Minimum-Volume Box Containing a Set of Points

David Eberly, Geometric Tools, Redmond WA 98052
https://www.geometrictools.com/

Created: July 25, 2015
Last Modified: September 11, 2020

Contents

1 Introduction

Given a finite set of 3D points, we want to compute the minimum-volume box that contains the points. The box is not required to be axis aligned, and generally it will not be. It is intuitive that the minimum-volume box for the points is supported by the convex hull of the points. The hull is a convex polyhedron, and any points interior to the polyhedron have no influence on the bounding box. Thus, the first step of the algorithm is to compute the convex hull of the points.

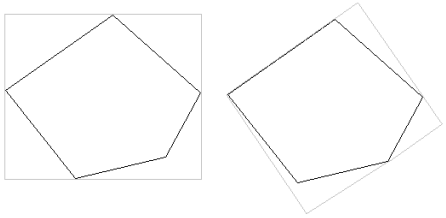
According to [2], the minimum-volume oriented box must have at least two adjacent faces flush with edges of the convex polyhedron. The box faces are necessarily perpendicular.[1] The implementation processes all pairs of edges, determining for each pair the relevant box-face normals that are candidates for the minimum-volume box. I use an approach different from that of the details of proof in the paper. The computations involve an iterative minimizer, so you cannot expect to obtain the exact minimum-volume box, but you will obtain a good approximation to it based on how many samples the minimizer uses in its search. You can also derive from a class and override the virtual functions that are used for minimization in order to provided your own minimizer algorithm.

The source code for the implementation is in the file MinimumVolumeBox3.h.

2 A Brief Sketch of the Theoretical Result

The idea is based on the *2D rotating calipers algorithm* for computing the minimum-area rectangle containing a convex polygon [1]. An edge of the minimum-area rectangle must be coincident with some edge of the polygon. In some cases, multiple edges of the rectangle can coincide with polygon edges. The theoretical result and an implementation of this algorithm is discussed in Minimum-Area Rectangle Containing a Set of Points. Figure 1 shows a polygon bounded by a non-minimal-area rectangle and by a minimal-area rectangle.

Figure 1. The left part of the figure shows a polygon with a bounding rectangle that is supported only by polygon vertices but not by polygon edges. The right part of the figure shows the same polygon with the minimum-area rectangle, which is supported by an edge of the polygon and vertices of the polygon.



Suppose that the minimum-volume box containing a convex polyhedron is supported only by polyhedron vertices, not by polyhedron edges or polyhedron faces. Project the box and polyhedron onto the plane of a

[1] The version of this document before the September 1, 2020 version incorrectly stated that the minimum-volume box has a face coincident with a hull face or is supported by three mutually orthogonal edges of the hull. I recall obtaining this information from the Usenet group *comp.graphics.algorithms* a very long time ago, but I never verified the result by reading the actual paper. My error, which I have now corrected.

box face (any face). The result is a convex polygon with a bounding rectangle that is of the type shown in the left image of Figure 1. Rotate the box about the normal line of the projection face until the projected polygon has a minimum-area rectangle that is the projection of the rotated box. The volume of the rotated box must be smaller than that of the original. Therefore, the minimum-volume box cannot be supported by only vertices and not polyhedron edges and not polyhedron faces.

Similary, suppose that the minimum-volume box containing a convex polyhedron is supported by a single polyhedron edge and by some polyhedron vertices, but no other polyhedron edges or polyhedron faces are coincident with the box faces. Project the box and polyhedron onto the plane of the box face that is coincident with the polyhedron edge. Once again the result is a convex polygon with a bounding rectangle that is of the type shown in the left image of Figure 1. The box can be rotated until another polyhedron edge is coincident with a second box face, the second face perpendicular to the first face. The rotated box must have volume smaller than that of the original. Therefore, the minimum-volume box cannot be supported by only a single edge and some vertices.

