Documentation for Geometry.h and Geometry.c

Steven Andrews, © 2006-2010

Header file

```
#ifndef __Geometry_h
#define __Geometry_h

// Center
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_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,int front,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 points
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);

// Point in
int Geo_PtInTriangle(double *pt1,double *pt2,double *pt3,double *norm,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);
void Geo_NearestLineSegPt(double *pt1,double *pt2,double *point,double*ans,int
      dim);
void Geo_NearestTriPt(double *pt1,double *pt2,double *pt3,double *norm,double
      *point,double *ans);
void Geo_NearestTrianglePt(double *pt1,double *pt2,double *pt3,double
      *norm,double *point,double *ans);
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);
void Geo_NearestCylinderPt(double *pt1,double *pt2,double rad,int dim,double
      *point,double *ans);
void Geo_NearestDiskPt(double *cent,double *axis,double rad,int dim,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_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);

// 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
```

Includes: `"Geometry.h", "math2.h"`.
Example program: `Smoldyn`

History: Started 2/06. Major rewrite and additions 12/06 and 2/07; included in this was a complete switch from floats to doubles. Added Center functions 9/3/07. Added some nearest functions 10/31/07. Added some area functions 1/12/09. Added several nearest functions 3/2/09. Added `Geo_LineXaabb` 3/22/10 and fixed a bug in `Geo_CylisXaabb3`. Fixed `Geo_NearestTriPt` and `Geo_NearestTrianglePt`, in which they didn't test nearest corner points correctly 3/8/11. Added `Geo_InsidePoints2` and `Geo_InsidePoints3` 6/24/11. Added `Geo_TriArea3D` 4/11/12.
12/~15/12 Ye Li added `Geo_Area3D` to library. I improved to use a more stable algorithm.
7/17/12 Added unit vector functions: `Geo_TriUnitVects`, `Geo_SphereUnitVects`, `Geo_CylUnitVects`, and `Geo_DiskUnitVects`.

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-existant crossings) but they never return false negatives (they never ignore a crossing that actually exists).

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. In general, it is permissable 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`.

The functions that test whether two objects intersect often make use of the separating axis theorem. However, they also often use my own methods.

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 `Geo_PtInTriangle`, `Geo_NearestTriPt`, and `Geo_NearestTrianglePt` use similar math, described here. A triangle is defined by points $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ and has unit normal $\mathbf{n}$. There is a test point $\mathbf{r}$. For the first two functions, we only care about the position of $\mathbf{r}$ relative to the triangular column that is defined as the column which is perpendicular to the plane of the triangle, and not about the position of $\mathbf{r}$ relative to the plane of the triangle. Nevertheless, it is most convenient to think about $\mathbf{r}$ being either in or nearly in the plane of the triangle. The triangle is drawn so that $\mathbf{n}$ points upward out of the plane of the paper. $\mathbf{r}$ is within this triangle if it is on the left of the vector from $\mathbf{p}_1$ to $\mathbf{p}_2$, and left of the vector from $\mathbf{p}_2$ to $\mathbf{p}_3$, and left of the vector from $\mathbf{p}_3$ to $\mathbf{p}_1$; otherwise it is outside of the triangle. This is the basis of the `Geo_PtInTriangle` function. The amount by which $\mathbf{r}$ is left of the vector from $\mathbf{p}_1$ to $\mathbf{p}_2$ is

$$x = \frac{\left[\left(\mathbf{p}_2 - \mathbf{p}_1\right)\times\left(\mathbf{r} - \mathbf{p}_2\right)\right]\cdot\mathbf{n}}{\left|\mathbf{p}_2 - \mathbf{p}_1\right|}$$

