Distance from an Oriented Box to a Cone Frustum

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

Created: May 22, 2021

Contents

1 Introduction **2**

2 Definitions for the Objects **2**

2.1 Definition for an Oriented Box 2

2.2 Definition for a Cone Frustum 2

3 The Quadratically Constrained Quadratic Program **3**

4 The Linearly Constrained Quadratic Programs **5**

4.1 Setting Up the CQP for a Box and a Cross Section 5

4.2 Searching for the Minimum Distance Among All Cross Sections 5

5 Distance between an Oriented Box and a Cone Frustum **7**

1 Introduction

This document describes an algorithm for estimating the distance from an oriented box to a cone frustum in 3D. For heights h measured along the cone axis direction, $0 \le h_{\min} \le h \le h_{\max} < +\infty$; that is, the maximum height is finite. A finite cone occurs When $h_{\min} = 0$. A cone frustum occurs when $h_{\min} > 0$. I refer to both objects as a cone frustum.

The theoretical distance between the box and cone frustum can be formulated as a *quadratically constrained quadratic program* (QCQP). Such problems are known to be NP-hard. The formulation for the box-cone distance query QCQP is given in Section 3. If you have a quality solver for a *semidefinite program* (SDP), you can solve the QCQP directly.

I provide an alternate approach. The algorithm for estimating the distance between the two objects is based on representing the cone frustum as a union of planar cross sections where the planes contain the cone vertex and the cone axis-direction. The cross sections are triangles when $h_{\min} = 0$ or convex quadrilaterals when $h_{\min} > 0$. Given one cross section, the others are obtained by rotating the plane about the cone axis direction. An angle ψ is associated with each plane, and each query assigns a distance from the box to the cross section. The result is a function $\delta(\psi)$ for $\psi \in (-\pi/2, \pi/2]$ that can be minimized using a numerical method.

The distance between a box and a cross section is either a box-triangle query or a box-quadrilateral. Either query can be formulated as a *convex quadratic program* (CQP) that can be solved by reformulating it as a *linear complementarity problem* (LCP). A discussion for this type of problem in the context of distance and intersection algorithms is provided in Sections 7 and 8 of my book [1].

2 Definitions for the Objects

2.1 Definition for an Oriented Box

An *oriented box* has a center $\boldsymbol{C}$, unit-length axis directions $\boldsymbol{U}_i$ that form a right-handed orthonormal set and extents $e_i > 0$ that measure half the edge length in each dimension i with $0 \le i \le 2$. A point $\boldsymbol{P}$ in the box may be written as

$$\boldsymbol{P} = \boldsymbol{C} + \sum_{i=0}^{2} y_i \boldsymbol{U}_i = \boldsymbol{C} + U\boldsymbol{Y} \tag{1}$$

where $y_i = \boldsymbol{U}_i \cdot (\boldsymbol{P} - \boldsymbol{C})$ and $|y_i| \le e_i$. The matrix $U = [\boldsymbol{U}_0\, \boldsymbol{U}_1\, \boldsymbol{U}_2]$ is a rotation matrix whose columns are the box axis directions. The vector $\boldsymbol{Y}$ is a column vector whose components are the y_i. Define the vector $\boldsymbol{e}$ to be the column vector whose components are the e_i. In coordinate-free notation, $-\boldsymbol{e} \le \boldsymbol{Y} \le \boldsymbol{e}$.

2.2 Definition for a Cone Frustum

An *infinite single-sided solid cone* has a vertex $\boldsymbol{V}$, an axis ray whose origin is $\boldsymbol{V}$ and unit-length direction is $\boldsymbol{D}$, and an acute cone angle $\theta \in (0, \pi/2)$. A point $\boldsymbol{P}$ is inside the cone when the angle between $\boldsymbol{D}$ and $\boldsymbol{P} - \boldsymbol{V}$ is in $[0, \theta]$. Algebraically, the containment is defined by

$$\boldsymbol{D} \cdot \frac{(\boldsymbol{X} - \boldsymbol{V})}{|\boldsymbol{X} - \boldsymbol{V}|} \ge \cos(\theta) \tag{2}$$

when $\boldsymbol{X} \neq \boldsymbol{V}$. Equivalently, the containment is defined by

$$\boldsymbol{D} \cdot (\boldsymbol{X} - \boldsymbol{V}) \geq |\boldsymbol{X} - \boldsymbol{V}| \cos(\theta) \tag{3}$$

which includes the case $\boldsymbol{X} = \boldsymbol{V}$. Finally, we can avoid computing square roots in the implementation by squaring the dot-product equation to obtain a quadratic equation and requiring that only points above the supporting plane of the single-sided cone be considered. The definition is

$$Q(\boldsymbol{X}) = (\boldsymbol{D} \cdot (\boldsymbol{X} - \boldsymbol{V}))^2 - \gamma^2 |\boldsymbol{X} - \boldsymbol{V}|^2 = (\boldsymbol{X} - \boldsymbol{V})^{\mathsf{T}} \mathcal{P} (\boldsymbol{X} - \boldsymbol{V}) \geq 0, \;\; \boldsymbol{D} \cdot (\boldsymbol{X} - \boldsymbol{V}) \geq 0 \tag{4}$$

where $\gamma = \cos\theta$, $\mathcal{P} = \boldsymbol{D}\boldsymbol{D}^{\mathsf{T}} - \gamma^2 I$ is a symmetric 3×3 projection matrix and I is the 3×3 identity matrix.