These arguments show that the minimum-volume box must contain at least two polyhedron edges, one coincident with a box face and the other coincident with a perpendicular box face. In many cases, the minimum-volume box is supported by a polyhedron face. However, there are examples for which the minimum-volume box is not supported by any polyhedron face. The classic example is a regular tetrahedron, say, one that is inscribed in the unit sphere centered at the origin. The vertices and triangles are

$$\begin{aligned}&\mathbf{V}_0 = (0,0,1), \mathbf{V}_1 = (2\sqrt{2}/3, 0, -1/3), \mathbf{V}_2 = (-\sqrt{2}/3, \sqrt{6}/3, -1/3), \mathbf{V}_3 = (-\sqrt{2}/3, -\sqrt{6}/3, -1/3)\\ &T_0 = \{0,1,2\}, T_1 = \{0,2,3\}, T_2 = \{0,3,1\}, T_3 = \{1,3,2\}\end{aligned} \tag{1}$$

The minimum-volume box is not unique; in fact, there are 4 such boxes, each having 3 faces flush with the edges emanating from a single vertex. The boxes are cubes centered at the origin with side length $1/\sqrt{3}$ and volume $8/\sqrt{27}$. One has axes $(\sqrt{2}/\sqrt{3}, 0, -1/\sqrt{3})$, $(1/\sqrt{6}, -1/\sqrt{2}, -1/\sqrt{3})$ and $(-1/\sqrt{6}, -1/\sqrt{2}, 1/\sqrt{3})$.

3 Processing a Pair of Edges of the Polyhedron

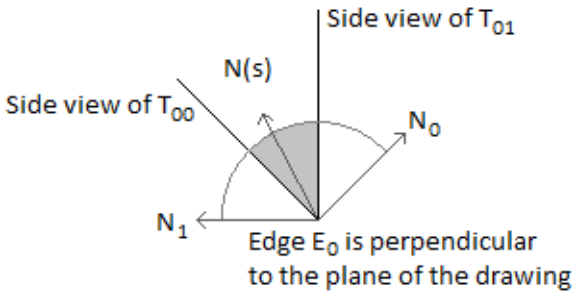
I make the assumption that the convex polyhedron is represented by a vertex-edge-triangle 2-manifold mesh. It is possible that two or more triangles are coplanar, but the algorithm does not suffer any degeneracies from this.

Given two edges of the convex polyhedron, $\mathbf{E}_0$ and $\mathbf{E}_1$, the edge $\mathbf{E}_0$ is shared by two triangles $\mathbf{T}_{00}$ and $\mathbf{T}_{01}$ and the edge $\mathbf{E}_1$ is shared by two triangles $\mathbf{T}_{10}$ and $\mathbf{T}_{11}$. The edges themselves are not necessarily perpendicular, but it is possible that there exist two perpendicular planes, one containing $\mathbf{E}_0$ nad one containing $\mathbf{E}_1$. The problem is to determine normal vectors for those perpendicular planes. As it turns out, there can be infinitely many pairs of perpendicular planes that contain the edges as described.

Consider edge $\mathbf{E}_0$ and a candidate supporting plane. The only possible unit-length inner-pointing normals for the supporting plane live on a spherical arc connecting the inner-pointing normals for the triangles $\mathbf{T}_{00}$ and $\mathbf{T}_{01}$.

Because each of the perpendicular planes supports the convex polyhedron at the corresponding edge, the only possible normals for the supporting plane live on an a spherical arc connecting the normals for the triangles shared by an edge. Figure 2 illustrates the idea.

Figure 2. The spherical arc connecting the normals for the triangles sharing an edge are the candidates for the box-face normal supporting the edge.



The edge $\mathbf{E}_0$ is perpendicular to the plane of the figure. The triangles sharing the edge are seen edge-on. The gray shaded region stresses that this is the inside of the convex polyhedron. The unit-length inner-pointing normals for the triangles are $\mathbf{N}_0$ and $\mathbf{N}_1$. The edge acts as a hinge joint. The supporting plane can rotate about the hinge with unit-length normals

$$\mathbf{N}(s) = \frac{(1-s)\mathbf{N}_0 + s\mathbf{N}_1}{|(1-s)\mathbf{N}_0 + s\mathbf{N}_1|}, \quad s \in [0,1] \tag{2}$$

