Documentation for Geometry.h and Geometry.c

Steven Andrews, © 2006-2015

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`.

Dependencies

```
Geometry.h
math2.h
```

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...` 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`

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.

Math

Several functions use the cross-product of two vectors. As a reminder, **c** = **a**×**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.

Code documentation

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 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, 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

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

$$a = \frac{m\sin\theta}{1-\cos\theta_{11}}$$

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 $90°-\theta_1$, implying that the interior angle at the new point is θ_1. For this angle,

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

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

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

```
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 triangular 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.

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

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

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

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

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

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

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

```
int Geo_PtInTriangle2(double **point,double *test);
```

This is identical to `Geo_PtInTriangle`, but requires a few more inputs and runs much faster. Enter `point` so that `point[0]` is triangle vertex 1, `point[1]` is vertex 2, `point[2]` is vertex 3, `point[3]` is the normal for the edge between vertices 1 and 2 (this normal is in the plane of the triangle), `point[4]` is the normal for the edge between vertices 2 and 3, and `point[5]` is the normal for the edge between vertices 3 and 1. New function Oct. 2015.

```
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`.

Here is the math. The slab is defined by points $\mathbf{p}_1$ and $\mathbf{p}_2$. The test point is $\mathbf{r}$ and the nearest point in the slab is $\mathbf{s}$. 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 $\mathbf{r}$ relative to the vector from $\mathbf{p}_1$ to $\mathbf{p}_2$: 0 means that $\mathbf{r}$ 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 $\mathbf{r}$ to yield $\mathbf{s}$, and similarly for $x > 1$.

```
int Geo_NearestLineSegPt(double *pt1,double *pt2,double *point,double*ans,int
    dim,double margin);
```

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`. This returns 0 if the result is not an endpoint of the line segment, 1 if it is at endpoint 1, and 2 if it is at endpoint 2. If point is within the segment but also within `margin` distance of an endpoint, it is treated as being at the endpoint.

```
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`.

For the math, see the description for `Geo_PtInTriangle`.

`void Geo_NearestTriPt2(double **point,double *norm,double *testpt,double *ans);`
Identical to `Geo_NearestTriPt`, except that this uses a little more information about the triangle and runs much faster. Send in `point` with `point[0...2]` equal to the three triangle vertices and `point[3...5]` equal to the three edge normals. New function Oct. 2015.

`int 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`. This returns 0 if the nearest point is within the interior of the triangle, 1 if it is on the edge between points 1 and 2, 2 if it is on the edge between points 2 and 3, and 3 if it is on the edge between points 3 and 1. Each edge includes the former of these points, but not the latter one (the function returns 1 if the closest point is at point `pt1`).

For the math, see the description for `Geo_PtInTriangle`.

`int Geo_NearestTrianglePt2(double **point,double *norm,double *testpt,double *ans,double margin);`
Essentially identical to `Geo_NearestTrianglePt` except that this uses a little more information and runs much faster. Send in `point` with `point[0...2]` equal to the three triangle vertices and `point[3...5]` equal to the three edge normals. If `testpt` is within `margin` distance of an edge it is treated as though it is at the edge.

`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`.

`int Geo_NearestCylinderPt(double *pt1,double *pt2,double rad,int dim,double *point,double *ans,double margin);`

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`. Returns 1 if the nearest point is within `margin` of the first cylinder edge, 2 if it is within `margin` of the second cylinder edge, and 0 if it is not on a cylinder edge.

`int 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`. Returns 1 if the nearest point is on the disk edge and 0 if it is not on the disk edge.

`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.

`double Geo_NearestAabbPt(const double *bpt1,const double *bpt2,int dim,const double *point,double *ans);`

Returns the closest distance between the point called `point` and the axis-aligned bounding box that has its low corner in `bpt1` and its high corner in `bpt2`, for a `dim` dimensional system. The closest point within, on the edge, or on the corner of the aabb, is return in `ans`. This works for all dimensions. `ans` cannot be `NULL` and needs to be different than `point`.

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);
```
Calculates 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. I fixed a bug here on 5/10/13 (vector `v2` wasn't being normalized).

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`.

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 $\mathbf{c} = \mathbf{d}$ or $\mathbf{r} = \mathbf{s}$.

