Documentation for Sphere.h and Sphere.c

Steve Andrews

2005 - 2024

1 Header file: Sphere.h

```
/* Steven Andrews 2/05.
See documentation called Sphere_doc.doc.
Copyright 2005-2007 by Steven Andrews.  This work is distributed under the terms
of the Gnu Lesser General Public License (LGPL). */


#ifndef __Sphere_h
#define __Sphere_h

/*
Cart = Cartesian coordinates (x, y, z)
Sc = Spherical coordinates (r, theta, phi)
Dcm = Direction cosine matrix (A00, A01, A02, A10, A11, A12, A20, A21, A22)
Eax = Euler angles x-convention (theta, phi, psi)
Eay = Euler angles y-convention (theta, phi, chi)
Qtn = Quaternions (q0, q1, q2, q3)
Ypr = xyz- or ypr- or Tait-Bryan convention (yaw, pitch, roll): rotate on z, then
    y, then x
*/

void Sph_Cart2Sc(const double *Cart,double *Sc);
void Sph_Sc2Cart(const double *Sc,double *Cart);
void Sph_Cart2Cart(const double *Cart1,double *Cart2);
void Sph_Ypr2Ypr(const double *Ypr1,double *Ypr2);
void Sph_Eax2Ypr(const double *Eax,double *Ypr);

void Sph_Eax2Dcm(const double *Eax,double *Dcm);
void Sph_Eay2Dcm(const double *Eay,double *Dcm);
void Sph_Ypr2Dcm(const double *Ypr,double *Dcm);
void Sph_Ypr2Dcmt(const double *Ypr,double *Dcmt);
void Sph_Dcm2Ypr(const double *Dcm,double *Ypr);
void Sph_Dcm2Dcm(const double *Dcm1,double *Dcm2);
void Sph_Dcm2Dcmt(const double *Dcm1,double *Dcm2);

void Sph_DcmxDcm(const double *Dcm1,const double *Dcm2,double *Dcm3);
void Sph_DcmxDcmt(const double *Dcm1,const double *Dcmt,double *Dcm3);
void Sph_DcmtxDcm(const double *Dcmt,const double *Dcm2,double *Dcm3);
void Sph_DcmxCart(const double *Dcm,const double *Cart,double *Cart2);
void Sph_DcmtxCart(const double *Dcm,const double *Cart,double *Cart2);

void Sph_One2Dcm(double *Dcm);
void Sph_Ypr2Yprr(const double *Ypr,double *Yprr);
void Sph_Dcm2Dcmr(const double *Dcm,double *Dcmr);
void Sph_Rot2Dcm(char axis,double angle,double *Dcm);
void Sph_Newz2Dcm(const double *Newz,double psi,double *Dcm);

void Sph_DcmtxUnit(const double *Dcmt,char unit,double *vect,const double *add,
    double mult);

void Sph_One2Qtn(double *Qtn);
void Sph_Qtn2Qtn(const double *Qtn1,double *Qtn2);
```

```
50 void Sph_Ypr2Qtn(const double *Ypr,double *Qtn);
51 void Sph_Ypr2Qtni(const double *Ypr,double *Qtni);
52 void Sph_Qtn2Ypr(const double *Qtn,double *Ypr);
53 void Sph_Dcm2Qtn(const double *Dcm,double *Qtn);
54 void Sph_Qtn2Dcm(const double *Qtn,double *Dcm);
55 void Sph_XZ2Qtni(const double *x,const double *z,double *Qtn);
56 void Sph_QtnxQtn(const double *Qtn1,const double *Qtn2,double *Qtn3);
57 void Sph_QtnixQtn(const double *Qtn1,const double *Qtn2,double *Qtn3);
58 void Sph_QtnxQtni(const double *Qtn1,const double *Qtn2,double *Qtn3);
59 void Sph_QtnRotate(const double *Qtn,const double *Cart,double *Cart2);
60 void Sph_QtniRotate(const double *Qtn,const double *Cart,double *Cart2);
61 void Sph_QtniRotateUnitx(const double *Qtni,double *vect,const double *add,double
       mult);
62 void Sph_QtniRotateUnitz(const double *Qtni,double *vect,const double *add,double
       mult);
63
64 double Sph_RotateVectWithNormals3D(const double *pt1,const double *pt2,double *
       newpt2,double *oldnorm,double *newnorm,int sign);
65 void Sph_RotateVectorAxisAngle(const double *vect,const double *axis,double angle,
       double *rotated);
66
67 #endif
```

2 Description

This is a collection of routines for manipulating rotational coordinates using a variety of conventions. Note that some coordinates are for vectors (e.g. spherical coordinates) whereas others are for transformations (e.g. Euler angles). Some of the math here is described in Goldstein (see Appendix B, pp. 606-610, and section 4.4, pp. 143-148), Wikipedia ("Rotation matrix" and "Euler Angles") and Wolfram MathWorld ("Rotation Matrix"). Most rotations are passive, meaning that the rotation matrix times a vector leads to a rotated coordinate system, while the vector is unchanged. If you want to leave the coordinate system in place and rotate the vector instead, use the transposed direction cosine matrix.