For edge $\mathbf{E}_1$ with shared triangles $\mathbf{T}_{10}$ and $\mathbf{T}_{11}$ with unit-length inner-pointing normals $\mathbf{M}_0$ and $\mathbf{M}_1$, the possible supporting unit-length plane normals are

$$\mathbf{M}(t) = \frac{(1-t)\mathbf{M}_0 + t\mathbf{M}_1}{|(1-t)\mathbf{M}_0 + t\mathbf{M}_1|}, \quad t \in [0,1] \tag{3}$$

The supporting planes are perpendicular when the following bilinear function is zero,

$$\begin{aligned} F(s,t) &= \mathbf{N}(s) \cdot \mathbf{M}(t) \\ &= (1-s)(1-t)\mathbf{N}_0 \cdot \mathbf{M}_0 + s(1-t)\mathbf{N}_1 \cdot \mathbf{M}_0 + (1-s)t\mathbf{N}_0 \cdot \mathbf{M}_1 + st\mathbf{N}_1 \cdot \mathbf{M}_1 \\ &= (1-s)(1-t)f_{00} + s(1-t)f_{10} + (1-s)tf_{01} + stf_{11} \end{aligned} \tag{4}$$

where the last equation defines the scalars $f_{ij} \in [-1,1]$. We need to solve $F(s,t) = 0$ for $(s,t) \in [0,1]$. The solution set can be empty or can contain points, line segments and/or hyperbolic curves.

The bilinear function can be rewritten as

$$\begin{aligned} F(s,t) &= f_{00} + (f_{10} - f_{00})s + (f_{01} - f_{00})t + (f_{11} - f_{10} - f_{01} + f_{00})st \\ &= g_{00} + g_{10}s + g_{01}t + g_{11}st \end{aligned} \tag{5}$$

where the last equation defines the g_{ij}. If $g_{11} \neq 0$, then $F(s,t)$ is a quadratic function that can be written as

$$F(s,t) = \frac{(g_{00}g_{11} - g_{10}g_{01}) + (g_{01} + g_{11}s)(g_{10} + g_{11}t)}{g_{11}} \tag{6}$$

Define $\Delta = g_{00}g_{11} - g_{10}g_{01} = f_{00}f_{11} - f_{10}f_{01}$. If $\Delta = 0$, the function is separable,

$$F(s,t) = \frac{(g_{01} + g_{11}s)(g_{10} + g_{11}t)}{g_{11}} \tag{7}$$

The solution set for $F(s,t) = 0$ for $(s,t) \in \mathbb{R}^2$ consists of two lines, one defined by $\hat{s} = -g_{01}/g_{11}$ and one defined by $\hat{t} = -g_{10}/g_{11}$. If $\Delta \neq 0$, the solution set consists of two hyperbolic curves. We may solve for either variable. Solving for s and for t,

$$s = -\frac{g_{00} + g_{01}t}{g_{10} + g_{11}t}, \quad t = -\frac{g_{00} + g_{10}s}{g_{01} + g_{11}s} \tag{8}$$

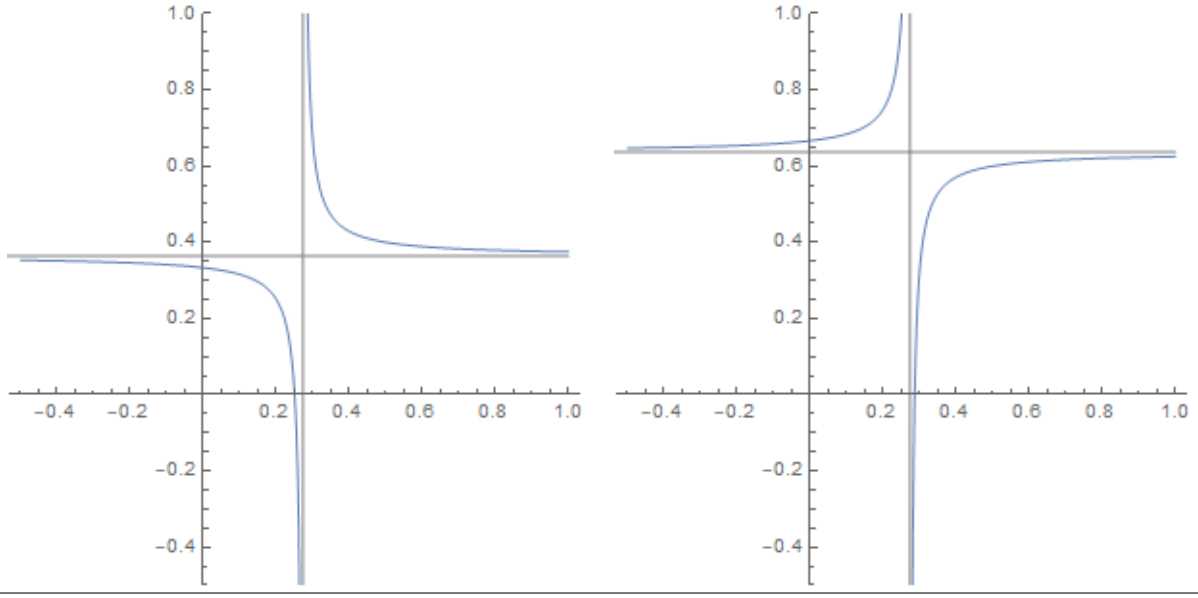
Figure 3 shows the graphs of the hyperbolic curves.

