << std::endl;
```

Approved for public release; distribution is unlimited.
72  std::cout << "a is now = " << a << std::endl;
73  std::cout << "b is now = " << b << std::endl;
74  // output the angle (deg) between a and b
75  std::cout << "angle between a and b (deg) is now = " << deg( angle( a, b ) ) << std::endl;
76  // compute and output the dot product
77  s = a * b;
78  std::cout << "dot product, a * b is now = " << s << std::endl;
79  // compute and output the cross product
80  c = a ^ b;
81  std::cout << "cross product, a ^ b is now = " << c << std::endl;
82  // set a and b to their original values and compute the cross product
83  a = Vector( 2., 1., -1. );
84  b = Vector( 3., -4., 1. );
85  c = a ^ b;
86  std::cout << "original cross product, c = " << c << std::endl;
87  // rotate c 120 deg about u and output
88  c.rotate( u, rad( 120. ) );
89  std::cout << "now rotate c 120 deg about u:
90  // after 120 deg rotation about u:
91  hat = 0 0 1
92  what = 0 1 0
93  what is now = 0 1 0
94  a = 2 1 -1
95  b = 3 -4 1
96  a + b = 5 -3 0
97  a - b = -1 5 -2
98  dot product, a · b = 1
99  cross product, a × b = -3 -5 -11
100  angle between a and b (deg) = 85.4078
101  proj( a, b ) = 0.115385 -0.153846 0.0384615
102  after rotating a and b 120 deg about u: -1 2 1
103  a is now = -1 2 1
104  b is now = -1 3 -4
105  angle between a and b (deg) is now = 85.4078
106  dot product, a · b is now = -11 -3 -5
107  cross product, a × b is now = -11 -3 -5
108  original cross product, c = -3 -5 -11
109  now rotate c 120 deg about u:
110  c is now = -11 -3 -5

Save this to a file vtest.cpp and compile it with the command

g++ -O2 -Wall -o vtest vtest.cpp -lm

Running it

./vtest

will print the following:

u = 0.57735 0.57735 0.57735
magnitude = 1
polar angle (deg) = 54.7356
azimuthal angle (deg) = 45
direction cosines = 0.57735 0.57735 0.57735
ihat = 1 0 0
jhat = 0 1 0
khat = 0 0 1
after 120 deg rotation about u:
ihat is now = 0 1 0
jhat is now = 0 0 1
khat is now = 1 0 0
a = 2 1 -1
b = 3 -4 1
a + b = 5 -3 0
a - b = -1 5 -2
dot product, a · b = 1
cross product, a × b = -3 -5 -11
angle between a and b (deg) = 85.4078
proj( a, b ) = 0.115385 -0.153846 0.0384615
after rotating a and b 120 deg about u: -1 2 1
a is now = -1 2 1
b is now = -1 3 -4
angle between a and b (deg) is now = 85.4078
dot product, a · b is now = -11 -3 -5
cross product, a × b is now = -11 -3 -5
original cross product, c = -3 -5 -11
now rotate c 120 deg about u:
c is now = -11 -3 -5

Approved for public release; distribution is unlimited.
3. Rotation Class

Similarly, the source code for the Rotation class is completely self-contained in the header file Rotation.h, which is listed and described in Appendix B. The program in Listing 2 provides some basic examples of usage.

Listing 2. rtest.cpp

```cpp
#include "Rotation.h"
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <iomanip>
using namespace va;

int main( void ) {
  // declare two unit vectors, u and v
  Vector u = Vector( 1.5, 2.1, 3.2 ).unit(), v = Vector( 1.2, 3.5, -2.3 ).unit();
  std::cout << "u = " << u << std::endl;
  std::cout << "v = " << v << std::endl;

  // let R be the rotation specified by the vector cross product u ^ v
  Rotation R( u, v );
  std::cout << "R * u = " << R * u << std::endl;

  // interpolate a unit vector, w, halfway between u and v
  Vector w = slerp( u, v, 0.5 );
  std::cout << "Vector halfway between u and v = " << w << std::endl;

  // output angle between u and v
  std::cout << "angle between u and v (deg) = " << deg( angle( u, v ) ) << std::endl;

  // output angle between u and w
  std::cout << "angle between u and w (deg) = " << deg( angle( u, w ) ) << std::endl;

  // let Rinv be the inverse rotation
  Rotation Rinv = inverse( R );
  std::cout << "Rinv * v = " << Rinv * v << std::endl;

  Vector ihat( 1., 0., 0. ), jhat( 0., 1., 0. ), khat( 0., 0., 1. );
  R = Rotation( ihat, jhat, khat, jhat, khat, ihat );
  std::cout << "R = " << R << std::endl;

  sequence s = factor( R, ZYX );
  std::cout << "yaw (deg ) = " << deg( s.first ) << std::endl;
  std::cout << "pitch (deg ) = " << deg( s.second ) << std::endl;
  std::cout << "roll (deg ) = " << deg( s.third ) << std::endl;

  R = Rotation( s.first, s.second, s.third, ZYX );
  std::cout << "R = " << R << std::endl;

  s = factor( R, XYZ );
  std::cout << "yaw (deg ) = " << deg( s.first ) << std::endl;
  std::cout << "pitch (deg ) = " << deg( s.second ) << std::endl;
  std::cout << "roll (deg ) = " << deg( s.third ) << std::endl;

  R = Rotation( s.first, s.second, s.third, XYZ );
  std::cout << "R = " << R << std::endl;

  rng::Random rng;
  R = Rotation( rng );
  std::cout << "R = " << R << std::endl;

  quaternion q = to_quaternion( R );
  std::cout << "the quaternion for this rotation is" << std::endl;
  std::cout << "q = " << q << std::endl;

  R = Rotation( q );
  std::cout << "R = " << R << std::endl;

  matrix A = to_matrix( R );
  std::cout << "the matrix for this rotation is" << std::endl;
  std::cout << std::setprecision(6) << std::fixed << std::showpos;
  std::cout << A << std::endl;
```

Approved for public release; distribution is unlimited.
std::cout << std::setprecision(ss) << std::noshowpos;
R = Rotation( A );
std::cout << "R = " << R << std::endl;

u = normalize( Vector( 1., 1., 1. ) );
double th = rad( 120. );
R = Rotation( u, th );
A = to_matrix( A );
q = to_quaternion( R );

return EXIT_SUCCESS;
}

Writing this to a file rtest.cpp, compiling and running it,
g++ -O2 -Wall -o rtest rtest.cpp -lm
./rtest
produces the following output:

u = 0.364878 0.510829 0.778407
v = 0.275444 0.880338 -0.527934
R * u = 0.275444 0.803378 -0.527934
Vector halfway between u and v = 0.431716 0.886861 0.166873
angle between u and v (deg) = 42.132
Rinv * v = 0.364878 0.510829 0.778407
R = 0.57735 0.57735 0.57735 120
yaw (deg ) = -90
pitch (deg ) = -180
roll (deg ) = -90
R = 0.57735 0.57735 0.57735 120
pitch (deg ) = 45
yaw (deg ) = 90
roll (deg ) = 45
R = 0.57735 0.57735 0.57735 120
R = 0.338694 0.929155 0.148182 174.087
the quaternion for this rotation is
q = 0.0515815 0.338243 0.927918 0.147985
R = 0.338694 0.929155 0.148182 174.087
the matrix for this rotation is
R = 0.338694 0.929155 0.148182 174.08566

Approved for public release; distribution is unlimited.
4. Example Applications

The preceding programs exercised some basic capabilities of the 2 classes. Now let us consider some more practical problems to see how these classes aid in their solution. Vectors are powerful tools in 3D problems and it is usually better to make use of the vectors directly in vector algebra rather than to decompose into coordinates, and these classes allow us to do that.

4.1 Impact Location on a Target Plate

Here is an example problem. We have a target plate that is initially placed in the $x$-$y$ plane, as shown in Fig. 1, where $L$ is the length, $W$ the width, and $T$ the thickness of the plate, defined such that $L \geq W \geq T$. We first define a standard orientation of the target plate as the initial orientation and then describe the operational procedure to give it a specific, final orientation. This standard orientation is with the center at the origin of a right-handed cartesian $(x, y, z)$ coordinate system and its length along the $x$-axis, its width along the $y$-axis, and its thickness along the $z$-axis, as illustrated in Fig. 1. Once it has been given a final orientation, we translate its center to the location $r_c$. Then we shoot a fragment at the plate along a ray from the origin and we want to find the impact point on the target, along with its distance and its impact obliquity.

![Fig. 1. Initial orientation of the target plate in a right-handed Cartesian coordinate system](image)

We also define the 3 unit vectors $\hat{i}$, $\hat{j}$, and $\hat{k}$ along the $x$-axis, $y$-axis, and $z$-axis, respectively. The target plate (entrance) normal $\hat{n}$ is initially aligned with $\hat{k}$. The final orientation is specified by performing a pitch-yaw-roll rotation sequence, where $\phi_p$ is pitch about the $x$-axis, $\phi_y$ is yaw about the $y$-axis, and $\phi_r$ is roll about the
$z$-axis, as shown in Fig. 2.

![Fig. 2. Description of pitch, yaw, and roll](image)

Here we adopt the FATEPEN (Fast Air Target Encounter Penetration)\textsuperscript{2,3} convention that first pitch is applied, then yaw, then roll.\textsuperscript{*} We denote a rotation about the unit axial vector $\hat{e}$ through an angle $\phi$ with the notation $R_{\hat{e}}(\phi)$. A rotation sequence can thus be written as

$$R \equiv R_{\hat{k}}(\phi_r)R_{\hat{j}}(\phi_y)R_{\hat{i}}(\phi_p),$$

\hspace{1cm} (1)

where the order is from right to left: first pitch is applied, then yaw, then roll. Notice that we have primes on the yaw and roll rotation operators to indicate that the unit vectors get transformed after pitch is applied and after yaw is applied, since the rotations are applied to the body axes of the plate. Now it is a fundamental theorem of rotation sequences that rotations about the body axes is equivalent to the same sequence in reverse order about the fixed axes\textsuperscript{†}, and since it is more efficient to rotate about fixed axes, we have

$$R = R_{\hat{k}}(\phi_r)R_{\hat{j}}(\phi_y)R_{\hat{i}}(\phi_p) = R_{\hat{i}}(\phi_p)R_{\hat{j}}(\phi_y)R_{\hat{k}}(\phi_r).$$

\hspace{1cm} (2)

Hence, the normal vector is given by

$$\hat{n} = R_{\hat{i}}(\phi_p)R_{\hat{j}}(\phi_y)R_{\hat{k}}(\phi_r)\hat{k}.$$ 

\hspace{1cm} (3)

Once the target plate is oriented, we specify the location of its center with the vector $r_c$. So the final position and orientation of the target plate is specified by the pair of vectors $(r_c, \hat{n})$.\textsuperscript{*} There are a total of 12 different conventions in the Rotation class that one may choose from. \textsuperscript{†} See Appendix D for a proof.

Approved for public release; distribution is unlimited.
Let \( r_0 \) be the origin of the fragment ray and let \( \hat{u} \) be a unit vector along the ray. We specify the orientation of \( \hat{u} \) by specifying the polar angle \( \theta \) and azimuth angle \( \phi \), so that
\[
\hat{u} = \sin \theta \cos \phi \hat{i} + \sin \theta \sin \phi \hat{j} + \cos \theta \hat{k},
\]
and the equation for the fragment ray is
\[
r = r_0 + t\hat{u},
\]
where \( t \geq 0 \) is a scalar that measures the distance from the origin. The target plane (infinite in extent) is specified by all points \( r \) such that
\[
(r - r_c) \cdot \hat{n} = 0,
\]
since the normal, by definition, is orthogonal to the plane. Therefore, the intersection of the fragment ray with the plane of the target is specified by the vector equation
\[
(r_0 + t\hat{u} - r_c) \cdot \hat{n} = 0.
\]
Solving for the distance \( t \) gives
\[
t = \frac{(r_c - r_0) \cdot \hat{n}}{\hat{u} \cdot \hat{n}}.
\]
There will be an intersection, provided \( \hat{u} \cdot \hat{n} \neq 0 \), and the intersection point is
\[
r = r_0 + \frac{(r_c - r_0) \cdot \hat{n}}{\hat{u} \cdot \hat{n}} \hat{u}.
\]
The impact obliquity is given by
\[
\theta_{\text{obl}} = \cos^{-1}(|\hat{u} \cdot \hat{n}|).
\]
The location of the impact point with respect to the target center is
\[
r - r_c = r_0 - r_c + \frac{(r_c - r_0) \cdot \hat{n}}{\hat{u} \cdot \hat{n}} \hat{u}
\]
To determine if the intersection point on the infinite target plane lies within the finite target plate, we apply the inverse rotation \( R^{-1} \) to the vector \( r - r_c \) and check to see
if the resulting vector lies within the plate dimensions. Thus, let

\[ d = R^{-1}(r - r_c) \]  \hspace{1cm} (12)

and then we check to see if

\[-L/2 \leq d_x \leq L/2 \quad \text{and} \quad -W/2 \leq d_y \leq W/2.\]  \hspace{1cm} (13)

Listing 3 is an implementation of these vector equations.

**Listing 3. geometry.cpp**

```cpp
#include <cmath>
#include <cassert>
using namespace std;

int main( void ) {
    const double L = 96., W = 48., L_2 = L / 2., W_2 = W / 2.;
    const va::Vector I( 1., 0., 0 ), J( 0., 1., 0 ), K( 0., 0., 1 );
    std::cout << std::setprecision(3) << std::fixed;
    va::Vector n = K; // normal vector is initially along K
    va::Vector u; // unit vector along fragment ray
    va::Vector rc( 0., 0., -24. ); // location of the target center
    va::Vector r0( 0., 0., 0. ); // origin of the fragment ray
    va::Rotation R( pitch, yaw, roll, va::XYZ );
    va::Rotation Rinv = va::inverse( R );
    n = R * n; // apply the orientation to the target normal
    double az = -35., el = 15.; // specify azimuth and elevation of fragment ray (deg)
    double th = va::rad( 180. - el ); // convert to polar angle (rad)
    double ph = va::rad( az ); // convert to azimuthal angle (rad)
    double obl = angle( u, -n ); // obliquity angle (rad)
    if ( obl >= M_PI_2 ) {
        cout << "ray missed target since obl (deg) = " << obl * va::R2D << endl;
        exit( EXIT_SUCCESS );
    }

    // compute distance to (infinite) target plane
    double t = ( rc - r0 ) * n / ( u * n );
    va::Vector r = r0 + t * u; // hit point in lab frame
    va::Vector d = r - rc; // hit point relative to the target center
    d = Rinv * d; // inverse rotation back to the laboratory reference frame
    double xhit = d.x(), yhit = d.y();
    // check if hit point on plane lies with target plate
    if ( ( -L_2 <= xhit && xhit <= L_2 ) && ( -W_2 <= yhit && yhit <= W_2 ) ) {
        cout << "hit point on plane lies with target plate\n";
    } else {
        cout << "ray missed target\n";
    }
    cout << "hit point (x,y) = (" << xhit << ", " << yhit << 
```
cout << "ray hit target at " << d << " with respect to the target at the origin" << endl;
+ cout << "obl (deg) = " << obl * va::R2D << endl;
+ cout << "distance = " << t << endl;
}
else {
+ cout << "ray missed target plate" << endl;
+ cout << "xhit = " << xhit << endl;
+ cout << "yhit = " << yhit << endl;
}
return EXIT_SUCCESS;

Compiling and running this program gives the following output:

```
ray hit target at 2.794 -5.560 0.000 with respect to the target at the origin
obl (deg) = 41.752
distance = 22.822
```

### 4.2 Computing the Rotation between 2 Orientations

We may know the unit vectors $\hat{i}$, $\hat{j}$, $\hat{k}$ in 2 different reference frames and we need to find the rotation that takes one to the other. Thus, let us suppose that we have 2 sets of orthonormal (basis) vectors, $(\hat{i}, \hat{j}, \hat{k})$ and $(\hat{i}', \hat{j}', \hat{k}')$, and we want to find the rotation $R$ such that

$$
\hat{i}' = R\hat{i}, \quad \hat{j}' = R\hat{j}, \quad \text{and} \quad \hat{k}' = R\hat{k}.
$$

This is a relatively simple problem for unit vectors. A much more difficult problem is to find the rotation between 2 sets of 3 vectors when the vectors are not unit vectors. A closed-form solution to this problem is summarized in Appendix H. Both of these cases have been implemented in the Rotation class. The Listing 4 provides a demonstration of this.

#### Listing 4. r2.cpp

```cpp
#include "Rotation.h"
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <iomanip>
#include <cassert>
using namespace std;

int main( int argc, char* argv[] ) {

    // constant unit vectors for the laboratory frame
    const va::Vector i( 1., 0., 0 ), j( 0., 1., 0.), k( 0., 0., 1. );

    double p = 0., y = 0., r = 0.;
    if ( argc == 4 ) {
        p = atof( argv[1] );
        y = atof( argv[2] );
        r = atof( argv[3] );
    }

    std::cout << std::setprecision(3) << std::fixed;
    // specify pitch, yaw and roll of the target (degrees converted to radians)
```
double pitch = va::rad( p );
double yaw = va::rad( y );
double roll = va::rad( r );

// specify the rotation
va::Rotation R1( pitch, yaw, roll, va::XYZ );

// apply the rotation to the basis vectors
va::Vector ip = R1 * i;
va::Vector jp = R1 * j;
va::Vector kp = R1 * k;

cout << "i = " << i << endl;
cout << "j = " << j << endl;
cout << "k = " << k << endl;
cout << "ip = " << ip << endl;
cout << "jp = " << jp << endl;
cout << "kp = " << kp << endl;
cout << endl << "Now we find the rotation" << endl;
va::Rotation R1( i, j, k, ip, jp, kp );
cout << "Applying the found rotation to i, j, k gives" << endl;
ip = R1 * i;
jp = R1 * j;
kp = R1 * k;
cout << "ip = " << ip << endl;
cout << "jp = " << jp << endl;
cout << "kp = " << kp << endl;
cout << endl << "Now suppose we want to factor this rotation into a FATEPEN sequence" << endl;
va::sequence s = va::factor( R1, va::XYZ );
cout << "This gives:" << endl;
cout << va::deg( s.first ) << endl;
cout << va::deg( s.second ) << endl;
cout << va::deg( s.third ) << endl;
cout << endl << "Now for something completely different: Start with the non-unit vectors" << endl;
va::Vector a( 1., 2., 3. ), b( -1., 2., 4. ), c( 4., 3., 9. );
va::Vector ap, bp, cp;
cout << "a = " << a << endl;
cout << "b = " << b << endl;
cout << "c = " << c << endl;
ap = R1 * a;
bp = R1 * b;
cp = R1 * c;
cout << "ap = " << ap << endl;
cout << "bp = " << bp << endl;
cout << "cp = " << cp << endl;
cout << endl << "Now we find the rotation that takes (a,b,c) to (ap,bp,cp)" << endl;
va::Rotation R2( a, b, c, ap, bp, cp );
cout << "Applying the found rotation to a, b, c gives" << endl;
ap = R2 * a;
bp = R2 * b;
cp = R2 * c;
cout << "ap = " << ap << endl;
cout << "bp = " << bp << endl;
cout << "cp = " << cp << endl;
cout << endl << "Also, suppose we want to factor this rotation into a FATEPEN sequence" << endl;
s = va::factor( R2, va::XYZ );
cout << "This gives:" << endl;
cout << va::deg( s.first ) << endl;
cout << va::deg( s.second ) << endl;
cout << va::deg( s.third ) << endl;
return EXIT_SUCCESS;
We don’t have to worry about whether the vectors are unit vectors or not, the Rotation
class figures that out and performs the simpler method when it’s able. Compiling
and running this program with the command

```
./r2 60. -45. 15.
```
gives the following output:

```
i = 1.000 0.000 0.000
j = 0.000 1.000 0.000
k = 0.000 0.000 1.000
ip = 0.683 -0.462 0.566
jp = -0.183 0.641 0.745
kp = -0.707 -0.612 0.354
```

Now we find the rotation

Applying the found rotation to i, j, k gives

```
ip = 0.683 -0.462 0.566
jp = -0.183 0.641 0.745
kp = -0.707 -0.612 0.354
```

Now suppose we want to factor this rotation into a FATEPEN sequence

This gives:

```
60.000
-45.000
15.000
```

Now for something completely different: Start with the non-unit vectors

```
a = 1.000 2.000 3.000
b = -1.000 2.000 4.000
c = 4.000 3.000 9.000
ap = -1.804 -1.016 3.116
bp = -3.877 -0.704 2.339
cp = -4.181 -5.435 7.680
```

Now we find the rotation that takes (a,b,c) to (ap,bp,cp)

Applying the found rotation to a, b, c gives

```
ap = -1.804 -1.016 3.116
bp = -3.877 -0.704 2.339
cp = -4.181 -5.435 7.680
```

Also, suppose we want to factor this rotation into a FATEPEN sequence

This gives:

