Intersection of a Line and a Cone

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

Created: October 17, 2000
Last Modified: September 11, 2020

Contents

1 Introduction

This document describes an algorithm for computing the set of intersection between a cone and a line, ray or segment. The algorithm is theoretically correct when using real-valued arithmetic. When computing using floating-point arithmetic, rounding errors can cause misclassifications of signs of important expressions, leading to failure of the implementation to produce correct results.

With the exception of the approximation error in computing the cosine of a user-specified cone angle, it is possible to obtain the theoretically correct result by using arbitrary precision arithmetic in conjunction with symbolic arithmetic that handles the square root operations to avoid the rounding errors inherent in computing those roots. An implementation of the algorithm is also described in this document. The set of intersection is either empty, a point, a segment, a ray or a line. When a point, segment or ray, the output of the line-cone intersection is stored using rational numbers and a symbolic representation of a square root. The output can then be converted to floating-point using as many bits of precision as desired.

1.1 Definition of Cones

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

$$\mathbf{D} \cdot \frac{(\mathbf{X}-\mathbf{V})}{|\mathbf{X}-\mathbf{V}|} \geq \cos(\theta) \tag{1}$$

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

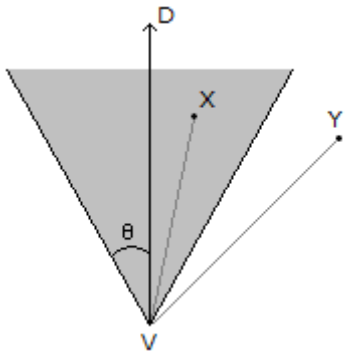
$$\mathbf{D} \cdot (\mathbf{X}-\mathbf{V}) \geq |\mathbf{X}-\mathbf{V}| \cos(\theta) \tag{2}$$

which includes the case $\mathbf{X} = \mathbf{V}$. 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

$$Q(\mathbf{X}) = (\mathbf{D} \cdot (\mathbf{X}-\mathbf{V}))^2 - \gamma^2 |\mathbf{X}-\mathbf{V}|^2 = (\mathbf{X}-\mathbf{V})^{\mathsf{T}} M (\mathbf{X}-\mathbf{V}) \geq 0, \;\; \mathbf{D} \cdot (\mathbf{X}-\mathbf{V}) \geq 0 \tag{3}$$

where $\gamma = \cos\theta$, $M = \mathbf{D}\mathbf{D}^{\mathsf{T}} - \gamma^2 I$ is a symmetric 3×3 matrix and I the 3×3 identity matrix. Figure 1 shows a 2D cone, which is sufficient to illustrate the quantites in 3D.

Figure 1. A 2D view of a single-sided cone. $\mathbf{X}$ is inside the cone and $\mathbf{Y}$ is outside the cone.



Because of the constraint on θ, both $\cos(\theta) > 0$ and $\sin(\theta) > 0$.

The *height* of a point $\mathbf{X}$ relative to the cone is the length of the projection of $\mathbf{X}-\mathbf{V}$ onto the $\mathbf{D}$-axis, namely, $h = \mathbf{D} \cdot (\mathbf{X} - \mathbf{V})$. 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 $0 \le h_{\min} < h_{\max} \le +\infty$.

- 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): $h_{\min} > 0$, $h_{\max} < +\infty$.

In the discussions here, I use the term *cone* to refer to any one of the four classifications, making it clear the specific details of the algorithms for each classification.

1.2 Practical Matters for Representing Infinity

A practical matter when defining a cone data structure is how to represent an infinite maximum height, $h_{\max} = +\infty$. With floating-point computations, it is reasonable to choose std::numeric_limits<Real>::infinity() for Real set to float or double. This representation is the special IEEE floating-point pattern used to represent infinity. If Real will be an arbitrary precision type that does not have a representation for infinity. The next choice could be to use the largest finite floating-point number, std::numeric_limits<Real>::max(). An arbitrary precision type will have code to convert this to a rational representation. However, the semantics for infinity are no longer clear, because the rational representation of the largest finite floating-point number has nothing to do with infinity. For example, if an algorithm required squaring the maximum height, the IEEE infinity squared still produces infinity. Squaring the largest finite floating-point number represented as a rational number will lead to another rational number, and now comparison semantics no longer work correctly.

In the Cone class of GTE, I chose the setting of minimum and maximum height to hide the representation. An infinite height is represented by setting internally the (hidden) class member for maximum height to -1. Internal query code can then test whether a cone is finite or infinite by examining this class member.

If an algorithm requires the concept of infinity, positive or negative, in comparisons, it is generally possible to do so without using the IEEE infinity representations and without requiring an arbitrary precision arithmetic libray to have infinity representations. Instead, we can use a *2-point compactification of the real numbers*. Let $x \in (-\infty, +\infty)$ be any real number. Define $y = x/(1 + |x|) \in (-1, 1)$. The function is a bijection and has inverse $x = y/(1 - |y|)$. The compactification occurs when we choose $+1$ to represent $+\infty$ and choose -1 to represent $-\infty$. In this sense, $y = x/(1 + |x|)$ maps $[-\infty, +\infty]$ to $[-1, 1]$. If we want to know whether $x_0 < x_1$ even when either number is an infinity, we can compute $y_i = x_i/(1 + |x_i|)$ and instead compare $y_0 < y_1$. The y-values are always finite. In fact, the bijection maps rational numbers to rational numbers, so the y-comparisons are error free when using arbitrary precision arithmetic. The idea also applies to symbolic arithmetic involving numbers of the form $z = x + y\sqrt{d}$, where x, y and d are rational numbers and $d > 0$; this is the underlying framework for dealing with lengths of vectors symbolically.

