Mark Tincknell
MIT Lincoln Laboratory
15 August 2013

matlab quaternion class

`quaternion.m` is a matlab class that implements quaternion mathematical operations, 3 dimensional rotations, transformations of rotations among several representations, and numerical propagation of Euler's equations for rotational motion. All `quaternion.m` class methods except `PropagateEulerEq` are fully vectorized.

Quaternions are a generalization of complex numbers. Quaternions have the form

$$q = e_1 + i \cdot e_2 + j \cdot e_3 + k \cdot e_4$$

where e_1, e_2, e_3, e_4 are real, and

$$i \cdot j = k,\ j \cdot i = -k,\ j \cdot k = i,\ k \cdot j = -i,\ k \cdot i = j,\ i \cdot k = -j,\ i \cdot i = j \cdot j = k \cdot k = -1.$$

Normalized quaternions can represent rotations in 3 dimensional space, and offer several conveniences over other representations of rotations. Other representations of 3D rotations include:

- angle-axis, an axis vector, and a rotation angle around that axis
- Euler angles, a set of 3 orthogonal body axes and 3 rotation angles about those axes
- Rotation or Direction Cosine Matrices, 3x3 orthogonal matrices

The convention used in this matlab class is that all rotation operations operate from left to right on 3x1 column vectors and create rotated vectors, not representations of those vectors in rotated coordinate systems.

Euler's equations are 3 coupled nonlinear differential equations for 3 orthogonal body angular accelerations as a function of the 3 body angular rotation rates (ω), 3 principal moments of inertia (I), and 3 torques (τ):

$$\begin{bmatrix} \dot{\omega}_1 \\ \dot{\omega}_2 \\ \dot{\omega}_3 \end{bmatrix} = \begin{bmatrix} \omega_2 \omega_3 \left(I_{22} - I_{33}\right) / I_{11} \\ \omega_3 \omega_1 \left(I_{33} - I_{11}\right) / I_{22} \\ \omega_1 \omega_2 \left(I_{11} - I_{22}\right) / I_{33} \end{bmatrix} + \begin{bmatrix} \tau_1 / I_{11} \\ \tau_2 / I_{22} \\ \tau_3 / I_{33} \end{bmatrix}$$

Euler's equations have complicated solutions, particularly in the case of torques, that make them most conveniently solved numerically.

The class help text for `quaternion.m`, which implements all of these functions, is printed below.

Acknowledgements to Charles Meins (MIT LL), Ethan Phelps (Raytheon and MIT LL), and John Fuller (National Institute of Aerospace). Helpful URLs:

http://www.mathworks.com/matlabcentral/fileexchange/33341-quaternion-m http://www.mathworks.com/matlabcentral/fileexchange/20696-function-to-convert-between-dcm-euler-angles-quaternions-and-euler-vectors

http://en.wikipedia.org/wiki/Rotation_formalisms_in_three_dimensions

http://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles

http://mathworld.wolfram.com/EulerAngles.html

Examples

```
>> q = quaternion( [1,2,3,4] )
q     = (1           ) + i(2           ) + j(3           ) + k(4           )
>> qn = q.normalize
qn    = (0.18257     ) + i(0.36515     ) + j(0.54772     ) + k(0.7303      )
>> [angle, axis] = qn.AngleAxis
angle =
       2.7744
axis =
      0.37139
      0.55709
      0.74278
>> angles = qn.EulerAngles( '123' )
angles =
       1.4289
     -0.33984
       2.3562
>> R = qn.RotationMatrix
R =  -0.66667      0.13333      0.73333
      0.66667     -0.33333      0.66667
      0.33333      0.93333      0.13333
>> equiv( qn, quaternion.angleaxis( angle, axis ))
ans =
     1
>> equiv( qn, quaternion.eulerangles( '123', angles ))
ans =
     1
>> equiv( qn, quaternion.rotationmatrix( R ), eps(2) )
ans =
     1
```

