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: February 14, 2019

Contents

1 Introduction

Given a finite set of 3D points, we want to compute the minimum-volume box that contains the points. The bos 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 3D points.

The minimum-volume box has a face coincident with a hull face or has three mutually orthogonal edges coincident with three hull edges that (of course) are mutually orthogonal [1]. The implementation consists of two parts. The first part processes all the hull faces and the second part processes all triples of mutually orthogonal edges. As each candidate box is computed, the minimum-volume box is updated to that candidate when the candidate's volume is smaller.

2 Processing Hull Faces

For a specified hull face, the minimum-volume box with a face coincident to the hull face can be computed by projecting the hull vertices to the plane of the hull face and to a normal line of that plane. Let the normal line have unit-length normal $\mathbf{N}$ which points to the side of the plane on which the hull lies.

We do not have to project all hull vertices onto the plane. The projection onto the plane is necessary only for an extreme polyline of hull edges. Each edge of this polyline is shared by two hull faces. Let the face normals be $\mathbf{N}_0$ and $\mathbf{N}_1$, both pointing inside the hull, where $\mathbf{N} \cdot \mathbf{N}_0 < 0 \le \mathbf{N} \cdot \mathbf{N}_1$ or $\mathbf{N} \cdot \mathbf{N}_1 < 0 \le \mathbf{N} \cdot \mathbf{N}_0$. The extreme polyline therefore is the terminator for the convex hull faces whose inner-pointing normals form an obtuse angle with the plane normal $\mathbf{N}$. The projection of the extreme polyline onto the plane is a convex polygon $\mathcal{P}$.

The projection of the hull vertices onto the normal line is a line segment of length h. This measurement is the height of the bounding box of the hull. The base of the bounding box has area A which we want to minimize to obtain the minimum volume $V_{\min} = hA_{\min}$. To accomplish this, we can use the algorithm for computing the minimum-area rectangle of a convex polygon living in a plane. The algorithm uses the method of rotating calipers, which is described in a document at our website and includes implementation details, Minimum-Area Rectangle Containing a Set of Points, referred to throughout this document as MAR2.

The rotating calipers algorithm is typically described as if you have real-valued arithmetic; that is, all computations involved are exact. On a computer, we have to be wary of numerical rounding errors when attempting to implement the algorithm using floating-point arithmetic. In MAR2, we used arbitrary precision arithmetic (rational arithmetic) to compute the minimum rectangle exactly. However, the algorithm depends on selecting an edge with (not necessarily unit-length) direction $\mathbf{U} = (u_0, u_1)$ and using an orthogonal vector of the same length, $\mathbf{U}^\perp = (-u_1, u_0)$, to avoid normalization of vectors by an inverse square root operation. The fact that $|\mathbf{U}| = |\mathbf{U}^\perp|$ was sufficient to push through the exact arithmetic.

In the 3D case, the projection of a point onto a plane can be done using exact arithmetic. If $\mathbf{P}_0$ is a point in the plane and $\mathbf{P}_1$ is a point off the plane, and if $\mathbf{N}$ is a normal vector for the plane (not necessarily unit length), the projection of $\mathbf{P}_1$ is

$$\mathbf{Q}_1 = \mathbf{P}_1 - \frac{\mathbf{N}}{|\mathbf{N}|} \cdot (\mathbf{P}_1 - \mathbf{P}_0) \frac{\mathbf{N}}{|\mathbf{N}|} = \mathbf{P}_1 - \frac{\mathbf{N} \cdot (\mathbf{P}_1 - \mathbf{P}_0)}{|\mathbf{N}|^2} \tag{1}$$

The right-hand side can be computed exactly using arbitrary precision arithmetic. To apply the 2D rotating calipers algorithm, for an edge of the convex polygon $\mathcal{P}$ (the projection of the extreme polyline), the direction is $\mathbf{U}$, which is not necessarily unit length, and is in the plane, so $\mathbf{N} \cdot \mathbf{U} = 0$. To obtain a vector in the plane perpendicular to $\mathbf{U}$, we can use $\mathbf{U}^{\perp} = \mathbf{N} \times \mathbf{U}$. However, $|\mathbf{U}^{\perp}| = |\mathbf{N}||\mathbf{U}|$, which is not equal to $|\mathbf{U}|$ when $|\mathbf{N}| \neq 1$. In the attempt to use exact arithmetic, we can only build normals $\mathbf{N}$ using cross products of two linearly independent edge directions, but such vectors are generally not unit length.