The *height* of a point $\boldsymbol{X}$ relative to the cone is the length of the projection of $\boldsymbol{X} - \boldsymbol{V}$ onto the $\boldsymbol{D}$-axis, namely, $h = \boldsymbol{D} \cdot (\boldsymbol{X} - \boldsymbol{V})$. The infinite cone can be truncated, either with a minimum height or a maximum height or both. The names and height constraints are as follows. For concise naming, I have dropped the term single-sided. Generally, the heights h satisfy $h \in [h_{\min}, h_{\max}]$ with $0 \leq h_{\min} < h_{\max} \leq +\infty$.

- Infinite cone: $h_{\min} = 0$, $h_{\max} = +\infty$.
- Infinite truncated cone: $h_{\min} > 0$, $h_{\max} = +\infty$.
- Finite truncated cone: $h_{\min} = 0$, $h_{\max} < +\infty$.
- Frustum of a cone (or cone frustum): $h_{\min} > 0$, $h_{\max} < +\infty$.

As mentioned previously, I combine the last two cases into $0 \leq h_{\min} < h \leq h_{\max} < +\infty$ and refer to the object as a cone frustum.

The cone frustum is parameterized by

$$\boldsymbol{P} = \boldsymbol{V} + \sum_{i=0}^{2} z_i \boldsymbol{W}_i = \boldsymbol{V} + W\boldsymbol{Z} \tag{5}$$

where $z_i = \boldsymbol{W}_i \cdot (\boldsymbol{P} - \boldsymbol{V})$. The matrix $W = [\boldsymbol{W}_0 \; \boldsymbol{W}_1 \; \boldsymbol{W}_2]$ is a rotation matrix whose last column is $\boldsymbol{W}_2 = \boldsymbol{D}$. The first two columns are arbitrarily chosen so that indeed W is a rotation matrix. The output of the algorithm does not depend theoretically on the choice, although there can be some slight differences in the implementation because of floating-point rounding errors. The components of $\boldsymbol{Z}$ are $z_0 = r\cos\phi$, $z_1 = r\sin\phi$ and $z_2 = h$ with constraints $h_{\min} \leq h \leq h_{\max}$, $0 \leq r \leq h$ and $-\pi < \phi \leq \pi$.

3 The Quadratically Constrained Quadratic Program

A point in the box is $\boldsymbol{B} = \boldsymbol{C} + U\boldsymbol{Y}$ and a point in the cone frustum is $\boldsymbol{P}$. The quadratic function to minimize is half the squared distance between a pair of points,

$$f(\boldsymbol{Y}, \boldsymbol{P}) = \frac{1}{2} |\boldsymbol{B} - \boldsymbol{P}|^2 \tag{6}$$

The QCQP solver requires the components of $\boldsymbol{Y}$ and $\boldsymbol{P}$ to be nonnegative, which they are not always.

A reparameterization for the box is

$$\boldsymbol{B} = \boldsymbol{C} + U\boldsymbol{Y} = (\boldsymbol{C} - U\boldsymbol{e}) + U(\boldsymbol{Y} + \boldsymbol{e}) = \boldsymbol{K} + U\boldsymbol{\xi} \tag{7}$$

The last equality defines $\boldsymbol{K}$ and $\boldsymbol{\xi}$. Observe that $\boldsymbol{0} \le \boldsymbol{\xi} \le 2\boldsymbol{e} = \boldsymbol{\ell}$. The components of $\boldsymbol{\ell}$ are the lengths of the box edges. The components ξ_i will be independent variables for the QCQP, and they are nonnegative as required.

The cone frustum has linear inequality constraints $h_{\min} \le \boldsymbol{D}^{\mathsf{T}}(\boldsymbol{P} - \boldsymbol{V}) \le h_{\max}$. A quadratic inequality constraint is $(\boldsymbol{P} - \boldsymbol{V})^{\mathsf{T}}(\boldsymbol{D}\boldsymbol{D}^{\mathsf{T}} - \gamma^2 I)(\boldsymbol{P} - \boldsymbol{V}) \ge 0$, where $\gamma = \cos\theta$ and $\theta \in (0, \pi/2)$ is the cone angle. To obtain the nonnegativity constraints for $\boldsymbol{P}$, it is sufficient to translate the cone frustum into the first octant. Let $\boldsymbol{T}$ be a vector such that $\boldsymbol{P} - \boldsymbol{T} \ge \boldsymbol{0}$ for all points $\boldsymbol{P}$ in the cone frustum. It is sufficient to compute an axis-aligned bounding box of the finite truncated cone, the box having minimum point $\boldsymbol{T}$. Define $\boldsymbol{\eta} = \boldsymbol{P} - \boldsymbol{T}$ and $\boldsymbol{\Delta} = \boldsymbol{V} - \boldsymbol{T}$. The linear inequality constraints become $h_{\min} \le \boldsymbol{D}^{\mathsf{T}}(\boldsymbol{\eta} - \boldsymbol{\Delta}) \le h_{\max}$ and the quadratic inequality constraint becomes $(\boldsymbol{\eta} - \boldsymbol{\Delta})^{\mathsf{T}}(\boldsymbol{D}\boldsymbol{D}^{\mathsf{T}} - \gamma^2 I)(\boldsymbol{\eta} - \boldsymbol{\Delta}) \ge 0$. The oriented box must also be translated by the same amount; that is, the new box center is $\boldsymbol{K}' = \boldsymbol{K} - \boldsymbol{T}$. The translation does not modify $\boldsymbol{\xi}$.