quaternion.m help

```
classdef quaternion, implements quaternion mathematics and 3D rotations

Properties (SetAccess = protected):
 e(4,1)   components, basis [1; i; j; k]: e(1) + i*e(2) + j*e(3) + k*e(4)
          i*j=k, j*i=-k, j*k=i, k*j=-i, k*i=j, i*k=-j, i*i = j*j = k*k = -1

Constructors:
 q  = quaternion                scalar zero quaternion, q.e = [0;0;0;0]
 q  = quaternion(x)             x is a matrix size [4,s1,s2,...] or [s1,4,s2,...],
                                q is size [s1,s2,...], q(i1,i2,...).e = ...
                                x(1:4,i1,i2,...) or x(i1,1:4,i2,...).'
 q  = quaternion(v)             v is a matrix size [3,s1,s2,...] or [s1,3,s2,...],
                                q is size [s1,s2,...], q(i1,i2,...).e = ...
                                [0;v(1:3,i1,i2,...)] or [0;v(i1,1:3,i2,...).']
 q  = quaternion(c)             c is a complex matrix size [s1,s2,...],
                                q is size [s1,s2,...], q(i1,i2,...).e = ...
                                [real(c(i1,i2,...));imag(c(i1,i2,...));0;0]
 q  = quaternion(x1,x2)         x1,x2 are matrices size [s1,s2,...] or scalars,
                                q(i1,i2,...).e = [x1(i1,i2,...);x2(i1,i2,...);0;0]
 q  = quaternion(v1,v2,v3)      v1,v2,v3 matrices size [s1,s2,...] or scalars,
                                q(i1,i2,...).e = [0;v1(i1,i2,...);v2(i1,i2,...);...
                                v3(i1,i2,...)]
 q  = quaternion(x1,x2,x3,x4) x1,x2,x3,x4 matrices size [s1,s2,...] or scalars,
                                q(i1,i2,...).e = [x1(i1,i2,...);x2(i1,i2,...);...
                                x3(i1,i2,...);x4(i1,i2,...)]

Quaternion array constructor methods:
 q  = quaternion.eye(N)         quaternion NxN identity matrix
 q  = quaternion.nan(siz)       q(:).e = [NaN;NaN;NaN;NaN]
 q  = quaternion.ones(siz)      q(:).e = [1;0;0;0]
 q  = quaternion.rand(siz)      uniform random quaternions, NOT normalized
                                to 1, 0 <= q.e(1) <= 1, -1 <= q.e(2:4) <= 1
 q  = quaternion.randRot(siz) random quaternions uniform in rotation space
 q  = quaternion.zeros(siz)     q(:).e = [0;0;0;0]
```

```
Rotation constructor methods (all lower case):
 q  = quaternion.angleaxis(angle,axis)
                              angle is an array in radians, axis is an array
                              of vectors size [3,s1,s2,...] or [s1,3,s2,...],
                              q is size [s1,s2,...], quaternions normalized to 1
                              equivalent to rotations about axis by angle
 q  = quaternion.eulerangles(axes,angles) or
 q  = quaternion.eulerangles(axes,ang1,ang2,ang3)
                              axes is a string array or cell string array,
                              '123' = 'xyz' = 'XYZ' = 'ijk', etc.,
                              angles is an array of Euler angles in radians,
                              size [3,s1,s2,...] or [s1,3,s2,...], or
                              (ang1, ang2, ang3) are arrays or scalars of
                              Euler angles in radians, q is size
                              [s1,s2,...], quaternions normalized to 1
                              equivalent to Euler Angle rotations
 q  = quaternion.rotateutov(u,v,dimu,dimv)
                              quaternions normalized to 1 that rotate 3
                              element vectors u into the directions of 3
                              element vectors v
 q  = quaternion.rotationmatrix(R)
                              R is an array of rotation or Direction Cosine
                              Matrices size [3,3,s1,s2,...] with det(R) == 1,
                              q(i1,i2,...) = quaternions normalized to 1,
                              equivalent to R(1:3,1:3,i1,i2,...)

Rotation methods (Mixed Case):
 [angle,axis] = AngleAxis(q)  angles in radians, unit vector rotation axes
                              equivalent to q
 qd = Derivative(q,w)         quaternion derivatives, w are 3 component
                              angular velocity vectors, qd = 0.5*q*quaternion(w)
 angles = EulerAngles(q,axes) angles are 3 Euler angles equivalent to q, axes
                              are strings or cell strings, '123' = 'xyz', etc.
 [omega,axis] = OmegaAxis(q,t,dim)
                              instantaneous angular velocities and rotation axes
 PlotRotation(q,interval)     plot columns of rotation matrices of q,
```

```
                                pause interval between figure updates in seconds
 [q1,w1,t1] = PropagateEulerEq(q0,w0,I,t,@torque,odeoptions)
