Freescale Semiconductor
Application Note

Document Number:AN4676
Rev. 1.0, 4/2013

Euler Angle, Rotation Matrix and Quaternion Representations of Orientation

in Aerospace, Android and Windows 8 Coordinates

by: Mark Pedley

Contents

1 Introduction

This Application Note documents the mathematics of describing orientation in the Aerospace/NED, Android and Windows 8 coordinate systems in terms of Euler angle, rotation matrix, quaternion and rotation vector representations. This Application Note is part of the documentation for Freescale's Xtrinsic eCompass and Magnetic Calibration software provided under license at www.freescale.com/ecompass. Its use and distribution are controlled by the license agreement.

1.1 Mathematical glossary

The following are used in this application note:

$a = a_0 + a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k}$	Components of quaternion a
$a^* = a_0 - a_1\boldsymbol{i} - a_2\boldsymbol{j} - a_3\boldsymbol{k}$	Quaternion conjugate of a
$a = \{a_0, \boldsymbol{a}\}$	Scalar and vector representation of quaternion a
$a^{-1} = \frac{a^*}{N(a)^2}$	Inverse or reciprocal of quaternion a
a_0	Scalar component of quaternion a
$\boldsymbol{a} = a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k}$	Vector component of quaternion a
$\boldsymbol{a} \cdot \boldsymbol{b}$	Scalar product of vectors $\boldsymbol{a}$and $\boldsymbol{b}$
$\boldsymbol{a} \times \boldsymbol{b}$	Vector product of vectors $\boldsymbol{a}$and $\boldsymbol{b}$
$\hat{\boldsymbol{n}}$	Unit vector representing rotation axis
$N(a) = \sqrt{a_0{}^2 + a_1{}^2 + a_2{}^2 + a_3{}^2}$	Norm or magnitude of quaternion a
q_x, q_y, q_z	Rotation quaternions about the *x*, *y*, and *z* axes
$\boldsymbol{R}_x, \boldsymbol{R}_y, \boldsymbol{R}_z$	Rotation matrices around the *x*, *y*, and *z* axes
r	Real number
z	Complex number
η	General rotation angle
ϕ	Roll angle
θ	Pitch angle
ψ	Yaw angle
ρ	Compass heading angle

2 Introduction to Quaternion Algebra

2.1 Definition

Quaternions form a class of four-component numbers also termed hyper-complex numbers. A complex number z has two (real and imaginary) components:

$$z = a + bi \tag{1}$$

A quaternion a has four components:

$$a = a_0 + a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k} = \{a_0, a_1, a_2, a_3\} \tag{2}$$

where a_0, a_1, a_2, and a_3 are real numbers.

a_0 is termed the scalar component and $\boldsymbol{a} = a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k}$ the vector component of the quaternion.

Equivalent representations of the quaternion in its scalar and vector components are:

$$a = a_0 + \boldsymbol{a} = \{a_0, \boldsymbol{a}\} = \left\{a_0, \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}\right\} \tag{3}$$

If the scalar component a_0 is zero, the quaternion is termed a pure quaternion or vector. If the vector component $\boldsymbol{a}$ is zero, then the quaternion is simply a real number.

2.2 Equality of two quaternions

Two quaternions a and b are equal if and only if all their components are equal:

$$a = b \;\Rightarrow\; a_0 = b_0, a_1 = b_1, a_2 = b_2 \; and \; a_3 = b_3 \tag{4}$$

2.3 Addition of two quaternions

Addition of quaternions is defined as the addition of the four components:

$$a + b = (a_0 + b_0) + (a_1 + b_1)\boldsymbol{i} + (a_2 + b_2)\boldsymbol{j} + (a_3 + b_3)\boldsymbol{k} \tag{5}$$

A consequence of this definition is that quaternion addition is commutative and associative since the addition of real numbers is commutative and associative:

$$a + b = b + a \tag{6}$$

$$(a + b) + c = a + (b + c) \tag{7}$$

2.4 Product of two quaternions

The product of two quaternions is defined to be the distributive product of the components:

$$c = ab = (a_0 + a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k})(b_0 + b_1\boldsymbol{i} + b_2\boldsymbol{j} + b_3\boldsymbol{k}) \tag{8}$$

$$\begin{aligned} &= a_0(b_0 + b_1\boldsymbol{i} + b_2\boldsymbol{j} + b_3\boldsymbol{k}) + a_1\boldsymbol{i}(b_0 + b_1\boldsymbol{i} + b_2\boldsymbol{j} + b_3\boldsymbol{k}) \\ &+ a_2\boldsymbol{j}(b_0 + b_1\boldsymbol{i} + b_2\boldsymbol{j} + b_3\boldsymbol{k}) + a_3\boldsymbol{k}(b_0 + b_1\boldsymbol{i} + b_2\boldsymbol{j} + b_3\boldsymbol{k} \end{aligned} \tag{9}$$

$$\begin{aligned} &= (a_0b_0 + a_0b_1\boldsymbol{i} + a_0b_2\boldsymbol{j} + a_0b_3\boldsymbol{k}) + (a_1b_0\boldsymbol{i} + a_1b_1\boldsymbol{ii} + a_1b_2\boldsymbol{ij} + a_1b_3\boldsymbol{ik}) \\ &+ (a_2b_0\boldsymbol{j} + a_2b_1\boldsymbol{ji} + a_2b_2\boldsymbol{jj} + a_2b_3\boldsymbol{jk}) + (a_3b_0\boldsymbol{k} + a_3b_1\boldsymbol{ki} + a_3b_2\boldsymbol{kj} + a_3b_3\boldsymbol{kk}) \end{aligned} \tag{10}$$

The products of components of a quaternion are defined to satisfy (where r is any real number):

$$r\boldsymbol{i} = \boldsymbol{i}r \tag{11}$$

$$r\boldsymbol{j} = \boldsymbol{j}r \tag{12}$$

$$r\boldsymbol{k} = \boldsymbol{k}r \tag{13}$$

$$\boldsymbol{ii} = \boldsymbol{jj} = \boldsymbol{kk} = -1 \tag{14}$$

$$\boldsymbol{ij} = -\boldsymbol{ji} = \boldsymbol{k} \tag{15}$$

$$\boldsymbol{jk} = -\boldsymbol{kj} = \boldsymbol{i} \tag{16}$$

$$\boldsymbol{ki} = -\boldsymbol{ik} = \boldsymbol{j} \tag{17}$$

A consequence of equations (14) to (17) is that:

$$(\boldsymbol{ij})\boldsymbol{k} = \boldsymbol{i}(\boldsymbol{jk}) = -1 \tag{18}$$

Substitution of equations (11) to (18) into equation (10) simplifies the quaternion product to:

$$\begin{aligned} c = ab = (a_0b_0 - a_1b_1 - a_2b_2 - a_3b_3) + (a_0b_1 + a_1b_0 + a_2b_3 - a_3b_2)\boldsymbol{i} \\ +(a_0b_2 - a_1b_3 + a_2b_0 + a_3b_1)\boldsymbol{j} + (a_0b_3 + a_1b_2 - a_2b_1 + a_3b_0)\boldsymbol{k} \end{aligned} \tag{19}$$

The four components of the product quaternion *c* are therefore:

$$c_0 = a_0b_0 - a_1b_1 - a_2b_2 - a_3b_3 \tag{20}$$

$$c_1 = a_0b_1 + a_1b_0 + a_2b_3 - a_3b_2 \tag{21}$$

$$c_2 = a_0b_2 - a_1b_3 + a_2b_0 + a_3b_1 \tag{22}$$

$$c_3 = a_0b_3 + a_1b_2 - a_2b_1 + a_3b_0 \tag{23}$$

Examination of equations (20) to (23) shows that quaternion multiplication does not commute:

$$ab \neq ba \tag{24}$$

Brute force evaluation proves that quaternion multiplication is associative.

$$\begin{aligned} (ab)c - a(bc) &= \{(a_0 + a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k})(b_0 + b_1\boldsymbol{i} + b_2\boldsymbol{j} + b_3\boldsymbol{k})\}(c_0 + c_1\boldsymbol{i} + c_2\boldsymbol{j} + c_3\boldsymbol{k}) \\ &- (a_0 + a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k})\{(b_0 + b_1\boldsymbol{i} + b_2\boldsymbol{j} + b_3\boldsymbol{k})(c_0 + c_1\boldsymbol{i} + c_2\boldsymbol{j} + c_3\boldsymbol{k})\} \\ &= 0 \end{aligned} \tag{25}$$

$$\Rightarrow (ab)c = a(bc) \tag{26}$$

2.5 Product of quaternion and scalar

A special case of the quaternion product occurs when one of the quaternions has zero vector components and is a scalar.

If *a* is a scalar so that $a = a_0$ then:

$$ab = a_0 b_0 + a_0 b_1 \boldsymbol{i} + a_0 b_2 \boldsymbol{j} + a_0 b_3 \boldsymbol{k} \quad (27)$$

$$ba = b_0 a_0 + b_1 a_0 \boldsymbol{i} + b_2 a_0 \boldsymbol{j} + b_3 a_0 \boldsymbol{k} = a_0 b_0 + a_0 b_1 \boldsymbol{i} + a_0 b_2 \boldsymbol{j} + a_0 b_3 \boldsymbol{k} \quad (28)$$

$$\Rightarrow ab = ba \; iff \; a \; is \; a \; scalar \quad (29)$$

The product of a quaternion with a scalar quaternion does therefore commute.

2.6 Product of a quaternion with a vector

The product of a quaternion with a vector or pure quaternion results in a general quaternion and not another vector (since the scalar component of the product is non-zero):

$$a\boldsymbol{b} = (-a_1 b_1 - a_2 b_2 - a_3 b_3) + (a_0 b_1 + a_2 b_3 - a_3 b_2)\boldsymbol{i} + (a_0 b_2 - a_1 b_3 + a_3 b_1)\boldsymbol{j} + (a_0 b_3 + a_1 b_2 - a_2 b_1)\boldsymbol{k} \quad (30)$$

$$\boldsymbol{b}a = (-b_1 a_1 - b_2 a_2 - b_3 a_3) + (b_1 a_0 + b_2 a_3 - b_3 a_2)\boldsymbol{i} + (-b_1 a_3 + b_2 a_0 + b_3 a_1)\boldsymbol{j} + (b_1 a_2 - b_2 a_1 + b_3 a_0)\boldsymbol{k} \quad (31)$$

By inspection of equations **(30)** and **(31)** , the product of the quaternion and vector does not commute:

$$a\boldsymbol{b} \neq \boldsymbol{b}a \quad (32)$$

2.7 Quaternion conjugate

The quaternion conjugate a^* is defined as:

$$a^* = a_0 - a_1 \boldsymbol{i} - a_2 \boldsymbol{j} - a_3 \boldsymbol{k} \quad (33)$$

From the definition of the quaternion product ab it can be shown that $(ab)^* = b^* a^*$:

$$\begin{aligned}(ab)^* = {} & (a_0 b_0 - a_1 b_1 - a_2 b_2 - a_3 b_3) - (a_0 b_1 + a_1 b_0 + a_2 b_3 - a_3 b_2)\boldsymbol{i} \\ & -(a_0 b_2 - a_1 b_3 + a_2 b_0 + a_3 b_1)\boldsymbol{j} - (a_0 b_3 + a_1 b_2 - a_2 b_1 + a_3 b_0)\boldsymbol{k}\end{aligned} \quad (34)$$

The product $b^* a^*$:

$$b^* a^* = (b_0 - b_1 \boldsymbol{i} - b_2 j - b_3 \boldsymbol{k})(a_0 - a_1 \boldsymbol{i} - a_2 \boldsymbol{j} - a_3 \boldsymbol{k}) \quad (35)$$

$$\begin{aligned}&= b_0 a_0 - b_0 a_1 \boldsymbol{i} - b_0 a_2 \boldsymbol{j} - b_0 a_3 \boldsymbol{k} - b_1 \boldsymbol{i} a_0 + b_1 \boldsymbol{i} a_1 \boldsymbol{i} + b_1 \boldsymbol{i} a_2 \boldsymbol{j} + b_1 \boldsymbol{i} a_3 \boldsymbol{k} \\ &- b_2 \boldsymbol{j} a_0 + b_2 \boldsymbol{j} a_1 \boldsymbol{i} + b_2 \boldsymbol{j} a_2 \boldsymbol{j} + b_2 \boldsymbol{j} a_3 \boldsymbol{k} - b_3 \boldsymbol{k} a_0 + b_3 \boldsymbol{k} a_1 \boldsymbol{i} + b_3 \boldsymbol{k} a_2 \boldsymbol{j} + b_3 \boldsymbol{k} a_3 \boldsymbol{k}\end{aligned} \quad (36)$$

$$\begin{aligned}&= b_0 a_0 - b_0 a_1 \boldsymbol{i} - b_0 a_2 \boldsymbol{j} - b_0 a_3 \boldsymbol{k} - b_1 a_0 \boldsymbol{i} - b_1 a_1 + b_1 a_2 \boldsymbol{k} - b_1 a_3 \boldsymbol{j} \\ &\; - b_2 a_0 \boldsymbol{j} - b_2 a_1 \boldsymbol{k} - b_2 a_2 + b_2 a_3 \boldsymbol{i} - b_3 a_0 \boldsymbol{k} + b_3 a_1 j - b_3 a_2 \boldsymbol{i} - b_3 a_3\end{aligned} \quad (37)$$

$$\begin{aligned}&= (b_0 a_0 - b_1 a_1 - b_2 a_2 - b_3 a_3) - (b_1 a_0 + b_0 a_1 + b_3 a_2 - b_2 a_3)\boldsymbol{i} \\ &\; -(b_2 a_0 - b_3 a_1 + b_0 a_2 + b_1 a_3)\boldsymbol{j} - (\, b_3 a_0 + b_2 a_1 - b_1 a_2 + b_0 a_3)\boldsymbol{k}\end{aligned} \quad (38)$$

$$\Rightarrow (ab)^* = b^* a^* \quad (39)$$

Simple extension to higher order products gives:

$$(abc \ldots z)^* = z^* \ldots c^* b^* a^* \tag{40}$$

The sum of the quaternion a and its conjugate a^* is the scalar $2a_0$:

$$a + a^* = (a_0 + a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k}) + (a_0 - a_1\boldsymbol{i} - a_2\boldsymbol{j} - a_3\boldsymbol{k}) = 2a_0 \tag{41}$$

2.8 Quaternion norm

The quaternion norm or magnitude $N(a)$ is defined as:

$$N(a) = \sqrt{{a_0}^2 + {a_1}^2 + {a_2}^2 + {a_3}^2} \tag{42}$$

The products of a quaternion a with its conjugate a^* evaluate to:

$$a^*a = (a_0 - a_1\boldsymbol{i} - a_2\boldsymbol{j} - a_3\boldsymbol{k})(a_0 + a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k}) \tag{43}$$

$$\begin{aligned} &= a_0a_0 + a_0a_1\boldsymbol{i} + a_0a_2\boldsymbol{j} + a_0a_3\boldsymbol{k} - a_1a_0\boldsymbol{i} + a_1a_1 - a_1a_2\boldsymbol{k} + a_1a_3\boldsymbol{j} \\ &\quad -a_2a_0\boldsymbol{j} + a_2a_1\boldsymbol{k} + a_2a_2 - a_2a_3\boldsymbol{i} - a_3a_0\boldsymbol{k} - a_3a_1\boldsymbol{j} + a_3a_2\boldsymbol{i} + a_3a_3 \end{aligned} \tag{44}$$

$$= {a_0}^2 + {a_1}^2 + {a_2}^2 + {a_3}^2 = N(a)^2 \tag{45}$$

$$aa^* = (a_0 + a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k})(a_0 - a_1\boldsymbol{i} - a_2\boldsymbol{j} - a_3\boldsymbol{k}) \tag{46}$$

$$\begin{aligned} &= a_0a_0 - a_0a_1\boldsymbol{i} - a_0a_2\boldsymbol{j} - a_0a_3\boldsymbol{k} + a_1a_0\boldsymbol{i} + a_1a_1 - a_1a_2\boldsymbol{k} + a_1a_3\boldsymbol{j} \\ &\quad +a_2a_0\boldsymbol{j} + a_2a_1\boldsymbol{k} + a_2a_2 - a_2a_3\boldsymbol{i} + a_3a_0\boldsymbol{k} - a_3a_1\boldsymbol{j} + a_3a_2\boldsymbol{i} + a_3a_3 \end{aligned} \tag{47}$$

$$= {a_0}^2 + {a_1}^2 + {a_2}^2 + {a_3}^2 = N(a)^2 \tag{48}$$

$$\Rightarrow N(a) = \sqrt{{a_0}^2 + {a_1}^2 + {a_2}^2 + {a_3}^2} = \sqrt{a^*a} = \sqrt{aa^*} \tag{49}$$