The numerator of this function is denoted `cross12` in the code. To determine if $\mathbf{r}$ is inside or outside of the triangle, only the signs of these values are required. To find the closest point that is in the triangle to $\mathbf{r}$, which is denoted $\mathbf{s}$, some tests are needed. If $\mathbf{r}$ is already in the triangle, then $\mathbf{s}$ equals $\mathbf{r}$. If $\mathbf{r}$ is on the right side of the vector from $\mathbf{p}_1$ to $\mathbf{p}_2$, then the scaled dot product

$$\frac{\left(\mathbf{p}_2 - \mathbf{p}_1\right)\cdot\left(\mathbf{r} - \mathbf{p}_1\right)}{\left(\mathbf{p}_2 - \mathbf{p}_1\right)^2}$$

is ≤ 0 if point 1 is closest, is between 0 and 1 if the side is closest, and is ≥ 1 if point 2 is closest. If the side is closest, then $\mathbf{s}$ is found by combining the scaled dot product with the edge vector. If the closest point is a corner, $\mathbf{s}$ is

$$\mathbf{s} = \mathbf{p}_2 + \left[\left(\mathbf{r} - \mathbf{p}_2\right) \cdot \mathbf{n}\right]\mathbf{n}$$

Similar conditions and equations apply for the other corners. The function `Geo_NearestTrianglePoint` performs the same tests to see if where **r** lies relative to the infinite column, and then uses them in slightly different ways. If **r** is within the triangle, it projects it down to the plane; if **r** is closest to a corner, it returns that corner; and if **r** is closest to an edge, it uses a dot product to project **r** along that edge and then it returns the proper edge point.

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.

For the line crossing sphere function, here is the math. Consider a circle with its center at the origin and radius R. A line segment goes from **r** to **s**. If it crosses the circle, a crossing point will be denoted **x** (it may cross 0, 1, or 2 times, and also the crossing point(s) may be on the line but not in the line segment). The relative position of the crossing point on the line, where **r** defines 0 and **s** defines 1, is p. Because the crossing point is on the circle and the center is the origin, $\mathbf{x}^2 = R^2$. Also, the crossing point can be given by interpolation between the ends: $\mathbf{x} = (1-p)\mathbf{r} + p\mathbf{s}$. These are set equal to each other and solved for p.

$$R^2 = \left[(1-p)\mathbf{r} + p\mathbf{s}\right]^2$$
$$0 = p^2\left(\mathbf{s}-\mathbf{r}\right)^2 + p\left[2\mathbf{r}\cdot\left(\mathbf{s}-\mathbf{r}\right)\right] + \mathbf{r}^2 - R^2$$

These lead to p being the solution of the quadratic equation with coefficients

$$a = \left(\mathbf{s}-\mathbf{r}\right)^2$$

$$b = 2\mathbf{r} \cdot (\mathbf{s} - \mathbf{r})$$
$$c = \mathbf{r}^2 - R^2$$

If the circle center is not at the origin, then the circle center position, **c**, is subtracted from each vector to yield the general solution:

$$a = (\mathbf{s} - \mathbf{r})^2$$
$$b = 2(\mathbf{r} - \mathbf{c}) \cdot (\mathbf{s} - \mathbf{r})$$
$$c = (\mathbf{r} - \mathbf{c})^2 - R^2$$

If $b^2–4ac$ is positive, there are two solutions; if it is negative, there are no solutions; and if it is zero, there is one solution. The number of solutions gives the number of crossings, and the values give the crossing positions. Exactly the same math works for spheres and higher dimension spheres. This was re-checked and found to be correct.

The function also returns information about the nearest point on the line to the sphere center. For that, the position along the line from **r** towards **s**, in absolute units, is

$$pos_{abs} = \frac{(\mathbf{c} - \mathbf{r}) \cdot (\mathbf{s} - \mathbf{r})}{\sqrt{(\mathbf{s} - \mathbf{r})^2}}$$

The distance to the center from the line, *dist*, is found with the Pythagorean theorem

$$dist^2 = (\mathbf{c} - \mathbf{r})^2 - pos_{abs}^2$$

$$dist = \sqrt{(\mathbf{c} - \mathbf{r})^2 - \frac{\left[(\mathbf{c} - \mathbf{r}) \cdot (\mathbf{s} - \mathbf{r})\right]^2}{(\mathbf{s} - \mathbf{r})^2}}$$

Converting the absolute position to relative units along the line from r to s yields

$$pos = \frac{(\mathbf{c} - \mathbf{r}) \cdot (\mathbf{s} - \mathbf{r})}{(\mathbf{s} - \mathbf{r})^2}$$

The vector products required for these calculations are the same as those for the crossing points, so the latter set of information can be produced with very little additional effort.