```
60.000
-45.000
15.000
```
4.3 Deflected Spall Cone from an Oblique Shot

When an overmatching penetrator perforates target armor, it generates a cone of spall fragments from the back of the armor. To better understand the threat this poses, individual spall fragments are registered as holes in a series of witness plates that are located some distance behind the target, typically 24 inches. In the case of an oblique shot, the target is at an obliquity angle $\alpha$ with respect to the shotline, and the witness plates are typically at half that angle, or $\alpha/2$, as depicted in Fig. 3. Although there are usually 5 plates in the witness pack during test conditions, here we will only consider the first plate of the pack.

![Fig. 3. Geometry for oblique shots](image)

What we are going to do here is first make use of vector algebra to describe the coordinate system and the geometry of the test layout, along with the operational procedure to orient the target and witness plate. Following that, we will then show that it is easy to implement this into a C++ program by making use of the Vector and Rotation classes and by a straightforward translation of the vector equations.
Of course we need a coordinate system, but the coordinate system is constructed from the test geometry, not vice versa. Once the coordinate system is established, we can pretty much ignore the actual coordinates and just deal with vector quantities.

So we begin with the direction of the shot line and take this to be along the negative $z$-axis. Then we place the target plate in standard orientation, with its center at the origin, perpendicular to the shotline, its length along the $x$-axis, and its width along the $y$-axis. The same applies to the witness plate. These are just initial orientations; we will subsequently describe the procedure to put them in their final orientations and positions. Let $\hat{u}_{sl}$ be a unit vector along the initial shotline. In our case, $\hat{u}_{sl} = -\hat{k}$, and remains fixed.

Next we orient the target with respect to the shotline. There are at least a couple of convenient ways to do this. We could specify directly the final orientation of the target, in which case the Rotation operator would be generated from the cross product from initial to final orientation.\footnote{Given an initial and a final orientation, $\hat{n}_1$ and $\hat{n}_2$, respectively, the Rotation operator that performs this rotation is $R_{\hat{n}}(\theta)$, where $\hat{n} = \hat{n}_1 \times \hat{n}_2 \overline{||\hat{n}_1 \times \hat{n}_2||}$ and $\theta = \cos^{-1}(|\hat{n}_1 \cdot \hat{n}_2|)$.}

Another method is to specify the rotation sequence that will take it from its initial orientation to its final orientation. We shall use that method here and adopt the FATEPEN\textsuperscript{2,3} convention of first pitch about the $x$-axis, followed by yaw about the $y$-axis, and end with roll about the $z$-axis, so that

$$R_{\text{trgt}} = R_{\hat{i}}(\phi_p) R_{\hat{j}}(\phi_y) R_{\hat{k}}(\phi_r),$$

(15)

where the order goes from right to left and is reversed since we rotate about fixed axes, not body axes.\footnote{See Appendix D for a justification.} Since the initial orientation is $-\hat{k}$, the final orientation is

$$\hat{n}_{\text{trgt}} = R_{\text{trgt}}(-\hat{k}).$$

(16)

The target obliquity is labeled $\alpha$, and

$$\alpha \equiv \theta_{\text{trgt}} = \cos^{-1}(|\hat{u}_{sl} \cdot \hat{n}_{\text{trgt}}|).$$

(17)

Next we want to orient the witness plate to be at half the obliquity angle of the target.
This can be achieved with the aid of the spherical linear interpolation function†

\[ \hat{n}_{wp} = \text{slerp}(-\hat{k}, \hat{n}_{\text{trgt}}, 1/2), \]  

(18)

which gives us a unit vector that is half the angle between \(-\hat{k}\) and \(\hat{n}_{\text{trgt}}\), and the witness plate rotation from standard orientation is

\[ R_{wp} = R_{\hat{n}}(\alpha/2) \quad \text{where} \quad \hat{n} = \frac{(-\hat{k}) \times \hat{n}_{wp}}{\|(-\hat{k}) \times \hat{n}_{wp}\|}. \]  

(19)

We might want to factor this rotation into a pitch-yaw-roll rotation sequence, since that would give us an operational procedure for achieving the orientation of the witness plate such that it is precisely at one-half the target orientation.

Now let us consider the deflection. If \(\hat{u}_{sl}\) and \(\hat{n}_{\text{trgt}}\) are not in the same direction (i.e., the target is at non-zero obliquity) then their vector cross product is non-zero and we can define the unit vector

\[ \hat{w} = \frac{\hat{u}_{sl} \times \hat{n}_{\text{trgt}}}{\|\hat{u}_{sl} \times \hat{n}_{sl}\|}. \]  

(20)

The deflected shot line is obtained by rotating \(\hat{u}_{sl}\) about the unit vector \(\hat{w}\) through the rotation angle \(\beta\), where \(0 \leq \beta \leq \alpha\). Our notation for this rotation is \(R_{\hat{w}}(\beta)\). The witness plate has its center at the location \(r_c\) with respect to the rear of the target plate, typically 24 inches.

Now consider spall fragments that are coming off from the rear of the target. We can parametrize these fragments with the polar angle \(\theta\), measured from the \(-z\)-axis, and the azimuthal angle \(\phi\), measured from the \(x\)-axis.

\[ \hat{u}_{\text{frag}} = \sin(\pi - \theta) \cos \phi \hat{i} + \sin(\pi - \theta) \sin \phi \hat{j} + \cos(\pi - \theta) \hat{k} \]
\[ = \sin \theta \cos \phi \hat{i} + \sin \theta \sin \phi \hat{j} - \cos \theta \hat{k} \]  

(21)

The spall fragments then get deflected according to the equation

\[ \hat{u}'_{\text{frag}} = R_{\hat{w}}(\beta) \hat{u}_{\text{frag}}. \]  

(22)

The (deflected) fragment will travel in a straight line until it encounters the witness

---

†See Appendix G.
plate, so the equation for its trajectory is

\[ r = t \hat{u}_{\text{frag}}, \]  \hspace{1cm} (23)  

where the scalar \( t > 0 \) measures distance from the spall origin. The equation of the plane of the witness plate is the set of all points \( r \) that satisfy

\[ (r - r_c) \cdot \hat{n}_{\text{wp}} = 0. \]  \hspace{1cm} (24)  

Substituting Eq. 23 into Eq. 24 and solving for \( t \) gives for the distance

\[ t = \frac{r_c \cdot \hat{n}_{\text{wp}}}{\hat{u}_{\text{frag}} \cdot \hat{n}_{\text{wp}}}. \]  \hspace{1cm} (25)  

Hence, the hit point on the witness plate is

\[ r_{\text{hit}} = \left( \frac{r_c \cdot \hat{n}_{\text{wp}}}{\hat{u}_{\text{frag}} \cdot \hat{n}_{\text{wp}}} \right) \hat{u}_{\text{frag}}. \]  \hspace{1cm} (26)  

To get the location of the hit point with respect to the witness plate center, we apply

\[ R_{\text{wp}}^{-1}(r_{\text{hit}} - r_c), \]  \hspace{1cm} (27)  

where \( R_{\text{wp}}^{-1} \) is the inverse of the witness plate rotation, given by Eq. 19. The obliquity angle of each frag as it strikes the witness plate is

\[ \theta_{\text{obl}} = \cos^{-1} \left( \frac{|\hat{u}_{\text{frag}} \cdot \hat{n}_{\text{wp}}|}{\hat{u}_{\text{frag}} \cdot \hat{n}_{\text{wp}}} \right). \]  \hspace{1cm} (28)  

The program in Listing 5 is an implementation of these equations and procedures.

**Listing 5. oblique.cpp**

```cpp
#include "Rotation.h"
#include <iostream>
#include <cstdlib>
using namespace va;
using namespace std;

int main( int argc, char* argv[] ) {
    int N = 300; // number of frags on the spall cone
    const double WP_TARGET_OBL = 0.5; // ratio of witness plate obliquity to target obliquity (typically 0.5)
    const double CONE_HALF_ANGLE = 30.; // cone half angle (deg)

    Vector ihat( 1., 0., 0. ), jhat( 0., 1., 0. ), khat( 0., 0., 1. ); // basis vectors

    // oblique.cpp: geometry for oblique shots using a right-handed cartesian coordinate system
    // shotline is fixed along the -z axis
    // initial orientation of target is in the x-y plane with center at the origin
    // initial orientation of witness plate is also in the x-y plane, but with center at z = -24 inches
    // include 'Rotation.h'
    // include <iostream>
    // include <cstdlib>
    using namespace va;
    using namespace std;

    int main( int argc, char* argv[] ) {
        const int N = 300; // number of frags on the spall cone
        const double WP_TARGET_OBL = 0.5; // ratio of witness plate obliquity to target obliquity (typically 0.5)
        const double CONE_HALF_ANGLE = 30.; // cone half angle (deg)

        Vector ihat( 1., 0., 0. ), jhat( 0., 1., 0. ), khat( 0., 0., 1. ); // basis vectors

        // oblique.cpp: geometry for oblique shots using a right-handed cartesian coordinate system
        // shotline is fixed along the -z axis
        // initial orientation of target is in the x-y plane with center at the origin
        // initial orientation of witness plate is also in the x-y plane, but with center at z = -24 inches
```

Approved for public release; distribution is unlimited.
Vector u_sl = -khat; // shotline fixed along -z axis
Vector n_trgt = -khat; // initial orientation of target
Vector n_wp = -khat; // initial orientation of witness plate

double pitch_t = 0., yaw_t = 0., roll_t = 0., def = 0.5;
if ( argc == 5 ) {
    pitch_t = rad( atof( argv[1] ) ); // pitch (converted to rad) about x-axis
    yaw_t = rad( atof( argv[2] ) ); // yaw (converted to rad) about y-axis
    roll_t = rad( atof( argv[3] ) ); // roll (converted to rad) about z-axis
    def = atof( argv[4] ); // deflection (dimensionless) ranges from 0 to 1
}

Rotation R_trgt( pitch_t, yaw_t, roll_t, XYZ ); // construct the rotation
n_trgt = R_trgt * n_trgt; // apply the rotation to the target

alpha = angle( u_sl, n_trgt ); // compute obliquity of target (rad)

Vector n = slerp( -khat, n_trgt, WP_TRGT_OBL ); // final orientation
Rotation R_wp = Rotation( n_wp, n ); // generate the rotation from the cross product
n_wp = n; // orient the witness plate
Rotation R_wp_inv = -R_wp; // construct inverse rotation of witness plate

sequence s = factor( R_wp, XYZ );
clog << "Witness Plate Orientation (pitch-yaw-roll sequence):" << endl;
clog << "pitch (deg) = " << deg( s.first ) << endl;
clog << "yaw (deg) = " << deg( s.second ) << endl;
clog << "roll (deg) = " << deg( s.third ) << endl;

// output target and witness plate obliquity
beta = def * alpha; // deflection angle (rad)
clog << "target obliquity (deg) = " << deg( alpha ) << endl;
clog << "wp obliquity (deg) = " << deg( angle( u_sl, n_wp ) ) << endl;
clog << "spall cone deflection (deg) = " << deg( beta ) << endl;

Vector rc = -24. * khat; // location of center of witness plate
for ( int i = 0; i < N; i++ ) { // generate frags on the spall cone
    ph = 2. * M_PI * i / double( N );
    u_frag = sin( th ) * cos( ph ) * ihat + sin( th ) * sin( ph ) * jhat + cos( th ) * khat;
    u_frag = R_def * u_frag;
    t = ( rc * n_wp ) / ( u_frag * n_wp );
    r_hit = t * u_frag;
    r = R_wp_inv * ( r_hit - rc );
    cout << r.x() << "\t" << r.y() << endl;
}

return EXIT_SUCCESS;
}

Compiling and running this program with the command
.

./oblique 45. -34.2644 15. 0.5 > output

prints the following back to the screen:

Witness Plate Orientation (pitch-yaw-roll sequence):
pitch (deg) = 20.246
yaw (deg) = -18.4381
roll (deg) = 3.31976
target obliquity (deg) = 54.2403
wp obliquity (deg) = 27.1201
spall cone deflection (deg) = 27.1201
The output file contains the impact points on the witness plate from the for loop (lines 69–79 of oblique.cpp). Fig. 4 shows the resulting plots.

![Graph showing impact points on the witness plate](image)

**Fig. 4.** Plots of the 30° (half-angle) spall cone on the witness plate for a range of deflections. On the left, the deflection parameter is 0, 0.25, 0.5, 0.75, and 1 as the ellipses go from lower left to upper right. On the right is the plot for a deflection of 0.5, which matches the witness plate obliquity and thus results in a circular spall cone. Target pitch, yaw, and roll are fixed at 45°, -34.2644°, and 15°, respectively giving a target obliquity for all these cases of 54.2403°.

### 5. Conclusion

It should be clear from the examples that the Vector and Rotation classes provide robust support for performing 3D vector algebra in C++ programs. To make use of the Vector class, simply include the `Vector.h` class file and to make use of both the Vector and Rotation classes, simply include the `Rotation.h` class file in the C++ program.
6. References


Appendix A. Vector Class
Listing A-1. Vector.h

// Vector.h: Definition & implementation of class for the algebra of 3D vectors
// R. Saucier, February 2000 (Revised June 2016)

#ifndef VECTOR_3D_H
#define VECTOR_3D_H

#include <cstdlib>
#include <cassert>
#include <cmath>
#include <iostream>

namespace va { // vector algebra namespace
enum rep { CART, POLAR }; // vector representation (cartesian or polar)
enum comp { X, Y, Z }; // for referencing cartesian components
const double R2D( 180. / M_PI ); // for converting radians to degrees
const double D2R( M_PI / 180. ); // for converting degrees to radians
inline double deg( const double rad ) { return rad * R2D; } // convert radians to degrees
inline double rad( const double deg ) { return deg * D2R; } // convert degrees to radians
_STATIC_ASSERT( IEEE_754_standad has 53 significant bits or 15 decimal digits of accuracy, so anything smaller is not significant
static const long double TOL = 1.e-15;

class Vector {

friend Vector operator+( const Vector& a, const Vector& b ) { // addition
    return Vector( a._x + b._x, a._y + b._y, a._z + b._z );
}

friend Vector operator-( const Vector& a, const Vector& b ) { // subtraction
    return Vector( a._x - b._x, a._y - b._y, a._z - b._z );
}

friend Vector operator*( const Vector& v, double s ) { // right multiply by scalar
    return Vector( s * v._x, s * v._y, s * v._z );
}

friend Vector operator*( double s, const Vector& v ) { // left multiply by scalar
    return Vector( s * v._x, s * v._y, s * v._z );
}

friend Vector operator/( const Vector& v, double s ) { // division by scalar
    assert( s != 0. );
    return Vector( v._x / s, v._y / s, v._z / s );
}

friend inline double operator*( const Vector& a, const Vector& b ) { // dot product
    double c( a._x * b._x +
             a._y * b._y +
             a._z * b._z );
    if ( fabs(c) < TOL ) c = 0.0L; // set precision to be no more than 15 digits
    return c;
}

friend inline Vector operator^ ( const Vector& a, const Vector& b ) { // cross product
    return Vector( a._y * b._z - a._z * b._y,
                   a._z * b._x - a._x * b._z,
                   a._x * b._y - a._y * b._x );
}

friend double x( const Vector& v ) { // x-coordinate
    return v._x;
}

friend double y( const Vector& v ) { // y-coordinate
    return v._y;
}

};
#endif // VECTOR_3D_H

Approved for public release; distribution is unlimited.
friend double z( const Vector& v ) { // z-coordinate
    return v._z;
}

friend double r( const Vector& v ) { // magnitude of vector
    return v._mag();
}

friend double theta( const Vector& v ) { // polar angle (radians)
    return v.theta();
}

friend double phi( const Vector& v ) { // azimuthal angle (radians)
    return v.phi();
}

friend double norm( const Vector& v ) { // norm or magnitude
    return v._mag();
}

friend double mag( const Vector& v ) { // magnitude
    return v._mag();
}

friend double scalar( const Vector& v ) { // magnitude
    return v._mag();
}

friend double angle( const Vector& a, const Vector& b ) { // angle (radians) between vectors
    double s = a.unit() * b.unit();
    if ( s >= 1. )
        return 0.0;
    else if ( s <= -1. )
        return M_PI;
    else
        return acos( s );
}

friend Vector unit( const Vector& v ) { // unit vector in same direction
    return v.unit();
}

friend Vector normalize( const Vector& v ) { // returns a unit vector in same direction
    return v.unit();
}

friend double dircos( const Vector& v, const comp& i ) { // direction cosine
    return v.dircos( i );
}

friend Vector proj( const Vector& a, const Vector& b ) { // along second vector
    return a.proj( b );
}

// overloaded stream operators

friend std::istream& operator>>( std::istream& is, Vector& v ) { // input vector
    return is >> v._x >> v._y >> v._z;
}

friend std::ostream& operator<< std::ostream& os, const Vector& v ) { // output vector
    Vector a( v );
    a._set_precision();
    return os << a._x << " " << a._y << " " << a._z;
}

public:

Vector( double x, double y, double z, // constructor (cartesian or polar
    rep mode = CART ) { // with cartesian as default)

Approved for public release; distribution is unlimited.
if ( mode == CART ) { // cartesian form
    this->_x = x;
    this->_y = y;
    this->_z = z;
} else if ( mode == POLAR ) // polar form
    _setCartesian( x, y, z );
else {
    std::cerr << "Vector: mode must be either CART or POLAR" << std::endl;
    exit( EXIT_FAILURE );
}

Vector( void ) : _x( 0. ), _y( 0. ), _z( 0. ) { // default constructor
}

~Vector( void ) { // default destructor
}

Vector( const Vector& v ) : _x( v._x ), // copy constructor
    _y( v._y ),
    _z( v._z ) {
}

Vector& operator=( const Vector& v ) { // assignment operator
    if ( this != &v ) {
        _x = v._x;
        _y = v._y;
        _z = v._z;
    }
    return *this;
}

// overloaded arithmetic operators

Vector& operator+=( const Vector& v ) { // addition assignment
    _x += v._x;
    _y += v._y;
    _z += v._z;
    return *this;
}

Vector& operator-=( const Vector& v ) { // subtraction assignment
    _x -= v._x;
    _y -= v._y;
    _z -= v._z;
    return *this;
}

Vector& operator*=( double s ) { // multiplication assignment
    _x *= s;
    _y *= s;
    _z *= s;
    return *this;
}

Vector& operator/=( double s ) { // division assignment
    assert( s != 0. );
    _x /= s;
    _y /= s;
    _z /= s;
    return *this;
}

Vector operator-( void ) { // negative of a vector
    return Vector( -_x, -_y, -_z );
}

const double& operator[]( comp i ) const { // index operator (component)
    if ( i == X ) return _x;
    if ( i == Y ) return _y;
    if ( i == Z ) return _z;
    std::cerr << "Vector: Array index out of range; must be X, Y, or Z" << std::endl;
    exit( EXIT_FAILURE );
}

// access functions
double x( void ) const { // x-component
    return _x;
}

double y( void ) const { // y-component
    return _y;
}

double z( void ) const { // z-component
    return _z;
}

double r( void ) const { // magnitude
    return _mag();
}

double theta( void ) const { // polar angle (radians)
    return _theta();
}

double phi( void ) const { // azimuthal angle (radians)
    return _phi();
}

double norm( void ) const { // norm or magnitude
    return _mag();
}

double mag( void ) const { // magnitude
    return _mag();
}

operator double( void ) const { // conversion operator to return magnitude
    return _mag();
}

Vector unit( void ) const { // returns a unit vector
    double m = _mag();
    if ( m > 0. ) return Vector( _x / m, _y / m, _z / m );
    std::cerr << "Vector: Cannot make a unit vector from a null vector" << std::endl;
    exit( EXIT_FAILURE );
}

Vector normalize( void ) const { // synonym for unit
    return unit();
}

double dircos( const comp& i ) const { // direction cosine
    if ( i == X ) return _x / _mag();
    if ( i == Y ) return _y / _mag();
    if ( i == Z ) return _z / _mag();
    std::cerr << "Vector dircos: comp out of range; must be X, Y, or Z" << std::endl;
    exit( EXIT_FAILURE );
}

Vector proj( const Vector& e ) const { // projects onto the given vector
    Vector u = e.unit();
    return u * ( *this * u );
}

Vector& rotate( const Vector& a, double angle ) { // rotates vector about given axial vector, a, through the given angle
    Vector a_hat = a.unit(); // unit vector along the axial vector
    Vector cross = a_hat ^ *this; // cross product of a hat with the given vector
    return *this += sin( angle ) * cross + ( 1. - cos( angle ) ) * ( a_hat ^ cross );
}

Vector& rot( const Vector& a, double angle ) { // synonym for rotate

Approved for public release; distribution is unlimited.

25
Notice that the class is enclosed in a va namespace, so that Vectors are declared by va::Vector. Table A-1 provides a reference sheet for basic usage.

Approved for public release; distribution is unlimited.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Mathematical notation</th>
<th>Vector class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition</td>
<td>Let ( v ) be an unspecified vector.</td>
<td>Vector ( v );</td>
</tr>
<tr>
<td></td>
<td>Let ( a ) be the cartesian vector ((1,2,3)).</td>
<td>Vector ( a(1,2,3); ) or</td>
</tr>
<tr>
<td></td>
<td>Let ( b ) be the polar vector ((r,\theta,\phi)).</td>
<td>Vector ( b(r,th,ph,POLAR); );</td>
</tr>
<tr>
<td>Input vector ( a )</td>
<td>( n/a )</td>
<td>cin &gt;&gt; ( a );</td>
</tr>
<tr>
<td>Output vector ( a )</td>
<td>( n/a )</td>
<td>cout &lt;&lt; ( a );</td>
</tr>
<tr>
<td>Cartesian representation</td>
<td>Let ( a = x\hat{\text{i}} + y\hat{\text{j}} + z\hat{\text{k}} ).</td>
<td>Vector ( a(x,y,z); ) or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vector ( a(x,y,z,CART); );</td>
</tr>
<tr>
<td>Polar representation</td>
<td>Let ( a = (r,\theta,\phi) ).</td>
<td>Vector ( a(r,th,ph,POLAR); );</td>
</tr>
<tr>
<td>Assign one vector to another</td>
<td>Let ( b = a ) or ( b \leftarrow a )</td>
<td>( b = a; ) or  ( b(a); )</td>
</tr>
<tr>
<td>Components of vector ( a )</td>
<td>( a_x, a_y, a_z )</td>
<td>( a.x(), a.y(), a.z() ) or</td>
</tr>
<tr>
<td></td>
<td>( r, \theta, \phi )</td>
<td>( x(a), y(a), z(a) ) or ( a[x], \ a[y], \ a[z] )</td>
</tr>
<tr>
<td>Direction cosines</td>
<td>( v \cdot \hat{\text{i}}/|v|, v \cdot \hat{\text{j}}/|v|, v \cdot \hat{\text{k}}/|v| )</td>
<td>( v.dircos(X); ) ... or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{dircos}(v,X); ) ...</td>
</tr>
<tr>
<td>Vector addition</td>
<td>( c = a + b )</td>
<td>( c = a + b; )</td>
</tr>
<tr>
<td>Addition assignment</td>
<td>( b \leftarrow b + a )</td>
<td>( b += a; )</td>
</tr>
<tr>
<td>Vector subtraction</td>
<td>( c = a - b )</td>
<td>( c = a - b; )</td>
</tr>
<tr>
<td>Subtraction assignment</td>
<td>( b \leftarrow b - a )</td>
<td>( b -= a; )</td>
</tr>
<tr>
<td>Multiplication by a scalar ( s )</td>
<td>( b = s \cdot a ) or ( b = a \cdot s )</td>
<td>( b = s * a; ) or ( b = a * s; )</td>
</tr>
<tr>
<td>Multiplication assignment</td>
<td>( a \leftarrow s \cdot a ) or ( a \leftarrow a \cdot s )</td>
<td>( a *= s; )</td>
</tr>
<tr>
<td>Dot (scalar) product</td>
<td>( c = a \cdot b )</td>
<td>( c = a \cdot b; )</td>
</tr>
<tr>
<td>Cross (vector) product</td>
<td>( c = a \times b )</td>
<td>( c = a \wedge b; )</td>
</tr>
<tr>
<td>Negative of a vector</td>
<td>( -v )</td>
<td>( -v; )</td>
</tr>
<tr>
<td>Norm, or magnitude, of a vector</td>
<td>( |v| )</td>
<td>( v.norm(); ) or ( norm(v); ) or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( v.mag(); ) or ( mag(v); ) or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( v.r(); ) or ( v.scalar(); )</td>
</tr>
<tr>
<td>Angle between two vectors</td>
<td>( \theta = \cos^{-1} \left( \frac{a \cdot b}{|a| |b|} \right) )</td>
<td>( \text{angle}(a, b); )</td>
</tr>
</tbody>
</table>
| Normalize a vector                | \( \hat{u} = v/\|v\| \)            | \( u = v.normalize(); \) or \( u = \text{normalize}(v); \) or |}
|                                   |                                    | \( u = v.unit(); \) or \( u = \text{unit}(v); \)          |
| Projection of \( a \) along \( b \)| \( \left( \frac{a \cdot b}{\|b\|} \right) \frac{b}{\|b\|} \) | \( \text{proj}(a, b); \) or \( a.proj(b); \)
| Rotate vector \( a \) about the   | \( a + \hat{u} \times a \sin \theta + \hat{u} \times (\hat{u} \times a)(1 - \cos \theta) \) | \( \text{a.rotate}(u, \text{theta}); \) or \( \text{a.rot}(u, \text{theta}); \) |

\(^a\) \( r \) is the magnitude, \( \theta \) is the polar angle measured from the \( z \)-axis, and \( \phi \) is the azimuthal angle measured from the \( x \)-axis to the plane that contains the vector and the \( z \)-axis. The angle \( \theta \) and \( \phi \) are in radians. Use \texttt{rad(deg)} to convert degrees to radians and \texttt{deg(rad)} to convert radians to degrees.

\(^b\) \( \hat{i}, \hat{j}, \text{and} \hat{k} \) are unit vectors along the \( x \)-axis, \( y \)-axis, and \( z \)-axis, respectively.

\(^c\) \texttt{normalize} does not change the vector it is invoked on; it merely returns the vector divided by its norm.
Intentionally left blank.
Appendix B. Rotation Class
Listing B-1. Rotation.h

