Documentation for Geometry.h and Geometry.c

Steve Andrews

2006 - 2023

1 Header file: Geometry.h

```
/* Steven Andrews 2/17/06
See documentation called Geometry_doc.doc.
Copyright 2006-2007 by Steven Andrews.  This work is distributed under the terms
of the Gnu Lesser General Public License (LGPL). */


#ifndef __Geometry_h
#define __Geometry_h

// Center
void Geo_LineCenter(double **point,double *cent,int dim);
void Geo_RectCenter(double **point,double *cent,int dim);
void Geo_TriCenter(double **point,double *cent,int dim);

// Normal
double Geo_LineNormal(double *pt1,double *pt2,double *ans);
double Geo_LineNormal2D(double *pt1,double *pt2,double *point,double *ans);
double Geo_LineNormal3D(double *pt1,double *pt2,double *point,double *ans);
double Geo_LineNormPos(double *pt1,double *pt2,double *point,int dim,double *
    distptr);
double Geo_TriNormal(double *pt1,double *pt2,double *pt3,double *ans);
double Geo_SphereNormal(double *cent,double *pt,int front,int dim,double *ans);

// Unit Vectors
double Geo_UnitCross(double *v1start,double *v1end,double *v2start,double *v2end,
    double *ans);
double Geo_TriUnitVects(double *pt1,double *pt2,double *pt3,double *unit0,double *
    unit1,double *unit2);
double Geo_SphereUnitVects(double *cent,double *top,double *point,int front,double
     *unit0,double *unit1,double *unit2);
double Geo_CylUnitVects(double *pt1,double *pt2,double *point,double *unit0,double
     *unit1,double *unit2);
double Geo_DiskUnitVects(double *cent,double *front,double *point,double *unit0,
    double *unit1,double *unit2);

// Area
double Geo_LineLength(double *pt1,double *pt2,int dim);
double Geo_TriArea2(double *pt1,double *pt2,double *pt3);
double Geo_TriArea3D(double *pt1,double *pt2,double *pt3);
double Geo_TriArea3(double *pt1,double *pt2,double *pt3,double *norm);
double Geo_QuadArea(double *pt1,double *pt2,double *pt3,double *pt4,int dim);

// Inside point
double Geo_InsidePoints2(double *pt1,double *pt2,double margin,double *ans1,double
     *ans2,int dim);
void Geo_InsidePoints3(double *pt1,double *pt2,double *pt3,double margin,double *
    ans1,double *ans2,double *ans3);
void Geo_InsidePoints32(double **point,double margin,double **ans);

// Point in
```

```
int Geo_PtInTriangle(double *pt1,double *pt2,double *pt3,double *norm,double *test
    );
int Geo_PtInTriangle2(double **point,double *test);
int Geo_PtInSlab(double *pt1,double *pt2,double *test,int dim);
int Geo_PtInSphere(double *test,double *cent,double rad,int dim);

// Nearest
void Geo_NearestSlabPt(double *pt1,double *pt2,double *point,double *ans,int dim);
int Geo_NearestLineSegPt(double *pt1,double *pt2,double *point,double *ans,int dim
    ,double margin);
void Geo_NearestTriPt(double *pt1,double *pt2,double *pt3,double *norm,double *
    point,double *ans);
void Geo_NearestTriPt2(double **point,double **edgenorm,double *norm,double *
    testpt,double *ans);
int Geo_NearestTrianglePt(double *pt1,double *pt2,double *pt3,double *norm,double
    *point,double *ans);
int Geo_NearestTrianglePt2(double **point,double *norm,double *testpt,double *ans,
    double margin);
double Geo_NearestSpherePt(double *cent,double rad,int front,int dim,double *point
    ,double *ans);
void Geo_NearestRingPt(double *cent,double *axis,double rad,int dim,double *point,
    double *ans);
void Geo_NearestCylPt(double *pt1,double *axis,double rad,int dim,double *point,
    double *ans);
int Geo_NearestCylinderPt(double *pt1,double *pt2,double rad,int dim,double *point
    ,double *ans,double margin);
int Geo_NearestDiskPt(double *cent,double *axis,double rad,int dim,double *point,
    double *ans,double margin);
double Geo_NearestLine2LineDist(double *ptA1,double *ptA2,double *ptB1,double *
    ptB2);
double Geo_NearestSeg2SegDist(double *ptA1,double *ptA2,double *ptB1,double *ptB2)
    ;
double Geo_NearestAabbPt(const double *bpt1,const double *bpt2,int dim,const
    double *point,double *ans);

// To Rect
void Geo_Semic2Rect(double *cent,double rad,double *outvect,double *r1,double *r2,
    double *r3);
void Geo_Hemis2Rect(double *cent,double rad,double *outvect,double *r1,double *r2,
    double *r3,double *r4);
void Geo_Cyl2Rect(double *pt1,double *pt2,double rad,double *r1,double *r2,double
    *r3,double *r4);

// Cross
double Geo_LineXLine(double *l1p1,double *l1p2,double *l2p1,double *l2p2,double *
    crss2ptr);
double Geo_LineXPlane(double *pt1,double *pt2,double *v,double *norm,double *
    crsspt);
double Geo_LineXSphs(double *pt1,double *pt2,double *cent,double rad,int dim,
    double *crss2ptr,double *nrdistptr,double *nrposptr);
double Geo_LineXCyl2s(double *pt1,double *pt2,double *cp1,double *cp2,double *norm
    ,double rad,double *crss2ptr,double *nrdistptr,double *nrposptr);
double Geo_LineXCyls(double *pt1,double *pt2,double *cp1,double *cp2,double rad,
    double *crss2ptr,double *nrdistptr,double *nrposptr);

// Reflection
double Geo_SphereReflectSphere(const double *a0,const double *a1,const double *b0,
    const double *b1,int dim,double radius2,double *a1p,double *b1p);

