



ARL-TR-7894 • DEC 2016



US Army Research Laboratory

Vectors and Rotations in 3-Dimensions: Vector Algebra for the C++ Programmer

by Richard Saucier

Approved for public release; distribution is unlimited.

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

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) December 2016		2. REPORT TYPE Technical Report		3. DATES COVERED (From - To) January 2016–July 2016	
4. TITLE AND SUBTITLE Vectors and Rotations in 3-Dimensions: Vector Algebra for the C++ Programmer				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Richard Saucier				5d. PROJECT NUMBER AH80	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) US Army Research Laboratory ATTN: RDRL-SLB-S Aberdeen Proving Ground, MD 21005-5068				8. PERFORMING ORGANIZATION REPORT NUMBER ARL-TR-7894	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT 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.					
15. SUBJECT TERMS vector, rotation, 3D, quaternion, C++ tools, rotation sequence, Euler angles, yaw, pitch, roll, orientation					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 98	19a. NAME OF RESPONSIBLE PERSON Richard Saucier
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified			19b. TELEPHONE NUMBER (Include area code) 410-278-6721

Contents

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

List of Figures

Fig. 1	Initial orientation of the target plate in a right-handed Cartesian coordinate system.....	7
Fig. 2	Description of pitch, yaw, and roll.	8
Fig. 3	Geometry for oblique shots.....	14
Fig. 4	Plots of the 30° (half-angle) spall cone on the witness plate for a range of deflections.....	19
Fig. G-1	Spherical linear interpolation over the unit sphere	74

List of Tables

Table A-1	Vector: A C++ Class for 3D Vector Algebra—Reference Sheet	27
Table B-1	Rotation: A C++ Class for 3D Rotations—Reference Sheet	42
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	55
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	55
Table E-3	Factorization into y - x - z rotation sequence	56
Table E-4	Factorization into z - x - y rotation sequence	56
Table E-5	Factorization into x - z - y rotation sequence	57
Table E-6	Factorization into y - z - x rotation sequence	57
Table E-7	Factorization into z - y - x rotation sequence	60
Table E-8	Factorization into z - x - z rotation sequence	60
Table E-9	Factorization into y - z - y rotation sequence	61
Table E-10	Factorization into y - x - y rotation sequence	61
Table E-11	Factorization into x - y - x rotation sequence	61
Table E-12	Factorization into x - z - x rotation sequence	62

List of Listings

Listing 1	vtest.cpp	3
Listing 2	rtest.cpp	5
Listing 3	geometry.cpp	10
Listing 4	r2.cpp	11
Listing 5	oblique.cpp	17
Listing A-1	Vector.h	22
Listing B-1	Rotation.h	30
Listing D-1	order.cpp	51
Listing E-1	factor.cpp	65
Listing F-1	convert.cpp	71
Listing G-1	slerp.cpp	76
Listing G-2	fast_slerp.cpp	78
Listing H-1	ao.cpp	84

INTENTIONALLY LEFT BLANK.

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

```
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

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

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

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

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

```
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 formulas¹ 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`

```
1 // vtest.cpp: simple program to demonstrate basic usage of Vector class
2
3 #include "Vector.h" // only need to include this header file
4 #include <iostream>
5 #include <cstdlib>
6 using namespace va; // vector algebra namespace
7
8 int main( void ) {
9
10     // let u be a unit vector that has equal components along all 3 axes
11     Vector u = normalize( Vector( 1., 1., 1. ) );
12
13     // output the vector
14     std::cout << "u = " << u << std::endl;
15
16     // show that the magnitude is 1
17     std::cout << "magnitude = " << u.r() << std::endl;
18
19     // output the polar angle in degrees
20     std::cout << "polar angle (deg) = " << deg( u.theta() ) << std::endl;
21
22     // output the azimuthal angle in degrees
23     std::cout << "azimuthal angle (deg) = " << deg( u.phi() ) << std::endl;
24
25     // output the direction cosines
26     std::cout << "direction cosines = " << u.dircos( X ) << " " << u.dircos( Y ) << " " << u.dircos( Z ) << std::endl;
27
28     // let ihat, jhat, khat be unit vectors along x-axis, y-axis, and z-axis, respectively
29     Vector ihat( 1., 0., 0. ), jhat( 0., 1., 0. ), khat( 0., 0., 1. );
30
31     // output the vectors
32     std::cout << "ihat = " << ihat << std::endl;
33     std::cout << "jhat = " << jhat << std::endl;
34     std::cout << "khat = " << khat << std::endl;
35
36     // rotate ihat, jhat, khat about u by 120 degrees
37     ihat.rotate( u, rad( 120. ) );
38     jhat.rotate( u, rad( 120. ) );
39     khat.rotate( u, rad( 120. ) );
40
41     // output the rotated vectors
42     std::cout << "after 120 deg rotation about u:" << std::endl;
43     std::cout << "ihat is now = " << ihat << std::endl;
44     std::cout << "jhat is now = " << jhat << std::endl;
45     std::cout << "khat is now = " << khat << std::endl;
46
47     // define two vectors, a and b
48     Vector a( 2., 1., -1. ), b( 3., -4., 1. );
49     std::cout << "a = " << a << std::endl;
50     std::cout << "b = " << b << std::endl;
51     std::cout << "a + b = " << a + b << std::endl;
52     std::cout << "a - b = " << a - b << std::endl;
53
54     // compute and output the dot product
55     double s = a * b;
56     std::cout << "dot product, a * b = " << s << std::endl;
57
58     // compute and output the cross product
59     Vector c = a ^ b;
60     std::cout << "cross product, a ^ b = " << c << std::endl;
61
62     // output the angle (deg) between a and b
63     std::cout << "angle between a and b (deg) = " << deg( angle( a, b ) ) << std::endl;
64
65     // compute and output the projection of a along b
66     std::cout << "proj( a, b ) = " << proj( a, b ) << std::endl;
67
68     // rotate a and b 120 deg about u
69     a.rotate( u, rad( 120. ) );
70     b.rotate( u, rad( 120. ) );
71     std::cout << "after rotating a and b 120 deg about u: " << a << std::endl;
```

```

72     std::cout << "a is now = " << a << std::endl;
73     std::cout << "b is now = " << b << std::endl;
74
75     // output the angle (deg) between a and b
76     std::cout << "angle between a and b (deg) is now = " << deg( angle( a, b ) ) << std::endl;
77
78     // compute and output the dot product
79     s = a * b;
80     std::cout << "dot product, a * b is now = " << s << std::endl;
81
82     // compute and output the cross product
83     c = a ^ b;
84     std::cout << "cross product, a ^ b is now = " << c << std::endl;
85
86     // set a and b to their original values and compute the cross product
87     a = Vector( 2., 1., -1. );
88     b = Vector( 3., -4., 1. );
89     c = a ^ b;
90     std::cout << "original cross product, c = " << c << std::endl;
91
92     // rotate c 120 deg about u and output
93     c.rotate( u, rad( 120. ) );
94     std::cout << "now rotate c 120 deg about u:" << std::endl;
95     std::cout << "c is now = " << c << std::endl;
96
97     return EXIT_SUCCESS;
98 }

```

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

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

Running it

```
./vtest
```

will print the following:

```

1  u = 0.57735 0.57735 0.57735
2  magnitude = 1
3  polar angle (deg) = 54.7356
4  azimuthal angle (deg) = 45
5  direction cosines = 0.57735 0.57735 0.57735
6  ihat = 1 0 0
7  jhat = 0 1 0
8  khat = 0 0 1
9  after 120 deg rotation about u:
10 ihat is now = 0 1 0
11 jhat is now = 0 0 1
12 khat is now = 1 0 0
13 a = 2 1 -1
14 b = 3 -4 1
15 a + b = 5 -3 0
16 a - b = -1 5 -2
17 dot product, a * b = 1
18 cross product, a ^ b = -3 -5 -11
19 angle between a and b (deg) = 85.4078
20 proj( a, b ) = 0.115385 -0.153846 0.0384615
21 after rotating a and b 120 deg about u: -1 2 1
22 a is now = -1 2 1
23 b is now = 1 3 -4
24 angle between a and b (deg) is now = 85.4078
25 dot product, a * b is now = 1
26 cross product, a ^ b is now = -11 -3 -5
27 original cross product, c = -3 -5 -11
28 now rotate c 120 deg about u:
29 c is now = -11 -3 -5

```

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`

```
1 // rtest.cpp
2
3 #include "Rotation.h"
4 #include <iostream>
5 #include <cstdlib>
6 #include <cmath>
7 #include <iomanip>
8 using namespace va;
9
10 int main( void ) {
11
12     // declare two unit vectors, u and v
13     Vector u = Vector( 1.5, 2.1, 3.2 ).unit(), v = Vector( 1.2, 3.5, -2.3 ).unit();
14     std::cout << "u      = " << u << std::endl;
15     std::cout << "v      = " << v << std::endl;
16
17     // let R be the rotation specified by the vector cross product u ^ v
18     Rotation R( u, v );
19
20     // show that R * u equals v
21     std::cout << "R * u    = " << R * u << std::endl;
22
23     // interpolate a unit vector, w, halfway between u and v
24     Vector w = slerp( u, v, 0.5 );
25     std::cout << "Vector halfway between u and v = " << w << std::endl;
26
27     // output angle between u and v
28     std::cout << "angle between u and v (deg) = " << deg( angle( u, v ) ) << std::endl;
29
30     // output angle between u and w
31     std::cout << "angle between u and w (deg) = " << deg( angle( u, w ) ) << std::endl;
32
33     // let Rinv be the inverse rotation
34     Rotation Rinv = inverse( R );
35
36     // show that Rinv * v equals u
37     std::cout << "Rinv * v = " << Rinv * v << std::endl;
38
39     Vector ihat( 1., 0., 0. ), jhat( 0., 1., 0. ), khat( 0., 0., 1. );
40     R = Rotation( ihat, jhat, khat, jhat, khat, ihat );
41     std::cout << "R = " << R << std::endl;
42
43     sequence s = factor( R, ZYX );
44     std::cout << "yaw (deg)   = " << deg( s.first ) << std::endl;
45     std::cout << "pitch (deg)  = " << deg( s.second ) << std::endl;
46     std::cout << "roll (deg)   = " << deg( s.third ) << std::endl;
47
48     R = Rotation( s.first, s.second, s.third, ZYX );
49     std::cout << "R = " << R << std::endl;
50
51     s = factor( R, XYZ );
52     std::cout << "pitch (deg) = " << deg( s.first ) << std::endl;
53     std::cout << "yaw (deg)   = " << deg( s.second ) << std::endl;
54     std::cout << "roll (deg)   = " << deg( s.third ) << std::endl;
55
56     std::streamsize ss = std::cout.precision();
57     R = Rotation( s.first, s.second, s.third, XYZ );
58     std::cout << "R = " << R << std::endl;
59
60     rng::Random rng;
61     R = Rotation( rng );
62     std::cout << "R = " << R << std::endl;
63     quaternion q = to_quaternion( R );
64     std::cout << "the quaternion for this rotation is" << std::endl;
65     std::cout << "q = " << q << std::endl;
66     R = Rotation( q );
67     std::cout << "R = " << R << std::endl;
68     matrix A = to_matrix( R );
69     std::cout << "the matrix for this rotation is" << std::endl;
70     std::cout << std::setprecision(6) << std::fixed << std::showpos;
71     std::cout << A << std::endl;
```

```

72
73     std::cout << std::setprecision(ss) << std::noshowpos;
74     R = Rotation( A );
75     std::cout << "R = " << R << std::endl;
76
77     u = normalize( Vector( 1., 1., 1. ) );
78     double th = rad( 120. );
79     R = Rotation( u, th );
80     std::cout << R << std::endl;
81     A = to_matrix( A );
82     std::cout << A << std::endl;
83     q = to_quaternion( R );
84     std::cout << q << std::endl;
85
86     return EXIT_SUCCESS;
87 }

```

Writing this to a file `rtest.cpp`, compiling and running it,

```

g++ -O2 -Wall -o rtest rtest.cpp -lm
./rtest

```

produces the following output:

```

1  u      = 0.364878 0.510829 0.778407
2  v      = 0.275444 0.803378 -0.527934
3  R * u   = 0.275444 0.803378 -0.527934
4  Vector halfway between u and v = 0.431716 0.886061 0.168873
5  angle between u and v (deg) = 84.264
6  angle between u and w (deg) = 42.132
7  Rinv * v = 0.364878 0.510829 0.778407
8  R = 0.57735 0.57735 0.57735    120
9  yaw (deg) = -90
10 pitch (deg) = -180
11 roll (deg) = -90
12 R = 0.57735 0.57735 0.57735    120
13 pitch (deg) = 45
14 yaw (deg) = 90
15 roll (deg) = 45
16 R = 0.57735 0.57735 0.57735    120
17 R = 0.338694 0.929155 0.148182 174.087
18 the quaternion for this rotation is
19 q = 0.0515815 0.338243 0.927918 0.147985
20 R = 0.338694 0.929155 0.148182 174.087
21 the matrix for this rotation is
22 -0.765862    +0.612457    +0.195837
23 +0.642990    +0.727384    +0.239741
24 +0.004383    +0.309530    -0.950880
25 R = 0.338694 0.929155 0.148182 174.086566

```

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 $\mathbf{r}_c$. 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.

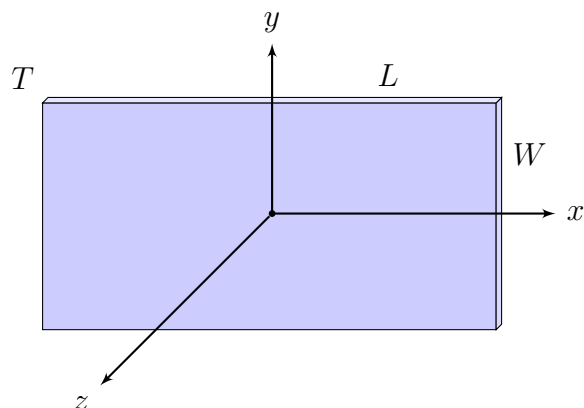


Fig. 1. Initial orientation of the target plate in a right-handed Cartesian coordinate system

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

z -axis, as shown in Fig. 2.

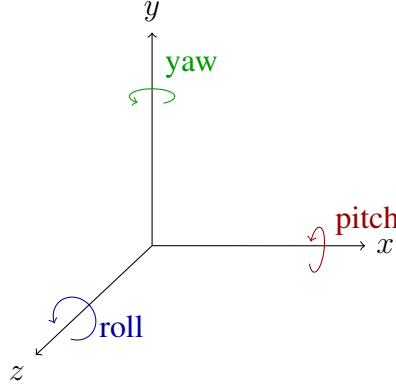


Fig. 2. Description of pitch, yaw, and roll

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

$$R \equiv R_{\hat{\mathbf{k}}''}(\phi_r)R_{\hat{\mathbf{j}}'}(\phi_y)R_{\hat{\mathbf{i}}}(\phi_p), \quad (1)$$

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[†], and since it is more efficient to rotate about fixed axes, we have

$$R = R_{\hat{\mathbf{k}}''}(\phi_r)R_{\hat{\mathbf{j}}'}(\phi_y)R_{\hat{\mathbf{i}}}(\phi_p) = R_{\hat{\mathbf{i}}}(\phi_p)R_{\hat{\mathbf{j}}}(\phi_y)R_{\hat{\mathbf{k}}}(\phi_r). \quad (2)$$

Hence, the normal vector is given by

$$\hat{\mathbf{n}} = R_{\hat{\mathbf{i}}}(\phi_p)R_{\hat{\mathbf{j}}}(\phi_y)R_{\hat{\mathbf{k}}}(\phi_r)\hat{\mathbf{k}}. \quad (3)$$

Once the target plate is oriented, we specify the location of its center with the vector $\mathbf{r}_c$. So the final position and orientation of the target plate is specified by the pair of vectors $(\mathbf{r}_c, \hat{\mathbf{n}})$.

*There are a total of 12 different conventions in the Rotation class that one may choose from.

[†]See Appendix D for a proof.

Let $\mathbf{r}_0$ be the origin of the fragment ray and let $\hat{\mathbf{u}}$ be a unit vector along the ray. We specify the orientation of $\hat{\mathbf{u}}$ by specifying the polar angle θ and azimuth angle ϕ , so that

$$\hat{\mathbf{u}} = \sin \theta \cos \phi \hat{\mathbf{i}} + \sin \theta \sin \phi \hat{\mathbf{j}} + \cos \theta \hat{\mathbf{k}}, \quad (4)$$

and the equation for the fragment ray is

$$\mathbf{r} = \mathbf{r}_0 + t\hat{\mathbf{u}}, \quad (5)$$

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 $\mathbf{r}$ such that

$$(\mathbf{r} - \mathbf{r}_c) \cdot \hat{\mathbf{n}} = 0, \quad (6)$$

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

$$(\mathbf{r}_0 + t\hat{\mathbf{u}} - \mathbf{r}_c) \cdot \hat{\mathbf{n}} = 0. \quad (7)$$

Solving for the distance t gives

$$t = \frac{(\mathbf{r}_c - \mathbf{r}_0) \cdot \hat{\mathbf{n}}}{\hat{\mathbf{u}} \cdot \hat{\mathbf{n}}}. \quad (8)$$

There will be an intersection, provided $\hat{\mathbf{u}} \cdot \hat{\mathbf{n}} \neq 0$, and the intersection point is

$$\boxed{\mathbf{r} = \mathbf{r}_0 + \frac{(\mathbf{r}_c - \mathbf{r}_0) \cdot \hat{\mathbf{n}}}{\hat{\mathbf{u}} \cdot \hat{\mathbf{n}}} \hat{\mathbf{u}}}. \quad (9)$$

The impact obliquity is given by

$$\theta_{\text{obl}} = \cos^{-1}(|\hat{\mathbf{u}} \cdot \hat{\mathbf{n}}|). \quad (10)$$

The location of the impact point with respect to the target center is

$$\mathbf{r} - \mathbf{r}_c = \mathbf{r}_0 - \mathbf{r}_c + \frac{(\mathbf{r}_c - \mathbf{r}_0) \cdot \hat{\mathbf{n}}}{\hat{\mathbf{u}} \cdot \hat{\mathbf{n}}} \hat{\mathbf{u}} \quad (11)$$

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 $\mathbf{r} - \mathbf{r}_c$ and check to see

if the resulting vector lies within the plate dimensions. Thus, let

$$\mathbf{d} = R^{-1}(\mathbf{r} - \mathbf{r}_c) \quad (12)$$

and then we check to see if

$$-L/2 \leq d_x \leq L/2 \quad \text{and} \quad -W/2 \leq d_y \leq W/2. \quad (13)$$

Listing 3 is an implementation of these vector equations.