```c++
// Rotation.h: Rotation class definition for the algebra of 3D rotations
//      Altman, S. L., Rotations, Quaternions, and Double Groups, 1986.
//      R. Saucier, March 2005 (last revised June 2016)

#ifndef ROTATION_H
#define ROTATION_H

#include "Vector.h"
#include "Random.h"
#include <iostream>

namespace va { // vector algebra namespace
    const Vector DEFAULT_UNIT_VECTOR( 0., 0., 1. ); // arbitrarily choose k
    const double DEFAULT_ROTATION_ANGLE( 0. ); // arbitrarily choose 0
    const double TWO_PI( 2. * M_PI );
    
    enum ORDER { // order of rotation sequence about body axes
        ZYX, // first about z-axis, second about y-axis and third about x-axis
        XYZ, // first about x-axis, second about y-axis and third about z-axis
        YXZ, // first about y-axis, second about x-axis and third about z-axis
        ZXY, // first about z-axis, second about x-axis and third about y-axis
        XZY, // first about x-axis, second about z-axis and third about y-axis
        YZX, // first about y-axis, second about z-axis and third about x-axis
    };

    struct quaternion { // q = w + v, where w is scalar part and v is vector part
        quaternion( void ) { } 
        quaternion( double scalar, Vector vector ) : w( scalar ), v( vector ) { } 
        quaternion( void ) { }
        friend quaternion operator*( const quaternion& q1, const quaternion& q2 ) { 
            double p0 = q1.w * q2.w; // scalar part
            double p1 = x( q1.v ) * q2.w + q1.w * x( q2.v ); // vector part
            double p2 = y( q1.v ) * q2.w + q1.w * y( q2.v );
            double p3 = z( q1.v ) * q2.w + q1.w * z( q2.v );
            return quaternion( p0 - x( p1 ) - ( q1.v * q2.v ), // scalar part
                p1 + q1.w * q2.v + ( q1.v * q2.v ), // vector part
                p2, p3 );
        }
        
        double w; // scalar part
        Vector v; // vector part (actually a bivector disguised as a vector)
    };

    struct sequence { // rotation sequence about three principal body axes
        sequence( void ) { }
        sequence( double phi_1, double phi_2, double phi_3 ) : first( phi_1 ), second( phi_2 ), third( phi_3 ) { }
        