```

```
// Exit
double Geo_LineExitRect(double *pt1,double *pt2,double *front,double *corner1,
    double *corner3,double *exitpt,int *exitside);
double Geo_LineExitLine2(double *pt1,double *pt2,double *end1,double *end2,double
    *exitpt,int *exitend);
void Geo_LineExitArc2(double *pt1,double *pt2,double *cent,double radius,double *
    norm,double *exitpt,int *exitend);
double Geo_LineExitTriangle(double *pt1,double *pt2,double *normal,double *v1,
    double *v2,double *v3,double *exitpt,int *exitside);
double Geo_LineExitTriangle2(double *pt1,double *pt2,double **point,double *exitpt
    ,int *exitside);
double Geo_LineExitSphere(double *pt1,double *pt2,double *cent,double rad,double *
    exitpt);
void Geo_LineExitHemisphere(double *pt1,double *pt2,double *cent,double rad,double
     *normal,double *exitpt);
double Geo_LineExitCylinder(double *pt1,double *pt2,double *end1,double *end2,
    double rad,double *exitpt,int *exitend);

// Cross aabbb
int Geo_LineXaabb2(double *pt1,double *pt2,double *norm,double *bpt1,double *bpt2)
    ;
int Geo_LineXaabb(double *pt1,double *pt2,double *bpt1,double *bpt2,int dim,int
    infline);
int Geo_TriXaabb3(double *pt1,double *pt2,double *pt3,double *norm,double *bpt1,
    double *bpt2);
int Geo_RectXaabb2(double *r1,double *r2,double *r3,double *bpt1,double *bpt2);
int Geo_RectXaabb3(double *r1,double *r2,double *r3,double *r4,double *bpt1,double
     *bpt2);
int Geo_CircleXaabb2(double *cent,double rad,double *bpt1,double *bpt2);
int Geo_SphsXaabb3(double *cent,double rad,double *bpt1,double *bpt2);
int Geo_CylisXaabb3(double *pt1,double *pt2,double rad,double *bpt1,double *bpt2);
int Geo_DiskXaabb3(double *cent,double rad,double *norm,double *bpt1,double *bpt2)
    ;

// Approx. cross aabb
int Geo_SemicXaabb2(double *cent,double rad,double *outvect,double *bpt1,double *
    bpt2);
int Geo_HemisXaabb3(double *cent,double rad,double *outvect,double *bpt1,double *
    bpt2);
int Geo_CylsXaabb3(double *pt1,double *pt2,double rad,double *bpt1,double *bpt2);

// Volumes
double Geo_SphVolume(double rad,int dim);
double Geo_SphOLSph(double *cent1,double *cent2,double r1,double r2,int dim);

#endif
```

2 Description

These functions do a variety of things that are useful for 1-D, 2-D, and 3-D geometry manipulations. A few functions do n-D geometry, but those are rare. Its sole current use is in Smoldyn.

Functions do not change input data arrays. Output arrays are written to but never read from. It is often permissible to use the same input array for multiple inputs, but every output array needs to be distinct from each other, and from each input array.

In general, functions include all boundaries as part of the region when testing whether a point is in a region or not, or whether two regions overlap. An axis-aligned bounding box, called an aabb,

has its low corner $(x_{min}, y_{min}, z_{min})$ at `bpt1` and its high corner $(x_{max}, y_{max}, z_{max})$ at `bpt2`.

3 Dependencies

Geometry.h, math2.h

4 History

2/06 Started.