The QCQP is to minimize the function $f(\boldsymbol{\xi}, \boldsymbol{\eta})$ with inequality constraints,

$$\begin{aligned} &f(\boldsymbol{\xi}, \boldsymbol{\eta}) = \tfrac{1}{2}\left|(\boldsymbol{K} + U\boldsymbol{\xi}) - \boldsymbol{\eta}\right|^2 = \tfrac{1}{2}\begin{bmatrix} \boldsymbol{\xi}^{\mathsf{T}} & \boldsymbol{\eta}^{\mathsf{T}} \end{bmatrix} A \begin{bmatrix} \boldsymbol{\xi} \\ \boldsymbol{\eta} \end{bmatrix} + \boldsymbol{b}^{\mathsf{T}} \begin{bmatrix} \boldsymbol{\xi} \\ \boldsymbol{\eta} \end{bmatrix} + c \\ &\boldsymbol{0} \le \boldsymbol{\xi} \le \boldsymbol{\ell},\ \boldsymbol{\eta} \ge \boldsymbol{0},\ 0 \le \boldsymbol{D}^{\mathsf{T}}(\boldsymbol{\eta} - \boldsymbol{\Delta}) \le h_{\max}, \\ &(\boldsymbol{\eta} - \boldsymbol{\Delta})^{\mathsf{T}}(\boldsymbol{D}\boldsymbol{D}^{\mathsf{T}} - \gamma^2 I)(\boldsymbol{\eta} - \boldsymbol{\Delta}) \ge 0 \end{aligned} \tag{8}$$

where

$$A = \begin{bmatrix} I & -U^{\mathsf{T}} \\ -U & I \end{bmatrix},\ \boldsymbol{b} = \begin{bmatrix} U^{\mathsf{T}}\boldsymbol{K} \\ -\boldsymbol{K} \end{bmatrix},\ c = \frac{1}{2}\boldsymbol{K}^{\mathsf{T}}\boldsymbol{K} \tag{9}$$

I made use of the fact U is a rotation matrix, so $U^{\mathsf{T}}U = I$ (the identity matrix) and $\boldsymbol{\xi}^{\mathsf{T}}U^{T}U\boldsymbol{\xi} = \boldsymbol{\xi}^{\mathsf{T}}\boldsymbol{\xi}$.

If the minimum of f occurs at $(\hat{\boldsymbol{\xi}}, \hat{\boldsymbol{\eta}})$, then $f_{\min} = f(\hat{\boldsymbol{\xi}}, \hat{\boldsymbol{\eta}})$ and the distance between the box and finite truncated cone is $\sqrt{2f_{\min}}$. The corresponding closest points, one from the box and one from the cone, are determined by transforming $\hat{\boldsymbol{\xi}}$ and $\hat{\boldsymbol{\eta}}$ to the original coordinate system. In its most general form, define the 6×1 vector $\boldsymbol{x} = [\boldsymbol{\xi}^{\mathsf{T}}\ \boldsymbol{\eta}^{\mathsf{T}}]^{\mathsf{T}}$; then

$$\begin{aligned} &f(\boldsymbol{x}) = \tfrac{1}{2}\boldsymbol{x}^{\mathsf{T}}A\boldsymbol{x} + \boldsymbol{b}^{\mathsf{T}}\boldsymbol{x} + c \\ &\boldsymbol{x} \ge \boldsymbol{0},\ D\boldsymbol{x} \ge \boldsymbol{\varepsilon},\ \tfrac{1}{2}\boldsymbol{x}^{\mathsf{T}}A_1\boldsymbol{x} + \boldsymbol{b}_1^{\mathsf{T}}\boldsymbol{x} + c_1 \ge 0 \end{aligned} \tag{10}$$

where $A_1 = \boldsymbol{D}\boldsymbol{D}^{\mathsf{T}} - \gamma^2 I$, $\boldsymbol{b}_1 = -A_1\boldsymbol{\Delta}$ and $c_1 = \boldsymbol{\Delta}^{\mathsf{T}}A_1\boldsymbol{\Delta}/2$. The constraint matrix D and the constraint vector $\boldsymbol{e}$ are

$$D = \begin{bmatrix} -I & Z \\ \boldsymbol{0}^{\mathsf{T}} & \boldsymbol{D}^{\mathsf{T}} \\ \boldsymbol{0}^{\mathsf{T}} & -\boldsymbol{D}^{\mathsf{T}} \end{bmatrix},\ \boldsymbol{\varepsilon} = \begin{bmatrix} -\boldsymbol{\ell} \\ \boldsymbol{D}^{\mathsf{T}}\boldsymbol{\Delta} \\ -\left(\boldsymbol{D}^{\mathsf{T}}\boldsymbol{\Delta} + h_{\max}\right) \end{bmatrix} \tag{11}$$

where I is the 3×3 identity matrix and Z is the 3×3 zero matrix.

Implementing a QCQP solver is a nontrivial exercise. As an alternative, I estimate the distance between box and cone frustum by minimizing a function of an angle ψ, each ψ leading to a distance query between a box and a triangle or quadrilateral cross section of the cone frustum. The next section describes the algorithm.

4 The Linearly Constrained Quadratic Programs

4.1 Setting Up the CQP for a Box and a Cross Section

As described previously, the cone frustum is parameterized by

$$\boldsymbol{P} = \boldsymbol{V} + r\cos\phi\boldsymbol{W}_0 + r\sin\phi\boldsymbol{W}_1 + h\boldsymbol{W}_2 \tag{12}$$

with constraints $h_{\min} \le h \le h_{\max}$, $0 \le r \le h\tan\theta$ and $-\pi < \phi \le \pi$. The box is parameterized by

$$\boldsymbol{B} = \boldsymbol{K} + U\boldsymbol{\xi} \tag{13}$$

We can choose a family of planes containing $\boldsymbol{V}$ and $\boldsymbol{D}$ that intersect the cone in a triangle ($h_{\min} = 0$) or in a convex quadrilateral ($h_{\min} > 0$). The construction here is for the quadrilateral, but it applies equally as well to a triangle where two quadrilateral vertices are the same point $\boldsymbol{V}$.