1.3 Definition of a Line, Ray and Segment

A line is parameterized by $\mathbf{X}(t) = \mathbf{P} + t\mathbf{U}$, where $\mathbf{P}$ is a point on the line (the line origin), $\mathbf{U}$ is typically a unit-length direction vector for the line, and t is a real number. A ray has the subset of the line with restriction $t \geq 0$. A segment is a subset of the line with restriction $t \in [0, t_{\max}]$, so the endpoints are $\mathbf{P}$ and $\mathbf{P} + t_{\max}\mathbf{U}$. However, the endpoints are typically specified, say, $\mathbf{E}_0$ and $\mathbf{E}_1$, with $\mathbf{X}(t) = (1-t)\mathbf{E}_0 + t\mathbf{E}_1$ for $t \in [0, 1]$.

The line-cone find-intersection query described here does not require $\mathbf{U}$ to be unit length. Sections 2, 3 and 4. These sections assume real-valued arithmetic, which is error-free in the theoretical sense.

In Section 8, the restriction that $\mathbf{D}$ be unit length is dropped, but we will then need to use a combination of rational and symbolic arithmetic to compute exact results.

2 Intersection with a Line

Let us find the points of intersection with the cone boundary $Q(\mathbf{X}) = 0$, where Q is defined by Equation (3). Substitute the line equation $\mathbf{X}(t) = \mathbf{P} + t\mathbf{U}$ into the quadratic polynomial of equation (1) to obtain $c_2t^2 + 2c_1t + c_0 = 0$, where $\mathbf{\Delta} = \mathbf{P} - \mathbf{V}$. The vector $\mathbf{U}$ is not required to be unit length. The coefficients are

$$\begin{aligned} c_2 &= \mathbf{U}^{\mathsf{T}} M \mathbf{U} = (\mathbf{D}\cdot\mathbf{U})^2 - \gamma^2(\mathbf{U}\cdot\mathbf{U}) \\ c_1 &= \mathbf{U}^{\mathsf{T}} M \mathbf{\Delta} = (\mathbf{D}\cdot\mathbf{U})(\mathbf{D}\cdot\mathbf{\Delta}) - \gamma^2(\mathbf{U}\cdot\mathbf{\Delta}) \\ c_0 &= \mathbf{\Delta}^{\mathsf{T}} M \mathbf{\Delta} = (\mathbf{D}\cdot\mathbf{\Delta})^2 - \gamma^2(\mathbf{\Delta}\cdot\mathbf{\Delta}) \end{aligned} \tag{4}$$

It is convenient to reduce algorithm branching cases by choosing the line direction $\mathbf{U}$ so that $\mathbf{D}\cdot\mathbf{U} \geq 0$. In practice, the user-specified line direction can be negated to ensure the nonnegative dot product. The t-values at the intersection points are computed, but when reporting this information to the caller, the t-values must be negated to undo the sign change in the line direction.

The bounds $h_{\min} \leq \mathbf{D}\cdot(\mathbf{X}(t) - \mathbf{V}) \leq h_{\max}$ become

$$h_{\min} \leq t(\mathbf{D}\cdot\mathbf{U}) + (\mathbf{D}\cdot\mathbf{\Delta}) \leq h_{\max} \tag{5}$$

We must compute the roots of a quadratic polynomial (possibly degenerate) subject to linear inequality constraints on t. The roots may be computed as if we have an infinite cone. For the infinite truncated cone, the finite cone and the cone frustum, the clamping to finite h-bounds can be applied as a postprocessing step.

2.1 Case $c_2 \neq 0$

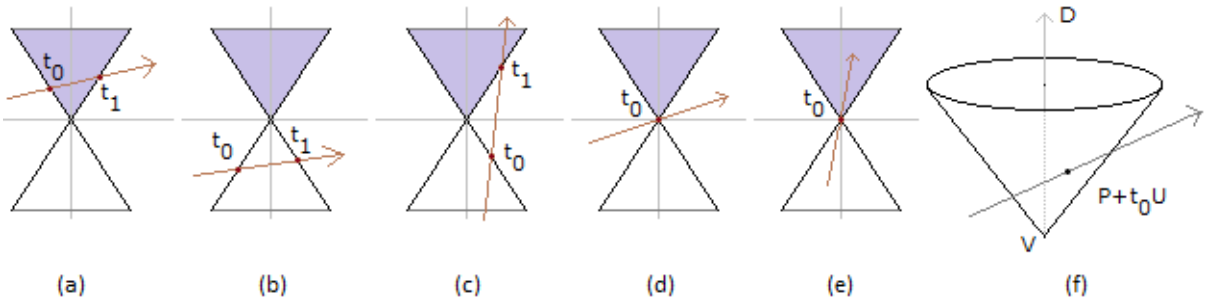
Suppose that $c_2 \neq 0$. The formal roots are $t = (-c_1 \pm \sqrt{\delta})/c_2$, where $\delta = c_1^2 - c_0c_2$.

If $\delta < 0$, the quadratic polynomial has no real-valued roots, in which case the line does not intersect the double-sided cone, which means it does not intersect cones of any of the 4 classifications.

If $\delta = 0$, the polynomial has a repeated real-value root $t = -c_1/c_2$. This occurs in two different geometric configurations. One configuration is when the line is tangent to the double-sided cone at a single point. The other is when the line contains the cone vertex $\mathbf{V}$, in which case $\mathbf{V} = \mathbf{P} + (-c_1/c_2)\mathbf{U}$.

If $\delta > 0$, the polynomial has two distinct real-valued roots, in which case the line intersects the double-sided cone at two points. Figure 2 illustrates the various cases.