`double Geo_LineXPlane(double *pt1,double *pt2,double *v,double *norm,double *crsspt);`

Computes the location on an infinite plane (in 3D) where the line that is defined by `pt1` and `pt2` crosses it. The plane is defined with a single point, called `v`, and its normal vector, called `norm`. The normal vector does not need to be of unit length. The 3D crossing point is returned in `crsspt`, which is not allowed to equal `NULL`. It is also returned directly as a scaled value along the line, where `pt1` equals 0 and `pt2` equals 1. This does not check for the possibility that the line is parallel to the plane. If that happens, then this function will probably return `±Inf` for a, or `NaN` if the line is in the plane, but I haven't tested this.

The math is easiest to see with a 2D drawing. Define $\mathbf{s}$ as the vector from $\mathbf{p}_1$ to $\mathbf{v}$ and $\mathbf{l}$ as the vector from $\mathbf{p}_1$ to $\mathbf{p}_2$. The dot product of these with the normal gives the length between $\mathbf{p}_1$ and the plane and between $\mathbf{p}_1$ and $\mathbf{p}_2$, respectively, each projected along the normal of the plane. The scaling is somewhat arbitrary, but that's ok because it's the same for both distances. The ratio of these two distances is the relative distance along the line.

`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 `pt2` 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`.

Here is the math. Consider a circle with its center at the origin and radius R. A line segment goes from $\mathbf{r}$ to $\mathbf{s}$. If it crosses the circle, a crossing point will be denoted $\mathbf{x}$ (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 $\mathbf{r}$ defines 0 and $\mathbf{s}$ 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.

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

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

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

If the circle center is not at the origin, then the circle center position, $\mathbf{c}$, is subtracted from each vector to yield the general solution:

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

Plugging this into the quadratic equation gives the solutions

$$p = \frac{-2(\mathbf{r}-\mathbf{c})\cdot(\mathbf{s}-\mathbf{r}) \pm \sqrt{\left[2(\mathbf{r}-\mathbf{c})\cdot(\mathbf{s}-\mathbf{r})\right]^2 - 4(\mathbf{s}-\mathbf{r})^2\left[(\mathbf{r}-\mathbf{c})^2 - R^2\right]}}{2(\mathbf{s}-\mathbf{r})^2}$$

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 $\mathbf{r}$ towards $\mathbf{s}$, 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

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

Note that this position is the same as the result from the quadratic equation shown above when the radical is set to zero. 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.

```
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.

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

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

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

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

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. Reflection leads to the equations

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

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 $(\mathbf{b}_p\text{-}\mathbf{a}_p)^2 = R^2$. See Wikipedia "reflection". These equations simplify to

$$\mathbf{a}_1' = \mathbf{a}_1 - 2\frac{(1-p)(\mathbf{a}_1 - \mathbf{a}_0)\cdot(\mathbf{b}_p - \mathbf{a}_p)}{R^2}(\mathbf{b}_p - \mathbf{a}_p)$$
$$\mathbf{b}_1' = \mathbf{b}_1 - 2\frac{(1-p)(\mathbf{b}_1 - \mathbf{b}_0)\cdot(\mathbf{b}_p - \mathbf{a}_p)}{R^2}(\mathbf{b}_p - \mathbf{a}_p)$$

These equations are in the code.

Exit functions

