Intersection of Swept Ellipses and Swept Ellipsoids

David Eberly
Geometric Tools, LLC
http://www.geometrictools.com/

Created: May 9, 2009
Last Modified: October 15, 2010

Contents

1 Introduction

Consider two ellipses that are initially separated and are moving with constant linear velocities. The first problem is to determine whether or not the ellipses intersect at a later time. If they do intersect, the second problem is to determine the contact time and contact point of the ellipses. The contact time is the smallest positive time $t_{\text{contact}} > 0$ such that the two ellipses are separated for times $0 \le t < t_{\text{contact}}$ and just touching at time $t = t_{\text{contact}}$. This document describes an algorithm for solving these problems.

At time zero, the ellipses are defined by quadratic equations in standard form. The first ellipse is defined by

$$(\mathbf{X} - \mathbf{K}_0)^{\mathrm{T}} M_0 (\mathbf{X} - \mathbf{K}_0) = 1 \tag{1}$$

The center of the ellipse is $\mathbf{K}_0$. The matrix M_0 is positive definite and can be factored as $M_0 = R_0 D_0 R_0^{\mathrm{T}}$, where R_0 is a rotation matrix and D_0 is a diagonal matrix whose diagonal entries are positive. The diagonal values of D_0 are eigenvalues of M_0 and the columns of R_0 are eigenvectors of M_0. The second ellipse is defined by

$$(\mathbf{X} - \mathbf{K}_1)^{\mathrm{T}} M_1 (\mathbf{X} - \mathbf{K}_1) = 1 \tag{2}$$

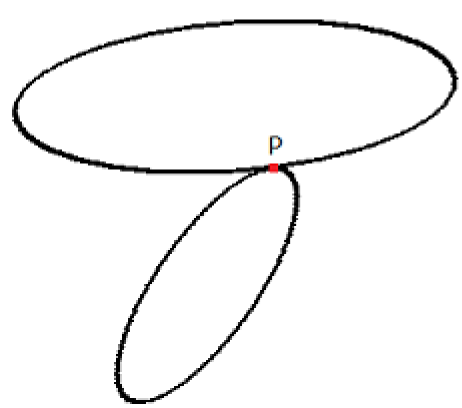
where $\mathbf{K}_1$ is the center and where $M_1 = R_1 D_1 R_1^{\mathrm{T}}$ is positive definite.

The ellipses are moving with constant linear velocities $\mathbf{V}_0$ and $\mathbf{V}_1$, where the indices make it clear which velocity goes with which ellipse. The ellipse centers at time $t \ge 0$ are $\mathbf{K}_0 + t\mathbf{V}_0$ and $\mathbf{K}_1 + t\mathbf{V}_1$. To simplify the problem, we may use relative velocity; the first ellipse is assumed to be stationary and the second ellipse is moving with constant linear velocity $\mathbf{V} = \mathbf{V}_1 - \mathbf{V}_0$.

2 The Configuration at Contact Time

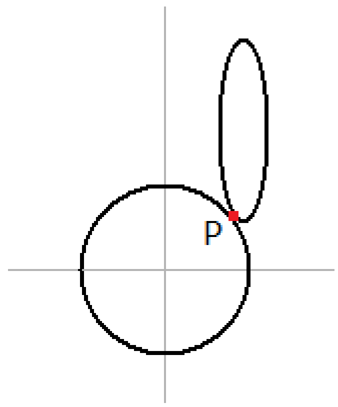
At the contact time, the contact set must contain a single point because of the convexity of the ellipses. Moreover, the ellipses must be tangent at the contact point. Figure 2.1 illustrates this.

Figure 2.1 The configuration of two ellipses at contact time. The contact point is $\mathbf{P}$.



Let us analyze this configuration to determine the location of the contact point. We make several changes of variables to reduce the problem to one involving a circle of radius 1 and centered at the origin and an ellipse that is axis-aligned. The transformations are affine and invertible, so the contact set still contains a single point. Figure 2.2 shows such a configuration.

Figure 2.2 The transformed configuration to a circle of radius 1 with center at the origin and an axis-aligned ellipse.



3 Reduction to Circle and Axis-Aligned Ellipse

For the sake of argument, we will use Equations (1) and (2) as if the contact time were $T = 0$.

The first change of variables is

$$\mathbf{Y} = D_0^{1/2} R_0^{\mathrm{T}} \left(\mathbf{X} - \mathbf{K}_0\right) \tag{3}$$

where $D_0^{1/2}$ is the diagonal matrix whose diagonal entries are the square roots of the diagonal entries of D_0. Equation (1) transforms to

$$\mathbf{Y}^{\mathrm{T}} \mathbf{Y} = 1 \tag{4}$$

which represents a circle of radius 1 centered at the origin. Define

$$\mathbf{K}_2 = D_0^{1/2} R_0^{\mathrm{T}} \left(\mathbf{K}_1 - \mathbf{K}_0\right) \tag{5}$$

and

$$M_2 = D_0^{-1/2} R_0^{\mathrm{T}} R_1 D_1 R_1^{\mathrm{T}} R_0 D_0^{-1/2} \tag{6}$$

Equations (3), (5), and (6) transform Equation (2) to

$$\left(\mathbf{Y} - \mathbf{K}_2\right)^{\mathrm{T}} M_2 \left(\mathbf{Y} - \mathbf{K}_2\right) = 1 \tag{7}$$