Figure 3. Plots of level curves for two bilinear functions.

The left image is the plot of the level curves when $f_{00} = 1$, $f_{10} = -2$, $f_{01} = -3$ and $f_{11} = 5$. The corresponding numbers are $g_{00} = 1$, $g_{10} = -3$, $g_{01} = -4$ and $g_{11} = 11$ with $\Delta = -1$. The horizontal asymptote is $\hat{t} = 4/11$ and the vertical asymptote is $\hat{s} = 3/11$.

The right image is the plot of the level curves when $f_{00} = -2$, $f_{10} = 1$, $f_{01} = 5$ and $f_{11} = -3$. The corresponding numbers are $g_{00} = -2$, $g_{10} = 3$, $g_{01} = 7$ and $g_{11} = -11$ with $\Delta = 1$. The horizontal asymptote is $\hat{t} = 7/11$ and the vertical asymptote is $\hat{s} = 4/11$.

In both images, the sign of $F(s,t)$ is positive in the region bounded by the two hyperbolic curves and that contains the asymptotes. The sign of $F(s,t)$ in the other two regions is negative.



The classification of the solution set to $F(s,t) = 0$ for $(s,t) \in [0,1]^2$ is based on analyzing the signs of f_{ij} at the corners of the domain. This is effectively an algorithm that arises in 2D image processing, where you want to extract 0-valued level curves of an image stored on a 2D raster. Figure 4 shows the general configuration for the domain and the corners to which the f_{ij} are assigned.

Figure 4. The domain of $F(s,t)$ with corners marked with the function values at those corners.