Figure 2. Geometric configurations when $c_2 \neq 0$. We care only about the intersections with the positive cone; that is, where $\mathbf{D} \cdot (\mathbf{X} - \mathbf{V}) \geq 0$. (a) $\delta > 0$ and both points are on the positive cone. (b) $\delta > 0$ and both points are on the negative cone. (c) $\delta > 0$, one point is on the positive cone and one point is on the negative cone. (d) $\delta = 0$, $c_2 < 0$ ($\mathbf{U}$ is outside the cone), and the line contains the vertex. (e) $\delta = 0$, $c_2 > 0$ ($\mathbf{U}$ is inside the cone), and the line contains the vertex. (f) $\delta = 0$ and the line is tangent to the cone.

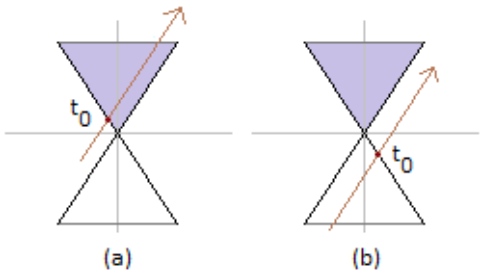


In Figure 2, the intersection types with the double-sided cone are as follows: (a) and (b), segments; (c) ray; (d) and (f) point; (e) line. The intersection types with the positive cone are: (a) segment; (b) none; (c) and (e) ray; (d) and (f) point.

2.2 Case $c_2 = 0$ and $c_1 \neq 0$

If $c_2 = 0$, the vector $\mathbf{U}$ is a direction vector on the cone boundary because $|\mathbf{D} \cdot \mathbf{U}| = \cos(\theta)$. If $c_1 \neq 0$, the polynomial is in fact linear and has a single root $t = -c_0/(2c_1)$. The line and double-sided cone have a single point of intersection. As before, we report the point as an intersection only when it is on the positive cone. If it is, we must choose the correct t-interval of intersection. Figure 3 illustrates a couple of configurations.

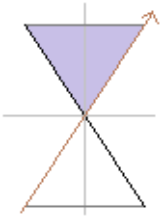
Figure 3. Geometric configurations when $c_2 = 0$ and $c_1 \neq 0$. We care only about the intersections with the positive cone; that is, where $\mathbf{D} \cdot (\mathbf{X} - \mathbf{V}) \geq 0$. (a) The line intersects the positive cone in a single point. (b) The line intersects the negative cone in a single point.



2.3 Case $c_2 = 0$ and $c_1 = 0$

The condition $c_2 = 0$ implies the line direction $\mathbf{U}$ is on the cone surface. The condition $c_1 = 0$ is necessary for the line to contain the cone vertex $\mathbf{V}$. Together these conditions and the quadratic equation imply $c_0 = 0$, in which case the line lies on the cone surface. If $c_0 \neq 0$, the line does not intersect the cone. Figure 4 illustrates the case $c_0 = c_1 = c_2 = 0$.

Figure 4. When $c_0 = c_1 = c_2 = 0$, the line lives on the cone surface and contains the cone vertex.



3 Clamping to the Cone Height Range

Generally, we defined various cone types and their corresponding height ranges $[h_{\min}, h_{\max}]$. The relationship between height h and line parameter t is in equation (5). We chose the line direction so that $\mathbf{D} \cdot \mathbf{U} \geq 0$.

For each intersection point of the line and cone, say, occurring at parameter t, we can compute $h = t(\mathbf{D} \cdot \mathbf{U}) + (\mathbf{D} \cdot \mathbf{\Delta})$. When $\mathbf{D} \cdot \mathbf{U} > 0$, given a height h we can solve for

$$t = (h - \mathbf{D} \cdot \mathbf{\Delta})/(\mathbf{D} \cdot \mathbf{U}) \tag{6}$$

The h-values and t-values increase jointly or decrease jointly. For two intersection points, $t_0 < t_1$ implies $h_0 < h_1$. When $\mathbf{D} \cdot \mathbf{U} = 0$, the line is perpendicular to the cone axis and the heights are the same value $\mathbf{D} \cdot \mathbf{\Delta}$ for all line points.

The idea is to compute the h-interval $[h_0, h_1]$ for the intersection set and intersect it with the h-interval

$[h_{\min}, h_{\max}]$ for the cone. If the set is nonempty, the corresponding t-values are computed using equation (6) and the linear component quantities are computed from them.

4 Pseudocode for Error-Free Real-Valued Arithmetic

The algorithm requires find-intersection queries for intervals, where the intervals are finite or semiinfinite. These queries are discussed first.

4.1 Intersection of Intervals

The find-intersection query requires computing the intersection of finite intervals $[u_0, u_1]$ and $[v_0, v_1]$, where $u_0 \le u_1$ and $v_0 \le v_1$. An interval can be degenerate in that its endpoints are the same number. Listing 1 contains pseudocode for computing the intersection.

Listing 1. Pseudocode for the find-intersection query of two finite intervals. The function returns an integer value that is the number of valid elements of overlap[]. It is 0 (no intersection), 1 (intervals touch only at an endpoint) or 2 (intervals overlap in an interval).

```
int FindIntersection(Real u0, Real u1, Real v0, Real v1, Real overlap[2])
{
    int numValid;
    if (u1 < v0 || v1 < u0)
    {
        numValid = 0;
    }
    else if (v0 < u1)
    {
        if (u0 < v1)
        {
            overlap[0] = (u0 < v0 ? v0 : u0);
            overlap[1] = (u1 > v1 ? v1 : u1);
            if (overlap[0] < overlap[1])
            {
                numValid = 2;
            }
            else
            {
                numValid = 1;
            }
        }
        else  // u0 == v1
        {
            overlap[0] = u0;
            overlap[1] = u0;
            numValid = 1;
        }
    }
    else  // u1 == v0
    {
        overlap[0] = v0;
        overlap[1] = v0;
        numValid = 1;
    }
    return numValid;
}
```

A semiinfinite interval is of the form $[v, +\infty)$ or $(-\infty, v]$. The first interval contains real numbers t for which $t \geq v$. The second interval contains real numbers t for which $t \leq v$. If the type Real of Listing 1 has a representation for infinities, that code applies as well when either input is a semiinfinite interval. We are not assuming such a representation, but additional functions can be implemented with the correct logic for semiinfinite intervals. For the line-cone intersection query, we need only deal with semiinfinite intervals $[v, +\infty)$, so only the query for such an interval is discussed.