If two different function arguments are the same size, such as two vectors or two matrices, then they are always allowed to point to the same memory. For example to invert the direction cosine matrix dcm in-place, the function call is `Sph_Dcm2Dcmt(dcm,dcm)`. While input angles are never required to be clamped to fixed domains, the output angle ranges are usually clamped, perhaps as listed below but usually based on whatever the inverse trig functions return. Input vectors and direction cosine matrices are assumed to be valid and are not checked.

Cartesian coordinates (Cart)
Vector is $[x, y, z]$, all of which are on $(-\infty, \infty)$.

Spherical coordinates (Sc)
Vector is $[r, \theta, \phi]$. r is on $[0, \infty)$, θ is on $[0, \pi]$, and ϕ is on $(-\pi, \pi]$.

Direction cosine matrix (Dcm)
Matrix is given as a 9 element array, which lists the matrix row by row. This is useful for all coordinate transformations and is not associated with any particular convention.

Direction cosine matrix transpose (Dcmt)
This is entered as a normal, non-transposed, direction cosine matrix. However, it is interpreted as a transposed direction cosine matrix in the code.

Euler angle x-convention (Eax)
Vector is $[\theta, \phi, \psi]$. ϕ is on $[0, 2\pi)$, θ is on $[0, \pi]$, ψ is on $[0, 2\pi)$. $\boldsymbol{A} = \boldsymbol{Z}(\psi)\boldsymbol{X}(\theta)\boldsymbol{Z}(\phi)$.

Euler angle y-convention (Eay)
Vector is $[\theta, \phi, \chi]$. ϕ is on $[0, 2\pi)$, θ is on $[0, \pi]$, χ is on $[0, 2\pi)$. $\boldsymbol{A} = \boldsymbol{Z}(\chi)\boldsymbol{Y}(\theta)\boldsymbol{Z}(\phi)$.

Quaternions (Qtn)
Vector is $[q_0, q_1, q_2, q_3]$.

Yaw-pitch-roll (Xyz or Ypr)
Vector is $[\phi, \theta, \psi]$. All are on $(-\pi, \pi]$. These angles are sometimes not clamped because multiple rotations are possible. $\boldsymbol{A} = \boldsymbol{X}(\psi)\boldsymbol{Y}(\theta)\boldsymbol{Z}(\phi)$.

3 Dependencies

random2.h

4 History

2/05 Started.

7/05 Documented.

10/24/07 Added `Eax2Dcm`, `Eay2Dcm`, `Newz2Dcm`.

5/55/12 Added `Sph_DcmtxUnit`.

5/28/12 Added `Sph_Xyz2Dcmt`.

8/6/15 Added `Sph_RotateVectWithNormals`.

3/13/24 Added `Sph_Eax2Xyz`.

3/15/24 Added `Sph_DcmxCart`.

3/16/24 Rewrote docs in LaTeX.

10/5/24 Renamed Xyz to Ypr. Added math text. Removed Euler parameters. Added quaternions.

5 Math

The following text was copied from SmoldynCodeDoc, and then edited. See also Goldstein Chapter 4 (page 128) and Appendix B (page 608), Andrews, 2004 (rotational averaging paper), and Andrews, 2014 (filament paper).

Yaw-pitch-roll angles are described by the Tait-Bryan convention:

name	symbol	perpendicular axis	positive direction	positive unit vector
yaw	ϕ	z	turn left	$\hat{z}$
pitch	θ	y	turn down	$\hat{y}$
roll	ψ	x	tilt right	$\hat{x}$

Note that pitch is defined so that positive pitch is rotation downward. This is the opposite of many conventions, but is used here so that the rotation vector is along the positive $\hat{y}$ vector while using a right-handed coordinate system.