12/06-2/07 Major rewrite and additions; included in this was a complete switch from floats to doubles.

9/3/07 Added Center functions.

10/31/07 Added some nearest functions.

1/12/09 Added some area functions.

3/2/09 Added several nearest functions.

3/22/10 Added `Geo_LineXaabb` and fixed a bug in `Geo_CylisXaabb3`.

3/8/11 Fixed `Geo_NearestTriPt` and `Geo_NearestTrianglePt`, in which they didn't test nearest corner points correctly.

6/24/11 Added `Geo_InsidePoints2` and `Geo_InsidePoints3`.

4/11/12 Added `Geo_TriArea3D`.

12/15/12 Ye Li added `Geo_Area3D` to library. I improved it to use a more stable algorithm.

7/17/12 Added unit vector functions: `Geo_TriUnitVects`, `Geo_SphereUnitVects`, `Geo_CylUnitVects`, and `Geo_DiskUnitVects`.

5/10/13 Fixed a bug in `Geo_Cyl2Rect`.

7/20/15 Improved documentation.

8/15] Wrote `Geo_LineExit`, etc. functions

9-10/15 Added `EPSILON`, added functions that use edge normals

1/8/16 Fixed a bug in `Geo_NearestTrianglePt2`

3/10/16 Replaced `EPSILON` with margin input in some cases

3/18/16 Added `Geo_SphereReflectSphere`.

4/11/16 Added `Geo_NearestAabbPt`

7/19/19 Minor bug fix in `Geo_CylisXaabb3`

8/16/23 Converted documentation to LaTeX.

5 Bugs

Several of the functions that check crossings of 3-D objects with others (aabbs in particular) ignore potential separation planes, so they report crossings when there aren't any. These need to be fixed. As it is, these functions can return false positives (they report non-existent crossings) but they never return false negatives (they never ignore a crossing that actually exists).

Note that most functions do not work correctly with values that are close to `DBL_MAX` because they can't do math with these numbers without running into errors.

6 Math

Several functions use the cross-product of two vectors. As a reminder, $\mathbf{c} = \mathbf{a} \times \mathbf{b}$ is:

$$c_x = a_y b_z - a_z b_y$$
$$c_y = a_z b_x - a_x b_z$$
$$c_z = a_x b_y - a_y b_x$$

The functions that test whether two objects intersect often make use of the separating axis theorem. However, they also often use my own methods. Here is my understanding of the separating axis theorem. Consider two convex polygons, in 2-D. A line segment is included as well, where this can be seen as a two-sided polygon with 0 area. If the objects do not cross, then there must a be at least one infinite line that separates them. One of these lines will be parallel and adjacent to one of the sides of one of the polygons. To check for crossing, (1) choose a polygon edge on object A, (2) find its outward normal, which does not have to normalized, (3) project a vertex of the test edge onto this normal by taking the dot product of the vertex and the normal vector, (4) project all vertices of polygon B onto this normal in the same way, (5) the objects do not cross if all projected values of object B are larger than the projected value of the test edge. Repeat for all edges of both objects; if all projections fail, then the objects must cross. If two edges are parallel, they will have opposite normal vectors and some steps can be saved. There are faster methods, but the one listed should work.

In three dimensions, if two convex polyhedra do not cross, then there must be an infinite plane that separates them. Before, I thought that non-crossing polyhedra necessarily implied a separating plane that is parallel to one of the faces on one of the objects. Now I see that that is sufficient to prove separation, but that other potential separating planes need to be checked as well. Other planes to test are those that are parallel to edges, including one edge from A and one vertex from B and vice versa.

7 Functions for external use

Center functions

`void Geo_LineCenter(double **point, double *cent,int dim);`

> Returns the center of the line for which the two ends are defined as `point[0][`*coordinate*`]` and `point[1][`*coordinate*`]`. This works for all dimensionalities and simply returns the mean of the `point[0]` and `point[1]` vectors.

`void Geo_RectCenter(double **point, double *cent,int dim);`

Returns the center coordinates of a rectangle. The rectangle corners are defined by `point[`*corner*`][`*coordinate*`]` and the result is returned in `cent`. `dim` is the dimensionality of space, not of the surface. If `dim` is 1, then the "rectangle" is really a point with only one "corner" and one coordinate; if `dim` is 2 then the rectangle is really a line with 2 corners (the two ends) and two coordinates each; if `dim` is 3 then the rectangle is a genuine rectangle with 4 corners that have three coordinates each. Results are undefined for other `dim` values.