The norm of a quaternion conjugate equals the norm of the quaternion:

$$N(a^*) = \sqrt{{a_0}^2 + (-a_1)^2 + (-a_2)^2 + (-a_3)^2} = \sqrt{{a_0}^2 + {a_1}^2 + {a_2}^2 + {a_3}^2} = N(a) \tag{50}$$

The norm of the product of two quaternions is the product of the individual quaternion norms:

$$N(ab) = \sqrt{\begin{aligned} &(a_0b_0 - a_1b_1 - a_2b_2 - a_3b_3)^2 + (a_0b_1 + a_1b_0 + a_2b_3 - a_3b_2)^2 \\ &+(a_0b_2 - a_1b_3 + a_2b_0 + a_3b_1)^2 + (a_0b_3 + a_1b_2 - a_2b_1 + a_3b_0)^2 \end{aligned}} \tag{51}$$

$$= \sqrt{({a_0}^2 + {a_1}^2 + {a_2}^2 + {a_3}^2)({b_0}^2 + {b_1}^2 + {b_2}^2 + {b_3}^2)} = N(a)N(b) \tag{52}$$

2.9 Quaternion inverse and division

The quaternion inverse a^{-1} is defined to be the quaternion which satisfies:

$$aa^{-1} = a^{-1}a = 1 \tag{53}$$

Pre- and post-multiplication of a by $\frac{a^*}{N(a)^2}$ evaluates to:

$$\frac{1}{N(a)^2}(a_0 - a_1\boldsymbol{i} - a_2\boldsymbol{j} - a_3\boldsymbol{k})(a_0 + a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k}) = \frac{({a_0}^2 + {a_1}^2 + {a_2}^2 + {a_3}^2)}{N(a)^2} = 1 \tag{54}$$

$$(a_0 + a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k})\frac{1}{N(a)^2}(a_0 - a_1\boldsymbol{i} - a_2\boldsymbol{j} - a_3\boldsymbol{k}) = \frac{({a_0}^2 + {a_1}^2 + {a_2}^2 + {a_3}^2)}{N(a)^2} = 1 \tag{55}$$

The quaternion inverse a^{-1} is therefore identified for all quaternions with non-zero norm as:

$$a^{-1} = \frac{a^*}{N(a)^2} = \left(\frac{a_0}{N(a)^2}\right) - \left(\frac{a_1}{N(a)^2}\right)\boldsymbol{i} - \left(\frac{a_2}{N(a)^2}\right)\boldsymbol{j} - \left(\frac{a_3}{N(a)^2}\right)\boldsymbol{k} \tag{56}$$

The norm of a quaternion inverse equals the reciprocal of the quaternion norm. The norm and reciprocation operations therefore commute.

$$N(a^{-1}) = N\left(\frac{a^*}{N(a)^2}\right) = \left(\frac{1}{N(a)^2}\right)N(a^*) = \left(\frac{N(a)}{N(a)^2}\right) = \frac{1}{N(a)} \tag{57}$$

2.10 Vector representation of quaternion product

The standard scalar and vector products between the two vectors $\boldsymbol{a} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}$ and $\boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$ are defined as:

$$\boldsymbol{a}.\boldsymbol{b} = a_1b_1 + a_2b_2 + a_3b_3 \tag{58}$$

$$\boldsymbol{a} \times \boldsymbol{b} = \begin{pmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{pmatrix} \tag{59}$$

The product between two quaternions $a = (a_0, \boldsymbol{a})$ and $b = (b_0, \boldsymbol{b})$ can be written in an alternative form involving scalar and vector products on their vector components. Direction expansion of the scalar and vector expression below shows it equals the quaternion product ab:

$$\{a_0b_0 - \boldsymbol{a}.\boldsymbol{b}, a_0\boldsymbol{b} + b_0\boldsymbol{a} + \boldsymbol{a}\times\boldsymbol{b}\} = \left\{a_0b_0 - a_1b_1 + a_2b_2 + a_3b_3, a_0\begin{pmatrix}b_1\\b_2\\b_3\end{pmatrix} + b_0\begin{pmatrix}a_1\\a_2\\a_3\end{pmatrix} + \begin{pmatrix}a_2b_3 - a_3b_2\\a_3b_1 - a_1b_3\\a_1b_2 - a_2b_1\end{pmatrix}\right\} \quad (60)$$

$$= \left\{a_0b_0 - a_1b_1 + a_2b_2 + a_3b_3, \begin{pmatrix}a_0b_1 + a_1b_0 + a_2b_3 - a_3b_2\\a_0b_2 - a_1b_3 + a_2b_0 + a_3b_1\\a_0b_3 + a_1b_2 - a_2b_1 + a_3b_0\end{pmatrix}\right\} \quad (61)$$

Comparison with equation (19) gives the identity:

$$ab = a_0b_0 - \boldsymbol{a}.\boldsymbol{b}, a_0\boldsymbol{b} + b_0\boldsymbol{a} + \boldsymbol{a}\times\boldsymbol{b} \quad (62)$$

2.11 Triple product of quaternion and vector

The two triple products $a^*\boldsymbol{v}a$ and $a\boldsymbol{v}a^*$, where a is a general quaternion and $\boldsymbol{v}$ is a vector, are discussed in more detail in the next section on quaternions as rotation operators. Direct expansion proves that both triple products result in another vector or pure quaternion with zero scalar component.

$$\begin{aligned}a^*\boldsymbol{v}a = &\{(a_0^2 + a_1^2 - a_2^2 - a_3^2)v_1 + 2a_0(a_3v_2 - a_2v_3) + 2a_1(a_2v_2 + a_3v_3)\}\boldsymbol{i}\\ &+\{2a_3(a_2v_3 - a_0v_1) + (a_0^2 - a_1^2 + a_2^2 - a_3^2)v_2 + 2a_1(a_2v_1 + a_0v_3)\}\boldsymbol{j}\\ &+\{2\,a_0(-a_1v_2 + a_2v_1) + 2a_3(a_1v_1 + a_2v_2) + (a_0^2 - a_1^2 - a_2^2 + a_3^2)v_3\}\boldsymbol{k}\end{aligned} \quad (63)$$

Direct expansion of the vector expression $(a_0^2 - |\boldsymbol{a}|^2)\boldsymbol{v} + 2(\boldsymbol{a}.\boldsymbol{v})\boldsymbol{a} - 2a_0(\boldsymbol{a}\times\boldsymbol{v})$ gives:

$$\begin{aligned}&(a_0^2 - |\boldsymbol{a}|^2)\boldsymbol{v} + 2(\boldsymbol{a}.\boldsymbol{v})\boldsymbol{a} - 2a_0(\boldsymbol{a}\times\boldsymbol{v}) = (a_0^2 - a_1^2 - a_2^2 - a_3^2)(v_1\boldsymbol{i} + v_2\boldsymbol{j} + v_3\boldsymbol{k}) +\\ &2(a_1v_1 + a_2v_2 + a_3v_3)(a_1\boldsymbol{i} + a_2\boldsymbol{j} + a_3\boldsymbol{k}) - 2a_0\{(a_2v_3 - a_3v_2)\boldsymbol{i} + (a_3v_1 - a_1v_3)\boldsymbol{j} + (a_1v_2 - a_2v_1)\boldsymbol{k}\}\end{aligned} \quad (64)$$

$$\begin{aligned}= &(a_0^2v_1 - a_1^2v_1 - a_2^2v_1 - a_3^2v_1 + 2v_1a_1^2 + 2a_2a_1v_2 + 2a_3a_1v_3 - 2a_0a_2v_3 + 2a_0a_3v_2)\boldsymbol{i}\\ &+(a_0^2v_2 - a_1^2v_2 - a_2^2v_2 - a_3^2v_2 + 2a_1v_1a_2 + 2v_2a_2^2 + 2a_3a_2v_3 - 2a_0a_3v_1 + 2a_0a_1v_3)\boldsymbol{j}\\ &+(a_0^2v_3 - a_1^2v_3 - a_2^2v_3 - a_3^2v_3 + 2a_1v_1a_3 + 2a_2v_2a_3 + 2v_3a_3^2 - 2a_0a_1v_2 + 2a_0a_2v_1)\boldsymbol{k}\end{aligned} \quad (65)$$

$$\begin{aligned}= &\{(a_0^2 + a_1^2 - a_2^2 - a_3^2)v_1 + 2a_0(a_3v_2 - a_2v_3) + 2a_1(a_2v_2 + a_3v_3)\}\boldsymbol{i}\\ &+\{2a_3(a_2v_3 - a_0v_1) + (a_0^2 - a_1^2 + a_2^2 - a_3^2)v_2 + 2a_1(a_2v_1 + a_0v_3)\}\boldsymbol{j}\\ &+\{2\,a_0(-a_1v_2 + a_2v_1) + 2a_3(a_1v_1 + a_2v_2) + (a_0^2 - a_1^2 - a_2^2 + a_3^2)v_3\}\boldsymbol{k}\end{aligned} \quad (66)$$

Comparison of equations (63) and (66) gives an identity decomposing $a^*\boldsymbol{v}a$ into three vector components aligned i) parallel with $\boldsymbol{v}$, ii) parallel with $\boldsymbol{a}$ and iii) orthogonal to both $\boldsymbol{a}$ and $\boldsymbol{v}$.

$$a^*\boldsymbol{v}a = (a_0^2 - |\boldsymbol{a}|^2)\boldsymbol{v} + 2(\boldsymbol{a}.\boldsymbol{v})\boldsymbol{a} - 2a_0(\boldsymbol{a}\times\boldsymbol{v}) \quad (67)$$

Equation (67) holds for all quaternions a and therefore also hold for the conjugate $a^* = a_0 - \boldsymbol{a}$ giving:

$$(a^*)^*\boldsymbol{v}(a^*) = (a_0^2 - |-\boldsymbol{a}|^2)\boldsymbol{v} + 2(-\boldsymbol{a}.\boldsymbol{v})(-\boldsymbol{a}) - 2a_0(-\boldsymbol{a}\times\boldsymbol{v}) \quad (68)$$

$$\Rightarrow a\boldsymbol{v}a^* = (a_0^2 - |\boldsymbol{a}|^2)\boldsymbol{v} + 2(\boldsymbol{a}.\boldsymbol{v})\boldsymbol{a} + 2a_0(\boldsymbol{a}\times\boldsymbol{v}) \quad (69)$$

3 Matrix and Quaternion Rotation Operators

3.1 General rotation matrix

The rotation matrix $\boldsymbol{R}$ which transforms a vector as a result of a rotation of the coordinate system around the axis $\hat{\boldsymbol{n}}$ by angle η is:

$$\boldsymbol{R} = \begin{pmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{pmatrix} = \begin{pmatrix} \hat{n}_x{}^2 + (1-\hat{n}_x{}^2)cos\eta & \hat{n}_x\hat{n}_y(1-cos\eta) + \hat{n}_z sin\eta & \hat{n}_x\hat{n}_z(1-cos\eta) - \hat{n}_y sin\eta \\ \hat{n}_x\hat{n}_y(1-cos\eta) - \hat{n}_z sin\eta & \hat{n}_y{}^2 + (1-\hat{n}_y{}^2)cos\eta & \hat{n}_y\hat{n}_z(1-cos\eta) + \hat{n}_x sin\eta \\ \hat{n}_x\hat{n}_z(1-cos\eta) + \hat{n}_y sin\eta & \hat{n}_y\hat{n}_z(1-cos\eta) - \hat{n}_x sin\eta & \hat{n}_z{}^2 + (1-\hat{n}_z{}^2)cos\eta \end{pmatrix} \quad \textbf{(70)}$$

Simple algebra proves that the columns of $\boldsymbol{R}$ are orthonormal as required in a rotation matrix:

$$R_{00}{}^2 + R_{10}{}^2 + R_{20}{}^2 = R_{01}{}^2 + R_{11}{}^2 + R_{21}{}^2 = R_{02}{}^2 + R_{12}{}^2 + R_{22}{}^2 = 1 \quad \textbf{(71)}$$

$$R_{00}R_{01} + R_{10}R_{11} + R_{20}R_{21} = R_{01}R_{02} + R_{11}R_{12} + R_{21}R_{22} = R_{02}R_{00} + R_{12}R_{10} + R_{22}R_{20} = 0 \quad \textbf{(72)}$$

The Freescale standard is that multiplication of a vector by a matrix encoding rotation axis $\hat{\boldsymbol{n}}$ and angle η changes a vector as a result of the coordinate system rotating about the specified axis $\hat{\boldsymbol{n}}$ by angle η. The alternative standard is that multiplication of the vector by the rotation matrix rotates the vector about axis $\hat{\boldsymbol{n}}$ and angle η within a fixed coordinate system. Rotations in the two standards have the same axis but opposite rotation angles.

The product $\boldsymbol{R}\hat{\boldsymbol{n}}$ equates to $\hat{\boldsymbol{n}}$ confirming that $\hat{\boldsymbol{n}}$ is the eigenvector of $\boldsymbol{R}$ with eigenvalue 1 and, since $\boldsymbol{R}$ is a rotation matrix, proves that $\hat{\boldsymbol{n}}$ is the rotation axis.

$$\boldsymbol{R}\hat{\boldsymbol{n}} = \begin{pmatrix} \hat{n}_x{}^2 + (1-\hat{n}_x{}^2)cos\eta & \hat{n}_x\hat{n}_y(1-cos\eta) + \hat{n}_z sin\eta & \hat{n}_x\hat{n}_z(1-cos\eta) - \hat{n}_y sin\eta \\ \hat{n}_x\hat{n}_y(1-cos\eta) - \hat{n}_z sin\eta & \hat{n}_y{}^2 + (1-\hat{n}_y{}^2)cos\eta & \hat{n}_y\hat{n}_z(1-cos\eta) + \hat{n}_x sin\eta \\ \hat{n}_x\hat{n}_z(1-cos\eta) + \hat{n}_y sin\eta & \hat{n}_y\hat{n}_z(1-cos\eta) - \hat{n}_x sin\eta & \hat{n}_z{}^2 + (1-\hat{n}_z{}^2)cos\eta \end{pmatrix} \begin{pmatrix} \hat{n}_x \\ \hat{n}_y \\ \hat{n}_z \end{pmatrix} \quad \textbf{(73)}$$

$$= \begin{pmatrix} \hat{n}_x(\hat{n}_x{}^2 + \hat{n}_y{}^2 + \hat{n}_z{}^2) - (\hat{n}_x{}^2 + \hat{n}_y{}^2 + \hat{n}_z{}^2 - 1) \\ \hat{n}_y(\hat{n}_x{}^2 + \hat{n}_y{}^2 + \hat{n}_z{}^2) - (\hat{n}_x{}^2 + \hat{n}_y{}^2 + \hat{n}_z{}^2 - 1) \\ \hat{n}_z(\hat{n}_x{}^2 + \hat{n}_y{}^2 + \hat{n}_z{}^2) - (\hat{n}_x{}^2 + \hat{n}_y{}^2 + \hat{n}_z{}^2 - 1) \end{pmatrix} = \begin{pmatrix} \hat{n}_x \\ \hat{n}_y \\ \hat{n}_z \end{pmatrix} \quad \textbf{(74)}$$

If the rotation matrix $\boldsymbol{R}$ is applied to an arbitrary vector $\boldsymbol{v}$, the result is:

$$\boldsymbol{R}\boldsymbol{v} = \begin{pmatrix} \hat{n}_x{}^2 + (1-\hat{n}_x{}^2)cos\eta & \hat{n}_x\hat{n}_y(1-cos\eta) + \hat{n}_z sin\eta & \hat{n}_x\hat{n}_z(1-cos\eta) - \hat{n}_y sin\eta \\ \hat{n}_x\hat{n}_y(1-cos\eta) - \hat{n}_z sin\eta & \hat{n}_y{}^2 + (1-\hat{n}_y{}^2)cos\eta & \hat{n}_y\hat{n}_z(1-cos\eta) + \hat{n}_x sin\eta \\ \hat{n}_x\hat{n}_z(1-cos\eta) + \hat{n}_y sin\eta & \hat{n}_y\hat{n}_z(1-cos\eta) - \hat{n}_x sin\eta & \hat{n}_z{}^2 + (1-\hat{n}_z{}^2)cos\eta \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \quad \textbf{(75)}$$

$$= \begin{pmatrix} \{\hat{n}_x{}^2 + (1-\hat{n}_x{}^2)cos\eta\}v_x + \{\hat{n}_x\hat{n}_y(1-cos\eta) + \hat{n}_z sin\eta\}v_y + \{\hat{n}_x\hat{n}_z(1-cos\eta) - \hat{n}_y sin\eta\}v_z \\ \{\hat{n}_x\hat{n}_y(1-cos\eta) - \hat{n}_z sin\eta\}v_x + \{\hat{n}_y{}^2 + (1-\hat{n}_y{}^2)cos\eta\}v_y + \{\hat{n}_y\hat{n}_z(1-cos\eta) + \hat{n}_x sin\eta\}v_z \\ \{\hat{n}_x\hat{n}_z(1-cos\eta) + \hat{n}_y sin\eta\}v_x + \{\hat{n}_y\hat{n}_z(1-cos\eta) - \hat{n}_x sin\eta\}v_y + \{\hat{n}_z{}^2 + (1-\hat{n}_z{}^2)cos\eta\}v_z \end{pmatrix} \quad \textbf{(76)}$$