2.1 Comparing Areas

We need to modify the rotating calipers algorithm differently for 3D than we did for 2D. In fact, we do not have to explicitly project the extreme polyline to $\mathcal{P}$. We can work directly with the polyline and generate for each edge a set of vectors, not necessarily unit length. The edge direction is the difference of endpoints, say, $\mathbf{E}$. We compute $\mathbf{U}_1 = \mathbf{N} \times \mathbf{E}$ and $\mathbf{U}_0 = \mathbf{U}_1 \times \mathbf{N}$. The vectors $\mathbf{N}$, $\mathbf{U}_0$, and $\mathbf{U}_1$ are mutually perpendicular. The lengths are $|\mathbf{N}|$, $|\mathbf{N}||\mathbf{E}|$, and $|\mathbf{N}|^2|\mathbf{E}|$, respectively. $\mathbf{U}_0$ acts as the convex polygon edge direction and $\mathbf{U}_1$ acts as a direction perpendicular to $\mathbf{U}_0$ that is also in the plane of the convex polygon. The difference of the extreme points of the extreme polyline in the direction $\mathbf{U}_0$ is named $\mathbf{\Delta}_0$. The difference of the extreme points of the extreme polyline in the direction $\mathbf{U}_1$ is named $\mathbf{\Delta}_1$. The area of the rectangle that has an edge of $\mathcal{P}$ coincident with $\mathbf{E}$ is

$$A = \left(\frac{\mathbf{U}_0}{|\mathbf{U}_0|} \mathbf{\Delta}_0\right) \left(\frac{\mathbf{U}_1}{|\mathbf{U}_1|} \mathbf{\Delta}_1\right) = \frac{(\mathbf{U}_0 \cdot \mathbf{\Delta}_0)(\mathbf{U}_1 \cdot \mathbf{\Delta}_1)}{|\mathbf{N}|^3|\mathbf{E}|^2} \tag{2}$$

The numerator can be computed exactly using arbitrary precision arithmetic. The denominator has an odd power of the length of $|\mathbf{N}|$, which requires a square root operation that leads to approximation error. But $|\mathbf{N}|$ is a constant regardless of the edge of $\mathcal{P}$ we are processing. Rather than comparing areas as we rotate the calipers for $\mathcal{P}$, we can instead compare the values of

$$A' = \frac{(\mathbf{U}_0 \cdot \mathbf{\Delta}_0)(\mathbf{U}_1 \cdot \mathbf{\Delta}_1)}{|\mathbf{N}|^2|\mathbf{E}|^2} \tag{3}$$

The denominator of this fraction can be computed exactly. Effectively, we are comparing values of $|\mathbf{N}|A$ without ever having to compute $|\mathbf{N}|$ directly.

2.2 Comparing Volumes

Once we have selected the minimum-area rectangle for $\mathcal{P}$, we will need to compute the volume of the bounding box in order to update the current minimum-volume box. Let the difference of the extreme points in the direction $\mathbf{N}$ be named $\mathbf{\Delta}$. The volume of the bounding box is

$$V = A\left(\frac{\mathbf{N}}{|\mathbf{N}|} \cdot \mathbf{\Delta}\right) = \frac{(\mathbf{U}_0 \cdot \mathbf{\Delta}_0)(\mathbf{U}_1 \cdot \mathbf{\Delta}_1)(\mathbf{N} \cdot \mathbf{\Delta})}{|\mathbf{N}|^4|\mathbf{E}|^2} \tag{4}$$

The numerator and denominator can be computed exactly using arbitrary precision arithmetic; therefore, we can update the minimum-volume box as we process each face of the hull without any approximation or rounding errors.

2.3 Comparing Angles

In the 2D rotating calipers algorithms, we needed to compute the minimum angle between a convex polygon edge and the bounding rectangle edge from which the polygon edge emanated. We arranged to search for the minimum without computing the angles, instead using quantities that are equivalently ordered with the angles. The same is true in the 3D setting. In this case $0 \geq \theta_0 < \theta_1 \leq \pi/2$ if and only if $\sin^2\theta_0 < \sin^2\theta_1$. If $\mathbf{D}$ is the direction for a rectangle edge, $\mathbf{E}$ is the direction for a convex polygon edge, and $\mathbf{U}_0 = (\mathbf{N} \times \mathbf{E}) \times \mathbf{N}$, all not necessarily unit length vectors, then $\mathbf{D}$ and $\mathbf{U}_0$ are in the projection plane and have an angle θ between them. From basic geometry,

$$|\mathbf{D} \times \mathbf{U}_0| = |\mathbf{D}||\mathbf{U}_0| \sin\theta \tag{5}$$

which can be solved for

$$\sin^2\theta = \frac{|\mathbf{D} \times \mathbf{U}_0|^2}{|\mathbf{D}|^2|\mathbf{U}_0|^2} \tag{6}$$

The numerator and denominator can be computed exactly using arbitrary precision arithmetic.