Listing 3. geometry.cpp

```

1 // geometry.cpp: compute hit point on an oriented target plate from a general fragment ray
2
3 #include "Rotation.h"
4 #include <iostream>
5 #include <cstdlib>
6 #include <cmath>
7 #include <iomanip>
8 #include <cassert>
9 using namespace std;
10
11 int main( void ) {
12
13     // target dimensions (in)
14     const double L = 96., W = 48., L_2 = L / 2., W_2 = W / 2.;
15
16     // constant unit vectors for the laboratory frame
17     const va::Vector I( 1., 0., 0 ), J( 0., 1., 0 ), K( 0., 0., 1 );
18
19     std::cout << std::setprecision(3) << std::fixed;
20
21     va::Vector n = K;           // normal vector is initially along K
22     va::Vector u;               // unit vector along fragment ray
23     va::Vector rc( 0., 0., -24. ); // location of the target center
24     va::Vector r0( 0., 0., 0. ); // origin of the fragment ray
25
26     // specify pitch, yaw and roll of the target (degrees converted to radians)
27     double pitch = va::rad( +30. ); // pitch (converted to rad)
28     double yaw   = va::rad( -35. ); // yaw (converted to rad)
29     double roll  = va::rad( +45. ); // roll (converted to rad)
30
31     // specify the target orientation with R and inverse with Rinv
32     va::Rotation R( pitch, yaw, roll, va::XYZ );
33     va::Rotation Rinv = va::inverse( R );
34
35     // apply the orientation to the target normal
36     n = R * n;
37
38     double az = -35., el = 15.; // specify azimuth and elevation of fragment ray (deg)
39     assert( -360. <= az && az <= 360. );
40     assert( 0. <= el && el <= 90. );
41     double th = va::rad( 180. - el ); // convert to polar angle (rad)
42     double ph = va::rad( az );       // convert to azimuthal angle (rad)
43
44     // specify direction of u vector
45     u = sin( th ) * cos( ph ) * I + sin( th ) * sin( ph ) * J + cos( th ) * K;
46
47     double obl = angle( u, -n ); // obliquity angle (rad)
48     if ( obl >= M_PI_2 ) {
49         cout << "ray missed target since obl (deg) = " << obl * va::R2D << endl;
50         exit( EXIT_SUCCESS );
51     }
52
53     // compute distance to (infinite) target plane
54     double t = ( rc - r0 ) * n / ( u * n );
55     va::Vector r = r0 + t * u; // hit point in lab frame
56     va::Vector d = r - rc;     // hit point relative to the target center
57     d = Rinv * d;             // inverse rotation back to the laboratory reference frame
58
59     double xhit = d.x(), yhit = d.y();
60
61     // check if hit point on plane lies with target plate
62     if ( ( -L_2 <= xhit && xhit <= L_2 ) && ( -W_2 <= yhit && yhit <= W_2 ) ) {

```

```

63     cout << "ray hit target at " << d << " with respect to the target at the origin" << endl;
64     cout << "obl (deg) = " << obl * va::R2D << endl;
65     cout << "distance = " << t << endl;
66 }
67 else {
68     cout << "ray missed target plate" << endl;
69     cout << "xhit = " << xhit << endl;
70     cout << "yhit = " << yhit << endl;
71 }
72
73 return EXIT_SUCCESS;
74 }

```

Compiling and running this program gives the following output:

```

1 ray hit target at 2.794 -5.560 0.000 with respect to the target at the origin
2 obl (deg) = 41.752
3 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

$$\hat{i}' = R\hat{i}, \quad \hat{j}' = R\hat{j}, \quad \text{and} \quad \hat{k}' = R\hat{k}. \quad (14)$$

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

```

1 // r2.cpp: find the rotation, given two sets of vectors related by a pure rotation
2
3 #include "Rotation.h"
4 #include <iostream>
5 #include <cstdlib>
6 #include <cmath>
7 #include <iomanip>
8 #include <cassert>
9 using namespace std;
10
11 int main( int argc, char* argv[] ) {
12
13     // constant unit vectors for the laboratory frame
14     const va::Vector i( 1., 0., 0 ), j( 0., 1., 0 ), k( 0., 0., 1 );
15
16     double p = 0., y = 0., r = 0.;
17     if ( argc == 4 ) {
18
19         p = atof( argv[1] );
20         y = atof( argv[2] );
21         r = atof( argv[3] );
22     }
23
24     std::cout << std::setprecision(3) << std::fixed;
25
26     // specify pitch, yaw and roll of the target (degrees converted to radians)

```

```

27     double pitch = va::rad( p );
28     double yaw   = va::rad( y );
29     double roll  = va::rad( r );
30
31     // specify the rotation
32     va::Rotation R( pitch, yaw, roll, va::XYZ );
33
34     // apply the rotation to the basis vectors
35     va::Vector ip = R * i;
36     va::Vector jp = R * j;
37     va::Vector kp = R * k;
38
39     cout << "i = " << i << endl;
40     cout << "j = " << j << endl;
41     cout << "k = " << k << endl;
42
43     cout << "ip = " << ip << endl;
44     cout << "jp = " << jp << endl;
45     cout << "kp = " << kp << endl;
46
47     cout << endl << "Now we find the rotation" << endl;
48
49     va::Rotation R1( i, j, k, ip, jp, kp );
50
51     cout << "Applying the found rotation to i, j, k gives" << endl;
52
53     ip = R1 * i;
54     jp = R1 * j;
55     kp = R1 * k;
56
57     cout << "ip = " << ip << endl;
58     cout << "jp = " << jp << endl;
59     cout << "kp = " << kp << endl;
60
61     cout << endl << "Now suppose we want to factor this rotation into a FATEPEN sequence" << endl;
62
63     va::sequence s = va::factor( R1, va::XYZ );
64
65     cout << "This gives:" << endl;
66     cout << va::deg( s.first ) << endl;
67     cout << va::deg( s.second ) << endl;
68     cout << va::deg( s.third ) << endl;
69
70     cout << endl << "Now for something completely different: Start with the non-unit vectors" << endl;
71
72     va::Vector a( 1., 2., 3. ), b( -1., 2., 4. ), c( 4., 3., 9. );
73     va::Vector ap, bp, cp;
74
75     cout << "a = " << a << endl;
76     cout << "b = " << b << endl;
77     cout << "c = " << c << endl;
78
79     ap = R * a;
80     bp = R * b;
81     cp = R * c;
82
83     cout << "ap = " << ap << endl;
84     cout << "bp = " << bp << endl;
85     cout << "cp = " << cp << endl;
86
87     cout << endl << "Now we find the rotation that takes (a,b,c) to (ap,bp,cp)" << endl;
88
89     va::Rotation R2( a, b, c, ap, bp, cp );
90
91     cout << "Applying the found rotation to a, b, c gives" << endl;
92
93     ap = R2 * a;
94     bp = R2 * b;
95     cp = R2 * c;
96
97     cout << "ap = " << ap << endl;
98     cout << "bp = " << bp << endl;
99     cout << "cp = " << cp << endl;
100
101     cout << endl << "Also, suppose we want to factor this rotation into a FATEPEN sequence" << endl;
102
103     s = va::factor( R2, va::XYZ );
104
105     cout << "This gives:" << endl;
106     cout << va::deg( s.first ) << endl;
107     cout << va::deg( s.second ) << endl;
108     cout << va::deg( s.third ) << endl;
109
110     return EXIT_SUCCESS;
111 }

```

Approved for public release; distribution is unlimited.

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:

```
1 i = 1.000 0.000 0.000
2 j = 0.000 1.000 0.000
3 k = 0.000 0.000 1.000
4 ip = 0.683 -0.462 0.566
5 jp = -0.183 0.641 0.745
6 kp = -0.707 -0.612 0.354
7
8 Now we find the rotation
9 Applying the found rotation to i, j, k gives
10 ip = 0.683 -0.462 0.566
11 jp = -0.183 0.641 0.745
12 kp = -0.707 -0.612 0.354
13
14 Now suppose we want to factor this rotation into a FATEPEN sequence
15 This gives:
16 60.000
17 -45.000
18 15.000
19
20 Now for something completely different: Start with the non-unit vectors
21 a = 1.000 2.000 3.000
22 b = -1.000 2.000 4.000
23 c = 4.000 3.000 9.000
24 ap = -1.804 -1.016 3.116
25 bp = -3.877 -0.704 2.339
26 cp = -4.181 -5.435 7.680
27
28 Now we find the rotation that takes (a,b,c) to (ap,bp,cp)
29 Applying the found rotation to a, b, c gives
30 ap = -1.804 -1.016 3.116
31 bp = -3.877 -0.704 2.339
32 cp = -4.181 -5.435 7.680
33
34 Also, suppose we want to factor this rotation into a FATEPEN sequence
35 This gives:
36 60.000
37 -45.000
38 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 α 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.

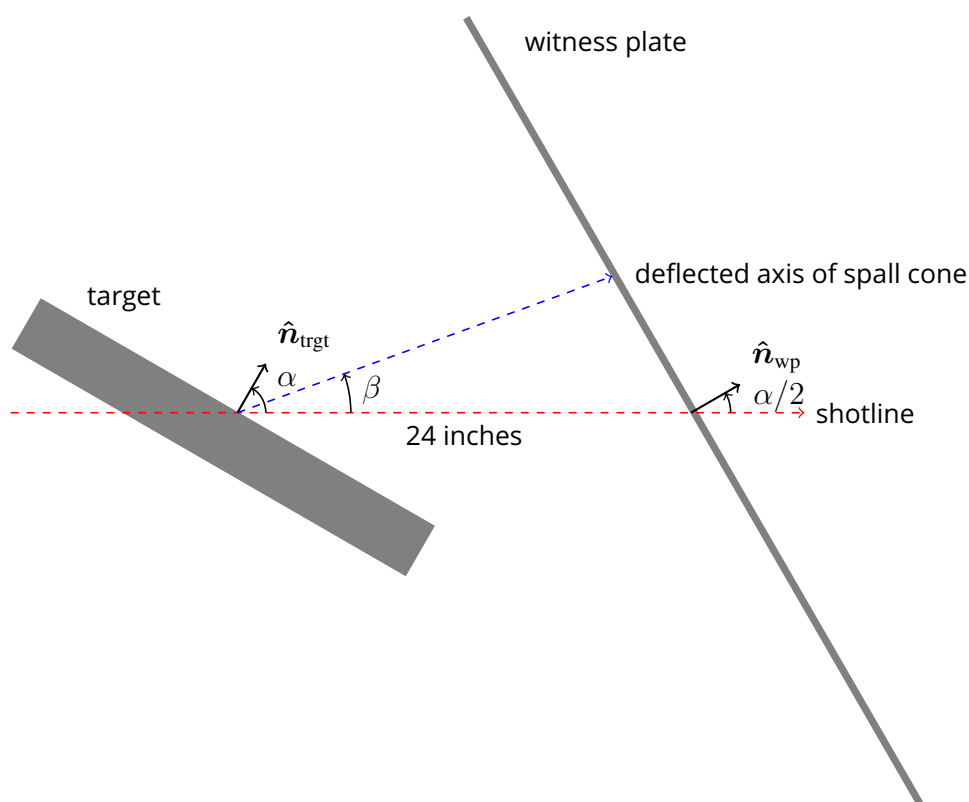


Fig. 3. Geometry for oblique shots

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 $\hat{\mathbf{u}}_{\text{sl}}$ be a unit vector along the *initial* shotline. In our case, $\hat{\mathbf{u}}_{\text{sl}} = -\hat{\mathbf{k}}$, 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.* 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^{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

$$R_{\text{trgt}} = R_{\hat{\mathbf{i}}}(\phi_p)R_{\hat{\mathbf{j}}}(\phi_y)R_{\hat{\mathbf{k}}}(\phi_r), \quad (15)$$

where the order goes from right to left and is reversed since we rotate about fixed axes, not body axes.* Since the initial orientation is $-\hat{\mathbf{k}}$, the final orientation is

$$\hat{\mathbf{n}}_{\text{trgt}} = R_{\text{trgt}}(-\hat{\mathbf{k}}). \quad (16)$$

The target obliquity is labeled α , and

$$\alpha \equiv \theta_{\text{trgt}} = \cos^{-1}(|\hat{\mathbf{u}}_{\text{sl}} \cdot \hat{\mathbf{n}}_{\text{trgt}}|). \quad (17)$$

Next we want to orient the witness plate to be at half the obliquity angle of the target.

*Given an initial and a final orientation, $\hat{\mathbf{n}}_1$ and $\hat{\mathbf{n}}_2$, respectively, the Rotation operator that performs this rotation is $R_{\hat{\mathbf{n}}}(\theta)$, where

$$\hat{\mathbf{n}} = \frac{\hat{\mathbf{n}}_1 \times \hat{\mathbf{n}}_2}{\|\hat{\mathbf{n}}_1 \times \hat{\mathbf{n}}_2\|} \quad \text{and} \quad \theta = \cos^{-1}(|\hat{\mathbf{n}}_1 \cdot \hat{\mathbf{n}}_2|).$$

*See Appendix D for a justification.

This can be achieved with the aid of the spherical linear interpolation function[†]

$$\hat{\mathbf{n}}_{\text{wp}} = \text{slerp}(-\hat{\mathbf{k}}, \hat{\mathbf{n}}_{\text{trgt}}, 1/2), \quad (18)$$

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

$$R_{\text{wp}} = R_{\hat{\mathbf{n}}}(\alpha/2) \quad \text{where} \quad \hat{\mathbf{n}} = \frac{(-\hat{\mathbf{k}}) \times \hat{\mathbf{n}}_{\text{wp}}}{\|(-\hat{\mathbf{k}}) \times \hat{\mathbf{n}}_{\text{wp}}\|}. \quad (19)$$

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 $\hat{\mathbf{u}}_{\text{sl}}$ and $\hat{\mathbf{n}}_{\text{trgt}}$ 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

$$\hat{\mathbf{w}} = \frac{\hat{\mathbf{u}}_{\text{sl}} \times \hat{\mathbf{n}}_{\text{trgt}}}{\|\hat{\mathbf{u}}_{\text{sl}} \times \hat{\mathbf{n}}_{\text{sl}}\|}. \quad (20)$$

The deflected shot line is obtained by rotating $\hat{\mathbf{u}}_{\text{sl}}$ about the unit vector $\hat{\mathbf{w}}$ through the rotation angle β , where $0 \leq \beta \leq \alpha$. Our notation for this rotation is $R_{\hat{\mathbf{w}}}(\beta)$. The witness plate has its center at the location $\mathbf{r}_c$ 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 θ , measured from the $-z$ -axis, and the azimuthal angle ϕ , measured from the x -axis.

$$\begin{aligned} \hat{\mathbf{u}}_{\text{frag}} &= \sin(\pi - \theta) \cos \phi \hat{\mathbf{i}} + \sin(\pi - \theta) \sin \phi \hat{\mathbf{j}} + \cos(\pi - \theta) \hat{\mathbf{k}} \\ &= \sin \theta \cos \phi \hat{\mathbf{i}} + \sin \theta \sin \phi \hat{\mathbf{j}} - \cos \theta \hat{\mathbf{k}} \end{aligned} \quad (21)$$

The spall fragments then get deflected according to the equation

$$\hat{\mathbf{u}}'_{\text{frag}} = R_{\hat{\mathbf{w}}}(\beta) \hat{\mathbf{u}}_{\text{frag}}. \quad (22)$$

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

$$\mathbf{r} = t\hat{\mathbf{u}}'_{\text{frag}}, \quad (23)$$

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 $\mathbf{r}$ that satisfy

$$(\mathbf{r} - \mathbf{r}_c) \cdot \hat{\mathbf{n}}_{\text{wp}} = 0. \quad (24)$$

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

$$t = \frac{\mathbf{r}_c \cdot \hat{\mathbf{n}}_{\text{wp}}}{\hat{\mathbf{u}}'_{\text{frag}} \cdot \hat{\mathbf{n}}_{\text{wp}}}. \quad (25)$$

Hence, the hit point on the witness plate is

$$\mathbf{r}_{\text{hit}} = \left(\frac{\mathbf{r}_c \cdot \hat{\mathbf{n}}_{\text{wp}}}{\hat{\mathbf{u}}'_{\text{frag}} \cdot \hat{\mathbf{n}}_{\text{wp}}} \right) \hat{\mathbf{u}}'_{\text{frag}}. \quad (26)$$

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

$$R_{\text{wp}}^{-1}(\mathbf{r}_{\text{hit}} - \mathbf{r}_c), \quad (27)$$

where R_{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

$$\theta_{\text{obl}} = \cos^{-1}(|\hat{\mathbf{u}}'_{\text{frag}} \cdot \hat{\mathbf{n}}_{\text{wp}}|). \quad (28)$$

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

Listing 5. oblique.cpp

```

1 // oblique.cpp: geometry for oblique shots using a right-handed cartesian coordinate system
2 // shotline is fixed along the -z axis
3 // initial orientation of target is in the x-y plane with center at the origin
4 // initial orientation of witness plate is also in the x-y plane, but with center at z = -24 inches
5
6 #include "Rotation.h"
7 #include <iostream>
8 #include <cstdlib>
9 using namespace va;
10 using namespace std;
11
12 int main( int argc, char* argv[] ) {
13
14     const int    N          = 300; // number of frags on the spall cone
15     const double WP_TRGT_OBL = 0.5; // ratio of witness plate obliquity to target obliquity (typically 0.5)
16     const double CONE_HALF_ANGLE = 30.; // cone half angle (deg)
17
18     Vector ihat( 1., 0., 0. ), jhat( 0., 1., 0. ), khat( 0., 0., 1. ); // basis vectors

```

Approved for public release; distribution is unlimited.

```

19 Vector u_sl = -khat; // shotline fixed along -z axis
20 Vector n_trgt = -khat; // initial orientation of target
21 Vector n_wp = -khat; // initial orientation of witness plate
22
23 double pitch_t = 0., yaw_t = 0., roll_t = 0., def = 0.5;
24 if ( argc == 5 ) {
25     pitch_t = rad( atof( argv[1] ) ); // pitch (converted to rad) about x-axis
26     yaw_t = rad( atof( argv[2] ) ); // yaw (converted to rad) about y-axis
27     roll_t = rad( atof( argv[3] ) ); // roll (converted to rad) about z-axis
28     def = atof( argv[4] ); // deflection (dimensionless) ranges from 0 to 1
29 }
30
31 // orient the target by specifying pitch-yaw-roll rotation sequence (totally arbitrary)
32 Rotation R_trgt( pitch_t, yaw_t, roll_t, XYZ ); // construct the rotation
33 n_trgt = R_trgt * n_trgt; // apply the rotation to the target
34
35 double alpha = angle( u_sl, n_trgt ); // compute obliquity of target (rad)
36
37 // orient the witness plate using spherical linear interpolation
38 Vector n = slerp( -khat, n_trgt, WP_TRGT_OBL ); // final orientation
39 Rotation R_wp( ihat, 0. ); // initialize rotation to zero
40 if ( alpha > 0. && WP_TRGT_OBL > 0. ) // case of non-zero obliquity
41     R_wp = Rotation( n_wp, n ); // generate the rotation from the cross product
42 n_wp = n; // orient the witness plate
43 Rotation R_wp_inv = -R_wp; // construct inverse rotation of witness plate
44
45 // factor witness plate rotation into a pitch-yaw-roll rotation sequence and output
46 sequence s = factor( R_wp, XYZ );
47 clog << "Witness Plate Orientation (pitch-yaw-roll sequence):" << endl;
48 clog << "pitch (deg) = " << deg( s.first ) << endl;
49 clog << "yaw (deg) = " << deg( s.second ) << endl;
50 clog << "roll (deg) = " << deg( s.third ) << endl;
51
52 // output target and witness plate obliquity
53 clog << "target obliquity (deg) = " << deg( alpha ) << endl;
54 clog << "wp obliquity (deg) = " << deg( angle( u_sl, n_wp ) ) << endl;
55 double beta = def * alpha; // deflection angle (rad)
56 clog << "spall cone deflection (deg) = " << deg( beta ) << endl;
57
58 Rotation R_def( ihat, 0. ); // rotation for deflection is initially zero
59 Vector w = u_sl ^ n_trgt; // w is cross product of shotline and target normal
60 if ( w.mag() > 0. ) { // w is nonzero iff target at obliquity
61     w = normalize( w ); // make w into a unit vector
62     R_def = Rotation( w, beta ); // rotation for deflection
63 }
64
65 Vector rc = -24. * khat; // location of center of witness plate
66 Vector u_frag, r_hit, r;
67 double th = M_PI - rad( CONE_HALF_ANGLE ), ph, t;
68
69 for ( int i = 0; i < N; i++ ) { // generate frags on the spall cone
70
71     ph = 2. * M_PI * i / double( N );
72     u_frag = sin( th ) * cos( ph ) * ihat + sin( th ) * sin( ph ) * jhat + cos( th ) * khat;
73     u_frag = R_def * u_frag;
74     t = ( rc * n_wp ) / ( u_frag * n_wp );
75     r_hit = t * u_frag;
76
77     r = R_wp_inv * ( r_hit - rc );
78     cout << r.x() << "\t" << r.y() << endl;
79 }
80
81 return EXIT_SUCCESS;
82 }

```

Compiling and running this program with the command

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

prints the following back to the screen:

```

1 Witness Plate Orientation (pitch-yaw-roll sequence):
2 pitch (deg) = 20.246
3 yaw (deg) = -18.4381
4 roll (deg) = 3.31976
5 target obliquity (deg) = 54.2403
6 wp obliquity (deg) = 27.1201
7 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.

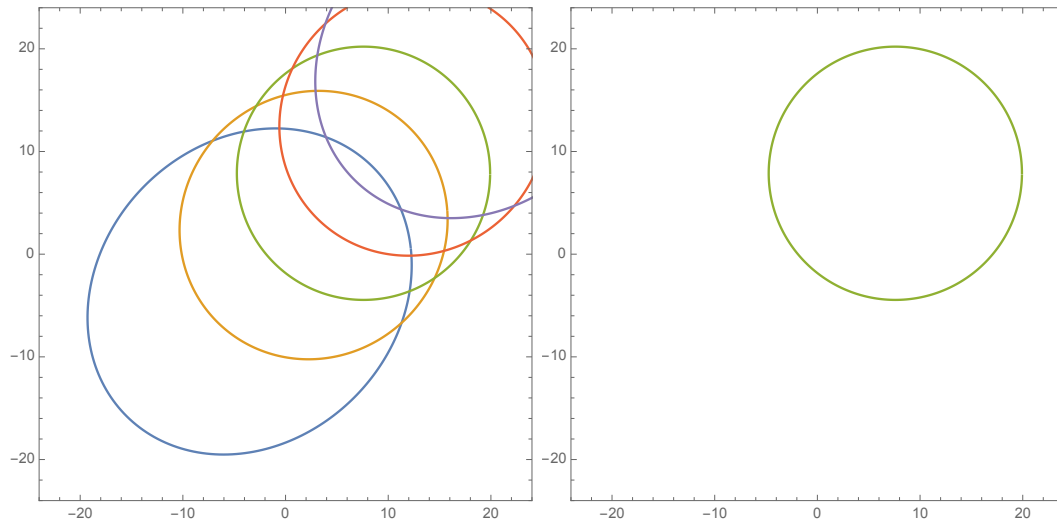


Fig. 4. Plots of the 30° (half-angle) spall cone on the witness plate for a range of deflections. On the left, the deflection parameter is 0, 0.25, 0.5, 0.75, and 1 as the ellipses go from lower left to upper right. On the right is the plot for a deflection of 0.5, which matches the witness plate obliquity 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

1. Kuipers JB. Quaternions and rotation sequences: a primer with applications to orbits, aerospace, and virtual reality. Princeton (NJ): Princeton University Press; 2002.
2. Yatteau JD, Zernow RH, Recht GW, Edquist KT. FATEPEN. Ver. 3.0.0b. Terminal ballistic penetration model. Applied Research Associates (ARA) Project 4714, prepared for Naval Surface Warfare Center, Dahlgren, VA; 1999 Jan.
3. Yatteau JD, Zernow RH, Recht GW, Edquist KT. FATEPEN fast air target encounter penetration (Ver. 3.0.0) terminal ballistic penetration model. Littleton (CO): Applied Research Associates, Inc.; 2001 Sep. Rev. 2005 Feb 2. (Analyst's manual; vol. 1).

Appendix A. Vector Class

Listing A-1. Vector.h