The trace (the sum of the diagonal elements) of the rotation matrix $\boldsymbol{R}$ gives a simple result expression for the rotation angle η encoded in the matrix for any rotation axis $\hat{\boldsymbol{n}}$:

$$tr(\boldsymbol{R}) = R_{00} + R_{00} + R_{00} \tag{77}$$

$$= \hat{n}_x{}^2 + \left(1 - \hat{n}_x{}^2\right)cos\eta + \hat{n}_y{}^2 + \left(1 - \hat{n}_y{}^2\right)cos\eta + \hat{n}_z{}^2 + \left(1 - \hat{n}_z{}^2\right)cos\eta = 1 + 2cos\eta \tag{78}$$

3.2 Quaternion rotation operator

The rotation quaternion q for a rotation about normalized rotation axis $\hat{\boldsymbol{n}}$ by angle η is defined to be:

$$q = cos\left(\frac{\eta}{2}\right) + \hat{\boldsymbol{n}}sin\left(\frac{\eta}{2}\right) \tag{79}$$

q is obviously a unit quaternion since its norm evaluates to 1:

$$N(q) = cos^2\left(\frac{\eta}{2}\right) + sin^2\left(\frac{\eta}{2}\right)\left(\hat{n}_x{}^2 + \hat{n}_y{}^2 + \hat{n}_z{}^2\right) = 1 \tag{80}$$

The justification for identifying the quaternion q as a rotation operator can be found by computing the triple product $q^*\boldsymbol{v}q$ which evaluates, for an arbitrary vector $\boldsymbol{v}$, to be:

$$q^*\boldsymbol{v}q = \left(cos\left(\frac{\eta}{2}\right) - \hat{\boldsymbol{n}}sin\left(\frac{\eta}{2}\right)\right)\boldsymbol{v}\left(cos\left(\frac{\eta}{2}\right) + \hat{\boldsymbol{n}}sin\left(\frac{\eta}{2}\right)\right) \tag{81}$$

$$= \left(cos\left(\frac{\eta}{2}\right) - \hat{n}_x sin\left(\frac{\eta}{2}\right)\boldsymbol{i} - \hat{n}_y sin\left(\frac{\eta}{2}\right)\boldsymbol{j} - \hat{n}_z sin\left(\frac{\eta}{2}\right)\boldsymbol{k}\right)\left(v_x\boldsymbol{i} + v_y\boldsymbol{j} + v_z\boldsymbol{k}\right)\left(cos\left(\frac{\eta}{2}\right) + \hat{n}_x sin\left(\frac{\eta}{2}\right)\boldsymbol{i} + \hat{n}_y sin\left(\frac{\eta}{2}\right)\boldsymbol{j} + \hat{n}_z sin\left(\frac{\eta}{2}\right)\boldsymbol{k}\right) \tag{82}$$

$$= \begin{pmatrix} \{\hat{n}_x{}^2 + \left(1 - \hat{n}_x{}^2\right)cos\eta\}v_x + \{\hat{n}_x\hat{n}_y(1 - cos\eta) + \hat{n}_z sin\eta\}v_y + \{\hat{n}_x\hat{n}_z(1 - cos\eta) - \hat{n}_y sin\eta\}v_z \\ \{\hat{n}_x\hat{n}_y(1 - cos\eta) - \hat{n}_z sin\eta\}v_x + \{\hat{n}_y{}^2 + \left(1 - \hat{n}_y{}^2\right)cos\eta\}v_y + \{\hat{n}_y\hat{n}_z(1 - cos\eta) + \hat{n}_x sin\eta\}v_z \\ \{\hat{n}_x\hat{n}_z(1 - cos\eta) + \hat{n}_y sin\eta\}v_x + \{\hat{n}_y\hat{n}_z(1 - cos\eta) - \hat{n}_x sin\eta\}v_y + \{\hat{n}_z{}^2 + \left(1 - \hat{n}_z{}^2\right)cos\eta\}v_z \end{pmatrix} \tag{83}$$

Equations (77) and (83) match, implying that the quaternion operator $q^*\boldsymbol{v}q$ operating on $\boldsymbol{v}$ is equivalent to the rotation matrix operating on $\boldsymbol{v}$:

$$\boldsymbol{R}\boldsymbol{v} = q^*\boldsymbol{v}q \tag{84}$$

The quaternion q is therefore the rotation quaternion operator for transforming a vector $\boldsymbol{v}$ under rotation of the coordinate system about axis $\hat{\boldsymbol{n}}$ and angle η via the operation $q^*\boldsymbol{v}q$.

From the definition of q^*, it follows that q^* is the rotation operator about the same axis $\hat{\boldsymbol{n}}$ but by angle $-\eta$ and is therefore the inverse of the rotation operator q:

$$q^* = cos\left(\frac{\eta}{2}\right) - \hat{\boldsymbol{n}}sin\left(\frac{\eta}{2}\right) = cos\left(\frac{\eta}{2}\right) + \hat{\boldsymbol{n}}sin\left(\frac{-\eta}{2}\right) \tag{85}$$

The inverse nature of the operators q and q^* can also be shown by computing the effects of the product rotations i) q followed by q^* and ii) q^* followed by q on the vector $\boldsymbol{v}$:

$$q(q^*\boldsymbol{v}q)q^* = (qq^*)\boldsymbol{v}(qq^*) = \boldsymbol{v} \tag{86}$$

$$q^*(q\boldsymbol{v}q^*)q = (q^*q)\boldsymbol{v}(q^*q) = \boldsymbol{v} \tag{87}$$

The result of successively applying rotation quaternions q_1 followed by q_2 through q_N to vector $\boldsymbol{v}$ is:

$$q_N{}^* \dots (q_2{}^*(q_1{}^*\boldsymbol{v}q_1)q_2) \dots q_N = (q_N{}^* \dots q_2{}^*q_1{}^*)\boldsymbol{v}(q_1q_2 \dots \dots q_N) \tag{88}$$

The rotation quaternion q equivalent to the N successive rotations represented by quaternions q_1 to q_N is then:

$$q = q_1q_1q_1 \dots q_N \tag{89}$$

Both rotation matrices and quaternions are unchanged if both the rotation angle and axis are simultaneously inverted:

$$\boldsymbol{R}(-\hat{\boldsymbol{n}},-\eta)=\begin{pmatrix} \hat{n}_x{}^2 + (1-\hat{n}_x{}^2)cos(-\eta) & \hat{n}_x\hat{n}_y(1-cos(-\eta)) - \hat{n}_z sin(-\eta) & \hat{n}_x\hat{n}_z(1-cos(-\eta)) + \hat{n}_y sin(-\eta) \\ \hat{n}_x\hat{n}_y(1-cos(-\eta)) + \hat{n}_z sin(-\eta) & \hat{n}_y{}^2 + (1-\hat{n}_y{}^2)cos(-\eta) & \hat{n}_y\hat{n}_z(1-cos(-\eta)) - \hat{n}_x sin(-\eta) \\ \hat{n}_x\hat{n}_z(1-cos(-\eta)) - \hat{n}_y sin(-\eta) & \hat{n}_y\hat{n}_z(1-cos(-\eta)) + \hat{n}_x sin(-\eta) & \hat{n}_z{}^2 + (1-\hat{n}_z{}^2)cos(-\eta) \end{pmatrix} = \boldsymbol{R}(\hat{\boldsymbol{n}},\eta) \tag{90}$$

$$q(-\hat{\boldsymbol{n}},-\eta) = cos\left(\frac{-\eta}{2}\right) - \hat{\boldsymbol{n}}sin\left(\frac{-\eta}{2}\right) = cos\left(\frac{\eta}{2}\right) + \hat{\boldsymbol{n}}sin\left(\frac{\eta}{2}\right) = (\hat{\boldsymbol{n}},\eta) \tag{91}$$

Finally, some texts define the quaternion rotation operator on vector $\boldsymbol{v}$ to be $q\boldsymbol{v}q^*$ instead of $q^*\boldsymbol{v}q$. The explanation is that the operator $q^*\boldsymbol{v}q$ transforms the vector $\boldsymbol{v}$ as a result of rotation of the coordinate system by angle η whereas the operator $q\boldsymbol{v}q^*$ rotates the vector $\boldsymbol{v}$ by angle η in a fixed coordinate system. The standard used in this document and all Freescale sensor fusion software is that it is the coordinate system that is rotating (normally as a result of the smartphone or tablet orientation changing) and the vector $\boldsymbol{v}$ (typically the earth's gravitational or geomagnetic field) is fixed in the un-rotated earth reference frame.

3.3 Deriving rotation quaternion from two vectors

This section derives an expression for the rotation quaternion required to rotate a vector $\boldsymbol{u}$ onto vector $\boldsymbol{v}$ such that:

$$\boldsymbol{v} = q^*\boldsymbol{u}q \tag{92}$$

Obviously the norm of both vectors must be the same since rotation does not change a vector's magnitude. The sign of the rotation angle is that the coordinate frame rotates by angle η about the axis parallel to the vector $\boldsymbol{u} \times \boldsymbol{v}$.

The angle η between the two vectors satisfies:

$$cos\eta = 2cos^2\left(\frac{\eta}{2}\right) - 1 = \frac{\boldsymbol{u}.\boldsymbol{v}}{|\boldsymbol{u}||\boldsymbol{v}|} \Rightarrow cos\left(\frac{\eta}{2}\right) = \sqrt{\frac{1}{2} + \frac{\boldsymbol{u}.\boldsymbol{v}}{2|\boldsymbol{u}||\boldsymbol{v}|}} = \sqrt{\frac{|\boldsymbol{u}||\boldsymbol{v}| + \boldsymbol{u}.\boldsymbol{v}}{2|\boldsymbol{u}||\boldsymbol{v}|}} \quad (93)$$

$$\hat{\boldsymbol{n}}sin\eta = 2\hat{\boldsymbol{n}}sin\left(\frac{\eta}{2}\right)cos\left(\frac{\eta}{2}\right) = \frac{-\boldsymbol{u}\times\boldsymbol{v}}{|\boldsymbol{u}||\boldsymbol{v}|} \Rightarrow \hat{\boldsymbol{n}}sin\left(\frac{\eta}{2}\right) = \frac{-\boldsymbol{u}\times\boldsymbol{v}}{2|\boldsymbol{u}||\boldsymbol{v}|\sqrt{\frac{|\boldsymbol{u}||\boldsymbol{v}| + \boldsymbol{u}.\boldsymbol{v}}{2|\boldsymbol{u}||\boldsymbol{v}|}}} \quad (94)$$

The minus sign in equation (93) derives from calculating the coordinate system rotation η to map vector $\boldsymbol{u}$ onto $\boldsymbol{v}$ rather than the angle between $\boldsymbol{u}$ and $\boldsymbol{v}$. The required rotation quaternion q is then:

$$q = cos\left(\frac{\eta}{2}\right) + \hat{\boldsymbol{n}}sin\left(\frac{\eta}{2}\right) = \sqrt{\frac{|\boldsymbol{u}||\boldsymbol{v}| + \boldsymbol{u}.\boldsymbol{v}}{2|\boldsymbol{u}||\boldsymbol{v}|}} - \frac{\boldsymbol{u}\times\boldsymbol{v}}{2|\boldsymbol{u}||\boldsymbol{v}|\sqrt{\frac{|\boldsymbol{u}||\boldsymbol{v}| + \boldsymbol{u}.\boldsymbol{v}}{2|\boldsymbol{u}||\boldsymbol{v}|}}} = \frac{|\boldsymbol{u}||\boldsymbol{v}| + \boldsymbol{u}.\boldsymbol{v} - \boldsymbol{u}\times\boldsymbol{v}}{\sqrt{2|\boldsymbol{u}||\boldsymbol{v}|(|\boldsymbol{u}||\boldsymbol{v}| + \boldsymbol{u}.\boldsymbol{v})}} \quad (95)$$

4 Aerospace, Android and Windows 8 Rotation Quaternions

4.1 Aerospace rotation matrix and quaternion

The Aerospace coordinate system (also termed the NED or x=North, y=East, z=Down system) is depicted in Figure 1. Rotations in the Aerospace coordinate system are positive if they are clockwise when viewed along an axis in its positive direction. The yaw rotation by angle ψ about the z axis is first, followed by the pitch rotation by angle θ about the y axis and then finally followed by a roll rotation by angle ϕ about the x axis. The yaw (or compass) angle ψ has range 0° to 360°, the pitch angle θ a range of -90° to 90° and the roll angle ϕ a range of -180° to 180°.

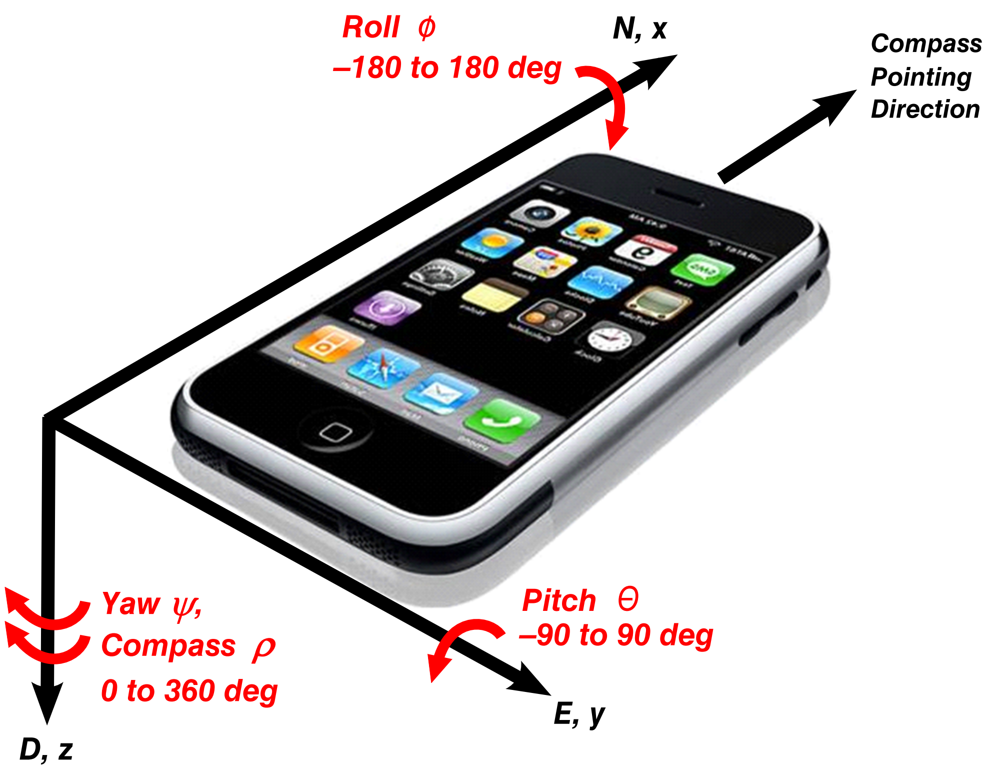

Figure 1. NED/Aerospace Coordinate System

The individual NED / Aerospace rotation matrices for the roll, pitch and yaw rotations are listed below.

$$\boldsymbol{R}_x(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{pmatrix} \tag{96}$$

$$\boldsymbol{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{pmatrix} \tag{97}$$

$$\boldsymbol{R}_z(\psi) = \begin{pmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{98}$$

The composite NED/Aerospace rotation matrix is then:

$$\boldsymbol{R}_{xyz} = \boldsymbol{R}_x(\phi)\boldsymbol{R}_y(\theta)\boldsymbol{R}_z(\psi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{pmatrix}\begin{pmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{pmatrix}\begin{pmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{99}$$

$$= \begin{pmatrix} \cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta \\ \cos\psi\sin\theta\sin\phi - \cos\phi\sin\psi & \cos\phi\cos\psi + \sin\theta\sin\phi\sin\psi & \cos\theta\sin\phi \\ \cos\phi\cos\psi\sin\theta + \sin\phi\sin\psi & \cos\phi\sin\theta\sin\psi - \cos\psi\sin\phi & \cos\theta\cos\phi \end{pmatrix} \tag{100}$$

The rotation matrix $\boldsymbol{R}_{xyz} = \boldsymbol{R}_x(\phi)\boldsymbol{R}_y(\theta)\boldsymbol{R}_z(\psi)$ is applied to the left of the vector $\boldsymbol{v}$ so the notation $\boldsymbol{R}_{xyz}$ indicates that the rotation order is about the z then y then x axes.