Rotation can be given with a direction cosine matrix (dcm), $\mathbf{\Phi}$, which can be expanded as

$$\mathbf{\Phi} = \begin{bmatrix} \Phi_{Xx} & \Phi_{Xy} & \Phi_{Xz} \\ \Phi_{Yx} & \Phi_{Yy} & \Phi_{Yz} \\ \Phi_{Zx} & \Phi_{Zy} & \Phi_{Zz} \end{bmatrix} \tag{1}$$

This dcm can convert between two frames of reference for a static object, called passive rotation, or can rotate an object in a fixed reference frame, called active rotation. Here, X, Y, and Z are the unit vectors of the "lab frame", meaning the absolute coordinates of the simulation, and x, y, and z are the unit vectors of the "molecule frame". The dcm expresses the dot products of these unit vectors, so each column gives the lab frame coordinates of each unit vector of the molecule frame, and each row gives the molecule frame coordinates of each unit vector of the lab frame. It is unitary, meaning that all its eigenvalues equal 1, and its transpose is its inverse, so $\mathbf{\Phi}^T\mathbf{\Phi} = \mathbf{1}$, where $\mathbf{1}$ is the identity matrix.

The dcm for a product of yaw, pitch, and roll transformations is (Goldstein pp. 135 and 609)

$$\begin{aligned} \mathbf{\Phi} &= \boldsymbol{X}(\psi)\boldsymbol{Y}(\theta)\boldsymbol{Z}(\phi) \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \mathrm{c}\psi & \mathrm{s}\psi \\ 0 & -\mathrm{s}\psi & \mathrm{c}\psi \end{bmatrix} \begin{bmatrix} \mathrm{c}\theta & 0 & -\mathrm{s}\theta \\ 0 & 1 & 0 \\ \mathrm{s}\theta & 0 & \mathrm{c}\theta \end{bmatrix} \begin{bmatrix} \mathrm{c}\phi & \mathrm{s}\phi & 0 \\ -\mathrm{s}\phi & \mathrm{c}\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \mathrm{c}\theta\mathrm{c}\phi & \mathrm{c}\theta\mathrm{s}\phi & -\mathrm{s}\theta \\ \mathrm{s}\psi\mathrm{s}\theta\mathrm{c}\phi - \mathrm{c}\psi\mathrm{s}\phi & \mathrm{s}\psi\mathrm{s}\theta\mathrm{s}\phi + \mathrm{c}\psi\mathrm{c}\phi & \mathrm{c}\theta\mathrm{s}\psi \\ \mathrm{c}\psi\mathrm{s}\theta\mathrm{c}\phi + \mathrm{s}\psi\mathrm{s}\phi & \mathrm{c}\psi\mathrm{s}\theta\mathrm{s}\phi - \mathrm{s}\psi\mathrm{c}\phi & \mathrm{c}\theta\mathrm{c}\psi \end{bmatrix} \end{aligned} \tag{2}$$

Right-multiplying the direction cosine matrix with a vector that's expressed in the lab frame returns the coordinates of the same vector but now using the molecule frame. For example, suppose $\boldsymbol{r}_L$ is a vector expressed in the lab frame. Then,

$$\boldsymbol{r} = \mathbf{\Phi}\boldsymbol{r}_L \tag{3}$$

gives the same vector but expressed in the molecule frame. (As a particularly simple vector, suppose $\boldsymbol{r}_L = [1, 0, 0]^T$; in the molecule frame, which is rotated counter-clockwise by ϕ from the lab frame, $\boldsymbol{r} = [c\phi, -s\phi, 0]^T$.) This expression can be transposed and/or multiplied on both sides by $\mathbf{\Phi}^T$ to yield

$$\boldsymbol{r} = \mathbf{\Phi}\boldsymbol{r}_L \qquad \boldsymbol{r}_L = \mathbf{\Phi}^T\boldsymbol{r} \tag{4}$$

$$\boldsymbol{r}^T = \boldsymbol{r}_L^T\mathbf{\Phi}^T \qquad \boldsymbol{r}_L^T = \boldsymbol{r}^T\mathbf{\Phi} \tag{5}$$

This library has functions for these multiplications. The transpose of the dcm listed above performs an active rotation of a vector in a constant coordinate system.

Quaternions work in relatively similar ways (see the online article "Rotation Quaternions, and How to Use Them" by Rose, 2015). A quaternion, $\boldsymbol{q}$ has 4 values, (q_0, q_1, q_2, q_3). If it represents a 3D point in space, then $\boldsymbol{q} = (0, x, y, z)$ and if it represents a rotation, then all four values can be non-zero and they have total magnitude of 1. The quaternion $\boldsymbol{q} = (1, 0, 0, 0)$ is the rotation identity quaternion, which produces no rotation.

6 Code documentation

6.1 Typical parameter names

name	meaning
`cf`	$\cos(\phi)$
`cq`	$\cos(\theta)$
`cy`	$\cos(\psi)$ or $\cos(\chi)$
`sf`	$\cos(\phi)$
`sq`	$\cos(\theta)$
`sy`	$\cos(\psi)$ or $\sin(\chi)$

6.2 Internal macros and global variables

`#define PI 3.14159265358979323846`
: π

`double Work[9],Work2[9];`
: Scratch space.