```

1 // Vector.h: Definition & implementation of class for the algebra of 3D vectors
2 //           R. Saucier, February 2000 (Revised June 2016)
3
4 #ifndef VECTOR_3D_H
5 #define VECTOR_3D_H
6
7 #include <cstdlib>
8 #include <cassert>
9 #include <cmath>
10 #include <iostream>
11
12 namespace va { // vector algebra namespace
13
14 enum rep { CART, POLAR }; // vector representation (cartesian or polar)
15 enum comp { X, Y, Z }; // for referencing cartesian components
16 const double R2D( 180. / M_PI ); // for converting radians to degrees
17 const double D2R( M_PI / 180. ); // for converting degrees to radians
18 inline double deg( const double rad ) { return rad * R2D; } // convert radians to degrees
19 inline double rad( const double deg ) { return deg * D2R; } // convert degrees to radians
20 // IEEE 754 standard has 53 significant bits or 15 decimal digits of accuracy, so anything smaller is not significant
21 static const long double TOL = 1.e-15;
22
23 class Vector {
24
25 // friends list
26 // overloaded arithmetic operators
27
28 friend Vector operator+( const Vector& a, const Vector& b ) { // addition
29
30     return Vector( a._x + b._x, a._y + b._y, a._z + b._z );
31 }
32
33 friend Vector operator-( const Vector& a, const Vector& b ) { // subtraction
34
35     return Vector( a._x - b._x, a._y - b._y, a._z - b._z );
36 }
37
38 friend Vector operator*( const Vector& v, double s ) { // right multiply by scalar
39
40     return Vector( s * v._x, s * v._y, s * v._z );
41 }
42
43 friend Vector operator*( double s, const Vector& v ) { // left multiply by scalar
44
45     return Vector( s * v._x, s * v._y, s * v._z );
46 }
47
48 friend Vector operator/( const Vector& v, double s ) { // division by scalar
49
50     assert( s != 0. );
51     return Vector( v._x / s, v._y / s, v._z / s );
52 }
53
54 // overloaded algebraic operators
55
56 friend inline double operator*( const Vector& a, const Vector& b ) { // dot product
57
58     double c( a._x * b._x +
59               a._y * b._y +
60               a._z * b._z );
61     if ( fabs(c) < TOL ) c = 0.0L; // set precision to be no more than 15 digits
62     return c;
63 }
64
65 friend inline Vector operator^( const Vector& a, const Vector& b ) { // cross product
66
67     return Vector( a._y * b._z - a._z * b._y,
68                   a._z * b._x - a._x * b._z,
69                   a._x * b._y - a._y * b._x );
70 }
71
72 // access functions
73
74 friend double x( const Vector& v ) { // x-coordinate
75
76     return v._x;
77 }
78
79 friend double y( const Vector& v ) { // y-coordinate
80
81     return v._y;
82 }
83

```

```

84     friend double z( const Vector& v ) { // z-coordinate
85
86         return v._z;
87     }
88
89     friend double r( const Vector& v ) { // magnitude of vector
90
91         return v._mag();
92     }
93
94     friend double theta( const Vector& v ) { // polar angle (radians)
95
96         return v.theta();
97     }
98
99     friend double phi( const Vector& v ) { // azimuthal angle (radians)
100
101         return v.phi();
102     }
103
104     friend double norm( const Vector& v ) { // norm or magnitude
105
106         return v._mag();
107     }
108
109     friend double mag( const Vector& v ) { // magnitude
110
111         return v._mag();
112     }
113
114     friend double scalar( const Vector& v ) { // magnitude
115
116         return v._mag();
117     }
118
119     friend double angle( const Vector& a, const Vector& b ) { // angle (radians) between vectors
120
121         double s = a.unit() * b.unit();
122         if ( s >= 1. )
123             return 0.;
124         else if ( s <= -1. )
125             return M_PI;
126         else
127             return acos( s );
128     }
129
130     friend Vector unit( const Vector& v ) { // unit vector in same direction
131
132         return v.unit();
133     }
134
135     friend Vector normalize( const Vector& v ) { // returns a unit vector in same direction
136
137         return v.unit();
138     }
139
140     friend double dircos( const Vector& v, const comp& i ) { // direction cosine
141
142         return v.dircos( i );
143     }
144
145     friend Vector proj( const Vector& a, // projection of first vector
146                       const Vector& b ) { // along second vector
147
148         return a.proj( b );
149     }
150
151     // overloaded stream operators
152
153     friend std::istream& operator>>( std::istream& is, Vector& v ) { // input vector
154
155         return is >> v._x >> v._y >> v._z;
156     }
157
158     friend std::ostream& operator<<( std::ostream& os, const Vector& v ) { // output vector
159
160         Vector a( v );
161         a._set_precision();
162         return os << a._x << " " << a._y << " " << a._z;
163     }
164
165 public:
166
167     Vector( double x, double y, double z, // constructor (cartesian or polar
168            rep mode = CART ) { // with cartesian as default)

```

```

169
170     if ( mode == CART ) { // cartesian form
171         this->_x = x;
172         this->_y = y;
173         this->_z = z;
174     }
175     else if ( mode == POLAR ) // polar form
176         _setCartesian( x, y, z );
177     else {
178         std::cerr << "Vector: mode must be either CART or POLAR" << std::endl;
179         exit( EXIT_FAILURE );
180     }
181 }
182
183 Vector( void ) : _x( 0. ), _y( 0. ), _z( 0. ) { // default constructor
184 }
185
186 ~Vector( void ) { // default destructor
187 }
188
189 Vector( const Vector& v ) : _x( v._x ), // copy constructor
190                             _y( v._y ),
191                             _z( v._z ) {
192 }
193
194 Vector& operator=( const Vector& v ) { // assignment operator
195
196     if ( this != &v ) {
197         _x = v._x;
198         _y = v._y;
199         _z = v._z;
200     }
201     return *this;
202 }
203
204 // overloaded arithmetic operators
205
206 Vector& operator+=( const Vector& v ) { // addition assignment
207
208     _x += v._x;
209     _y += v._y;
210     _z += v._z;
211     return *this;
212 }
213
214 Vector& operator-=( const Vector& v ) { // subtraction assignment
215
216     _x -= v._x;
217     _y -= v._y;
218     _z -= v._z;
219     return *this;
220 }
221
222 Vector& operator*=( double s ) { // multiplication assignment
223
224     _x *= s;
225     _y *= s;
226     _z *= s;
227     return *this;
228 }
229
230 Vector& operator/=( double s ) { // division assignment
231
232     assert( s != 0. );
233     _x /= s;
234     _y /= s;
235     _z /= s;
236     return *this;
237 }
238
239 Vector operator-( void ) { // negative of a vector
240
241     return Vector( -_x, -_y, -_z );
242 }
243
244 const double& operator[]( comp i ) const { // index operator (component)
245
246     if ( i == X ) return _x;
247     if ( i == Y ) return _y;
248     if ( i == Z ) return _z;
249     std::cerr << "Vector: Array index out of range; must be X, Y, or Z" << std::endl;
250     exit( EXIT_FAILURE );
251 }
252
253 // access functions

```

```

254
255     double x( void ) const { // x-component
256
257         return _x;
258     }
259
260     double y( void ) const { // y-component
261
262         return _y;
263     }
264
265     double z( void ) const { // z-component
266
267         return _z;
268     }
269
270     double r( void ) const { // magnitude
271
272         return _mag();
273     }
274
275     double theta( void ) const { // polar angle (radians)
276
277         return _theta();
278     }
279
280     double phi( void ) const { // azimuthal angle (radians)
281
282         return _phi();
283     }
284
285     double norm( void ) const { // norm or magnitude
286
287         return _mag();
288     }
289
290     double mag( void ) const { // magnitude
291
292         return _mag();
293     }
294
295     operator double( void ) const { // conversion operator to return magnitude
296
297         return _mag();
298     }
299
300     // utility functions
301
302     Vector unit( void ) const { // returns a unit vector
303
304         double m = _mag();
305         if ( m > 0. ) return Vector( _x / m, _y / m, _z / m );
306         std::cerr << "Vector: Cannot make a unit vector from a null vector" << std::endl;
307         exit( EXIT_FAILURE );
308     }
309
310     Vector normalize( void ) const { // synonym for unit
311
312         return unit();
313     }
314
315     double dircos( const comp& i ) const { // direction cosine
316
317         if ( i == X ) return _x / _mag();
318         if ( i == Y ) return _y / _mag();
319         if ( i == Z ) return _z / _mag();
320         std::cerr << "Vector dircos: comp out of range; must be X, Y, or Z" << std::endl;
321         exit( EXIT_FAILURE );
322     }
323
324     Vector proj( const Vector& e ) const { // projects onto the given vector
325
326         Vector u = e.unit();
327         return u * ( *this * u );
328     }
329
330     Vector& rotate( const Vector& a, double angle ) { // rotates vector about given axial vector, a, through the given
331         angle
332         Vector a_hat = a.unit(); // unit vector along the axial vector
333         Vector cross = a_hat ^ *this; // cross product of a_hat with the given vector
334         return *this += sin( angle ) * cross + ( 1. - cos( angle ) ) * ( a_hat ^ cross );
335     }
336
337     Vector& rot( const Vector& a, double angle ) { // synonym for rotate

```

```

338
339     return rotate( a, angle );
340 }
341
342 private:
343
344     double _x, _y, _z; // cartesian representation
345
346     double _mag( void ) const { // compute the magnitude
347
348         double mag = sqrt( _x * _x + _y * _y + _z * _z );
349         if ( mag < TOL ) mag = 0.0L;
350         return mag;
351     }
352
353     double _theta( void ) const { // compute the polar angle (radians)
354
355         if ( _z != 0. ) return atan2( sqrt( _x * _x + _y * _y ), _z );
356         else return M_PI_2;
357     }
358
359     double _phi( void ) const { // compute the azimuthal angle (radians)
360
361         if ( _x != 0. )
362             return atan2( _y, _x );
363         else {
364             if ( _y > 0 )
365                 return M_PI_2;
366             else if ( _y == 0. )
367                 return 0.;
368             else
369                 return -M_PI_2;
370         }
371     }
372
373     // set cartesian representation from polar
374     void _setCartesian( double r, double theta, double phi ) {
375
376         _x = r * sin( theta ) * cos( phi );
377         _y = r * sin( theta ) * sin( phi );
378         _z = r * cos( theta );
379     }
380
381     Vector _set_precision( void ) { // no more than 15 digits of precision
382
383         if ( fabs(_x) < TOL ) _x = 0.0L;
384         if ( fabs(_y) < TOL ) _y = 0.0L;
385         if ( fabs(_z) < TOL ) _z = 0.0L;
386
387         return *this;
388     }
389 };
390
391 // declaration of friends
392 Vector operator+( const Vector& a, const Vector& b );
393 Vector operator-( const Vector& a, const Vector& b );
394 Vector operator*( const Vector& v, double s );
395 Vector operator*( double s, const Vector& v );
396 Vector operator/( const Vector& v, double s );
397 double operator*( const Vector& a, const Vector& b );
398 Vector operator^( const Vector& a, const Vector& b );
399 double x( const Vector& v );
400 double y( const Vector& v );
401 double z( const Vector& v );
402 double r( const Vector& v );
403 double theta( const Vector& v );
404 double phi( const Vector& v );
405 double mag( const Vector& v );
406 double scalar( const Vector& v );
407 double angle( const Vector& a, const Vector& b );
408 Vector unit( const Vector& v );
409 Vector normalize( const Vector& v );
410 double dircos( const Vector& v, const comp& i );
411 Vector proj( const Vector& a, const Vector& b );
412 std::istream& operator>>( std::istream& is, Vector& v );
413 std::ostream& operator<<( std::ostream& os, const Vector& v );
414 } // vector algebra namespace
415 #endif

```

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.

Table A-1. Vector: A C++ class for 3-dimensional vector algebra—reference sheet

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

^a r is the magnitude, θ is the polar angle measured from the z -axis, and ϕ is the azimuthal angle measured from the x -axis to the plane that contains the vector and the z -axis. The angle θ and ϕ are in radians. Use `rad(deg)` to convert degrees to radians and `deg(rad)` to convert radians to degrees.

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

^c`normalize` does not change the vector it is invoked on; it merely returns the vector divided by its norm.

Approved for public release; distribution is unlimited.

INTENTIONALLY LEFT BLANK.

Appendix B. Rotation Class

Listing B-1. Rotation.h

```

1 // Rotation.h: Rotation class definition for the algebra of 3D rotations
2 // Ref: Kuipers, J. B., Quaternions and Rotation Sequences, Princeton, 1999.
3 //      Altman, S. L., Rotations, Quaternions, and Double Groups, 1986.
4 //      Doran & Lasenby, Geometric Algebra for Physicists, Cambridge, 2003.
5 // R. Saucier, March 2005 (Last revised June 2016)
6
7 #ifndef ROTATION_H
8 #define ROTATION_H
9
10 #include "Vector.h"
11 #include "Random.h"
12 #include <iostream>
13
14 namespace va { // vector algebra namespace
15
16     const Vector DEFAULT_UNIT_VECTOR( 0., 0., 1. ); // arbitrarily choose k
17     const double DEFAULT_ROTATION_ANGLE( 0. ); // arbitrarily choose 0
18     const double TWO_PI( 2. * M_PI );
19
20     enum ORDER { // order of rotation sequence about body axes
21
22         // six distinct principal axes factorizations
23         ZYX, // first about z-axis, second about y-axis and third about x-axis
24         XYZ, // first about x-axis, second about y-axis and third about z-axis
25         YXZ, // first about y-axis, second about x-axis and third about z-axis
26         ZXY, // first about z-axis, second about x-axis and third about y-axis
27         XZY, // first about x-axis, second about z-axis and third about y-axis
28         YZX, // first about y-axis, second about z-axis and third about x-axis
29
30         // six repeated principal axes factorizations
31         ZYZ, // first about z-axis, second about y-axis and third about z-axis
32         ZXZ, // first about z-axis, second about x-axis and third about z-axis
33         YZY, // first about y-axis, second about z-axis and third about y-axis
34         YXY, // first about y-axis, second about x-axis and third about y-axis
35         XYX, // first about x-axis, second about y-axis and third about x-axis
36         XZX, // first about x-axis, second about z-axis and third about x-axis
37     };
38
39     struct quaternion { // q = w + v, where w is scalar part and v is vector part
40
41         quaternion( void ) {
42         }
43
44         quaternion( double scalar, Vector vector ) : w( scalar ), v( vector ) {
45         }
46
47         ~quaternion( void ) {
48         }
49
50         // overloaded multiplication of two quaternions
51         friend quaternion operator*( const quaternion& q1, const quaternion& q2 ) {
52
53             return quaternion(
54                 ( q1.w * q2.w ) - ( q1.v * q2.v ), // scalar part
55                 ( q1.w * q2.v ) + ( q2.w * q1.v ) + ( q1.v ^ q2.v ) // vector part
56             );
57         }
58
59         double w; // scalar part
60         Vector v; // vector part (actually a bivector disguised as a vector)
61     };
62
63     struct sequence { // rotation sequence about three principal body axes
64
65         sequence( void ) {
66         }
67
68         sequence( double phi_1, double phi_2, double phi_3 ) : first( phi_1 ), second( phi_2 ), third( phi_3 ) {
69         }
70
71         ~sequence( void ) {
72         }
73
74         // factor quaternion into Euler rotation sequence about three distinct principal axes
75         sequence factor( const quaternion& p, ORDER order ) {
76
77             double p0 = p.w;
78             double p1 = x( p.v );
79             double p2 = y( p.v );
80             double p3 = z( p.v );
81
82             double phi_1, phi_2, phi_3;
83             if ( order == ZYX ) { // distinct principal axes zyx

```

```

84
85     double A = p0 * p1 + p2 * p3;
86     double B = ( p2 - p0 ) * ( p2 + p0 );
87     double D = ( p1 - p3 ) * ( p1 + p3 );
88
89     phi_3 = atan( - 2. * A / ( B + D ) );
90     double c0 = cos( 0.5 * phi_3 );
91     double c1 = sin( 0.5 * phi_3 );
92
93     double q0 = p0 * c0 + p1 * c1;
94     //double q1 = p1 * c0 - p0 * c1;
95     double q2 = p2 * c0 - p3 * c1;
96     double q3 = p3 * c0 + p2 * c1;
97
98     phi_1 = 2. * atan( q3 / q0 );
99     phi_2 = 2. * atan( q2 / q0 );
100 }
101 else if ( order == XYZ ) { // distinct principal axes xyz
102
103     double A = p1 * p2 - p0 * p3;
104     double B = ( p1 - p3 ) * ( p1 + p3 );
105     double D = ( p0 - p2 ) * ( p0 + p2 );
106
107     phi_3 = atan( - 2. * A / ( B + D ) );
108     double c0 = cos( 0.5 * phi_3 );
109     double c3 = sin( 0.5 * phi_3 );
110
111     double q0 = p0 * c0 + p3 * c3;
112     double q1 = p1 * c0 - p2 * c3;
113     double q2 = p2 * c0 + p1 * c3;
114     //double q3 = p3 * c0 - p0 * c3;
115
116     phi_1 = 2. * atan( q1 / q0 );
117     phi_2 = 2. * atan( q2 / q0 );
118 }
119 else if ( order == YXZ ) { // distinct principal axes yxz
120
121     double A = p1 * p2 + p0 * p3;
122     double B = p1 * p1 + p3 * p3;
123     double D = -( p0 * p0 + p2 * p2 );
124
125     phi_3 = atan( - 2. * A / ( B + D ) );
126     double c0 = cos( 0.5 * phi_3 );
127     double c3 = sin( 0.5 * phi_3 );
128
129     double q0 = p0 * c0 + p3 * c3;
130     double q1 = p1 * c0 - p2 * c3;
131     double q2 = p2 * c0 + p1 * c3;
132     //double q3 = p3 * c0 - p0 * c3;
133
134     phi_1 = 2. * atan( q2 / q0 );
135     phi_2 = 2. * atan( q1 / q0 );
136 }
137 else if ( order == ZXY ) { // distinct principal axes zxy
138
139     double A = p1 * p3 - p0 * p2;
140     double B = ( p0 - p1 ) * ( p0 + p1 );
141     double D = ( p3 - p2 ) * ( p3 + p2 );
142
143     phi_3 = atan( - 2. * A / ( B + D ) );
144     double c0 = cos( 0.5 * phi_3 );
145     double c2 = sin( 0.5 * phi_3 );
146
147     double q0 = p0 * c0 + p2 * c2;
148     double q1 = p1 * c0 + p3 * c2;
149     //double q2 = p2 * c0 - p0 * c2;
150     double q3 = p3 * c0 - p1 * c2;
151
152     phi_1 = 2. * atan( q3 / q0 );
153     phi_2 = 2. * atan( q1 / q0 );
154 }
155 else if ( order == XZY ) { // distinct principal axes xzy
156
157     double A = p1 * p3 + p0 * p2;
158     double B = -( p0 * p0 + p1 * p1 );
159     double D = p2 * p2 + p3 * p3;
160
161     phi_3 = atan( - 2. * A / ( B + D ) );
162     double c0 = cos( 0.5 * phi_3 );
163     double c2 = sin( 0.5 * phi_3 );
164
165     double q0 = p0 * c0 + p2 * c2;
166     double q1 = p1 * c0 + p3 * c2;
167     //double q2 = p2 * c0 - p0 * c2;
168     double q3 = p3 * c0 - p1 * c2;

```

```

169
170     phi_1 = 2. * atan( q1 / q0 );
171     phi_2 = 2. * atan( q3 / q0 );
172 }
173 else if ( order == YZX ) { // distinct principal axes yzx
174
175     double A = p2 * p3 - p0 * p1;
176     double B = p0 * p0 + p2 * p2;
177     double D = -( p1 * p1 + p3 * p3 );
178
179     phi_3 = atan( - 2. * A / ( B + D ) );
180     double c0 = cos( 0.5 * phi_3 );
181     double c1 = sin( 0.5 * phi_3 );
182
183     double q0 = p0 * c0 + p1 * c1;
184     //double q1 = p1 * c0 - p0 * c1;
185     double q2 = p2 * c0 - p3 * c1;
186     double q3 = p3 * c0 + p2 * c1;
187
188     phi_1 = 2. * atan( q2 / q0 );
189     phi_2 = 2. * atan( q3 / q0 );
190 }
191 else if ( order == ZYZ ) { // repeated principal axes zyz
192
193     double A = p0 * p1 + p2 * p3;
194     double B = -2. * p0 * p2;
195     double D = 2. * p1 * p3;
196
197     phi_3 = atan( -2. * A / ( B + D ) );
198     double c0 = cos( 0.5 * phi_3 );
199     double c3 = sin( 0.5 * phi_3 );
200
201     double q0 = p0 * c0 + p3 * c3;
202     //double q1 = p1 * c0 - p2 * c3;
203     double q2 = p2 * c0 + p1 * c3;
204     double q3 = p3 * c0 - p0 * c3;
205
206     phi_1 = 2. * atan( q3 / q0 );
207     phi_2 = 2. * atan( q2 / q0 );
208 }
209 else if ( order == ZXZ ) { // repeated principal axes zxz
210
211     double A = p0 * p2 - p1 * p3;
212     double B = 2. * p0 * p1;
213     double D = 2. * p2 * p3;
214
215     phi_3 = atan( -2. * A / ( B + D ) );
216     double c0 = cos( 0.5 * phi_3 );
217     double c3 = sin( 0.5 * phi_3 );
218
219     double q0 = p0 * c0 + p3 * c3;
220     double q1 = p1 * c0 - p2 * c3;
221     //double q2 = p2 * c0 + p1 * c3;
222     double q3 = p3 * c0 - p0 * c3;
223
224     phi_1 = 2. * atan( q3 / q0 );
225     phi_2 = 2. * atan( q1 / q0 );
226 }
227 else if ( order == YZY ) { // repeated principal axes yzy
228
229     double A = p0 * p1 - p2 * p3;
230     double B = p0 * p3 + p1 * p2;
231
232     phi_3 = atan( -A / B );
233     double c0 = cos( 0.5 * phi_3 );
234     double c2 = sin( 0.5 * phi_3 );
235
236     double q0 = p0 * c0 + p2 * c2;
237     //double q1 = p1 * c0 + p3 * c2;
238     double q2 = p2 * c0 - p0 * c2;
239     double q3 = p3 * c0 - p1 * c2;
240
241     phi_1 = 2. * atan( q2 / q0 );
242     phi_2 = 2. * atan( q3 / q0 );
243 }
244 else if ( order == YXY ) { // repeated principal axes yxy
245
246     double A = p0 * p3 + p1 * p2;
247     double B = -2. * p0 * p1;
248     double D = 2. * p2 * p3;
249
250     phi_3 = atan( -2. * A / ( B + D ) );
251     double c0 = cos( 0.5 * phi_3 );
252     double c2 = sin( 0.5 * phi_3 );
253

```