The individual NED/Aerospace rotation quaternions are given by the definition in equation (79) :

$$q_x(\phi) = cos\left(\frac{\phi}{2}\right) + sin\left(\frac{\phi}{2}\right)\boldsymbol{i} \tag{101}$$

$$q_y(\theta) = cos\left(\frac{\theta}{2}\right) + sin\left(\frac{\theta}{2}\right)\boldsymbol{j} \tag{102}$$

$$q_z(\psi) = cos\left(\frac{\psi}{2}\right) + sin\left(\frac{\psi}{2}\right)\boldsymbol{k} \tag{103}$$

The composite NED / Aerospace rotation quaternion $q_{zyx} = q_z(\psi)q_y(\theta)q_x(\phi)$ for the rotation sequence about the z then y and finally x axes evaluates to:

$$q_{zyx} = q_z(\psi)q_y(\theta)q_x(\phi) = \left\{cos\left(\frac{\psi}{2}\right) + sin\left(\frac{\psi}{2}\right)\boldsymbol{k}\right\}\left\{cos\left(\frac{\theta}{2}\right) + sin\left(\frac{\theta}{2}\right)\boldsymbol{j}\right\}\left\{cos\left(\frac{\phi}{2}\right) + sin\left(\frac{\phi}{2}\right)\boldsymbol{i}\right\} \tag{104}$$

$$\begin{aligned} &= \left\{cos\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) + sin\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right)\right\} + \left\{cos\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right) - sin\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right)\right\}\boldsymbol{i} \\ &+ \left\{cos\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) + sin\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right)\right\}\boldsymbol{j} + \left\{sin\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) - cos\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right)\right\}\boldsymbol{k} \end{aligned} \tag{105}$$

The rotation quaternion $q_{zyx} = q_z q_y q_x$ is applied directly to the right of the vector $\boldsymbol{v}$ and in conjugate form to the left of vector $\boldsymbol{v}$ so here the notation q_{zyx} indicates that the rotation sequence is about the z then y then x axes as for the rotation matrix $\boldsymbol{R}_{xyz}$.

The four components of the NED / Aerospace rotation quaternion q_{zyx} are therefore:

$$q_0 = cos\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) + sin\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right) \tag{106}$$

$$q_1 = cos\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right) - sin\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) \tag{107}$$

$$q_2 = cos\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) + sin\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right) \tag{108}$$

$$q_3 = sin\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) - cos\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right) \tag{109}$$

Direct evaluation of the rotation matrix and quaternion products with an arbitrary vector $\boldsymbol{v}$ confirms the equivalence of transforming the vector by NED/Aerospace rotation matrix or by NED/Aerospace quaternion multiplication:

$$\boldsymbol{R}_{xyz}\boldsymbol{v} = \boldsymbol{R}_x(\phi)\boldsymbol{R}_y(\theta)\boldsymbol{R}_z(\psi)\boldsymbol{v} = q_{zyx}{}^*\boldsymbol{v}q_{zyx} = \left(q_z q_y q_x\right)^*\boldsymbol{v}q_z q_y q_x = q_x{}^* q_y{}^* q_z{}^*\boldsymbol{v}q_z q_y q_x \tag{110}$$

$$= \begin{pmatrix} v_x \cos\theta\cos\psi + v_y\cos\theta\, sin\psi - v_z sin\theta \\ v_x(\cos\psi\, sin\theta sin\phi - \cos\phi\, sin\psi) + v_y(\cos\phi\cos\psi + sin\theta sin\phi sin\psi) + v_z\cos\theta\, sin\phi \\ v_x(cos\phi\cos\psi\, sin\theta + sin\phi sin\psi) + v_y(\cos\phi\, sin\theta sin\psi - \cos\psi\, sin\phi) + v_z\cos\theta\, cos\phi \end{pmatrix} \tag{111}$$

4.2 Android rotation matrix and quaternion

The Android sensor coordinate system is shown in Figure 2. The Android coordinate system, as is also the Windows 8 coordinate system discussed in Windows 8 rotation matrix and quaternion, is an example of an ENU or x=East, y=North, z=Up system. Rotations in the Android coordinate system are defined in the opposite manner to NED and also opposite to common mathematical usage by having positive sign when anticlockwise as viewed along an axis in its positive direction.

The yaw rotation by angle ψ about the z axis is first, followed by the roll rotation by angle ϕ about the y axis and then finally followed by a pitch rotation by angle θ about the x axis. The yaw (or compass) angle ψ has range 0° to 360°, the roll angle ϕ a range of –90° to 90° and the pitch angle θ a range of -180° to 180°.

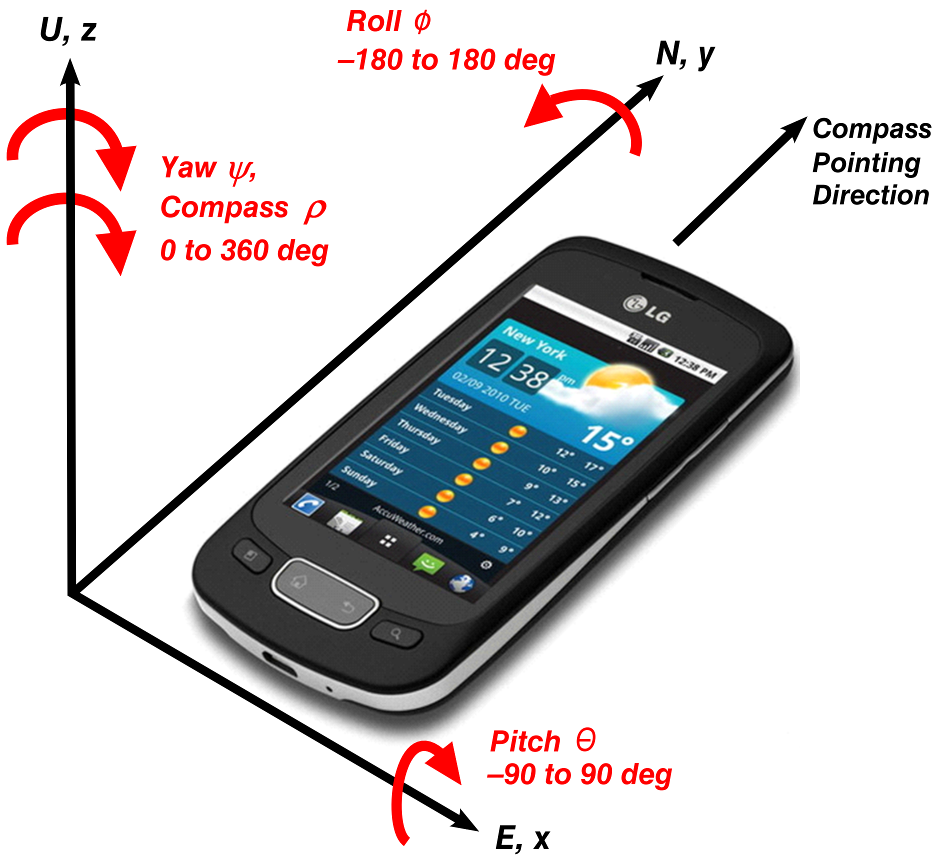

Figure 2. Android Coordinate System

The individual Android pitch, roll and yaw rotation matrices are listed below. The convention that θ is the pitch angle, ϕ is the roll angle and ψ is the yaw angle remains unchanged from NED except that in Android the pitch rotation is about the x axis and the roll rotation is about the y axis.

$$\boldsymbol{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \quad \textbf{(112)}$$

$$\boldsymbol{R}_y(\phi) = \begin{pmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{pmatrix} \quad \textbf{(113)}$$

$$\boldsymbol{R}_z(\psi) = \begin{pmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \textbf{(114)}$$

The composite Android rotation matrix is then:

$$\boldsymbol{R}_{xyz} = \boldsymbol{R}_{\boldsymbol{x}}(\theta)\boldsymbol{R}_{\boldsymbol{y}}(\phi)\boldsymbol{R}_{\boldsymbol{z}}(\psi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{pmatrix}\begin{pmatrix} \cos\phi & 0 & sin\phi \\ 0 & 1 & 0 \\ -sin\phi & 0 & cos\phi \end{pmatrix}\begin{pmatrix} \cos\psi & -sin\psi & 0 \\ sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{115}$$

$$= \begin{pmatrix} \cos\phi\cos\psi & -cos\phi sin\psi & sin\phi \\ \cos\theta\, sin\psi + \cos\psi\, sin\phi sin\theta & \cos\psi\cos\theta - sin\phi sin\psi sin\theta & -\cos\phi\, sin\theta \\ -\cos\psi\cos\theta\, sin\phi + sin\psi sin\theta & \cos\theta\, sin\phi sin\psi + \cos\psi\, sin\theta & \cos\phi\cos\theta \end{pmatrix} \tag{116}$$

The individual Android rotation quaternions are given by the definition in equation 3.2.1 remembering that rotations in Android are specified in the opposite sense to normal mathematical usage:

$$q_x(\theta) = cos\left(\frac{-\theta}{2}\right) + \boldsymbol{i}sin\left(\frac{-\theta}{2}\right) = cos\left(\frac{\theta}{2}\right) - \boldsymbol{i}sin\left(\frac{\theta}{2}\right) \tag{117}$$

$$q_y(\phi) = cos\left(\frac{-\phi}{2}\right) + \boldsymbol{j}sin\left(\frac{-\phi}{2}\right) = cos\left(\frac{\phi}{2}\right) - \boldsymbol{j}sin\left(\frac{\phi}{2}\right) \tag{118}$$

$$q_z(\psi) = cos\left(\frac{-\psi}{2}\right) + \boldsymbol{k}sin\left(\frac{-\psi}{2}\right) = cos\left(\frac{\psi}{2}\right) - \boldsymbol{k}sin\left(\frac{\psi}{2}\right) \tag{119}$$

The composite Android rotation quaternion $q_{zyx} = q_z(\psi)q_y(\phi)q_x(\theta)$ for the rotation sequence about the z then y and finally x axes evaluates to:

$$q_{zyx} = q_z(\psi)q_y(\phi)q_x(\theta) = \left\{cos\left(\frac{\psi}{2}\right) - \boldsymbol{k}sin\left(\frac{\psi}{2}\right)\right\}\left\{cos\left(\frac{\phi}{2}\right) - \boldsymbol{j}sin\left(\frac{\phi}{2}\right)\right\}\left\{cos\left(\frac{\theta}{2}\right) - \boldsymbol{i}sin\left(\frac{\theta}{2}\right)\right\} \tag{120}$$

$$\begin{aligned} &= \left\{cos\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) - sin\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right)\right\} - \left\{cos\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) + sin\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right)\right\}\boldsymbol{i} \\ &- \left\{cos\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right) - sin\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right)\right\}\boldsymbol{j} - \left\{sin\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) + cos\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right)\right\}\boldsymbol{k} \end{aligned} \tag{121}$$

The four components of the Android rotation quaternion q_{zyx} are therefore:

$$q_0 = cos\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) - sin\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right) \tag{122}$$

$$q_1 = -cos\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) - sin\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right) \tag{123}$$

$$q_2 = -cos\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right) + sin\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) \tag{124}$$

$$q_3 = -sin\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) - cos\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right) \tag{125}$$

Direct evaluation of the rotation matrix and quaternion products with an arbitrary vector $\boldsymbol{v}$ confirms the equivalence of transforming the vector by Android rotation matrix or by Android quaternion multiplication:

$$\boldsymbol{R}_{xyz}\boldsymbol{v} = \boldsymbol{R}_x(\theta)\boldsymbol{R}_y(\phi)\boldsymbol{R}_z(\psi)\boldsymbol{v} = q_{zyx}{}^*\boldsymbol{v}q_{zyx} = \left(q_zq_yq_x\right)^*\boldsymbol{v}q_zq_yq_x = q_x{}^*q_y{}^*q_z{}^*\boldsymbol{v}q_zq_yq_x \tag{126}$$

$$= \begin{pmatrix} v_x\cos\phi\cos\psi - v_y cos\phi sin\psi + v_z sin\phi \\ v_x(\cos\theta\, sin\psi + \cos\psi\, sin\phi sin\theta) + v_y(\cos\psi\cos\theta - sin\phi sin\psi sin\theta) - v_z\cos\phi\, sin\theta \\ v_x(-\cos\psi\cos\theta\, sin\phi + sin\psi sin\theta) + v_y(\cos\theta\, sin\phi sin\psi + \cos\psi\, sin\theta) + v_z\cos\phi\cos\theta \end{pmatrix} \tag{127}$$

4.3 Windows 8 rotation matrix and quaternion

The Windows 8 coordinate system is shown in Figure 3. The x, y, and z axes are identical to the Android coordinate system (and Windows 8 is therefore also an ENU or x=East, y=North, z=Up system). The sense of the Windows 8 rotation angles is, however, opposite to Android and therefore compatible with normal mathematical usage by being positive when clockwise as viewed along a axis in its positive direction. A consequence is that the Windows 8 compass heading angle ρ is defined in the opposite sense to the Windows 8 yaw angle ψ. The order of the rotations in Windows 8 matches NED but not Android. The yaw rotation about the z axis is first, followed by the pitch rotation by angle θ about the x axis and then finally followed by a roll rotation about the y axis. The ranges of the Euler angles are the same as Android. The yaw angle ψ has range 0° to 360°, the pitch angle θ has range –180° to 180° and the roll angle ϕ a range of –90° to 90°. The compass heading angle ρ also has range 0° to 360° but is the negative (modulo 360°) of the yaw angle ψ so that a yaw angle ψ=90° implies that the compass heading angle ρ=270°.