For the line crossing line function, here is the math. Line 1 goes from **c** to **d**. Line 2 goes from **r** to **s**. Where they cross, a point along line 1 equals a point along line 2: $\mathbf{c} + a(\mathbf{d}–\mathbf{c}) = \mathbf{r} + b(\mathbf{s}–\mathbf{r})$, where *a* is the position along line 1 and *b* is the position along line 2. This is in two dimensions, so we have two equations and two unknowns. The math is a bit lengthy but results in the equations:

$$a = \frac{(r_x - c_x)(s_y - r_y) - (r_y - c_y)(s_x - r_x)}{(d_x - c_x)(s_y - r_y) - (d_y - c_y)(s_x - r_x)}$$

$$b = \frac{(r_x - c_x)(d_y - c_y) - (r_y - c_y)(d_x - c_x)}{(d_x - c_x)(s_y - r_y) - (d_y - c_y)(s_x - r_x)}$$

The line segments cross if $0 \le a \le 1$ and $0 \le b \le 1$, where the "or equal" portions of these inequalities depend on what ends are included or excluded. Since a and b have the same denominators, both will have zero denominators for the same situations, which occurs when then lines are parallel or when **c** = **d** or **r** = **s**.

For the disk crossing aabb function, here is the math. Given a normalized test vector (*e.g.* a normal to an aabb side), how far does the disk extend in the positive and negative vector directions? The answer is that it extends $\pm R\sin\theta$, where θ is the angle between the test vector and the disk's unit normal vector (**n**). I am finding $\sin\theta$ using the length of the cross-product vector. This sounds lengthy but simplifies dramatically if the test vector is parallel to a Cartesian coordinate. Results are:

test = (1,0,0)	$\sin\theta = (n_y^2+n_z^2)^{1/2}$
test = (0,1,0)	$\sin\theta = (n_x^2+n_z^2)^{1/2}$
test = (0,0,1)	$\sin\theta = (n_x^2+n_y^2)^{1/2}$

For the function `Geo_NearestSlabPt`, here is the math. The slab is defined by points $\mathbf{p}_1$ and $\mathbf{p}_2$. The test point is **r** and the nearest point in the slab is **s**. The dot product of $\mathbf{r}-\mathbf{p}_1$ with $\mathbf{p}_2-\mathbf{p}_1$, divided by the length of $\mathbf{p}_2-\mathbf{p}_1$, is the length of $\mathbf{r}-\mathbf{p}_1$ in the $\mathbf{p}_2-\mathbf{p}_1$ direction. Dividing again by the length of $\mathbf{p}_2-\mathbf{p}_1$ yields the position of **r** relative to the vector from $\mathbf{p}_1$ to $\mathbf{p}_2$: 0 means that **r** is at the $\mathbf{p}_1$ side of the slab, 1 means that it's at the $\mathbf{p}_2$ side, a number between 0 and 1 means that it's within the slab, and so forth. This ratio is denoted x. If the point is past the $\mathbf{p}_1$ side, then x amount of the vector from $\mathbf{p}_1$ to $\mathbf{p}_2$ is subtracted from **r** to yield **s**, and similarly for $x > 1$.

`Geo_LineXaabb` uses Kay and Kayjia's algorithm, which I found on the web somewhere. The idea is that a D-dimensional aabb can be seen as the intersection of D slabs, with one perpendicular to the x-axis, one for the y-axis, etc. For a line to intersect this aabb, it must be within all slabs at once. To determine if this is the case, I find the "near" and "far" points for each slab. If the largest "near" point is larger than the smallest "far" point, then the line misses. Otherwise, it intersects (*i.e.* for 3-d, the sequence of near and far points must be NNNFFF for intersection to occur). If this is a finite line segment, then the largest "near" has to be ≤1 and the smallest far has to be ≥0.