```

254     double q0 = p0 * c0 + p2 * c2;
255     double q1 = p1 * c0 + p3 * c2;
256     double q2 = p2 * c0 - p0 * c2;
257     //double q3 = p3 * c0 - p1 * c2;
258
259     phi_1 = 2. * atan( q2 / q0 );
260     phi_2 = 2. * atan( q1 / q0 );
261 }
262 else if ( order == XYZ ) { // repeated principal axes xyz
263
264     double A = p0 * p3 - p1 * p2;
265     double B = p0 * p2 + p1 * p3;
266
267     phi_3 = atan( -A / B );
268     double c0 = cos( 0.5 * phi_3 );
269     double c1 = sin( 0.5 * phi_3 );
270
271     double q0 = p0 * c0 + p1 * c1;
272     double q1 = p1 * c0 - p0 * c1;
273     double q2 = p2 * c0 - p3 * c1;
274     //double q3 = p3 * c0 + p2 * c1;
275
276     phi_1 = 2. * atan( q1 / q0 );
277     phi_2 = 2. * atan( q2 / q0 );
278 }
279 else if ( order == XZX ) { // repeated principal axes xzx
280
281     double A = p0 * p2 + p1 * p3;
282     double B = -p0 * p3 + p1 * p2;
283
284     phi_3 = atan( -A / B );
285     double c0 = cos( 0.5 * phi_3 );
286     double c1 = sin( 0.5 * phi_3 );
287
288     double q0 = p0 * c0 + p1 * c1;
289     double q1 = p1 * c0 - p0 * c1;
290     //double q2 = p2 * c0 - p3 * c1;
291     double q3 = p3 * c0 + p2 * c1;
292
293     phi_1 = 2. * atan( q1 / q0 );
294     phi_2 = 2. * atan( q3 / q0 );
295 }
296 else {
297     std::cerr << "ERROR in Rotation: invalid sequence order: " << order << std::endl;
298     exit( EXIT_FAILURE );
299 }
300 return sequence( phi_1, phi_2, phi_3 );
301 }
302
303 double first, // about first body axis
304 second, // about second body axis
305 third; // about third body axis
306 }; // end struct sequence
307
308 struct matrix { // all matrices here are rotations in three-space
309
310     // overloaded multiplication of two matrices
311     // (defined as a convenience to the user; not used in Rotation class)
312     friend matrix operator*( const matrix& A, const matrix& B ) {
313
314         matrix C;
315         C.a11 = A.a11 * B.a11 + A.a12 * B.a21 + A.a13 * B.a31;
316         C.a12 = A.a11 * B.a12 + A.a12 * B.a22 + A.a13 * B.a32;
317         C.a13 = A.a11 * B.a13 + A.a12 * B.a23 + A.a13 * B.a33;
318
319         C.a21 = A.a21 * B.a11 + A.a22 * B.a21 + A.a23 * B.a31;
320         C.a22 = A.a21 * B.a12 + A.a22 * B.a22 + A.a23 * B.a32;
321         C.a23 = A.a21 * B.a13 + A.a22 * B.a23 + A.a23 * B.a33;
322
323         C.a31 = A.a31 * B.a11 + A.a32 * B.a21 + A.a33 * B.a31;
324         C.a32 = A.a31 * B.a12 + A.a32 * B.a22 + A.a33 * B.a32;
325         C.a33 = A.a31 * B.a13 + A.a32 * B.a23 + A.a33 * B.a33;
326
327         return C;
328     }
329
330     // transpose of a matrix
331     // (defined as a convenience to the user; not used in Rotation class)
332     friend matrix transpose( const matrix& A ) {
333
334         matrix B;
335         B.a11 = A.a11;
336         B.a12 = A.a21;
337         B.a13 = A.a31;
338

```

```

339     B.a21 = A.a12;
340     B.a22 = A.a22;
341     B.a23 = A.a32;
342
343     B.a31 = A.a13;
344     B.a32 = A.a23;
345     B.a33 = A.a33;
346
347     return B;
348 }
349
350 // inverse of a matrix
351 // (defined as a convenience to the user; not used in Rotation class)
352 friend matrix inverse( const matrix& A ) {
353
354     double det = A.a11 * ( A.a22 * A.a33 - A.a23 * A.a32 ) +
355                 A.a12 * ( A.a23 * A.a31 - A.a21 * A.a33 ) +
356                 A.a13 * ( A.a21 * A.a32 - A.a22 * A.a31 );
357     assert( det != 0. );
358     matrix B;
359     B.a11 = +( A.a22 * A.a33 - A.a23 * A.a32 ) / det;
360     B.a12 = -( A.a12 * A.a33 - A.a13 * A.a32 ) / det;
361     B.a13 = +( A.a12 * A.a23 - A.a13 * A.a22 ) / det;
362
363     B.a21 = -( A.a21 * A.a33 - A.a23 * A.a31 ) / det;
364     B.a22 = +( A.a11 * A.a33 - A.a13 * A.a31 ) / det;
365     B.a23 = -( A.a11 * A.a23 - A.a13 * A.a21 ) / det;
366
367     B.a31 = +( A.a21 * A.a32 - A.a22 * A.a31 ) / det;
368     B.a32 = -( A.a11 * A.a32 - A.a12 * A.a31 ) / det;
369     B.a33 = +( A.a11 * A.a22 - A.a12 * A.a21 ) / det;
370
371     return B;
372 }
373
374 // returns the matrix with no more than 15 decimal digit accuracy
375 friend matrix set_precision( const matrix& A ) {
376
377     matrix B( A );
378     if ( fabs(B.a11) < TOL ) B.a11 = 0.0L;
379     if ( fabs(B.a12) < TOL ) B.a12 = 0.0L;
380     if ( fabs(B.a13) < TOL ) B.a13 = 0.0L;
381
382     if ( fabs(B.a21) < TOL ) B.a21 = 0.0L;
383     if ( fabs(B.a22) < TOL ) B.a22 = 0.0L;
384     if ( fabs(B.a23) < TOL ) B.a23 = 0.0L;
385
386     if ( fabs(B.a31) < TOL ) B.a31 = 0.0L;
387     if ( fabs(B.a32) < TOL ) B.a32 = 0.0L;
388     if ( fabs(B.a33) < TOL ) B.a33 = 0.0L;
389
390     return B;
391 }
392
393 // convenient matrix properties, but not essential to the Rotation class
394
395 friend double tr( const matrix& A ) { // trace of a matrix
396
397     return A.a11 + A.a22 + A.a33;
398 }
399
400 friend double det( const matrix& A ) { // determinant of a matrix
401
402     return A.a11 * ( A.a22 * A.a33 - A.a23 * A.a32 ) +
403            A.a12 * ( A.a23 * A.a31 - A.a21 * A.a33 ) +
404            A.a13 * ( A.a21 * A.a32 - A.a22 * A.a31 );
405 }
406
407 friend Vector eigenvector( const matrix& A ) { // eigenvector of a matrix
408
409     return unit( ( A.a32 - A.a23 ) * Vector( 1., 0., 0. ) +
410                ( A.a13 - A.a31 ) * Vector( 0., 1., 0. ) +
411                ( A.a21 - A.a12 ) * Vector( 0., 0., 1. ) );
412 }
413
414 friend double angle( const matrix& A ) { // angle of rotation
415
416     return acos( 0.5 * ( A.a11 + A.a22 + A.a33 - 1. ) );
417 }
418
419 Vector eigenvector( void ) { // axis of rotation
420
421     return unit( ( a32 - a23 ) * Vector( 1., 0., 0. ) +
422                ( a13 - a31 ) * Vector( 0., 1., 0. ) +
423                ( a21 - a12 ) * Vector( 0., 0., 1. ) );

```

```

424     }
425
426     double tr( void ) { // trace of the matrix
427
428         return a11 + a22 + a33;
429     }
430
431     double angle( void ) { // angle of rotation
432
433         return acos( 0.5 * ( a11 + a22 + a33 - 1. ) );
434     }
435
436     double a11, a12, a13, // 1st row
437            a21, a22, a23, // 2nd row
438            a31, a32, a33; // 3rd row
439 }; // end struct matrix
440
441 class Rotation {
442
443     // friends list
444
445     // overloaded multiplication of two successive rotations (using quaternions)
446     // notice the order is important; first the right is applied and then the left
447     friend Rotation operator*( const Rotation& R1, const Rotation& R2 ) {
448
449         return Rotation( to_quaternion( R1 ) * to_quaternion( R2 ) );
450     }
451
452     // rotation of a vector
453     // overloaded multiplication of a vector by a rotation (using quaternions)
454     friend Vector operator*( const Rotation& R, const Vector& a ) {
455
456         quaternion q( to_quaternion( R ) );
457         double w( q.w );
458         Vector v( q.v );
459
460         Vector b = 2. * ( v ^ a );
461         return a + ( w * b ) + ( v ^ b );
462     }
463
464     // spherical linear interpolation on the unit sphere from u1 to u2
465     friend Vector slerp( const Vector& u1, const Vector& u2, double theta, double t ) {
466
467         assert( theta != 0 );
468         assert( 0. <= t && t <= 1. );
469         return ( sin( ( 1. - t ) * theta ) * u1 + sin( t * theta ) * u2 ) / sin( theta );
470     }
471
472     // spherical linear interpolation on the unit sphere from u1 to u2
473     friend Vector slerp( const Vector& u1, const Vector& u2, double t ) {
474
475         double theta = angle( u1, u2 );
476         if ( theta == 0. ) return u1;
477         assert( 0. <= t && t <= 1. );
478         return ( sin( ( 1. - t ) * theta ) * u1 + sin( t * theta ) * u2 ) / sin( theta );
479     }
480
481     // access functions
482
483     // return the unit axial vector
484     friend Vector vec( const Rotation& R ) { // return axial unit eigenvector
485
486         return R._vec;
487     }
488
489     // return the rotation angle
490     friend double ang( const Rotation& R ) { // return rotation angle (rad)
491
492         return R._ang;
493     }
494
495     // inverse rotation
496     friend Rotation inverse( Rotation R ) {
497
498         return Rotation( R._vec, -R._ang );
499     }
500
501     // conversion to quaternion
502     friend quaternion to_quaternion( const Rotation& R ) {
503
504         double a = 0.5 * R._ang;
505         Vector u = R._vec;
506
507         return quaternion( cos( a ), u * sin( a ) );
508     }

```

```

509
510 // conversion to rotation matrix
511 friend matrix to_matrix( const Rotation& R ) {
512
513     quaternion q( to_quaternion( R ) );
514     double w = q.w;
515     Vector v = q.v;
516     double v1 = v[ X ], v2 = v[ Y ], v3 = v[ Z ];
517
518     matrix A;
519     A.a11 = 2. * ( w * w - 0.5 + v1 * v1 ); // 1st row, 1st col
520     A.a12 = 2. * ( v1 * v2 - w * v3 ); // 1st row, 2nd col
521     A.a13 = 2. * ( v1 * v3 + w * v2 ); // 1st row, 3rd col
522
523     A.a21 = 2. * ( v1 * v2 + w * v3 ); // 2nd row, 1st col
524     A.a22 = 2. * ( w * w - 0.5 + v2 * v2 ); // 2nd row, 2nd col
525     A.a23 = 2. * ( v2 * v3 - w * v1 ); // 2nd row, 3rd col
526
527     A.a31 = 2. * ( v1 * v3 - w * v2 ); // 3rd row, 1st col
528     A.a32 = 2. * ( v2 * v3 + w * v1 ); // 3rd row, 2nd col
529     A.a33 = 2. * ( w * w - 0.5 + v3 * v3 ); // 3rd row, 3rd col
530
531     return A;
532 }
533
534 // factor rotation into a rotation sequence
535 friend sequence factor( const Rotation& R, ORDER order ) {
536
537     sequence s;
538     return s.factor( to_quaternion( R ), order );
539 }
540
541 // factor matrix representation of rotation into a rotation sequence
542 friend sequence factor( const matrix& A, ORDER order ) {
543
544     sequence s;
545     return s.factor( to_quaternion( Rotation( A ) ), order );
546 }
547
548 // overloaded stream operators
549
550 // input a rotation
551 friend std::istream& operator>>( std::istream& is, Rotation& R ) {
552
553     std::cout << "Specify axis of rotation by entering an axial vector (need not be a unit vector)" << std::endl;
554     is >> R._vec;
555     std::cout << "Enter the angle of rotation (deg): ";
556     is >> R._ang;
557
558     R._vec = unit( R._vec ); // store the unit vector representing the axis
559     R._ang = R._ang * D2R; // store the rotation angle in radians
560     return is;
561 }
562
563 // output a rotation
564 friend std::ostream& operator<<( std::ostream& os, const Rotation& R ) {
565
566     return os << R._vec << "\t" << R._ang * R2D;
567 }
568
569 // output a quaternion
570 friend std::ostream& operator<<( std::ostream& os, const quaternion& q ) {
571
572     return os << q.w << "\t" << q.v;
573 }
574
575 // output a matrix
576 friend std::ostream& operator<<( std::ostream& os, const matrix& A ) {
577
578     matrix B = set_precision( A ); // no more than 15 decimal digits of accuracy
579     return os << B.a11 << "\t" << B.a12 << "\t" << B.a13 << std::endl
580         << B.a21 << "\t" << B.a22 << "\t" << B.a23 << std::endl
581         << B.a31 << "\t" << B.a32 << "\t" << B.a33;
582 }
583
584 public:
585
586 // constructor from three angles (rad), phi_1, phi_2, phi_3 (in that order, left to right)
587 // about three distinct principal body axes
588 Rotation( double phi_1, double phi_2, double phi_3, ORDER order ) {
589
590     double ang_1 = 0.5 * phi_1, c1 = cos( ang_1 ), s1 = sin( ang_1 );
591     double ang_2 = 0.5 * phi_2, c2 = cos( ang_2 ), s2 = sin( ang_2 );
592     double ang_3 = 0.5 * phi_3, c3 = cos( ang_3 ), s3 = sin( ang_3 );
593     double w;

```

```

594     Vector v;
595
596     if ( order == ZYX ) { // 1st about z-axis, 2nd about y-axis, 3rd about x-axis (Aerospace sequence)
597
598         w = c1 * c2 * c3 + s1 * s2 * s3;
599         v = Vector( c1 * c2 * s3 - s1 * s2 * c3,
600                   c1 * s2 * c3 + s1 * c2 * s3,
601                   -c1 * s2 * s3 + s1 * c2 * c3 );
602     }
603     else if ( order == XYZ ) { // 1st about x-axis, 2nd about y-axis, 3rd about z-axis (FATEPEN sequence)
604
605         w = c1 * c2 * c3 - s1 * s2 * s3;
606         v = Vector( c1 * s2 * s3 + s1 * c2 * c3,
607                   c1 * s2 * c3 - s1 * c2 * s3,
608                   c1 * c2 * s3 + s1 * s2 * c3 );
609     }
610     else if ( order == YXZ ) { // 1st about y-axis, 2nd about x-axis, 3rd about z-axis
611
612         w = c1 * c2 * c3 + s1 * s2 * s3;
613         v = Vector( c1 * s2 * c3 + s1 * c2 * s3,
614                   -c1 * s2 * s3 + s1 * c2 * c3,
615                   c1 * c2 * s3 - s1 * s2 * c3 );
616     }
617     else if ( order == ZXY ) { // 1st about z-axis, 2nd about x-axis, 3rd about y-axis
618
619         w = c1 * c2 * c3 - s1 * s2 * s3;
620         v = Vector( c1 * s2 * c3 - s1 * c2 * s3,
621                   c1 * c2 * s3 + s1 * s2 * c3,
622                   c1 * s2 * s3 + s1 * c2 * c3 );
623     }
624     else if ( order == XZY ) { // 1st about x-axis, 2nd about z-axis, 3rd about y-axis
625
626         w = c1 * c2 * c3 + s1 * s2 * s3;
627         v = Vector( -c1 * s2 * s3 + s1 * c2 * c3,
628                   c1 * c2 * s3 - s1 * s2 * c3,
629                   c1 * s2 * c3 + s1 * c2 * s3 );
630     }
631     else if ( order == YZX ) { // 1st about y-axis, 2nd about z-axis, 3rd about x-axis
632
633         w = c1 * c2 * c3 - s1 * s2 * s3;
634         v = Vector( c1 * c2 * s3 + s1 * s2 * c3,
635                   c1 * s2 * s3 + s1 * c2 * c3,
636                   c1 * s2 * c3 - s1 * c2 * s3 );
637     }
638     else if ( order == ZYZ ) { // Euler sequence, 1st about z-axis, 2nd about y-axis, 3rd about z-axis
639
640         w = c1 * c2 * c3 - s1 * c2 * s3;
641         v = Vector( c1 * s2 * s3 - s1 * s2 * c3,
642                   c1 * s2 * c3 + s1 * s2 * s3,
643                   c1 * c2 * s3 + s1 * c2 * c3 );
644     }
645     else if ( order == ZXZ ) { // Euler sequence, 1st about z-axis, 2nd about x-axis, 3rd about z-axis
646
647         w = c1 * c2 * c3 - s1 * c2 * s3;
648         v = Vector( c1 * s2 * c3 + s1 * s2 * s3,
649                   -c1 * s2 * s3 + s1 * s2 * c3,
650                   c1 * c2 * s3 + s1 * c2 * c3 );
651     }
652     else if ( order == YZY ) { // Euler sequence, 1st about y-axis, 2nd about z-axis, 3rd about y-axis
653
654         w = c1 * c2 * c3 - s1 * c2 * s3;
655         v = Vector( -c1 * s2 * s3 + s1 * s2 * c3,
656                   c1 * c2 * s3 + s1 * c2 * c3,
657                   c1 * s2 * c3 + s1 * s2 * s3 );
658     }
659     else if ( order == YXY ) { // Euler sequence, 1st about y-axis, 2nd about x-axis, 3rd about y-axis
660
661         w = c1 * c2 * c3 - s1 * c2 * s3;
662         v = Vector( c1 * s2 * c3 + s1 * s2 * s3,
663                   c1 * c2 * s3 + s1 * c2 * c3,
664                   c1 * s2 * s3 - s1 * s2 * c3 );
665     }
666     else if ( order == XYX ) { // Euler sequence, 1st about x-axis, 2nd about y-axis, 3rd about x-axis
667
668         w = c1 * c2 * c3 - s1 * c2 * s3;
669         v = Vector( c1 * c2 * s3 + s1 * c2 * c3,
670                   c1 * s2 * c3 + s1 * s2 * s3,
671                   -c1 * s2 * s3 + s1 * s2 * c3 );
672     }
673     else if ( order == XZX ) { // Euler sequence, 1st about x-axis, 2nd about z-axis, 3rd about x-axis
674
675         w = c1 * c2 * c3 - s1 * c2 * s3;
676         v = Vector( c1 * c2 * s3 + s1 * c2 * c3,
677                   c1 * s2 * s3 - s1 * s2 * c3,
678                   c1 * s2 * c3 + s1 * s2 * s3 );

```

```

679     }
680     else {
681
682         std::cerr << "ERROR in Rotation: invalid order: " << order << std::endl;
683         exit( EXIT_FAILURE );
684     }
685     if ( w >= 1. || v == 0. ) {
686         _ang = DEFAULT_ROTATION_ANGLE;
687         _vec = DEFAULT_UNIT_VECTOR;
688     }
689     else {
690         _ang = 2. * acos( w );
691         _vec = v / sqrt( 1. - w * w );
692     }
693     _set_angle(); // angle in the range [-M.PI, M.PI]
694 }
695
696 // constructor from a rotation sequence
697 Rotation( const sequence& s, ORDER order ) {
698
699     Rotation R( s.first, s.second, s.third, order );
700
701     _vec = vec( R ); // set the axial vector
702     _ang = ang( R ); // set the rotation angle
703     _set_angle(); // angle in the range [-M.PI, M.PI]
704 }
705
706 // constructor from an axial vector and rotation angle (rad)
707 Rotation( const Vector& v, double a ) : _vec( v ), _ang( a ) {
708
709     _vec = _vec.unit(); // store the unit vector representing the axis
710     _set_angle(); // angle in the range [-M.PI, M.PI]
711 }
712
713 // constructor using sphericalCoord (of axial vector) and rotation angle (rad)
714 Rotation( rng::sphericalCoord s, double ang ) {
715
716     _vec = Vector( 1., s.theta, s.phi, POLAR ); // unit vector
717     _ang = ang;
718     _set_angle(); // angle in the range [-M.PI, M.PI]
719 }
720
721 // constructor from the cross product of two vectors
722 // generate the rotation that, when applied to vector a, will result in vector b
723 Rotation( const Vector& a, const Vector& b ) {
724
725     _vec = unit( a ^ b ); // unit vector
726     double s = a.unit() * b.unit();
727     if ( s >= 1. )
728         _ang = 0.;
729     else if ( s <= -1. )
730         _ang = M.PI;
731     else
732         _ang = acos( s );
733
734     _set_angle(); // angle in the range [-M.PI, M.PI]
735 }
736
737 // constructor from unit quaternion
738 Rotation( const quaternion& q ) {
739
740     double w = q.w;
741     Vector v = q.v;
742
743     if ( w >= 1. || v == 0. ) {
744         _ang = DEFAULT_ROTATION_ANGLE;
745         _vec = DEFAULT_UNIT_VECTOR;
746     }
747     else {
748         double n = sqrt( w * w + v * v ); // need to insure it's a unit quaternion
749         w /= n;
750         v /= n;
751         _ang = 2. * acos( w );
752         _vec = v / sqrt( 1. - w * w );
753     }
754     _set_angle(); // angle in the range [-M.PI, M.PI]
755 }
756
757 // constructor from rotation matrix
758 Rotation( const matrix& A ) {
759
760     _vec = ( A.a32 - A.a23 ) * Vector( 1., 0., 0. ) +
761           ( A.a13 - A.a31 ) * Vector( 0., 1., 0. ) +
762           ( A.a21 - A.a12 ) * Vector( 0., 0., 1. );
763     if ( _vec == 0. ) { // then it must be the identity matrix

```