`void Geo_TriCenter(double **point, double *cent,int dim);`

This returns the center coordinates of a triangle, where the triangle vertices are defined by `point[`*vertex*`][`*coordinate*`]` and the result is returned in `cent`. `dim` is the dimensionality of space, not of the surface. If `dim` is 1, then the "triangle" is really a point with only one vertex and one coordinate; if `dim` is 2 then the triangle is really a line with two vertices (the two ends) and two coordinates each; and if `dim` is 3 then the triangle is a genuine triangle with 3 vertices that have 3 coordinates each. Results are undefined for other `dim` values.

Normal functions

`double Geo_LineNormal(double *pt1, double *pt2, double *ans);`

Finds the 2-D unit normal for line segment that goes from `pt1` to `pt2` (all 2-D) and puts it in `ans`. The result vector is perpendicular to the line segment and points to the right, for travel from `pt1` to `pt2`. If `pt1` and `pt2` are equal, the unit normal points towards the positive x-axis. Returns the length of the line segment from `pt1` to `pt2`.

`double Geo_LineNormal2D(double *pt1, double *pt2, double *point, double *ans);`

Identical to `Geo_LineNormal3D`, except that this is for a 2-D system. The returned vector is either identical to or the negative of that which is returned by `Geo_LineNormal`. This returns the perpendicular distance between the line and the point, and can handle multiple points being equal to each other.

`double Geo_LineNormal3D(double *pt1, double *pt2, double *point, double *ans);`

Finds the 3-D unit normal for line that includes `pt1` and `pt2`, and that includes the point `point`, and puts it in `ans`. The result vector is perpendicular to the line. Returns the perpendicular distance between the line and point. To decrease round-off error, this function calculates the result using `pt1` as a basis point, and then recalculates the result using the new point. If `pt1` and `pt2` are the same, this returns the normalized vector from `pt1` to point. If point is on the line that includes `pt1` and `pt2`, this returns the perpendicular unit vector that is in the x, y-plane and that points to the right of the projection of the line in the x, y-plane, if possible; if not, it returns the unit x-vector which, again, is perpendicular to the line.

`double Geo_LineNormPos(double *pt1, double *pt2, double *point,int dim, double *distptr);`

Given the line that includes points `pt1` and `pt2`, this considers the normal of this line that goes to point. The position of the intersection between the normal and the line is returned, where it is scaled and offset such that `pt1` is at 0 and `pt2` is at 1. The unscaled length of the normal segment, from the intersection to point, is returned in `distptr`, if that pointer is not sent in as `NULL`. `dim` is the dimensionality of the system, which can be any positive value.

`double Geo_TriNormal(double *pt1, double *pt2, double *pt3, double *ans);`

Finds the 3-D unit normal for the triangle that is defined by the 3-D points `pt1`, `pt2`, and `pt3` and puts it in `ans`. If one looks at the triangle backwards along the unit normal, the three points show counterclockwise winding; i.e. the right-hand rule for the points in sequence yields the direction of the unit normal. The triangle area is returned. If the area is zero, `ans` is returned in the x, y-plane. This finds the normal and the area using the cross-product of the first two triangle edges.

`double Geo_SphereNormal(double *cent, double *pt,int front,int dim, double *ans);`

Returns the unit normal vector from the sphere center at cent to the point at `pt`. Enter `front` as 1 for an outward normal and -1 for an inward normal. `dim` is the system dimensionality, which can be any positive integer. The result is returned in `ans` and the distance between `cent` and `pt` is returned by the function.

Unit vector functions

`double Geo_UnitCross(double *v1start, double *v1end, double *v2start, double *v2end, double *ans);`

Computes the cross product between the vector that goes from `v1start` to `v1end` and the vector that goes from `v2start` to `v2end`, then normalizes this cross product to unit length and returns it in `ans`. Everything is assumed to be in 3D. The length of the unnormalized answer vector is returned directly. If this length result is 0, meaning that the two input vectors were parallel, then 0 is returned and the vector in `ans` is set to $(0, 0, 0)$. If either or both of the input vectors starts at the origin, then enter the start value as `NULL`.

`double Geo_TriUnitVects(double *pt1, double *pt2, double *pt3, double *unit0, double *unit1, double *unit2);`

Returns the unit vectors of a 3-dimensional triangle that has its corners at `pt1`, `pt2`, and `pt3`, in vectors `unit0`, `unit1`, and `unit2`. The first is the triangle normal, the second is parallel to the edge from `pt1` to `pt2`, and the third is orthogonal to the previous two, using a right-handed coordinate system. Returns the triangle area. Behavior is undefined if the area equals zero.