                                Euler equation numerical propagator, see
                                help quaternion.PropagateEulerEq
 vp = RotateVector(q,v,dim)     vp are 3 component vectors, rotations q acting
                                on vectors v, uses rotation matrix multiplication
 vp = RotateVectorQ(q,v,dim)    vp are 3 component vectors, rotations q acting
                                on vectors v, uses quaternion multiplication,
                                RotateVector is 7 times faster than RotateVectorQ
 R  = RotationMatrix(q)         3x3 rotation matrices equivalent to q

Note:
 In all rotation operations, the rotations operate from left to right on
 3x1 column vectors and create rotated vectors, not representations of
 those vectors in rotated coordinate systems.
 For Euler angles, '123' means rotate the vector about x first, about y
 second, about z third, i.e.:
 vp = rotate(z,angle(3)) * rotate(y,angle(2)) * rotate(x,angle(1)) * v

Ordinary methods:
 n  = abs(q)                    quaternion norm, n = sqrt( sum( q.e.^2 ))
 q3 = bsxfun(func,q1,q2)        binary singleton expansion of operation func
 c  = complex(q)                complex( real(q), imag(q) )
 qc = conj(q)                   quaternion conjugate, qc.e =
                                [q.e(1);-q.e(2);-q.e(3);-q.e(4)]
 qt = ctranspose(q)             qt = q'; quaternion conjugate transpose,
                                2-D (or scalar) q only
 qp = cumprod(q,dim)            cumulative quaternion array product over
                                dimension dim
 qs = cumsum(q,dim)             cumulative quaternion array sum over dimension dim
 qd = diff(q,ord,dim)           quaternion array difference, order ord, over
                                dimension dim
 ans = display(q)               'q = ( e(1) ) + i( e(2) ) + j( e(3) ) + k( e(4) )'
 d  = dot(q1,q2)                quaternion element dot product, d = dot(q1.e,q2.e)
 d  = double(q)                 d = q.e; if size(q) == [s1,s2,...], size(d) ==
                                [4,s1,s2,...]
 l  = eq(q1,q2)                 quaternion equality, l = all( q1.e == q2.e )
```

```
l  = equiv(q1,q2,tol)         quaternion rotational equivalence, within
                              tolerance tol, l = (q1 == q2) | (q1 == -q2)
qe = exp(q)                   quaternion exponential, v = q.e(2:4), qe.e =
                              exp(q.e(1))*[cos(|v|);v.*sin(|v|)./|v|]
ei = imag(q)                  imaginary e(2) components
qi = interp1(t,q,ti,method)   interpolate quaternion array
qi = inverse(q)               quaternion inverse, qi = conj(q)./norm(q).^2,
                              q .* qi = qi .*.q = 1 for q ~= 0
l  = isequal(q1,q2,...)       true if equal sizes and values
l  = isequaln(q1,q2,...)      true if equal including NaNs
l  = isequalwithequalnans(q1,q2,...) true if equal including NaNs
l  = isfinite(q)              true if all( isfinite( q.e ))
l  = isinf(q)                 true if any( isinf( q.e ))
l  = isnan(q)                 true if any( isnan( q.e ))
ej = jmag(q)                  e(3) components
ek = kmag(q)                  e(4) components
q3 = ldivide(q1,q2)           quaternion left division, q3 = q1 \. q2 =
                              inverse(q1) *. q2
ql = log(q)                   quaternion logarithm, v = q.e(2:4), ql.e =
                              [log(|q|);v.*acos(q.e(1)./|q|)./|v|]
q3 = minus(q1,q2)             quaternion subtraction, q3 = q1 - q2
q3 = mldivide(q1,q2)          left division only defined for scalar q1
qp = mpower(q,p)              quaternion matrix power, qp = q^p, p scalar
                              integer >= 0, q square quaternion matrix
q3 = mrdivide(q1,q2)          right division only defined for scalar q2
q3 = mtimes(q1,q2)            2-D matrix quaternion multiplication, q3 = q1 * q2
l  = ne(q1,q2)                quaternion inequality, l = ~all( q1.e == q2.e )
n  = norm(q)                  quaternion norm, n = sqrt( sum( q.e.^2 ))
[q,n] = normalize(q)          make quaternion norm == 1, unless q == 0,
                              n = matrix of previous norms
q3 = plus(q1,q2)              quaternion addition, q3 = q1 + q2
qp = power(q,p)               quaternion power, qp = q.^p
qp = prod(q,dim)              quaternion array product over dimension dim
qp = product(q1,q2)           quaternion product of scalar quaternions,
                              qp = q1 .* q2, noncommutative
q3 = rdivide(q1,q2)           quaternion right division, q3 = q1 ./ q2 =
                              q1 .* inverse(q2)
```