        -sequence( void ) { }
        
        // factor quaternion into Euler rotation sequence about three distinct principal axes
        sequence factor( const quaternion& q, ORDER order ) {
            double p0 = q.w;
            double p1 = x( q.v );
            double p2 = y( q.v );
            double p3 = z( q.v );
            if ( order == ZYX ) { // distinct principal axes ZYX
```
double A = p0 * p1 + p2 * p3;
double B = ( p2 - p0 ) * ( p2 + p0 );
double D = ( p1 - p3 ) * ( p1 + p3 );

phi_3 = atan( - 2. * A / ( B + D ) );
double c0 = cos( 0.5 * phi_3 );
double c1 = sin( 0.5 * phi_3 );
double q0 = p0 * c0 + p1 * c1;
//double q1 = p1 * c0 - p0 * c1;
double q2 = p2 * c0 - p3 * c1;
phi_1 = 2. * atan( q3 / q0 );
phi_2 = 2. * atan( q2 / q0 );
}
else if ( order == XYZ ) { // distinct principal axes xyz
double A = p1 * p2 - p0 * p3;
double B = ( p1 - p3 ) * ( p1 + p3 );
double D = ( p0 - p2 ) * ( p0 + p2 );

phi_3 = atan( - 2. * A / ( B + D ) );
double c0 = cos( 0.5 * phi_3 );
double c3 = sin( 0.5 * phi_3 );
double q0 = p0 * c0 + p3 * c3;
double q1 = p1 * c0 - p2 * c3;
double q2 = p2 * c0 + p1 * c3;
//double q3 = p3 * c0 - p0 * c3;
phi_1 = 2. * atan( q1 / q0 );
phi_2 = 2. * atan( q2 / q0 );
}
else if ( order == YXZ ) { // distinct principal axes yxz
double A = p1 * p2 + p0 * p3;
double B = p1 * p1 + p3 * p3;
double D = -( p0 * p0 + p2 * p2 );

phi_3 = atan( - 2. * A / ( B + D ) );
double c0 = cos( 0.5 * phi_3 );
double c3 = sin( 0.5 * phi_3 );
double q0 = p0 * c0 + p3 * c3;
double q1 = p1 * c0 - p2 * c3;
double q2 = p2 * c0 + p1 * c3;
//double q3 = p3 * c0 - p0 * c3;
phi_1 = 2. * atan( q2 / q0 );
phi_2 = 2. * atan( q1 / q0 );
}
else if ( order == ZXY ) { // distinct principal axes zxy
double A = p1 * p3 - p0 * p2;
double B = ( p0 - p1 ) * ( p0 + p1 );
double D = ( p3 - p2 ) * ( p3 + p2 );

phi_3 = atan( - 2. * A / ( B + D ) );
double c0 = cos( 0.5 * phi_3 );
double c2 = sin( 0.5 * phi_3 );
double q0 = p0 * c0 + p2 * c2;
double q1 = p1 * c0 + p3 * c2;
//double q2 = p2 * c0 + p3 * c2;
phi_1 = 2. * atan( q3 / q0 );
phi_2 = 2. * atan( q1 / q0 );
}
else if ( order == XZY ) { // distinct principal axes xzy
double A = p1 * p3 + p0 * p2;
double B = -( p0 * p0 + p1 * p1 );
double D = p2 * p2 + p3 * p3;

phi_3 = atan( - 2. * A / ( B + D ) );
double c0 = cos( 0.5 * phi_3 );
double c2 = sin( 0.5 * phi_3 );
double q0 = p0 * c0 + p2 * c2;
double q1 = p1 * c0 + p3 * c2;
//double q2 = p2 * c0 - p3 * c2;
phi_1 = 2. * atan( q3 / q0 );
phi_2 = 2. * atan( q1 / q0 );
}
phi_1 = 2. * atan( q1 / q0 );
phi_2 = 2. * atan( q3 / q0 );

} else if ( order == YZX ) { // distinct principal axes yzx

double A = p2 * p3 - p0 * p1;
double B = p0 * p0 + p2 * p2;
double D = -( p1 * p1 + p3 * p3 );
phi_3 = atan( - 2. * A / ( B + D ) );
double c0 = cos( 0.5 * phi_3 );
double c1 = sin( 0.5 * phi_3 );
double q0 = p0 * c0 + p1 * c1;
//double q1 = p1 * c0 - p0 * c1;
double q2 = p2 * c0 - p3 * c1;
double q3 = p3 * c0 + p2 * c1;
phi_1 = 2. * atan( q2 / q0 );
phi_2 = 2. * atan( q3 / q0 );
}

} else if ( order == ZYZ ) { // repeated principal axes yzy

double A = p0 * p1 + p2 * p3;
double B = -2. * p0 * p2;
double D = 2. * p1 * p3;
phi_3 = atan( -2. * A / ( B + D ) );
double c0 = cos( 0.5 * phi_3 );
double c3 = sin( 0.5 * phi_3 );
double q0 = p0 * c0 + p3 * c3;
//double q1 = p1 * c0 + p2 * c3;
double q2 = p2 * c0 + p1 * c3;
double q3 = p3 * c0 - p0 * c3;
phi_1 = 2. * atan( q3 / q0 );
phi_2 = 2. * atan( q2 / q0 );
}

} else if ( order == ZXZ ) { // repeated principal axes zxz

double A = p0 * p2 - p1 * p3;
double B = 2. * p0 * p1;
double D = 2. * p2 * p3;
phi_3 = atan( -2. * A / ( B + D ) );
double c0 = cos( 0.5 * phi_3 );
double c3 = sin( 0.5 * phi_3 );
double q0 = p0 * c0 + p3 * c3;
//double q1 = p1 * c0 - p2 * c3;
double q2 = p2 * c0 + p1 * c3;
double q3 = p3 * c0 - p0 * c3;
phi_1 = 2. * atan( q3 / q0 );
phi_2 = 2. * atan( q1 / q0 );
}

} else if ( order == YXY ) { // repeated principal axes yxy

double A = p0 * p3 + p1 * p2;
double B = -2. * p0 * p1;
double D = 2. * p2 * p3;
phi_3 = atan( -2. * A / ( B + D ) );
double c0 = cos( 0.5 * phi_3 );
double c2 = sin( 0.5 * phi_3 );
double q0 = p0 * c0 + p2 * c2;
//double q1 = p1 * c0 + p3 * c2;
double q2 = p2 * c0 - p0 * c2;
double q3 = p3 * c0 - p1 * c2;
phi_1 = 2. * atan( q2 / q0 );
phi_2 = 2. * atan( q3 / q0 );
}
double q0 = p0 * c0 + p2 * c2;
double q1 = p1 * c0 + p3 * c2;
double q2 = p2 * c0 - p0 * c2;
  //double q3 = p3 * c0 - p1 * c2;

phi_1 = 2. * atan( q2 / q0 );
phi_2 = 2. * atan( q1 / q0 );

else if ( order == XXY ) { // repeated principal axes xxy

double A = p0 * p3 - p1 * p2;
double B = p0 * p2 + p1 * p3;

phi_3 = atan( -A / B );
double c0 = cos( 0.5 * phi_3 );
double c1 = sin( 0.5 * phi_3 );

double q0 = p0 * c0 + p1 * c1;
double q1 = p1 * c0 - p0 * c1;
  //double q2 = p2 * c0 - p3 * c1;

phi_1 = 2. * atan( q1 / q0 );
phi_2 = 2. * atan( q2 / q0 );

} else if ( order == XZX ) { // repeated principal axes xzx

double A = p0 * p2 + p1 * p3;
double B = -p0 * p3 + p1 * p2;

phi_3 = atan( -A / B );
double c0 = cos( 0.5 * phi_3 );
double c1 = sin( 0.5 * phi_3 );

double q0 = p0 * c0 + p1 * c1;
double q1 = p1 * c0 - p0 * c1;
  //double q2 = p2 * c0 - p3 * c1;

double q3 = p3 * c0 + p2 * c1;

phi_1 = 2. * atan( q1 / q0 );
phi_2 = 2. * atan( q3 / q0 );

} else {
  std::cerr << "ERROR in Rotation: invalid sequence order: " << order << std::endl;
  exit( EXIT_FAILURE );
}

return sequence( phi_1, phi_2, phi_3 );

}; // end struct sequence

struct matrix { // all matrices here are rotations in three-space

  // overloaded multiplication of two matrices
  // (defined as a convenience to the user; not used in Rotation class)

friend matrix operator*( const matrix& A, const matrix& B ) {

  matrix C;
  C.a11 = A.a11 * B.a11 + A.a12 * B.a21 + A.a13 * B.a31;
  C.a12 = A.a11 * B.a12 + A.a12 * B.a22 + A.a13 * B.a32;
  C.a13 = A.a11 * B.a13 + A.a12 * B.a23 + A.a13 * B.a33;
  C.a21 = A.a21 * B.a11 + A.a22 * B.a21 + A.a23 * B.a31;
  C.a22 = A.a21 * B.a12 + A.a22 * B.a22 + A.a23 * B.a32;
  C.a23 = A.a21 * B.a13 + A.a22 * B.a23 + A.a23 * B.a33;
  C.a31 = A.a31 * B.a11 + A.a32 * B.a21 + A.a33 * B.a31;
  C.a32 = A.a31 * B.a12 + A.a32 * B.a22 + A.a33 * B.a32;
  C.a33 = A.a31 * B.a13 + A.a32 * B.a23 + A.a33 * B.a33;

  return C;
}

}; // end struct matrix

// transpose of a matrix
// (defined as a convenience to the user; not used in Rotation class)

friend matrix transpose( const matrix& A ) {

  matrix B;
  B.a11 = A.a11;
  B.a12 = A.a21;
  B.a13 = A.a31;

  B.a21 = A.a12;
  B.a22 = A.a22;
  B.a23 = A.a32;

  B.a31 = A.a13;
  B.a32 = A.a32;
  B.a33 = A.a33;

  return B;
}
339 B.a21 = A.a12;
340 B.a22 = A.a22;
341 B.a23 = A.a32;
342 B.a31 = A.a13;
343 B.a32 = A.a23;
344 B.a33 = A.a33;
345 return B;
346 }
347 }
348 }
349 // inverse of a matrix
350 // (defined as a convenience to the user; not used in Rotation class)
351 friend matrix inverse( const matrix& A ) {
352 double det = A.a11 * ( A.a22 * A.a33 - A.a23 * A.a32 ) +
353 A.a12 * ( A.a23 * A.a31 - A.a21 * A.a33 ) +
354 A.a13 * ( A.a21 * A.a32 - A.a22 * A.a31 );
355 assert( det != 0. );
356 matrix B;
357 B.a11 = +( A.a22 * A.a33 - A.a23 * A.a32 ) / det;
358 B.a12 = -( A.a12 * A.a33 - A.a13 * A.a32 ) / det;
359 B.a13 = +( A.a12 * A.a23 - A.a13 * A.a22 ) / det;
360 B.a21 = -( A.a21 * A.a33 - A.a23 * A.a31 ) / det;
361 B.a22 = +( A.a11 * A.a33 - A.a13 * A.a31 ) / det;
362 B.a23 = -( A.a11 * A.a23 - A.a13 * A.a21 ) / det;
363 B.a31 = +( A.a21 * A.a32 - A.a22 * A.a31 ) / det;
364 B.a32 = -( A.a11 * A.a32 - A.a12 * A.a31 ) / det;
365 B.a33 = +( A.a11 * A.a22 - A.a12 * A.a21 ) / det;
366 return B;
367 }
368 }
369 // returns the matrix with no more than 15 decimal digit accuracy
370 friend matrix set_precision( const matrix& A ) {
371 matrix B( A );
372 if ( fabs(B.a11) < TOL ) B.a11 = 0.0L;
373 if ( fabs(B.a12) < TOL ) B.a12 = 0.0L;
374 if ( fabs(B.a13) < TOL ) B.a13 = 0.0L;
375 if ( fabs(B.a21) < TOL ) B.a21 = 0.0L;
376 if ( fabs(B.a22) < TOL ) B.a22 = 0.0L;
377 if ( fabs(B.a23) < TOL ) B.a23 = 0.0L;
378 if ( fabs(B.a31) < TOL ) B.a31 = 0.0L;
379 if ( fabs(B.a32) < TOL ) B.a32 = 0.0L;
380 if ( fabs(B.a33) < TOL ) B.a33 = 0.0L;
381 return B;
382 }
383 }
384 // convenient matrix properties, but not essential to the Rotation class
385 friend double tr( const matrix& A ) { // trace of a matrix
386 return A.a11 + A.a22 + A.a33;
387 }
388 }
389 friend double det( const matrix& A ) { // determinant of a matrix
390 return A.a11 * ( A.a22 * A.a33 - A.a23 * A.a32 ) +
391 A.a12 * ( A.a23 * A.a31 - A.a21 * A.a33 ) +
392 A.a13 * ( A.a21 * A.a32 - A.a22 * A.a31 );
393 }
394 friend Vector eigenvector( const matrix& A ) { // eigenvector of a matrix
395 return unit( ( A.a32 - A.a23 ) * Vector( 1., 0., 0. ) +
396 ( A.a13 - A.a31 ) * Vector( 0., 1., 0. ) +
397 ( A.a21 - A.a12 ) * Vector( 0., 0., 1. ) );
398 }
399 friend double angle( const matrix& A ) { // angle of rotation
400 return acos( 0.5 * ( A.a11 + A.a22 + A.a33 - 1. ) );
401 }
402 friend Vector eigenvector( void ) { // axis of rotation
403 return unit( ( A.a32 - A.a23 ) * Vector( 1., 0., 0. ) +
404 ( A.a13 - A.a31 ) * Vector( 0., 1., 0. ) +
405 ( A.a21 - A.a12 ) * Vector( 0., 0., 1. ) );
406 Approved for public release; distribution is unlimited.
34
double tr( void ) { // trace of the matrix
    return a11 + a22 + a33;
}

double angle( void ) { // angle of rotation
    return acos( 0.5 * ( a11 + a22 + a33 - 1. ) );
}

double a11, a12, a13, // 1st row
    a21, a22, a23, // 2nd row
    a31, a32, a33; // 3rd row
}; // end struct matrix

global Rotation {
    // friends list
}

// overloaded multiplication of two successive rotations (using quaternions)
friend Rotation operator*( const Rotation& R1, const Rotation& R2 ) {
    return Rotation( to_quaternion( R1 ) * to_quaternion( R2 ) );
}

// rotation of a vector
// overloaded multiplication of a vector by a rotation (using quaternions)
friend Vector operator*( const Rotation& R, const Vector& a ) {
    quaternion q{ to_quaternion( R ) };
    double w{ q.w };
    Vector v{ q.v };
    Vector b = 2. * ( v ^ a );
    return a + ( w * b ) + ( v ^ b );
}

// spherical linear interpolation on the unit sphere from u1 to u2
friend Vector slerp( const Vector& u1, const Vector& u2, double theta, double t ) {
    assert( theta != 0 );
    assert( 0. <= t && t <= 1. );
    return ( sin( ( 1. - t ) * theta ) * u1 + sin( t * theta ) * u2 ) / sin( theta );
}

// spherical linear interpolation on the unit sphere from u1 to u2
friend Vector slerp( const Vector& u1, const Vector& u2, double t ) {
    double theta = angle( u1, u2 );
    if ( theta == 0. ) return u1;
    assert( 0. <= t && t <= 1. );
    return ( sin( 1. - t ) * theta ) * u1 + sin( t * theta ) * u2 ) / sin( theta );
}

// access functions
friend Vector vec( const Rotation& R ) { // return axial unit eigenvector
    return R.vec;
}

friend double ang( const Rotation& R ) { // return rotation angle (rad)
    return R._ang;
}

friend Rotation inverse( Rotation R ) {
    return Rotation( R._vec, -R._ang );
}

friend quaternion to_quaternion( const Rotation& R ) {
    double a = 0.5 * R._ang;
    Vector u = R._vec;
    return quaternion( cos( a ), u + sin( a ) );
}
// conversion to rotation matrix
friend matrix to_matrix( const Rotation& R ) {
    quaternion q = to_quaternion( R );
    double w = q.w;
    Vector v = q.v;
    double v1 = v[ X ], v2 = v[ Y ], v3 = v[ Z ];

    matrix A;
    A.a11 = 2. * ( w * w - 0.5 + v1 * v1 ); // 1st row, 1st col
    A.a12 = 2. * ( v1 * v2 - w * v3 ); // 1st row, 2nd col
    A.a13 = 2. * ( v1 * v3 + w * v2 ); // 1st row, 3rd col

    A.a21 = 2. * ( v1 * v2 + w * v3 ); // 2nd row, 1st col
    A.a22 = 2. * ( w * w - 0.5 + v2 * v2 ); // 2nd row, 2nd col
    A.a23 = 2. * ( v2 * v3 - w * v1 ); // 2nd row, 3rd col

    A.a31 = 2. * ( v1 * v3 - w * v2 ); // 3rd row, 1st col
    A.a32 = 2. * ( v2 * v3 + w * v1 ); // 3rd row, 2nd col
    A.a33 = 2. * ( w * w - 0.5 + v3 * v3 ); // 3rd row, 3rd col

    return A;
}

// factor rotation into a rotation sequence
friend sequence factor( const Rotation& R, ORDER order ) {
    sequence s;
    return s.factor( to_quaternion( R ), order );
}

// factor matrix representation of rotation into a rotation sequence
friend sequence factor( const matrix& A, ORDER order ) {
    sequence s;
    return s.factor( to_quaternion( Rotation( A ) ), order );
}

// overloaded stream operators

friend std::istream& operator>>( std::istream& is, Rotation& R ) {
    std::cout << "Specify axis of rotation by entering an axial vector (need not be a unit vector)" << std::endl;
    is >> R._vec;
    std::cout << "Enter the angle of rotation (deg): ";
    is >> R._ang;
    R._vec = unit( R._vec ); // store the unit vector representing the axis
    R._ang = R._ang * D2R; // store the rotation angle in radians
    return is;
}

friend std::ostream& operator<<( std::ostream& os, const Rotation& R ) {
    return os << R._vec << "\t" << R._ang * R2D;
}

friend std::ostream& operator<<( std::ostream& os, const quaternion& q ) {
    return os << q.w << "\t" << q.v;
}

friend std::ostream& operator<<( std::ostream& os, const matrix& A ) {
    matrix B = set_precision( A ); // no more than 15 decimal digits of accuracy
    return os << B.a11 << "\t" << B.a12 << "\t" << B.a13 << std::endl
              << B.a21 << "\t" << B.a22 << "\t" << B.a23 << std::endl
              << B.a31 << "\t" << B.a32 << "\t" << B.a33;
}

public:

// constructor from three angles (rad), phi_1, phi_2, phi_3 (in that order, left to right)
// about three distinct principal body axes
Rotation( double phi_1, double phi_2, double phi_3, ORDER order ) {
    double ang_1 = 0.5 * phi_1, c1 = cos( ang_1 ), s1 = sin( ang_1 );
    double ang_2 = 0.5 * phi_2, c2 = cos( ang_2 ), s2 = sin( ang_2 );
    double ang_3 = 0.5 * phi_3, c3 = cos( ang_3 ), s3 = sin( ang_3 );
    double w;
}
Vector v;

if ( order == ZYX ) { // 1st about z-axis, 2nd about y-axis, 3rd about x-axis (Aerospace sequence)
    w = c1 * c2 * c3 + s1 * s2 * s3;
    v = Vector( c1 * c2 * s3 - s1 * s2 * c3,
                c1 * s2 * c3 + s1 * c2 * s3,
                c1 * s2 * s3 - s1 * c2 * c3 );
} else if ( order == XYZ ) { // 1st about x-axis, 2nd about y-axis, 3rd about z-axis (FATEPEN sequence)
    w = c1 * c2 * c3 - s1 * s2 * s3;
    v = Vector( c1 * s2 * s3 + s1 * c2 * c3,
                c1 * s2 * c3 - s1 * c2 * s3,
                c1 * c2 * s3 + s1 * s2 * c3 );
} else if ( order == YXZ ) { // 1st about y-axis, 2nd about x-axis, 3rd about z-axis
    w = c1 * c2 * c3 + s1 * s2 * s3;
    v = Vector( c1 * s2 * c3 + s1 * c2 * s3,
                c1 * s2 * s3 + s1 * c2 * c3,
                c1 * c2 * s3 - s1 * s2 * c3 );
} else if ( order == ZXY ) { // 1st about z-axis, 2nd about x-axis, 3rd about y-axis
    w = c1 * c2 * c3 - s1 * s2 * s3;
    v = Vector( c1 * s2 * s3 - s1 * c2 * c3,
                c1 * s2 * c3 - s1 * c2 * s3,
                c1 * c2 * s3 + s1 * s2 * c3 );
} else if ( order == XZY ) { // 1st about x-axis, 2nd about z-axis, 3rd about y-axis
    w = c1 * c2 * c3 + s1 * s2 * s3;
    v = Vector( -c1 * s2 * s3 + s1 * c2 * c3,
                c1 * c2 * s3 - s1 * s2 * c3,
                c1 * s2 * c3 + s1 * s2 * s3 );
} else if ( order == YZX ) { // 1st about y-axis, 2nd about z-axis, 3rd about x-axis
    w = c1 * c2 * c3 - s1 * s2 * s3;
    v = Vector( c1 * c2 * s3 + s1 * c2 * c3,
                c1 * s2 * s3 + s1 * c2 * c3,
                c1 * s2 * c3 - s1 * c2 * s3 );
} else if ( order == ZYZ ) { // Euler sequence, 1st about z-axis, 2nd about y-axis, 3rd about z-axis
    w = c1 * c2 * c3 - s1 * c2 * s3;
    v = Vector( c1 * s2 * s3 - s1 * s2 * c3,
                c1 * s2 * c3 + s1 * s2 * s3,
                c1 * c2 * s3 + s1 * c2 * c3 );
} else if ( order == ZXZ ) { // Euler sequence, 1st about z-axis, 2nd about x-axis, 3rd about z-axis
    w = c1 * c2 * c3 - s1 * c2 * s3;
    v = Vector( c1 * c2 * s3 + s1 * c2 * c3,
                c1 * s2 * s3 - s1 * s2 * c3,
                c1 * s2 * c3 + s1 * s2 * s3 );
} else if ( order == YZY ) { // Euler sequence, 1st about y-axis, 2nd about z-axis, 3rd about y-axis
    w = c1 * c2 * c3 - s1 * c2 * s3;
    v = Vector( -c1 * s2 * s3 + s1 * c2 * c3,
                c1 * c2 * s3 - s1 * s2 * c3,
                c1 * s2 * c3 + s1 * s2 * s3 );
} else if ( order == YXY ) { // Euler sequence, 1st about y-axis, 2nd about x-axis, 3rd about y-axis
    w = c1 * c2 * c3 - s1 * c2 * s3;
    v = Vector( c1 * s2 * s3 + s1 * c2 * c3,
                c1 * c2 * s3 + s1 * c2 * c3,
                c1 * s2 * c3 - s1 * c2 * s3 );
} else if ( order == XYX ) { // Euler sequence, 1st about x-axis, 2nd about y-axis, 3rd about x-axis
    w = c1 * c2 * c3 - s1 * c2 * s3;
    v = Vector( c1 * c2 * s3 + s1 * s2 * c3,
                c1 * s2 * s3 + s1 * s2 * c3,
                c1 * s2 * c3 - s1 * s2 * s3 );
} else if ( order == XZX ) { // Euler sequence, 1st about x-axis, 2nd about z-axis, 3rd about x-axis
    w = c1 * c2 * c3 - s1 * c2 * s3;
    v = Vector( c1 * c2 * s3 + s1 * s2 * c3,
                c1 * s2 * s3 - s1 * s2 * c3,
                c1 * s2 * c3 + s1 * s2 * s3 );
679  else {
680     std::cerr << "ERROR in Rotation: invalid order: " << order << std::endl;
681     exit( EXIT_FAILURE );
682  }
683  }
684  if ( w >= 1. || v == 0. ) {
685      _ang = DEFAULT_ROTATION_ANGLE;
686      _vec = DEFAULT_UNIT_VECTOR;
687  } else {
688      _ang = 2. * acos( w );
689      _vec = v / sqrt( 1. - w * w );
690  }
691  _set_angle(); // angle in the range [-M_PI, M_PI]
692
693  // constructor from a rotation sequence
694  Rotation( const sequence& s, ORDER order ) {
695      Rotation R( s.first, s.second, s.third, order );
696      _vec = vec( R ); // set the axial vector
697      _ang = ang( R ); // set the rotation angle
698      _set_angle(); // angle in the range [-M_PI, M_PI]
699  }
700
701  // constructor from an axis vector and rotation angle (rad)
702  Rotation( const Vector& v, double a ) : _vec( v ), _ang( a ) {
703      _vec = _vec.unit(); // store the unit vector representing the axis
704      _set_angle(); // angle in the range [-M_PI, M_PI]
705  }
706
707  // constructor using sphericalCoord (of axial vector) and rotation angle (rad)
708  Rotation( rng::sphericalCoord s, double ang ) {
709      _vec = Vector( 1., s.theta, s.phi, POLAR ); // unit vector
710      _ang = ang;
711      _set_angle(); // angle in the range [-M_PI, M_PI]
712  }
713
714  // constructor from the cross product of two vectors
715  // generate the rotation that, when applied to vector a, will result in vector b
716  Rotation( const Vector& a, const Vector& b ) {
717      _vec = unit( a ^ b ) + Vector( 1., 0., 0. ) + Vector( 0., 1., 0. ) + Vector( 0., 0., 1. );
718      if ( _vec == 0. ) { // then it must be the identity matrix
719          _ang = M_PI;
720      } else {
721          _ang = acos( s );
722          _set_angle(); // angle in the range [-M_PI, M_PI]
723  }
724
725  // constructor from unit quaternion
726  Rotation( const quaternion& q ) {
727      double w = q.w;
728      Vector v = q;v;
729  }
730  if ( w >= 1. || w == 0. ) {
731      _ang = DEFAULT_ROTATION_ANGLE;
732      _vec = DEFAULT_UNIT_VECTOR;
733  } else {
734      double n = sqrt( w * w + v * v ) ; // need to insure it’s a unit quaternion
735      w /= n;
736      v /= n;
737      _ang = 2. * acos( w );
738      _vec = v / sqrt( 1. - w * w );
739  }
740  _set_angle(); // angle in the range [-M_PI, M_PI]
741  }
742
743  // constructor from rotation matrix
744  Rotation( const matrix& A ) {
745      _vec = ( A.a32 - A.a23 ) * Vector( 1., 0., 0. ) +
746              ( A.a31 - A.a13 ) * Vector( 0., 1., 0. ) +
747              ( A.a21 - A.a12 ) * Vector( 0., 0., 1. );
748  }
749  if ( _vec == 0. ) { // than it must be the identity matrix
750
751  // constructor from a rotation sequence
752  Rotation( const sequence& s, ORDER order ) {
753      Rotation R( s.first, s.second, s.third, order );
754      _vec = vec( R ); // set the axial vector
755      _ang = ang( R ); // set the rotation angle
756      _set_angle(); // angle in the range [-M_PI, M_PI]
757  }
758
759  // constructor from an axis vector and rotation angle (rad)
760  Rotation( const Vector& v, double a ) : _vec( v ), _ang( a ) {
761      _vec = _vec.unit(); // store the unit vector representing the axis
762      _set_angle(); // angle in the range [-M_PI, M_PI]
763  }
764
765  // constructor using sphericalCoord (of axial vector) and rotation angle (rad)
766  Rotation( rng::sphericalCoord s, double ang ) {
767      _vec = Vector( 1., s.theta, s.phi, POLAR ); // unit vector
768      _ang = ang;
769      _set_angle(); // angle in the range [-M_PI, M_PI]
770  }
771
772  // constructor from the cross product of two vectors
773  // generate the rotation that, when applied to vector a, will result in vector b
774  Rotation( const Vector& a, const Vector& b ) {
775      _vec = unit( a ^ b ) + Vector( 1., 0., 0. ) + Vector( 0., 1., 0. ) + Vector( 0., 0., 1. );
776  }
777  if ( _vec == 0. ) { // than it must be the identity matrix
778  
779  Approved for public release; distribution is unlimited.
vec = DEFAULT_UNIT_VECTOR;
_ang = DEFAULT_ROTATION_ANGLE;
}
else {
  vec = _vec.unit();
  _ang = acos( 0.5 * ( A.a11 + A.a22 + A.a33 - 1. ) );
}
_set_angle(); // angle in the range [-M_PI, M_PI]
}

// constructor from two sets of three vectors, where the pair must be related by a pure rotation
// returns the rotation that will take a1 to b1, a2 to b2, and a3 to b3
// Ref: Micheals, R. J. and Boult, T. E., "Increasing Robustness in Self-Localization and Pose Estimation," online paper.

Rotation( const Vector& a1, const Vector& a2, const Vector& a3, // initial vectors
        const Vector& b1, const Vector& b2, const Vector& b3 ) { // rotated vectors
  assert( det( a1, a2, a3 ) != 0. && det( b1, b2, b3 ) != 0. ); // all vectors must be nonzero
  assert( fabs( det( a1, a2, a3 ) - det( b1, b2, b3 ) ) < 0.001 ); // if it doesn’t preserve volume, it’s not a pure rotation
  if ( det( a1, a2, a3 ) == 1 ) { // these are basis vectors so use simpler method to construct the rotation
    matrix A;
    A.a11 = b1 * a1; A.a12 = b2 * a1; A.a13 = b3 * a1;
    A.a21 = b1 * a2; A.a22 = b2 * a2; A.a23 = b3 * a2;
    A.a31 = b1 * a3; A.a32 = b2 * a3; A.a33 = b3 * a3;
    _vec = ( A.a32 - A.a23 ) * Vector( 1., 0., 0. ) +
            ( A.a13 - A.a31 ) * Vector( 0., 1., 0. ) +
            ( A.a21 - A.a12 ) * Vector( 0., 0., 1. ) ;
    if ( _vec == 0. ) { // then it must be the identity matrix
      _vec = DEFAULT_UNIT_VECTOR;
      _ang = DEFAULT_ROTATION_ANGLE;
    }
  }
  else { // use R. J. Micheals’ closed-form solution to the absolute orientation problem
    Vector c1, c2, c3;
    double aaa, baa, aba, aab, caa, aca, aac;
    double q02, q0, q0q1, q1, q0q2, q2, q0q3, q3;
    aaa = det( a1, a2, a3 );
    baa = det( b1, a2, a3 );
    aba = det( a1, b2, a3 );
    aab = det( a1, a2, b3 );
    _vec = fabs( { aaa + baa + aba + aab } ) / ( 4. * aaa ) ;
    q02 = sqrt(qq2);
    c1 = Vector( 0., b1[Z], -b1[Y] ) ;
    c2 = Vector( 0., b2[Z], -b2[Y] ) ;
    c3 = Vector( 0., b3[Z], -b3[Y] ) ;
    caa = det( c1, a2, a3 );
    aca = det( a1, c2, a3 );
    aac = det( a1, a2, c3 );
    q0 = ( { caa + aca + aac } ) / ( 4. * aaa ) ;
    q1 = q0q1 / q0;
    c1 = Vector( -b1[Z], 0., b1[X] ) ;
    c2 = Vector( -b2[Z], 0., b2[X] ) ;
    c3 = Vector( -b3[Z], 0., b3[X] ) ;
    caa = det( c1, a2, a3 );
    aca = det( a1, c2, a3 );
    aac = det( a1, a2, c3 );
    q0q2 = ( { caa + aca + aac } ) / ( 4. * aaa ) ;
    q2 = q0q2 / q0;
    c1 = Vector( b1[Y], -b1[X], 0. ) ;
    c2 = Vector( b2[Y], -b2[X], 0. ) ;
    c3 = Vector( b3[Y], -b3[X], 0. ) ;
    caa = det( c1, a2, a3 );
    aca = det( a1, c2, a3 );
    aac = det( a1, a2, c3 );
\[ q_0 q_3 = \frac{(c_0 + a_0 + a_0 c_0)}{(4 \times a_0)}; \]

\[ q_3 = \frac{q_0 q_3}{q_0}; \]

// no need to normalize since constructed to be unit quaternions

double w; q0;
Vector v1, q2, q3; 
if ( w >= 1. || v == 0. ) {
    _ang = DEFAULT_ROTATION_ANGLE;
    _vec = DEFAULT_UNIT_VECTOR;
} else {
    _ang = 2. * acos( w );
    _vec = v / sqrt( 1. - w * w );
}
_set_angle(); // angle in the range [-PI, PI]

// constructor for a uniform random rotation, uniformly distributed over the unit sphere,
// by fast generation of random quaternions, uniformly-distributed over the 4D unit sphere
Rotation( rng::Random &rng ) { // random rotation in canonical form
    double s = rng.uniform( 0., 1. );
    double s1 = sqrt( 1. - s );
    double th1 = rng.uniform( 0., TWO_PI );
    double x = s1 * sin( th1 );
    double s2 = sqrt( s );
    double th2 = rng.uniform( 0., TWO_PI );
    double z = s2 * sin( th2 );
    double w = s2 * cos( th2 );
    Vector v( x, y, z ); 
    if ( w >= 1. || v == 0. ) {
        _ang = DEFAULT_ROTATION_ANGLE;
        _vec = DEFAULT_UNIT_VECTOR;
    } else {
        _ang = 2. * acos( w );
        _vec = v / sqrt( 1. - w * w );
    }
    _set_angle(); // angle in the range [-PI, PI]
}

// default constructor
Rotation( void ) {
    _vec = DEFAULT_UNIT_VECTOR;
    _ang = DEFAULT_ROTATION_ANGLE;
}

// default destructor
~Rotation( void ) {
}

// copy constructor
Rotation( const Rotation &r ) : _vec( r._vec ), _ang( r._ang ) {
    _set_angle(); // angle in the range [-PI, PI]
}

// overloaded assignment operator
Rotation &operator=( const Rotation &R ) {
    if ( this != &R ) {
        _vec = R._vec;
        _ang = R._ang;
    }
    _set_angle(); // angle in the range [-PI, PI]
    return *this;
}

// conversion operator to return the eigenvector vec
operator Vector( void ) const { 
    return _vec;
}

// conversion operator to return the angle of rotation about the eigenvector
operator double( void ) const {
    return _ang;
The Rotation class is also enclosed in a va namespace, so that Rotations are declared by va::Rotation or by the declaration using namespace va. Table B-1 provides a reference sheet for basic usage.

Approved for public release; distribution is unlimited.
Table B-1. Rotation: A C++ class for 3-dimensional rotations—reference sheet

<table>
<thead>
<tr>
<th>Operation</th>
<th>Mathematical notation</th>
<th>Rotation class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition(^a)</td>
<td>Let ( R ) be an unspecified rotation.</td>
<td>Rotation ( R );</td>
</tr>
<tr>
<td></td>
<td>Let ( R ) be a rotation specified by yaw, pitch, and roll.(^b)</td>
<td>Rotation ( R(y,p,r,\text{ZYX}) );</td>
</tr>
<tr>
<td></td>
<td>Let ( R ) be the rotation specified by three angles, ( \phi_1, \phi_2, \phi_3 ) applied in the order ( x\text{-}y\text{-}z ).</td>
<td>Rotation ( b(\phi_1,\phi_2,\phi_3,\text{XYZ}) );</td>
</tr>
<tr>
<td></td>
<td>Let ( R_a(\alpha) ) be the rotation about the vector ( a ) through the angle ( \alpha ).</td>
<td>Vector ( a );</td>
</tr>
<tr>
<td></td>
<td>Let ( R ) be the rotation specified by the three angles, ( \phi ), through the angle ( \alpha ).</td>
<td>Rotation ( R(\text{p}(\theta,\phi)) );</td>
</tr>
<tr>
<td></td>
<td>Let ( R ) be the rotation specified by the vector cross product ( a \times b ).</td>
<td>Vector ( a, b );</td>
</tr>
<tr>
<td></td>
<td>Let ( R ) be the rotation that maps the set of linearly independent vectors ( {a_i} ) to the set ( {b_i} ), where ( i = 1,2,3 ).</td>
<td>Rotation ( R(a_1,a_2,a_3,b_1,b_2,b_3) );</td>
</tr>
<tr>
<td></td>
<td>Let ( R ) be the rotation specified by the (unit) quaternion ( q ).(^c)</td>
<td>Quaternion ( q );</td>
</tr>
<tr>
<td></td>
<td>Let ( R ) be the rotation specified by the 3 ( \times ) 3 rotation matrix ( A_{ij} ).(^d)</td>
<td>Matrix ( A );</td>
</tr>
<tr>
<td></td>
<td>Let ( R ) be a random rotation, designed to randomly orient any vector uniformly over the unit sphere.</td>
<td>Rotation ( R(\text{rng}) );</td>
</tr>
<tr>
<td>Input a rotation ( R )</td>
<td>( n/a )</td>
<td>( \text{cin} &gt;&gt; R; )</td>
</tr>
<tr>
<td>Output the rotation ( R )</td>
<td>( n/a )</td>
<td>( \text{cout} &lt;&lt; R; )</td>
</tr>
<tr>
<td>Assign one rotation to another</td>
<td>Let ( R_2 = R_1 ) or ( R_2 \Leftarrow R_1 )</td>
<td>( R_2 = R_1; ) or ( R2(R1); )</td>
</tr>
<tr>
<td>Product of two successive rotations(^e)</td>
<td>( R_2R_1 )</td>
<td>( R2 * R1; )</td>
</tr>
<tr>
<td>Rotation of a vector ( a )</td>
<td>( Ra )</td>
<td>( R * a; )</td>
</tr>
<tr>
<td>Inverse rotation</td>
<td>( R^{-1} )</td>
<td>( \text{inverse}(R); ) or ( -R; )</td>
</tr>
<tr>
<td>Convert a rotation to a quaternion</td>
<td>If ( R_\theta(u,\theta) ) is the rotation, then ( q = \cos(\theta/2) + u \sin(\theta/2) ).</td>
<td>To quaternion ( R; )</td>
</tr>
<tr>
<td>Convert a rotation to a 3 ( \times ) 3 matrix</td>
<td>See description on next page.</td>
<td>To matrix ( R; )</td>
</tr>
<tr>
<td>Factor a rotation into a rotation sequence</td>
<td>See description on next page.</td>
<td>Sequence ( s = \text{factor}(R,\text{ZYX}); )(^i)</td>
</tr>
<tr>
<td>Unit vector along the axis of rotation</td>
<td>Unit vector ( u ) in the rotation ( R_\theta(u) )</td>
<td>Vector ( R; ) or ( \text{vec}(R); )</td>
</tr>
<tr>
<td>Rotation angle</td>
<td>Angle ( \theta ) in the rotation ( R_\theta(u) )</td>
<td>Double ( R; ) or ( \text{ang}(R); )</td>
</tr>
</tbody>
</table>

\(^a\)A rotation is represented in the Rotation class by the pair \((\hat{u}, \theta)\), where \( \hat{u} \) is the unit vector along the axis of rotation, and \( \theta \) is the counterclockwise rotation angle.

\(^b\)The order is significant: first yaw is applied as a counterclockwise (CCW) rotation about the \( z \)-axis, then pitch is applied as a CCW rotation about the \( y' \)-axis, and finally, roll is applied as a CCW rotation about the \( x'' \)-axis. The coordinate system is constructed from the local tangent plane in which the \( z \)-axis points toward earth center, the \( x \)-axis points along the direction of travel, and the \( y \)-axis points to the right, to form a right-handed coordinate system. The particular order is specified by using \( \text{ZYX} \). There are a total of 12 possible orderings available to the user, 6 of them have distinct principal rotation axes: \( \text{XYZ}, \text{XZY}, \text{YXZ}, \text{YZX}, \text{ZXY}, \text{ZYX} \); and 6 have repeated principal rotation axes: \( \text{XYX}, \text{XXZ}, \text{YYX}, \text{YYZ}, \text{ZZX}, \text{ZZY} \).

\(^c\)Approved for public release; distribution is unlimited.
**Convert rotation to a 3 × 3 matrix:** First we convert the rotation $R_u(\theta)$ into the unit quaternion, via $q = \cos(\theta/2) + \hat{u} \sin(\theta/2)$, and set $w = \cos(\theta/2)$, the scalar part, and $v = \hat{u} \sin(\theta/2)$, the vector part. Then the rotation matrix is

$$
\begin{bmatrix}
2w^2 - 1 + 2v_1^2 & 2v_1v_2 - 2wv_3 & 2v_1v_3 + 2wv_2 \\
2v_1v_2 + 2wv_3 & 2w^2 - 1 + 2v_2^2 & 2v_2v_3 - 2wv_1 \\
2v_1v_3 - 2wv_2 & 2v_2v_3 + 2wv_1 & 2w^2 - 1 + 2v_3^2
\end{bmatrix}.
$$

**Factor a rotation into a rotation sequence:** First we convert the rotation $R_u(\theta)$ into the unit quaternion, via $q = \cos(\theta/2) + \hat{u} \sin(\theta/2)$, and set $w = \cos(\theta/2)$, the scalar part, and $v = \hat{u} \sin(\theta/2)$, the vector part. Next, let $p_0 = w$, $p_1 = v_1$, $p_2 = v_2$, $p_3 = v_3$ and set $A = p_0p_1 + p_2p_3$, $B = p_2^2 - p_0^2$, $D = p_1^2 - p_3^2$. Then $\phi_3 = \tan^{-1}(-2A/(B + D))$ is the third angle, which is roll about the $x$-axis in this case. Now set $c_0 = \cos(\phi_3/2)$, $c_1 = \sin(\phi_3/2)$, $q_0 = p_0c_0 + p_1c_1$, $q_2 = p_2c_0 - p_3c_1$, and $q_3 = p_3c_0 + p_2c_1$. Then $\phi_1 = 2\tan^{-1}(q_3/q_0)$ is the first angle, which is yaw about the $z$-axis, and $\phi_2 = 2\tan^{-1}(q_2/q_0)$ is the second angle, which is pitch about the $y$-axis.

---

*A quaternion is defined in the Rotation class as follows:

```c
struct quaternion {
    double w; // scalar part
    Vector v; // vector part
};
```

A unit quaternion requires that $w^2 + ||v||^2 = 1$.

*A matrix is defined in the Rotation class as follows:

```c
struct matrix {
    double a11, a12, a13; // 1st row
    double a21, a22, a23; // 2nd row
    double a31, a32, a33; // 3rd row
};
```

To qualify as a rotation, the 3 matrix $A$ must satisfy the 2 conditions: $A^\dagger = A^{-1}$ and det $A = 1$.

*In general, rotations do not commute, i.e. $R_1R_2 \neq R_2R_1$, so the order is significant and goes from right to left.

*A (rotation) sequence is defined in the Rotation class as follows:

```c
struct sequence {
    double first; // 1st rotation (rad) to apply to body axis
    double second; // 2nd rotation (rad) to apply to body axis
    double third; // 3rd rotation (rad) to apply to body axis
};
```

The order these are applied is always left to right: first, second, third. How they get applied is specified by using one of the following, which is applied left to right: ZYX, XYZ, ZXY, YXZ, ZYX, ZYX, ZYX, YZX, YXY, YXX, XZX. For example, the order XYZ would apply first to rotation about the $x$-axis, second to rotation about the $y$-axis, and third to rotation about the $z$-axis.

---

*Approved for public release; distribution is unlimited.
Appendix C. Quaternion Algebra and Vector Rotations
C-1. Quaternion Multiplication

Starting with the multiplication rule

\[ \hat{\mathbf{i}}^2 = \hat{\mathbf{j}}^2 = \hat{\mathbf{k}}^2 = \hat{\mathbf{i}} \hat{\mathbf{j}} \hat{\mathbf{k}} = -1, \]  

(C-1)

it then follows that

\[ \hat{\mathbf{i}} \hat{\mathbf{j}} = -\hat{\mathbf{j}} \hat{\mathbf{i}} = \hat{\mathbf{k}}, \quad \hat{\mathbf{j}} \hat{\mathbf{k}} = -\hat{\mathbf{k}} \hat{\mathbf{j}} = \hat{\mathbf{i}}, \quad \text{and} \quad \hat{\mathbf{k}} \hat{\mathbf{i}} = -\hat{\mathbf{i}} \hat{\mathbf{k}} = \hat{\mathbf{j}}. \]  

(C-2)

These rules are then sufficient to establish any other multiplication. Thus, let

\[ q_1 = w_1 + \mathbf{v}_1 = w_1 + \hat{\mathbf{i}} x_1 + \hat{\mathbf{j}} y_1 + \hat{\mathbf{k}} z_1 \]
\[ q_2 = w_2 + \mathbf{v}_2 = w_2 + \hat{\mathbf{i}} x_2 + \hat{\mathbf{j}} y_2 + \hat{\mathbf{k}} z_2 \]

be 2 quaternions. Unlike vectors, where there are 2 different products—the scalar product and the vector product—in the case of quaternions there is only one product, as follows:

\[ q_1 q_2 = (w_1 + \hat{\mathbf{i}} x_1 + \hat{\mathbf{j}} y_1 + \hat{\mathbf{k}} z_1)(w_2 + \hat{\mathbf{i}} x_2 + \hat{\mathbf{j}} y_2 + \hat{\mathbf{k}} z_2) \]
\[ = (w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2) \]
\[ + \hat{\mathbf{i}} (w_1 x_2 + w_2 x_1 + y_1 z_2 - z_1 y_2) \]
\[ + \hat{\mathbf{j}} (w_1 y_2 + w_2 y_1 + z_1 x_2 - x_1 z_2) \]
\[ + \hat{\mathbf{k}} (w_1 z_2 + w_2 z_1 + x_1 y_2 - y_1 x_2) \]
\[ = w_1 w_2 - \mathbf{v}_1 \cdot \mathbf{v}_2 + w_1 \mathbf{v}_2 + w_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2. \]  

(C-3)

Thus, if we represent a quaternion as an ordered pair, \( q = (w, \mathbf{v}) \), of a scalar and a vector, then

\[ (w_1, \mathbf{v}_1)(w_2, \mathbf{v}_2) = (w_1 w_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, w_1 \mathbf{v}_2 + w_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2). \]  

(C-4)

The scalar part of the product is

\[ w_1 w_2 - \mathbf{v}_1 \cdot \mathbf{v}_2 \]

and the vector part is

\[ w_1 \mathbf{v}_2 + w_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2. \]
C-2. Quaternion Division

Let \( q = w + v \) be a unit quaternion, in the sense that \( w^2 + ||v||^2 = 1 \). Then \( q^{-1} = w - v \) is the inverse, since

\[
qq^{-1} = (w, v)(w, -v) = (w^2 - v \cdot (-v), w(-v) + wv + v \times (-v)) = (w^2 + ||v||^2, 0) = 1.
\]

Thus, the inverse of a unit quaternion is the quaternion with a negative vector part. In effect, this serves to define quaternion division.*

C-3. Rotation of a Vector

Let \( \mathbf{a} \) be an arbitrary vector, let \( \mathbf{\hat{u}} \) be a unit vector along the axis of rotation, and let \( \theta \) be the angle of rotation. The (unit) quaternion that represents the rotation is given by

\[
q = \left( \cos \frac{\theta}{2}, \mathbf{\hat{u}} \sin \frac{\theta}{2} \right) \equiv (w, v).
\]

Then the rotated vector, \( \mathbf{a}' \) is given by

\[
\mathbf{a}' = qaq^{-1}
\]

\[
= (w, v)(0, \mathbf{a})(w, -v) = (w, v)(\mathbf{\hat{u}} \cdot \mathbf{v}, w\mathbf{a} - \mathbf{a} \times \mathbf{v})
\]

\[
= (w\mathbf{a} \cdot \mathbf{v} - \mathbf{v} \cdot (w\mathbf{a} - \mathbf{a} \times \mathbf{v}), w(w\mathbf{a} - \mathbf{a} \times \mathbf{v}) + (\mathbf{\hat{u}} \cdot \mathbf{v})\mathbf{v} + \mathbf{v} \times (w\mathbf{a} - \mathbf{a} \times \mathbf{v})) = (0, w^2\mathbf{a} + wv \times \mathbf{a} + (\mathbf{\hat{u}} \cdot \mathbf{v})\mathbf{v} + w\mathbf{v} \times \mathbf{a} + \mathbf{v} \times (\mathbf{v} \times \mathbf{a}))
\]

\[
= (0, w^2\mathbf{a} + v^2\mathbf{a} - v^2\mathbf{a} + (\mathbf{\hat{u}} \cdot \mathbf{v})\mathbf{v} + 2w\mathbf{a} \times \mathbf{v} + \mathbf{a} \times (\mathbf{v} \times \mathbf{a}) = (0, \mathbf{a} + 2wv \times \mathbf{a} + 2\mathbf{v} \times (\mathbf{v} \times \mathbf{a})),
\]

where we used the fact that \( w^2 + v^2 = 1 \) and \( (\mathbf{\hat{u}} \cdot \mathbf{v})\mathbf{v} - v^2\mathbf{a} = \mathbf{v} \times (\mathbf{v} \times \mathbf{a}) \). Therefore,*


Approved for public release; distribution is unlimited.
the rotated vector is given by

\[ a' = a + 2wv \times a + 2v \times (v \times a). \]  \hspace{1cm} (C-8)

Using \( w = \cos \theta/2 \) and \( v = \hat{u} \sin \theta/2 \), we also have

\[ a' = a + \hat{u} \times a \sin \theta + \hat{u} \times (\hat{u} \times a) (1 - \cos \theta), \]  \hspace{1cm} (C-9)

where we made use of the half-angle formulas \( 2 \cos(\theta/2) \sin(\theta/2) = \sin \theta \) and \( 2 \sin^2(\theta/2) = 1 - \cos \theta \).

Since this is such a fundamental formula, let us derive it in another way. For an arbitrary vector \( a \), we can always write

\[ a = a - (a \cdot \hat{u})\hat{u} + (a \cdot \hat{u})\hat{u}, \]  \hspace{1cm} (C-10)

where the third term on the right is the component of \( a \) that is parallel to \( \hat{u} \) and so will remain unchanged after a rotation about \( \hat{u} \). The first 2 terms form the component of \( a \) that is perpendicular to \( \hat{u} \) and will be rotated into

\[ [a - (a \cdot \hat{u})\hat{u}] \cos \theta + \hat{u} \times [a - (a \cdot \hat{u})\hat{u}] \sin \theta. \]  \hspace{1cm} (C-11)

Hence,

\[ a' = Ra = [a - (a \cdot \hat{u})\hat{u}] \cos \theta + \hat{u} \times [a - (a \cdot \hat{u})\hat{u}] \sin \theta + (a \cdot \hat{u})\hat{u} \]
\[ = [a - (a \cdot \hat{u})\hat{u}] \cos \theta + \hat{u} \times a \sin \theta + (a \cdot \hat{u})\hat{u}. \]  \hspace{1cm} (C-12)

Now,

\[ \hat{u} \times (a \times \hat{u}) = a(\hat{u} \cdot \hat{u}) - \hat{u}(\hat{u} \cdot a) = a - (a \cdot \hat{u})\hat{u}, \]

and therefore

\[ a' = \hat{u} \times (a \times \hat{u}) \cos \theta + \hat{u} \times a \sin \theta + a - \hat{u} \times (a \times \hat{u}) \]
\[ = -\hat{u} \times (\hat{u} \times a) \cos \theta + \hat{u} \times a \sin \theta + a + \hat{u} \times (\hat{u} \times a) \]
\[ = a + \hat{u} \times a \sin \theta + \hat{u} \times (\hat{u} \times a) (1 - \cos \theta). \]  \hspace{1cm} (C-13)
Appendix D. Fundamental Theorem of Rotation Sequences
**Fundamental Theorem of Rotation Sequences:** A rotation sequence about body axes is equivalent to the same rotation sequence applied in reverse order about fixed axes.

The conventional way of performing a rotation sequence is to account for the transformation of the body axes of the object we are rotating. For example, if we wanted to first perform pitch about the \( x \)-axis, followed by yaw about the \( y \)-axis, and ending with roll about the \( z \)-axis, then the rotation, applied right to left, is

\[
R = R_k''(\phi_r)R_y'(\phi_y)R_1(\phi_p),
\]

where \( j' = R_1(\phi_p)j \), \( k' = R_1(\phi_p)k \), and \( \hat{k}'' = R_y'(\phi_y)\hat{k}' = R_y'(\phi_y)R_1(\phi_p)\hat{k} \). But it is a fundamental result of rotation sequences that you get the same result by applying the rotation sequence in reverse order about fixed axes. That is,

\[
R = R_k''(\phi_r)R_y'(\phi_y)R_1(\phi_p) = R_1(\phi_p)R_y(\phi_y)R_k(\phi_r),
\]

which is simpler and more efficient. This can be proved with quaternions as follows. We use the notation,

\[
q_{\hat{u}}(\phi) = \cos \frac{\phi}{2} + \hat{u} \sin \frac{\phi}{2}
\]

for the unit quaternion that represents a counterclockwise rotation of \( \phi \) radians about the unit vector \( \hat{u} \). Then,

\[
R_1 = q_{\hat{e}_1}(\phi_1),
\]

\[
R_2 = q_{\hat{e}_2}(\phi_2) = \cos \frac{\phi_2}{2} + \hat{e}_2' \sin \frac{\phi_2}{2},
\]

where

\[
\hat{e}_2' = q_{\hat{e}_1}(\phi_1)\hat{e}_2q_{\hat{e}_1}(\phi_1),
\]

so that

\[
q_{\hat{e}_2}(\phi_2) = \cos \frac{\phi_2}{2} + \hat{e}_2' \sin \frac{\phi_2}{2}
\]

\[
= \cos \frac{\phi_2}{2} + q_{\hat{e}_1}(\phi_1)\hat{e}_2q_{\hat{e}_1}(\phi_1) \sin \frac{\phi_2}{2}
\]

\[
= q_{\hat{e}_1}(\phi_1)\left( \cos \frac{\phi_2}{2} + \hat{e}_2 \sin \frac{\phi_2}{2} \right) q_{\hat{e}_1}(\phi_1)
\]

\[
= q_{\hat{e}_1}(\phi_1)q_{\hat{e}_2}(\phi_2)q_{\hat{e}_1}(\phi_1).
\]
And
\[
R_3 = q_{e_3''}^{\prime}(\phi_3) = \cos \frac{\phi_3}{2} + \hat{e}_3'' \sin \frac{\phi_3}{2},
\]
where
\[
\hat{e}_3'' = q_{e_2'}(\phi_2)q_{e_2}^{-1}(\phi_2)
\]
\[
= q_{e_2'}(\phi_2)q_{e_1}(\phi_1)\hat{e}_3q_{e_1}^{-1}(\phi_1)q_{e_2}^{-1}(\phi_2)
\]
\[
= [q_{e_1}(\phi_1)q_{e_2}(\phi_2)q_{e_1}^{-1}(\phi_1)]q_{e_1}(\phi_1)\hat{e}_3q_{e_1}^{-1}(\phi_1)[q_{e_1}(\phi_1)q_{e_2}(\phi_2)q_{e_1}^{-1}(\phi_1)]^{-1}
\]
\[
= q_{e_1}(\phi_1)q_{e_2}(\phi_2)q_{e_1}^{-1}(\phi_1)\hat{e}_3q_{e_1}^{-1}(\phi_1)q_{e_1}(\phi_1)q_{e_2}^{-1}(\phi_2)q_{e_1}^{-1}(\phi_1)
\]
\[
= q_{e_1}(\phi_1)q_{e_2}(\phi_2)\hat{e}_3q_{e_2}^{-1}(\phi_2)q_{e_1}^{-1}(\phi_1).
\]
so that
\[
q_{e_3''}(\phi_3) = \cos \frac{\phi_3}{2} + \hat{e}_3'' \sin \frac{\phi_3}{2}
\]
\[
= \cos \frac{\phi_3}{2} + q_{e_1}(\phi_1)q_{e_2}(\phi_2)\hat{e}_3q_{e_2}^{-1}(\phi_2)q_{e_1}^{-1}(\phi_1) \sin \frac{\phi_3}{2}
\]
\[
= q_{e_1}(\phi_1)q_{e_2}(\phi_2)q_{e_3}(\phi_3)q_{e_2}^{-1}(\phi_2)q_{e_1}^{-1}(\phi_1)
\]
\[
= q_{e_1}(\phi_1)q_{e_2}(\phi_2)q_{e_3}(\phi_3)q_{e_2}^{-1}(\phi_2)q_{e_1}^{-1}(\phi_1).
\]
Therefore, the total combined rotation is
\[
R = R_3R_2R_1 = q_{e_3''}(\phi_3)q_{e_2'}(\phi_2)q_{e_1}(\phi_1)
\]
\[
= [q_{e_1}(\phi_1)q_{e_2}(\phi_2)q_{e_3}(\phi_3)q_{e_2}^{-1}(\phi_2)q_{e_1}^{-1}(\phi_1)]q_{e_1}(\phi_1)q_{e_2}(\phi_2)q_{e_1}^{-1}(\phi_1)
\]
\[
= q_{e_1}(\phi_1)q_{e_2}(\phi_2)q_{e_3}(\phi_3),
\]
as was to be shown.

The program in Listing D-1 is designed to test this result.

Listing D-1. order.cpp