`double Geo_SphereUnitVects(double *cent, double *top, double *point,int front, double *unit0, double *unit1, double *unit2);`

Returns the unit vectors of a 3-dimensional sphere that has its center at `cent` and its top (i.e. where $q = 0$, in a spherical coordinate system) at `top`, for the point `point`. Enter `front` as 1 for an outward normal and -1 for an inward normal. Results are returned in `unit0`, `unit1`, and `unit2`. The first unit vector is the local sphere normal, the second points from point towards top, but in the local plane of the sphere at point, and the third is orthogonal to the previous two, using a right-handed coordinate system. Returns the distance from the center to point. Behavior is undefined if the distance from `cent` to `top` is zero, or the distance from `cent` to `point` is zero However, it's ok for `point` and `top` to equal each other.

`double Geo_CylUnitVects(double *pt1, double *pt2, double *point, double *unit0, double *unit1, double *unit2);`

Returns the unit vectors of a 3-dimensional cylinder that has its axis along the line from `pt1` to `pt2`, for the reference point `point`. Results are returned in `unit0`, `unit1`, and `unit2`. The first unit vector is the local cylinder normal, the second is parallel to the cylinder axis, and the third is orthogonal to the previous two using a right- handed coordinate system. Returns the distance from the axis to `point`. Behavior is undefined if this distance, or if the distance from `pt1` to `pt2`, equals zero.

```
double Geo_DiskUnitVects(double *cent, double *front, double *point, double *unit0,
    double *unit1, double *unit2);
```

Returns the unit vectors of a 3-dimensional disk that has its center at `cent` and is oriented perpendicular to the unit vector in `front`, for the reference point `point`. Results are returned in `unit0`, `unit1`, and `unit2`. The first unit vector is simply copied over from front, the second is in the direction from `cent` to `point`, and the third is orthogonal to the previous two using a right-handed coordinate system. Returns the distance from `cent` to `point`. Behavior is undefined if this distance is zero.

Length, area functions

```
double Geo_LineLength(double *pt1, double *pt2,int dim);
```

Returns the length of the line that extends from `pt1` to `pt2`, and its dimensionality in `dim`.

```
double Geo_TriArea2(double *pt1, double *pt2, double *pt3);
```

Returns the area of the 2-D triangle that is defined by points `pt1`, `pt2`, and `pt3`. The returned area will be positive if these are counter-clockwise (right-hand winding rule) and negative if these are clockwise.

```
double Geo_TriArea3D(double *pt1, double *pt2, double *pt3);
```

VCell addition (Ye Li). This calculates the area of a 3-D triangle, defined by the points `pt1`, `pt2`, and `pt3`. Unlike `Geo_TriArea3`, this does not require a unit normal. This always returns a positive value. This uses Heron's formula. I modified Ye's original code to use a numerically stable formula (see Wikipedia Heron's formula).

```
double Geo_TriArea3(double *pt1, double *pt2, double *pt3, double *norm);
```

Returns the area of a 3-dimensional triangle which is defined by the 3-D points `pt1`, `pt2`, and `pt3` and which has unit normal `norm`. The returned area will be positive if `norm` follows the right-hand winding rule, and vice versa. The base is defined as the side from `pt1` to `pt2`; the cross product is found of the base and the unit normal to yield a triangle height vector that has the length of the base and which points away from the triangle; the dot product of this height and the side from `pt1` to `pt3`, with a sign change and divided by 2, is the area.

```
double Geo_QuadArea(double *pt1, double *pt2, double *pt3, double *pt4,int dim);
```

Returns the area of the quadrilateral that is defined by points `pt1`, `pt2`, `pt3`, and `pt4`, in dimension `dim`. Returns 0 if `dim` is not 2 or 3. The right-hand winding rule is used for the sign of the answer, meaning that if the area is positive if the points cycle counterclockwise for increasing point numbers. If `dim` is 3, all points are assumed to be coplanar.

This works by dividing the quadrilateral into two triangles and returning the sum of their areas.

Inside point functions

`double Geo_InsidePoints2(double *pt1, double *pt2, double margin, double *ans1, double *ans2,int dim);`

Takes two points in `pt1` and `pt2`, which are in 1, 2, or 3 dimensions (listed in `dim`), and returns versions of these points, in `ans1` and `ans2`, that are moved together by absolute distance margin on each side. For example, the 1-D points at 0 and 5 are moved to positions 1 and 4 if `margin` is 1. This function returns the separation between `pt1` and `pt2`; if this length is smaller than twice `margin`, then `ans1` will be closer to `pt2` and `ans2` will be closer to `pt1`. It is permitted for `ans1` to equal `pt1` and for `ans2` to equal `pt2`. Enter a negative value for `margin` if you want to move points outwards.

Math: **delta** is the vector from `pt1` to `pt2`, which has length *len*. Dividing **delta** by *len* yields a unit vector, and then multiplying it by *margin* yields a vector with length *margin* that points in the direction from `pt1` to `pt2`. The new points are found by adding **delta** to `pt1` and subtracting **delta** from `pt2`.