Suppose that the first input is the finite interval $[u_0, u_1]$ and the second interval is $[v_0, +\infty]$, where $v_1 = +\infty$ and the closed bracket is suggestive that $+\infty$ is an element of that set. In Listing 1, the tests v[1] < u[0] and u[1] > v[1] are always false and the test u[0] < v[1] is always true. The code can be rewritten using these facts, and the input v-interval is passed as the endpoint v_0. Listing 2 shows pseudocode for the find-intersection query.

Listing 2. Pseudocode for the find-intersection query of a finite interval $[u_0, u_1]$ and a semiinfinite interval $[v, +\infty)$. The function returns an integer value that is the number of valid elements of overlap[]. It is 0 (no intersection), 1 (intervals touch only at an endpoint) or 2 (intervals overlap in an interval).

```
int FindIntersection(Real u0, Real u1, Real v, Real overlap[2])
{
    int numValid;
    if (u1 > v)
    {
        numValid = 2;
        overlap[0] = max(u0, v);
        overlap[1] = u1;
    }
    else if (u1 == v)
    {
        numValid = 1;
        overlap[0] = v;
    }
    else // u1 < v
    {
        numValid = 0;
    }
    return numValid;
}
```

The final find-intersection query involves two semiinfinite intervals $[u, +\infty)$ and $[v, +\infty)$. This query is required when computing the intersection of a ray and a cone with infinite height. It is a simple query that produces the interval $[\max\{u, v\}, +\infty)$.

4.2 Line-Cone Query

The pseudocode for the line-cone find-intersection query must compute the subset of the line that is contained inside the positive cone. The possible types of subsets are characterized next. The query function returns one of these types. It also returns a 2-tuple Real t[2] that stores the line parameters defining the intersection. The number of valid elements (0, 1 or 2) depends on the type. A corresponding 2-tuple of points is returned, Vector3 P[2]. The type list is

- NONE: There is no intersection. The elements of t[] and P[] are invalid.

- POINT: The intersection consists of a single point. The values t[0] and P[0] are valid, the other pair invalid.
- SEGMENT: The intersection is a segment. All values t[] and P[] are valid. The difference of the two points is necessarily parallel to $\mathbf{U}$.
- RAY_POSITIVE: The intersection is a ray $\mathbf{P} + t\mathbf{U}$ with $t \in [t_0, +\infty)$. The value t[0] stores t_0 and the value P[0] stores $\mathbf{P} + t_0\mathbf{U}$.
- RAY_NEGATIVE: The intersection is a ray $\mathbf{P} + t\mathbf{U}$ with $t \in (-\infty, t_1]$. The value t[1] stores t_1 and the value P[1] stores $\mathbf{P} + t_1\mathbf{U}$.

Listing 3 contains pseudocode for the find-intersection query between a line and a cone.