```

764     _vec = DEFAULT_UNIT_VECTOR;
765     _ang = DEFAULT_ROTATION_ANGLE;
766 }
767 else {
768     _vec = _vec.unit();
769     _ang = acos( 0.5 * ( A.a11 + A.a22 + A.a33 - 1. ) );
770 }
771 _set_angle(); // angle in the range [-M_PI, M_PI]
772 }
773
774 // constructor from two sets of three vectors, where the pair must be related by a pure rotation
775 // returns the rotation that will take a1 to b1, a2 to b2, and a3 to b3
776 // Ref: Micheals, R. J. and Boulton, T. E., "Increasing Robustness in Self-Localization and Pose Estimation," online
       paper.
777
778 Rotation( const Vector& a1, const Vector& a2, const Vector& a3, // initial vectors
779           const Vector& b1, const Vector& b2, const Vector& b3 ) { // rotated vectors
780
781     assert( det( a1, a2, a3 ) != 0. && det( b1, b2, b3 ) != 0. ); // all vectors must be nonzero
782     assert( fabs( det( a1, a2, a3 ) - det( b1, b2, b3 ) ) < 0.001 ); // if it doesn't preserve volume, it's not a pure
       rotation
783
784     if ( det( a1, a2, a3 ) == 1 ) { // these are basis vectors so use simpler method to construct the rotation
785
786         matrix A;
787         A.a11 = b1 * a1; A.a12 = b2 * a1; A.a13 = b3 * a1;
788         A.a21 = b1 * a2; A.a22 = b2 * a2; A.a23 = b3 * a2;
789         A.a31 = b1 * a3; A.a32 = b2 * a3; A.a33 = b3 * a3;
790
791         _vec = ( A.a32 - A.a23 ) * Vector( 1., 0., 0. ) +
792               ( A.a13 - A.a31 ) * Vector( 0., 1., 0. ) +
793               ( A.a21 - A.a12 ) * Vector( 0., 0., 1. );
794         if ( _vec == 0. ) { // then it must be the identity matrix
795             _vec = DEFAULT_UNIT_VECTOR;
796             _ang = DEFAULT_ROTATION_ANGLE;
797         }
798         else {
799             _vec = _vec.unit();
800             _ang = acos( 0.5 * ( A.a11 + A.a22 + A.a33 - 1. ) );
801         }
802         _set_angle(); // angle in the range [-M_PI, M_PI]
803     }
804     else { // use R. J. Micheals' closed-form solution to the absolute orientation problem
805
806         Vector c1, c2, c3;
807         double aaa, baa, aba, aab, caa, aca, aac;
808         double q02, q0, q0q1, q1, q0q2, q2, q0q3, q3;
809
810         aaa = det( a1, a2, a3 );
811         baa = det( b1, a2, a3 );
812         aba = det( a1, b2, a3 );
813         aab = det( a1, a2, b3 );
814
815         q02 = fabs( ( aaa + baa + aba + aab ) / ( 4. * aaa ) );
816         q0 = sqrt( q02 );
817
818         c1 = Vector( 0., b1[Z], -b1[Y] );
819         c2 = Vector( 0., b2[Z], -b2[Y] );
820         c3 = Vector( 0., b3[Z], -b3[Y] );
821
822         caa = det( c1, a2, a3 );
823         aca = det( a1, c2, a3 );
824         aac = det( a1, a2, c3 );
825
826         q0q1 = ( caa + aca + aac ) / ( 4. * aaa );
827         q1 = q0q1 / q0;
828
829         c1 = Vector( -b1[Z], 0., b1[X] );
830         c2 = Vector( -b2[Z], 0., b2[X] );
831         c3 = Vector( -b3[Z], 0., b3[X] );
832
833         caa = det( c1, a2, a3 );
834         aca = det( a1, c2, a3 );
835         aac = det( a1, a2, c3 );
836
837         q0q2 = ( caa + aca + aac ) / ( 4. * aaa );
838         q2 = q0q2 / q0;
839
840         c1 = Vector( b1[Y], -b1[X], 0. );
841         c2 = Vector( b2[Y], -b2[X], 0. );
842         c3 = Vector( b3[Y], -b3[X], 0. );
843
844         caa = det( c1, a2, a3 );
845         aca = det( a1, c2, a3 );
846         aac = det( a1, a2, c3 );

```

```

847
848     q0q3 = ( caa + aca + aac ) / ( 4. * aaa );
849     q3 = q0q3 / q0;
850
851     // no need to normalize since constructed to be unit quaternions
852     double w( q0 );
853     Vector v( q1, q2, q3 );
854
855     if ( w >= 1. || v == 0. ) {
856         _ang = DEFAULT_ROTATION_ANGLE;
857         _vec = DEFAULT_UNIT_VECTOR;
858     }
859     else {
860         _ang = 2. * acos( w );
861         _vec = v / sqrt( 1. - w * w );
862     }
863     _set_angle(); // angle in the range [-M_PI, M_PI]
864 }
865 }
866
867 // constructor for a uniform random rotation, uniformly distributed over the unit sphere,
868 // by fast generation of random quaternions, uniformly-distributed over the 4D unit sphere
869 // Ref: Shoemake, K., "Uniform Random Rotations," Graphic Gems III, September, 1991.
870 Rotation( rng::Random& rng ) { // random rotation in canonical form
871
872     double s = rng.uniform( 0., 1. );
873     double s1 = sqrt( 1. - s );
874     double th1 = rng.uniform( 0., TWO_PI );
875     double x = s1 * sin( th1 );
876     double y = s1 * cos( th1 );
877     double s2 = sqrt( s );
878     double th2 = rng.uniform( 0., TWO_PI );
879     double z = s2 * sin( th2 );
880     double w = s2 * cos( th2 );
881     Vector v( x, y, z );
882
883     if ( w >= 1. || v == 0. ) {
884         _ang = DEFAULT_ROTATION_ANGLE;
885         _vec = DEFAULT_UNIT_VECTOR;
886     }
887     else {
888         _ang = 2. * acos( w );
889         _vec = v / sqrt( 1. - w * w );
890     }
891     _set_angle(); // angle in the range [-M_PI, M_PI]
892 }
893
894 // default constructor
895 Rotation( void ) {
896
897     _vec = DEFAULT_UNIT_VECTOR;
898     _ang = DEFAULT_ROTATION_ANGLE;
899 }
900
901 // default destructor
902 ~Rotation( void ) {
903 }
904
905 // copy constructor
906 Rotation( const Rotation& r ) : _vec( r._vec ), _ang( r._ang ) {
907
908     _set_angle(); // angle in the range [-M_PI, M_PI]
909 }
910
911 // overloaded assignment operator
912 Rotation& operator=( const Rotation& R ) {
913
914     if ( this != &R ) {
915         _vec = R._vec;
916         _ang = R._ang;
917         _set_angle(); // angle in the range [-M_PI, M_PI]
918     }
919     return *this;
920 }
921
922 // conversion operator to return the eigenvector vec
923 operator Vector( void ) const {
924
925     return _vec;
926 }
927
928 // conversion operator to return the angle of rotation about the eigenvector
929 operator double( void ) const {
930
931     return _ang;

```

```

932     }
933
934     // overloaded arithmetic operators
935
936     // inverse rotation
937     Rotation operator-( void ) {
938
939         return Rotation( -_vec, _ang );
940     }
941
942     // triple scalar product, same as a * ( b ^ c )
943     inline double det( const Vector& a, const Vector& b, const Vector& c ) {
944
945         return a[X] * ( b[Y] * c[Z] - b[Z] * c[Y] ) +
946                a[Y] * ( b[Z] * c[X] - b[X] * c[Z] ) +
947                a[Z] * ( b[X] * c[Y] - b[Y] * c[X] );
948     }
949
950 private:
951
952     inline void _set_angle( void ) { // always choose the smaller of the two angles
953
954         if ( _ang > TWO_PI ) _ang -= TWO_PI;
955         if ( _ang < -TWO_PI ) _ang += TWO_PI;
956         if ( _ang > M_PI ) {
957             _ang = TWO_PI - _ang;
958             _vec = -_vec;
959         }
960         else if ( _ang < -M_PI ) {
961             _ang = TWO_PI + _ang;
962             _vec = -_vec;
963         }
964     }
965
966     Vector _vec; // unit eigenvector representing the axis of rotation
967     double _ang; // angle of rotation (rad) falls in the range [-M_PI, M_PI]
968 };
969
970 // declaration of friends
971 quaternion operator*( const quaternion& q1, const quaternion& q2 ); // product of two quaternions
972 matrix operator*( const matrix& A, const matrix& B ); // product of two matrices, first B, then
973     A // transpose of a matrix
974 matrix transpose( const matrix& A ); // transpose of a matrix
975 matrix inverse( const matrix& A ); // inverse of a matrix
976 matrix set_precision( const matrix& A ); // returns a matrix with no more than 15
977     decimal digit accuracy // trace of a matrix
978 double tr( const matrix& A ); // trace of a matrix
979 double det( const matrix& A ); // determinant of a matrix
980 Vector eigenvector( const matrix& A ); // eigenvector of a rotation matrix
981 double angle( const matrix& A ); // angle of rotation (rad)
982 Rotation operator*( const Rotation& R1, const Rotation& R2 ); // successive rotations, first right,
983     then left // rotation of a vector
984 Vector operator*( const Rotation& R, const Vector& a ); // rotation of a vector
985 Vector slerp( const Vector& u1, const Vector& u2, double t ); // spherical linear interpolation on the
986     unit sphere // slerp, given the angle between the
987     vectors // return axial unit eigenvector
988 Vector vec( const Rotation& R ); // return axial unit eigenvector
989 double ang( const Rotation& R ); // return rotation angle (rad)
990 Rotation inverse( const Rotation& R ); // inverse rotation
991 quaternion to_quaternion( const Rotation& R ); // convert rotation to a quaternion
992 matrix to_matrix( const Rotation& R ); // convert a rotation to a rotation
993     matrix // factor a rotation into a rotation
994 sequence factor( const Rotation& R, ORDER order ); // factor a rotation into a rotation
995     sequence // factor a rotation matrix into a
996     rotation sequence // input rotation
997 std::istream& operator>>( std::istream& is, Rotation& R ); // input rotation
998 std::ostream& operator<<( std::ostream& os, const Rotation& R ); // output a rotation
999 std::ostream& operator<<( std::ostream& os, const quaternion& q ); // output a quaternion
1000 std::ostream& operator<<( std::ostream& os, const matrix& A ); // output a rotation matrix
1001
1002 } // vector algebra namespace
1003 #endif

```

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.

Table B-1. Rotation: A C++ class for 3-dimensional rotations—reference sheet

Operation	Mathematical notation	Rotation class
Definition ^a	Let R be an unspecified rotation.	<code>Rotation R;</code>
	Let R be a rotation specified by yaw, pitch, and roll. ^b	<code>Rotation R(y,p,r,ZYX);</code>
	Let R be the rotation specified by three angles, ϕ_1, ϕ_2, ϕ_3 applied in the order x - y - z .	<code>Rotation R(ph1,ph2,ph3,XYZ);</code>
	Let $R_{\hat{\mathbf{a}}}(\alpha)$ be the rotation about the vector $\mathbf{a}$ through the angle α .	<code>Vector a;</code> <code>Rotation R(a,alpha);</code>
	Let R be the rotation about the direction specified by the angles (θ, ϕ) through the angle α .	<code>pair<double,double> p(th,ph);</code> <code>Rotation R(p,alpha);</code>
	Let R be the rotation specified by the vector cross product $\mathbf{a} \times \mathbf{b}$.	<code>Vector a, b;</code> <code>Rotation R(a,b);</code>
	Let R be the rotation that maps the set of linearly independent vectors $\mathbf{a}_i$ to the set $\mathbf{b}_i$, where $i = 1, 2, 3$.	<code>Vector a1,a2,a3,b1,b2,b3;</code> <code>Rotation R(a1,a2,a3,b1,b2,b3);</code>
	Let R be the rotation specified by the (unit) quaternion q . ^c	<code>quaternion q;</code> <code>Rotation R(q);</code>
	Let R be the rotation specified by the 3×3 rotation matrix A_{ij} . ^d	<code>matrix A;</code> <code>Rotation R(A);</code>
	Let R be a random rotation, designed to randomly orient any vector uniformly over the unit sphere.	<code>rng::Random rng;</code> <code>R(rng);</code>
Input a rotation R	n/a	<code>cin >> R;</code>
Output the rotation R	n/a	<code>cout << R;</code>
Assign one rotation to another	Let $R_2 = R_1$ <i>or</i> $R_2 \Leftarrow R_1$	<code>R2 = R1; or</code> <code>R2(R1);</code>
Product of two successive rotations ^e	$R_2 R_1$	<code>R2 * R1;</code>
Rotation of a vector $\mathbf{a}$	$R\mathbf{a}$	<code>R * a;</code>
Inverse rotation	R^{-1}	<code>inverse(R); or -R;</code>
Convert a rotation to a quaternion	If $R_{\hat{\mathbf{u}}}(\theta)$ is the rotation, then $q = \cos(\theta/2) + \hat{\mathbf{u}} \sin(\theta/2)$.	<code>to_quaternion(R);</code>
Convert a rotation to a 3×3 matrix	<i>See description on next page.</i>	<code>to_matrix(R);</code>
Factor a rotation into a rotation sequence	<i>See description on next page.</i>	<code>sequence s = factor(R,ZYX);^f</code>
Unit vector along the axis of rotation	Unit vector $\hat{\mathbf{u}}$ in the rotation $R_{\hat{\mathbf{u}}}(\theta)$	<code>Vector(R); or</code> <code>vec(R);</code>
Rotation angle	Angle θ in the rotation $R_{\hat{\mathbf{u}}}(\theta)$	<code>double(R); or</code> <code>ang(R);</code>

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

^bThe 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 ZYX. There are a total of 12 possible orderings available to the user, 6 of them have distinct principal rotation axes: XYZ, XZY, YXZ, YZX, ZXY, ZYX; and 6 have repeated principal rotation axes: YYX, XZX, YXY, YZY, ZXZ, ZYZ.

Convert rotation to a 3×3 matrix: First we convert the rotation $R_{\hat{\mathbf{u}}}(\theta)$ into the unit quaternion, via $q = \cos(\theta/2) + \hat{\mathbf{u}} \sin(\theta/2)$, and set $w = \cos(\theta/2)$, the scalar part, and $\mathbf{v} = \hat{\mathbf{u}} \sin(\theta/2)$, 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 $R_{\hat{\mathbf{u}}}(\theta)$ into the unit quaternion, via $q = \cos(\theta/2) + \hat{\mathbf{u}} \sin(\theta/2)$, and set $w = \cos(\theta/2)$, the scalar part, and $\mathbf{v} = \hat{\mathbf{u}} \sin(\theta/2)$, 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.

^cA quaternion is defined in the Rotation class as follows:

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

A *unit* quaternion requires that $w^2 + \|\mathbf{v}\|^2 = 1$.

^dA matrix is defined in the Rotation class as follows:

```
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 $\det A = 1$.

^eIn 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.