`void Geo_InsidePoints3(double *pt1, double *pt2, double *pt3, double margin, double *ans1, double *ans2, double *ans3);`

This takes in a triangle defined by its corners `pt1`, `pt2`, and `pt3` and returns a smaller triangle with corners `ans1`, `ans2`, and `ans3`. This smaller triangle is inscribed inside the original one with distance margin between the original edges and the new edges. All edge lengths of the original triangle must be greater than zero, which is not checked for in this function. Enter `margin` as a negative number for a larger new triangle. It is not permitted for `ans1`, `ans2`, or `ans3` to occupy the same memory as `pt1`, `pt2`, and `pt3`.

Math: A triangle is defined by points $\mathbf{p}_1$, $\mathbf{p}_1$, and $\mathbf{p}_3$ (`pt1`, `pt2`, and `pt3` in the code). The side lengths are l_{12}, l_{23}, and l_{31}. We want to find inside points $\mathbf{s}_1$, $\mathbf{s}_2$, and $\mathbf{s}_3$ (`ans1`, `ans2`, and `ans3` in the code) so that the inner triangle is inscribed inside the original triangle with a margin of size m on all sides. Consider corner 1. The vector that points from $\mathbf{p}_1$ to $\mathbf{s}_1$ bisects the two edges that meet at $\mathbf{p}_1$. Thus, its direction is

$$direction = \frac{\mathbf{p}_2 - \mathbf{p}_1}{l_{12}} - \frac{\mathbf{p}_1 - \mathbf{p}_3}{l_{31}}$$

This equation is more obvious by putting a '+' sign in the middle and reversing the order of the last difference, but this form is easier for converting to other corners. The length of this bisecting vector can be found using two applications of the law of cosines. The inside angle of corner 1 is denoted θ_1 (and likewise for the other corners). From the law of cosines and the original triangle,

$$2l_{12}l_{31}\cos\theta = l_{12}^2 + l_{31}^2 - l_{23}^2$$

Now consider the triangle that is formed when two sides are added together to yield the bisecting vector listed above. The lengths of the two sides that we added are l_{12} and l_{31} and

the exterior angle is θ_1; this means that the interior angle is $\pi - \theta_1$ and $\cos(\pi - \theta_1) = -\cos\theta_1$. Now use the law of cosines to find the squared length of the bisecting vector to be

$$\begin{aligned} L^2 &= l_{12}^2 + l_{31}^2 + 2l_{12}l_{31}\cos\theta_1 \\ &= 2l_{12}^2 + 2l_{31}^2 - l_{23}^2 \end{aligned}$$

The length of the vector that points from $\mathbf{p}_1$ to $\mathbf{s}_1$ is found using the half-angle formula for the law of cosines.

$$\cos\frac{\theta_1}{2} = \sqrt{\frac{s(s-l_{23})}{l_{12}l_{31}}}$$

where

$$s = \frac{l_{12} + l_{23} + l_{31}}{2}$$

This is from the Math CRC p. 176. For a margin of m (`margin` in the code), the length of the vector from $\mathbf{p}_1$ to $\mathbf{s}_1$ is

$$d_1 = \frac{m}{\cos(\theta_1/2)}$$

This simplifies to

$$d_1 = m\sqrt{\frac{l_{12}l_{31}}{s(s-l_{23})}}$$

Putting the whole works together, the vector from $\mathbf{p}_1$ to $\mathbf{s}_1$ is

$$\mathbf{s}_1 - \mathbf{p}_1 = \left(\frac{\mathbf{p}_2 - \mathbf{p}_1}{l_{12}} - \frac{\mathbf{p}_1 - \mathbf{p}_3}{l_{31}}\right) m\sqrt{\frac{l_{12}l_{31}}{s(s-l_{23})(2l_{12}^2 + 2l_{31}^2 - l_{23}^2)}}$$

For the other corners, indices can be simply incremented, modulo 3.

`void Geo_InsidePoints32(double **point, double margin, double **ans);`

This is identical to `Geo_InsidePoints3`, but uses a little more information and runs much faster. Enter `point` so that `point[0...2]` are the three triangle vertices and `point[3...5]` are the three triangle edge normals. This returns `ans[0...2]` with the new triangle vertices. New function Oct. 2015.