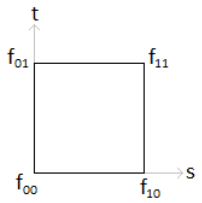
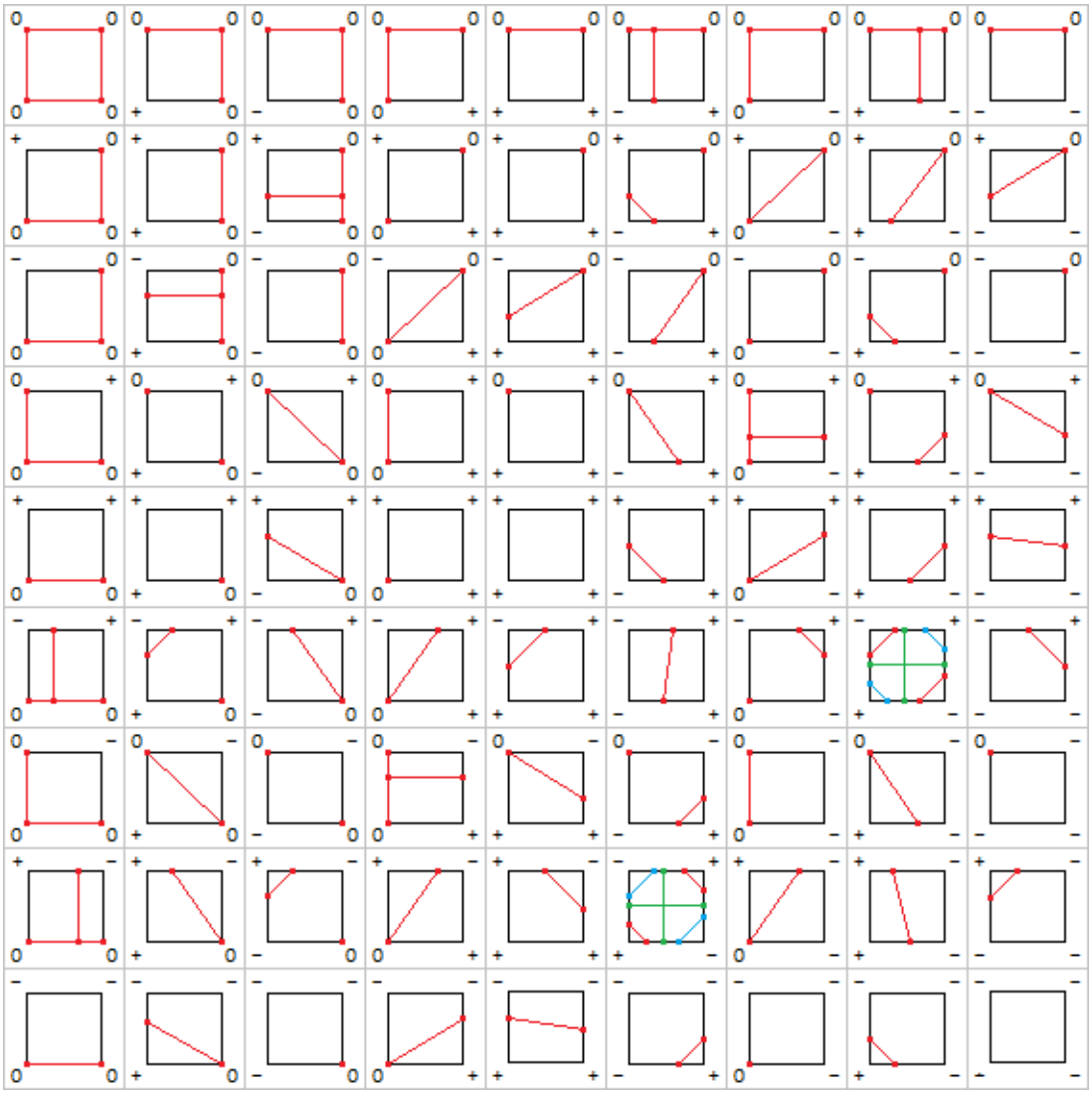


Figure 5 shows the level sets (drawn in red) for the 81 possible sign combinations for the f_{ij}. The horizontal and vertical line segments are exactly that for level sets. The nonhorizontal and nonvertical line segments are for simplicity in visualization. The level curves are usually hyperbolic curves, although several configurations can lead to line segments based on the relative values of the f_{ij} for those configurations.

Figure 5. The 81 possible sign configurations and the corresponding level sets.



The configurations are listed in a specific order based on the signs. In the implementation, 8 bits are used to store the 3 signs with 2 bits per sign as $b_7b_6b_5b_4b_3b_2b_1b_0$. The sign of f_{00} is represented by b_1b_0. If the sign is 0, then $b_1b_0 = 00$. If the sign is $+1$, then $b_1b_0 = 01$. If the sign is -1, then $b_1b_0 = 10$. The bit pattern 11 is unused. Similarly, b_3b_2 represent the sign of f_{10}, b_5b_4 represent the sign of f_{01} and b_7b_6 represent the sign of f_{11}. The 8-bit index is used as a lookup into a table of 256 function pointers, where only 81 of those pointers are not null. This avoids the somewhat complicated nesting logic use if-then-else statements applied to the signs of the f_{ij}.

Two configurations in Figure 5 have level sets with three colors (red, blue, green). The set that is used depends on whether Δ is positive or negative (either the red or blue set is chosen) or whether Δ is zero (the

green set is chosen).