Listing 3. Pseudocode for the line-cone find-intersection query.

```
struct Line3
{
    Vector3 P;  // origin
    Vector3 U;  // direction, not required to be unit length
};

struct Cone3
{
    Vector3 V;  // vertex
    Vector3 D;  // axis direction, required to be unit length
    Real cosAngleSqr;  // the square of cos(angle) for the cone angle

    // The height range of the cone.  The minimum height is hmin >= 0.
    // The maximum height is hmax > hmin for a finite cone but -1 for an
    // infinite cone.  The Boolean value indicates whether or not the
    // hmax is valid (true for a finite cone, false for an infinite cone).
    Real hmin, hmax;
    bool isFinite;
};

// The returned 'int' is one of NONE, POINT, SEGMENT, RAY_POSITIVE or RAY_NEGATIVE.
// The returned t[] and P[] values are as described previously.
int FindIntersection(Line3 line, Cone3 cone, Real t[2], Vector3 P[2])
{
    int intersectionType = DoQuery(line.P, line.U, cone, t);
    ComputePoints(intersectionType, line.P, line.U, t, P);
    return intersectionType;
}

void ComputePoints(int intersectionType, Vector3 origin, Vector3 direction, Real t[2], Vector3 P[2])
{
    switch (intersectionType)
    {
        case NONE:
            P[0] = invalid_point;
            P[1] = invalid_point;
            break;
        case POINT:
            P[0] = origin + t[0] * direction;
            P[1] = invalid_point;
            break;
        case SEGMENT:
            P[0] = origin + t[0] * direction;
            P[1] = origin + t[1] * direction;
            break;
        case RAY_POSITIVE:
            P[0] = origin + t[0] * direction;
            P[1] = invalid_point;
            break;
```

```
        case RAY_NEGATIVE:
            P[0] = invalid_point;
            P[1] = origin + t[1] * direction;
            break;
    }
}

int DoQuery(Vector3 P, Vector3 U, Cone3 cone, Real t[2])
{
    // Arrange for an acute angle between the cone direction and line direction.
    // This simplifies the logic later in the code, and it supports additional
    // queries involving rays or segments instead of lines.
    int intersectionType;
    Real DdU = Dot(cone.D, U);
    if (DdU >= 0)
    {
        intersectionType = DoQuerySpecial(P, U, cone, t);
    }
    else
    {
        intersectionType = DoQuerySpecial(P, -U, cone, t);
        t[0] = -t[0];
        t[1] = -t[1];
        swap(t[0], t[1]);
        if (intersectionType == RAY_POSITIVE)
        {
            intersectionType = RAY_NEGATIVE;
        }
    }
    return intersectionType;
}

int DoQuerySpecial(Vector3 P, Vector3 U, Cone3 cone, Real t[2])
{
    // Compute the quadratic coefficients.
    Vector3 PmV = P - cone.V;
    Real DdU = Dot(cone.D, U);
    Real UdU = Dot(U, U);
    Real DdPmV = Dot(cone.D, PmV);
    Real UdPmV = Dot(U, PmV);
    Real PmVdPmV = Dot(PmV, PmV);
    Real c2 = DdU * DdU - cone.cosThetaSqr * UdU;
    Real c1 = DdU * DdPmV - cone.cosThetaSqr * UdPmV;
    Real c0 = DdPmV * DdPmV - cone.cosThetaSqr * PmVdPmV;

    if (c2 != 0)
    {
        Real discr = c1 * c1 - c0 * c2;
        if (discr < 0)
        {
            return CaseC2NotZeroDiscrNeg(t);
        }
        else if (discr > 0)
        {
            return CaseC2NotZeroDiscrPos(c1, c2, discr, DdU, DdPmV, cone, t);
        }
        else
        {
            return CaseC2NotZeroDiscrZero(c1, c2, UdU, UdPmV, DdU, DdPmV, cone, t);
        }
    }
    else if (c1 != 0)
    {
        return CaseC2ZeroC1NotZero(c0, c1, DdU, DdPmV, cone, t);
    }
    else
    {
        return CaseC2ZeroC1Zero(c0, UdU, UdPmV, DdU, DdPmV, cone, t);
    }
}

int CaseC2NotZeroDiscrNeg(Real t[2])
```

```
{
    // Block 0. The quadratic polynomial has no real-valued roots. The line does not intersect the
    // double-sided cone.
    return SetEmpty(t);
}

int CaseC2NotZeroDiscrPos(Real c1, Real c2, Real discr, Real DdU, Real DdPmV, Cone3 cone, Real t[2])
{
    // The quadratic has two distinct real-valued roots, t0 and t1 with t0 < t1.  Also compute the signed
    // heights at the intersection points, h0 and h1 with h0 <= h1. The ordering is guaranteed because we
    // have arranged for the input line to satisfy DdU >= 0.
    Real x = -c1 / c2;
    Real y = (c2 > 0 ? 1 / c2 : -1 / c2);
    Real t0 = x - y * sqrt(discr), t1 = x + y * sqrt(discr);
    Real h0 = t0 * DdU + DdPmV, h1 = t1 * DdU + DdPmV;

    if (h0 >= 0)
    {
        // Block 1, Figure 2(a).  The line intersects the positive cone in two points.
        return SetSegmentClamp(t0, t1, h0, h1, DdU, DdPmV, cone, t);
    }
    else if (h1 <= 0)
    {
        // Block 2, Figure 2(b). The line intersects the negative cone in two points.
        return SetEmpty(t);
    }
    else  // h0 < 0 < h1
    {
        // Block 3, Figure 2(c). The line intersects the positive cone in a single point and the
        // negative cone in a single point.
        return SetRayClamp(h1, DdU, DdPmV, cone, t);
    }
}

int CaseC2NotZeroDiscrZero(Real c1, Real c2, Real UdU, Real UdPmV, Real DdU, Real DdPmV, Cone3 cone,
    Real t[2])
{
    Real t = -c1 / c2;
    if (t * UdU + UdPmV == 0)
    {
        // To get here, it must be that V = P + (-c1/c2) * U, where U is not necessarily a unit-length
        // vector. The line intersects the cone vertex.
        if (c2 < 0)
        {
            // Block 4, Figure 2(d). The line is outside the double-sided cone and intersects it only at V.
            Real h = 0;
            return SetPointClamp(t, h, cone, t);
        }
        else
        {
            // Block 5, Figure 2(e). The line is inside the double-sided cone, so the intersection is a ray
            // with origin V.
            Real h = 0;
            return SetRayClamp(h, DdU, DdPmV, cone, t);
        }
    }
    else
    {
        // The line is tangent to the cone at a point different from the vertex.
        Real h = t * DdU + DdPmV;
        if (h >= 0)
        {
            // Block 6, Figure 2(f). The line is tangent to the positive cone.
            return SetPointClamp(t, h, cone, t);
        }
        else
        {
            // Block 7. The line is tangent to the negative cone.
            return SetEmpty(t);
        }
    }
}
```

```
int CaseC2ZeroC1NotZero(Real c0, Real c1, Real DdU, Real DdPmV, Cone3 cone, Real t[2])
{
    // U is a direction vector on the cone boundary. Compute the t-value for the intersection point
    // and compute the corresponding height h to determine whether that point is on the positive cone
    // or negative cone.
    Real t = -c0 / (2 * c1);
    Real h = t * DdU + DdPmV;
    if (h > 0)
    {
        // Block 8, Figure 3(a). The line intersects the positive cone and the ray of intersection is
        // interior to the positive cone. The intersection is a ray or segment.
        return SetRayClamp(h, DdU, DdPmV, cone, t);
    }
    else
    {
        // Block 9, Figure 3(b). The line intersects the negative cone and the ray of intersection is
        // interior to the negative cone.
        return SetEmpty(t);
    }
}

int CaseC2ZeroC1Zero(Real c0, Real UdU, Real UdPmV, Real DdU, Real DdPmV, Cone3 cone, Real t[2])
{
    if (c0 != 0)
    {
        // Block 10. The line does not intersect the double-sided cone.
        return SetEmpty(t);
    }
    else
    {
        // Block 11, Figure 4. The line is on the cone boundary. The intersection with the positive cone
        // is a ray that contains the cone vertex.  The intersection is either a ray or segment.
        Real t = -UdPmV / UdU;
        Real h = t * DdU + DdPmV;
        return SetRayClamp(h, DdU, DdPmV, cone, t);
    }
}

int SetEmpty(Real t[2])
{
    t[0] = invalid_real;
    t[1] = invalid_real;
    return NONE;
}

int SetPoint(Real t0, Real t[2])
{
    t[0] = t0;
    t[1] = invalid_real;
    return POINT;
}

int SetSegment(Real t0, Real t1, Real t[2])
{
    t[0] = t0;
    t[1] = t1;
    return SEGMENT;
}

int SetRayPositive(Real t0, Real t[2])
{
    t[0] = t0;
    t[1] = invalid_real;
    return RAY_POSITIVE;
}

int SetRayNegative(Real t1, Real t[2])
{
    t[0] = invalid_real;
    t[1] = t1;
    return RAY_NEGATIVE;
```

```
}

int SetPointClamp(Real t0, Real h0, Cone3 cone, Real t[2])
{
    if (cone.HeightInRange(h0))
    {
        // P0.
        return SetPoint(t0, t);
    }
    else
    {
        // P1.
        return SetEmpty(t);
    }
}

void SetSegmentClamp(Real t0, Real t1, Real h0, Real h1, Real DdU, Real DdPmV, Cone3 cone, Real t[2])
{
    if (h1 > h0)
    {
        int numValid;
        Real overlap[2];
        if (cone.isFinite)
        {
            numValid = FindIntersection(h0, h1, cone.hmin, cone.hmax, overlap);
        }
        else
        {
            numValid = FindIntersection(h0, h1, cone.hmin, overlap);
        }

        if (numValid == 2)
        {
            // S0.
            Real t0 = (overlap[0] - DdPmV) / DdU, t1 = (overlap[1] - DdPmV) / DdU;
            return SetSegment(t0, t1, t);
        }
        else if (numValid == 1)
        {
            // S1.
            Real t0 = (overlap[0] - DdPmV) / DdU;
            return SetPoint(t0, t);
        }
        else  // numValid == 0
        {
            // S2.
            return SetEmpty(t);
        }
    }
    else  // h1 == h0
    {
        if (cone.HeightInRange(h0))
        {
            // S3. DdU > 0 and the line is not perpendicular to the cone axis.
            return SetSegment(t0, t1, t);
        }
        else
        {
            // S4. DdU == 0 and the line is perpendicular to the cone axis.
            return SetEmpty(t);
        }
    }
}

void SetRayClamp(Real h, Real DdU, Real DdPmV, Cone3 cone, Real t[2])
{
    if (cone.isFinite)
    {
        Real overlap[2];
        int numValid = FindIntersection(cone.hmin, cone.hmax, h, overlap);
        if (numValid == 2)
        {
```

```
            // R0.
            return SetSegment((overlap[0] - DdPmV) / DdU, (overlap[1] - DdPmV) / DdU, t);
        }
        else if (numValid == 1)
        {
            // R1.
            return SetPoint((overlap[0] - DdPmV) / DdU, t);
        }
        else  // numValid == 0
        {
            // R2.
            return SetEmpty(t);
        }
    }
    else
    {
        // R3.
        return SetRayPositive((max(cone.hmin, h) - DdPmV) / DdU, t);
    }
}
```