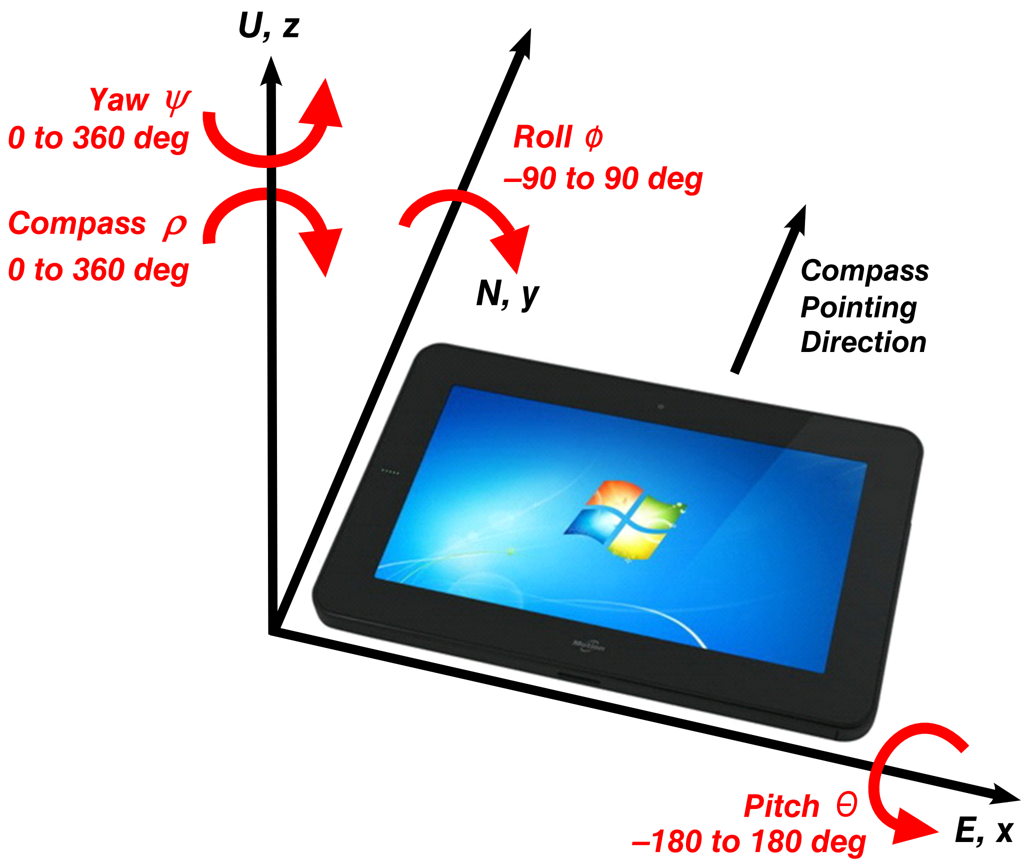

Figure 3. Windows 8 Coordinate System

The individual Windows 8 pitch, roll and yaw rotation matrices are listed below. The convention is retained with the Aerospace and Android coordinate systems that θ is the pitch angle, ϕ is the roll angle and ψ is the yaw angle. Like Android, the pitch rotation is about the x axis and the roll rotation about the y axis. Unlike Android, the pitch rotation in Windows 8 occurs before the roll rotation.

$$\boldsymbol{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{pmatrix} \quad \textbf{(128)}$$

Table continues on the next page...

$$\boldsymbol{R_y}(\phi) = \begin{pmatrix} cos\,\phi & 0 & -sin\phi \\ 0 & 1 & 0 \\ sin\phi & 0 & cos\phi \end{pmatrix} \quad (129)$$

$$\boldsymbol{R_z}(\psi) = \begin{pmatrix} cos\,\psi & sin\psi & 0 \\ -sin\psi & cos\,\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (130)$$

The composite Windows 8 rotation matrix is then:

$$\boldsymbol{R_{yxz}} = \boldsymbol{R_y}(\phi)\boldsymbol{R_x}(\theta)\boldsymbol{R_z}(\psi) = \begin{pmatrix} cos\,\phi & 0 & -sin\phi \\ 0 & 1 & 0 \\ sin\phi & 0 & cos\phi \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & cos\,\theta & sin\theta \\ 0 & -sin\theta & cos\theta \end{pmatrix}\begin{pmatrix} cos\,\psi & sin\psi & 0 \\ -sin\psi & cos\,\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (131)$$

$$= \begin{pmatrix} cos\,\phi\,cos\,\psi - sin\phi sin\theta sin\psi & cos\phi sin\psi + cos\,\psi\,sin\phi sin\theta & -sin\phi\,cos\,\theta \\ -\,cos\,\theta\,sin\psi & cos\,\theta\,cos\,\psi & sin\theta \\ cos\,\psi\,sin\phi + cos\,\phi\,sin\psi sin\theta & sin\phi sin\psi - cos\,\phi\,cos\,\psi\,sin\theta & cos\,\phi\,cos\,\theta \end{pmatrix} \quad (132)$$

The individual Windows 8 rotation quaternions are:

$$q_x(\theta) = cos\left(\frac{\theta}{2}\right) + \boldsymbol{i}sin\left(\frac{\theta}{2}\right) \quad (133)$$

$$q_y(\phi) = cos\left(\frac{\phi}{2}\right) + \boldsymbol{j}sin\left(\frac{\phi}{2}\right) \quad (134)$$

$$q_z(\psi) = cos\left(\frac{\psi}{2}\right) + \boldsymbol{k}sin\left(\frac{\psi}{2}\right) \quad (135)$$

The composite Windows 8 rotation quaternion $q_{zyx} = q_z(\psi)q_y(\theta)q_x(\phi)$ for the rotation sequence about the z then x and finally y axes evaluates to:

$$q_{zxy} = q_z(\psi)q_x(\theta)q_y(\phi) = \left\{cos\left(\frac{\psi}{2}\right) + \boldsymbol{k}sin\left(\frac{\psi}{2}\right)\right\}\left\{cos\left(\frac{\theta}{2}\right) + \boldsymbol{i}sin\left(\frac{\theta}{2}\right)\right\}\left\{cos\left(\frac{\phi}{2}\right) + \boldsymbol{j}sin\left(\frac{\phi}{2}\right)\right\} \quad (136)$$

$$= \left\{cos\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) - sin\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right)\right\} + \left\{cos\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) - sin\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right)\right\}\boldsymbol{i}$$
$$+ \left\{cos\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right) + sin\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right)\right\}\boldsymbol{j} + \left\{sin\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) + cos\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right)\right\}\boldsymbol{k} \quad (137)$$

The four components of the Windows 8 aerospace quaternion q_{zxy} are therefore:

$$q_0 = cos\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) - sin\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right) \quad (138)$$

$$q_1 = cos\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) - sin\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right) \quad (139)$$

$$q_2 = cos\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right) + sin\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) \quad (140)$$

$$q_3 = sin\left(\frac{\psi}{2}\right)cos\left(\frac{\theta}{2}\right)cos\left(\frac{\phi}{2}\right) + cos\left(\frac{\psi}{2}\right)sin\left(\frac{\theta}{2}\right)sin\left(\frac{\phi}{2}\right) \quad (141)$$

Direct evaluation of the rotation matrix and quaternion products with an arbitrary vector $\boldsymbol{v}$ confirms the equivalence of transforming the vector by Windows 8 rotation matrix or by Windows 8 quaternion multiplication:

$$\boldsymbol{R}_{yxz}\boldsymbol{v} = \boldsymbol{R}_{\boldsymbol{y}}(\phi)\boldsymbol{R}_{\boldsymbol{x}}(\theta)\boldsymbol{R}_{\boldsymbol{z}}(\psi)\boldsymbol{v} = q_{zxy}{}^{*}\boldsymbol{v}q_{zxy} = \left(q_z q_x q_y\right)^{*}\boldsymbol{v}q_z q_x q_y = q_y{}^{*}q_x{}^{*}q_z{}^{*}\boldsymbol{v}q_z q_x q_y \tag{142}$$

$$= \begin{pmatrix} v_x(\cos\phi\cos\psi - \sin\phi\sin\theta\sin\psi) + v_y(\cos\phi\sin\psi + \cos\psi\sin\phi\sin\theta) - v_z\cos\theta\sin\phi \\ -v_x\cos\theta\sin\psi + v_y\cos\psi\cos\theta + v_z\sin\theta \\ v_x(\cos\psi\sin\phi + \cos\phi\sin\psi\sin\theta) + v_y(\sin\phi\sin\psi - \cos\phi\cos\psi\sin\theta) + v_z\cos\phi\cos\theta \end{pmatrix} \tag{143}$$

5 C Functions Returning Euler Angles

5.1 Introduction

The functions in this section compute the roll, pitch, and yaw Euler angles and the compass heading angle from the corresponding rotation matrix or quaternion for the NED/Aerospace, Android, and Windows 8 coordinate systems.

The Euler angle representation of orientation becomes unstable at "gimbal lock" where the second of the three Euler angle rotations happens to align the axis of the third Euler angle rotation with the axis of the first Euler angle rotation. For the NED/Aerospace and Windows 8 coordinate systems, gimbal occurs at pitch angles of ±90° which align the final roll rotation with the initial yaw rotation. For Android, gimbal lock occurs at roll angles of ±90° which align the final pitch rotation with the initial yaw rotation. Gimbal lock orientations can therefore be defined in terms of just two, rather than the normal three, Euler angles. There is no unique solution for all three Euler angles.

Gimbal lock is a fundamental mathematical limitation of Euler angle representations which manifests as unstable roll and yaw angles at ±90° pitch angles (for NED/Aerospace and Windows 8) and as unstable pitch and yaw angles at ±90° roll angles (for Android). Rotation matrix and rotation quaternion representations of orientations do not suffer from gimbal lock instabilities. It is therefore common practice to only use Euler angles to represent initial and final orientations with all intermediate calculations performed using rotation matrix or quaternion.

5.2 NED/Aerospace Euler angles from rotation matrix

The NED/Aerospace rotation matrix derived in equation (100) has the form:

$$\begin{pmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{pmatrix} = \begin{pmatrix} \cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta \\ \cos\psi\sin\theta\sin\phi - \cos\phi\sin\psi & \cos\phi\cos\psi + \sin\theta\sin\phi\sin\psi & \cos\theta\sin\phi \\ \cos\phi\cos\psi\sin\theta + \sin\phi\sin\psi & \cos\phi\sin\theta\sin\psi - \cos\psi\sin\phi & \cos\theta\cos\phi \end{pmatrix} \tag{144}$$

The solution for the three NED / Aerospace Euler angles is:

$$\phi = \tan^{-1}\left(\frac{R_{12}}{R_{22}}\right), -180 \le \phi < 180\ deg \tag{145}$$

$$\theta = \sin^{-1}(-R_{02}), -90 \le \theta \le 90\ deg \tag{146}$$

$$\psi = \tan^{-1}\left(\frac{R_{01}}{R_{00}}\right), 0 \le \psi < 360\ deg \tag{147}$$

The NED/Aerospace compass heading angle ρ always equals the yaw angle ψ:

$$\rho = \psi \tag{148}$$

The function `fNEDAnglesDegFromRotationMatrix` computes the roll ϕ, pitch θ, yaw ψ and compass heading ρ in units of degrees from the NED rotation matrix $\boldsymbol{R}$. The matrix $\boldsymbol{R}$ is assumed to be a valid rotation matrix, but bounds checking is performed on R_{02} before taking the inverse sine to prevent run-time errors.

```
// extract the NED angles in degrees from the NED rotation matrix
void fNEDAnglesDegFromRotationMatrix(float **R, float *pfPhiDeg, float *pfTheDeg,
    float *pfPsiDeg, float *pfRhoDeg)
{
    // calculate the pitch angle -90.0 <= Theta <= 90.0 deg
    if (R[0][2] >= 1.0F)
        *pfTheDeg = -90.0F;
    else if (R[0][2] <= -1.0F)
        *pfTheDeg = 90.0F;
    else
        *pfTheDeg = (float) asin(-R[0][2]) * FRADTODEG;

    // calculate the roll angle range -180.0 <= Phi < 180.0 deg
    *pfPhiDeg = (float)atan2(R[1][2], R[2][2]) * FRADTODEG;
    // map +180 roll onto the functionally equivalent -180 deg roll
    if (*pfPhiDeg == 180.0F)
        *pfPhiDeg = -180.0F;

    // calculate the yaw and compass angle 0.0 <= Psi < 360.0 deg
    *pfPsiDeg = (float)atan2(R[0][1], R[0][0]) * FRADTODEG;
    if (*pfPsiDeg < 0.0F)
        *pfPsiDeg += 360.0F;
    // check for rounding errors mapping small negative angle to 360 deg
    if (*pfPsiDeg >= 360.0F)
        *pfPsiDeg = 0.0F;
    // for NED, the compass heading Rho equals the yaw angle Psi
    *pfRhoDeg = *pfPsiDeg;

    return;
}
```

5.3 Android Euler angles from rotation matrix

The Android rotation matrix derived in equation (116) has the form:

$$\begin{pmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{pmatrix} = \begin{pmatrix} \cos\phi\cos\psi & -\cos\phi\sin\psi & \sin\phi \\ \cos\theta\sin\psi + \cos\psi\sin\phi\sin\theta & \cos\psi\cos\theta - \sin\phi\sin\psi\sin\theta & -\cos\phi\sin\theta \\ -\cos\psi\cos\theta\sin\phi + \sin\psi\sin\theta & \cos\theta\sin\phi\sin\psi + \cos\psi\sin\theta & \cos\phi\cos\theta \end{pmatrix} \tag{149}$$

The solution for the three Android Euler angles is:

$$\phi = \sin^{-1}(R_{02}), -90 \le \phi < 90\ deg \tag{150}$$

$$\theta = \tan^{-1}\left(\frac{-R_{12}}{R_{22}}\right), -180 \le \theta < 180\ deg \tag{151}$$

$$\psi = \tan^{-1}\left(\frac{-R_{01}}{R_{00}}\right), 0 \le \psi < 360\ deg \tag{152}$$

The Android compass heading angle ρ always equals the yaw angle ψ:

$$\rho = \psi \tag{153}$$

The function `fAndroidAnglesDegFromRotationMatrix` computes the roll ϕ, pitch θ, yaw ψ and compass heading ρ in units of degrees from the NED rotation matrix $\boldsymbol{R}$. The matrix $\boldsymbol{R}$ is assumed to be a valid rotation matrix but bounds checking is performed on R_{02} before taking the inverse sine to prevent run-time errors.

```
// extract the Android angles in degrees from the Android rotation matrix
void fAndroidAnglesDegFromRotationMatrix(float **R, float *pfPhiDeg, float *pfTheDeg,
    float *pfPsiDeg, float *pfRhoDeg)
{
    // calculate the roll angle -90.0 <= Phi <= 90.0 deg
    if (R[0][2] >= 1.0F)
        *pfPhiDeg = 90.0F;
    else if (R[0][2] <= -1.0F)
        *pfPhiDeg = -90.0F;
    else
        *pfPhiDeg = (float) asin(R[0][2]) * FRADTODEG;

    // calculate the pitch angle -180.0 <= The < 180.0 deg
    *pfTheDeg = (float)atan2(-R[1][2], R[2][2]) * FRADTODEG;
    // map +180 pitch onto the functionally equivalent -180 deg pitch
    if (*pfTheDeg == 180.0F)
        *pfTheDeg = -180.0F;

    // calculate the yaw=compass angle 0.0 <= Psi < 360.0 deg
    *pfPsiDeg = (float)atan2(-R[0][1], R[0][0]) * FRADTODEG;
    if (*pfPsiDeg < 0.0F)
        *pfPsiDeg += 360.0F;
    // check for rounding errors mapping small negative angle to 360 deg
    if (*pfPsiDeg >= 360.0F)
        *pfPsiDeg = 0.0F;
    // the compass heading angle Rho equals the yaw angle Psi
    // this definition is compliant with Motorola Xoom tablet behavior
    *pfRhoDeg = *pfPsiDeg;

    return;
}
```

5.4 Windows 8 Euler angles from rotation matrix

The Windows 8 rotation matrix derived in equation (132) has the form:

$$\begin{pmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{pmatrix} = \begin{pmatrix} \cos\phi\cos\psi - \sin\phi\sin\theta\sin\psi & \cos\phi\sin\psi + \cos\psi\sin\phi\sin\theta & -\sin\phi\cos\theta \\ -\cos\theta\sin\psi & \cos\theta\cos\psi & \sin\theta \\ \cos\psi\sin\phi + \cos\phi\sin\psi\sin\theta & \sin\phi\sin\psi - \cos\phi\cos\psi\sin\theta & \cos\phi\cos\theta \end{pmatrix} \tag{154}$$

The solution for the three Windows 8 Euler angles is:

$$\phi = tan^{-1}\left(\frac{-R_{02}}{R_{22}}\right), -90 \leq \phi < 90\ deg \tag{155}$$

The pitch angle θ has a range of $-180 \leq \theta < 180\ deg$, but is first computed in the range –90° to 90°.

$$\theta = sin^{-1}\,(R_{12}), -90 \le \theta < 90\ deg \qquad \textbf{(156)}$$

Since ϕ is in the range –90° to 90°, it follows that $cos\phi$ is non-negative and that $R_{22} = cos\,\phi\,cos\,\theta$ has the same sign as $cos\,\theta$ and can be used to correct θ into the range –180° ≤ θ

$$\theta \leftarrow \pi - \theta\ if\ R_{22} < 0\ and\ \theta > 0 \qquad \textbf{(157)}$$

$$\theta \leftarrow -\pi - \theta\ if\ R_{22} < 0\ and\ \theta \le 0 \qquad \textbf{(158)}$$

The general solution for the yaw angle ψ for non-zero $cos\,\theta$ is:

$$\psi = tan^{-1}\left(\frac{-cos\theta R_{10}}{cos\theta R_{11}}\right), 0 \le \psi < 360\ deg\,, \theta \neq -90deg,, \theta \neq 90deg \qquad \textbf{(159)}$$

At $\theta = 90deg$, equation **(154)** simplifies to:

$$\begin{pmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{pmatrix} = \begin{pmatrix} cos\,\phi\,cos\,\psi - sin\phi sin\psi & cos\phi sin\psi + cos\,\psi\,sin\phi & 0 \\ 0 & 0 & 1 \\ cos\,\psi\,sin\phi + cos\,\phi\,sin\psi & sin\phi sin\psi - cos\,\phi\,cos\,\psi & 0 \end{pmatrix} for\ \theta = 90deg \qquad \textbf{(160)}$$

$$= \begin{pmatrix} cos(\psi + \phi) & sin(\psi + \phi) & 0 \\ 0 & 0 & 1 \\ sin(\psi + \phi) & -cos(\psi + \phi) & 0 \end{pmatrix} for\ \theta = 90deg \qquad \textbf{(161)}$$

$$\Rightarrow tan(\psi + \phi) = \left(\frac{R_{01}}{R_{00}}\right) for\ \theta = 90deg \qquad \textbf{(162)}$$