```cpp
// order.cpp: demonstrate that rotation about fixed axis in reverse order is correct
// for both rotation about distinct axes and for rotation about repeated axes
#include "Rotation.h"
#include <iostream>
#include <cstdlib>
using namespace std;

int main( int argc, char* argv[] )
{
    va::Vector i( 1.0, 0.0, 0.0 );
    va::Vector j( 0.0, 1.0, 0.0 );
    va::Vector k( 0.0, 0.0, 1.0 );
    va::Vector v1( 1.2, 3.4, 6.7 );
    va::Vector v2( 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1, 1.1 );

```

Approved for public release; distribution is unlimited.
```cpp
15 double ang_1 = 0.;
16 double ang_2 = 0.;
17 double ang_3 = 0.;
18 if ( argc == 4 ) {
19    ang_1 = va::rad( atof( argv[ 1 ] ) );
20    ang_2 = va::rad( atof( argv[ 2 ] ) );
21    ang_3 = va::rad( atof( argv[ 3 ] ) );
22 }
23
24 R = va::Rotation( ang_1, ang_2, ang_3, order );
25 cout << "The constructed rotation is " << R << endl;
26 cout << "The rotated vector is " << R * v << endl;
27 cout << "The following rotations should match this:" << endl;
28 R1 = va::Rotation( i, ang_1 ); // rotation about x-axis
29 R2 = va::Rotation( j, ang_2 ); // rotation about y-axis
30 R3 = va::Rotation( k, ang_3 ); // rotation about z-axis
31 R = R1 * R2 * R3; // note the order is the reverse: first 3, then 2, then 1
32 cout << "Reverse order about fixed axes:" << endl;
33 cout << "Rotation: " << R << endl;
34 cout << "Rotated vector: " << R * v << endl;
35
36 cout << endl << "Now the conventional way via transformed axes:" << endl << endl;
37 // first rotation is about i
38 R1 = va::Rotation( i, ang_1 ); // rotation about x-axis
39 i1 = R1 * i;
40 j1 = R1 * j;
41 k1 = R1 * k;
42 // second rotation is about j1
43 R2 = va::Rotation( j1, ang_2 ); // rotation about transformed y-axis
44 i2 = R2 * i1;
45 j2 = R2 * j1;
46 k2 = R2 * k1;
47 // third rotation is about k2
48 R3 = va::Rotation( k2, ang_3 ); // rotation about doubly transformed z-axis
49 i3 = R3 * i2;
50 j3 = R3 * j2;
51 k3 = R3 * k2;
52 R = R3 * R2 * R1; // note the order is the original: first 1, then 2, then 3
53 cout << "Original order about transformed axes:" << endl;
54 cout << "Rotation: " << R << endl;
55 cout << "Rotated vector: " << R * v << endl;
56 cout << "This also works for repeated axes." << endl;
57 cout << "Using the same rotation angles, let’s do the whole thing over again." << endl;
58 order = va::XYX; // select rotation sequence about repeated axes
59 R = va::Rotation( ang_1, ang_2, ang_3, order );
60 cout << "The constructed rotation is " << R << endl;
61 cout << "The rotated vector is " << R * v << endl;
62 cout << "The following rotations should match this:" << endl;
63 R1 = va::Rotation( i, ang_1 ); // rotation about x-axis
64 R2 = va::Rotation( j, ang_2 ); // rotation about y-axis
65 R3 = va::Rotation( i, ang_3 ); // rotation about x-axis
66 R = R1 * R2 * R3; // note the order is the reverse: first 3, then 2, then 1
67 cout << "Reverse order about fixed axes:" << endl;
68 cout << "Rotation: " << R << endl;
69 cout << "Rotated vector: " << R * v << endl;
70 cout << endl << "Now the conventional way via transformed axes:" << endl << endl;
71 // first rotation is about i
72 R1 = va::Rotation( i, ang_1 ); // rotation about x-axis
73 i1 = R1 * i;
74 j1 = R1 * j;
75 k1 = R1 * k;
76 // second rotation is about j1
77 R2 = va::Rotation( j1, ang_2 ); // rotation about transformed y-axis
78 i2 = R2 * i1;
79 j2 = R2 * j1;
80 k2 = R2 * k1;
81 // third rotation is about i2
82 // first rotation is about i
83 R1 = va::Rotation( i, ang_1 ); // rotation about x-axis
84 i1 = R1 * i;
85 j1 = R1 * j;
86 k1 = R1 * k;
87 // second rotation is about j1
88 R2 = va::Rotation( j1, ang_2 ); // rotation about transformed y-axis
89 i2 = R2 * i1;
90 j2 = R2 * j1;
91 k2 = R2 * k1;
92 // third rotation is about i2
93
```

Approved for public release; distribution is unlimited.
R3 = va::Rotation( i2, ang_3 );  // rotation about doubly transformed x-axis
i3 = R3 * i2;
j3 = R3 * j2;
k3 = R3 * k2;
R = R3 * R2 * R1;  // note the order is the original: first 1, then 2, then 3

cout << "Original order about transformed axes:" << endl;
cout << "Rotation: " << R << endl;
cout << "Rotated vector: " << R * v << endl;
return EXIT_SUCCESS;
}

The command
./order 35. -15. 60.

will give the following results:
The constructed rotation is -0.533171 -0.0686517 -0.843218 73.8825
The rotated vector is -0.530512 1.81621 7.36953
The following rotations should match this:
Reverse order about fixed axes:
Rotation: -0.533171 -0.0686517 -0.843218 73.8825
Rotated vector: -0.530512 1.81621 7.36953
Now the conventional way via transformed axes:
Original order about transformed axes:
Rotation: -0.530512 1.81621 7.36953
Rotated vector: -0.530512 1.81621 7.36953
This also works for repeated axes.
Using the same rotation angles, we do the whole thing over again.
The constructed rotation is -0.984429 0.171618 0.0380469 95.8951
The rotated vector is 2.78824 6.66967 2.37301
The following rotations should match this:
Reverse order about fixed axes:
Rotation: -0.984429 0.171618 0.0380469 95.8951
Rotated vector: 2.78824 6.66967 2.37301
Now the conventional way via transformed axes:
Original order about transformed axes:
Rotation: -0.984429 0.171618 0.0380469 95.8951
Rotated vector: 2.78824 6.66967 2.37301

Approved for public release; distribution is unlimited.
INTENTIONALLY LEFT BLANK.
Appendix E. Factoring a Rotation into a Rotation Sequence
E-1. Distinct Principal Axis Factorization

The most common rotation sequence is probably the aerospace sequence, which consists of yaw about the body z-axis, pitch about the body y-axis, and roll about the body x-axis—in that order. However, there are a total of 6 such distinct principal axis rotation sequences and we will factor each one.¹

Let us begin with the z-y-x (aerospace) rotation sequence, consisting of yaw about the z-axis, followed by pitch about the y-axis and ending with roll about the x-axis.

Let the given rotation be represented by the quaternion

\[ p = p_0 + \hat{i}p_1 + \hat{j}p_2 + \hat{k}p_3 . \] (E-1)

In the notation of Kuipers¹ (see pp. 194–196), we want to factor this as \( a^3b^2c^1 \), so we write

\[ p = (a_0 + \hat{k}a_3)(b_0 + \hat{j}b_2)(c_0 + \hat{i}c_1) . \] (E-2)

Let \( q \) represent the first 2 factors:

\[ q = (a_0 + \hat{k}a_3)(b_0 + \hat{j}b_2) = a_0b_0 - \hat{i}a_3b_2 + \hat{j}a_0b_2 + \hat{k}a_3b_0 . \] (E-3)

Then

\[ q = p(c^1)^{-1} = (p_0 + \hat{i}p_1 + \hat{j}p_2 + \hat{k}p_3)(c_0 - \hat{i}c_1) \]
\[ = (p_0c_0 + p_1c_1) + \hat{i}(p_1c_0 - p_0c_1) + \hat{j}(p_2c_0 - p_3c_1) + \hat{k}(p_3c_0 + p_2c_1) , \] (E-4)

from which we identify

\[ q_0 = p_0c_0 + p_1c_1, \quad q_1 = p_1c_0 - p_0c_1, \quad q_2 = p_2c_0 - p_3c_1, \quad q_3 = p_3c_0 + p_2c_1 . \] (E-5)

The constraint equation for this to be a tracking rotation sequence follows from Eq. E-3:

\[ q_0q_1 + q_2q_3 = \begin{bmatrix} q_0 & q_2 \\ q_1 & q_3 \end{bmatrix} = (a_0b_0)(-a_3b_2) + (a_0b_2)(a_3b_0) = 0 . \] (E-6)

Now, from Eq. E-5,

\[
\begin{bmatrix}
q_0 \\
q_2
\end{bmatrix} =
\begin{bmatrix}
p_0c_0 + p_1c_1 \\
p_2c_0 - p_3c_1
\end{bmatrix} =
\begin{bmatrix}
p_0 \\
p_2
\end{bmatrix}
\begin{bmatrix}
p_1 \\
p_3 \\
- p_3
\end{bmatrix}
\begin{bmatrix}
c_0 \\
c_1
\end{bmatrix}
\quad \text{(E-7)}
\]

and

\[
\begin{bmatrix}
q_1 \\
q_3
\end{bmatrix} =
\begin{bmatrix}
p_1c_0 - p_0c_1 \\
p_3c_0 + p_2c_1
\end{bmatrix} =
\begin{bmatrix}
p_1 \\
p_3
\end{bmatrix}
\begin{bmatrix}
-p_0 \\
p_2
\end{bmatrix}
\begin{bmatrix}
c_0 \\
c_1
\end{bmatrix},
\quad \text{(E-8)}
\]

so the constraint equation, Eq. E-6, may be written as

\[
\begin{bmatrix}
c_0 \\
c_1
\end{bmatrix}
\begin{bmatrix}
p_0 & p_2 \\
p_1 & -p_3
\end{bmatrix}
\begin{bmatrix}
p_1 \\
p_2
\end{bmatrix}
\begin{bmatrix}
-c_0 \\
c_1
\end{bmatrix} =
\begin{bmatrix}
c_0 \\
c_1
\end{bmatrix}
\begin{bmatrix}
p_0 & p_2 \\
p_1 & -p_3
\end{bmatrix}
\begin{bmatrix}
-p_0 \\
p_2
\end{bmatrix}
\begin{bmatrix}
c_0 \\
c_1
\end{bmatrix} = 0.
\quad \text{(E-9)}
\]

Define the quantities

\[
A = p_0p_1 + p_2p_3, \quad B = -p_0^2 + p_2^2, \quad D = p_1^2 - p_3^2.
\quad \text{(E-10)}
\]

Then the constraint equation becomes

\[
\begin{bmatrix}
c_0 \\
c_1
\end{bmatrix}
\begin{bmatrix}
A & B \\
D & -A
\end{bmatrix}
\begin{bmatrix}
c_0 \\
c_1
\end{bmatrix} = A(c_0^2 - c_1^2) + (B + D)c_0c_1 = 0.
\quad \text{(E-11)}
\]

Finally, this may be written as

\[
-\frac{2A}{B + D} = \frac{2c_0c_1}{c_0^2 - c_1^2} = \frac{2 \cos \frac{\phi_3}{2} \sin \frac{\phi_3}{2}}{\cos^2 \frac{\phi_3}{2} - \sin^2 \frac{\phi_3}{2}} = \frac{\sin \phi_3}{\cos \phi_3} = \tan \phi_3,
\quad \text{(E-12)}
\]

where we used

\[
c_0 = \cos \frac{\phi_3}{2} \quad \text{and} \quad c_1 = \sin \frac{\phi_3}{2}.
\quad \text{(E-13)}
\]

Therefore, the final rotation (roll angle in this case) is

\[
\phi_3 = \tan^{-1} \left( \frac{-2A}{B + D} \right),
\quad \text{(E-14)}
\]

and this quantity is known since \(A, B,\) and \(D\) are known from Eq. E-10. Furthermore,
since
\[ a^3 = a_0 + \hat{k}a_3 = \cos \frac{\phi_1}{2} + \hat{k} \sin \frac{\phi_1}{2}, \] (E-15)
it follows that
\[ \tan \frac{\phi_1}{2} = \frac{a_3}{a_0} = \frac{a_3b_0}{a_0b_0} = \frac{q_3}{q_0}, \] (E-16)
from Eq. E-3. Therefore, the first rotation (yaw angle in this case) is given by
\[ \phi_1 = 2 \tan^{-1} \left( \frac{q_3}{q_0} \right). \] (E-17)
Similarly,
\[ \tan \frac{\phi_2}{2} = \frac{b_2}{b_0} = \frac{a_0b_2}{a_0b_0} = \frac{q_2}{q_0}, \] (E-18)
again using Eq. E-3, and therefore the second rotation (pitch angle in this case) is given by
\[ \phi_2 = 2 \tan^{-1} \left( \frac{q_2}{q_0} \right). \] (E-19)
In summary, the prescription for factoring an arbitrary rotation into yaw (about the z-axis), pitch (about the y-axis), and roll (about the x-axis) (in that order) is given in Table E-1.

<table>
<thead>
<tr>
<th>Table E-1. Factorization into z-y-x (aerospace) rotation sequence, consisting of yaw about the z-axis, pitch about the y-axis, and roll about the x-axis</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A = p_0p_1 + p_2p_3, ) ( B = p_2^2 - p_0^2, ) ( D = p_1^2 - p_3^2 )</td>
</tr>
<tr>
<td>( \phi_3 = \tan^{-1} \left( \frac{-2A}{B + D} \right) ) [third rotation, roll about x-axis]</td>
</tr>
<tr>
<td>( c_0 = \cos \frac{\phi_3}{2}, ) ( c_1 = \sin \frac{\phi_3}{2} )</td>
</tr>
<tr>
<td>( q_0 = p_0c_0 + p_1c_1, ) ( q_1 = p_1c_0 - p_0c_1, ) ( q_2 = p_2c_0 - p_3c_1, ) ( q_3 = p_3c_0 + p_2c_1 )</td>
</tr>
<tr>
<td>( \phi_1 = 2 \tan^{-1} \left( \frac{q_3}{q_0} \right) ) [first rotation, yaw about z-axis]</td>
</tr>
<tr>
<td>( \phi_2 = 2 \tan^{-1} \left( \frac{q_2}{q_0} \right) ) [second rotation, pitch about y-axis]</td>
</tr>
</tbody>
</table>

The calculations for the other 5 sequential orders are entirely similar and we simply summarize the results in Tables E-2 through E-6.
Table E-2. Factorization into $x$-$y$-$z$ (FATEPEN) rotation sequence, consisting of pitch about the $x$-axis, yaw about the $y$-axis, and roll about the $z$-axis

\[
A = p_1 p_2 - p_0 p_3, \quad B = p_1^2 - p_3^2, \quad D = p_0^2 - p_2^2
\]

\[
\phi_3 = \tan^{-1}\left(\frac{-2A}{B + D}\right) \quad \text{[third rotation, roll about z-axis]}
\]

\[
c_0 = \cos \frac{\phi_3}{2}, \quad c_3 = \sin \frac{\phi_3}{2}
\]

\[
q_0 = p_0 c_0 + p_3 c_3, \quad q_1 = p_1 c_0 - p_2 c_3, \quad q_2 = p_2 c_0 + p_1 c_3, \quad q_3 = p_3 c_0 - p_0 c_3
\]

\[
\phi_1 = 2 \tan^{-1}\left(\frac{q_1}{q_0}\right) \quad \text{[first rotation, pitch about x-axis]}
\]

\[
\phi_2 = 2 \tan^{-1}\left(\frac{q_2}{q_0}\right) \quad \text{[second rotation, yaw about y-axis]}
\]

Table E-3. Factorization into $y$-$x$-$z$ rotation sequence

\[
A = p_1 p_2 + p_0 p_3, \quad B = p_1^2 + p_3^2, \quad D = -p_0^2 - p_2^2
\]

\[
\phi_3 = \tan^{-1}\left(\frac{-2A}{B + D}\right) \quad \text{[third rotation about z-axis]}
\]

\[
c_0 = \cos \frac{\phi_3}{2}, \quad c_3 = \sin \frac{\phi_3}{2}
\]

\[
q_0 = p_0 c_0 + p_3 c_3, \quad q_1 = p_1 c_0 - p_2 c_3, \quad q_2 = p_2 c_0 + p_1 c_3, \quad q_3 = p_3 c_0 - p_0 c_3
\]

\[
\phi_1 = 2 \tan^{-1}\left(\frac{q_2}{q_0}\right) \quad \text{[first rotation about y-axis]}
\]

\[
\phi_2 = 2 \tan^{-1}\left(\frac{q_1}{q_0}\right) \quad \text{[second rotation about x-axis]}
\]
### Table E-4. Factorization into $z$-$x$-$y$ rotation sequence

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1p_3 - p_0p_2$</td>
<td>$p_0^2 - p_1^2$</td>
<td>$p_3^2 - p_2^2$</td>
</tr>
</tbody>
</table>

$\phi_3 = \tan^{-1}\left(\frac{-2A}{B + D}\right)$ [third rotation about $y$-axis]

$c_0 = \cos \frac{\phi_3}{2}$, $c_2 = \sin \frac{\phi_3}{2}$

$q_0 = p_0c_0 + p_2c_2$, $q_1 = p_1c_0 + p_3c_2$, $q_2 = p_2c_0 - p_0c_2$, $q_3 = p_3c_0 - p_1c_2$

$\phi_1 = 2\tan^{-1}\left(\frac{q_3}{q_0}\right)$ [first rotation about $z$-axis]

$\phi_2 = 2\tan^{-1}\left(\frac{q_1}{q_0}\right)$ [second rotation about $x$-axis]

### Table E-5. Factorization into $x$-$z$-$y$ rotation sequence

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1p_3 + p_0p_2$</td>
<td>$-p_0^2 - p_1^2$</td>
<td>$p_2^2 + p_3^2$</td>
</tr>
</tbody>
</table>

$\phi_3 = \tan^{-1}\left(\frac{-2A}{B + D}\right)$ [third rotation about $y$-axis]

$c_0 = \cos \frac{\phi_3}{2}$, $c_2 = \sin \frac{\phi_3}{2}$

$q_0 = p_0c_0 + p_2c_2$, $q_1 = p_1c_0 + p_3c_2$, $q_2 = p_2c_0 - p_0c_2$, $q_3 = p_3c_0 - p_1c_2$

$\phi_1 = 2\tan^{-1}\left(\frac{q_1}{q_0}\right)$ [first rotation about $x$-axis]

$\phi_2 = 2\tan^{-1}\left(\frac{q_3}{q_0}\right)$ [second rotation about $z$-axis]

### Table E-6. Factorization into $y$-$z$-$x$ rotation sequence

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_2p_3 - p_0p_1$</td>
<td>$p_0^2 + p_2^2$</td>
<td>$-p_1^2 - p_3^2$</td>
</tr>
</tbody>
</table>

$\phi_3 = \tan^{-1}\left(\frac{-2A}{B + D}\right)$ [third rotation about $x$-axis]

$c_0 = \cos \frac{\phi_3}{2}$, $c_1 = \sin \frac{\phi_3}{2}$

$q_0 = p_0c_0 + p_1c_1$, $q_1 = p_1c_0 - p_0c_1$, $q_2 = p_2c_0 - p_3c_1$, $q_3 = p_3c_0 + p_2c_1$

$\phi_1 = 2\tan^{-1}\left(\frac{q_2}{q_0}\right)$ [first rotation about $y$-axis]

$\phi_2 = 2\tan^{-1}\left(\frac{q_3}{q_0}\right)$ [second rotation about $z$-axis]
E-2. Repeated Principal Axis Factorization

We define a repeated principal axis sequence as first a rotation about one of the principal body axes, then a second rotation about another body axis, and finally a third rotation about the first body axes. There are a total of 6 such rotation sequences and we will factor each one.

We begin with the $z$-$y$-$z$ rotation sequence, consisting of first about the $z$-axis, second about the $y$-axis and third about the $z$-axis.

Let the given rotation be represented by the quaternion

$$p = p_0 + \hat{i}p_1 + \hat{j}p_2 + \hat{k}p_3.$$  \hspace{1cm} (E-20)

In the notation of Kuipers\(^1\), we want to factor this as $a^3b^2c^3$, so we write

$$p = (a_0 + \hat{k}a_3)(b_0 + \hat{j}b_2)(c_0 + \hat{k}c_3).$$  \hspace{1cm} (E-21)

Let $q$ represent the first 2 factors:

$$q = (a_0 + \hat{k}a_3)(b_0 + \hat{j}b_2) = a_0b_0 - \hat{i}a_3b_2 + \hat{j}a_0b_2 + \hat{k}a_3b_0.$$  \hspace{1cm} (E-22)

Then

$$q = p(c^1)^{-1} = (p_0 + \hat{i}p_1 + \hat{j}p_2 + \hat{k}p_3)(c_0 - \hat{k}c_3)$$

$$= (p_0c_0 + p_3c_3) + \hat{i}(p_1c_0 - p_2c_3) + \hat{j}(p_2c_0 + p_1c_3) + \hat{k}(p_3c_0 - p_0c_3),$$  \hspace{1cm} (E-23)

from which we identify

$$q_0 = p_0c_0 + p_3c_3, \quad q_1 = p_1c_0 - p_2c_3, \quad q_2 = p_2c_0 + p_1c_3, \quad q_3 = p_3c_0 - p_0c_3.$$  \hspace{1cm} (E-24)

The constraint equation for this to be a tracking rotation sequence follows from Eq. E-22:

$$q_0q_1 + q_2q_3 = \begin{bmatrix} q_0 & q_2 \\ q_1 & q_3 \end{bmatrix} \begin{bmatrix} q_1 \\ q_3 \end{bmatrix} = (a_0b_0)(-a_3b_2) + (a_0b_2)(a_3b_0) = 0.$$  \hspace{1cm} (E-25)

---

\(^1\)See Kuipers, pp. 200–201, for the technique, but note that there is a typo in Eq. 8.31, which leads to an error in Eq. 8.32. This has been corrected here.

Approved for public release; distribution is unlimited.
Now, from Eq. E-24,

\[
\begin{bmatrix}
    q_0 \\
    q_2
\end{bmatrix} = \begin{bmatrix}
    p_0 c_0 + p_3 c_3 \\
    p_2 c_0 + p_1 c_3
\end{bmatrix} = \begin{bmatrix}
    p_0 & p_3 \\
    p_2 & p_1
\end{bmatrix} \begin{bmatrix}
    c_0 \\
    c_3
\end{bmatrix}
\] (E-26)

and

\[
\begin{bmatrix}
    q_1 \\
    q_3
\end{bmatrix} = \begin{bmatrix}
    p_1 c_0 - p_2 c_3 \\
    p_3 c_0 - p_0 c_3
\end{bmatrix} = \begin{bmatrix}
    p_1 & -p_2 \\
    p_3 & -p_0
\end{bmatrix} \begin{bmatrix}
    c_0 \\
    c_3
\end{bmatrix},
\] (E-27)

so that the constraint equation, Eq. E-25, may be written as

\[
\begin{bmatrix}
    c_0 & c_3
\end{bmatrix} \begin{bmatrix}
    p_0 & p_2 \\
    p_3 & p_1
\end{bmatrix} \begin{bmatrix}
    p_1 & -p_2 \\
    p_3 & -p_0
\end{bmatrix} \begin{bmatrix}
    c_0 \\
    c_3
\end{bmatrix} = \begin{bmatrix}
    p_0 p_1 + p_2 p_3 & -p_0 p_2 - p_0 p_3 \\
    p_1 p_3 + p_1 p_3 & -p_2 p_3 - p_0 p_1
\end{bmatrix} \begin{bmatrix}
    c_0 \\
    c_3
\end{bmatrix} = 0.
\] (E-28)

Define the quantities

\[
A = p_0 p_1 + p_2 p_3, \quad B = -2p_0 p_2, \quad D = 2p_1 p_3.
\] (E-29)

Then the constraint equation becomes

\[
\begin{bmatrix}
    c_0 & c_3
\end{bmatrix} \begin{bmatrix}
    A & B \\
    D & -A
\end{bmatrix} \begin{bmatrix}
    c_0 \\
    c_3
\end{bmatrix} = A(c_0^2 - c_3^2) + (B + D)c_0 c_3 = 0.
\] (E-30)

Finally, this may be written as

\[
-\frac{2A}{B + D} = \frac{2c_0 c_3}{c_0^2 - c_3^2} = \frac{2 \cos \frac{\phi_3}{2} \sin \frac{\phi_3}{2}}{\cos^2 \frac{\phi_3}{2} - \sin^2 \frac{\phi_3}{2}} = \frac{\sin \phi_3}{\cos \phi_3} = \tan \phi_3,
\] (E-31)

where we used

\[
c_0 = \cos \frac{\phi_3}{2} \quad \text{and} \quad c_3 = \sin \frac{\phi_3}{2}.
\] (E-32)

Therefore, the final rotation is

\[
\phi_3 = \tan^{-1} \left( \frac{-2A}{B + D} \right),
\] (E-33)

and this quantity is known since \(A, B,\) and \(D\) are known from Eq. E-29. Furthermore,
since
\[ a^3 = a_0 + \hat{k}a_3 = \cos \frac{\phi_1}{2} + \hat{k} \sin \frac{\phi_1}{2}, \quad (E-34) \]
it follows that
\[ \tan \frac{\phi_1}{2} = \frac{a_3}{a_0} = \frac{a_3b_0}{a_0b_0} = \frac{q_3}{q_0}, \quad (E-35) \]
where we used Eq. E-22. Therefore, the first rotation is given by
\[ \phi_1 = 2 \tan^{-1} \left( \frac{q_3}{q_0} \right). \quad (E-36) \]
Similarly,
\[ \tan \frac{\phi_2}{2} = \frac{b_2}{b_0} = \frac{a_0b_2}{a_0b_0} = \frac{q_2}{q_0}, \quad (E-37) \]
again using Eq. E-22, and therefore the second rotation (pitch angle in this case) is given by
\[ \phi_2 = 2 \tan^{-1} \left( \frac{q_2}{q_0} \right). \quad (E-38) \]
In summary, the prescription for factoring an arbitrary rotation into an Euler sequence of first a rotation about the \( z \)-axis, followed by a rotation about the body \( y \)-axis, and finally ending with a rotation about the body \( z \)-axis, is given in Table E-7.

<table>
<thead>
<tr>
<th>Table E-7. Factorization into ( z-y-x ) rotation sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ A = p_0p_1 + p_2p_3, \quad B = -2p_0p_2, \quad D = 2p_1p_3 ]</td>
</tr>
<tr>
<td>[ \phi_3 = \tan^{-1} \left( \frac{-2A}{B + D} \right) ] [third rotation about ( z )-axis]</td>
</tr>
<tr>
<td>[ c_0 = \cos \frac{\phi_3}{2}, \quad c_3 = \sin \frac{\phi_3}{2} ]</td>
</tr>
<tr>
<td>[ q_0 = p_0c_0 + p_3c_3, \quad q_1 = p_1c_0 - p_2c_3, \quad q_2 = p_2c_0 + p_1c_3, \quad q_3 = p_3c_0 - p_0c_3 ]</td>
</tr>
<tr>
<td>[ \phi_1 = 2 \tan^{-1} \left( \frac{q_3}{q_0} \right) ] [first rotation about ( z )-axis]</td>
</tr>
<tr>
<td>[ \phi_2 = 2 \tan^{-1} \left( \frac{q_2}{q_0} \right) ] [second rotation about ( y )-axis]</td>
</tr>
</tbody>
</table>

The calculations for the other 5 sequential orders are entirely similar and we simply summarize the results in Tables E-8 through E-12.
Table E-8. Factorization into $z$-$x$-$z$ rotation sequence

\[
\begin{align*}
A &= p_0p_2 - p_1p_3, \\
B &= 2p_0p_1, \\
D &= 2p_2p_3 \\
\phi_3 &= \tan^{-1}\left(\frac{-2A}{B + D}\right) \quad \text{[third rotation about } z\text{-axis]} \\
c_0 &= \cos \frac{\phi_3}{2}, \\
c_3 &= \sin \frac{\phi_3}{2} \\
q_0 &= p_0c_0 + p_3c_3, \\
q_1 &= p_1c_0 - p_2c_3, \\
q_2 &= p_2c_0 + p_1c_3, \\
q_3 &= p_3c_0 - p_0c_3 \\
\phi_1 &= 2 \tan^{-1}\left(\frac{q_3}{q_0}\right) \quad \text{[first rotation about } z\text{-axis]} \\
\phi_2 &= 2 \tan^{-1}\left(\frac{q_1}{q_0}\right) \quad \text{[second rotation about } x\text{-axis]} 
\end{align*}
\]  

Table E-9. Factorization into $y$-$z$-$y$ rotation sequence

\[
\begin{align*}
A &= p_0p_1 - p_2p_3, \\
B &= p_0p_3 + p_1p_2 \\
\phi_3 &= \tan^{-1}\left(\frac{A}{B}\right) \quad \text{[third rotation about } y\text{-axis]} \\
c_0 &= \cos \frac{\phi_3}{2}, \\
c_2 &= \sin \frac{\phi_3}{2} \\
q_0 &= p_0c_0 + p_2c_2, \\
q_1 &= p_1c_0 + p_3c_2, \\
q_2 &= p_2c_0 - p_0c_2, \\
q_3 &= p_3c_0 - p_1c_2 \\
\phi_1 &= 2 \tan^{-1}\left(\frac{q_2}{q_0}\right) \quad \text{[first rotation about } y\text{-axis]} \\
\phi_2 &= 2 \tan^{-1}\left(\frac{q_1}{q_0}\right) \quad \text{[second rotation about } z\text{-axis]} 
\end{align*}
\]  

Table E-10. Factorization into $y$-$x$-$y$ rotation sequence

\[
\begin{align*}
A &= p_0p_3 + p_1p_2, \\
B &= -2p_0p_1, \\
D &= 2p_2p_3 \\
\phi_3 &= \tan^{-1}\left(\frac{-2A}{B + D}\right) \quad \text{[third rotation about } y\text{-axis]} \\
c_0 &= \cos \frac{\phi_3}{2}, \\
c_2 &= \sin \frac{\phi_3}{2} \\
q_0 &= p_0c_0 + p_2c_2, \\
q_1 &= p_1c_0 + p_3c_2, \\
q_2 &= p_2c_0 - p_0c_2, \\
q_3 &= p_3c_0 - p_1c_2 \\
\phi_1 &= 2 \tan^{-1}\left(\frac{q_2}{q_0}\right) \quad \text{[first rotation about } y\text{-axis]} \\
\phi_2 &= 2 \tan^{-1}\left(\frac{q_1}{q_0}\right) \quad \text{[second rotation about } x\text{-axis]} 
\end{align*}
\]
Table E-11. Factorization into $x-y-x$ rotation sequence

\[
A = p_0 p_3 - p_1 p_2, \quad B = p_0 p_2 + p_1 p_3
\]

\[
\phi_3 = \tan^{-1} \left( \frac{-A}{B} \right) \quad \text{[third rotation about $x$-axis]}
\]

\[
c_0 = \cos \frac{\phi_3}{2}, \quad c_1 = \sin \frac{\phi_3}{2}
\]

\[
q_0 = p_0 c_0 + p_1 c_1, \quad q_1 = p_1 c_0 - p_0 c_1, \quad q_2 = p_2 c_0 - p_3 c_1, \quad q_3 = p_3 c_0 + p_2 c_1
\]

\[
\phi_1 = 2 \tan^{-1} \left( \frac{q_1}{q_0} \right) \quad \text{[first rotation about $x$-axis]}
\]

\[
\phi_2 = 2 \tan^{-1} \left( \frac{q_2}{q_0} \right) \quad \text{[second rotation about $y$-axis]}
\]

Table E-12. Factorization into $x-z-x$ rotation sequence

\[
A = p_0 p_2 + p_1 p_3, \quad B = -p_0 p_3 + p_1 p_2
\]

\[
\phi_3 = \tan^{-1} \left( \frac{-A}{B} \right) \quad \text{[third rotation about $x$-axis]}
\]

\[
c_0 = \cos \frac{\phi_3}{2}, \quad c_1 = \sin \frac{\phi_3}{2}
\]

\[
q_0 = p_0 c_0 + p_1 c_1, \quad q_1 = p_1 c_0 - p_0 c_1, \quad q_2 = p_2 c_0 - p_3 c_1, \quad q_3 = p_3 c_0 + p_2 c_1
\]

\[
\phi_1 = 2 \tan^{-1} \left( \frac{q_1}{q_0} \right) \quad \text{[first rotation about $x$-axis]}
\]

\[
\phi_2 = 2 \tan^{-1} \left( \frac{q_3}{q_0} \right) \quad \text{[second rotation about $z$-axis]}
\]

The program in Listing E-1 is designed to test these formulas.

Listing E-1. factor.cpp