3 Processing Hull Edges

For n edges, the algorithm is theoretically $O(n^3)$ because we have three nested loops searching for three mutually orthogonal edges. This sounds expensive, but in practice the constant of the asymptotic order analysis is small because in most cases we compute a dot product between two edge directions, find out the edges are not orthogonal, and continue on with the next loop iteration. The pseudocode is

```
array<Vertices> vertices(m);  // m vertices in the hull
array<Edges> edges(n);  // n edges in the hull
Vector3 U[3], sqrlen;
for (int i2 = 0; i2 < n; ++i2)
{
    U[2] = vertices[edges[i2].V[1]] - vertices[edges[i2].V[0]];
    for (int i1 = i2 + 1; i1 < n; ++i1)
    {
        U[1] = vertices[edges[i1].V[1]] - vertices[edges[i1].V[0]];
        if (Dot(U[1], U[2]) != 0)
        {
            continue;
        }
        sqrlen[1] = Dot(U[1], U[1]);

        for (int i0 = i1 + 1; i0 < n; ++i0)
        {
            U[0] = vertices[edges[i0].V[1]] - vertices[edges[i0].V[0]];
            if (Dot(U[0], U[1]) != 0)
            {
                continue;
            }
            sqrlen[0] = Dot(U[0], U[0]);

            // The three edges are mutually orthogonal.  To support exact
            // arithmetic for volume computation, replace U[2] by a parallel
            U[2] = Cross(U[0], U[1]);
            sqrlen[2] = sqrlen[0] * sqrlen[1];

            // Compute the extremes of the scaled projections onto the axes.
            // using vertices(0) as the origin.
            Vector3 umin(0,0,0), umax(0,0,0);  // extremes in U[] directions
            for (int j = 0; j < m; ++j)
            {
```

```
                Vector3 diff = vertices[j] - vertices[0];
                for (int k = 0; k < 3; ++k)
                {
                    Numeric dot = Dot(diff, U[k]);
                    if (dot < umin[k])
                    {
                        umin[k] = dot;
                    }
                    else if (dot > umax[k])
                    {
                        umax[k] = dot;
                    }
                }
            }

            // Compute the volume of the box.
            Vector3 delta = umax - umin;
            Numeric volume = (delta[0] / sqrlen[0]) * (delta[1] / sqrlen[1]) * delta[2];

            // Compare the volume to the current candidate's volume for the
            // minimum-volume box and update if necessary.
            <code goes here>;
        }
    }
}
```

4 Conversion to a Floating-Point Box

Our implementation of the rotating calipers algorithm tracks the minimum-volume box via the data structure

```
struct Box
{
    Vector3<ComputeType> P, U[3];
    ComputeType sqrLenU[3], range[3][2], volume;
};
```

The ComputeType is a rational type. The point P acts as an origin and the box axis directions (not necessarily unit length) are U[]. The squared lengths of the axis directions are sqrLenU[]. The range member stores the extreme values for the three directions, the minimum stored in range[][0] and the maximum stored in range[][1].

The conversion to the floating-point representation of the box has some loss of precision, but this occurs at the very end.

```
OrientedBox3<Real> itMinBox;  // input-type minimum-volume box
ComputeType half(0.5), one(1);

// Compute the box center exactly.
for (int i = 0; i < 3 ++i)
{
    ComputeType average =
        half * (minBox.range[i][0] + minBox.range[i][1]);
    center += (average / minBox.sqrLenU[i]) * minBox.U[i];
}

// Compute the squared extents exactly.
Vector3<ComputeType> sqrExtent;
for (int i = 0; i < 3; ++i)
{
    sqrExtent[i] = half * (minBox.range[i][1] - minBox.range[i][0]);
    sqrExtent[i] *= sqrExtent[i];
    sqrExtent[i] /= minBox.sqrLenU[i];
}

for (int i = 0; i < 3; ++i)
```

```
    {
        itMinBox.center[i] = (InputType)center[i];
        itMinBox.extent[i] = sqrt((InputType)sqrExtent[i]);

        // Before converting to floating-point, factor out the maximum
        // component using ComputeType to generate rational numbers in a
        // range that avoids loss of precision during the conversion and
        // normalization.
        Vector3<ComputeType> const& axis = minBox.U[i];
        ComputeType cmax = std::max(std::abs(axis[0]), std::abs(axis[1]));
        cmax = std::max(cmax, std::abs(axis[2]));
        ComputeType invCMax = one / cmax;
        for (int j = 0; j < 3; ++j)
        {
            itMinBox.axis[i][j] = (InputType)(axis[j] * invCMax);
        }
        Normalize(itMinBox.axis[i]);
    }

    // Quantity accessible through the get-accessor in the public interface.
    volume = (Real)minBox.volume;
```

The source code for the implementation is in the file GteMinimumVolumeBox3.h.

References

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