At $\theta = -90deg$, equation 5.4.1 simplifies to:

$$\begin{pmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{pmatrix} = \begin{pmatrix} cos\,\phi\,cos\,\psi + sin\phi sin\psi & cos\phi sin\psi - cos\,\psi\,sin\phi & 0 \\ 0 & 0 & -1 \\ cos\,\psi\,sin\phi - cos\,\phi\,sin\psi & sin\phi sin\psi + cos\,\phi\,cos\,\psi & 0 \end{pmatrix} for\ \theta = -90deg \qquad \textbf{(163)}$$

$$= \begin{pmatrix} cos(\psi - \phi) & sin(\psi - \phi) & 0 \\ 0 & 0 & -1 \\ -sin(\psi - \phi) & cos(\psi - \phi) & 0 \end{pmatrix} for\ \theta = -90deg \qquad \textbf{(164)}$$

$$\Rightarrow tan(\psi - \phi) = \left(\frac{R_{01}}{R_{00}}\right) for\ \theta = -90deg \qquad \textbf{(165)}$$

The Windows 8 compass heading angle ρ is the negative (modulo 360°) of the yaw angle ψ:

$$\rho = -\psi \qquad \textbf{(166)}$$

The function `fWin8AnglesDegFromRotationMatrix` computes the roll ϕ, pitch θ, yaw ψ and compass heading ρ in units of degrees from the Windows 8 rotation matrix ***R***.

```
// extract the Windows 8 angles in degrees from the Windows 8 rotation matrix
void fWin8AnglesDegFromRotationMatrix(float **R, float *pfPhiDeg, float *pfTheDeg,
    float *pfPsiDeg, float *pfRhoDeg)
{
    // calculate the roll angle -90.0 <= Phi <= 90.0 deg
    // later we correct this to the range -90.0 <= Phi < 90.0 deg
    if (R[2][2] == 0.0F)
    {
        if (R[0][2] >= 0.0F)        // tan(phi) is -infinity
            *pfPhiDeg = -90.0F;
        else                            // tan(phi) is +infinity
            *pfPhiDeg = 90.0F;
    }
    else
        *pfPhiDeg = (float)atan(-R[0][2] / R[2][2]) * FRADTODEG;

    // first calculate the pitch angle in the range -90.0 <= The <= 90.0 deg
    if (R[1][2] >= 1.0F)
        *pfTheDeg = 90.0F;
    else if (R[1][2] <= -1.0F)
        *pfTheDeg = -90.0F;
    else
        *pfTheDeg = (float) asin(R[1][2]) * FRADTODEG;
    // use R[2][2]=cos(Phi)*cos(The) to correct the quadrant of The remembering
    // cos(Phi) is non-negative and that R[2][2] has the same sign as cos(The).
    // the result is -180.0 <= The < 180.0 deg
    if (R[2][2] < 0.0F)
    {
        // The is either in range 90.0 < The < 180.0 deg or -180.0 <= The < -90.0 deg
        if (*pfTheDeg > 0.0F)
            // wrap The around +90 deg
            *pfTheDeg = 180.0F - *pfTheDeg;
        if (*pfTheDeg <= 0.0F)
            // wrap The around -90 deg
            *pfTheDeg = -180.0F - *pfTheDeg;
    }

    // calculate the yaw angle -180.0 <= Psi < 180.0 deg
    if (*pfTheDeg == 90.0F)
    {
        *pfPsiDeg = (float)atan2(R[0][1], R[0][0]) * FRADTODEG - *pfPhiDeg;
    }
    else if (*pfTheDeg == -90.0F)
    {
        *pfPsiDeg = (float)atan2(R[0][1], R[0][0]) * FRADTODEG + *pfPhiDeg;
    }
    else
    {
        // general case where cos(The) is not zero
        *pfPsiDeg = (float)atan2(-cos(*pfTheDeg * FDEGTORAD) * R[1][0],
                    cos(*pfTheDeg * FDEGTORAD) * R[1][1]) * FRADTODEG;
    }

    // need to correct the special case where Phi = +90 deg
    if (*pfPhiDeg == 90.0F)
    {
        *pfPhiDeg = -90.0F;
        *pfTheDeg = 180.0F - *pfTheDeg;
        *pfPsiDeg += 180.0F;
    }

    // map +180 deg pitch onto equivalent -180 deg giving -180.0 <= The < 180.0 deg
    if (*pfTheDeg == 180.0F)
        *pfTheDeg = -180.0F;

    // map yaw angle onto range 0.0 <= Psi < 360.0 deg
    if (*pfPsiDeg < 0.0F)
        *pfPsiDeg += 360.0F;
    // check for rounding errors mapping small negative angle to 360 deg
    if (*pfPsiDeg >= 360.0F)
```

```
        *pfPsiDeg = 0.0F;

    // compute the compass angle Rho checking for the special case of 0.0 deg
    if (*pfPsiDeg == 0.0F)
        *pfRhoDeg = 0.0F;
    else
        *pfRhoDeg = 360.0F - *pfPsiDeg;

    return;
}
```

5.5 Euler angles from quaternion

Euler angles are most simply computed from the quaternion by computing the equivalent rotation matrix from the quaternion as described in section Rotation matrix from rotation quaternion and then computing the Euler angles for the required coordinate system from the rotation matrix using the functions in sections Introduction, NED/Aerospace Euler angles from rotation matrix and Android Euler angles from rotation matrix.

5.6 Euler angles from axis-angle vector

If required, this operation can be performed in two steps. First convert from Axis-Angle format to rotation matrix or quaternion format. Then, convert the rotation matrix or quaternion to Euler angles.

6 C Functions Returning Rotation Matrix

6.1 Introduction

This section documents the algorithms and C functions to compute the rotation matrix from the corresponding Euler angles, quaternion or axis-angle vector.

6.2 Rotation matrix from NED/aerospace Euler angles

Equation **(100)** defines the NED/Aerospace rotation matrix in terms of the roll, pitch and yaw Euler angles. Function `fNEDRotationMatrixFromAnglesDeg` directly implements equation **(100)**.

```
void fNEDRotationMatrixFromAnglesDeg(float **R, float fPhiDeg, float fTheDeg, float fPsiDeg)
{
    float sinPhi, cosPhi, sinThe, cosThe, sinPsi, cosPsi;        // sines and cosines

    // calculate the sines and cosines
    sinPhi = (float) sin(fPhiDeg * FDEGTORAD);
    sinThe = (float) sin(fTheDeg * FDEGTORAD);
    sinPsi = (float) sin(fPsiDeg * FDEGTORAD);
    cosPhi = (float) cos(fPhiDeg * FDEGTORAD);
    cosThe = (float) cos(fTheDeg * FDEGTORAD);
    cosPsi = (float) cos(fPsiDeg * FDEGTORAD);

    // construct the matrix
    R[0][0] = cosThe * cosPsi;
    R[0][1] = cosThe * sinPsi;
    R[0][2] = -sinThe;
    R[1][0] = cosPsi * sinThe * sinPhi - cosPhi * sinPsi;
    R[1][1] = cosPhi * cosPsi + sinPhi * sinThe * sinPsi;
    R[1][2] = cosThe * sinPhi;
    R[2][0] = cosPhi * cosPsi * sinThe + sinPhi * sinPsi;
    R[2][1] = cosPhi * sinThe * sinPsi - cosPsi * sinPhi;
    R[2][2] = cosThe * cosPhi;

    return;
}
```

6.3 Rotation matrix from android Euler angles

Equation **(116)** defines the Android rotation matrix in terms of the roll, pitch, and yaw Euler angles. Function `fAndroidRotationMatrixFromAnglesDeg` directly implements equation **(116)**.

```
void fAndroidRotationMatrixFromAnglesDeg(float **R, float fPhiDeg, float fTheDeg, float
fPsiDeg)
{
    float sinPhi, cosPhi, sinThe, cosThe, sinPsi, cosPsi;        // sines and cosines

    // calculate the sines and cosines
    sinPhi = (float) sin(fPhiDeg * FDEGTORAD);
    sinThe = (float) sin(fTheDeg * FDEGTORAD);
    sinPsi = (float) sin(fPsiDeg * FDEGTORAD);
    cosPhi = (float) cos(fPhiDeg * FDEGTORAD);
    cosThe = (float) cos(fTheDeg * FDEGTORAD);
    cosPsi = (float) cos(fPsiDeg * FDEGTORAD);

    // construct the matrix
    R[0][0] = cosPhi * cosPsi;
    R[0][1] = -cosPhi * sinPsi;
    R[0][2] = sinPhi;
    R[1][0] = cosThe * sinPsi + cosPsi * sinPhi * sinThe;
    R[1][1] = cosPsi * cosThe - sinPhi * sinThe * sinPsi;
    R[1][2] = -cosPhi * sinThe;
    R[2][0] = -cosPsi * cosThe * sinPhi + sinPsi * sinThe;
    R[2][1] = cosThe * sinPhi * sinPsi + cosPsi * sinThe;
    R[2][2] = cosPhi * cosThe;

    return;
}
```

6.4 Rotation matrix from Windows 8 Euler angles

Equation **(132)** defines the Windows 8 rotation matrix in terms of the roll, pitch and yaw Euler angles. Function `fWin8RotationMatrixFromAnglesDeg` directly implements equation **(132)**.

```
void fWin8RotationMatrixFromAnglesDeg(float **R, float fPhiDeg, float fTheDeg, float fPsiDeg)
{
    float sinPhi, cosPhi, sinThe, cosThe, sinPsi, cosPsi;        // sines and cosines

    // calculate the sines and cosines
    sinPhi = (float) sin(fPhiDeg * FDEGTORAD);
    sinThe = (float) sin(fTheDeg * FDEGTORAD);
    sinPsi = (float) sin(fPsiDeg * FDEGTORAD);
    cosPhi = (float) cos(fPhiDeg * FDEGTORAD);
    cosThe = (float) cos(fTheDeg * FDEGTORAD);
    cosPsi = (float) cos(fPsiDeg * FDEGTORAD);

    // construct the matrix
    R[0][0] = cosPhi * cosPsi - sinPhi * sinThe * sinPsi;
    R[0][1] = cosPhi * sinPsi +    cosPsi * sinPhi * sinThe;
    R[0][2] = -cosThe * sinPhi;
    R[1][0] = -cosThe * sinPsi;
    R[1][1] = cosThe * cosPsi;
    R[1][2] = sinThe;
    R[2][0] = cosPsi * sinPhi + cosPhi * sinPsi * sinThe;
    R[2][1] = sinPhi * sinPsi - cosPhi * cosPsi * sinThe;
    R[2][2] = cosPhi * cosThe;

    return;
}
```

6.5 Rotation matrix from rotation quaternion

It is straightforward to determine the rotation matrix $\boldsymbol{R}$ which is equivalent to a known rotation quaternion q by equating the results of applying the quaternion and matrix to a vector $\boldsymbol{v}$:

$$q^*\boldsymbol{v}q = \boldsymbol{R}\boldsymbol{v} \tag{167}$$

$$\Rightarrow (q_0 - q_1\boldsymbol{i} - q_2\boldsymbol{j} - q_3\boldsymbol{k})(v_x\boldsymbol{i} + v_y\boldsymbol{j} + v_z\boldsymbol{k})(q_0 + q_1\boldsymbol{i} + q_2\boldsymbol{j} + q_3\boldsymbol{k}) = \begin{pmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{pmatrix}\begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \tag{168}$$

$$\Rightarrow \begin{pmatrix} {q_0}^2 + {q_1}^2 - {q_2}^2 - {q_3}^2 & 2q_1q_2 + 2q_0q_3 & 2q_1q_3 - 2q_0q_2 \\ 2q_1q_2 - 2q_0q_3 & {q_0}^2 - {q_1}^2 + {q_2}^2 - {q_3}^2 & 2q_2q_3 + 2q_0q_1 \\ 2q_1q_3 + 2q_0q_2 & 2q_2q_3 - 2q_0q_1 & {q_0}^2 - {q_1}^2 - {q_2}^2 + {q_3}^2 \end{pmatrix}\begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{pmatrix}\begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \tag{169}$$

Since the rotation quaternion q has unit norm ${q_0}^2 + {q_1}^2 + {q_2}^2 + {q_3}^2 = 1$, equation (169) simplifies to give a unique mapping from the four components of the quaternion to the nine elements of the equivalent rotation matrix:

$$\boldsymbol{R} = \begin{pmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{pmatrix} = \begin{pmatrix} 2({q_0}^2 + {q_1}^2) - 1 & 2(q_1q_2 + q_0q_3) & 2(q_1q_3 - q_0q_2) \\ 2(q_1q_2 - q_0q_3) & 2({q_0}^2 + {q_2}^2) - 1 & 2(q_2q_3 + q_0q_1) \\ 2(q_1q_3 + q_0q_2) & 2(q_2q_3 - q_0q_1) & 2({q_0}^2 + {q_3}^2) - 1 \end{pmatrix} \tag{170}$$

Function `fRotationMatrixFromQuaternion` implements equation (170). It is assumed that the quaternion is normalized (and is therefore a valid rotation quaternion). If it is suspected that the quaternion is not normalized then it should be explicitly re-normalized to ensure that the computed rotation matrix is orthonormal.

```
void fRotationMatrixFromQuaternion(float **R, const struct fquaternion *pq)
{
    // calculate the nine rotation matrix elements assuming the quaternion
    // is correctly normalized
    R[0][0] = 2.0F * (pq->q0 * pq->q0 + pq->q1 * pq->q1) - 1.0F;
    R[0][1] = 2.0F * (pq->q1 * pq->q2 + pq->q0 * pq->q3);
    R[0][2] = 2.0F * (pq->q1 * pq->q3 - pq->q0 * pq->q2);
    R[1][0] = 2.0F * (pq->q1 * pq->q2 - pq->q0 * pq->q3);
    R[1][1] = 2.0F * (pq->q0 * pq->q0 + pq->q2 * pq->q2) - 1.0F;
    R[1][2] = 2.0F * (pq->q2 * pq->q3 + pq->q0 * pq->q1);
    R[2][0] = 2.0F * (pq->q1 * pq->q3 + pq->q0 * pq->q2);
    R[2][1] = 2.0F * (pq->q2 * pq->q3 - pq->q0 * pq->q1);
    R[2][2] = 2.0F * (pq->q0 * pq->q0 + pq->q3 * pq->q3) - 1.0F;
    return;
}
```

6.6 Rotation matrix from axis-angle vector

Equation **(70)** defines the rotation matrix in terms of the rotation axis $\hat{\boldsymbol{n}}$ and the rotation and η. Since the axis-angle representation of a rotation is the vector $\hat{\boldsymbol{n}}\eta$ containing the product of the axis and angle, the first step is to extract the rotation axis and angle from $\hat{\boldsymbol{n}}\eta$.

The rotation angle η is the modulus of the axis-angle vector $\hat{\boldsymbol{n}}\eta$. It is acceptable to take the positive square root for η provided that the sign of the rotation axis is later computed in accordance with this selection.

$$\eta = \left\| \begin{matrix} \hat{n}_x\eta \\ \hat{n}_y\eta \\ \hat{n}_z\eta \end{matrix} \right\| = \sqrt{\hat{n}_x{}^2\eta^2 + \hat{n}_y{}^2\eta^2 + \hat{n}_z{}^2\eta^2} \tag{171}$$

If $\eta = 0$, the rotation axis is irrelevant and disappears from equation **(70)** leaving the identity matrix:

$$\boldsymbol{R} = \begin{pmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} for\ \eta = 0 \tag{172}$$

If $\eta \neq 0$, the rotation axis $\hat{\boldsymbol{n}}$ is obtained by dividing the axis-angle vector by the rotation angle η. This is where the direction of the rotation axis is computed to be in agreement with the selection of the positive square root for η. If the negative square root were taken, the rotation axis $\hat{\boldsymbol{n}}$ would simply be computed in the opposite direction.

$$\hat{\boldsymbol{n}} = \left(\frac{1}{\eta}\right) \begin{pmatrix} \hat{n}_x\eta \\ \hat{n}_y\eta \\ \hat{n}_z\eta \end{pmatrix} \tag{173}$$

With $\hat{\boldsymbol{n}}$ and η both known, equation **(70)** can be directly computed.