```
double Geo_LineExitRect(double *pt1,double *pt2,double *front,double
    *corner1,double *corner3,double *exitpt,int *exitside);
```

Finds the point on the rectangle edge where the line defined by point `pt1` and `pt2` exits the rectangle. The rectangle is assumed to be axis-aligned. This function works in 3D only. The rectangle is actually an infinite rectangular prism. Its alignment is given with the `front` vector, from Smoldyn, where `front[0]` is whether the perpendicular axis faces in the + or - direction, `front[1]` is the rectangle's perpendicular axis number, and `front[2]` is the axis parallel to the rectangle edge from point 0 to point 1. The rectangle has its point 0 corner at `corner1` and the diagonal corner, the point 2 corner, at `corner3`. The exit point is returned in `exitpt` (which cannot be `NULL`). The exit point's position along the line, scaled so that 0 represents `pt1` and 1 represents `pt2`, is returned from the function directly. `pt1` is not allowed to equal `pt2`. If the line does not actually intersect the rectangle, then the returned exit point is still on the line, but is not on the rectangle edge. The side of the rectangle where the line exits is returned in `exitside` (which cannot be `NULL`). This value is 1 for the side between points 0 and 1, 2 for the side between points 1 and 2, 3 for the side between points 2 and 3, and 4 for the side between points 3 and 0.

This function first determines the axis numbers for the rectangle sides, calling them x and y internally, although these do not necessarily correspond to the actual x and y axes. The internal x axis is the rectangle's "primary" axis, meaning the one that goes from point 0 to point 1, and the y axis is the rectangle's "secondary" axis, which goes from point 1 to point 2. It then computes scaled crossing values for each of the 4 sides, using the numbering given above. Because this is the exit point, the larger of the two values is chosen for each axis. Then, the smaller of the values for the two different axes is chosen. This value is returned.

`double Geo_LineExitLine2(double *pt1,double *pt2,double *end1,double *end2,double *exitpt,int *exitend);`

Assumes that two line segments lie essentially along the same line, all of which is in 2D; one segment goes from `pt1` to `pt2` and the other goes from `end1` to `end2`. This determines where the first line segment exits the second segment, which is returned in `exitpt`, which cannot be `NULL`. It is at either `end1` or `end2`, depending on the relative directions of the two line segments. This also determines where along the first segment this exit point is, which is returned directly. `pt1` and `pt2` are not allowed to be the same point. However, `end1` and `end2` are allowed to be the same. `exitend`, which is not allowed to equal `NULL` is returned pointing to the end number that the line exits through, which is 1 for `end1` and 2 for `end2`.

This determines whether it's best to work with the *x*-axis or *y*-axis and then does computations with just that axis.

`void Geo_LineExitArc2(double *pt1,double *pt2,double *cent,double radius,double *norm,double *exitpt,int *exitend);`

Finds the endpoint of a 2D semicircular arc where the curved line that is defined by `pt1` and `pt2` will leave the semicircle. The arc is defined with its center at `cent`, with radius `radius`, and with an outward-pointing unit normal vector of `norm`. This normal vector points in the direction that the semicircle is open. This function assumes that `pt1` and `pt2` are not the same and that `pt1` does not equal the center of the arc. It returns the position where the line leaves the arc as (x,y) coordinates in `exitpt`, which is not allowed to equal `NULL`; this is one of the two arc endpoints, where which one depends on the line direction. `exitend`, which cannot be `NULL`, is returned pointing to the end number that the line exits through. It is 1 if the line exits through the clockwise end of the arc and 2 if the line exits through the counter-clockwise end of the arc.

This function computes the *z*-component of the cross product between the vector from the arc center to `pt1` and the vector from `pt1` to `pt2`. The sign of this cross-product indicates the line's direction of travel along the arc and hence which end of the arc the line will exit through. Next, it computes the position of the appropriate arc end point.

`double Geo_LineExitTriangle(double *pt1,double *pt2,double *normal,double *v1,double *v2,double *v3,double *exitpt,int *exitside);`

Finds the point on the infinitely long 3D triangular prism through which the line that is defined by points `pt1` and `pt2` exits the triangular prism. The triangular prism is defined by a triangle with vertices at `v1`, `v2`, and `v3` and has a normal vector equal to `normal`. The exiting point is returned in `exitpt`, which may not be `NULL`, and the location of this exit point along the line, where `pt1` is defined as 0 and `pt2` is defined as 1, is returned directly. Most often, the line will be nearly co-planar with the triangle, in which case this finds the place along one of the triangle's edges where the line leaves the triangle. `pt1` and `pt2` are not allowed to be the same point and none of the triangle vertices are allowed to equal each other. This is not designed for the case where the line does not intersect the triangular prism, although it will still return a reasonably sensible answer (the returned exit point is along the line but not within the triangular prism). `exitside`, which cannot be `NULL`, is returned pointing to the side number that the line exits through. The number is 1 for the side between `v1` and `v2`, 2 for the side between `v2` and `v3`, and 3 for the side between `v3` and `v1`.

First, this finds differences between the points to compute relative vectors rather than absolute points. Then, it computes cross-products of the triangle edges with the triangle normal in order to find the outward-pointing normal for each edge. For each one, it dots the outward-pointing normal with the line to see if the line enters or leaves the triangle through this edge. For each edge that the line leaves through, `Geo_LineXPlane` is called, which determines where along the edge the line leaves (it's a plane because this is really one side of the triangular prism). Finally, the leaving point that is closest along the line (i.e. smallest *a* value) is chosen because this is where the line leaves the triangle. There may be a larger *a* value as well for a second plane crossing, but the line will have already left the triangle by then.