The family has the parameter $\psi \in (-\pi/2, \pi/2]$. Define $s = \sin\psi$, $c = \cos\psi$ and $t = \tan\theta$. The plane of the quadrilateral corresponding to ψ contains $\boldsymbol{V}$ and has unit-length normal $\boldsymbol{N}(\psi) = -s\boldsymbol{W}_0 + c\boldsymbol{W}_1$. Define $\boldsymbol{G}_0 = -tc\boldsymbol{W}_0 - ts\boldsymbol{W}_1 + \boldsymbol{W}_2$ and $\boldsymbol{G}_1 = tc\boldsymbol{W}_0 + ts\boldsymbol{W}_1 + \boldsymbol{W}_2$. The quadrilateral vertices are $\boldsymbol{Q}_{00} = \boldsymbol{V} + h_{\min}\boldsymbol{G}_0$, $\boldsymbol{Q}_{10} = \boldsymbol{V} + h_{\min}\boldsymbol{G}_1$, $\boldsymbol{Q}_{01} = \boldsymbol{V} + h_{\max}\boldsymbol{G}_0$ and $\boldsymbol{Q}_{11} = \boldsymbol{V} + h_{\max}\boldsymbol{G}_1$. A parameterization of the quadrilateral is

$$\boldsymbol{Q} = \boldsymbol{V} + \sum_{i=0}^{1} \eta_i \boldsymbol{G}_i = \boldsymbol{V} + J\boldsymbol{\eta} \tag{14}$$

where J is a 3×2 matrix whose columns are $\boldsymbol{G}_0$ and $\boldsymbol{G}_1$.

Define the 5×1 vector $\boldsymbol{x} = [\boldsymbol{\xi}^{\mathsf{T}}\ \boldsymbol{\eta}^{\mathsf{T}}]^{\mathsf{T}}$. The convex quadratic program is to minimize

$$f(\boldsymbol{x}) = \frac{1}{2}\left|(\boldsymbol{K} + U\boldsymbol{\xi}) - (\boldsymbol{V} + J\boldsymbol{\eta})\right| = \frac{1}{2}\boldsymbol{x}^{\mathsf{T}}A\boldsymbol{x} + \boldsymbol{b}^{\mathsf{T}}\boldsymbol{x} \tag{15}$$

where

$$A = \begin{bmatrix} I & -U^{\mathsf{T}}J \\ -J^{\mathsf{T}}U & J^{\mathsf{T}}J \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} U^{\mathsf{T}}(\boldsymbol{K} - \boldsymbol{V}) \\ -J^{\mathsf{T}}(\boldsymbol{K} - \boldsymbol{V}) \end{bmatrix} \tag{16}$$

and I is the 3×3 identity matrix. The constraints are

$$\boldsymbol{x} \ge \boldsymbol{0},\ \boldsymbol{\xi} \le \boldsymbol{\ell},\ h_{\min} \le \boldsymbol{1}^{\mathsf{T}}\boldsymbol{\eta} \le h_{\max} \tag{17}$$

where $\boldsymbol{1}$ is the 2×1 vector whose components are all 1.

As described in [1], the CQP can be formulated as an LCP: Given constants $\boldsymbol{q} \in \mathbb{R}^{10}$ and $M \in \mathbb{R}^{10\times 10}$, find $\boldsymbol{z} \in \mathbb{R}^{10}$ such that $\boldsymbol{z} \ge \boldsymbol{0}$, $\boldsymbol{q} + M\boldsymbol{z} \ge \boldsymbol{0}$ and $\boldsymbol{z}^{\mathsf{T}}(\boldsymbol{q} + M\boldsymbol{z}) = \boldsymbol{0}$. Defining $\boldsymbol{w} = \boldsymbol{q} + M\boldsymbol{z}$, we need to find $\boldsymbol{z}$ for which $\boldsymbol{z} \ge \boldsymbol{0}$, $\boldsymbol{w} \ge \boldsymbol{w}$ and $\boldsymbol{z}^{\mathsf{T}}\boldsymbol{w} = \boldsymbol{0}$. The Geometric Tools source code includes an LCP solver based on Lemke's method.

4.2 Searching for the Minimum Distance Among All Cross Sections

For each $\psi \in [-\pi/2, \pi/2]$, the distance is computed between the oriented box and the quadrilateral cross section using the LCP solver discussed in the previous subsection. Effectively, this defines a function $\delta(\psi)$

that is periodic with period π. A numerical minimizer can be used to locate the angle $\hat{\psi}$ at which the function attains a global minimum $\delta_{\min} = \delta(\hat{\psi})$.

In the Geometric Tools implementation, I use a form of *successive parabolic interpolation* [2] to locate $\hat{\psi}$. The interval $[-\pi/2, \pi/2]$ is processed by examining subintervals. On each subinterval $[a, b]$, the values $\delta_0 = \delta(a)$, $\delta_1 = \delta((a + b)/2)$ and $\delta_2 = \delta(b)$ are examined.

If $\{\delta_0, \delta_1, \delta_2\}$ is monotonic, then $[a, b]$ is subdivided and each subinterval processed. The maximum depth of the monotonic subdivision is limited by a user-specified number.

If $\{\delta_0, \delta_1, \delta_2\}$ is not monotonic, then two cases arise. The first case is $\delta_1 = \min\{\delta_0, \delta_1, \delta_2\}$, then $\{a, (a+b)/2, b\}$ is said to *bracket a minimum*. The points (a, δ_0), $((a + b)/2, \delta_1)$ and (b, δ_2) are fit with a parabola. The vertex $(c, \delta(c))$ of the parabola is guaranteed to be in the open interval (a, b). If $c \in (a, (a+b)/2]$, then a new minimum bracket is $\{a, c, (a+b)/2\}$ and parabolic interpolation is applied to this bracket. If $c \in ((a+b)/2, b)$, then a new minimum bracket is $\{(a + b)/2, c, b\}$ and parabolic interpolation is applied to this bracket. The maximum depth of the bracket subdivision is limited by a user-specified number.