er = real(q)	real e(1) components
qs = slerp(q0,q1,t)	quaternion spherical linear interpolation
qr = sqrt(q)	qr = q.^0.5, square root
qs = sum(q,dim)	quaternion array sum over dimension dim
q3 = times(q1,q2)	matrix component quaternion multiplication, q3 = q1 .* q2, noncommutative
qm = uminus(q)	quaternion negation, qm = -q
qp = uplus(q)	quaternion unitary plus, qp = +q
ev = vector(q)	vector e(2:4) components

quaternion method help

quaternion.angleaxis

```
function q = quaternion.angleaxis( angle, axis )
Construct quaternions from rotation axes and rotation angles
Inputs:
 angle    array of rotation angles in radians
 axis     3xN or Nx3 array of axes (need not be unit vectors)
Output:
 q        quaternion array
```

quaternion.AngleAxis

```
function [angle, axis] = AngleAxis( q )  or  [angle, axis] = q.AngleAxis
Construct angle-axis pairs equivalent to quaternion rotations
Input:
 q        quaternion array
Outputs:
 angle    rotation angles in radians, 0 <= angle <= 2*pi
 axis     3xN or Nx3 rotation axis unit vectors
Note: angle and axis are constructed so at least 2 out of 3 elements of
axis are >= 0.
```

quaternion.bsxfun

```
function q3 = bsxfun( func, q1, q2 )
Binary Singleton Expansion for quaternion arrays. Apply the element by
element binary operation specified by the function handle func to arrays
q1 and q2. All dimensions of q1 and q2 must either agree or be length 1.
Inputs:
 func     function handle (e.g. @plus) of quaternion function or operator
 q1(n1)   quaternion array
 q2(n2)   quaternion array
Output:
 q3(n3)   quaternion array of function or operator outputs
          size(q3) = max( size(q1), size(q2) )
```

quaternion.dot

```
function d = dot( q1, q2 )
```

quaternion element dot product: d = dot(q1.e, q2.e), using binary singleton expansion of quaternion arrays

dn = dot(q1, q2)/(norm(q1) * norm(q2)) is the cosine of the angle in 4D space between 4D vectors q1.e and q2.e

---

quaternion.equiv

```
function l  = equiv( q1, q2, tol )
```

quaternion rotational equivalence, within tolerance tol,

l = (q1 == q2) | (q1 == -q2)

optional argument tol (default = eps) sets tolerance for difference from exact equality