Function `fRotationMatrixFromAxisAngleDeg` implements these equations.

```
void fRotationMatrixFromAxisAngleDeg(float **R, float **axisangledeg)
{
    float fsineta, fcoseta;                 // sin(eta) and cos(eta)
    float f1mcoseta;                     // 1.-cos(eta)
    float feta;                         // rotation angle (non-negative deg)
    float fetarad;                     // eta in radians
    float nx, ny, nz;                     // x, y, z components of normalised axis
    float fnxsineta, fnysineta, fnzsineta;        // n * sin(eta)
    float fnx1mcoseta, fny1mcoseta, fnz1mcoseta;    // n * (1 - cos(eta))

    // compute the rotation angle eta (deg) from the modulus of the axis * angle vector
    feta = (float)sqrt(axisangledeg[0][0] * axisangledeg[0][0] +
       axisangledeg[1][0] * axisangledeg[1][0] +
       axisangledeg[2][0] * axisangledeg[2][0]);

    // return identity matrix (zero rotation) if the rotation angle is zero
    if (feta == 0.0F)
    {
        R[0][0] = R[1][1]= R[2][2] = 1.0F;
        R[0][1] = R[0][2] = R[1][0] = R[1][2] = R[2][0] = R[2][1] = 0.0F;
        return;
    }

    // calculate the sine and cosine of the rotation angle eta
    fetarad = feta * FDEGTORAD;
    fsineta = (float)sin(fetarad);
    fcoseta = (float)cos(fetarad);
    f1mcoseta = 1.0F - fcoseta;

    // normalize the axis * angle vector
    nx = axisangledeg[0][0] / feta;
    ny = axisangledeg[1][0] / feta;
    nz = axisangledeg[2][0] / feta;

    // construct the rotation matrix
    fnxsineta = nx * fsineta;
    fnysineta = ny * fsineta;
    fnzsineta = nz * fsineta;
    fnx1mcoseta = nx * f1mcoseta;
    fny1mcoseta = ny * f1mcoseta;
    fnz1mcoseta = nz * f1mcoseta;
    R[0][0] = nx * fnx1mcoseta + fcoseta;
    R[1][1] = ny * fny1mcoseta + fcoseta;
    R[2][2] = nz * fnz1mcoseta + fcoseta;
    R[0][1] = nx * fny1mcoseta + fnzsineta;
    R[1][0] = nx * fny1mcoseta - fnzsineta;
    R[0][2] = nz * fnx1mcoseta - fnysineta;
    R[2][0] = nz * fnx1mcoseta + fnysineta;
    R[1][2] = ny * fnz1mcoseta + fnxsineta;
    R[2][1] = ny * fnz1mcoseta - fnxsineta;

    return;
}
```

7 C Functions Returning Rotation Quaternion

7.1 Introduction

This section documents the algorithms and C functions to compute the rotation quaternion from the corresponding Euler angles, rotation matrix or axis-angle vector.

7.2 Rotation quaternion from NED/aerospace Euler angles

Equation (105) defines the NED/Aerospace rotation matrix in terms of the roll, pitch and yaw Euler angles. Function `fQuaternionFromNEDAnglesDeg` directly implements equation (105).

```
void fQuaternionFromNEDAnglesDeg(struct fquaternion *pq, float fPhiDeg, float fTheDeg, float
fPsiDeg)
{
    float sPhi2, cPhi2;        // sin(phi/2) and cos(phi/2)
    float sThe2, cThe2;        // sin(theta/2) and cos(theta/2)
    float sPsi2, cPsi2;        // sin(psi/2) and cos(psi/2)

    // calculate sines and cosines
    sPhi2 = (float)sin(0.5F * fPhiDeg * FDEGTORAD);
    cPhi2 = (float)cos(0.5F * fPhiDeg * FDEGTORAD);
    sThe2 = (float)sin(0.5F * fTheDeg * FDEGTORAD);
    cThe2 = (float)cos(0.5F * fTheDeg * FDEGTORAD);
    sPsi2 = (float)sin(0.5F * fPsiDeg * FDEGTORAD);
    cPsi2 = (float)cos(0.5F * fPsiDeg * FDEGTORAD);

    // compute the quaternion elements
    pq->q0 = cPsi2 * cThe2 * cPhi2 + sPsi2 * sThe2 * sPhi2;
    pq->q1 = cPsi2 * cThe2 * sPhi2 - sPsi2 * sThe2 * cPhi2;
    pq->q2 = cPsi2 * sThe2 * cPhi2 + sPsi2 * cThe2 * sPhi2;
    pq->q3 = sPsi2 * cThe2 * cPhi2 - cPsi2 * sThe2 * sPhi2;

    // negate the quaternion if q0 is negative
    if (pq->q0 < 0.0F)
    {
        pq->q0 = -pq->q0;
        pq->q1 = -pq->q1;
        pq->q2 = -pq->q2;
        pq->q3 = -pq->q3;
    }

    return;
}
```

7.3 Rotation quaternion from android Euler angles

Equation (121) defines the Android rotation quaternion in terms of the roll, pitch, and yaw Euler angles. Function `fQuaternionFromAndroidAnglesDeg` directly implements equation (121).

```
void fQuaternionFromAndroidAnglesDeg(struct fquaternion *pq, float fPhiDeg, float fTheDeg,
float fPsiDeg)
{
    float sPhi2, cPhi2;         // sin(phi/2) and cos(phi/2)
    float sThe2, cThe2;         // sin(theta/2) and cos(theta/2)
    float sPsi2, cPsi2;         // sin(psi/2) and cos(psi/2)

    // calculate sines and cosines
    sPhi2 = (float)sin(0.5F * fPhiDeg * FDEGTORAD);
    cPhi2 = (float)cos(0.5F * fPhiDeg * FDEGTORAD);
    sThe2 = (float)sin(0.5F * fTheDeg * FDEGTORAD);
    cThe2 = (float)cos(0.5F * fTheDeg * FDEGTORAD);
    sPsi2 = (float)sin(0.5F * fPsiDeg * FDEGTORAD);
    cPsi2 = (float)cos(0.5F * fPsiDeg * FDEGTORAD);

    // compute the quaternion elements
    pq->q0 = cPsi2 * cThe2 * cPhi2 - sPsi2 * sThe2 * sPhi2;
    pq->q1 = -cPsi2 * sThe2 * cPhi2 - sPsi2 * cThe2 * sPhi2;
    pq->q2 = -cPsi2 * cThe2 * sPhi2 + sPsi2 * sThe2 * cPhi2;
    pq->q3 = -sPsi2 * cThe2 * cPhi2 - cPsi2 * sThe2 * sPhi2;

    // negate the quaternion if q0 is negative
    if (pq->q0 < 0.0F)
    {
        pq->q0 = -pq->q0;
        pq->q1 = -pq->q1;
        pq->q2 = -pq->q2;
        pq->q3 = -pq->q3;
    }

    return;
}
```

7.4 Rotation quaternion from Windows 8 Euler angles

Equation (137) defines the Windows 8 rotation matrix in terms of the roll, pitch and yaw Euler angles. Function `fQuaternionFromWin8AnglesDeg` directly implements equation (137).

```
void fQuaternionFromWin8AnglesDeg(struct fquaternion *pq, float fPhiDeg, float fTheDeg,
float fPsiDeg)
{
    float sPhi2, cPhi2;         // sin(phi/2) and cos(phi/2)
    float sThe2, cThe2;         // sin(theta/2) and cos(theta/2)
    float sPsi2, cPsi2;         // sin(psi/2) and cos(psi/2)

    // calculate sines and cosines
    sPhi2 = (float)sin(0.5F * fPhiDeg * FDEGTORAD);
    cPhi2 = (float)cos(0.5F * fPhiDeg * FDEGTORAD);
    sThe2 = (float)sin(0.5F * fTheDeg * FDEGTORAD);
    cThe2 = (float)cos(0.5F * fTheDeg * FDEGTORAD);
    sPsi2 = (float)sin(0.5F * fPsiDeg * FDEGTORAD);
    cPsi2 = (float)cos(0.5F * fPsiDeg * FDEGTORAD);

    // compute the quaternion elements
    pq->q0 = cPsi2 * cThe2 * cPhi2 - sPsi2 * sThe2 * sPhi2;
    pq->q1 = cPsi2 * sThe2 * cPhi2 - sPsi2 * cThe2 * sPhi2;
    pq->q2 = cPsi2 * cThe2 * sPhi2 + sPsi2 * sThe2 * cPhi2;
    pq->q3 = sPsi2 * cThe2 * cPhi2 + cPsi2 * sThe2 * sPhi2;

    // negate the quaternion if q0 is negative
    if (pq->q0 < 0.0F)
    {
        pq->q0 = -pq->q0;
        pq->q1 = -pq->q1;
        pq->q2 = -pq->q2;
        pq->q3 = -pq->q3;
    }

    return;
}
```

7.5 Rotation quaternion from rotation matrix

Determining the components of the quaternion q equivalent to the rotation matrix ***R*** is slightly more complicated since some care is needed to work around the potential for sign ambiguity when taking the square roots of expressions derived from equation **(170)**. The trace of equation **(170)** gives:

$$tr(\boldsymbol{R}) = R_{00} + R_{11} + R_{22} = 4{q_0}^2 + 2({q_0}^2 + {q_1}^2 + {q_2}^2 + {q_3}^2) - 3 = 4{q_0}^2 - 1 = 1 + 2cos\eta \quad \textbf{(174)}$$

The rotation angle η can always be assumed to always lie in the range 0° to 180° since a negative rotation angle is equivalent to positive rotation angle about a negated axis. q_0 will therefore always be non-negative and the positive square root of equation **(174)** can be taken giving:

$$q_0 = \frac{\sqrt{1 + tr(\boldsymbol{R})}}{2} = \frac{\sqrt{1 + R_{00} + R_{11} + R_{22}}}{2} \quad \textbf{(175)}$$

Differencing elements across the diagonal in equation **(170)** gives expressions for the remaining elements of the rotation quaternion which are valid except for small $q_0 = cos\left(\frac{\eta}{2}\right)$ which occurs when the rotation angle η is close to 180°.

$$q_1 = \frac{(R_{12} - R_{21})}{4q_0} when\ q_0 \neq 0, \eta \neq 180^o \quad \textbf{(176)}$$

$$q_2 = \frac{(R_{20} - R_{02})}{4q_0} when\ q_0 \neq 0, \eta \neq 180^o \quad \textbf{(177)}$$

$$q_3 = \frac{(R_{01} - R_{10})}{4q_0} when\ q_0 \neq 0, \eta \neq 180^o \quad \textbf{(178)}$$

Equations (176) to (178) become undefined for zero q_0 and numerically ill-conditioned for small q_0. Alternative expressions for the absolute values of the vector components are:

$$q_1{}^2 = \left(\frac{1+R_{00}}{2}\right) - q_0{}^2 \Rightarrow q_1 = \pm\sqrt{\left(\frac{1+R_{00}}{2}\right) - q_0{}^2} \tag{179}$$

$$q_2{}^2 = \left(\frac{1+R_{11}}{2}\right) - q_0{}^2 \Rightarrow q_2 = \pm\sqrt{\left(\frac{1+R_{11}}{2}\right) - q_0{}^2} \tag{180}$$

$$q_3{}^2 = \left(\frac{1+R_{22}}{2}\right) - q_0{}^2 \Rightarrow q_3 = \pm\sqrt{\left(\frac{1+R_{22}}{2}\right) - q_0{}^2} \tag{181}$$

Adding components across the leading diagonal allows the signs of q_2 and q_3 to be determined relative to the sign of q_1.

$$R_{01} + R_{10} = 4q_1q_2 \Rightarrow sign(R_{01} + R_{10}) = sign(q_1)sign(q_2) \tag{182}$$

$$R_{02} + R_{20} = 4q_1q_3 \Rightarrow sign(R_{02} + R_{20}) = sign(q_1)sign(q_3) \tag{183}$$

$$R_{12} + R_{21} = 4q_2q_3 \Rightarrow sign(R_{12} + R_{21}) = sign(q_2)sign(q_3) \tag{184}$$

Function `fQuaternionFromRotationMatrix` implements this algorithm to compute the rotation quaternion from the equivalent rotation matrix.

```
// compute the orientation quaternion from a rotation matrix
void fQuaternionFromRotationMatrix(float **R, struct fquaternion *pq)
{
  float fq0sq;              // q0^2
  float recip4q0;           // 1/4q0
  float fmag;               // quaternion magnitude
#define SMALLQ0 0.01F         // limit where rounding errors may appear

  // get q0^2 and q0
  fq0sq = 0.25F * (1.0F + R[0][0] + R[1][1] + R[2][2]);
  pq->q0 = (float)sqrt(fabs(fq0sq));

  // normal case when q0 is not small meaning rotation angle not near 180 deg
  if (pq->q0 > SMALLQ0)
  {
      // calculate q1 to q3
      recip4q0 = 0.25F / pq->q0;
      pq->q1 = recip4q0 * (R[1][2] - R[2][1]);
      pq->q2 = recip4q0 * (R[2][0] - R[0][2]);
      pq->q3 = recip4q0 * (R[0][1] - R[1][0]);
  }
  else
  {
      // special case of near 180 deg corresponds to nearly symmetric matrix
      // which is not numerically well conditioned for division by small q0
      // instead get absolute values of q1 to q3 from leading diagonal
      pq->q1 = (float)sqrt(fabs(0.5F * (1.0F + R[0][0]) - fq0sq));
      pq->q2 = (float)sqrt(fabs(0.5F * (1.0F + R[1][1]) - fq0sq));
      pq->q3 = (float)sqrt(fabs(0.5F * (1.0F + R[2][2]) - fq0sq));

      // first assume q1 is positive and ensure q2 and q3 are consistent with q1
      if ((R[0][1] + R[1][0]) < 0.0F)
      {
          // q1*q2 < 0 so q2 is negative
          pq->q2 = -pq->q2;
          if ((R[1][2] + R[2][1]) > 0.0F)
          {
              // q1*q2 < 0 and q2*q3 > 0 so q3 is also both negative
              pq->q3 = -pq->q3;
          }
      }
      else if ((R[0][1] + R[1][0]) > 0.0F)
      {
          if ((R[1][2] + R[2][1]) < 0.0F)
          {
              // q1*q2 > 0 and q2*q3 < 0 so q3 is negative
              pq->q3 = -pq->q3;
          }
      }

      // negate the vector components if q1 should be negative
      if ((R[1][2] - R[2][1]) < 0.0F)
      {
          pq->q1 = -pq->q1;
          pq->q2 = -pq->q2;
          pq->q3 = -pq->q3;
      }
  }

  // finally re-normalize for safety
  fmag = (float) sqrt(pq->q0 * pq->q0 + pq->q1 * pq->q1 + pq->q2 * pq->q2 + pq->q3 * pq->q3);
  if (fmag > SMALLQ0)
  {
      // normal case
      pq->q0 /= fmag;
      pq->q1 /= fmag;
      pq->q2 /= fmag;
      pq->q3 /= fmag;
    }
    else
    {
      // severe rounding errors so return identity quaternion
      pq->q0 = 1.0F;
      pq->q1 = pq->q2 = pq->q3 = 0.0F;
    }

  return;
}
```

7.6 Rotation quaternion from axis-angle vector

The rotation quaternion is computed from the axis-angle vector using a similar algorithm to that used to compute the rotation matrix from the axis-angle vector $\hat{\boldsymbol{n}}\eta$ in Rotation matrix from axis-angle vector. The rotation angle η is first computed from the modulus of $\hat{\boldsymbol{n}}\eta$. If η is non-zero, the vector component q of the rotation quaternion is computed as:

$$\boldsymbol{q} = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix} = \hat{\boldsymbol{n}} sin\left(\frac{\eta}{2}\right) = \left(\frac{sin\left(\frac{\eta}{2}\right)}{\eta}\right)\begin{pmatrix} \hat{n}_x\eta \\ \hat{n}_y\eta \\ \hat{n}_z\eta \end{pmatrix} = \frac{sin\left(\frac{\eta}{2}\right)\hat{\boldsymbol{n}}\eta}{|\hat{\boldsymbol{n}}\eta|} \; for\; \eta \neq 0 \tag{186}$$

$$\boldsymbol{q} = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix} = \hat{\boldsymbol{n}} sin\left(\frac{\eta}{2}\right) = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \; for\; \eta = 0 \tag{187}$$

The scalar component q_0 of the quaternion is non-negative since the rotation angle η can be assumed to be in the range –1800 to 1800. q_0 can then most simply be determined by the constraint that the rotation quaternion has unit norm:

$$q_0 = \sqrt{1 - {q_1}^2 - {q_2}^2 - {q_3}^2} \tag{188}$$

The function `fQuaternionFromAxisAngleDeg` implements this algorithm.

```
void fQuaternionFromAxisAngleDeg(struct fquaternion *pq, float **axisangledeg)
{
    float fetadeg;             // rotation angle (deg)
    float fetarad;             // rotation angle (rad)
    float sinhalfeta;         // sin(eta/2)
    float fvecsq;             // q1^2+q2^2+q3^2
    float ftmp;                 // scratch variable

    // compute the rotation angle eta in range 0 <= eta <= 180 deg
    fetadeg = (float) sqrt(axisangledeg[0][0] * axisangledeg[0][0] +
                       axisangledeg[1][0] * axisangledeg[1][0] +
                       axisangledeg[2][0] * axisangledeg[2][0]);

    // convert the rotation angle into radians 0 <= eta <= pi rad
    fetarad = fetadeg * FDEGTORAD;

    // compute the vector quaternion components q1, q2, q3
    sinhalfeta = (float)sin(0.5 * fetarad);
    if (fetadeg == 0.0F)
    {
        // check for zero rotation angle giving zero vector component
        pq->q1 = pq->q2 = pq->q3 = 0.0F;
    }
    else
    {
        // general case with non-zero rotation angle
        ftmp = sinhalfeta / fetadeg;
        pq->q1 = axisangledeg[0][0] * ftmp;        // q1 = nx * sin(eta/2)
        pq->q2 = axisangledeg[1][0] * ftmp;        // q2 = ny * sin(eta/2)
        pq->q3 = axisangledeg[2][0] * ftmp;        // q3 = nz * sin(eta/2)
    }

    // compute the scalar quaternion component q0 by explicit normalization
    // taking care to avoid rounding errors giving negative operand to sqrt
    fvecsq = pq->q1 * pq->q1 + pq->q2 * pq->q2 + pq->q3 * pq->q3;
    if (fvecsq < 1.0F)
    {
        // normal case
        pq->q0 = (float) sqrt(1.0F - fvecsq);
    }
    else
    {
        // rounding errors are present
        pq->q0 = 0.0F;
    }

    return;
}
```

8 C Functions Returning Rotation Axis-Angle Vector

8.1 Introduction

This section documents the algorithms and C functions to compute the Axis-Angle vector $\hat{n}\eta$ from Euler angles, rotation matrix or rotation quaternion.