The second case is $\delta_1 = \max\{\delta_0, \delta_1, \delta_2\}$, in which case $\{a, (a + b)/2, b\}$ is said to *bracket a maximum*. It is unknown whether a occurs in $[a, (a+b)/2]$ or $[(a+b)/2, b]$, so the monotonic subdivision mentioned previously is used, processing both of the subintervals.

Additional two parameters are provided by the user to decide whether or not the iterations have converged within floating-point rounding errors. An angle tolerance $\varepsilon_\psi \geq 0$ and a function tolerance $\varepsilon_\delta \geq 0$ are specified by the user. The tolerances are used jointly to decide whether or not to continue the subdivisions in the search for the minimum.

The implementation of the minimizer is the class template <typename T> Minimizer1.

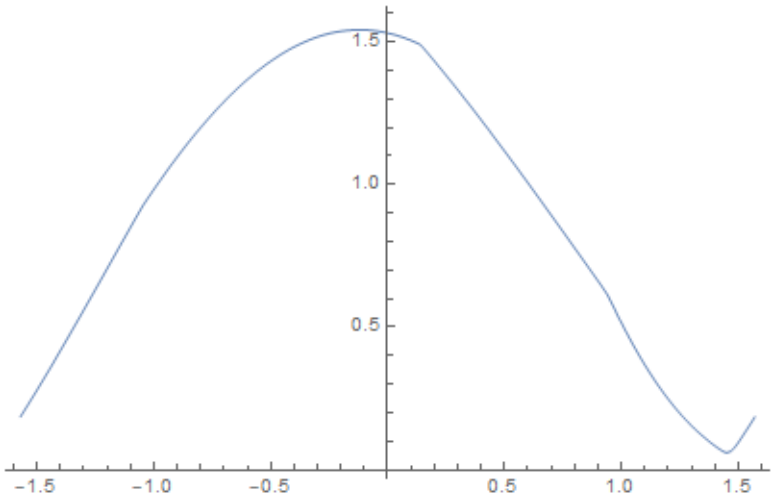
Figure 1 shows the graph of $\delta(\psi)$ for a specific oriented box and cone frustum. The box is

$$\begin{aligned}
\boldsymbol{C} &= (1.46770716, -1.06458569, 0.403121471)\\
\boldsymbol{U}_0 &= (4/30, 28/30, 10/30)\\
\boldsymbol{U}_1 &= (20/30, -10/30, 20/30)\\
\boldsymbol{U}_2 &= (22/30, 4/30, -20/30)\\
\boldsymbol{e} &= (3/4, 1/2, 1/4)
\end{aligned} \tag{18}$$

The cone is

$$\begin{aligned}
\boldsymbol{V} &= (-1, -1, -1)\\
\boldsymbol{D} = \boldsymbol{W}_2 &= (0.267261237, 0.534522474, 0.801783681)\\
\boldsymbol{W}_0 &= (0.0, 0.832050264, -0.554700196)\\
\boldsymbol{W}_1 &= (-0.963624120, 0.148249879, 0.222374782)\\
\theta &= 0.6\\
h_{\min} &= 0.5\\
h_{\max} &= 3
\end{aligned} \tag{19}$$

Figure 1. The graph of $\delta(\psi)$ for $\psi \in (-\pi/2, \pi/2]$. The plot was drawn using Mathematica [3]. The minimum distance is $\delta_{\min} \doteq 0.185247719$,



The function $\delta(\psi)$ is continuous, but its derivative is not continuous. The discontinuities occur at angles ψ where there is some type of parallelism, for example when an edge of the box is parallel to the plane of a quadrilateral cross section.

5 Distance between an Oriented Box and a Cone Frustum

The implementation of the distance query is in the Geometric Tools file DistOrientedBox3Cone3.h, shown in Listing 1.