Code documentation

Externally accessible functions

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 this 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_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 θ=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,int front,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. 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 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 $\mathbf{p}_1$, $\mathbf{p}_2$, 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,

$$2l_{12}l_{31}\cos\theta_1 = l_{12}^2 + l_{31}^2 - l_{23}^2$$

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.

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

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

$$d_1 = \frac{m}{\cos\frac{\theta_1}{2}}$$

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, indicies can be simply incremented, modulo 3.

Point in functions

```
int Geo_PtInTriangle(double *pt1,double *pt2,double *pt3,double *norm,double
    *test);
```

Tests to see if the 3-D point `test` is inside the triangle defined by `pt1`, `pt2`, and `pt3`, and which has normal vector `norm`. `norm` does not have to be normalized. Typically, `test` will be in the plane of the triangle, but this is not required; if it isn't, this determines whether `test` is in the trianglular column that is defined by the other

points and perpendicular to the plane of the triangle. The function returns 0 if not and 1 if so. If `test` is on a triangle boundary, 1 is returned, although round-off errors often make the boundary imperfectly defined. The method used here is to find the cross product of each triangle edge with the vector that goes from the second point on that edge to `test`; the dot product of that result with the triangle normal is positive if `test` is inside the triangle.

`int Geo_PtInSlab(double *pt1,double *pt2,double *test,int dim);`
Tests to see if the point test is in the slab of space between `pt1` and `pt2`, inclusive, which works for all dimensions. The slab of space is defined to have its boundaries perpendicular to the line that includes `pt1` and `pt2`.

`int Geo_PtInSphere(double *test,double *cent,double rad,int dim);`
Tests to see if the `dim`-D point `test` is inside the sphere centered about `cent` with radius `rad`. This works in all dimensions. The boundary is considered to be part of the sphere.

Nearest functions

`void Geo_NearestSlabPt(double *pt1,double *pt2,double *point,double *ans,int dim);`
Given a slab in `dim`-dimensional space that is defined by `pt1` and `pt2`, which has its boundaries perpendicular to the line that includes `pt1` and `pt2`, this finds the point within the slab that is closest to `point`. The result is returned in `ans`. `ans` may equal `point`.

`void Geo_NearestLineSegPt(double *pt1,double *pt2,double *point,double*ans,int dim);`
Given a line segment in `dim`-dimensional space that goes from `pt1` to `pt2`, this finds the point within the segment that is closest to the point `point`. The result is returned in `ans`, which may equal `point`.

`void Geo_NearestTriPt(double *pt1,double *pt2,double *pt3,double *norm,double *point,double *ans);`
Given a triangular column in 3-dimensional space that is defined by points `pt1`, `pt2`, and `pt3`, and with unit normal `norm`, this finds the point that is within the triangular column that is closest to the point `point` and returns it in `ans`. The distance between `point` and the plane of the triangle is maintained for `ans`. `ans` may equal `point`.

`void Geo_NearestTrianglePt(double *pt1,double *pt2,double *pt3,double *norm,double *point,double *ans);`
Given a triangle in 3-dimensional space that is defined by points `pt1`, `pt2`, and `pt3`, and with unit normal `norm`, this finds the point that is within the triangle that is closest to the point `point` and returns it in `ans`. The result will be in the plane of the triangle. `ans` may equal `point`.