^fA (rotation) sequence is defined in the Rotation class as follows:

```
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, XZY, YZX, YXZ, ZYX, ZYZ, ZXZ, YZY, YXY, XYX, 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.

INTENTIONALLY LEFT BLANK.

Appendix C. Quaternion Algebra and Vector Rotations

C-1. Quaternion Multiplication

Starting with the multiplication rule

$$\hat{\mathbf{i}}^2 = \hat{\mathbf{j}}^2 = \hat{\mathbf{k}}^2 = \hat{\mathbf{i}}\hat{\mathbf{j}}\hat{\mathbf{k}} = -1, \quad (\text{C-1})$$

it then follows that

$$\hat{\mathbf{i}}\hat{\mathbf{j}} = -\hat{\mathbf{j}}\hat{\mathbf{i}} = \hat{\mathbf{k}}, \quad \hat{\mathbf{j}}\hat{\mathbf{k}} = -\hat{\mathbf{k}}\hat{\mathbf{j}} = \hat{\mathbf{i}}, \quad \text{and} \quad \hat{\mathbf{k}}\hat{\mathbf{i}} = -\hat{\mathbf{i}}\hat{\mathbf{k}} = \hat{\mathbf{j}}. \quad (\text{C-2})$$

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

$$\begin{aligned} q_1 &= w_1 + \mathbf{v}_1 = w_1 + \hat{\mathbf{i}}x_1 + \hat{\mathbf{j}}y_1 + \hat{\mathbf{k}}z_1 \\ q_2 &= w_2 + \mathbf{v}_2 = w_2 + \hat{\mathbf{i}}x_2 + \hat{\mathbf{j}}y_2 + \hat{\mathbf{k}}z_2 \end{aligned}$$

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:

$$\begin{aligned} q_1 q_2 &= (w_1 + \hat{\mathbf{i}}x_1 + \hat{\mathbf{j}}y_1 + \hat{\mathbf{k}}z_1)(w_2 + \hat{\mathbf{i}}x_2 + \hat{\mathbf{j}}y_2 + \hat{\mathbf{k}}z_2) \\ &= (w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2) \\ &\quad + \hat{\mathbf{i}}(w_1 x_2 + w_2 x_1 + y_1 z_2 - z_1 y_2) \\ &\quad + \hat{\mathbf{j}}(w_1 y_2 + w_2 y_1 + z_1 x_2 - x_1 z_2) \\ &\quad + \hat{\mathbf{k}}(w_1 z_2 + w_2 z_1 + x_1 y_2 - y_1 x_2) \\ &= w_1 w_2 - \mathbf{v}_1 \cdot \mathbf{v}_2 + w_1 \mathbf{v}_2 + w_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2. \end{aligned} \quad (\text{C-3})$$

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

$$(w_1, \mathbf{v}_1)(w_2, \mathbf{v}_2) = (w_1 w_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, w_1 \mathbf{v}_2 + w_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2). \quad (\text{C-4})$$

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

$$\begin{aligned} qq^{-1} &= (w, \mathbf{v})(w, -\mathbf{v}) \\ &= (w^2 - \mathbf{v} \cdot (-\mathbf{v}), w(-\mathbf{v}) + w\mathbf{v} + \mathbf{v} \times (-\mathbf{v})) \\ &= (w^2 + \|\mathbf{v}\|^2, \mathbf{0}) \\ &= 1. \end{aligned} \tag{C-5}$$

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

$$q = \left(\cos \frac{\theta}{2}, \hat{\mathbf{u}} \sin \frac{\theta}{2} \right) \equiv (w, \mathbf{v}). \tag{C-6}$$

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

$$\begin{aligned} \mathbf{a}' &= q\mathbf{a}q^{-1} \\ &= (w, \mathbf{v})(0, \mathbf{a})(w, -\mathbf{v}) \\ &= (w, \mathbf{v})(\mathbf{a} \cdot \mathbf{v}, w\mathbf{a} - \mathbf{a} \times \mathbf{v}) \\ &= (w\mathbf{a} \cdot \mathbf{v} - \mathbf{v} \cdot (w\mathbf{a} - \mathbf{a} \times \mathbf{v}), w(w\mathbf{a} - \mathbf{a} \times \mathbf{v}) + (\mathbf{a} \cdot \mathbf{v})\mathbf{v} + \mathbf{v} \times (w\mathbf{a} - \mathbf{a} \times \mathbf{v})) \\ &= (0, w^2\mathbf{a} + w\mathbf{v} \times \mathbf{a} + (\mathbf{a} \cdot \mathbf{v})\mathbf{v} + w\mathbf{v} \times \mathbf{a} + \mathbf{v} \times (\mathbf{v} \times \mathbf{a})) \\ &= (0, w^2\mathbf{a} + v^2\mathbf{a} - v^2\mathbf{a} + (\mathbf{a} \cdot \mathbf{v})\mathbf{v} + 2w\mathbf{v} \times \mathbf{a} + \mathbf{v} \times (\mathbf{v} \times \mathbf{a})) \\ &= (0, \mathbf{a} + 2w\mathbf{v} \times \mathbf{a} + 2\mathbf{v} \times (\mathbf{v} \times \mathbf{a})), \end{aligned} \tag{C-7}$$

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

*Quaternions form what is known as a *division algebra*, meaning that every non-zero quaternion has a multiplicative inverse. Vectors by themselves form an algebra but without division. For an interesting discussion of the relative merits of Hamilton's quaternions and Gibbs' vectors, see Chappell JM, Iqbal A, Hartnett JG, Abbott D. The vector algebra war: a historical perspective. Proc IEEE. 2016;4:1997–2004.

the rotated vector is given by

$$\boxed{\mathbf{a}' = \mathbf{a} + 2w\mathbf{v} \times \mathbf{a} + 2\mathbf{v} \times (\mathbf{v} \times \mathbf{a})}. \quad (\text{C-8})$$

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

$$\boxed{\mathbf{a}' = \mathbf{a} + \hat{\mathbf{u}} \times \mathbf{a} \sin \theta + \hat{\mathbf{u}} \times (\hat{\mathbf{u}} \times \mathbf{a}) (1 - \cos \theta)}, \quad (\text{C-9})$$

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 $\mathbf{a}$, we can always write

$$\mathbf{a} = \mathbf{a} - (\mathbf{a} \cdot \hat{\mathbf{u}})\hat{\mathbf{u}} + (\mathbf{a} \cdot \hat{\mathbf{u}})\hat{\mathbf{u}}, \quad (\text{C-10})$$

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

$$[\mathbf{a} - (\mathbf{a} \cdot \hat{\mathbf{u}})\hat{\mathbf{u}}] \cos \theta + \hat{\mathbf{u}} \times [\mathbf{a} - (\mathbf{a} \cdot \hat{\mathbf{u}})\hat{\mathbf{u}}] \sin \theta. \quad (\text{C-11})$$

Hence,

$$\begin{aligned} \mathbf{a}' = R\mathbf{a} &= [\mathbf{a} - (\mathbf{a} \cdot \hat{\mathbf{u}})\hat{\mathbf{u}}] \cos \theta + \hat{\mathbf{u}} \times [\mathbf{a} - (\mathbf{a} \cdot \hat{\mathbf{u}})\hat{\mathbf{u}}] \sin \theta + (\mathbf{a} \cdot \hat{\mathbf{u}})\hat{\mathbf{u}} \\ &= [\mathbf{a} - (\mathbf{a} \cdot \hat{\mathbf{u}})\hat{\mathbf{u}}] \cos \theta + \hat{\mathbf{u}} \times \mathbf{a} \sin \theta + (\mathbf{a} \cdot \hat{\mathbf{u}})\hat{\mathbf{u}}. \end{aligned} \quad (\text{C-12})$$

Now,

$$\hat{\mathbf{u}} \times (\mathbf{a} \times \hat{\mathbf{u}}) = \mathbf{a}(\hat{\mathbf{u}} \cdot \hat{\mathbf{u}}) - \hat{\mathbf{u}}(\hat{\mathbf{u}} \cdot \mathbf{a}) = \mathbf{a} - (\mathbf{a} \cdot \hat{\mathbf{u}})\hat{\mathbf{u}},$$

and therefore

$$\begin{aligned} \mathbf{a}' &= \hat{\mathbf{u}} \times (\mathbf{a} \times \hat{\mathbf{u}}) \cos \theta + \hat{\mathbf{u}} \times \mathbf{a} \sin \theta + \mathbf{a} - \hat{\mathbf{u}} \times (\mathbf{a} \times \hat{\mathbf{u}}) \\ &= -\hat{\mathbf{u}} \times (\hat{\mathbf{u}} \times \mathbf{a}) \cos \theta + \hat{\mathbf{u}} \times \mathbf{a} \sin \theta + \mathbf{a} + \hat{\mathbf{u}} \times (\hat{\mathbf{u}} \times \mathbf{a}) \\ &= \mathbf{a} + \hat{\mathbf{u}} \times \mathbf{a} \sin \theta + \hat{\mathbf{u}} \times (\hat{\mathbf{u}} \times \mathbf{a}) (1 - \cos \theta). \end{aligned} \quad (\text{C-13})$$

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

$$R = R_{\hat{\mathbf{k}}''}(\phi_r) R_{\hat{\mathbf{j}}'}(\phi_y) R_{\hat{\mathbf{i}}}(\phi_p), \quad (\text{D-1})$$

where $\hat{\mathbf{j}}' = R_{\hat{\mathbf{i}}}(\phi_p) \hat{\mathbf{j}}$, $\hat{\mathbf{k}}' = R_{\hat{\mathbf{i}}}(\phi_p) \hat{\mathbf{k}}$, and $\hat{\mathbf{k}}'' = R_{\hat{\mathbf{j}}'}(\phi_y) \hat{\mathbf{k}}' = R_{\hat{\mathbf{j}}'}(\phi_y) R_{\hat{\mathbf{i}}}(\phi_p) \hat{\mathbf{k}}$. 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,

$$R = R_{\hat{\mathbf{k}}''}(\phi_r) R_{\hat{\mathbf{j}}'}(\phi_y) R_{\hat{\mathbf{i}}}(\phi_p) = R_{\hat{\mathbf{i}}}(\phi_p) R_{\hat{\mathbf{j}}}(\phi_y) R_{\hat{\mathbf{k}}}(\phi_r), \quad (\text{D-2})$$

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

$$q_{\hat{\mathbf{u}}}(\phi) = \cos \frac{\phi}{2} + \hat{\mathbf{u}} \sin \frac{\phi}{2} \quad (\text{D-3})$$

for the unit quaternion that represents a counterclockwise rotation of ϕ radians about the unit vector $\hat{\mathbf{u}}$. Then,

$$R_1 = q_{\hat{\mathbf{e}}_1}(\phi_1). \quad (\text{D-4})$$

$$R_2 = q_{\hat{\mathbf{e}}'_2}(\phi_2) = \cos \frac{\phi_2}{2} + \hat{\mathbf{e}}'_2 \sin \frac{\phi_2}{2}, \quad (\text{D-5})$$

where

$$\hat{\mathbf{e}}'_2 = q_{\hat{\mathbf{e}}_1}(\phi_1) \hat{\mathbf{e}}_2 q_{\hat{\mathbf{e}}_1}^{-1}(\phi_1), \quad (\text{D-6})$$

so that

$$\begin{aligned} q_{\hat{\mathbf{e}}'_2}(\phi_2) &= \cos \frac{\phi_2}{2} + \hat{\mathbf{e}}'_2 \sin \frac{\phi_2}{2} \\ &= \cos \frac{\phi_2}{2} + q_{\hat{\mathbf{e}}_1}(\phi_1) \hat{\mathbf{e}}_2 q_{\hat{\mathbf{e}}_1}^{-1}(\phi_1) \sin \frac{\phi_2}{2} \\ &= q_{\hat{\mathbf{e}}_1}(\phi_1) \left(\cos \frac{\phi_2}{2} + \hat{\mathbf{e}}_2 \sin \frac{\phi_2}{2} \right) q_{\hat{\mathbf{e}}_1}^{-1}(\phi_1) \\ &= q_{\hat{\mathbf{e}}_1}(\phi_1) q_{\hat{\mathbf{e}}_2}(\phi_2) q_{\hat{\mathbf{e}}_1}^{-1}(\phi_1). \end{aligned} \quad (\text{D-7})$$

And

$$R_3 = q_{\hat{e}_3''}(\phi_3) = \cos \frac{\phi_3}{2} + \hat{e}_3'' \sin \frac{\phi_3}{2}, \quad (\text{D-8})$$

where

$$\begin{aligned} \hat{e}_3'' &= q_{\hat{e}_2'}(\phi_2) \hat{e}_3' q_{\hat{e}_2'}^{-1}(\phi_2) \\ &= q_{\hat{e}_2'}(\phi_2) q_{\hat{e}_1}(\phi_1) \hat{e}_3 q_{\hat{e}_1}^{-1}(\phi_1) q_{\hat{e}_2'}^{-1}(\phi_2) \\ &= [q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1)] q_{\hat{e}_1}(\phi_1) \hat{e}_3 q_{\hat{e}_1}^{-1}(\phi_1) [q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1)]^{-1} \\ &= q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1) q_{\hat{e}_1}(\phi_1) \hat{e}_3 q_{\hat{e}_1}^{-1}(\phi_1) q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}^{-1}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1) \\ &= q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) \hat{e}_3 q_{\hat{e}_2}^{-1}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1), \end{aligned} \quad (\text{D-9})$$

so that

$$\begin{aligned} q_{\hat{e}_3''}(\phi_3) &= \cos \frac{\phi_3}{2} + \hat{e}_3'' \sin \frac{\phi_3}{2} \\ &= \cos \frac{\phi_3}{2} + q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) \hat{e}_3 q_{\hat{e}_2}^{-1}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1) \sin \frac{\phi_3}{2} \\ &= q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) \left(\cos \frac{\phi_3}{2} + \hat{e}_3 \sin \frac{\phi_3}{2} \right) q_{\hat{e}_2}^{-1}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1) \\ &= q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) q_{\hat{e}_3}(\phi_3) q_{\hat{e}_2}^{-1}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1). \end{aligned} \quad (\text{D-10})$$

Therefore, the total combined rotation is

$$\begin{aligned} R &= R_3 R_2 R_1 = q_{\hat{e}_3''}(\phi_3) q_{\hat{e}_2'}(\phi_2) q_{\hat{e}_1}(\phi_1) \\ &= [q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) q_{\hat{e}_3}(\phi_3) q_{\hat{e}_2}^{-1}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1)] [q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1)] q_{\hat{e}_1}(\phi_1) \\ &= q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) q_{\hat{e}_3}(\phi_3), \end{aligned} \quad (\text{D-11})$$

as was to be shown.

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

Listing D-1. order.cpp

```

1 // order.cpp: demonstrate that rotation about fixed axis in reverse order is correct
2 // for both rotation about distinct axes and for rotation about repeated axes
3
4 #include "Rotation.h"
5 #include <iostream>
6 #include <cstdlib>
7 using namespace std;
8
9 int main( int argc, char* argv[] ) {
10
11     va::Vector i( 1., 0., 0. ), j( 0., 1., 0. ), k( 0., 0., 1. ), v( 1.2, -3.4, 6.7 );
12     va::Vector i1, j1, k1, i2, j2, k2, i3, j3, k3;
13     va::Rotation R, R1, R2, R3;
14     va::ORDER order = va::XYZ; // select rotation sequence about distinct axes

```

```

15 double ang_1 = 0.;
16 double ang_2 = 0.;
17 double ang_3 = 0.;
18 if ( argc == 4 ) {
19
20     ang_1 = va::rad( atof( argv[ 1 ] ) );
21     ang_2 = va::rad( atof( argv[ 2 ] ) );
22     ang_3 = va::rad( atof( argv[ 3 ] ) );
23 }
24
25 R = va::Rotation( ang_1, ang_2, ang_3, order );
26
27 cout << "The constructed rotation is " << R << endl;
28 cout << "The rotated vector is " << R * v << endl;
29 cout << "The following rotations should match this:" << endl;
30
31 R1 = va::Rotation( i, ang_1 ); // rotation about x-axis
32 R2 = va::Rotation( j, ang_2 ); // rotation about y-axis
33 R3 = va::Rotation( k, ang_3 ); // rotation about z-axis
34
35 R = R1 * R2 * R3; // note the order is the reverse: first 3, then 2, then 1
36 cout << "Reverse order about fixed axes:" << endl;
37 cout << "Rotation: " << R << endl;
38 cout << "Rotated vector: " << R * v << endl;
39
40 cout << endl << "Now the conventional way via transformed axes:" << endl << endl;
41
42 // first rotation is about i
43 R1 = va::Rotation( i, ang_1 ); // rotation about x-axis
44 i1 = R1 * i;
45 j1 = R1 * j;
46 k1 = R1 * k;
47
48 // second rotation is about j1
49 R2 = va::Rotation( j1, ang_2 ); // rotation about transformed y-axis
50 i2 = R2 * i1;
51 j2 = R2 * j1;
52 k2 = R2 * k1;
53
54 // third rotation is about k2
55 R3 = va::Rotation( k2, ang_3 ); // rotation about doubly transformed z-axis
56 i3 = R3 * i2;
57 j3 = R3 * j2;
58 k3 = R3 * k2;
59
60 R = R3 * R2 * R1; // note the order is the original: first 1, then 2, then 3
61
62 cout << "Original order about transformed axes:" << endl;
63 cout << "Rotation: " << R << endl;
64 cout << "Rotated vector: " << R * v << endl;
65
66 cout << endl << "This also works for repeated axes." << endl
67 << "Using the same rotation angles, let's do the whole thing over again." << endl << endl;
68 order = va::XYX; // select rotation sequence about repeated axes
69
70 R = va::Rotation( ang_1, ang_2, ang_3, order );
71
72 cout << "The constructed rotation is " << R << endl;
73 cout << "The rotated vector is " << R * v << endl;
74 cout << "The following rotations should match this:" << endl;
75
76 R1 = va::Rotation( i, ang_1 ); // rotation about x-axis
77 R2 = va::Rotation( j, ang_2 ); // rotation about y-axis
78 R3 = va::Rotation( i, ang_3 ); // rotation about x-axis
79
80 R = R1 * R2 * R3; // note the order is the reverse: first 3, then 2, then 1
81 cout << "Reverse order about fixed axes:" << endl;
82 cout << "Rotation: " << R << endl;
83 cout << "Rotated vector: " << R * v << endl;
84
85 cout << endl << "Now the conventional way via transformed axes:" << endl << endl;
86
87 // first rotation is about i
88 R1 = va::Rotation( i, ang_1 ); // rotation about x-axis
89 i1 = R1 * i;
90 j1 = R1 * j;
91 k1 = R1 * k;
92
93 // second rotation is about j1
94 R2 = va::Rotation( j1, ang_2 ); // rotation about transformed y-axis
95 i2 = R2 * i1;
96 j2 = R2 * j1;
97 k2 = R2 * k1;
98
99 // third rotation is about i2

```

```

100     R3 = va::Rotation( i2, ang_3 ); // rotation about doubly transformed x-axis
101     i3 = R3 * i2;
102     j3 = R3 * j2;
103     k3 = R3 * k2;
104
105     R = R3 * R2 * R1; // note the order is the original: first 1, then 2, then 3
106
107     cout << "Original order about transformed axes:" << endl;
108     cout << "Rotation: " << R << endl;
109     cout << "Rotated vector: " << R * v << endl;
110
111     return EXIT_SUCCESS;
112 }

```

The command

```
1 ./order 35. -15. 60.
```

will give the following results:

```

1 The constructed rotation is -0.533171 -0.0686517 -0.843218 73.8825
2 The rotated vector is -0.530512 1.81621 7.36953
3 The following rotations should match this:
4 Reverse order about fixed axes:
5 Rotation: -0.533171 -0.0686517 -0.843218 73.8825
6 Rotated vector: -0.530512 1.81621 7.36953
7
8 Now the conventional way via transformed axes:
9
10 Original order about transformed axes:
11 Rotation: -0.533171 -0.0686517 -0.843218 73.8825
12 Rotated vector: -0.530512 1.81621 7.36953
13
14 This also works for repeated axes.
15 Using the same rotation angles, we do the whole thing over again.
16
17 The constructed rotation is -0.984429 0.171618 0.0380469 95.8951
18 The rotated vector is 2.78824 6.66967 2.37301
19 The following rotations should match this:
20 Reverse order about fixed axes:
21 Rotation: -0.984429 0.171618 0.0380469 95.8951
22 Rotated vector: 2.78824 6.66967 2.37301
23
24 Now the conventional way via transformed axes:
25
26 Original order about transformed axes:
27 Rotation: -0.984429 0.171618 0.0380469 95.8951
28 Rotated vector: 2.78824 6.66967 2.37301

```

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

$$p = p_0 + \hat{\mathbf{i}}p_1 + \hat{\mathbf{j}}p_2 + \hat{\mathbf{k}}p_3. \quad (\text{E-1})$$

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

$$p = (a_0 + \hat{\mathbf{k}}a_3)(b_0 + \hat{\mathbf{j}}b_2)(c_0 + \hat{\mathbf{i}}c_1). \quad (\text{E-2})$$

Let q represent the first 2 factors:

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

Then

$$\begin{aligned} q &= p(c^1)^{-1} = (p_0 + \hat{\mathbf{i}}p_1 + \hat{\mathbf{j}}p_2 + \hat{\mathbf{k}}p_3)(c_0 - \hat{\mathbf{i}}c_1) \\ &= (p_0c_0 + p_1c_1) + \hat{\mathbf{i}}(p_1c_0 - p_0c_1) + \hat{\mathbf{j}}(p_2c_0 - p_3c_1) + \hat{\mathbf{k}}(p_3c_0 + p_2c_1), \end{aligned} \quad (\text{E-4})$$

from which we identify

$$q_0 = p_0c_0 + p_1c_1, \quad q_1 = p_1c_0 - p_0c_1, \quad q_2 = p_2c_0 - p_3c_1, \quad q_3 = p_3c_0 + p_2c_1. \quad (\text{E-5})$$

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

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

¹Kuipers JB. Quaternions and rotation sequences: a primer with applications to orbits, aerospace, and virtual reality. Princeton (NJ): Princeton University Press; 2002.

Now, from Eq. E-5,

$$\begin{bmatrix} q_0 \\ q_2 \end{bmatrix} = \begin{bmatrix} p_0 c_0 + p_1 c_1 \\ p_2 c_0 - p_3 c_1 \end{bmatrix} = \begin{bmatrix} p_0 & p_1 \\ p_2 & -p_3 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \quad (\text{E-7})$$

and

$$\begin{bmatrix} q_1 \\ q_3 \end{bmatrix} = \begin{bmatrix} p_1 c_0 - p_0 c_1 \\ p_3 c_0 + p_2 c_1 \end{bmatrix} = \begin{bmatrix} p_1 & -p_0 \\ p_3 & p_2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}, \quad (\text{E-8})$$

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

$$\begin{aligned} \begin{bmatrix} c_0 & c_1 \end{bmatrix} \begin{bmatrix} p_0 & p_2 \\ p_1 & -p_3 \end{bmatrix} \begin{bmatrix} p_1 & -p_0 \\ p_3 & p_2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \\ \begin{bmatrix} c_0 & c_1 \end{bmatrix} \begin{bmatrix} p_0 p_1 + p_2 p_3 & -p_0^2 + p_2^2 \\ p_1^2 - p_3^2 & -p_0 p_1 - p_2 p_3 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = 0. \end{aligned} \quad (\text{E-9})$$

Define the quantities

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

Then the constraint equation becomes

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

Finally, this may be written as

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

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

$$a^3 = a_0 + \hat{\mathbf{k}}a_3 = \cos \frac{\phi_1}{2} + \hat{\mathbf{k}} \sin \frac{\phi_1}{2}, \quad (\text{E-15})$$

it follows that

$$\tan \frac{\phi_1}{2} = \frac{a_3}{a_0} = \frac{a_3 b_0}{a_0 b_0} = \frac{q_3}{q_0}, \quad (\text{E-16})$$

from Eq. E-3. Therefore, the first rotation (yaw angle in this case) is given by

$$\phi_1 = 2 \tan^{-1} \left(\frac{q_3}{q_0} \right). \quad (\text{E-17})$$

Similarly,

$$\tan \frac{\phi_2}{2} = \frac{b_2}{b_0} = \frac{a_0 b_2}{a_0 b_0} = \frac{q_2}{q_0}, \quad (\text{E-18})$$

again using Eq. E-3, and therefore the second rotation (pitch angle in this case) is given by

$$\phi_2 = 2 \tan^{-1} \left(\frac{q_2}{q_0} \right). \quad (\text{E-19})$$

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 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

$A = p_0 p_1 + p_2 p_3, \quad B = p_2^2 - p_0^2, \quad D = p_1^2 - p_3^2$	
$\phi_3 = \tan^{-1} \left(\frac{-2A}{B + D} \right)$	[third rotation, roll about x -axis]
$c_0 = \cos \frac{\phi_3}{2}, \quad c_1 = \sin \frac{\phi_3}{2}$	
$q_0 = p_0 c_0 + p_1 c_1, \quad q_1 = p_1 c_0 - p_0 c_1, \quad q_2 = p_2 c_0 - p_3 c_1, \quad q_3 = p_3 c_0 + p_2 c_1$	
$\phi_1 = 2 \tan^{-1} \left(\frac{q_3}{q_0} \right)$	[first rotation, yaw about z -axis]
$\phi_2 = 2 \tan^{-1} \left(\frac{q_2}{q_0} \right)$	[second rotation, pitch about y -axis]

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

$A = p_1 p_2 - p_0 p_3, \quad B = p_1^2 - p_3^2, \quad D = p_0^2 - p_2^2$	
$\phi_3 = \tan^{-1} \left(\frac{-2A}{B + D} \right)$	[third rotation, roll about z -axis]
$c_0 = \cos \frac{\phi_3}{2}, \quad c_3 = \sin \frac{\phi_3}{2}$	
$q_0 = p_0 c_0 + p_3 c_3, \quad q_1 = p_1 c_0 - p_2 c_3, \quad q_2 = p_2 c_0 + p_1 c_3, \quad q_3 = p_3 c_0 - p_0 c_3$	
$\phi_1 = 2 \tan^{-1} \left(\frac{q_1}{q_0} \right)$	[first rotation, pitch about x -axis]
$\phi_2 = 2 \tan^{-1} \left(\frac{q_2}{q_0} \right)$	[second rotation, yaw about y -axis]

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

$A = p_1 p_2 + p_0 p_3, \quad B = p_1^2 + p_3^2, \quad D = -p_0^2 - p_2^2$	
$\phi_3 = \tan^{-1} \left(\frac{-2A}{B + D} \right)$	[third rotation about z -axis]
$c_0 = \cos \frac{\phi_3}{2}, \quad c_3 = \sin \frac{\phi_3}{2}$	
$q_0 = p_0 c_0 + p_3 c_3, \quad q_1 = p_1 c_0 - p_2 c_3, \quad q_2 = p_2 c_0 + p_1 c_3, \quad q_3 = p_3 c_0 - p_0 c_3$	
$\phi_1 = 2 \tan^{-1} \left(\frac{q_2}{q_0} \right)$	[first rotation about y -axis]
$\phi_2 = 2 \tan^{-1} \left(\frac{q_1}{q_0} \right)$	[second rotation about x -axis]

Table E-4. Factorization into z - x - y rotation sequence

$A = p_1 p_3 - p_0 p_2, \quad B = p_0^2 - p_1^2, \quad D = p_3^2 - p_2^2$	
$\phi_3 = \tan^{-1} \left(\frac{-2A}{B + D} \right)$	[third rotation about y -axis]
$c_0 = \cos \frac{\phi_3}{2}, \quad c_2 = \sin \frac{\phi_3}{2}$	
$q_0 = p_0 c_0 + p_2 c_2, \quad q_1 = p_1 c_0 + p_3 c_2, \quad q_2 = p_2 c_0 - p_0 c_2, \quad q_3 = p_3 c_0 - p_1 c_2$	
$\phi_1 = 2 \tan^{-1} \left(\frac{q_3}{q_0} \right)$	[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

$A = p_1 p_3 + p_0 p_2, \quad B = -p_0^2 - p_1^2, \quad D = p_2^2 + p_3^2$	
$\phi_3 = \tan^{-1} \left(\frac{-2A}{B + D} \right)$	[third rotation about y -axis]
$c_0 = \cos \frac{\phi_3}{2}, \quad c_2 = \sin \frac{\phi_3}{2}$	
$q_0 = p_0 c_0 + p_2 c_2, \quad q_1 = p_1 c_0 + p_3 c_2, \quad q_2 = p_2 c_0 - p_0 c_2, \quad q_3 = p_3 c_0 - p_1 c_2$	
$\phi_1 = 2 \tan^{-1} \left(\frac{q_1}{q_0} \right)$	[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

$A = p_2 p_3 - p_0 p_1, \quad B = p_0^2 + p_2^2, \quad D = -p_1^2 - p_3^2$	
$\phi_3 = \tan^{-1} \left(\frac{-2A}{B + D} \right)$	[third rotation about x -axis]
$c_0 = \cos \frac{\phi_3}{2}, \quad c_1 = \sin \frac{\phi_3}{2}$	
$q_0 = p_0 c_0 + p_1 c_1, \quad q_1 = p_1 c_0 - p_0 c_1, \quad q_2 = p_2 c_0 - p_3 c_1, \quad q_3 = p_3 c_0 + p_2 c_1$	
$\phi_1 = 2 \tan^{-1} \left(\frac{q_2}{q_0} \right)$	[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.¹

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

$$p = p_0 + \hat{\mathbf{i}}p_1 + \hat{\mathbf{j}}p_2 + \hat{\mathbf{k}}p_3. \quad (\text{E-20})$$

In the notation of Kuipers¹, we want to factor this as $a^3b^2c^3$, so we write

$$p = (a_0 + \hat{\mathbf{k}}a_3)(b_0 + \hat{\mathbf{j}}b_2)(c_0 + \hat{\mathbf{k}}c_3). \quad (\text{E-21})$$

Let q represent the first 2 factors:

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

Then

$$\begin{aligned} q &= p(c^1)^{-1} = (p_0 + \hat{\mathbf{i}}p_1 + \hat{\mathbf{j}}p_2 + \hat{\mathbf{k}}p_3)(c_0 - \hat{\mathbf{k}}c_3) \\ &= (p_0c_0 + p_3c_3) + \hat{\mathbf{i}}(p_1c_0 - p_2c_3) + \hat{\mathbf{j}}(p_2c_0 + p_1c_3) + \hat{\mathbf{k}}(p_3c_0 - p_0c_3), \end{aligned} \quad (\text{E-23})$$

from which we identify

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

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

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

¹See 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.

Now, from Eq. E-24,

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

and

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

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

$$\begin{aligned} \begin{bmatrix} c_0 & c_3 \end{bmatrix} \begin{bmatrix} p_0 & p_2 \\ p_3 & p_1 \end{bmatrix} \begin{bmatrix} p_1 & -p_2 \\ p_3 & -p_0 \end{bmatrix} \begin{bmatrix} c_0 \\ c_3 \end{bmatrix} = \\ \begin{bmatrix} c_0 & c_3 \end{bmatrix} \begin{bmatrix} p_0 p_1 + p_2 p_3 & -p_0 p_2 - p_0 p_2 \\ p_1 p_3 + p_1 p_3 & -p_2 p_3 - p_0 p_1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_3 \end{bmatrix} = 0. \end{aligned} \quad (\text{E-28})$$

Define the quantities

$$A = p_0 p_1 + p_2 p_3, \quad B = -2p_0 p_2, \quad D = 2p_1 p_3. \quad (\text{E-29})$$

Then the constraint equation becomes

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

Finally, this may be written as

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

where we used

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

Therefore, the final rotation is

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

and this quantity is known since A , B , and D are known from Eq. E-29. Furthermore,

since

$$a^3 = a_0 + \hat{\mathbf{k}}a_3 = \cos \frac{\phi_1}{2} + \hat{\mathbf{k}} \sin \frac{\phi_1}{2}, \quad (\text{E-34})$$

it follows that

$$\tan \frac{\phi_1}{2} = \frac{a_3}{a_0} = \frac{a_3 b_0}{a_0 b_0} = \frac{q_3}{q_0}, \quad (\text{E-35})$$

where we used Eq. E-22. Therefore, the first rotation is given by

$$\phi_1 = 2 \tan^{-1} \left(\frac{q_3}{q_0} \right). \quad (\text{E-36})$$

Similarly,

$$\tan \frac{\phi_2}{2} = \frac{b_2}{b_0} = \frac{a_0 b_2}{a_0 b_0} = \frac{q_2}{q_0}, \quad (\text{E-37})$$

again using Eq. E-22, and therefore the second rotation (pitch angle in this case) is given by

$$\phi_2 = 2 \tan^{-1} \left(\frac{q_2}{q_0} \right). \quad (\text{E-38})$$

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 E-7. Factorization into z - y - x rotation sequence

$A = p_0 p_1 + p_2 p_3, \quad B = -2p_0 p_2, \quad D = 2p_1 p_3$	
$\phi_3 = \tan^{-1} \left(\frac{-2A}{B + D} \right)$	[third rotation about z -axis]
$c_0 = \cos \frac{\phi_3}{2}, \quad c_3 = \sin \frac{\phi_3}{2}$	
$q_0 = p_0 c_0 + p_3 c_3, \quad q_1 = p_1 c_0 - p_2 c_3, \quad q_2 = p_2 c_0 + p_1 c_3, \quad q_3 = p_3 c_0 - p_0 c_3$	
$\phi_1 = 2 \tan^{-1} \left(\frac{q_3}{q_0} \right)$	[first rotation about z -axis]
$\phi_2 = 2 \tan^{-1} \left(\frac{q_2}{q_0} \right)$	[second rotation about y -axis]

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

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

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

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

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

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

Table E-11. Factorization into x - y - x rotation sequence

$A = p_0p_3 - p_1p_2, \quad B = p_0p_2 + p_1p_3$	
$\phi_3 = \tan^{-1} \left(\frac{-A}{B} \right)$	[third rotation about x -axis]
$c_0 = \cos \frac{\phi_3}{2}, \quad c_1 = \sin \frac{\phi_3}{2}$	
$q_0 = p_0c_0 + p_1c_1, \quad q_1 = p_1c_0 - p_0c_1, \quad q_2 = p_2c_0 - p_3c_1, \quad q_3 = p_3c_0 + p_2c_1$	
$\phi_1 = 2 \tan^{-1} \left(\frac{q_1}{q_0} \right)$	[first rotation about x -axis]
$\phi_2 = 2 \tan^{-1} \left(\frac{q_2}{q_0} \right)$	[second rotation about y -axis]

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

$A = p_0p_2 + p_1p_3, \quad B = -p_0p_3 + p_1p_2$	
$\phi_3 = \tan^{-1} \left(\frac{-A}{B} \right)$	[third rotation about x -axis]
$c_0 = \cos \frac{\phi_3}{2}, \quad c_1 = \sin \frac{\phi_3}{2}$	
$q_0 = p_0c_0 + p_1c_1, \quad q_1 = p_1c_0 - p_0c_1, \quad q_2 = p_2c_0 - p_3c_1, \quad q_3 = p_3c_0 + p_2c_1$	
$\phi_1 = 2 \tan^{-1} \left(\frac{q_1}{q_0} \right)$	[first rotation about x -axis]
$\phi_2 = 2 \tan^{-1} \left(\frac{q_3}{q_0} \right)$	[second rotation about z -axis]

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

Listing E-1. factor.cpp