In summary, given two edges of the polyhedron, the f_{ij} are computed as the dot products of pairs of normals. The signs of the f_{ij} are used to form the 8-bit index to look up the correct function to process that configuration. The level set for the configuration is searched to locate the pair of normals $\mathbf{N}(s)$ and $\mathbf{M}(t)$ that lead to the minimum-volume bounding box for all pairs in the level set. This box is minimum only for that pair of edges. All pairs of edges are processed to find the global minimum-volume box for the polyhedron.

4 Computational Considerations

4.1 Locating Extreme Vertices

Given two unit-length normal vectors $\mathbf{U}_0$ and $\mathbf{U}_1$ that are perpendicular and are normals for supporting planes for the pair of edges, the final candidate box face has normal $\mathbf{U}_2 = \mathbf{U}_0 \times \mathbf{U}_1$. The smallest box with these axes and containing the polyhedron vertices can be found by iterating over all vertices and computing dot products. Listing 1 illustrates.

Listing 1. Exhaustive algorithm to iterate over all polyhedron vertices to find the extreme projections along the axes. In the code, the first vertex in the list of polyhedron vertices is subtracted from all vertices (to help with robustness when Real is a floating-point type), so vertex[0] is the zero vector. Therefore, the initial pmin and pmax are valid extreme candidates.

```
Real GetExtremes(
    std::array<Vector3<Real>, 3> const& axis,
    std::vector<Vector3<Real>> const& vertex,
    Vector3<Real>& pmin, Vector3<Real>& pmax)
{
    pmin = { 0, 0, 0 };
    pmax = { 0, 0, 0 };
    for (size_t i = 0; i < vertex.size(); ++i)
    {
        for (size_t j = 0; j < 3; ++j)
        {
            Real dot = Dot(axis[j], vertex[i]);
            pmin[j] = std::min(pmin[j], dot);
            pmax[j] = std::max(pmax[j], dot);
        }
    }
    Real volume = (pmax[0] - pmin[0]) * (pmax[1] - pmin[1]) * (pmax[2] - pmin[2]);
    return volume;
}
```

This algorithm is $O(n)$ for a polyhedron with n vertices.

A better choice is to create a data structure that supports an $O(\log n)$ query for locating the extreme points in a specified direction. The Geometric Tool class ExtremalQuery3BSP supposedly implements such a data structure (using binary space partitioning trees on the unit sphere) according to Extremal Queries using BSP Trees, but during my experiments with the minimum-volume box code, the query appears not to work. The end-to-end test for the BSP tree approach is the GTE/Samples/Geometrics/ExtremalQuery sample application, and this appears to work correctly. Until I can investigate this problem, I implemented a query algorithm that is faster than the exhaustive search.

A vertex-edge-triangle mesh data structure is created for the convex polyhedron. The vertex stores adjacency information including adjacent vertices, adjacent edges and adjacent triangles. To locate the extreme value in a specified direction, start with any vertex of the polyhedron and compute its projection onto the direction vector (a dot product). Search the adjacent vertices and compute their dot products. Select the adjacent vertex with the smallest dot product and repeat the algorithm on it. The iteration repeats until the current vertex has no adjacent vertices with a larger dot product. The algorithm is essentially a hill-climbing algorithm that finds a path of increasing dot products from the initial vertex to the extreme vertex. I used the Geometric Tools class VETManifoldMesh for the mesh data structure. During the experiments, profiling showed that the adjacent-vertex lookups was relatively expensive because of the underlying std::set iterations. To avoid this, after building the mesh data structure, I created a pair of arrays, one storing all adjacent indices in contiguous memory for the vertices in the order of their vertex index. The other array maps the vertex index to the location in the first array where its adjacent vertex indices live. This turns out to be quite fast for computing extreme points.

The code is shown in Listing 2.