6.3 Externally accessible functions

Vector conversion functions

`void Sph_Cart2Sc(double *Cart,double *Sc);`
: Converts Cartesian coordinates to spherical coordinates.

`void Sph_Sc2Cart(double *Sc,double *Cart);`
: Converts spherical coordinates to Cartesian coordinates.

`void Sph_Cart2Cart(const double *Cart1,double *Cart2);`
: Copies the 3D vector `Cart1` to the 3D vector `Cart2`.

`void Sph_Ypr2Ypr(double *Ypr1,double *Ypr2);`
: Copies yaw-pitch-roll vector (or any other 3D vector) `Ypr1` to `Ypr2`. This does not clamp angles.

`void Sph_Eax2Ypr(double *Eax,double *Ypr);`
: Converts Euler angle x-convention to yaw-pitch-roll vector. Equations are from Sphere.nb.

`void Sph_Eax2Dcm(double *Eax,double *Dcm);`
: Calculates direction cosine matrix from Euler angle x-convention vector. Equations from Wolfram Mathworld and Sphere.nb.

`void Sph_Eay2Dcm(double *Eay,double *Dcm);`
: Calculates direction cosine matrix from Euler angle y-convention vector. Equations from Wolfram Mathworld and Sphere.nb.

`void Sph_Ypr2Dcm(double *Ypr,double *Dcm);`
: Calculates direction cosine matrix from yaw-pitch-roll vector. Equations from Goldstein p. 609 and Sphere.nb. $\boldsymbol{A} = \boldsymbol{X}(\psi)\boldsymbol{Y}(\theta)\boldsymbol{Z}(\phi)$.

`void Sph_Ypr2Dcmt(double *Ypr,double *Dcmt);`
Calculates transposed direction cosine matrix from yaw-pitch-roll vector. This is just `Sph_Ypr2Dcm`, but for a transposed result. Rather than using this function, it's usually best to use the untransposed version in a `...Dcmt...` function.

`void Sph_Dcm2Ypr(double *Dcm,double *Ypr);`
Calculates yaw-pitch-roll vector from a direction cosine matrix. Equations derived from Goldstein p. 609 and from Sphere.nb.

`void Sph_Dcm2Dcm(double *Dcm1,double *Dcm2);`
Copies direction cosine matrix (or any other 9 element vector) `Dcm1` to a new one in `Dcm2`.

`void Sph_Dcm2Dcmt(double *Dcm1,double *Dcm2);`
Transposes direction cosine matrix `Dcm1` to yield matrix inverse in `Dcm2`. $\boldsymbol{A}_2 = \boldsymbol{A}_1^{-1}$.

`void Sph_DcmxDcm(double *Dcm1,double *Dcm2,double *Dcm3);`
Matrix multiplies `Dcm1` by `Dcm2` and returns result in `Dcm3`. Note that the transformation is `Dcm2` first, then `Dcm1`. $\boldsymbol{A}_3 = \boldsymbol{A}_1\boldsymbol{A}_2$.

`void Sph_DcmxDcmt(double *Dcm1,double *Dcmt,double *Dcm3);`
Matrix multiplies `Dcm1` by the transpose of `Dcmt` and returns result in `Dcm3` (`Dcmt` is entered as an untransposed matrix). Essentially, this is a negative rotation of `Dcmt` followed by a positive rotation of `Dcm1`. $\boldsymbol{A}_3 = \boldsymbol{A}_1\boldsymbol{A}_2^{-1}$.

`void Sph_DcmtxDcm(double *Dcmt,double *Dcm2,double *Dcm3);`
Matrix multiplies the transpose of `Dcmt` by `Dcm2` and returns the result in `Dcm3` (`Dcmt` is entered as an untransposed matrix). Essentially, this is a positive rotation of `Dcm2` followed by a negative rotation of `Dcmt`. $\boldsymbol{A}_3 = \boldsymbol{A}_1^{-1}\boldsymbol{A}_2$.

`void Sph_DcmxCart(const double *Dcm,const double *Cart,double *Cart2);`
Multiplies matrix `Dcm` by Cartesian vector `Cart` and returns the result in `Cart2`. This is a passive rotation of the coordinate system for the vector, $\boldsymbol{x}_2 = \boldsymbol{A} \cdot \boldsymbol{x}$. This is just a matrix times a vector, but is specifically for 3D.

`void Sph_One2Dcm(double *Dcm);`
Returns the identity direction cosine matrix. $A_{ij} = \delta_{ij}$.