As before, assume that the three triangle vertices are $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$, and that the inside points are $\mathbf{s}_1$, $\mathbf{s}_2$, and $\mathbf{s}_3$. Of these, we only consider $\mathbf{p}_1$ and $\mathbf{s}_1$ since the others are analogous. Also, assume that $\mathbf{e}_{12}$, $\mathbf{e}_{23}$, and $\mathbf{e}_{31}$ are normalized outward-pointing normals for the three edges. The inside angle of corner 1 is θ_1. Point $\mathbf{s}_1$ is distance m from edge 12, measured perpendicular to the edge. Suppose this line hits the edge at distance a from $\mathbf{p}_1$. Then,

$$\tan\frac{\theta_1}{2} = \frac{m}{a} = \frac{1-\cos\theta_1}{\sin\theta_1}$$

$$a = \frac{m\sin\theta_1}{1-\cos\theta_1}$$

The latter equality in the first equation is from the Math CRC p. 171. Now extend this line by distance b, until a line drawn from the endpoint is perpendicular to edge 31. The angle between this new line and edge 12 is $90^\circ - \theta_1$, implying that the interior angle at the new point is θ_1. For this angle,

$$\tan\theta_1 = \frac{a}{b} = \frac{\sin\theta_1}{\cos\theta_1}$$

$$b = \frac{a\cos\theta_1}{\sin\theta_1} = \frac{m\sin\theta_1\cos\theta_1}{\sin\theta_1(1-\cos\theta_1)} = \frac{m\cos\theta_1}{1-\cos\theta_1}$$

The distance between this new point and $\mathbf{s}_1$ is l_1, which is

$$l_1 = m + b = m\left(\frac{1-\cos\theta_1}{1-\cos\theta_1} + \frac{\cos\theta_1}{1-\cos\theta_1}\right) = \frac{m}{1-\cos\theta_1}$$

The new point, $\mathbf{s}_1$, is

$$\mathbf{s}_1 = \mathbf{p}_1 - l_1\mathbf{e}_{31} - l_1\mathbf{e}_{12}$$

Finally,

$$\cos\theta_1 = -\mathbf{e}_{31}\cdot\mathbf{e}_{12}$$

The other corners are the same.

Point in functions To be copied.

Nearest functions To be copied.

To rect functions To be copied.

X (cross) functions To be copied.

Reflect function

`double Geo_SphereReflectSphere(const double *a0, const double *a1, const double *b0, const double *b1, int dim, double radius2, double *a1p, double *b1p);`

Reflects hard spheres off of each other while maintaining trajectory lengths for each of them (this does not conserve momentum). Sphere A moves from `a0` to `a1` and sphere B moves from `b0` to `b1`. Enter `dim` as the system dimensionality and `radius2` as the squared sum of their radii. The reflected endpoints are returned in `a1p` for sphere A and `b1p` for sphere B. This function assumes that the spheres collide at some point, creating an error if not (returning `NaN`, `Inf`, `-Inf`, or some comparable error). Preferably, they collide at some point between positions 0 and 1 (e.g. their separation is greater than the radius at positions 0 and is less than the radius at positions 1). Returns the relative collision position, p (see below).

Here is the math. The first part is similar to that used in `Geo_LineXSphs`. Define p as the relative collision position along the two trajectories, equal to 0 for positions 0, 1 for positions

1, and an intermediate value for other collision positions. Using this, define the A and B positions along their trajectories at the collision time as

$$\begin{aligned}
\mathbf{a}_p &= (1-p)\mathbf{a}_0 + p\mathbf{a}_1 \\
\mathbf{b}_p &= (1-p)\mathbf{b}_0 + p\mathbf{b}_1 \\
R^2 &= (\mathbf{b}_p - \mathbf{a}_p)^2 \\
&= [((1-p)\mathbf{b}_0 + p\mathbf{b}_1) - ((1-p)\mathbf{a}_0 + p\mathbf{a}_1)]^2 \\
&= [(1-p)(\mathbf{b}_0 - \mathbf{a}_0) + p(\mathbf{b}_1 - \mathbf{a}_1)]^2
\end{aligned}$$

This last equation is identical in form to one from the `Geo_LineXSphs` derivation, so use that solution,

$$\begin{aligned}
a &= (\mathbf{b}_1 - \mathbf{a}_1 - \mathbf{b}_0 + \mathbf{a}_0)^2 \\
b &= 2(\mathbf{b}_0 - \mathbf{a}_0) \cdot (\mathbf{b}_1 - \mathbf{a}_1 - \mathbf{b}_0 + \mathbf{a}_0) \\
c &= (\mathbf{b}_0 - \mathbf{a}_0)^2 - R^2 \\
p &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\end{aligned}$$

The two solutions for p are for the time when the trajectories are first within R of each other and the time when they are last within R of each other. If they do not cross, then there are no solutions and the code will try to take the square root of a negative number. The negative root is the one that is of interest.