8.2 Axis-angle vector from Euler angles

This operation is most simply performed in two stages by i) first converting from Euler angles to rotation matrix or to quaternion format and then ii) converting the rotation matrix or quaternion to the Axis-Angle vector.

8.3 Axis-angle vector from rotation matrix

Equation **(73)** allows the calculation of the rotation angle η from the trace of a rotation matrix which must lie in the range –1 to +3 inclusive.

$$\eta = cos^{-1}\left(\frac{tr(\boldsymbol{R}) - 1}{2}\right) \tag{189}$$

Equation **(189)** does not allow the distinguishing between negative and positive values of η and, by convention, software inverse cosine functions will return a non-negative value for η in the range 0° to 180° inclusive. As discussed previously, this is not important, provided that the rotation axis $\hat{\boldsymbol{n}}$ is selected with the correct sign for the non-negative rotation angle η. A trace equal to +3 corresponds to a rotation $\eta = 0\ deg$ deg and a trace of –1 corresponds to a rotation $\eta = +180\ deg$.

Differencing pairs of elements across the leading diagonal of equation **(70)** gives the vector $2\hat{\boldsymbol{n}}sin\eta$ with modulus $2sin\eta$ (since $sin\eta$ is non-negative). $sin\eta$ is zero for rotation angles η equal to 0° or 180°.

$$\begin{pmatrix} R_{12} - R_{21} \\ R_{20} - R_{02} \\ R_{01} - R_{10} \end{pmatrix} = 2sin\eta \begin{pmatrix} \hat{n}_x \\ \hat{n}_y \\ \hat{n}_z \end{pmatrix} = 2\hat{\boldsymbol{n}}sin\eta \Rightarrow 2sin\eta = \left\| \begin{matrix} R_{12} - R_{21} \\ R_{20} - R_{02} \\ R_{01} - R_{10} \end{matrix} \right\| \tag{190}$$

For the general case where $sin\eta$ is non-zero ($\eta \neq 0, \eta \neq 180$ deg), the Axis-Angle vector $\hat{\boldsymbol{n}}\eta$ is given by normalizing equation **(190)** :

$$\hat{\boldsymbol{n}}\eta = \begin{pmatrix} \hat{n}_x\eta \\ \hat{n}_y\eta \\ \hat{n}_z\eta \end{pmatrix} = \left(\frac{\eta}{2sin\eta}\right) \begin{pmatrix} R_{12} - R_{21} \\ R_{20} - R_{02} \\ R_{01} - R_{10} \end{pmatrix} = \eta \left\| \begin{matrix} R_{12} - R_{21} \\ R_{20} - R_{02} \\ R_{01} - R_{10} \end{matrix} \right\|^{-1} \begin{pmatrix} R_{12} - R_{21} \\ R_{20} - R_{02} \\ R_{01} - R_{10} \end{pmatrix} for\ sin\eta \neq 0, \eta \neq 0, \eta \neq 180\ deg \tag{191}$$

Normalization using division by $2sin\eta$ in equation **(191)** becomes inaccurate near angles of to 0° and 180°. Near these angles, two different approaches are required, one for 0° and one for 180°. These two cases are easily separated since the trace of the rotation matrix has value approximately +3 near 0° and approximately –1 near 180°.

For η near to 0°, the rotation matrix approximates to:

$$\boldsymbol{R} = \begin{pmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{pmatrix} = \begin{pmatrix} 1 & \hat{n}_z\eta & -\hat{n}_y\eta \\ -\hat{n}_z\eta & 1 & \hat{n}_x\eta \\ \hat{n}_y\eta & -\hat{n}_x\eta & 1 \end{pmatrix} for\ \eta\ in\ rad \tag{192}$$

The Axis-Angle vector $\hat{\boldsymbol{n}}\eta$ is then given by simply differencing terms across the leading diagonal:

$$\hat{\boldsymbol{n}}\eta = \begin{pmatrix} \hat{n}_x\eta \\ \hat{n}_y\eta \\ \hat{n}_z\eta \end{pmatrix} = \frac{1}{2}\begin{pmatrix} R_{12} - R_{21} \\ R_{20} - R_{02} \\ R_{01} - R_{10} \end{pmatrix} \; for\, small\, \eta\, in\, rad \tag{193}$$

If the rotation angle η is 180°, the rotation matrix of equation (70) simplifies to:

$$\boldsymbol{R} = \begin{pmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{pmatrix} = \begin{pmatrix} 2\hat{n}_x{}^2 - 1 & 2\hat{n}_x\hat{n}_y & 2\hat{n}_x\hat{n}_z \\ 2\hat{n}_x\hat{n}_y & 2\hat{n}_y{}^2 - 1 & 2\hat{n}_y\hat{n}_z \\ 2\hat{n}_x\hat{n}_z & 2\hat{n}_y\hat{n}_z & 2\hat{n}_z{}^2 - 1 \end{pmatrix} for\, \eta\, near\, 180\, deg \tag{194}$$

The absolute values of the three components of the axis vector can be computed from the leading diagonal elements of the rotation matrix.

$$|\hat{n}_x| = \sqrt{\frac{1 + R_{00}}{2}} for\, \eta\, near\, 180\, deg \tag{195}$$

Equations (196) and (197) can be used to compute the absolute value of the z and z axis components.

$$|\hat{n}_y| = \sqrt{\frac{1 + R_{11}}{2}} for\, \eta\, near\, 180\, deg \tag{196}$$

$$|\hat{n}_z| = \sqrt{\frac{1 + R_{22}}{2}} for\, \eta\, near\, 180\, deg \tag{197}$$

The unknown signs in equation (196) and (197) can be determined using the signs of the summed off-diagonal elements in the same manner as for the quaternion vector elements discussed in Rotation quaternion from rotation matrix.

$$R_{01} + R_{10} = 4\hat{n}_x\hat{n}_y\; for\, \eta\, near\, 180\, deg \Rightarrow sign(R_{01} + R_{10}) = sign(\hat{n}_x)sign(\hat{n}_y) \tag{198}$$

$$R_{02} + R_{20} = 4\hat{n}_x\hat{n}_z\; for\, \eta\, near\, 180\, deg \Rightarrow sign(R_{02} + R_{20}) = sign(\hat{n}_x)sign(\hat{n}_z) \tag{199}$$

$$R_{12} + R_{21} = 4\hat{n}_y\hat{n}_z\; for\, \eta\, near\, 180\, deg \Rightarrow sign(R_{12} + R_{21}) = sign(\hat{n}_y)sign(\hat{n}_z) \tag{200}$$

The signs of $\hat{n}_y$ and $\hat{n}_z$ are first calculated to be consistent on the assumption that $\hat{n}_x$ is positive and then the entire rotation vector is negated if $\hat{n}_x$ should be positive using the result:

$$R_{12} - R_{21} = 2\hat{n}_x sin\eta \Rightarrow sign(\hat{n}_x) = sign(R_{12} - R_{21}) \tag{201}$$

The function `fAxisAngleDegFromRotationMatrix` following implements this algorithm.

```
void fAxisAngleDegFromRotationMatrix(float **R, float **axisangledeg)
{
    float ftrace;             // trace of the rotation matrix
    float fetadeg;          // rotation angle eta (deg)
    float fmodulus;          // modulus of axis * angle vector = 2|sin(eta)|
    float ftmp;              // scratch variable
#define SMALLMODULUS 0.01F        // limit where rounding errors may appear

    // calculate the trace of the rotation matrix = 1+2cos(eta) in range -1 to +3
    // and eta (deg) in range 0 to 180 deg inclusive
    // checking for rounding errors that might take the trace outside this range
    ftrace = R[0][0] + R[1][1] + R[2][2];
    if (ftrace >= 3.0F)
    {
        fetadeg = 0.0F;
    }
    else if (ftrace <= -1.0F)
    {
        fetadeg = 180.0F;
    }
    else
    {
        fetadeg = (float)acos(0.5F * (ftrace - 1.0F)) * FRADTODEG;
    }

    // set the axisangledeg vector to differences across the diagonal = 2*n*sin(eta)
    // and calculate its modulus equal to 2|sin(eta)|
    // the modulus approaches zero near 0 and 180 deg (when sin(eta) approaches zero)
    axisangledeg[0][0] = R[1][2] - R[2][1];
    axisangledeg[1][0] = R[2][0] - R[0][2];
    axisangledeg[2][0] = R[0][1] - R[1][0];
    fmodulus = (float) sqrt(axisangledeg[0][0] * axisangledeg[0][0] +
                    axisangledeg[1][0] * axisangledeg[1][0] +
                    axisangledeg[2][0] * axisangledeg[2][0]);

    // handle numerically stable case away from 0 and 180 deg rotation
    if (fmodulus > SMALLMODULUS)
    {
        ftmp = fetadeg / fmodulus;
        axisangledeg[0][0] *= ftmp;    // set x component to eta(deg) * nx
        axisangledeg[1][0] *= ftmp;    // set y component to eta(deg) * ny
        axisangledeg[2][0] *= ftmp;    // set z component to eta(deg) * nz
    }
    else if (ftrace >= 0.0F)
    {
        // near 0 deg rotation (trace = 3): matrix is nearly identity matrix
        // R12-R21=2*nx*eta(rad) and similarly for other components
        axisangledeg[0][0] *= (0.5F * FRADTODEG);
        axisangledeg[1][0] *= (0.5F * FRADTODEG);
        axisangledeg[2][0] *= (0.5F * FRADTODEG);
    }
    else
    {
        // near 180 deg (trace = -1): matrix is nearly symmetric
        // calculate the absolute value of the components of the axis-angle vector
        axisangledeg[0][0] = 180.0F * (float)sqrt(fabs(0.5F * (R[0][0] + 1.0F)));
        axisangledeg[1][0] = 180.0F * (float)sqrt(fabs(0.5F * (R[1][1] + 1.0F)));
        axisangledeg[2][0] = 180.0F * (float)sqrt(fabs(0.5F * (R[2][2] + 1.0F)));

        // first assume nx is positive and ensure ny and nz are consistent with nx
        if ((R[0][1] + R[1][0]) < 0.0F)
        {
            // nx*ny < 0 so ny is negative
            axisangledeg[1][0] = -axisangledeg[1][0];
            if ((R[1][2] + R[2][1]) > 0.0F)
            {
                // nx*ny < 0 and ny*nz > 0 so nz is negative
                axisangledeg[2][0] = -axisangledeg[2][0];
            }
        }
        else if ((R[0][1] + R[1][0]) > 0.0F)
        {
            if ((R[1][2] + R[2][1]) < 0.0F)
```

```
                // nx*ny > 0 and ny*nz < 0 so ny is positive and nz negative
                axisangledeg[2][0] = -axisangledeg[2][0];
        }

        // negate the axis if nx should be negative
        if ((R[1][2] - R[2][1]) < 0.0F)
        {
            axisangledeg[0][0] = -axisangledeg[0][0];
            axisangledeg[1][0] = -axisangledeg[1][0];
            axisangledeg[2][0] = -axisangledeg[2][0];
        }
    }

    return;
}
```

8.4 Axis-angle vector from rotation quaternion

By the definition of a rotation quaternion, the rotation angle η and $sin\left(\frac{\eta}{2}\right)$ can be computed from the scalar component q_0:

$$\eta = 2cos^{-1}(q_0) \tag{202}$$

$$sin\left(\frac{\eta}{2}\right) = sin\left(cos^{-1}(q_0)\right) \tag{203}$$

In the general case where $sin\left(\frac{\eta}{2}\right)$ is non-zero, the axis-angle vector is simply found by scaling the vector components of the rotation quaternion.

$$\hat{\boldsymbol{n}}\eta = \begin{pmatrix} \hat{n}_x\eta \\ \hat{n}_y\eta \\ \hat{n}_z\eta \end{pmatrix} = \frac{1}{sin\left(\frac{\eta}{2}\right)}\begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix} \; for \; sin\left(\frac{\eta}{2}\right) \neq 0, \eta \neq 0 \tag{204}$$

If $sin\left(\frac{\eta}{2}\right) = 0$, the rotation angle is zero and the axis-angle vector is zero:

$$\hat{\boldsymbol{n}}\eta = \begin{pmatrix} \hat{n}_x\eta \\ \hat{n}_y\eta \\ \hat{n}_z\eta \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \; for \; sin\left(\frac{\eta}{2}\right) = 0, \eta = 0 \tag{205}$$

Function `fAxisAngleDegFromQuaternion` implements this algorithm to compute the axis-angle vector from a rotation quaternion. The algorithm checks for rounding errors leading to the scalar component of the quaternion q_0 lying outside the range –1 to +1 and does not assume that the sign convention that q_0 is non-negative is followed. If q_0 is negative, with a corresponding sign change in the scalar components of the quaternion, both the rotation angle and rotation axis are negated leading to the same axis angle vector.

```
void fAxisAngleDegFromQuaternion(const struct fquaternion *pq, float **axisangledeg)
{
    float fetarad;              // rotation angle (rad)
    float fetadeg;              // rotation angle (deg)
    float sinhalfeta;              // sin(eta/2)
    float ftmp;                  // scratch variable

    // calculate the rotation angle in the range 0 <= eta < 360 deg
    if ((pq->q0 >= 1.0F) || (pq->q0 <= -1.0F))
    {
        // rotation angle is 0 deg or 2*180 deg = 360 deg = 0 deg
        fetarad = 0.0F;
        fetadeg = 0.0F;
    }
    else
    {
        // general case returning 0 < eta < 360 deg
        fetarad = 2.0F * (float) acos(pq->q0);
        fetadeg = fetarad * FRADTODEG;
    }

    // map the rotation angle onto the range -180 deg <= eta < 180 deg
    if (fetadeg >= 180.0F)
    {
        fetadeg -= 360.0F;
        fetarad = fetadeg * FDEGTORAD;
    }

    // calculate sin(eta/2) which will be in the range -1 to +1
    sinhalfeta = (float)sin(0.5F * fetarad);

    // calculate the axis * angle vector (deg)
    if (sinhalfeta == 0.0F)
    {
        // the rotation angle eta is zero and the axis is irrelevant
        axisangledeg[0][0] = 0.0F;
        axisangledeg[1][0] = 0.0F;
        axisangledeg[2][0] = 0.0F;
    }
    else
    {
        // general case with non-zero rotation angle
        ftmp = fetadeg / sinhalfeta;
        axisangledeg[0][0] = pq->q1 * ftmp;
        axisangledeg[1][0] = pq->q2 * ftmp;
        axisangledeg[2][0] = pq->q3 * ftmp;
    }

    return;
}
```

Document Number: AN4676
Rev. 1.0, 4/2013