`void Sph_Ypr2Yprr(double *Ypr,double *Yprr);`
Converts the forward-direction yaw-pitch-roll vector `Ypr` to a relative direction change, but for travel in the reverse direction. For example, suppose an airplane performs the direction change that corresponds to `Ypr`. If it then turns around, with the local z-vector as it was initially, but with both x- and y-vectors reversed (180°yaw), then it needs to execute rotation `Yprr` to retrace its original track. $\boldsymbol{A} = \boldsymbol{Z}^{-1}(\phi)\boldsymbol{Y}(\theta)\boldsymbol{X}(\psi)$. Note that this reverses a relative direction change between two vectors and does not reverse an absolute vector (the airplane traveling west being converted to it traveling east). Equations from Sphere.nb.

`void Sph_Dcm2Dcmr(double *Dcm,double *Dcmr);`
Converts an absolute dcm to a dcm in the reverse direction. This reverses the local x and y directions, while preserving the local z direction. This is unlike `Sph_Ypr2Yprr` in that this is for absolute directions while that one was for relative directions. $\boldsymbol{A}_r = \boldsymbol{Z}(\pi)\boldsymbol{A}$.

`void Sph_Rot2Dcm(char axis,double angle,double *Dcm);`
Returns the direction cosine matrix that corresponds to rotation by angle `angle` about axis `axis`, where this latter parameter is the character `x`, `y`, or `z` (or upper-case). $\boldsymbol{A} = \boldsymbol{X}(a)$ or $\boldsymbol{A} = \boldsymbol{Y}(a)$ or $\boldsymbol{A} = \boldsymbol{Z}(a)$.

`void Sph_Newz2Dcm(double *Newz,double psi,double *Dcm);`
Returns the direction cosine matrix that can be used to rotate the coordinate system such that the original z-axis will line up with the vector `Newz`. The length of `Newz` is irrelevent; it does not need to be normalized. Additional rotation about the new z-axis is entered with `psi`. This works as follows: `Newz` is converted to spherical coordinates θ and ϕ, then the dcm is $\boldsymbol{A} = \boldsymbol{Z}(\psi - \phi)\boldsymbol{X}(\theta)\boldsymbol{Z}(\phi)$, which is transposed to yield the active matrix.

`void Sph_DcmtxUnit(double *Dcmt,char axis,double *vect,double *add,double mult);`
Multiplies the transpose of `Dcmt` (entered as a non-transposed direction cosine matrix) with the unit vector for axis `axis` (entered as `x`, `y`, or `z`, or upper case) and returns the result in the 3-dimensional vector `vect`. This multiplies the result by the scalar `mult`. If `add` is non-`NULL`, this adds `add` to `vect` before returning the result.

Quaternions

`void Sph_One2Qtn(double *Qtn)`
Returns the rotation identity quaternion, which is $\boldsymbol{q} = (1, 0, 0, 0)$.

`void Sph_Qtn2Qtn(const double *Qtn1,double *Qtn2)`
Copies quaternion `Qtn1` to `Qtn2`.

`void Sph_Ypr2Qtn(const double Ypr,double *Qtn)`
Converts Ypr angles to quaternion `Qtn`. This uses the following equation, which is from Rose 2015 but with sign changes because theirs is for active rotation and this is for passive rotation.

$$\boldsymbol{q} = \begin{bmatrix} \mathrm{c}\frac{\psi}{2}\mathrm{c}\frac{\theta}{2}\mathrm{c}\frac{\phi}{2} + \mathrm{s}\frac{\psi}{2}\mathrm{s}\frac{\theta}{2}\mathrm{s}\frac{\phi}{2} \\ -\mathrm{s}\frac{\psi}{2}\mathrm{c}\frac{\theta}{2}\mathrm{c}\frac{\phi}{2} + \mathrm{c}\frac{\psi}{2}\mathrm{s}\frac{\theta}{2}\mathrm{s}\frac{\phi}{2} \\ -\mathrm{c}\frac{\psi}{2}\mathrm{s}\frac{\theta}{2}\mathrm{c}\frac{\phi}{2} - \mathrm{s}\frac{\psi}{2}\mathrm{c}\frac{\theta}{2}\mathrm{s}\frac{\phi}{2} \\ -\mathrm{c}\frac{\psi}{2}\mathrm{c}\frac{\theta}{2}\mathrm{s}\frac{\phi}{2} + \mathrm{s}\frac{\psi}{2}\mathrm{s}\frac{\theta}{2}\mathrm{c}\frac{\phi}{2} \end{bmatrix}$$

`void Sph_Ypr2Qtni(const double Ypr,double *Qtni)`
Converts Ypr angles to the inverse quaternion `Qtn`, but returned as a normal quaternion. This uses the following equation, from Rose, 2015. The signs are the same as in Rose because this is for an inverse passive rotation and theirs is for a forward active rotation.