```
#include "Rotation.h"
#include <iostream>
#include <cstdlib>

int main( int argc, char* argv[] ) {
    va::Rotation R;
    rng::Random rng;
    va::ORDER order = va::ORDER( rng.uniformDiscrete( 0, 11 ) );
    double ang_1, ang_2, ang_3;
    if ( argc == 4 ) {
        ang_1 = va::rad( atof( argv[ 1 ] ) );
        ang_2 = va::rad( atof( argv[ 2 ] ) );
        ang_3 = va::rad( atof( argv[ 3 ] ) );
        R = va::Rotation( ang_1, ang_2, ang_3, order );
        std::cout << "order = " << order << std::endl;
    }
    return 0;
}
```
The command

```bash
./factor
```

will generate a random rotation, so each run will be different. But the factored rotation must match the randomly generated rotation, as shown here:

```
The rotation is 0.18777 0.958993 -0.212307 86.6526

The following rotations should match this
1st rotation = -24.8355 order = 0
2nd rotation = 84.2077
3rd rotation = 0.18777 0.958993 -0.212307 86.6526
```

Approved for public release; distribution is unlimited.
We can also input an explicit rotation:

```plaintext
[factor -13. 67. -23.]
```

This gives the following results:

```plaintext
<table>
<thead>
<tr>
<th>Order</th>
<th>Rotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.33597 0.863186 -0.376875 73.8475</td>
</tr>
<tr>
<td>2</td>
<td>-0.33597 0.863186 -0.376875 73.8475</td>
</tr>
<tr>
<td>3</td>
<td>-0.33597 0.863186 -0.376875 73.8475</td>
</tr>
</tbody>
</table>

The order variable is arbitrary so it is randomized in the program. It is output so that we can check that it found the same rotation sequence for that particular order (in this case, order = 1).
Appendix F. Conversion between Quaternion and Rotation Matrix
F-1. Quaternion to Rotation Matrix

For rotations about a principal axis, the correspondence is as follows:

- Rotation about the \( x \)-axis: \( \cos \frac{\theta}{2} + \hat{i} \sin \frac{\theta}{2} \) \( \Leftrightarrow \) \[
\begin{bmatrix}
1 & 0 & 0 \\
0 & \cos \theta & -\sin \theta \\
0 & \sin \theta & \cos \theta
\end{bmatrix}
\] (F-1)

- Rotation about the \( y \)-axis: \( \cos \frac{\theta}{2} + \hat{j} \sin \frac{\theta}{2} \) \( \Leftrightarrow \) \[
\begin{bmatrix}
\cos \theta & 0 & \sin \theta \\
0 & 1 & 0 \\
-\sin \theta & 0 & \cos \theta
\end{bmatrix}
\] (F-2)

- Rotation about the \( z \)-axis: \( \cos \frac{\theta}{2} + \hat{k} \sin \frac{\theta}{2} \) \( \Leftrightarrow \) \[
\begin{bmatrix}
\cos \theta & -\sin \theta & 0 \\
\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{bmatrix}
\] (F-3)

In general, if the quaternion is given by

\[
q = q_0 + q_1 \hat{i} + q_2 \hat{j} + q_3 \hat{k},
\] (F-4)

then the corresponding rotation matrix is

\[
A = \begin{bmatrix}
2q_0^2 - 1 + 2q_1^2 & 2q_1q_2 - 2q_0q_3 & 2q_1q_3 + 2q_0q_2 \\
2q_1q_2 + 2q_0q_3 & 2q_0^2 - 1 + 2q_2^2 & 2q_2q_3 - 2q_0q_1 \\
2q_1q_3 - 2q_0q_2 & 2q_2q_3 + 2q_0q_1 & 2q_0^2 - 1 + 2q_3^2
\end{bmatrix}.
\] (F-5)

F-2. Rotation Matrix to Quaternion

Let the rotation be given by the matrix

\[
A = \begin{bmatrix}
a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23} \\
a_{31} & a_{32} & a_{33}
\end{bmatrix}.
\] (F-6)

Then a vector along the axis of rotation is

\[
v = (a_{32} - a_{23}) \hat{i} + (a_{13} - a_{31}) \hat{j} + (a_{21} - a_{12}) \hat{k}.
\] (F-7)
If this vector turns out to be zero, then $A$ is the identity matrix. Otherwise, a unit vector along the axis of rotation is

$$\hat{u} = \frac{v}{\|v\|} = \frac{(a_{32} - a_{23}) \hat{i} + (a_{13} - a_{31}) \hat{j} + (a_{21} - a_{12}) \hat{k}}{\sqrt{(a_{32} - a_{23})^2 + (a_{13} - a_{31})^2 + (a_{21} - a_{12})^2}} \quad (F-8)$$

The rotation angle is

$$\theta = \cos^{-1} \left( \frac{a_{11} + a_{22} + a_{33} - 1}{2} \right). \quad (F-9)$$

And the corresponding quaternion is

$$q = \cos \frac{\theta}{2} + \hat{u} \sin \frac{\theta}{2}. \quad (F-10)$$

### F-3. Conversion between Rotation, Rotation Matrix, and Quaternion

The program in Listing F-1 will convert between the 3 different representations.

#### Listing F-1. convert.cpp

```cpp
#include "Rotation.h"
#include <iostream>
#include <cstdlib>
#include <iomanip>
using namespace va;

int main( int argc, char* argv[]) {
    // specify a Rotation from an axis vector and rotation angle
    Vector u = normalize( Vector( 1., 1., 1. ) );
    double th = rad( 120. );

    if ( argc == 5 ) {
        th = rad( atof( argv[4] ) );
    }

    std::cout << std::setprecision(6) << std::fixed << std::showpos;

    Rotation R( u, th );
    matrix A;
    quaternion q;
    std::cout << "Given the Rotation:" << std::endl;
    std::cout << R << std::endl << std::endl;

    // convert Rotation to a rotation matrix
    std::cout << "convert Rotation to rotation matrix:" << std::endl;
    A = to_matrix( R );
    std::cout << A << std::endl << std::endl;

    // convert a Rotation to a quaternion
    std::cout << "convert Rotation to quaternion:" << std::endl;
    q = to_quaternion( R );
    std::cout << q << std::endl << std::endl;

    std::cout << "Given the rotation matrix:" << std::endl;
    std::cout << A << std::endl << std::endl;

    // convert a rotation matrix to a Rotation
    std::cout << "convert a rotation matrix to a Rotation"
```

Approved for public release; distribution is unlimited.
```cpp
std::cout << "convert rotation matrix to Rotation:" << std::endl;
R = Rotation( A );
std::cout << R << std::endl << std::endl;

//convert a rotation matrix to a quaternion
std::cout << "convert rotation matrix to quaternion:" << std::endl;
q = to_quaternion( Rotation( A ) );
std::cout << q << std::endl;

std::cout << "Given the quaternion:" << std::endl;
std::cout << q << std::endl;

// convert a quaternion to a Rotation
std::cout << "convert quaternion to Rotation:" << std::endl;
R = Rotation( q );
std::cout << R << std::endl;

// convert a quaternion to a rotation matrix
std::cout << "convert quaternion to rotation matrix:" << std::endl;
A = to_matrix( Rotation( q ) );
std::cout << A << std::endl;
```

Compiling this program with
```
g++ -O2 -Wall -o convert convert.cpp -lm
```
and then running it via the command
```
./convert 2.35 6.17 -4.6 35.6
```
produces the following output:

```
Given the Rotation:
+0.292041 +0.766762 -0.571654 +35.600000

convert Rotation to rotation matrix:
+0.829041 +0.374624 +0.415148
-0.290921 +0.922983 -0.251926
-0.477552 +0.088081 +0.874177

convert Rotation to quaternion:
+0.952129 +0.089275 +0.234396 -0.174752

Given the rotation matrix:
+0.829041 +0.374624 +0.415148
-0.290921 +0.922983 -0.251926
-0.477552 +0.088081 +0.874177

convert rotation matrix to Rotation:
+0.952129 +0.089275 +0.234396 -0.174752

Given the quaternion:
+0.952129 +0.089275 +0.234396 -0.174752

convert quaternion to Rotation:
+0.952129 +0.089275 +0.234396 -0.174752

convert quaternion to rotation matrix:
+0.829041 +0.374624 +0.415148
-0.290921 +0.922983 -0.251926
-0.477552 +0.088081 +0.874177
```

Approved for public release; distribution is unlimited.
Appendix G. Slerp (Spherical Linear Interpolation)
This is a derivation of the spherical linear interpolation (Slerp) formula that was introduced by Shomake.\(^1\) As depicted in Fig. G-1, the rotation that takes the unit vector \(\hat{u}_1\) to the unit vector \(\hat{u}_2\) is the unit quaternion

\[
q \equiv \cos \frac{\theta}{2} + \hat{n} \sin \frac{\theta}{2}, \tag{G-1}
\]

where \(\theta\) is the angle between \(\hat{u}_1\) and \(\hat{u}_2\), and

\[
\hat{n} \equiv \frac{\hat{u}_1 \times \hat{u}_2}{\|\hat{u}_1 \times \hat{u}_2\|} \tag{G-2}
\]

is the unit vector along the axis of rotation. This means that

\[
\hat{u}_2 = q\hat{u}_1q^{-1}. \tag{G-3}
\]

Now let us parametrize the angle as \(t\theta\), where \(0 \leq t \leq 1\), and let

\[
q(t) \equiv \cos \frac{t\theta}{2} + \hat{n} \sin \frac{t\theta}{2}. \tag{G-4}
\]

Then an intermediate unit vector \(\hat{u}(t)\) that runs along the arc on the unit circle from \(\hat{u}_1\) to \(\hat{u}_2\) is given by

\[
\hat{u}(t) = q(t)\hat{u}_1q(t)^{-1} = \left(\cos \frac{t\theta}{2} + \hat{n} \sin \frac{t\theta}{2}\right)\hat{u}_1 \left(\cos \frac{t\theta}{2} - \hat{n} \sin \frac{t\theta}{2}\right). \tag{G-5}
\]

\[\text{Fig. G-1. Spherical linear interpolation over the unit sphere}\]