Next, using the assumption that the trajectory lengths are not affected by the collision, the trajectories from the collision times to the end times, meaning from $\mathbf{a}_p$ to $\mathbf{a}_1$ and from $\mathbf{b}_p$ to $\mathbf{b}_1$, are reflected across the plane that is perpendicular to the line that extends from $\mathbf{a}_p$ to $\mathbf{b}_p$. This plane is the contact plane for the A and B spheres, regardless of the relative sphere radii. From Wikipedia "Reflection (mathematics)", section on "Reflection through a hyperplane in n dimensions", reflecting vector $\mathbf{v}$ across a plane with normal vector $\mathbf{a}$ is achieved with

$$\mathbf{v}' = \mathbf{v} - 2\frac{\mathbf{v} \cdot \mathbf{a}}{\mathbf{a} \cdot \mathbf{a}}\mathbf{a}$$

Applying this to the current problem leads to the equations

$$\begin{aligned}
\mathbf{a}_1' - \mathbf{a}_p &= (\mathbf{a}_1 - \mathbf{a}_p) - 2\frac{(\mathbf{a}_1 - \mathbf{a}_p) \cdot (\mathbf{b}_p - \mathbf{a}_p)}{R^2}(\mathbf{b}_p - \mathbf{a}_p) \\
\mathbf{b}_1' - \mathbf{b}_p &= (\mathbf{b}_1 - \mathbf{b}_p) - 2\frac{(\mathbf{b}_1 - \mathbf{b}_p) \cdot (\mathbf{b}_p - \mathbf{a}_p)}{R^2}(\mathbf{b}_p - \mathbf{a}_p)
\end{aligned}$$

Here, twice the projection of the initial displacement on the reflection vector is subtracted from the initial displacement to produce the final displacement. Note that $(\mathbf{b}_p - \mathbf{a}_p)^2 = R^2$. These equations rearrange to

$$\begin{aligned}
\mathbf{a}_1' &= \mathbf{a}_1 - \frac{2(1-p)}{R^2}\left[(\mathbf{a}_1 - \mathbf{a}_0) \cdot \Delta_p\right]\Delta_p \\
\mathbf{b}_1' &= \mathbf{b}_1 - \frac{2(1-p)}{R^2}\left[(\mathbf{b}_1 - \mathbf{b}_0) \cdot \Delta_p\right]\Delta_p
\end{aligned}$$

where $\Delta_p = \mathbf{b}_p - \mathbf{a}_p$. These equations are in the code.

Extension added on 8/16/23. The above equations and logic are correct for spheres that are hit front to front, such as 2 balls that collide head-on. This is all that I investigated in my 2017 paper. However, they are incorrect for a sphere that is hit in the back (relative to its direction of motion). In this case, the direction of the reflected vector needs to be reversed, causing a slight change to the equations from above,

$$\mathbf{a}_p - \mathbf{a}_1' = (\mathbf{a}_1 - \mathbf{a}_p) - 2\frac{(\mathbf{a}_1 - \mathbf{a}_p)\cdot(\mathbf{b}_p - \mathbf{a}_p)}{R^2}(\mathbf{b}_p - \mathbf{a}_p)$$
$$\mathbf{b}_p - \mathbf{b}_1' = (\mathbf{b}_1 - \mathbf{b}_p) - 2\frac{(\mathbf{b}_1 - \mathbf{b}_p)\cdot(\mathbf{b}_p - \mathbf{a}_p)}{R^2}(\mathbf{b}_p - \mathbf{a}_p)$$

These rearrange to

$$\mathbf{a}_1' = 2(1-p)\mathbf{a}_0 + (2p-1)\mathbf{a}_1 + \frac{2(1-p)}{R^2}\left[(\mathbf{a}_1 - \mathbf{a}_0)\cdot\Delta_p\right]\Delta_p$$
$$\mathbf{b}_1' = 2(1-p)\mathbf{b}_0 + (2p-1)\mathbf{b}_1 + \frac{2(1-p)}{R^2}\left[(\mathbf{b}_1 - \mathbf{b}_0)\cdot\Delta_p\right]\Delta_p$$

A sphere is hit from the front, in which the first set of equations should be used, if the dot product of its trajectory and the collision vector is negative. Vice versa, a sphere is hit from the back if this dot product is positive. Thus, the following equalities imply that it's hit from the front, in which case the first set of equations should be used.

$$(\mathbf{a}_1 - \mathbf{a}_0)\cdot(\mathbf{a}_p - \mathbf{b}_p) \le 0$$
$$(\mathbf{b}_1 - \mathbf{b}_0)\cdot(\mathbf{b}_p - \mathbf{a}_p) \le 0$$

Exit functions To be copied.

Xaabb functions To be copied.

Approximate Xaabb functions To be copied.

Volume functions To be copied.