The code blocks for the find-intersection query are marked with red comments and specify a block number and, if relevant, the figure reference that illustrates the intersection. The Set*Clamp functions also have red comments that label their code blocks. These are actually included in the GTE code and have unit tests associated with each possible combination of block and clamp. The possible combinations are listed next, where B is the block number, S is segment-clamp, R is ray-clamp, P is point-clamp, F is finite cone and I is infinite cone: B0, B1S0F, B1S1F, B1S2F, B1S0I, B1S1I, B1S2I, B1S3, B1S4, B2, B3R0, B3R1, B3R2, B3R3, B4P0, B4P1, B5R0, B5R3, B6P0, B6P1, B7, B8R0, B8R3, B9, B10, B11R0 and B11R3. It is not theoretically possible for a geometric configuration to lead to B5R1, B5R2, B8R1, B8R2, B11R1 or B11R2.

5 Intersection with a Ray

When the line does not intersect the cone, neither does the ray. When the line intersects the cone (finite or infinite), let the t-interval of intersection be $[t_0, t_1]$ with $t_0 \le t_1 \le +\infty$. The ray includes an additional constraint, that $[t_0, t_1]$ overlap with the ray's t-interval $[0, +\infty)$. The final candidate interval for the ray-cone intersection is $[t_0, t_1] \cap [0, +\infty)$, which can be semiinfinite, finite, or empty. An implementation must determine which of these is the case and report the appropriate intersection points.

Listing 4 contains pseudocode for the find-intersection query between a ray and a cone. It shares several of the functions already mentioned in the line-cone find-intersection query.

Listing 4. Pseudocode for the ray-cone find-intersection query. The ray has origin ray.P and direction ray.U, where the direction is not required to be unit length.

```
// The returned 'int' is one of NONE, POINT, SEGMENT, RAY_POSITIVE or RAY_NEGATIVE.
// The returned t[] and P[] values are as described previously.
int FindIntersection(Ray3 ray, Cone3 cone, Real t[2], Vector3 P[2])
{
    // Execute the line-cone query.
    int intersectionType = DoQuery(ray.P, ray.U, cone, t);

    // Clamp the line-cone t-interval of intersection to the ray t-interval [0,+infinity).
    switch (intersectionType)
    {
        case NONE:
```

```
            break;
        case POINT:
            if (t[0] < 0)
            {
                // Block 12.
                SetEmpty(t);
            }
            // else Block 13.
            break;
        case SEGMENT:
            if (t[1] > 0)
            {
                // Block 14.
                SetSegment(max(t[0], 0), t[1], t);
            }
            else if (t[1] < 0)
            {
                // Block 15.
                SetEmpty(t);
            }
            else  // t[1] == 0
            {
                // Block 16.
                SetPoint(0, t);
            }
            break;
        case RAY_POSITIVE:
            // Block 17.
            SetRayPositive(max(t[0], 0), t);
            break;
        case RAY_NEGATIVE:
            if (t[1] > 0)
            {
                // Block 18.
                SetSegment(0, t[1], t);
            }
            else if (t[1] < 0)
            {
                // Block 19.
                SetEmpty(t);
            }
            else  // t[1] == 0
            {
                // Block 20.
                SetPoint(0, t);
            }
            break;
    }

    ComputePoints(intersectionType, ray.P, ray.U, t, P);
    return intersectionType;
}
```

The code blocks for the find-intersection query are marked with red comments and specify a block number. These are actually included in the GTE code and have unit tests associated with each block.