`double Geo_NearestSpherePt(double *cent,double rad,int front,int dim,double *point,double *ans);`

Finds the point on the surface of a spherical shell that is closest to the point `point`. The sphere is centered at `cent`, has radius `rad`, and has its front facing outwards if `front` is 1 and inwards if `front` is -1. Space is `dim` dimensional, which may be any value. The result is returned in `ans` and the function also returns the distance between `point` and the point returned in `ans`; this returned distance is positive if point is on the front side and negative if it is on the back side. `ans` may equal `point`.

```
void Geo_NearestRingPt(double *cent,double *axis,double rad,int dim,double
     *point,double *ans);
```
Finds the point on a ring that is closest to the point `point`. This ring is centered at `cent`, has axis `axis`, and has radius `rad`. It is in `dim` dimensional space, which may be any value. The result is returned in `ans`. `ans` may equal `point`.

```
void Geo_NearestCylPt(double *pt1,double *axis,double rad,int dim,double
     *point,double *ans);
```
This finds the point in an infinite solid cylinder that is closest to the point `point`. The cylinder is defined by the point `pt1`, which is on the axis, the normalized axis vector `axis`, and the radius `rad`. This works for any dimensionality, `dim`. The answer is returned in `ans`, which is allowed to be the same as `point`.

```
void Geo_NearestCylinderPt(double *pt1,double *pt2,double rad,int dim,double
     *point,double *ans);
```
Finds the point on a finite length cylindrical shell that is closest to the point `point`. The cylindrical shell axis extends from `pt1` to `pt2` and its radius is `rad`. It is in `dim` dimensions, which can equal any number. The result is returned in `ans`. `ans` may equal `point`.

```
void Geo_NearestDiskPt(double *cent,double *axis,double rad,int dim,double
     *point,double *ans);
```
Finds the point on the circular disk that is closest to the point `point`. The disk is centered at `cent`, is perpendicular to the normalized vector `axis`, and has radius `rad`. It is in `dim` dimensional space, which can be any value. The result is returned in `ans`. `ans` may equal `point`.

```
double Geo_NearestLine2LineDist(double *ptA1,double *ptA2,double *ptB1,double
     *ptB2)
```
Returns the closest distance between two lines, for a 3-D system. I didn't keep the math for this, so I'm not sure how it works. I probably found the algorithm on the web somewhere.

```
double Geo_NearestSeg2SegDist(double *ptA1,double *ptA2,double *ptB1,double
     *ptB2);
```
Returns the closest distance between two line segments, for a 3-D system. I didn't keep the math for this, so I'm not sure how it works. I probably found the algorithm on the web somewhere.

To Rect functions

`void Geo_Semic2Rect(double *cent,double rad,double *outvect,double *r1,double *r2,double *r3);`

Calculates the smallest non-axis-aligned rectangle that encloses the semicircle that has center at `cent`, radius `rad`, and outward pointing normalized vector `outvect`. The rectangle is returned with the two perpendicular edges extending from `r1` to `r2`, and `r1` to `r3`. `r1` is an end of the semicircle, on the right side of `outvect`, `r2` is the other end, and `r3` is behind `r1`, not on the semicircle.

`void Geo_Hemis2Rect(double *cent,double rad,double *outvect,double *r1,double *r2,double *r3,double *r4);`

Calculate the smallest non-axis-aligned rectangle that encloses the hemisphere that has center at `cent`, radius `rad`, and outward pointing normalized vector `outvect`. The rectangle is returned with the three perpendicular edges extending from `r1` to `r2`, `r1` to `r3`, and `r1` to `r4`. `r1`, `r2`, and `r3` are in the plane of the opening of the hemisphere and `r4` is behind `r1`; no points contact the surface.

`void Geo_Cyl2Rect(double *pt1,double *pt2,double rad,double *r1,double *r2,double *r3,double *r4);`

Calculate the smallest non-axis-aligned rectangle that encloses the cylinder whose axis extends from `pt1` to `pt2`, and has radius `rad`. The rectangle is returned with the three perpendicular edges extending from `r1` to `r2`, `r1` to `r3`, and `r1` to `r4`. `r1`, `r2`, and `r3` are in the plane of the end of the cylinder around `pt1` and `r4` is in the plane of the end around `pt2`, behind `r1`; no points contact the surface.