Listing 1. The interface for the distance query between an oriented box and a cone frustum.

```
template <typename T>
class DCPQuery<T, OrientedBox<3, T>, Cone<3, T>>
{
public:
    DCPQuery()
    {
        static_assert(
            std::is_floating_point<T>::value,
            "The input type must be a floating-point type.");
    }

    // Parameters used internally for controlling the minimizer.
    struct Control
    {
        int32_t maxSubdivisions;
        int32_t maxBisections;
        T epsilon;
        T tolerance;
    };

    // The output of the query, which is the distance between the
```

```
// objects and a pair of closest points, one from each object.
struct Result
{
    Result()
        :
        distance(std::numeric_limits<T>::max()),
        boxClosestPoint{},
        coneClosestPoint{}
    {
        boxClosestPoint.MakeZero();
        coneClosestPoint.MakeZero();
    }

    T distance;
    Vector<3, T> boxClosestPoint, coneClosestPoint;
};

// The default minimizer controls are reasonable choices generally,
// in which case you can use
//   using BCQuery = DCPQuery<T, OrientedBox<3, T>, Cone<3, T>>;
//   BCQuery bcQuery{};
//   BCQuery::Result bcResult = bcQuery(box, cone);
// If your application requires specialized controls,
//   using BCQuery = DCPQuery<T, OrientedBox<3, T>, Cone<3, T>>;
//   BCQuery bcQuery{};
//   BCQuery::Control bcControl(your_parameters);
//   BCQuery::Result bcResult = bcQuery(box, cone, &bcControl);
Result operator()(OrientedBox<3, T> const& box, Cone<3, T> const& cone,
    Control const* inControl = nullptr)
{
    Control control{};
    if (inControl != nullptr)
    {
        control = *inControl;
    }

    // Compute a basis for the cone coordinate system.
    std::array<Vector<3, T>, 3> basis{};
    basis[0] = cone.ray.direction;
    ComputeOrthogonalComplement(1, basis.data());
    Vector<3, T> coneW0 = basis[1];
    Vector<3, T> coneW1 = basis[2];

    Result result{};
    result.distance = std::numeric_limits<T>::max();

    auto F = [this, &box, &cone, &coneW0, &coneW1, &result](T angle)
    {
        T distance = std::numeric_limits<T>::max();
        Vector<3, T> boxClosestPoint{}, quadClosestPoint{};
        DoBoxQuadQuery(box, cone, coneW0, coneW1, angle,
            distance, boxClosestPoint, quadClosestPoint);

        if (distance < result.distance)
        {
            result.distance = distance;
            result.boxClosestPoint = boxClosestPoint;
            result.coneClosestPoint = quadClosestPoint;
        }

        return distance;
    };

    Minimize1<T> minimizer(F, control.maxSubdivisions, control.maxBisections,
        control.epsilon, control.tolerance);
    T angle0 = static_cast<T>(-GTE_C_HALF_PI);
    T angle1 = static_cast<T>(+GTE_C_HALF_PI);
    T angleMin = static_cast<T>(0);
    T distanceMin = std::numeric_limits<T>::max();
    minimizer.GetMinimum(angle0, angle1, angleMin, distanceMin);
    LogAssert(
        distanceMin == result.distance,
```

```
            "Unexpected mismatch in minimum distance.");

        return result;
    }

private:
    void DoBoxQuadQuery(OrientedBox<3, T> const& box, Cone<3, T> const& cone,
        Vector<3, T> const& coneW0, Vector<3, T> const& coneW1,
        T const& quadAngle, T& distance, Vector<3, T>& boxClosestPoint,
        Vector<3, T>& quadClosestPoint)
    {
        T const zero = static_cast<T>(0);
        T const one = static_cast<T>(1);
        T const two = static_cast<T>(2);

        Vector<3, T> K = box.center, ell{};
        for (int32_t i = 0; i < 3; ++i)
        {
            K -= box.extent[i] * box.axis[i];
            ell[i] = two * box.extent[i];
        }

        T cs = std::cos(quadAngle), sn = std::sin(quadAngle);
        Vector<3, T> term = cone.tanAngle * (cs * coneW0 + sn * coneW1);
        std::array<Vector<3, T>, 2> G{};
        G[0] = cone.ray.direction - term;
        G[1] = cone.ray.direction + term;

        Matrix<5, 5, T> A{};  // A is the zero matrix
        A(0, 0) = one;
        A(0, 1) = zero;
        A(0, 2) = zero;
        A(0, 3) = -Dot(box.axis[0], G[0]);
        A(0, 4) = -Dot(box.axis[0], G[1]);
        A(1, 0) = A(0, 1);
        A(1, 1) = one;
        A(1, 2) = zero;
        A(1, 3) = -Dot(box.axis[1], G[0]);
        A(1, 4) = -Dot(box.axis[1], G[1]);
        A(2, 0) = A(0, 2);
        A(2, 1) = A(1, 2);
        A(2, 2) = one;
        A(2, 3) = -Dot(box.axis[2], G[0]);
        A(2, 4) = -Dot(box.axis[2], G[1]);
        A(3, 0) = A(0, 3);
        A(3, 1) = A(1, 3);
        A(3, 2) = A(2, 3);
        A(3, 3) = Dot(G[0], G[0]);
        A(3, 4) = Dot(G[0], G[1]);
        A(4, 0) = A(0, 4);
        A(4, 1) = A(1, 4);
        A(4, 2) = A(2, 4);
        A(4, 3) = A(3, 4);
        A(4, 4) = Dot(G[1], G[1]);

        Vector<3, T> KmV = K - cone.ray.origin;
        Vector<5, T> b{};
        b[0] = Dot(box.axis[0], KmV);
        b[1] = Dot(box.axis[1], KmV);
        b[2] = Dot(box.axis[2], KmV);
        b[3] = -Dot(G[0], KmV);
        b[4] = -Dot(G[1], KmV);

        Matrix<5, 5, T> D{};  // D is the zero matrix
        D(0, 0) = -one;
        D(1, 1) = -one;
        D(2, 2) = -one;
        D(3, 3) = one;
        D(3, 4) = one;
        D(4, 3) = -one;
        D(4, 4) = -one;
```

```
        Vector<5, T> e{};
        e[0] = -ell[0];
        e[1] = -ell[1];
        e[2] = -ell[2];
        e[3] = cone.GetMinHeight();
        e[4] = -cone.GetMaxHeight();

        std::array<T, 10> q;
        for (int32_t i = 0; i < 5; ++i)
        {
            q[i] = b[i];
            q[i + 5] = -e[i];
        }

        std::array<std::array<T, 10>, 10> M;
        for (int32_t r = 0; r < 5; ++r)
        {
            for (int32_t c = 0; c < 5; ++c)
            {
                M[r][c] = A(r, c);
                M[r + 5][c] = D(r, c);
                M[r][c + 5] = -D(c, r);
                M[r + 5][c + 5] = zero;
            }
        }

        std::array<T, 10> w, z;
        if (mLCP.Solve(q, M, w, z))
        {
            boxClosestPoint = K;
            for (int32_t i = 0; i < 3; ++i)
            {
                boxClosestPoint += z[i] * box.axis[i];
            }

            quadClosestPoint = cone.ray.origin;
            for (int32_t i = 0; i < 2; ++i)
            {
                quadClosestPoint += z[i + 3] * G[i];
            }

            distance = Length(boxClosestPoint - quadClosestPoint);
        }
        else
        {
            boxClosestPoint.MakeZero();
            quadClosestPoint.MakeZero();
            distance = std::numeric_limits<T>::max();
        }
    }

    LCPSolver<T, 10> mLCP;
};
```

The top-level query is the operator()(...) function. Its job is to prepare for the minimization of distances to the quadrilateral cross sections. The two steps are to compute $\boldsymbol{W}_0$ and $\boldsymbol{W}_1$ for which $\{\boldsymbol{W}_0, \boldsymbol{W}_1, \boldsymbol{D}\}$ is a right-handed orthonormal basis and to create the function object that wraps the distance query for the box and a quadrilateral cross section for a specified angle. After preparation, the minimizer is called.