\(^{1}\)Shoemake K. Animating rotation with quaternion curves. SIGGRAPH '85; 1985;245–254.
Treating $\hat{u}_1$ as the pure quaternion $(0, \hat{u}_1)$ and using the fact that $\hat{u}_1 \cdot \hat{n} = 0$, $\hat{u}_2 \cdot \hat{n} = 0$, $\hat{n} \cdot (\hat{n} \times \hat{u}_1) = 0$, and $\|\hat{u}_1 \times \hat{u}_2\| = \sin \theta$, we can carry out the quaternion multiplication to get

\[
\hat{u}(t) = \left( \cos \frac{t\theta}{2} + \frac{\hat{n} \sin t\theta}{2} \right) \hat{u}_1 \left( \cos \frac{t\theta}{2} - \frac{\hat{n} \sin t\theta}{2} \right)
\]

\[
= \left( \cos \frac{t\theta}{2} \hat{u}_1 + \frac{\hat{n} \times \hat{u}_1 \sin t\theta}{2} \right) \left( \cos \frac{t\theta}{2} - \frac{\hat{n} \sin t\theta}{2} \right)
\]

\[
= \cos^2 \frac{t\theta}{2} \hat{u}_1 + \cos \frac{t\theta}{2} \sin \frac{t\theta}{2} \hat{n} \times \hat{u}_1 + \cos \frac{t\theta}{2} \sin \frac{t\theta}{2} \hat{n} \times \hat{u}_1 - \sin^2 \frac{t\theta}{2} (\hat{n} \times \hat{u}_1) \times \hat{n}
\]

\[
= \cos^2 \frac{t\theta}{2} \hat{u}_1 + 2 \cos \frac{t\theta}{2} \sin \frac{t\theta}{2} \hat{n} \times \hat{u}_1 + \sin^2 \frac{t\theta}{2} \hat{n} \times (\hat{n} \times \hat{u}_1). \tag{G-6}
\]

Now

\[
\hat{n} \times \hat{u}_1 = -\hat{u}_1 \times \hat{n} = -\frac{\hat{u}_1 \times (\hat{u}_1 \times \hat{u}_2)}{\sin \theta}
\]

\[
= -\frac{\hat{u}_1 (\hat{u}_1 \cdot \hat{u}_2) - \hat{u}_2 (\hat{u}_1 \cdot \hat{u}_1)}{\sin \theta}
\]

\[
= \frac{\hat{u}_2 - \cos \theta \hat{u}_1}{\sin \theta} \tag{G-7}
\]

and

\[
\hat{n} \times (\hat{n} \times \hat{u}_1) = \hat{n}(\hat{n} \cdot \hat{u}_1) - \hat{u}_1 (\hat{n} \cdot \hat{n}) = -\hat{u}_1 \tag{G-8}
\]

so that

\[
\hat{u}(t) = \left( \cos^2 \frac{t\theta}{2} - \sin^2 \frac{t\theta}{2} \right) \hat{u}_1 + 2 \cos \frac{t\theta}{2} \sin \frac{t\theta}{2} \left( \frac{\hat{u}_2 - \cos \theta \hat{u}_1}{\sin \theta} \right)
\]

\[
= \cos t\theta \hat{u}_1 + \sin t\theta \left( \frac{\hat{u}_2 - \cos \theta \hat{u}_1}{\sin \theta} \right)
\]

\[
= \frac{\cos t\theta \sin \theta \hat{u}_1 + \sin t\theta \hat{u}_2 - \sin t\theta \cos \theta \hat{u}_1}{\sin \theta}
\]

\[
= \frac{\sin(\theta - t\theta) \hat{u}_1 + \sin t\theta \hat{u}_2}{\sin \theta}
\]

\[
= \frac{\sin(1 - t)\theta}{\sin \theta} \hat{u}_1 + \frac{\sin t\theta}{\sin \theta} \hat{u}_2. \tag{G-9}
\]
G-1. Slerp Formula

Thus, the spherical linear interpolation of the unit vector on the arc of the unit sphere from \( \hat{u}_1 \) to \( \hat{u}_2 \) is given by

\[
\hat{u}(t) = \frac{\sin(1-t)t}{\sin \theta} \hat{u}_1 + \frac{\sin t \theta}{\sin \theta} \hat{u}_2,
\]

where \( 0 \leq t \leq 1 \). This gives us the C++ implementation in Listing G-1.

Listing G-1. slerp.cpp

```
// slerp.cpp: original slerp formula

#include "Vector.h"
#include <iostream>
#include <cstdlib>
using namespace va;

int main( int argc, char* argv[] ) {
    Vector i( 1., 0., 0. ), j( 0., 1., 0. ), k( 0., 0., 1. );

    Vector u1 = i; // default initial vector
    Vector u2 = j; // default final vector
    if ( argc > 1 ) { // or specify initial and final vectors on the command line
    }

    const int N = 1000;
    const double THETA = acos( u1 * u2 );
    const double A = 1. / sin( THETA );
    Vector u;

    double t;
    for( int n = 0; n < N; n++ ) {
        t = double( n ) / double( N-1 );
        u = A * ( sin( ( 1. - t ) * THETA ) * u1 + sin( t * THETA ) * u2 ) ;
        std::cout << u << std::endl;
    }
    return EXIT_SUCCESS;
}
```

G-2. Fast Incremental Slerp

The straightforward application of Eq. G-10, as we incrementally vary \( t \) from 0 to 1, involves the computationally expensive evaluation of trigonometric functions in an inner loop. We show here how this can be avoided.1

Starting with Eq. G-10, using the double angle formula, and

\[
\hat{u}_1 \cdot \hat{u}_2 = \cos \theta,
\]


Approved for public release; distribution is unlimited.
we have

\[
\hat{\mathbf{u}}(t) = \frac{[\sin \theta \cos(t\theta) - \cos \theta \sin(t\theta)] \hat{\mathbf{u}}_1 + \sin t\theta \hat{\mathbf{u}}_2}{\sin \theta}
\]

\[
= \cos(t\theta) \hat{\mathbf{u}}_1 + \frac{\sin(t\theta)}{\sin \theta} [\hat{\mathbf{u}}_2 - \cos \theta \hat{\mathbf{u}}_1]
\]

\[
= \cos(t\theta) \hat{\mathbf{u}}_1 + \sin(t\theta) \left[ \frac{\hat{\mathbf{u}}_2 - \cos \theta \hat{\mathbf{u}}_1}{\sqrt{1 - \cos^2 \theta}} \right]
\]

\[
= \cos(t\theta) \hat{\mathbf{u}}_1 + \sin(t\theta) \left[ \frac{\hat{\mathbf{u}}_2 - (\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2) \hat{\mathbf{u}}_1}{\sqrt{1 - (\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2)^2}} \right].
\] (G-12)

Now consider the term in square brackets. The numerator is \( \hat{\mathbf{u}}_2 \) minus the projection of \( \hat{\mathbf{u}}_2 \) onto \( \hat{\mathbf{u}}_1 \), and thus is orthogonal to \( \hat{\mathbf{u}}_1 \). Also, the denominator is the norm of the numerator, since

\[
[\hat{\mathbf{u}}_2 - (\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2) \hat{\mathbf{u}}_1] \cdot [\hat{\mathbf{u}}_2 - (\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2) \hat{\mathbf{u}}_1] = 1 - (\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2)^2 - (\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2)^2 + (\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2)^2
\]

\[
= 1 - (\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2)^2.
\] (G-13)

Thus, the term in square brackets is a fixed unit vector that is tangent to \( \hat{\mathbf{u}}_1 \), which we label \( \hat{\mathbf{u}}_0 \):

\[
\hat{\mathbf{u}}_0 \equiv \frac{\hat{\mathbf{u}}_2 - (\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2) \hat{\mathbf{u}}_1}{\sqrt{1 - (\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2)^2}}.
\] (G-14)

Therefore, Eq. G-12 can be written as

\[
\hat{\mathbf{u}}(t) = \cos(t\theta) \hat{\mathbf{u}}_1 + \sin(t\theta) \hat{\mathbf{u}}_0.
\] (G-15)

We want to evaluate \( \hat{\mathbf{u}} \) incrementally, so let us discretize this equation by setting \( \delta \theta = \theta/(N - 1) \) and let \( x = \delta \theta \). Then Eq. G-15 becomes

\[
\hat{\mathbf{u}}[n] = \cos(nx) \hat{\mathbf{u}}_1 + \sin(nx) \hat{\mathbf{u}}_0
\] (G-16)

for \( n = 0, 1, 2, \ldots, N - 1 \).

---

1Hast A, Barrera T, Bengtsson E. Shading by spherical linear interpolation using De Moivre’s formula. WSCG’03. 2003;Short Paper;57–60.

Approved for public release; distribution is unlimited.
Now we make use of the trigonometric identities

\[
\cos(n + 1)x + \cos(n - 1)x = 2 \cos nx \cos x,
\]
\[
\sin(n + 1)x + \sin(n - 1)x = 2 \sin nx \cos x.
\] (G-17)

Or, changing \( n \to n - 1 \) and rearranging,

\[
\cos nx = 2 \cos x \cos(n - 1)x - \cos(n - 2)x,
\]
\[
\sin nx = 2 \cos x \sin(n - 1)x - \sin(n - 2)x.
\] (G-18)

Substituting these into Eq. G-16 results in a simple recurrence relation:

\[
\hat{u}[n] = [2 \cos x \cos(n - 1)x - \cos(n - 2)x] \hat{u}_1 +
[2 \cos x \sin(n - 1)x - \sin(n - 2)x] \hat{u}_0
= 2 \cos x[\cos(n - 1)x \hat{u}_1 + \sin(n - 1)x \hat{u}_0] -
[\cos(n - 2)x \hat{u}_1 + \sin(n - 2)x \hat{u}_0]
= 2 \cos x \hat{u}[n - 1] - \hat{u}[n - 2].
\] (G-19)

It is also easy to evaluate the first 2 values directly from Eq. G-16:

\[
\hat{u}[0] = \hat{u}_1 \quad \text{and} \quad \hat{u}[1] = \cos x \hat{u}_1 + \sin x \hat{u}_0.
\] (G-20)

(G-21)

Putting this all together gives us the C++ implementation in Listing G-2.

Listing G-2. fast_slerp.cpp

```cpp
// fast_slerp.cpp: fast incremental slerp evaluates trig functions only once
// R. Saucier, June 2016

#include "Vector.h"
#include <iostream>
#include <cstdlib>
using namespace va;

int main( int argc, char* argv[] ) {
    Vector i( 1., 0., 0. ), j( 0., 1., 0. ), k( 0., 0., 1. );
    Vector u1 = i; // default initial vector
    Vector u2 = j; // default final vector
    if ( argc > 1 ) { // or specify initial and final vectors on the command line
    }
    const int N = 1000;
    const double U12 = u1 * u2;
    const double THETA = acos( U12 );
    const double TH = THETA / double( N-1 );
    const double C = 2. * cos( TH );
    // ...
Removing the trigonometric functions from the inner loop results in a speedup of about 12 times over the original slerp formula in Eqs. G-10 or G-16.
INTENTIONALLY LEFT BLANK.
Appendix H. Exact Solution to the Absolute Orientation Problem
This solution follows the approach of Micheals and Boult.\footnote{Micheals RJ, Boult TE. Increasing robustness in self-localization and pose estimation. [date unknown; accessed 2010 Jun]. http://www.vast.ucsd.edu/~tboult/PAPERS/SPIE99-Increasing-robustness-in-self-localization-and-pose-estimation-Micheals-Boult.pdf.} Given 2 sets of 3 linearly independent vectors, \(\{a_1, a_2, a_3\}\) and \(\{b_1, b_2, b_3\}\), where the 2 sets of vectors are not necessarily unit vectors but are related by a pure rotation, the absolute orientation problem is to find this rotation. Since rotations can be represented by unit quaternions, there must be a unit quaternion \(q\) such that

\[
b_i = qa_iq^{-1} \quad \text{for} \quad i = 1, 2, 3
\]

where \(q = q_0 + q_1i + q_2j + q_3k\) and \(q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1\). Using \(i^2 = j^2 = k^2 = ijk = -1\), expanding, we get

\[
b_x = a_xq_0^2 + 2a_xq_0q_2 - 2a_yq_1q_3 + a_xq_1^2 + 2a_yq_1q_2 + 2a_zq_1q_3 - a_xq_2^2 - a_xq_3^2 \quad \text{(H-2)}
\]

\[
b_y = a_yq_0^2 - 2a_xq_0q_1 + 2a_xq_0q_3 - a_yq_1^2 + 2a_xq_1q_2 + a_yq_2^2 + 2a_zq_2q_3 - a_yq_3^2 \quad \text{(H-3)}
\]

\[
b_z = a_zq_0^2 + 2a_yq_0q_1 - 2a_xq_0q_2 - a_zq_1^2 + 2a_xq_1q_3 - a_zq_2^2 + 2a_yq_2q_3 + a_zq_3^2 \quad \text{(H-4)}
\]

for each of the 3 vectors. Imposing the normalization condition then gives us 10 equations in 10 unknowns:

\[
\begin{bmatrix}
a_{1x}

a_{1y}

a_{1z}

a_{2x}

a_{2y}

a_{2z}

a_{3x}

a_{3y}

a_{3z}

1
\end{bmatrix}
\begin{bmatrix}
a_{1x}
a_{1y}
a_{1z}
a_{2x}
a_{2y}
a_{2z}
a_{3x}
a_{3y}
a_{3z}
1
\end{bmatrix}
= \begin{bmatrix}
q_0^2
q_0q_1
q_0q_2
q_0q_3
q_1
q_1q_2
q_1q_3
q_2
q_2q_3
q_3
\end{bmatrix}
\]

\[
\text{(H-5)}
\]

The 10 \times 10 coefficient matrix can be inverted with \textsc{Mathematica}. And we find that the solution for the 10 products of the 4 quaternion components can be expressed in terms of scalar triple products as follows:

\[
q_0^2 = \frac{\det(a_1, a_2, a_3) + \det(b_1, a_2, a_3) + \det(a_1, b_2, a_3) + \det(a_1, a_2, b_3)}{4\det(a_1, a_2, a_3)}
\]

\[
q_1^2 = \frac{\det(a_1, a_2, a_3) + \det(P_{11}b_1, a_2, a_3) + \det(a_1, P_{11}b_2, a_3) + \det(a_1, a_2, P_{11}b_3)}{4\det(a_1, a_2, a_3)}
\]

\[
q_2^2 = \frac{\det(a_1, a_2, a_3) + \det(P_{22}b_1, a_2, a_3) + \det(a_1, P_{22}b_2, a_3) + \det(a_1, a_2, P_{22}b_3)}{4\det(a_1, a_2, a_3)}
\]

\[
q_3^2 = \frac{\det(a_1, a_2, a_3) + \det(P_{33}b_1, a_2, a_3) + \det(a_1, P_{33}b_2, a_3) + \det(a_1, a_2, P_{33}b_3)}{4\det(a_1, a_2, a_3)}
\]

\[
q_iq_j = \frac{\det(P_{ij}b_1, a_2, a_3) + \det(a_1, P_{ij}b_2, a_3) + \det(a_1, a_2, P_{ij}b_3)}{4\det(a_1, a_2, a_3)}
\]

\[
\text{(H-10)}
\]
where

\[
\det(a, b, c) = \det \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{bmatrix}
= a_x(b_y c_z - b_z c_y) + a_y(b_z c_x - b_x c_z) + a_z(b_x c_y - b_y c_x)
= a \cdot (b \times c).
\]

(H-11)

and

\[
P_{11} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix},
P_{22} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix},
P_{33} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}
\]

(H-12)

\[
P_{01} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix},
P_{02} = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix},
P_{03} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\]

(H-13)

\[
P_{12} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix},
P_{13} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix},
P_{23} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}
\]

(H-14)

We set \( q_0 \) as the positive square root of Eq. H-6 and then use Eq. H-10 to get \( q_1 = q_0 q_1 / q_0 \), \( q_2 = q_0 q_2 / q_0 \), and \( q_3 = q_0 q_3 / q_0 \). The axis of rotation is along the unit vector

\[
\hat{u} = \frac{q_1 \hat{i} + q_2 \hat{j} + q_3 \hat{k}}{\sqrt{1 - q_0^2}},
\]

(H-15)

and the angle of rotation is

\[
\theta = 2 \cos^{-1} q_0.
\]

(H-16)

The full rotation matrix is

\[
R = \begin{bmatrix}
2q_0^2 - 1 + 2q_1^2 & 2q_1 q_2 - 2q_0 q_3 & 2q_1 q_3 + 2q_0 q_2 \\
2q_1 q_2 + 2q_0 q_3 & 2q_0^2 - 1 + 2q_2^2 & 2q_2 q_3 - 2q_0 q_1 \\
2q_1 q_3 - 2q_0 q_2 & 2q_2 q_3 + 2q_0 q_1 & 2q_0^2 - 1 + 2q_3^2
\end{bmatrix}
\]

(H-17)
The program in Listing H-1 is designed to test the implemented closed-form solution.

Listing H-1. ao.cpp

```cpp
#include "Rotation.h"
#include <iostream>
#include <cstdlib>
#include <iomanip>
using namespace va; // vector algebra namespace

int main( int argc, char* argv[] ) {
    Vector a1( 3.5, 1.0, 2.3 ), a2( 1.5, 2.1, 7.1 ), a3( 4.3, -5.8, 1.7 ); // 3 linearly independent vectors
    Vector b1, b2, b3;

double pitch = rad( 45. ); // default pitch (deg converted to radians)
double yaw = rad( -30. ); // default yaw (deg converted to radians)
double roll = rad( 60. ); // default roll (deg converted to radians)
if ( argc == 4 ) { // or specify pitch, yaw, roll (deg) on command line
    pitch = rad( atof( argv[1] ) );
yaw = rad( atof( argv[2] ) );
roll = rad( atof( argv[3] ) );
}

std::cout << std::setprecision(6) << std::fixed << std::showpos;
std::cout << "The 3 vectors are linearly independent iff det(a1,a2,a3) is non-zero: ";
std::cout << "det(a1,a2,a3) = " << ( a1 * ( a2 ^ a3 ) ) << std::endl << std::endl;

// perform rotation sequence on initial vectors
Rotation R1( pitch, yaw, roll, XYZ );
b1 = R1 * a1;
b2 = R1 * a2;
b3 = R1 * a3;

// output the rotated vectors
std::cout << "The rotated vectors are:" << std::endl;
std::cout << "b1 = " << b1 << std::endl;
std::cout << "b2 = " << b2 << std::endl;
std::cout << "b3 = " << b3 << std::endl << std::endl;

// given only the two sets of vectors, find the rotation that takes {a1,a2,a3} to {b1,b2,b3}
Rotation R2( a1, a2, a3, b1, b2, b3 );

// apply this rotation to the original vectors
b1 = R2 * a1;
b2 = R2 * a2;
b3 = R2 * a3;

// output rotated vectors to show they match previous output
std::cout << "The following vectors should match those above: " << std::endl;
std::cout << "b1 = " << b1 << std::endl;
std::cout << "b2 = " << b2 << std::endl;
std::cout << "b3 = " << b3 << std::endl;

// factor this rotation into a pitch-yaw-roll rotation sequence
sequence s = factor( R2, XYZ );

// output rotation sequence to show it matches the input values
std::cout << "Factoring this rotation into a rotation sequence gives: " << std::endl;
std::cout << "pitch = " << deg( s.first ) << std::endl;
std::cout << "yaw = " << deg( s.second ) << std::endl;
std::cout << "roll = " << deg( s.third ) << std::endl;
return EXIT_SUCCESS;
}
```

Compiling this program with

```
g++ -O2 -Wall -o ao ao.cpp -lm
```

and then running it with the command

```
./ao -35.2 43.5 -75.6
```

Approved for public release; distribution is unlimited.
The 3 vectors need not be orthonormal—nor even mutually orthogonal—but they must be linearly independent. A necessary and sufficient condition for this is $\det(a_1, a_2, a_3) \neq 0$. We see that is the case from line 1. Lines 4–6 show the effect of the given rotation sequence upon the original 3 vectors (see lines 28–38 of Listing H-1). The program then computes the rotation that will take the original vectors to these 3 vectors (line 41 of Listing H-1) and then applies it to the original vectors (lines 43–52 of Listing H-1). We see on lines 9–11 that these do indeed match lines 4–6. Finally, the program factors the computed rotation into a pitch-yaw-roll rotation sequence (line 55 of Listing H-1) and the lines 14–16 show that we retrieve the input values. Thus we verify that the program is able to find the rotation as long as the original vectors are linearly independent.
1  DEFENSE TECHNICAL INFORMATION CTR
(PDF)  DTIC OCA

2  DIRECTOR
(PDF)  US ARMY RESEARCH LAB
       RDRL CIO L
       IMAL HRA MAIL & RECORDS MGMT

1  GOVT PRINTG OFC
(PDF)  A MALHOTRA

ABERDEEN PROVING GROUND

12  RDRL SLB D
(PDF,  J COLLINS
1 HC)  RDRL SLB G
       J ABELL
       D CARABETTA
       T MALLORY
       RDRL SLB S
       J AUTEN
       R DIBELKA
       C HUNT
       J HUNT
       S MORRISON
       R SAUCIER (1 HC)
       G SAUERBORN

Approved for public release; distribution is unlimited.

87