---

quaternion.eulerangles

```
function q = quaternion.eulerangles( axes, angles )  OR
function q = quaternion.eulerangles( axes, ang1, ang2, ang3 )
```

Construct quaternions from triplets of axes and Euler angles

```
Inputs:
 axes                 string array or cell string array
                      '123' = 'xyz' = 'XYZ' = 'ijk', etc.
 angles               3xN or Nx3 array of angles in radians  OR
 ang1, ang2, ang3     arrays of angles in radians
Output:
 q                    quaternion array
```

---

quaternion.EulerAngles

```
function angles = EulerAngles( q, axes )   or   angles = q.EulerAngles( axes )
```

Construct Euler angle triplets equivalent to quaternion rotations

```
Inputs:
 q        quaternion array
 axes     axes designation strings (e.g. '123' = xyz) or cell strings
          (e.g. {'123'})
Output:
 angles   3 element Euler Angle vectors in radians
```

---

quaternion.exp

```
function qe = exp( q )
quaternion exponential, v = q.e(2:4),
qe.e = exp(q.e(1))*[cos(|v|);v.*sin(|v|)./|v|]
```

---

quaternion.interp1

```
function qi = interp1( t, q, ti, method ) or
         qi = q.interp1( t, ti, method )  or
         qi = interp1( q, ti, method )
Interpolate quaternion array. If q are rotation quaternions (i.e.
normalized to 1), then -q is equivalent to q, and the sign of q to use as
the second knot of the interpolation is chosen by which ever is closer to
the first knot. Extrapolation (i.e. ti < min(t) or ti > max(t)) gives
qi = quaternion.nan.
Inputs:
 t(nt)      array of ordinates (e.g. times); if t is not provided t=1:nt
 q(nt,nq)   quaternion array
 ti(ni)     array of query (interpolation) points, t(1) <= ti <= t(end)
 method [OPTIONAL] 'slerp' or 'linear'; default = 'slerp'
Output:
 qi(ni,nq)  interpolated quaternion array
```

---

quaternion.log

```
function ql = log( q )
quaternion logarithm, v = q.e(2:4), ql.e = [log(|q|);v.*acos(q.e(1)./|q|)./|v|]
logarithm of negative real quaternions is ql.e = [log(|q|);pi;0;0]
```

---

quaternion.normalize

```
function [q, n] = normalize( q )
q = quaternions with norm == 1 (unless q == 0), n = former norms
```

---

quaternion.OmegaAxis

```
function [omega, axis] = OmegaAxis( q, t, dim )  or
         [omega, axis] = q.OmegaAxis( t, dim )
Estimate instantaneous angular velocities and rotation axes from a time
series of quaternions. The angular velocity vector omegav is computed by:
```

```
 omegav(:,1) = vector( 2*log( q(1) * inverse(q(2)) )/(t(2) - t(1)) );
 omegav(:,i) = vector(...
    (log( q(i-1) * inverse(q(i)) ) + log( q(i) * inverse(q(i+1))) )/...
    (0.5*(t(i+1) - t(i-1))) );
 omegav(:,end) = vector( 2*log( q(end-1) * inverse(q(end)) )/...
    (t(end) - t(end-1)) );
 [axis, omega] = unitvector( omegav );
Inputs:
 q           array of normalized (rotation) quaternions
 t    [OPT]  array of monotonically increasing (or decreasing) times.
             if omitted or empty, unit time steps are assumed.
             t must either be a vector with the same length as dimension
             dim of q, or the same size as q.
 dim [OPT]   dimension of q that is varying in time; if omitted or empty,
             the first non-singleton dimension is used.
Outputs:
 omega       array of instantaneous angular velocities, radians/(unit time)
             omega >= 0
 axis        instantaneous 3D rotation axis unit vectors at each time
```

---

quaternion.PlotRotation

```
function PlotRotation( q, interval )   or   q.PlotRotation( interval )
Inputs:
 q           quaternion array
 interval    pause between figure updates in seconds, default = 0.1
Output:
 figure plotting the 3 Cartesian axes orientations for the series of
 quaternions in array q
```