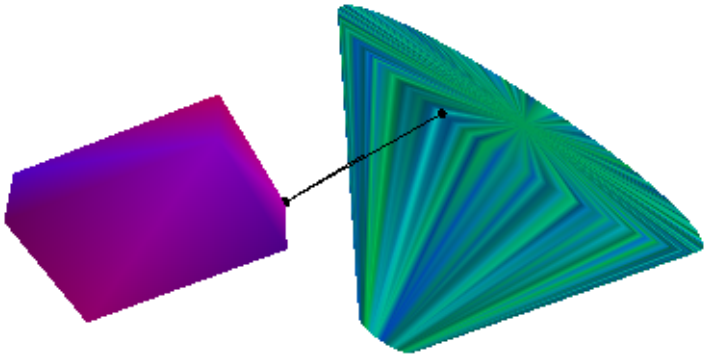
The distance query between the box and a quadrilateral cross section is implemented in DoBoxQuadQuery. As discussed previously, the query sets up a convex quadratic program and solves it using a linear complementarity program.

A sample application to illustrate is in the folder

GeometricTools/Samples/Distance/DistanceOrientedBoxConeFrustum

Figure 2 shows the initial display of various objects.

Figure 2. The initial display of the DistanceOrientedBoxConeFrustum sample application.



The box mesh contains 8 vertices and 12 triangles. The cone mesh is created as an inscribed convex polyhedron. The triangles of the minimum-height disk form a regular polygon having a user-specified number of vertices. The triangles of the maximum-height disk form a regular polygon with more vertices than the minimum-height disk. The number of vertices is chosen internally to make the arc lengths between two consecutive vertices approximately the same for both regular polygons. The cone wall is triangulated connecting vertices between both regular polygons. Two small spheres are shown. These are centered at the closest points corresponding to the box-cone distance. A line segment is drawn to connect the two points.

Figure 3 shows a different view of the box and cone, this time in wireframe mode. The pair of closest points for the box and cone are still shown as small spheres connected by a line segment. The quadrilateral cross section of the cone is drawn as a solid object inside the cone. This quadrilateral corresponds to angle $\phi = 0$. The closest points between the box and the quadrilateral are also drawn as small spheres and connected by a line segment. In this configuration, a box corner is closest both to the quadrilateral and to the cone, but the distance to the quadrilateral is necessarily larger than that to the cone.

Figure 3. The box and cone are drawn in wireframe. A quadrilateral cross section is shown inside the cone. The closest box-cone points and the closest box-quadrilateral points are drawn as small spheres connected by line segments.

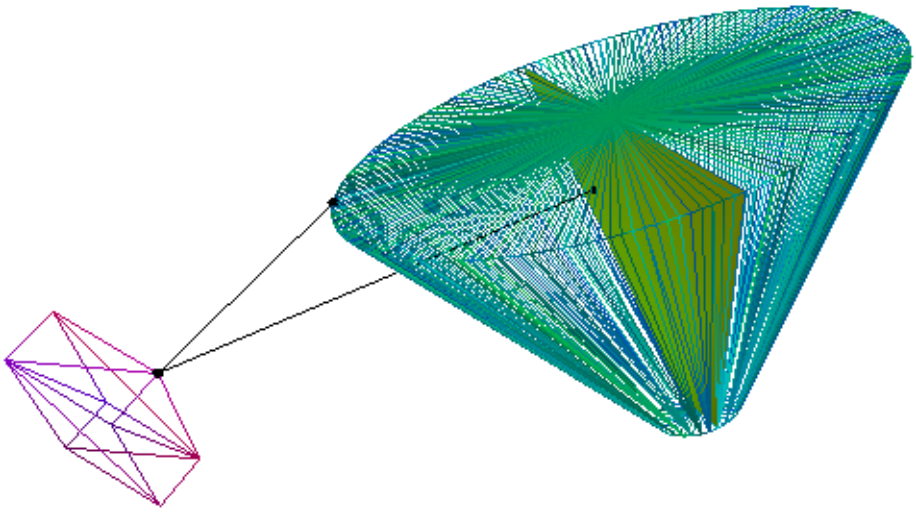


Figure 4 shows yet a different view of the box and cone. However, the quadrilateral cross section is different from that of the previous two figures. The quadrilateral selection is obtained by rotation, pressing the '+' key or '-' key. The displayed quadrilateral is not the one that attains the box-cone distance, but it is close.

Figure 4. The box and cone are drawn in wireframe. A quadrilateral cross section is shown inside the cone. The closest box-cone points and the closest box-quadrilateral points are drawn as small spheres connected by line segments. The quadrilateral is not the one that attains the box-cone distance, but it is close.