$$\boldsymbol{q} = \begin{bmatrix} \mathrm{c}\frac{\psi}{2}\mathrm{c}\frac{\theta}{2}\mathrm{c}\frac{\phi}{2} + \mathrm{s}\frac{\psi}{2}\mathrm{s}\frac{\theta}{2}\mathrm{s}\frac{\phi}{2} \\ \mathrm{s}\frac{\psi}{2}\mathrm{c}\frac{\theta}{2}\mathrm{c}\frac{\phi}{2} - \mathrm{c}\frac{\psi}{2}\mathrm{s}\frac{\theta}{2}\mathrm{s}\frac{\phi}{2} \\ \mathrm{c}\frac{\psi}{2}\mathrm{s}\frac{\theta}{2}\mathrm{c}\frac{\phi}{2} + \mathrm{s}\frac{\psi}{2}\mathrm{c}\frac{\theta}{2}\mathrm{s}\frac{\phi}{2} \\ \mathrm{c}\frac{\psi}{2}\mathrm{c}\frac{\theta}{2}\mathrm{s}\frac{\phi}{2} - \mathrm{s}\frac{\psi}{2}\mathrm{s}\frac{\theta}{2}\mathrm{c}\frac{\phi}{2} \end{bmatrix}$$

`void Sph_Qtn2Ypr(const double *Qtn,double *Ypr)`
Converts quaternion `Qtn` to ypr angles in `Ypr`. This uses the following equation, which is from Rose, 2015.

$$\boldsymbol{ypr} = \begin{bmatrix} \mathrm{atan2}(2q_0q_3 + 2q_1q_2, q_0^2 + q_1^2 - q_2^2 - q_3^2) \\ \mathrm{asin}(2q_0q_2 - 2q_1q_3) \\ \mathrm{atan2}(2q_0q_1 + 2q_2q_3, q_0^2 - q_1^2 - q_2^2 + q_3^2) \end{bmatrix}$$

`void Sph_Dcm2Qtn(const double *Dcm,double *Qtn)`

Converts direction cosine matrix `Dcm` to quaternion `Qtn`. This is based on equations from Rose, 2015, but modified. Those equations are slightly slower to compute and also unstable to roundoff error (one can end up taking the square root of a negative number). My equations are as follows. First, compute

$$\begin{aligned} q_0 &= r_{11} + r_{22} + r_{33} \\ q_1 &= r_{11} - r_{22} - r_{33} \\ q_2 &= -r_{11} + r_{22} - r_{33} \\ q_3 &= -r_{11} - r_{22} + r_{33} \end{aligned}$$

Next, determine which of these q values is largest and recompute each one according the following equations:

q_0 largest	q_1 largest	q_2 largest	q_3 largest
$q_0 = 0.5\sqrt{1+q_0}$	$q_1 = 0.5\sqrt{1+q_1}$	$q_2 = 0.5\sqrt{1+q_2}$	$q_3 = 0.5\sqrt{1+q_3}$
$f = 0.25/q_0$	$f = 0.25/q_1$	$f = 0.25/q_2$	$f = 0.25/q_3$
$q_1 = f(r_{32} - r_{23})$	$q_0 = f(r_{32} - r_{23})$	$q_0 = f(r_{13} - r_{31})$	$q_0 = f(r_{21} - r_{12})$
$q_2 = f(r_{13} - r_{31})$	$q_2 = f(r_{12} + r_{21})$	$q_1 = f(r_{12} + r_{21})$	$q_1 = f(r_{13} + r_{31})$
$q_3 = f(r_{21} - r_{12})$	$q_3 = f(r_{13} + r_{31})$	$q_3 = f(r_{23} + r_{32})$	$q_2 = f(r_{23} + r_{32})$

`void Sph_Qtn2Dcm(const double *Qtn,double *Dcm)`

Converts quaternion `Qtn` to direction cosine matrix `Dcm`. This uses the equation, from Rose, 2015,

$$\boldsymbol{Q} = \begin{bmatrix} 1-2q_2^2-2q_3^2 & 2q_1q_2-2q_0q_3 & 2q_1q_3+2q_0q_2 \\ 2q_1q_2+2q_0q_3 & 1-2q_1^2-2q_3^2 & 2q_2q_3-2q_0q_1 \\ 2q_1q_3-2q_0q_2 & 2q_2q_3+2q_0q_1 & 1-2q_1^2-2q_2^2 \end{bmatrix}$$