---

quaternion.PropagateEulerEq

```
function [q1, w1, t1] = PropagateEulerEq( q0, w0, I, t, torque, odeoptions )
Inputs:
 q0            initial orientation quaternion (normalized, scalar)
 w0(3)         initial body frame angular velocity vector
 I(3)          principal body moments of inertia (if no torque, only
               ratios of elements of I are used)
```

```
 t(nt)         initial and subsequent (or previous) times t = [t0,t1,...]
               (monotonic)
 @torque [OPTIONAL] function handle to calculate torque vector:
               tau(1:3) = torque( t, y ), where y = [q.e(1:4); w(1:3)]
 odeoptions [OPTIONAL] ode45 options
Outputs:
 q1(1,nt)      array of normalized quaternions at times t1
 w1(3,nt)      array of body frame angular velocity vectors at times t1
 t1(1,nt)      array of output times
Calls:
 Derivative    quaternion derivative method
 odeset        matlab ode options setter
 ode45         matlab ode numerical differential equation integrator
 torque [OPTIONAL] user-supplied torque as function of time, orientation,
               and angular rates; default is no torque
```

---

quaternion.randRot

```
function q = quaternion.randRot( siz )
Random quaternions uniform in rotation space
Input:
 siz       size of output array q
Output:
 q         random quaternions, normalized to 1, 0 <= q.e(1) <= 1,
           uniform over the 3D surface of a 4 dimensional hypersphere
```

---

quaternion.rotateutov

```
function q = quaternion.rotateutov( u, v, dimu, dimv )
Construct quaternions to rotate vectors u into directions of vectors v
Inputs:
 u         3x1 or 3xN or 1x3 or Nx3 arrays of vectors
 v         3x1 or 3xN or 1x3 or Nx3 arrays of vectors
 dimu [OPTIONAL] dimension of u with size 3 to use
 dimv [OPTIONAL] dimension of v with size 3 to use
Output:
 q         quaternion array
```

---

quaternion.RotateVector

```
function vp = RotateVector( q, v, dim ) or
         vp = q.RotateVector( v, dim )
3x3 rotation matrices are created from q and matrix multiplication
rotates v into vp. RotateVector is 7 times faster than RotateVectorQ.
Inputs:
 q        quaternion array
 v        3xN or Nx3 element Cartesian vectors
 dim [OPTIONAL] dimension of v with size 3 to rotate
Output:
 vp       3xN or Nx3 element rotated vectors
```

---

quaternion.rotationmatrix

```
function q = quaternion.rotationmatrix( R )
Construct quaternions from rotation (or direction cosine) matrices
Input:
 R        3x3xN rotation (or direction cosine) matrices
Output:
 q        quaternion array
```

---

quaternion.RotationMatrix

```
function R = RotationMatrix( q )   or   R = q.RotationMatrix
Construct rotation (or direction cosine) matrices from quaternions
Input:
 q        quaternion array
Output:
 R        3x3xN rotation (or direction cosine) matrices
```