Listing 2. The search algorithm for the extreme vertex in a specified direction. The index of the vertex (vMax) and the associated dot product (dMax) are both returned from the function.

```
size_t GetExtreme(Vector3<Real> const& direction, Real& dMax)
{
    size_t vMax = 0;
    dMax = Dot(direction, mVertices[vMax]);
    for (size_t i = 0; i < mNumVertices; ++i)
    {
        size_t vLocalMax = vMax;
        ComputeType dLocalMax = dMax;
        size_t const* adjacent = &mAdjacentPool[mVertexAdjacent[vMax]];
        size_t numAdjacent = *adjacent++;
        for (size_t j = 1; j <= numAdjacent; ++j)
        {
            size_t vCandidate = *adjacent++;
            ComputeType dCandidate = Dot(direction, mVertices[vCandidate]);
            if (dCandidate > dLocalMax)
            {
                vLocalMax = vCandidate;
                dLocalMax = dCandidate;
            }
        }
        if (vMax != vLocalMax)
        {
            vMax = vLocalMax;
            dMax = dLocalMax;
        }
        else
        {
            break;
        }
    }
    return vMax;
}

// The code is used as follows for a pair of edges (E0,E1). Because
// we know the edge is on a supporting box face, we automatically
// know the minimum projection value.
Vector3<Real> pmin, pmax;
std::array<size_t, 3> minSupportIndex, maxSupportIndex;

minSupportIndex[0] = index of a vertex endpoint of E0;
pmin[0] = Dot(axis[0], mVertices[minSupportIndex[0]]);
maxSupportIndex[0] = GetExtreme(axis[0], pmax[0]);

minSupportIndex[1] = index of a vertex endpoint of E1;
```

```
pmin[1] = Dot(axis[1], mVertices[minSupportIndex[1]]);
maxSupportIndex[1] = GetExtreme(axis[1], pmax[1]);

// We know axis[2] = Cross(axis[0], axis[1]), but we must find
// extreme in both the axis[2] and the -axis[2] directions.
minSupportIndex[2] = GetExtreme(-axis[2], pmin[2]);
pmin[2] = -pmin[2];
maxSupportIndex[2] = GetExtreme(axis[2], pmax[2]);
```

The polyhedron vertices are stored in mVertices and there are mNumVertices of them. The mAdjacentPool is a std::vector<size_t> that stores a block of values per vertex. The first element of the block is the number of adjacent vertices. The remaining elements of the block are the indices for the adjacent vertices of the vertex. The mVertexAdjacent is a std::vector<size_t> that stores the location in mAdjacentPool for a vertex of interest. In the code, vMax is a vertex index and mVertexAdjacent[vMax] is the index of the adjacent-vertex information in mAdjacentPool for the vertex mVertex[vMax].

4.2 Error-Free Computation of Signs and Volumes

If floating-point arithmetic is used, it is possible that the computed sign of f_{ij} does not match the theoretical value. This is unavoidable when the inner-pointing triangle normals are normalized to be unit length. The length computation involves a square root operation, which generally has rounding errors. Instead, the polyhedron vertices can be converted to rational form and the non-unit-length normal vectors can be computed exactly via cross products of edge directions. The normalization is skipped. The parameterized vectors in equations (2) and (3) are replaced by

$$\mathbf{N}(s) = (1-s)\mathbf{N}_0 + s\mathbf{N}_1, \quad \mathbf{M}(t) = (1-t)\mathbf{M}_0 + t\mathbf{M}_1 \tag{9}$$

The $\mathbf{N}_i$ and $\mathbf{M}_j$ are not necessarily unit length.

Each f_{ij} involves a dot product of non-unit-length normals, and these are not generally the same numbers as what you obtain using the normalized normals and floating-point arithmetic. However, the solution set for $F(s,t) = \mathbf{N}(s) \cdot \mathbf{M}(t) = 0$ does match that if the normal vectors were unit length. The idea is clear geometrically: If $\mathbf{N}(s) \cdot \mathbf{M}(t) = 0$ for non-unit-length normals, then $\mathbf{N}(s)/|\mathbf{N}(s)| \cdot \mathbf{M}(t)/|\mathbf{M}(t)| = 0$.

Therefore, we can compute the exact signs of f_{ij} which leads to the selection of the correct configuration of level curves in Figure 5.