`void Sph_QtnxQtn(const double Qtn1,const double Qtn2,double *Qtn3)`

Multiplies quaternion `Qtn1` by `Qtn2` to yield `Qtn3`. Equation is from Rose, 2015. For the product $\boldsymbol{t} = \boldsymbol{r}\boldsymbol{s}$,

$$\boldsymbol{t} = \begin{bmatrix} r_0s_0 - r_1s_1 - r_2s_2 - r_3s_3 \\ r_0s_1 + r_1s_0 - r_2s_3 + r_3s_2 \\ r_0s_2 + r_1s_3 + r_2s_0 - r_3s_1 \\ r_0s_3 - r_1s_2 + r_2s_1 + r_3s_0 \end{bmatrix}$$

`void Sph_QtnixQtn(const double Qtn1,const double Qtn2,double *Qtn3)`

Multiplies the inverse of quaternion `Qtn1` (entered in its non-inverse version) by `Qtn2` to yield `Qtn3`. For the product $\boldsymbol{t} = \boldsymbol{r}^{-1}\boldsymbol{s}$,

$$\boldsymbol{t} = \begin{bmatrix} r_0s_0 + r_1s_1 + r_2s_2 + r_3s_3 \\ r_0s_1 - r_1s_0 + r_2s_3 - r_3s_2 \\ r_0s_2 - r_1s_3 - r_2s_0 + r_3s_1 \\ r_0s_3 + r_1s_2 - r_2s_1 - r_3s_0 \end{bmatrix}$$

`void Sph_QtnxQtni(const double Qtn1,const double Qtn2,double *Qtn3)`

Multiplies quaternion `Qtn1` by the inverse of `Qtn2` (entered in its non-inverse version) to yield `Qtn3`. For the product $\boldsymbol{t} = \boldsymbol{r}^{-1}\boldsymbol{s}$,

$$\boldsymbol{t} = \begin{bmatrix} r_0s_0 + r_1s_1 + r_2s_2 + r_3s_3 \\ -r_0s_1 + r_1s_0 + r_2s_3 - r_3s_2 \\ -r_0s_2 - r_1s_3 + r_2s_0 + r_3s_1 \\ -r_0s_3 + r_1s_2 - r_2s_1 + r_3s_0 \end{bmatrix}$$

`void Sph_QtnRotate(const double *Qtn,const double *Cart,double *Cart2);`
Rotates 3D Cartesian vector `Cart` using quaternion `Qtn`, returning the answer in 3D vector `Cart2`. This computes the product

$$\boldsymbol{c}_2 = \boldsymbol{q}^* \boldsymbol{c} \boldsymbol{q}$$

where $\boldsymbol{c}_2$ is `Cart2`, $\boldsymbol{q}$ is `Qtn`, and $\boldsymbol{c}$ is `Cart`. The math is derived in the Mathematica document SphereQuat.nb. This equation performs passive rotation from the $\boldsymbol{c}$ frame to the $\boldsymbol{c}_2$ frame, where the rotation angles represent the rotation of $\boldsymbol{c}_2$ from $\boldsymbol{c}$ (e.g. $\boldsymbol{c}$ is the lab frame and $\boldsymbol{c}_2$ is the molecule frame).

`void Sph_QtniRotate(const double *Qtn,const double *Cart,double *Cart2);`
Performs the inverse rotation of 3D Cartesian vector `Cart` using quaternion `Qtn`, returning the answer in 3D vector `Cart2`. This computes the product

$$\boldsymbol{c}_2 = \boldsymbol{q} \boldsymbol{c} \boldsymbol{q}^*$$

where $\boldsymbol{c}_2$ is `Cart2`, $\boldsymbol{q}$ is `Qtn`, and $\boldsymbol{c}$ is `Cart`. The math is derived in the Mathematica document SphereQuat.nb. See the discussion for `Sph_QtnRotate`.

`void Sph_QtniRotateUnitx(const double *Qtni, double *vect, const double *add, double mult)`
Rotates the $\hat{\boldsymbol{x}}$ unit vector with the inverse quaternion `Qtni`, which is entered as a non-inverted quaternion, and then multiplies the result by `mult` and adds it to `add`, returning the answer in `vect`. The `add` vector is required and cannot be the same as `vect`. Using $\boldsymbol{a}$ for `add`, m for `mult`, $\boldsymbol{v}$ for `vect`, and $\boldsymbol{q}$ for `Qtni`, the equation for this is

$$\boldsymbol{v} = \boldsymbol{a} + m \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 \\ 2q_1q_2 - 2q_0q_3 \\ 2q_0q_2 + 2q_1q_3 \end{bmatrix}$$

This is derived in the Mathematica document SphereQuat.nb.