X (cross) functions

`double Geo_LineXLine(double *l1p1,double *l1p2,double *l2p1,double *l2p2,double *crss2ptr);`

Returns the point at which a line segment that goes from `l1p1` to `l2p2` crosses another which goes from `l2p1` to `l2p2`. These are in 2 dimensions. The returned value is the distance along line 1 where they cross. If `crss2ptr` is not `NULL`, then it is returned pointing to the distance along line 2 where the crossing is. Distances are between 0 and 1 for crossing within the line segment and other values for other crossing positions on the infinite lines. If the two points that define a line segment are the same, for either line, or if the two lines are parallel, the function returns `NaN` in both the returned values and `crss2ptr`.

`double Geo_LineXSphs(double *pt1,double *pt2,double *cent,double rad,int dim,double *crss2ptr,double *nrdistptr,double *nrposptr);`

Returns various information about a line segment from `pt1` to `pt2` that may or may not cross a spherical surface which has its center at `cent` and radius `rad`. The dimensionality of space, `dim`, can be any positive integer. The infinite line may cross the sphere surface once, twice, or not at all. The pointers `crss2ptr`, `nrdistptr`, and `nrposptr` are used for this function to return values and any or all can be set to `NULL` if those values aren't wanted. `nrdistptr` returns the nearest distance between `cent` and the infinite line and `nrposptr` returns the position on the line where that nearest distance occurs. This position is offset and scaled such that `pt1` is defined as 0 and `pt1` as 1. If the line does not cross the spherical surface, the

return value and *crss2ptr are returned as NaN. Otherwise, the return value and *crss2ptr are returned with the 2 crossing points on the line, scaled as before with pt1 as 0 and pt2 as 1, and the return value is the smaller of the two values. An error code (NaN, Inf, or similar) is returned if pt1 and pt2 are the same point. nrdistptr and nrposptr are identical to values returned by Geo_LineNormPos. If *nrdistptr is returned less than rad, it is guaranteed that the returned value and cross2 will be legitimate numbers and not NaN.

```
double Geo_LineXCyl2s(double *pt1,double *pt2,double *cp1,double *cp2,double
    *norm,double rad,double *crss2ptr,double *nrdistptr,double *nrposptr);
```
Determines if the infinite line defined by 2-dimensional points pt1 and pt2 crosses the infinite 2-dimensional "cylinder" surface that has an axis that includes points cp1 and cp2, has a unit right-side normal norm, and has radius rad. This cylinder is really two parallel lines. If the line crosses the cylinder surface at all, it will cross twice. The smaller of the two results is returned normally and the other is returned in crss2ptr. If the line does not cross the cylinder surface at all, both values will be NaN. Also, nrdistptr returns the nearest distance from the cylinder axis to the crossing line, which will be 0 unless they are parallel, in which case it returns the actual distance. If they are not parallel, nrpos returns the position along the pt1 to pt2 line at which it crosses the cylinder axis.

```
double Geo_LineXCyls(double *pt1,double *pt2,double *cp1,double *cp2,double
    rad,double *crss2ptr,double *nrdistptr,double *nrposptr);
```
Determines if the infinite line defined by points pt1 and pt2 crosses the infinite cylinder surface that has an axis that includes points cp1 and cp2 and has radius rad. This is in 3 dimensions. The infinite line may cross the cylinder surface once, twice, or not at all. See Geo_LineXSphs for details, all of which are the same for the two functions.

Xaabb functions

```
int Geo_LineXaabb2(double *pt1,double *pt2,double *norm,double *bpt1,double
    *bpt2);
```
Tests to see if a line segment crosses an axis-aligned boundary-box (aabb), all in 2-D. The line extents from pt1 to pt2 and has normal vector norm; norm does not have to be normalized. The aabb is defined by the low coordinates bpt1 and the high coordinates bpt2. Returns 0 if they do not cross and 1 if so.

```
int Geo_LineXaabb(double *pt1,double *pt2,double *bpt1,double *bpt2,int dim,int
    infline);
```
Tests to see if the line defined by pt1 and pt2 intersects the aabb with corners at bpt1 and bpt2, for a dim dimensional system. This is an infinite line if infline is 1 and a finite line segment from pt1 to pt2 if infline is 0. Returns 1 if they intersect and 0 if not.