```

1 // factor.cpp: test program for the rotation factorization
2
3 #include "Rotation.h"
4 #include <iostream>
5 #include <cstdlib>
6
7 int main( int argc, char* argv[] ) {
8
9     va::Rotation R;
10    rng::Random rng;
11    va::ORDER order = va::ORDER( rng.uniformDiscrete( 0, 11 ) );
12    double ang_1, ang_2, ang_3;
13    if ( argc == 4 ) {
14
15        ang_1 = va::rad( atof( argv[ 1 ] ) );
16        ang_2 = va::rad( atof( argv[ 2 ] ) );
17        ang_3 = va::rad( atof( argv[ 3 ] ) );
18        R = va::Rotation( ang_1, ang_2, ang_3, order );
19        std::cout << "order = " << order << std::endl;

```

```

20     }
21     else if ( argc == 1 ) {
22         R = va::Rotation( rng );
23     }
24     else {
25         std::cerr << argv[ 0 ] << " usage: Enter angles 1, 2, and 3 (deg) on commandline" << std::endl;
26         exit( EXIT_FAILURE );
27     }
28     std::cout << "The rotation is " << R << std::endl << std::endl;; // output the rotation
29     std::cout << "The following rotations should match this" << std::endl;
30
31     for ( int order = 0; order < 12; order++ ) {
32
33         va::sequence s = va::factor( R, va::ORDER( order ) ); // factor the rotation
34
35         std::cout << "1st rotation = " << va::deg( s.first ) << "\torder = " << order << std::endl; // output the
36         // factorization
37         std::cout << "2nd rotation = " << va::deg( s.second ) << std::endl;
38         std::cout << "3rd rotation = " << va::deg( s.third ) << std::endl;
39
40         R = va::Rotation( s, va::ORDER( order ) ); // generate the rotation with this sequence
41         std::cout << R << std::endl; // output the rotation so that it can be compared
42     }
43     return EXIT_SUCCESS;
44 }

```

The command

```
1 ./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:

```

1 The rotation is 0.18777 0.958993 -0.212307      86.6526
2
3 The following rotations should match this
4 1st rotation = -24.8355 order = 0
5 2nd rotation = 84.2077
6 3rd rotation = -2.42093
7 0.18777 0.958993 -0.212307      86.6526
8 1st rotation = 75.1077 order = 1
9 2nd rotation = 66.8997
10 3rd rotation = -76.5002
11 0.18777 0.958993 -0.212307      86.6526
12 1st rotation = 83.7441 order = 2
13 2nd rotation = 22.2818
14 3rd rotation = -2.62561
15 0.18777 0.958993 -0.212307      86.6526
16 1st rotation = -22.4269 order = 3
17 2nd rotation = -0.244254
18 3rd rotation = 84.2128
19 0.18777 0.958993 -0.212307      86.6526
20 1st rotation = -0.264239 order = 4
21 2nd rotation = -22.4267
22 3rd rotation = 84.3136
23 0.18777 0.958993 -0.212307      86.6526
24 1st rotation = 84.7402 order = 5
25 2nd rotation = -2.42944
26 3rd rotation = 22.303
27 0.18777 0.958993 -0.212307      86.6526
28 1st rotation = -22.4022 order = 6
29 2nd rotation = 84.2129
30 3rd rotation = -0.245506
31 0.18777 0.958993 -0.212307      86.6526
32 1st rotation = -112.402 order = 7
33 2nd rotation = -84.2129
34 3rd rotation = 89.7545
35 0.18777 0.958993 -0.212307      86.6526
36 1st rotation = 0.640217 order = 8
37 2nd rotation = -22.4282
38 3rd rotation = 83.621
39 0.18777 0.958993 -0.212307      86.6526
40 1st rotation = 90.6402 order = 9
41 2nd rotation = 22.4282
42 3rd rotation = -6.37896
43 0.18777 0.958993 -0.212307      86.6526
44 1st rotation = -2.4397 order = 10
45 2nd rotation = 84.745

```

Approved for public release; distribution is unlimited.

```

46 3rd rotation = 22.5265
47 0.18777 0.958993 -0.212307 86.6526
48 1st rotation = 87.5603 order = 11
49 2nd rotation = -84.745
50 3rd rotation = -67.4735
51 0.18777 0.958993 -0.212307 86.6526

```

We can also input an explicit rotation:

```

1 ./factor -13. 67. -23.

```

This gives the following results:

```

1 order = 1
2 The rotation is -0.33597 0.863186 -0.376875 73.8475
3
4 The following rotations should match this
5 1st rotation = -57.8081 order = 0
6 2nd rotation = 47.5373
7 3rd rotation = -55.6718
8 -0.33597 0.863186 -0.376875 73.8475
9 1st rotation = -13 order = 1
10 2nd rotation = 67
11 3rd rotation = -23
12 -0.33597 0.863186 -0.376875 73.8475
13 1st rotation = 67.5302 order = 2
14 2nd rotation = -5.04254
15 3rd rotation = -34.9977
16 -0.33597 0.863186 -0.376875 73.8475
17 1st rotation = -10.5973 order = 3
18 2nd rotation = -33.8845
19 3rd rotation = 62.7029
20 -0.33597 0.863186 -0.376875 73.8475
21 1st rotation = -34.3421 order = 4
22 2nd rotation = -8.78174
23 3rd rotation = 68.6579
24 -0.33597 0.863186 -0.376875 73.8475
25 1st rotation = 64.0087 order = 5
26 2nd rotation = -34.8426
27 3rd rotation = -6.14787
28 -0.33597 0.863186 -0.376875 73.8475
29 1st rotation = 5.45441 order = 6
30 2nd rotation = 67.6219
31 3rd rotation = -37.0797
32 -0.33597 0.863186 -0.376875 73.8475
33 1st rotation = -84.5456 order = 7
34 2nd rotation = -67.6219
35 3rd rotation = 52.9203
36 -0.33597 0.863186 -0.376875 73.8475
37 1st rotation = 74.6856 order = 8
38 2nd rotation = -35.3132
39 3rd rotation = -8.7461
40 -0.33597 0.863186 -0.376875 73.8475
41 1st rotation = -15.3144 order = 9
42 2nd rotation = -35.3132
43 3rd rotation = 81.2539
44 -0.33597 0.863186 -0.376875 73.8475
45 1st rotation = -37.756 order = 10
46 2nd rotation = 68.9201
47 3rd rotation = 9.4171
48 -0.33597 0.863186 -0.376875 73.8475
49 1st rotation = 52.244 order = 11
50 2nd rotation = -68.9201
51 3rd rotation = -80.5829
52 -0.33597 0.863186 -0.376875 73.8475

```

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).

INTENTIONALLY LEFT BLANK.

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: $\cos \frac{\theta}{2} + \hat{\mathbf{i}} \sin \frac{\theta}{2} \Leftrightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix}$ (F-1)

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

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

In general, if the quaternion is given by

$$q = q_0 + q_1 \hat{\mathbf{i}} + q_2 \hat{\mathbf{j}} + q_3 \hat{\mathbf{k}}, \quad (\text{F-4})$$

then the corresponding rotation matrix is

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

F-2. Rotation Matrix to Quaternion

Let the rotation be given by the matrix

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

Then a vector along the axis of rotation is

$$\mathbf{v} = (a_{32} - a_{23})\hat{\mathbf{i}} + (a_{13} - a_{31})\hat{\mathbf{j}} + (a_{21} - a_{12})\hat{\mathbf{k}}. \quad (\text{F-7})$$

If this vector turns out to be zero, then A is the identity matrix. Otherwise, a unit vector along the axis of rotation is

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

The rotation angle is

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

And the corresponding quaternion is

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

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

```

1 // convert.cpp: convert between Rotation, rotation matrix, and quaternion
2
3 #include "Rotation.h"
4 #include <iostream>
5 #include <cstdlib>
6 #include <iomanip>
7 using namespace va;
8
9 int main( int argc, char* argv[] ) {
10
11     // specify a Rotation from an axis vector and rotation angle
12     Vector u = normalize( Vector( 1., 1., 1. ) );
13     double th = rad( 120. );
14
15     if ( argc == 5 ) {
16
17         u = normalize( Vector( atof( argv[1] ), atof( argv[2] ), atof( argv[3] ) ) );
18         th = rad( atof( argv[4] ) );
19     }
20
21     std::cout << std::setprecision(6) << std::fixed << std::showpos;
22
23     Rotation R( u, th );
24     matrix A;
25     quaternion q;
26     std::cout << "Given the Rotation:" << std::endl;
27     std::cout << R << std::endl << std::endl;
28
29     // convert Rotation to a rotation matrix
30     std::cout << "convert Rotation to rotation matrix:" << std::endl;
31     A = to_matrix( R );
32     std::cout << A << std::endl << std::endl;
33
34     // convert a Rotation to a quaternion
35     std::cout << "convert Rotation to quaternion:" << std::endl;
36     q = to_quaternion( R );
37     std::cout << q << std::endl << std::endl;
38
39     std::cout << "Given the rotation matrix:" << std::endl;
40     std::cout << A << std::endl << std::endl;
41
42     //convert a rotation matrix to a Rotation

```

```

43     std::cout << "convert rotation matrix to Rotation:" << std::endl;
44     R = Rotation( A );
45     std::cout << R << std::endl << std::endl;
46
47     //convert a rotation matrix to a quaternion
48     std::cout << "convert rotation matrix to quaternion:" << std::endl;
49     q = to.quaternion( Rotation( A ) );
50     std::cout << q << std::endl << std::endl;;
51
52     std::cout << "Given the quaternion:" << std::endl;
53     std::cout << q << std::endl << std::endl;
54
55     // convert a quaternion to a Rotation
56     std::cout << "convert quaternion to Rotation:" << std::endl;
57     R = Rotation( q );
58     std::cout << R << std::endl << std::endl;
59
60     // convert a quaternion to a rotation matrix
61     std::cout << "convert quaternion to rotation matrix:" << std::endl;
62     A = to.matrix( Rotation( q ) );
63     std::cout << A << std::endl;
64
65     return EXIT_SUCCESS;
66 }

```

Compiling this program with

```
1 g++ -O2 -Wall -o convert convert.cpp -lm
```

and then running it via the command

```
1 ./convert 2.35 6.17 -4.6 35.6
```

produces the following output:

```

1 Given the Rotation:
2 +0.292041 +0.766762 -0.571654 +35.600000
3
4 convert Rotation to rotation matrix:
5 +0.829041 +0.374624 +0.415148
6 -0.290921 +0.922983 -0.251926
7 -0.477552 +0.088081 +0.874177
8
9 convert Rotation to quaternion:
10 +0.952129 +0.089275 +0.234396 -0.174752
11
12 Given the rotation matrix:
13 +0.829041 +0.374624 +0.415148
14 -0.290921 +0.922983 -0.251926
15 -0.477552 +0.088081 +0.874177
16
17 convert rotation matrix to Rotation:
18 +0.292041 +0.766762 -0.571654 +35.600000
19
20 convert rotation matrix to quaternion:
21 +0.952129 +0.089275 +0.234396 -0.174752
22
23 Given the quaternion:
24 +0.952129 +0.089275 +0.234396 -0.174752
25
26 convert quaternion to Rotation:
27 +0.292041 +0.766762 -0.571654 +35.600000
28
29 convert quaternion to rotation matrix:
30 +0.829041 +0.374624 +0.415148
31 -0.290921 +0.922983 -0.251926
32 -0.477552 +0.088081 +0.874177

```

Appendix G. Slerp (Spherical Linear Interpolation)

This is a derivation of the spherical linear interpolation (Slerp) formula that was introduced by Shoemake.¹ As depicted in Fig. G-1, the rotation that takes the unit vector $\hat{\mathbf{u}}_1$ to the unit vector $\hat{\mathbf{u}}_2$ is the unit quaternion

$$q \equiv \cos \frac{\theta}{2} + \hat{\mathbf{n}} \sin \frac{\theta}{2}, \quad (\text{G-1})$$

where θ is the angle between $\hat{\mathbf{u}}_1$ and $\hat{\mathbf{u}}_2$, and

$$\hat{\mathbf{n}} \equiv \frac{\hat{\mathbf{u}}_1 \times \hat{\mathbf{u}}_2}{\|\hat{\mathbf{u}}_1 \times \hat{\mathbf{u}}_2\|} \quad (\text{G-2})$$

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

$$\hat{\mathbf{u}}_2 = q\hat{\mathbf{u}}_1q^{-1}. \quad (\text{G-3})$$

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

$$q(t) \equiv \cos \frac{t\theta}{2} + \hat{\mathbf{n}} \sin \frac{t\theta}{2}. \quad (\text{G-4})$$

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

$$\hat{\mathbf{u}}(t) = q(t)\hat{\mathbf{u}}_1q(t)^{-1} = \left(\cos \frac{t\theta}{2} + \hat{\mathbf{n}} \sin \frac{t\theta}{2} \right) \hat{\mathbf{u}}_1 \left(\cos \frac{t\theta}{2} - \hat{\mathbf{n}} \sin \frac{t\theta}{2} \right). \quad (\text{G-5})$$

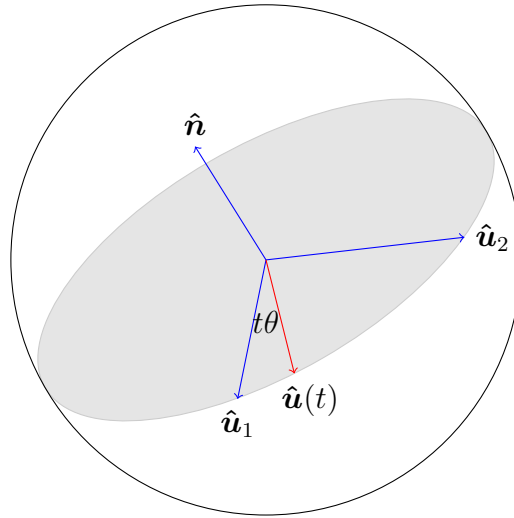


Fig. G-1. Spherical linear interpolation over the unit sphere

¹Shoemake K. Animating rotation with quaternion curves. SIGGRAPH '85; 1985;245–254.