`void Sph_QtniRotateUnitz(const double *Qtni, double *vect, const double *add, double mult)`
Rotates the $\hat{\boldsymbol{z}}$ unit vector with the inverse quaternion `Qtni`, which is entered as a non-inverted quaternion, and then multiplies the result by `mult` and adds it to `add`, returning the answer in `vect`. The `add` vector is required and cannot be the same as `vect`. Using $\boldsymbol{a}$ for `add`, m for `mult`, $\boldsymbol{v}$ for `vect`, and $\boldsymbol{q}$ for `Qtni`, the equation for this is

$$\boldsymbol{v} = \boldsymbol{a} + m \begin{bmatrix} -2q_0q_2 + 2q_1q_3 \\ 2q_0q_1 + 2q_2q_3 \\ q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}$$

This is derived in the Mathematica document SphereQuat.nb.

More rotations

`void Sph_RotateAxisAngle(const double *pt1,const double *pt2,double theta,const double *oldvect,double *newvect)`
Text.

`double Sph_RotateVectWithNormals3D(double *pt1, double *pt2, double *newpt2, double *oldnorm, double *newnorm, int sign);`

This is for the case where the line from `pt1` to `pt2` is in the plane that has normal `oldnorm`, and then the plane is rotated about point `pt1` to so that its normal becomes `newnorm`. This function calculates the new value for `pt2`, returned in `newpt2`. `newpt2` and `pt2` are allowed to point to the same memory. Both `oldnorm` and `newnorm` need to have unit length. This returns the cosine of the angle between the two normals, which is also the dot product of the two normal vectors. If this cosine is 1, then the two normals are parallel to each other and `newpt2` is set equal to `pt2` because no rotation takes place. If this cosine is -1, then the two normals are anti-parallel to each other, in which case the problem is ill-determined because the rotation axis cannot be determined; if that's the case, then this function assumes that the rotation axis is perpendicular to the vector from `pt1` to `pt2`, with the result that the new vector is in the opposite direction as the original vector. New function Sept. 2015.

The `sign` input is here to allow the normals to internally inconsistent. That is, it is good practice for all normals to points toward the same face of a surface, such as the outside or inside. If this is the case, then enter `sign` as 0. However, if this is not done, then enter `sign` as 1 if the total rotation should be less than 90°and as -1 if the total rotation should be more than 90°.

It is permitted to enter `oldnorm` as `NULL`. In this case, the vector is rotated around a random rotation axis that is perpendicular to `newnorm`. In other words, `newpt2` is still placed in the new plane and it is still the correct distance from `pt1`, but the rotation direction to this new position is random.

The math is as follows. Define $\boldsymbol{p}_1$ as `pt1`, $\boldsymbol{p}_2$ as `pt2`, $\boldsymbol{o}$ as `oldnorm`, and $\boldsymbol{n}$ as `newnorm`. Also, define $\boldsymbol{p}$ as the vector from $\boldsymbol{p}_1$ to $\boldsymbol{p}_2$, meaning that $\boldsymbol{p} = \boldsymbol{p}_2 - \boldsymbol{p}_1$. Also define $\boldsymbol{a}$ as the unit vector for the axis about which the rotation takes place; it is the line that is shared by the old plane and the new plane. Define θ as the rotation angle about this axis. These values are

$$\boldsymbol{a} = \frac{\boldsymbol{o} \times \boldsymbol{n}}{\sqrt{(\boldsymbol{o} \times \boldsymbol{n}) \cdot (\boldsymbol{o} \times \boldsymbol{n})}}$$

$$\cos\theta = \boldsymbol{o} \cdot \boldsymbol{n}$$

The θ equation relies on the requirement that $\boldsymbol{o}$ and $\boldsymbol{n}$ have unit length. The direction cosine matrix for rotation by angle θ about axis $\boldsymbol{a}$ is (from Wikipedia "Rotation matrix")

$$\begin{bmatrix} c\theta + a_x^2(1-c\theta) & a_x a_y(1-c\theta) - a_z s\theta & a_x a_z(1-c\theta) + a_y s\theta \\ a_y a_x(1-c\theta) + a_z s\theta & c\theta + a_y^2(1-c\theta) & a_y z_z(1-c\theta) - a_x s\theta \\ a_z a_x(1-c\theta) - a_y s\theta & a_z a_y(1-c\theta) + a_x s\theta & c\theta + a_z^2(1-c\theta) \end{bmatrix}$$

`void Sph_RotateVectorAxisAngle(const double *vect, const double *axis, double angle, double *rotated)`

Rotates vector `vect` about axis `axis` by angle `angle`, returning the answer in `rotated`. This uses Rodrigues's rotation formula. I didn't write this function, but told ChatGPT to write it for me, and then I edited the result.