---

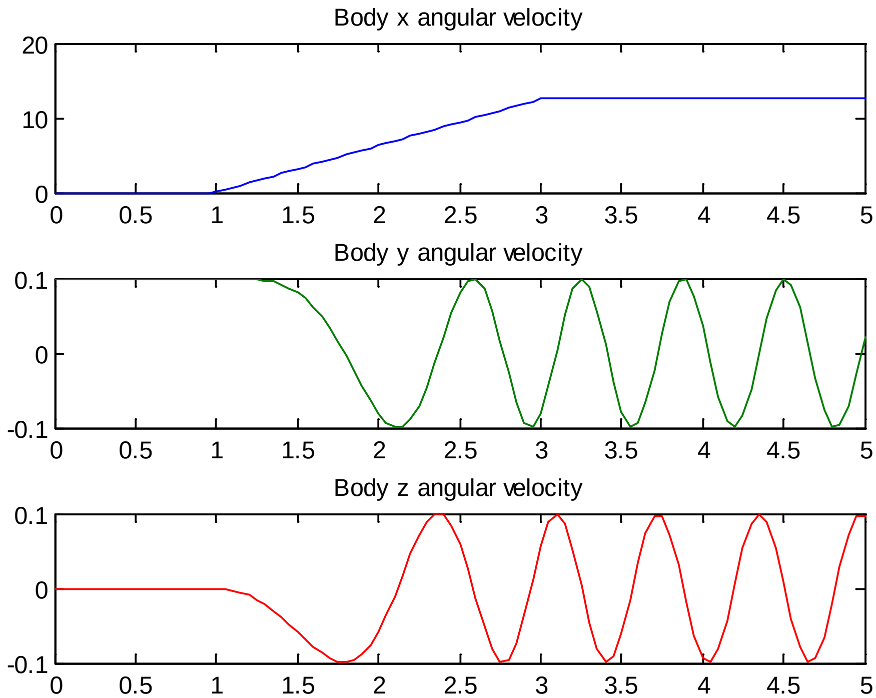
quaternion.slerp

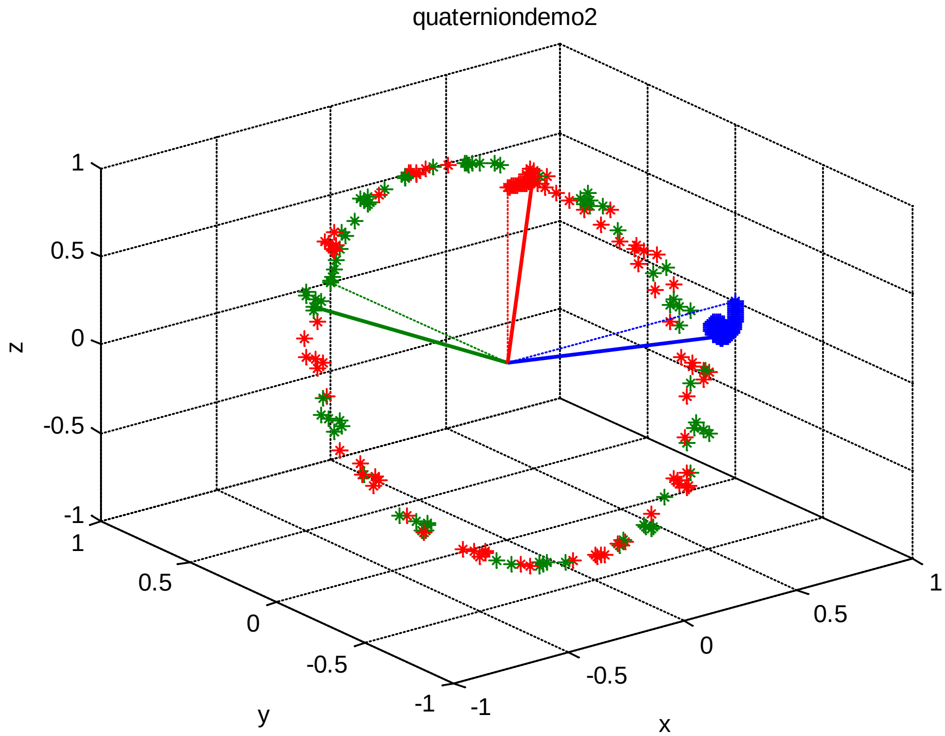
```
function qs = slerp( q0, q1, t )
quaternion spherical linear interpolation, qs = q0.*(q0.inverse.*q1).^t,
default t = 0.5; see http://en.wikipedia.org/wiki/Slerp
```

PropagateEulerEq Demonstration

```
function quaterniondemo2

 quaternion demo 2, Reentry Vehicle tip off on separation and spin-up
```





quaterniondemo2
z
1
0.5
0
-0.5
-1
y
1
0.5
0
-0.5
-1
x
-1
-0.5
0
0.5
1