Treating $\hat{\mathbf{u}}_1$ as the pure quaternion $(0, \hat{\mathbf{u}}_1)$ and using the fact that $\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{n}} = 0$, $\hat{\mathbf{u}}_2 \cdot \hat{\mathbf{n}} = 0$, $\hat{\mathbf{n}} \cdot (\hat{\mathbf{n}} \times \hat{\mathbf{u}}_1) = 0$, and $\|\hat{\mathbf{u}}_1 \times \hat{\mathbf{u}}_2\| = \sin \theta$, we can carry out the quaternion multiplication to get

$$\begin{aligned}
\hat{\mathbf{u}}(t) &= \left(\cos \frac{t\theta}{2} + \hat{\mathbf{n}} \sin \frac{t\theta}{2} \right) \hat{\mathbf{u}}_1 \left(\cos \frac{t\theta}{2} - \hat{\mathbf{n}} \sin \frac{t\theta}{2} \right) \\
&= \left(\cos \frac{t\theta}{2} \hat{\mathbf{u}}_1 + \hat{\mathbf{n}} \times \hat{\mathbf{u}}_1 \sin \frac{t\theta}{2} \right) \left(\cos \frac{t\theta}{2} - \hat{\mathbf{n}} \sin \frac{t\theta}{2} \right) \\
&= \cos^2 \frac{t\theta}{2} \hat{\mathbf{u}}_1 + \cos \frac{t\theta}{2} \sin \frac{t\theta}{2} \hat{\mathbf{n}} \times \hat{\mathbf{u}}_1 + \cos \frac{t\theta}{2} \sin \frac{t\theta}{2} \hat{\mathbf{n}} \times \hat{\mathbf{u}}_1 - \sin^2 \frac{t\theta}{2} (\hat{\mathbf{n}} \times \hat{\mathbf{u}}_1) \times \hat{\mathbf{n}} \\
&= \cos^2 \frac{t\theta}{2} \hat{\mathbf{u}}_1 + 2 \cos \frac{t\theta}{2} \sin \frac{t\theta}{2} \hat{\mathbf{n}} \times \hat{\mathbf{u}}_1 + \sin^2 \frac{t\theta}{2} \hat{\mathbf{n}} \times (\hat{\mathbf{n}} \times \hat{\mathbf{u}}_1). \tag{G-6}
\end{aligned}$$

Now

$$\begin{aligned}
\hat{\mathbf{n}} \times \hat{\mathbf{u}}_1 &= -\hat{\mathbf{u}}_1 \times \hat{\mathbf{n}} = -\frac{\hat{\mathbf{u}}_1 \times (\hat{\mathbf{u}}_1 \times \hat{\mathbf{u}}_2)}{\sin \theta} \\
&= -\frac{\hat{\mathbf{u}}_1(\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2) - \hat{\mathbf{u}}_2(\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_1)}{\sin \theta} \\
&= \frac{\hat{\mathbf{u}}_2 - \cos \theta \hat{\mathbf{u}}_1}{\sin \theta} \tag{G-7}
\end{aligned}$$

and

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

so that

$$\begin{aligned}
\hat{\mathbf{u}}(t) &= \left(\cos^2 \frac{t\theta}{2} - \sin^2 \frac{t\theta}{2} \right) \hat{\mathbf{u}}_1 + 2 \cos \frac{t\theta}{2} \sin \frac{t\theta}{2} \left(\frac{\hat{\mathbf{u}}_2 - \cos \theta \hat{\mathbf{u}}_1}{\sin \theta} \right) \\
&= \cos t\theta \hat{\mathbf{u}}_1 + \sin t\theta \left(\frac{\hat{\mathbf{u}}_2 - \cos \theta \hat{\mathbf{u}}_1}{\sin \theta} \right) \\
&= \frac{\cos t\theta \sin \theta \hat{\mathbf{u}}_1 + \sin t\theta \hat{\mathbf{u}}_2 - \sin t\theta \cos \theta \hat{\mathbf{u}}_1}{\sin \theta} \\
&= \frac{\sin(\theta - t\theta) \hat{\mathbf{u}}_1 + \sin t\theta \hat{\mathbf{u}}_2}{\sin \theta} \\
&= \frac{\sin(1-t)\theta}{\sin \theta} \hat{\mathbf{u}}_1 + \frac{\sin t\theta}{\sin \theta} \hat{\mathbf{u}}_2. \tag{G-9}
\end{aligned}$$

G-1. Slerp Formula

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

$$\hat{\mathbf{u}}(t) = \frac{\sin(1-t)\theta}{\sin \theta} \hat{\mathbf{u}}_1 + \frac{\sin t\theta}{\sin \theta} \hat{\mathbf{u}}_2, \quad (\text{G-10})$$

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

Listing G-1. slerp.cpp

```
1 // slerp.cpp: original slerp formula
2
3 #include "Vector.h"
4 #include <iostream>
5 #include <cstdlib>
6 using namespace va;
7
8 int main( int argc, char* argv[] ) {
9
10     Vector i( 1., 0., 0. ), j( 0., 1., 0. ), k( 0., 0., 1. );
11
12     Vector u1 = i; // default initial vector
13     Vector u2 = j; // default final vector
14     if ( argc > 1 ) { // or specify initial and final vectors on the command line
15
16         u1 = Vector( atof( argv[1] ), atof( argv[2] ), atof( argv[3] ) );
17         u2 = Vector( atof( argv[4] ), atof( argv[5] ), atof( argv[6] ) );
18     }
19
20     const int N = 1000;
21     const double THETA = acos( u1 * u2 );
22     const double A = 1. / sin( THETA );
23     Vector u;
24
25     double t;
26     for ( int n = 0; n < N; n++ ) {
27
28         t = double( n ) / double( N-1 );
29         u = A * ( sin( ( 1. - t ) * THETA ) * u1 + sin( t * THETA ) * u2 );
30         std::cout << u << std::endl;
31     }
32
33     return EXIT_SUCCESS;
34 }
```

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

$$\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2 = \cos \theta, \quad (\text{G-11})$$

¹Barrera T, Hast A, Bengtsson E. Incremental spherical linear interpolation. SIGRAD 2004. 2005;7–10.

we have¹

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

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

$$\begin{aligned}
[\hat{\mathbf{u}}_2 - (\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2) \hat{\mathbf{u}}_1] \cdot [\hat{\mathbf{u}}_2 - (\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2) \hat{\mathbf{u}}_1] &= 1 - (\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2)^2 - (\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2)^2 + (\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2)^2 \\
&= 1 - (\hat{\mathbf{u}}_1 \cdot \hat{\mathbf{u}}_2)^2. \tag{G-13}
\end{aligned}$$

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

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

Therefore, Eq. G-12 can be written as

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

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

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

for $n = 0, 1, 2, \dots, N - 1$.

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

Now we make use of the trigonometric identities

$$\begin{aligned}\cos(n+1)x + \cos(n-1)x &= 2\cos nx \cos x, \\ \sin(n+1)x + \sin(n-1)x &= 2\sin nx \cos x.\end{aligned}\tag{G-17}$$

Or, changing $n \rightarrow n-1$ and rearranging,

$$\begin{aligned}\cos nx &= 2\cos x \cos(n-1)x - \cos(n-2)x, \\ \sin nx &= 2\cos x \sin(n-1)x - \sin(n-2)x.\end{aligned}\tag{G-18}$$

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

$$\begin{aligned}\hat{\mathbf{u}}[n] &= [2\cos x \cos(n-1)x - \cos(n-2)x] \hat{\mathbf{u}}_1 + \\ &\quad [2\cos x \sin(n-1)x - \sin(n-2)x] \hat{\mathbf{u}}_0 \\ &= 2\cos x [\cos(n-1)x \hat{\mathbf{u}}_1 + \sin(n-1)x \hat{\mathbf{u}}_0] - \\ &\quad [\cos(n-2)x \hat{\mathbf{u}}_1 + \sin(n-2)x \hat{\mathbf{u}}_0] \\ &= 2\cos x \hat{\mathbf{u}}[n-1] - \hat{\mathbf{u}}[n-2].\end{aligned}\tag{G-19}$$

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

$$\hat{\mathbf{u}}[0] = \hat{\mathbf{u}}_1 \quad \text{and} \tag{G-20}$$

$$\hat{\mathbf{u}}[1] = \cos x \hat{\mathbf{u}}_1 + \sin x \hat{\mathbf{u}}_0. \tag{G-21}$$

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

Listing G-2. fast_slerp.cpp

```

1 // fast_slerp.cpp: fast incremental slerp evaluates trig functions only once
2 // R. Saucier, June 2016
3
4 #include "Vector.h"
5 #include <iostream>
6 #include <cstdlib>
7 using namespace va;
8
9 int main( int argc, char* argv[] ) {
10
11     Vector i( 1., 0., 0. ), j( 0., 1., 0. ), k( 0., 0., 1. );
12
13     Vector u1 = i; // default initial vector
14     Vector u2 = j; // default final vector
15     if ( argc > 1 ) { // or specify initial and final vectors on the command line
16
17         u1 = Vector( atof( argv[1] ), atof( argv[2] ), atof( argv[3] ) );
18         u2 = Vector( atof( argv[4] ), atof( argv[5] ), atof( argv[6] ) );
19     }
20
21     const int N = 1000;
22     const double U12 = u1 * u2;
23     const double THETA = acos( U12 );
24     const double TH = THETA / double( N-1 );
25     const double C = 2. * cos( TH );

```

Approved for public release; distribution is unlimited.

```

26     const Vector U0    = ( u2 - U12 * u1 ) / sqrt( ( 1. - U12 ) * ( 1. + U12 ) );
27
28     Vector u_2, u_1, u;
29
30     for ( int n = 0; n < N; n++ ) {
31
32         if ( n == 0 ) u = u_2 = u1;                                // n = 0 will become u at n - 2;
33         else if ( n == 1 ) u = u_1 = cos( TH ) * u1 + sin( TH ) * U0; // n = 1 will become u at n - 1
34         else {
35             u    = C * u_1 - u_2;
36             u_2 = u_1;
37             u_1 = u;
38         }
39         std::cout << u << std::endl;
40     }
41
42     return EXIT_SUCCESS;
43 }

```

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 Boulton.¹ Given 2 sets of 3 linearly independent vectors, $\{\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3\}$ and $\{\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3\}$, 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

$$\mathbf{b}_i = q\mathbf{a}_iq^{-1} \quad \text{for } i = 1, 2, 3 \quad (\text{H-1})$$

where $q = q_0 + q_1\hat{\mathbf{i}} + q_2\hat{\mathbf{j}} + q_3\hat{\mathbf{k}}$ and $q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1$. Using $\hat{\mathbf{i}}^2 = \hat{\mathbf{j}}^2 = \hat{\mathbf{k}}^2 = \hat{\mathbf{i}}\hat{\mathbf{j}}\hat{\mathbf{k}} = -1$, $q^{-1} = q_0 - q_1\hat{\mathbf{i}} - q_2\hat{\mathbf{j}} - q_3\hat{\mathbf{k}}$, and expanding, we get

$$b_x = a_x q_0^2 + 2a_z q_0 q_2 - 2a_y q_0 q_3 + a_x q_1^2 + 2a_y q_1 q_2 + 2a_z q_1 q_3 - a_x q_2^2 - a_x q_3^2 \quad (\text{H-2})$$

$$b_y = a_y q_0^2 - 2a_z q_0 q_1 + 2a_x q_0 q_3 - a_y q_1^2 + 2a_x q_1 q_2 + a_y q_2^2 + 2a_z q_2 q_3 - a_y q_3^2 \quad (\text{H-3})$$

$$b_z = a_z q_0^2 + 2a_y q_0 q_1 - 2a_x q_0 q_2 - a_z q_1^2 + 2a_x q_1 q_3 - a_z q_2^2 + 2a_y q_2 q_3 + a_z q_3^2 \quad (\text{H-4})$$

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

$$\begin{bmatrix} a_{1x} & 0 & 2a_{1z} & -2a_{1y} & a_{1x} & 2a_{1y} & 2a_{1z} & -a_{1x} & 0 & -a_{1x} \\ a_{1y} & -2a_{1z} & 0 & 2a_{1x} & -a_{1y} & 2a_{1x} & 0 & a_{1y} & 2a_{1z} & -a_{1y} \\ a_{1z} & 2a_{1y} & -2a_{1x} & 0 & -a_{1z} & 0 & 2a_{1x} & -a_{1z} & 2a_{1y} & a_{1z} \\ a_{2x} & 0 & 2a_{2z} & -2a_{2y} & a_{2x} & 2a_{2y} & 2a_{2z} & -a_{2x} & 0 & -a_{2x} \\ a_{2y} & -2a_{2z} & 0 & 2a_{2x} & -a_{2y} & 2a_{2x} & 0 & a_{2y} & 2a_{2z} & -a_{2y} \\ a_{2z} & 2a_{2y} & -2a_{2x} & 0 & -a_{2z} & 0 & 2a_{2x} & -a_{2z} & 2a_{2y} & a_{2z} \\ a_{3x} & 0 & 2a_{3z} & -2a_{3y} & a_{3x} & 2a_{3y} & 2a_{3z} & -a_{3x} & 0 & -a_{3x} \\ a_{3y} & -2a_{3z} & 0 & 2a_{3x} & -a_{3y} & 2a_{3x} & 0 & a_{3y} & 2a_{3z} & -a_{3y} \\ a_{3z} & 2a_{3y} & -2a_{3x} & 0 & -a_{3z} & 0 & 2a_{3x} & -a_{3z} & 2a_{3y} & a_{3z} \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} q_0^2 \\ q_0 q_1 \\ q_0 q_2 \\ q_0 q_3 \\ q_1^2 \\ q_1 q_2 \\ q_1 q_3 \\ q_2^2 \\ q_2 q_3 \\ q_3^2 \end{bmatrix} = \begin{bmatrix} b_{1x} \\ b_{1y} \\ b_{1z} \\ b_{2x} \\ b_{2y} \\ b_{2z} \\ b_{3x} \\ b_{3y} \\ b_{3z} \\ 1 \end{bmatrix} \quad (\text{H-5})$$

The 10×10 coefficient matrix can be inverted with 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:

$$q_0^2 = \frac{\det(\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3) + \det(\mathbf{b}_1, \mathbf{a}_2, \mathbf{a}_3) + \det(\mathbf{a}_1, \mathbf{b}_2, \mathbf{a}_3) + \det(\mathbf{a}_1, \mathbf{a}_2, \mathbf{b}_3)}{4 \det(\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3)} \quad (\text{H-6})$$

$$q_1^2 = \frac{\det(\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3) + \det(P_{11}\mathbf{b}_1, \mathbf{a}_2, \mathbf{a}_3) + \det(\mathbf{a}_1, P_{11}\mathbf{b}_2, \mathbf{a}_3) + \det(\mathbf{a}_1, \mathbf{a}_2, P_{11}\mathbf{b}_3)}{4 \det(\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3)} \quad (\text{H-7})$$

$$q_2^2 = \frac{\det(\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3) + \det(P_{22}\mathbf{b}_1, \mathbf{a}_2, \mathbf{a}_3) + \det(\mathbf{a}_1, P_{22}\mathbf{b}_2, \mathbf{a}_3) + \det(\mathbf{a}_1, \mathbf{a}_2, P_{22}\mathbf{b}_3)}{4 \det(\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3)} \quad (\text{H-8})$$

$$q_3^2 = \frac{\det(\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3) + \det(P_{33}\mathbf{b}_1, \mathbf{a}_2, \mathbf{a}_3) + \det(\mathbf{a}_1, P_{33}\mathbf{b}_2, \mathbf{a}_3) + \det(\mathbf{a}_1, \mathbf{a}_2, P_{33}\mathbf{b}_3)}{4 \det(\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3)} \quad (\text{H-9})$$

$$q_i q_j = \frac{\det(P_{ij}\mathbf{b}_1, \mathbf{a}_2, \mathbf{a}_3) + \det(\mathbf{a}_1, P_{ij}\mathbf{b}_2, \mathbf{a}_3) + \det(\mathbf{a}_1, \mathbf{a}_2, P_{ij}\mathbf{b}_3)}{4 \det(\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3)} \quad (\text{H-10})$$

¹Micheals RJ, Boulton TE. Increasing robustness in self-localization and pose estimation. [date unknown; accessed 2010 Jun]. <http://www.vast.uccs.edu/~tboulton/PAPERS/SPIE99-Increasing-robustness-in-self-localization-and-pose-estimation-Micheals-Boulton.pdf>.

Approved for public release; distribution is unlimited.

where

$$\begin{aligned}
\det(\mathbf{a}, \mathbf{b}, \mathbf{c}) &= \det \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{bmatrix} \\
&= a_x(b_y c_z - b_z c_y) + a_y(b_z c_x - b_x c_z) + a_z(b_x c_y - b_y c_x) \\
&= \mathbf{a} \cdot (\mathbf{b} \times \mathbf{c}).
\end{aligned} \tag{H-11}$$

and

$$P_{11} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \quad P_{22} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \quad P_{33} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{H-12}$$

$$P_{01} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}, \quad P_{02} = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad P_{03} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{H-13}$$

$$P_{12} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad P_{13} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad P_{23} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{H-14}$$

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

$$\hat{\mathbf{u}} = \frac{q_1 \hat{\mathbf{i}} + q_2 \hat{\mathbf{j}} + q_3 \hat{\mathbf{k}}}{\sqrt{1 - q_0^2}}, \tag{H-15}$$

and the angle of rotation is

$$\theta = 2 \cos^{-1} q_0. \tag{H-16}$$

The full rotation matrix is

$$R = \begin{bmatrix} 2q_0^2 - 1 + 2q_1^2 & 2q_1 q_2 - 2q_0 q_3 & 2q_1 q_3 + 2q_0 q_2 \\ 2q_1 q_2 + 2q_0 q_3 & 2q_0^2 - 1 + 2q_2^2 & 2q_2 q_3 - 2q_0 q_1 \\ 2q_1 q_3 - 2q_0 q_2 & 2q_2 q_3 + 2q_0 q_1 & 2q_0^2 - 1 + 2q_3^2 \end{bmatrix}. \tag{H-17}$$

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

Listing H-1. ao.cpp

```
1 // ao.cpp: test of absolute orientation as implemented in Rotation class
2
3 #include "Rotation.h"
4 #include <iostream>
5 #include <cstdlib>
6 #include <iomanip>
7 using namespace va; // vector algebra namespace
8
9 int main( int argc, char* argv[] ) {
10
11     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
12     Vector b1, b2, b3;
13
14     double pitch = rad( 45. ); // default pitch (deg converted to radians)
15     double yaw = rad( -30. ); // default yaw (deg converted to radians)
16     double roll = rad( 60. ); // default roll (deg converted to radians)
17     if ( argc == 4 ) { // or specify pitch, yaw, roll (deg) on command line
18
19         pitch = rad( atof( argv[1] ) );
20         yaw = rad( atof( argv[2] ) );
21         roll = rad( atof( argv[3] ) );
22     }
23
24     std::cout << std::setprecision(6) << std::fixed << std::showpos;
25     std::cout << "The 3 vectors are linearly independent iff det(a1,a2,a3) is non-zero: ";
26     std::cout << "det(a1,a2,a3) = " << ( a1 * ( a2 ^ a3 ) ) << std::endl << std::endl;
27
28     // perform rotation sequence on initial vectors
29     Rotation R1( pitch, yaw, roll, XYZ );
30     b1 = R1 * a1;
31     b2 = R1 * a2;
32     b3 = R1 * a3;
33
34     // output the rotated vectors
35     std::cout << "The rotated vectors are:" << std::endl;
36     std::cout << "b1 = " << b1 << std::endl;
37     std::cout << "b2 = " << b2 << std::endl;
38     std::cout << "b3 = " << b3 << std::endl << std::endl;
39
40     // given only the two sets of vectors, find the rotation that takes {a1,a2,a3} to {b1,b2,b3}
41     Rotation R2( a1, a2, a3, b1, b2, b3 );
42
43     // apply this rotation to the original vectors
44     b1 = R2 * a1;
45     b2 = R2 * a2;
46     b3 = R2 * a3;
47
48     // output rotated vectors to show they match previous output
49     std::cout << "The following vectors should match those above:" << std::endl;
50     std::cout << "b1 = " << b1 << std::endl;
51     std::cout << "b2 = " << b2 << std::endl;
52     std::cout << "b3 = " << b3 << std::endl << std::endl;
53
54     // factor this rotation into a pitch-yaw-roll rotation sequence
55     sequence s = factor( R2, XYZ );
56
57     // output rotation sequence to show it matches the input values
58     std::cout << "Factoring this rotation into a rotation sequence gives:" << std::endl;
59     std::cout << "pitch = " << deg( s.first ) << std::endl;
60     std::cout << "yaw = " << deg( s.second ) << std::endl;
61     std::cout << "roll = " << deg( s.third ) << std::endl;
62
63     return EXIT_SUCCESS;
64 }
```

Compiling this program with

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

and then running it with the command

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

Approved for public release; distribution is unlimited.

prints out the following:

```
1 The 3 vectors are linearly independent iff det(a1,a2,a3) is non-zero: det(a1,a2,a3) = +143.826000
2
3 The rotated vectors are:
4 b1 = +2.917177 -2.334937 +2.139660
5 b2 = +6.633337 +1.253165 +3.390932
6 b3 = -2.129101 -2.066399 +6.798303
7
8 The following vectors should match those above:
9 b1 = +2.917177 -2.334937 +2.139660
10 b2 = +6.633337 +1.253165 +3.390932
11 b3 = -2.129101 -2.066399 +6.798303
12
13 Factoring this rotation into a rotation sequence gives:
14 pitch = -35.200000
15 yaw = +43.500000
16 roll = -75.600000
```

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 $\det(\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3) \neq 0$. 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.

INTENTIONALLY LEFT BLANK.

1 DEFENSE TECHNICAL
(PDF) INFORMATION CTR
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
(11 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

INTENTIONALLY LEFT BLANK.