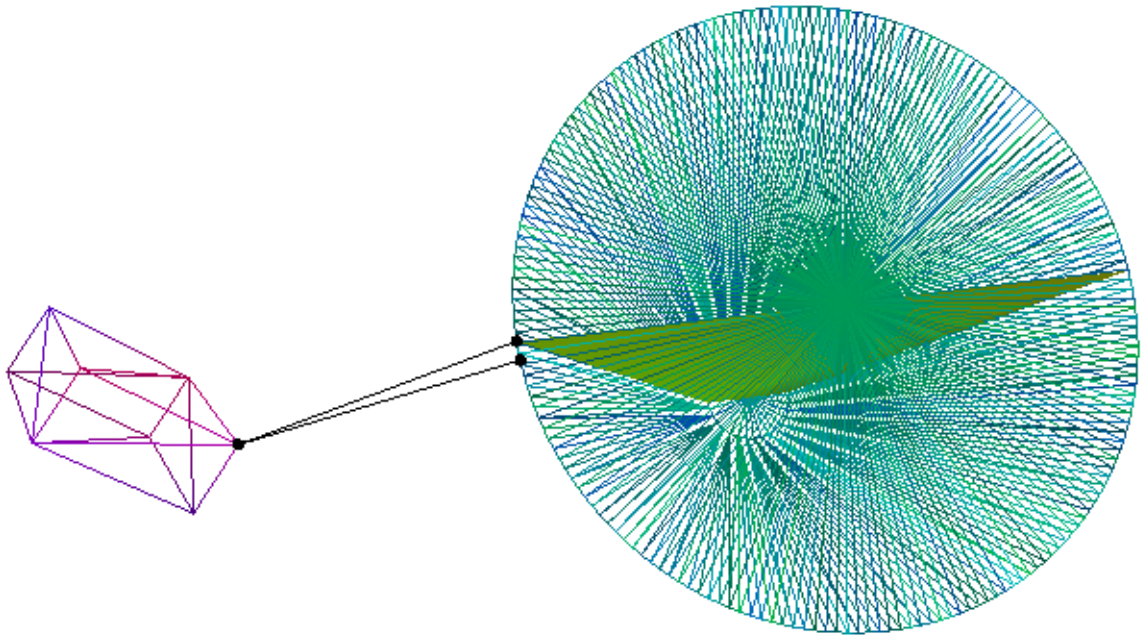
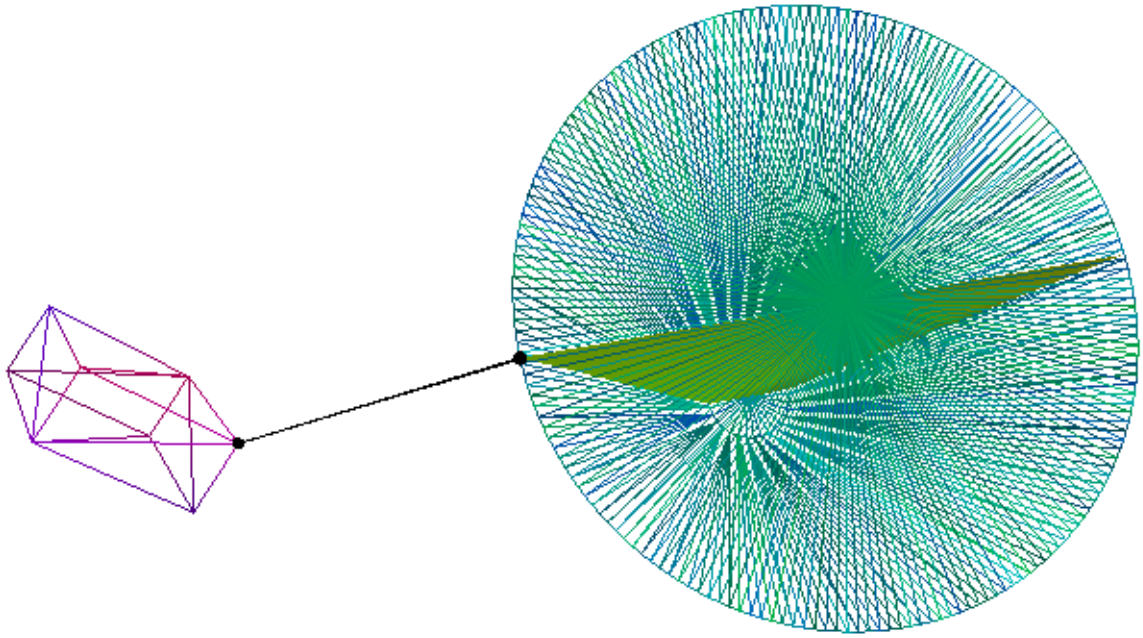


Figure 5 shows the same view as that of figure 4, but the quadrilateral was rotated to become the one that attains the box-cone distance.

Figure 5. The box and cone are drawn in wireframe. The quadrilateral cross section that attains box-cone distance is shown inside the cone.



The sample application also allows you to translate the box using keys 'x', 'X', 'y', 'Y', 'z' or 'Z'. You can rotate the box using keys 'p', 'P', 'r', 'R', 'h' or 'H'. During translation or rotation, the pairs of closest points will vary. The closest box point can be at a box corner, on a box edge or on a box face. If the box and cone have overlap with positive volume, all points in the overlap have zero distance between box and cone. The implementation will return some pair, stored in Result, but these might differ slightly because of floating-point rounding errors, so the distance is nearly zero.

References

[1] Dave Eberly. *Robust and Error-Free Geometric Computing.* CRC Press, Taylor & Francis Group LLC, Boca Raton, FL, 2020.

[2] Wikipedia. Successive parabolic interpolation. https://en.wikipedia.org/wiki/Successive_parabolic_interpolation. accessed May 20, 2021.

[3] Wolfram Research, Inc. *Mathematica 12.1.1.0.* Wolfram Research, Inc., Champaign, Illinois, 2020.