This finds the highest "near" point in nearhi, which is the line's last entry into one of the axis-aligned slabs, and the lowest "far" point in farlo, which is the line's first

exit from one of the axis-aligned slabs. If the line is in all of the slabs at once, then the line crosses the aabb and 1 is returned.

`int Geo_TriXaabb3(double *pt1,double *pt2,double *pt3,double *norm,double *bpt1,double *bpt2);`
Tests to see if a triangle intersects an aabb, all in 3-D. The triangle is defined by `pt1`, `pt2`, `pt3`, and its normal vector `norm`; `norm` does not have to be normalized. The entire triangle area is considered, and not just the perimeter. The aabb is defined by the low coordinates `bpt1` and the high coordinates `bpt2`. Returns 1 for crossing and 0 for not.

`int Geo_RectXaabb2(double *r1,double *r2,double *r3,double *bpt1,double *bpt2);`
Tests to see if the 2-D rectangle, which is not necessarily axis-aligned, intersects a 2-D aabb, where the whole rectangle area is considered and not just the perimeter. The rectangle has reference point `r1`; `r2` is in one direction from the `r1` while `r3` is in the perpenicular direction. In the rectangle coordinates, `r1` is the origin, `r2` defines the local *x*-axis, and `r3` defines the local *y*-axis. Returns 1 for crossing a 0 for not. The aabb boundaries are as usual.

`int Geo_RectXaabb3(double *r1,double *r2,double *r3,double *r4,double *bpt1,double *bpt2);`
Tests to see if the 3-D rectangle (perpendicular parallelpiped, technically), which is not necessarily axis-aligned, intersects a 3-D aabb, where the whole rectangle volume is considered and not just the perimeter. The rectangle has reference point `r1`; `r2` is in one direction from the `r1`, `r3` is in a perpendicular direction, and `r4` is in the other perpendicular direction. Returns 1 for crossing a 0 for not. The aabb boundaries are as usual.

`int Geo_CircleXaabb2(double *cent,double rad,double *bpt1,double *bpt2);`
Tests to see if the perimeter of a circle crosses an aabb, all in 2-D. The circle is defined by its center location `cent` and its radius `rad`, while the aabb is defined by the low coordinates `bpt1` and the high coordinates `bpt2`. Returns 1 for crossing and 0 for not. The tests are: 1) if SAT on aabb fails, returns 0; 2) if all corners are inside circle, returns 0; 3) if any corners, but not all, are inside circle, returns 1; and 4) if circle center is within the axis-aligned stripes defined by the aabb, returns 1. Tests 2 and 3 treat all box corners as being included with the box.

`int Geo_SphsXaabb3(double *cent,double rad,double *bpt1,double *bpt2);`
Tests to see if the surface of a sphere cross an aabb, all in 3-D. The sphere is defined by its center location `cent` and its radius `rad`, while the aabb is defined by the low coordinates `bpt1` and the high coordinates `bpt2`. Returns 1 for crossing and 0 for not. See `Geo_CircleXaabb2` for the tests that are carried out.

`int Geo_CylisXaabb3(double *pt1,double *pt2,double rad,double *bpt1,double *bpt2);`
Tests to see if an infinite cylindrical shell crosses an aabb, in 3-D. The axis of the radius `rad` cylinder includes the points `pt1` and `pt2`, but is infinitely long. This

returns 1 for crossing and 0 for not. The method that this uses is to project along the cylinder axis so that the cylinder becomes a circle and the aabb becomes 8 2-D points that are connected by 12 edges. Then, several tests are run to see if the circle and the box shadow overlap: 1) if SAT on any edge direction fails, returns 0; 2) if all corners are inside the circle, returns 0; 3) if any corners, but not all, are inside the circle, returns 1; 4) if the cylinder center crosses the aabb, meaning that the cylinder may be fully inside the aabb, return 1; 5) if any edge crosses the circle, returns 1. While these many tests create a long function, I think that it is reasonably efficient. It should also be accurate.

```
int Geo_DiskXaabb3(double *cent,double rad,double *norm,double *bpt1,double
     *bpt2);
```

