Documentation for Sphere.h and Sphere.c

Steve Andrews

2005 - 2024

1 Header file: Sphere.h

```c
/* 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)
Ep = Euler parameters (e0, e1, e2, e3)
Xyz = 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\_Eay2Ep(const double *Eay,double *Ep);
void Sph\_Xyz2Xyz(const double *Xyz1,double *Xyz2);
void Sph\_Eax2Xyz(const double *Eax,double *Xyz);

void Sph\_Eax2Dcm(const double *Eax,double *Dcm);
void Sph\_Eay2Dcm(const double *Eay,double *Dcm);
void Sph\_Xyz2Dcm(const double *Xyz,double *Dcm);
void Sph\_Xyz2Dcmt(const double *Xyz,double *Dcmt);
void Sph\_Dcm2Xyz(const double *Dcm,double *Xyz);
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\_One2Dcm(double *Dcm);
void Sph\_Xyz2Xyzr(const double *Xyz,double *Xyzr);
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);

double Sph\_RotateVectWithNormals3D(const double *pt1,const double *pt2,double *
    newpt2,double *oldnorm,double *newnorm,int sign);

```

```
#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 $[0, 2\pi)$. $\boldsymbol{A} = \boldsymbol{Z}(\chi)\boldsymbol{Y}(\theta)\boldsymbol{Z}(\phi)$.

Euler parameters (Eap)
Vector is $[e_0, e_1, e_2, e_3]$.

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.

5 Math

This library performs passive rotations, meaning that the coordinate system rotates while the vector stays fixed. This is the standard textbook method, such as in Goldstein. Elementary rotation matrices are:

$$\boldsymbol{X}(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \mathrm{c}\psi & \mathrm{s}\psi \\ 0 & -\mathrm{s}\psi & \mathrm{c}\psi \end{bmatrix}$$

$$\boldsymbol{Y}(\theta) = \begin{bmatrix} \mathrm{c}\theta & 0 & -\mathrm{s}\theta \\ 0 & 1 & 0 \\ \mathrm{s}\theta & 0 & \mathrm{c}\theta \end{bmatrix}$$

$$\boldsymbol{Z}(\phi) = \begin{bmatrix} \mathrm{c}\phi & \mathrm{s}\phi & 0 \\ -\mathrm{s}\phi & \mathrm{c}\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

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` hfill
   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_Eay2Ep(double *Eay,double *Ep);`
   Converts Euler angle y-convention transformation to Euler parameters. Equations from Goldstein p. 608.

`void Sph_Xyz2Xyz(double *Xyz1,double *Xyz2);`
   Copies yaw-pitch-roll vector (or any other 3D vector) `Xyz1` to `Xyz2`. This does not clamp angles.

`void Sph_Eax2Xyz(double *Eax,double *Xyz);`
   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_Xyz2Dcm(double *Xyz,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_Xyz2Dcmt(double *Xyz,double *Dcmt);`
   Calculates transposed direction cosine matrix from yaw-pitch-roll vector. This is just `Sph_Xyz2Dcm`, 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_Dcm2Xyz(double *Dcm,double *Xyz);`
   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_Xyz2Xyzr(double *Xyz,double *Xyzr);`
: Converts the forward-direction yaw-pitch-roll vector `Xyz` to a relative direction change, but for travel in the reverse direction. For example, suppose an airplane performs the direction change that corresponds to `Xyz`. 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 `Xyzr` 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_Xyz2Xyzr` 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.

`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")

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