Volumes of the boxes can also be computed exactly when using non-unit-length normals and rational arithmetic. Using real numbers, if $\mathbf{U}_i$ are unit-length box axes that are mutually perpendicular, the smallest box extreme points are $\mathbf{P}_{\min}$ and $\mathbf{P}_{\max}$ as computed in Listing 2,

$$\mathbf{P}_{\min} = (\mathbf{U}_0 \cdot \mathbf{V}_{i_0}, \mathbf{U}_1 \cdot \mathbf{V}_{i_1}, \mathbf{U}_2 \cdot \mathbf{V}_{i_2}), \mathbf{P}_{\max} = (\mathbf{U}_0 \cdot \mathbf{V}_{j_0}, \mathbf{U}_1 \cdot \mathbf{V}_{j_1}, \mathbf{U}_2 \cdot \mathbf{V}_{j_2}) \tag{10}$$

where (i_0, i_1, i_2) are the minimum support indices and (j_0, j_1, j_2) are the maximum support indices. The box volume is the product of the components of $\mathbf{P}_{\max} - \mathbf{P}_{\min}$,

$$\begin{aligned} \nu &= (\mathbf{P}_{\max}(0) - \mathbf{P}_{\min}(0))(\mathbf{P}_{\max}(1) - \mathbf{P}_{\min}(1))(\mathbf{P}_{\max}(2) - \mathbf{P}_{\min}(2)) \\ &= (\mathbf{U}_0 \cdot (\mathbf{V}_{j_0} - \mathbf{V}_{i_0}))(\mathbf{U}_1 \cdot (\mathbf{V}_{j_1} - \mathbf{V}_{i_1}))(\mathbf{U}_2 \cdot (\mathbf{V}_{j_2} - \mathbf{V}_{i_2})) \end{aligned} \tag{11}$$

Using non-unit-length normals $\mathbf{N}_i$, we have $\mathbf{U}_i = \mathbf{N}_i/|\mathbf{N}_i|$. The volume equation becomes

$$\nu = \frac{(\mathbf{N}_0 \cdot (\mathbf{V}_{j_0} - \mathbf{V}_{i_0}))(\mathbf{N}_1 \cdot (\mathbf{V}_{j_1} - \mathbf{V}_{i_1}))(\mathbf{N}_2 \cdot (\mathbf{V}_{j_2} - \mathbf{V}_{i_2}))}{|\mathbf{N}_0||\mathbf{N}_1||\mathbf{N}_2|} \tag{12}$$

We know that $\mathbf{N}_2 = \mathbf{N}_0 \times \mathbf{N}_1$ and that $\mathbf{N}_0$ and $\mathbf{N}_1$ are perpendicular, so $|\mathbf{N}_2| = |\mathbf{N}_0||\mathbf{N}_1|$. The volume equation is finally

$$\nu = \frac{(\mathbf{N}_0 \cdot (\mathbf{V}_{j_0} - \mathbf{V}_{i_0}))(\mathbf{N}_1 \cdot (\mathbf{V}_{j_1} - \mathbf{V}_{i_1}))(\mathbf{N}_2 \cdot (\mathbf{V}_{j_2} - \mathbf{V}_{i_2}))}{|\mathbf{N}_2|^2} \tag{13}$$

which can be computed without error using rational arithmetic.

4.3 The Minimizer for $F(s,t)$

As indicated, the level curves of $F(s,t) = 0$ must be searched for the minimum volume box for the specified pair of edges and corresponding hinge normals. The Geometric Tools implementation uses a simple sampling scheme, where the parameter for a level curve is sampled 2^p times for user-specified p. However, the minimizer functions are virtual and can be overridden by a derived class to allow for a user-created minimization algorithm.

If p is large when using rational arithmetic, the computational time is excessive. One remedy is already built into the MinimumVolumeBox3 class. The maximum number of bits is known for computing the expressions in the algorithm, so a type BSNumber<UIntegerFP<N>> is used where N is hard-coded (and selected via a type traits mechanism). Another remedy is that the edge-pair processing is naturally parallel. The caller can specify the number of CPU threads to use in the computations.

References

[1] H. Freeman and R. Shapira. Determining the minimum-area encasing rectangle for an arbitrary closed curve. In *Communications of the ACM*, volume 18, pages 409–413, New York, NY, July 1975.

[2] Joseph O'Rourke. Finding minimal enclosing boxes. *International Journal of Computer & Information Sciences*, 14(3):183–199, 1985.