Tests to see if a solid, infinitely thin disk intersects an aabb, in 3-D. The disk is defined by its center coordinates in `cent`, its radius in `rad`, and its unit normal vector in `norm`. This looks for separating planes that are parallel to the aabb sides and that are parallel to the disk plane. It ignores other potential separating planes, which ought to be fixed. It returns 1 for crossing and 0 for not.

Approximate Xaabb functions

```
int Geo_SemicXaabb2(double *cent,double rad,double *outvect,double *bpt1,double
     *bpt2);
```

Tests to see if a semicircle perimeter crosses an aabb, in 2-D. This can return false positives, described below. `cent` is the semicircle center, `rad` is the semicircle radius, and `outvect` is the normalized outward pointing vector that defines the direction of the semicircle opening. The aabb is defined by `bpt1` and `bpt2`. This returns 1 for crossing and 0 for not. This returns 1 if both the full circle perimeter crosses the aabb, and the minimal rectangle that encloses the semicircle also crosses the aabb. False positive results can occur if the aabb extends into the semicircle interior but does not cross the semicircle.

```
int Geo_HemisXaabb3(double *cent,double rad,double *outvect,double *bpt1,double
     *bpt2);
```

Tests to see if a hemisphere shell crosses an aabb, in 3-D. This can return false positives, described below. `cent` is the hemisphere center, `rad` is the hemisphere radius, and `outvect` is the normalized outward pointing vector that defines the direction of the hemisphere opening. The aabb is defined by `bpt1` and `bpt2`. This returns 1 for crossing and 0 for not. This returns 1 if both the full spherical shell crosses the aabb, and the minimal rectangle that encloses the hemisphere also crosses the aabb. False positive results can occur if the aabb extends into the hemisphere interior but does not cross the hemisphere.

```
int Geo_CylsXaabb3(double *pt1,double *pt2,double rad,double *bpt1,double
     *bpt2);
```

Tests to see if a cylindrical shell crosses an aabb, in 3-D. This can return false positives, described below. The cylinder axis extends from `pt1` to `pt2` and `rad` is the cylinder radius. The aabb is defined by `bpt1` and `bpt2`. This returns 1 for crossing and 0 for not. This returns 1 if both the infinite length cylindrical shell crosses the aabb, and the minimal rectangle that encloses the cylinder also crosses the aabb. False positive results can occur if the aabb extends into the cylinder interior but does not cross the cylinder.

Volume functions

`double Geo_SphVolume(double rad,int dim);`

This returns the volume of a sphere of radius `rad` for any integer dimension `dim` that is greater than or equal to zero. The equation for arbitrary dimension volume is from Wolfram Mathworld "Hypersphere" eqs. 4 and 9.

`double Geo_SphOLSph(double *cent1,double *cent2,double r1,double r2,int dim);`

This returns the volume in the overlap region between two spheres that are centered at `cent1` and `cent2`, that have radii `r1` and `r2`, and that are `dim` dimensional. Zero is returned if the spheres do not overlap at all, the volume of the smaller sphere is returned if it is totally inside the larger, and values in between are returned for partial overlap. The former two results are valid for all sphere dimensions, whereas the last only works for `dim` equal to 1, 2, or 3; –1 is returned for higher dimensional spheres with partial overlap. Equations are from Wolfram Mathworld. The equation of 2-D sphere overlap is from their page on "Circle-Circle Intersection", eq. 14. The equation of 3-D sphere overlap is from their page on "Sphere-Sphere Intersection", eq. 16.