```
double Geo_LineExitTriangle2(double *pt1,double *pt2,double **point,double
     *exitpt,int *exitside);
```

This is identical to `Geo_LineExitTriangle`, except that it requires a little more prior information and it runs much faster. It finds the point on the infinitely long 3D triangular prism through which the line that is defined by point `pt1` and `pt2` exits the triangular prism. Enter the triangle parameters in `point`. Vertex 1 is in `point[0]`, vertex 2 is in `point[1]`, vertex 3 is in `point[2]`, the normal for edge 1-2 is in `point[3]`, the normal for edge 2-3 is in `point[4]`, and the normal for edge 3-1 is in `point[5]`. The exiting point is returned in `exitpt`, which may not be `NULL`, and the location of this exit point along the line, where `pt1` is defined as 0 and `pt2` is defined as 1, is returned directly. Most often, the line will be nearly co-planar with the triangle, in which case this finds the place along one of the triangle's edges where the line leaves the triangle. `pt1` and `pt2` are not allowed to be the same point and none of the triangle vertices are allowed to equal each other. This is not designed for the case where the line does not intersect the triangular prism, although it will still return a reasonably sensible answer (the returned exit point is along the line but not within the triangular prism). `exitside`, which cannot be `NULL`, is returned pointing to the side number that the line exits through. The number is 1 for the side between `v1` and `v2`, 2 for the side between `v2` and `v3`, and 3 for the side between `v3` and `v1`. New function Oct. 2015.

The mathematics, and variable names, are identical to `Geo_LineExitTriangle`, with some additions from `Geo_LineXPlane`.

```
double Geo_LineExitSphere(double *pt1,double *pt2,double *cent,double
     rad,double *exitpt);
```

Finds the position on the 3D spherical shell where the line that is defined by points `pt1` and `pt2` exits the spherical shell. The sphere is defined with its center at `cent` and its radius equal to `rad`. The 3D position of the exit is returned in `exitpt` which may not be `NULL`. Also, the relative position on the line, defining `pt1` as 0 and `pt2` as 1, is returned directly. If the line does not intersect the spherical shell, then the point on the line that is closest to the shell is returned instead.

This function is very similar to `Geo_LineXSphs`, but is a little less general and is written to run substantially faster. It also handles non-intersecting situations slightly differently, in that it returns a crossing point despite the fact that there actually isn't one.

```
void Geo_LineExitHemisphere(double *pt1,double *pt2,double *cent,double
     rad,double *normal,double *exitpt);
```

Finds the position along the edge of a hemisphere where the line that is defined by points `pt1` and `pt2` exits, assuming that it travels along the surface of the hemisphere by a great circle route. The hemisphere has its center at `cent`, has radius `rad`, and has an outward-pointing normal vector of `normal`. The result is returned in `exitpt`, which is not allowed to be `NULL`.

Assume that `pt1` and `pt2` are both on the hemisphere surface. The great circle route that includes these two points is clearly in the plane that is defined by a straight line between the two points and by the hemisphere center. This considers where the straight line that connects the two points crosses the hemisphere plane. Then, it figures out where this is compared to the hemisphere surface.

```
double Geo_LineExitCylinder(double *pt1,double *pt2,double *end1,double
     *end2,double rad,double *exitpt,int *exitend);
```

Finds the approximate position at the end of a 3D cylinder surface where the line that is defined by `pt1` and `pt2` exits. The cylinder endpoints are `end1` and `end2` and the cylinder has radius `rad`. The coordinates where the line exits are returned in `exitpt`. The end is returned in `exitend`, which may not be `NULL`, where this is set to 1 if it is at `end1` and to 2 if it is at `end2`.

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.

`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.

`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, meaning that this function can return false positives (see the tests in Geo_CylisXaabb3 for tests that could be used here). This returns 1 for crossing and 0 for not.

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

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.