6 Intersection with a Segment

When the line does not intersect the cone, neither does the segment. When the line intersects the cone (finite or infinite), let the t-interval of intersection be $[t_0, t_1]$ with $t_0 \le t_1$. The segment includes an additional constraint, that $[t_0, t_1]$ overlap with the segment's t-interval $[t_2, t_3]$. The final candidate interval for the segment-cone intersection is $[t_0, t_1] \cap [t_2, t_3]$, which can be finite or empty. An implementation must determine which of these is the case and report the appropriate intersection points.

Listing 5 contains pseudocode for the find-intersection query between a ray and a cone. It shares several of the functions already mentioned in the line-cone find-intersection query.

Listing 5. Pseudocode for the segment-cone find-intersection query. The segment has endpoints segment.p[0] and segment.p[1]. The first endpoint is used as the line origin. The difference of endpoints is used as the line direction, which is not necessarily unit length.

```
// The returned 'int' is one of NONE, POINT, SEGMENT, RAY_POSITIVE or RAY_NEGATIVE.
// The returned t[] and P[] values are as described previously.
int FindIntersection(Segment3 segment, Cone3 cone, Real t[2], Vector3 P[2])
{
    // Execute the line-cone query.
    Vector3 U = segment.p[1] - segment.p[0];
    int intersectionType = DoQuery(segment.p[0], U, cone, t);

    // Clamp the line-cone t-interval of intersection to the segment t-interval [0,1].
    switch (intersectionType)
    {
        case NONE:
            break;
        case POINT:
            if (t[0] < 0 || t[0] > 1)
            {
                // Block 21.
                SetEmpty(t);
            }
            // else Block 22.
            break;
        case SEGMENT:
            if (t[1] < 0 || t[0] > 1)
            {
                // Block 23.
                SetEmpty(t);
            }
            else
            {
                Real t0 = max(0, t[0]), t1 = min(1, t[1]);
                if (t0 < t1)
                {
                    // Block 24.
                    SetSegment(t0, t1, t);
                }
                else
                {
                    // Block 25.
                    SetPoint(t0, t);
                }
            }
            break;
        case RAY_POSITIVE:
            if (1 < t[0])
            {
                // Block 26.
                SetEmpty(t);
            }
            else if (1 > t[0])
            {
                // Block 27.
                SetSegment(max(0, t[0]), 1, t);
            }
            else
            {
                // Block 28.
                SetPoint(1, t);
            }
            break;
        case RAY_NEGATIVE:
            if (0 > t[1])
```

```
            {
                // Block 29.
                SetEmpty(t);
            }
            else if (0 < t[1])
            {
                // Block 30.
                SetSegment(0, min(1, t[1]), t);
            }
            else
            {
                // Block 31.
                SetPoint(0, t);
            }
            break;
    }

    ComputePoints(intersectionType, segment.p[0], U, t, P);
    return intersectionType;
}
```

The code blocks for the find-intersection query are marked with red comments and specify a block number. These are actually included in the GTE code and have unit tests associated with each block.

7 Rational and Symbolic Arithmetic to Avoid Square Roots

The Listings 3, 4 and 5 are written as if Real represents the set of all real numbers. Naturally, this is not possible for a computer implementation. Typically, we use floating-point types such as 32-bit float or 64-bit double. As with any floating-point implementation of a geometric algorithm, rounding errors inherent in floating-point arithmetic can lead to erroneous results. In particular, the code is heavy with floating-point comparisons, so any rounding errors that occur in computing the numbers to be compared can lead to incorrect classifications of the particular geometric configuration of the input data. If the implementation is robust, the computed output and the theoretical output are similar enough that the computed output is an acceptable result for the application at hand.

The goal of this document is to provide an arbitrary precision implementation to avoid the rounding errors yet still have an implementation that is fast enough for an application. The floating-point inputs are rational numbers and can be converted to an arithmetic system that has a data type for such numbers. In GTE, these classes are BSNumber and BSRational. The presence of divisions in the algorithm requires use of BSRational. Arithmetic computations involving addition, subtraction, multiplication and division can be performed without errors.

There are three places in the algorithm where rational arithmetic cannot immediately help.

1. The cone has an angle θ for which we need to compute $\cos^2\theta$. The value of $\cos\theta$ can be any real number in $[-1, 1]$, and if it is irrational, we cannot represent it exactly. In some special cases, $\cos\theta$ is irrational but $\cos^2\theta$ is rational; for example, this is the case when $\theta = \pi/4$, whereby $\cos\theta = 1/\sqrt{2}$ and $\cos^2\theta = 1/2$. Generally, you must estimate $\cos^2\theta$ by a rational number to whatever number of significant digits makes sense for your application. This rational number is then assumed to be an error-free input.

2. We must compute the roots to the quadratic polynomial $c_2t^2 + 2c_1t + c_0$. The real-valued roots are

$(-c_1 \pm \sqrt{c_1^2 - c_0 c_2})/c_2$. The discriminant is $\delta = c_1^2 - c_0 c_2$. When $\delta > 0$, the roots are irrational numbers. Floating-point systems provide estimates of the square roots, but these have rounding errors.

3. Although we saw that the line direction $\mathbf{U}$ is not required to be unit length, the cone axis direction $\mathbf{D}$ was assumed to be unit length. In an application, one might choose a cone axis direction vector $\mathbf{E}$ that is not unit length and then normalize it to obtain $\mathbf{D} = \mathbf{E}/|\mathbf{E}|$. By doing so, we must compute the length $|\mathbf{E}|$ which involves yet another square root operation that will have rounding errors. If you compute the squared length $\mathbf{D} \cdot \mathbf{D}$ using rational arithmetic, the result is typically not 1 (although it will be approximately 1).

Item 1 is up to the user of the line-cone find-intersection code to provide as accurate and precise an estimate of $\cos^2 \theta$ as is required by the application.

An illustration of the problem with rounding errors in item 3 is provided next. A unit-length direction vector is usually provided by specifying a 3-tuple $\mathbf{E} = (e_0, e_1, e_2)$ and then normalizing it to $\mathbf{D} = \mathbf{E}/|\mathbf{E}| = (e_0, e_1, e_2)/\sqrt{e_0^2 + e_1^2 + e_2^2}$. When computing with floating-point arithmetic, the square root operation and the division usually have floating-point rounding errors. The computed vector is actually $\hat{\mathbf{D}}$ and is only an approimation to $\mathbf{D}$, so it does not have unit length. Listing 6 shows code that illustrates the rounding errors when normalizing a vector.

Listing 6. Approximation errors when normalizing a vector to obtain a unit-length vector. The code uses the GTE library.

```
using Rational = BSNumber<UIntegerAP32>;

// Normalize a float-component 3-tuple.
Vector3<float> fE = { 1.0f, 2.0f, 3.0f };
Vector3<float> fD = fE / Length(fE);
// Represent fD as a rational-component 3-tuple.
Vector3<Rational> rD = { fD[0], fD[1], fD[2] };
// Compute the exact squared length of rD.
Rational rSqrLen = Dot(rD, rD);
// = 0x0003FFFF FA4757A5 * 2^{-50}
// = 1 - epsilon
// < 1
// = 0x00040000 00000000 * 2^{-50}
float fSqrLen = (float)rSqrLen;  // 0.999999940

// Normalize a double-component 3-tuple.
Vector3<double> dE = { 1.0, 2.0, 3.0 };
Vector3<float> dD = dE / Length(dE);
// Represent dE as a rational-component 3-tuple.
Rational rD = { dD[0], dD[1], dD[2] };
// Compute the exact squared length of rD.
Rational rSqrLen = Dot(rD, rD);
// = 0x00000200 00000000 000CC8B2 FF10B80F * 2^{-106}
// = 1 + delta
// > 1
// = 0x00000200 00000000 00000000 00000000 * 2^{-106}
double dSqrLen = (double)rSqrLen;  // 1.0000000000000000
```

The float-component vector fE is normalized using floating-point arithmetic, but the result is a floating-point vector that when represented as a rational vector has squared length smaller than 1. The rounding error is within float precision, so fSqrLen shows there is error. The double-component vector dE is normalized using floating-point arithmetic, but the result is a floating-point vector that when represented as a rational

vector has squared length larger than 1. The rounding error is smaller than double precision can represent, so dSqrLen makes it appear as if there is no error (because the result is rounded to 1).

Items 2 and are handled by using a mixture of rational arithmetic and symbolic computing to obtain an exact result. This is the topic of the next section, which first assumes you have a unit-length **D**. The section after that one makes additional modifications to allow for a non-unit-length vector **E** in the direction of the cone axis.

7.1 Symbolic Arithmetic for the Square Root of the Discriminant

The roots of the quadratic polynomial $c_2t^2 + 2c_1t + c_0$ are symbolically $t = (-c_1 \pm \sqrt{d})/c_2$, where the discriminant is $d = c_1^2 - c_0c_2$. Whether you use floating-point or rational arithmetic for computing δ, avoiding rounding errors when estimating $\sqrt{d}$ requires symbolic manipulation.

The roots are of the form $x + y\sqrt{d}$, where x, y and d are rational numbers. I will refer to these as *rational quadratic numbers*. When d is not the square of a rational number (*i.e.* $\sqrt{d}$ is irrational), the rational quadratic numbers are denoted by $\mathbb{Q}[d]$, which is a *rational quadratic field* (a term from abstract algebra). Rational quadratic fields are endowed with an addition operation and a multiplication operations. The elements have additive inverses and multiplicative inverses, thus supporting the arithmetic operations of addition, subtraction, multiplication and division:

$$\begin{aligned}
(x_0 + y_0\sqrt{d}) + (x_1 + y_1\sqrt{d}) &= (x_0 + x_1) + (y_0 + y_1)\sqrt{d} \\
(x_0 + y_0\sqrt{d}) - (x_1 + y_1\sqrt{d}) &= (x_0 - x_1) + (y_0 - y_1)\sqrt{d} \\
(x_0 + y_0\sqrt{d}) * (x_1 + y_1\sqrt{d}) &= (x_0x_1 + y_0y_1d) + (x_1y_0 + x_0y_1)\sqrt{d} \\
(x_0 + y_0\sqrt{d})/(x_1 + y_1\sqrt{d}) &= \tfrac{x_0x_1 - y_0y_1d}{x_1^2 - y_1^2d} + \tfrac{x_1y_0 - x_0y_1}{x_1^2 - y_1^2d}\sqrt{d}
\end{aligned} \tag{7}$$

7.2 Symbolic Arithmetic for the Length of the Cone Axis Direction

8 Pseudocode for Rational and Symbolic Arithmetic