The matrix M_2 may be factored using an eigendecomposition to $M_2 = RDR^{\mathrm{T}}$, where D is a diagonal matrix and R is a rotation matrix. The second change of variables is

$$\mathbf{Z} = R^{\mathrm{T}} \mathbf{Y} \tag{8}$$

Equation (4) is transformed to

$$\mathbf{Z}^{\mathrm{T}}\mathbf{Z} = 1 \tag{9}$$

which is the same circle, but rotated about the origin. Defining

$$\mathbf{K} = R^{\mathrm{T}}\mathbf{K}_2 \tag{10}$$

Equation (7) is transformed to

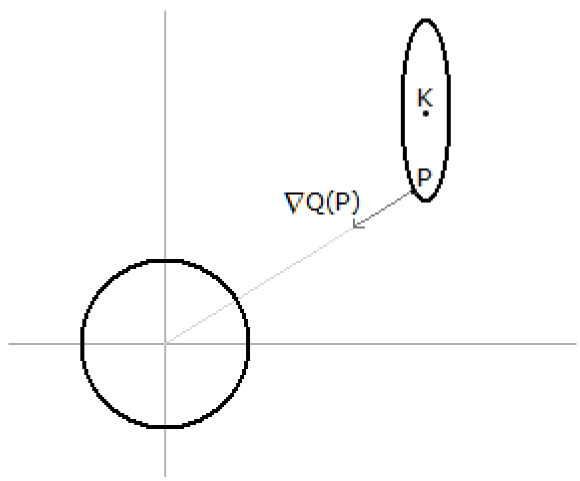
$$(\mathbf{Z} - \mathbf{K})^{\mathrm{T}} D\,(\mathbf{Z} - \mathbf{K}) = 1 \tag{11}$$

Because D is diagonal, this equation represents an axis-aligned ellipse with center point $\mathbf{K}$, as illustrated in Figure 3.1. Define

$$Q(\mathbf{Z}) = (\mathbf{Z} - \mathbf{K})^{\mathrm{T}} D\,(\mathbf{Z} - \mathbf{K}) - 1 \tag{12}$$

The ellipse is implicitly defined by $Q(\mathbf{Z}) = 0$, and the gradient vector $\nabla Q(\mathbf{Z})$ is an outer-pointing normal to the point $\mathbf{Z}$ on the ellipse.

Figure 3.1 The transformed configuration to a circle of radius 1 with center at the origin and an axis-aligned ellipse. The circle and ellipse are separated. The point $\mathbf{P}$ is the ellipse point that is closest to the origin. An outer-pointing normal at this point is $\nabla Q(\mathbf{P})$.



4 Computing the Closest Point P

The point $\mathbf{P}$ shown in Figure 3.1 is the ellipse point closest to the origin (and to the circle). We know that $\mathbf{P}$ is on the ellipse, so $Q(\mathbf{P}) = 0$. An outer-pointing normal to the ellipse at $\mathbf{P}$ is the gradient vector $\nabla Q(\mathbf{P}) = 2D(\mathbf{P} - \mathbf{K})$. In fact, this vector is parallel to $\mathbf{P}$ but in the opposite direction; thus, there is some scalar $s < 0$ for which

$$sD(\mathbf{P} - \mathbf{K}) = \mathbf{P} \tag{13}$$

If $\mathbf{P} = (p_0, p_1)$, $\mathbf{K} = (k_0, k_1)$, and $D = \mathrm{Diag}(d_0, d_1)$ with $d_1 \geq d_0$, then Equation (13) may be solved for

$$p_0 = \frac{d_0 k_0 s}{d_0 s - 1}, \quad p_1 = \frac{d_1 k_1 s}{d_1 s - 1} \tag{14}$$

Knowing that $\mathbf{P}$ is on the ellipse, we have

$$0 = Q(\mathbf{P}) = \frac{d_0 k_0^2}{(d_0 s - 1)^2} + \frac{d_1 k_1^2}{(d_1 s - 1)^2} - 1 \tag{15}$$

We will show that Equation (15) has a unique negative root $\bar{s}$.

The summary of the algorithm is the following. It assumes that the eigendecompositions for M_0 and M_1 are computed once and stored with the ellipse data structure.

1. Given two separated ellipses, $\mathcal{E}_0 = \{\mathbf{K}_0, R_0, D_0\}$ and $\mathcal{E}_1 = \{\mathbf{K}_1, R_1, D_1\}$.
2. Compute $\mathbf{K}_2 = D_0^{1/2} R_0^{\mathrm{T}} (\mathbf{K}_1 - \mathbf{K}_0)$.
3. Compute $M_2 = D_0^{-1/2} R_0^{\mathrm{T}} R_1 D_1 R_1^{\mathrm{T}} R_0 D_0^{-1/2}$.
4. Use an eigendecomposition to factor $M_2 = R D R^{\mathrm{T}}$ with $D = \mathrm{Diag}(d_0, d_1)$.
5. Compute $\mathbf{K} = R^{\mathrm{T}} \mathbf{K}_2 = (k_0, k_1)$.
6. Compute the unique negative root $\bar{s}$ to Equation (15).
7. Compute $\mathbf{P} = (d_0 k_0 \bar{s}/(d_0 \bar{s} - 1), d_1 k_1 \bar{s}/(d_1 \bar{s} - 1))$.

What remains is to determine an algorithm for computing $\bar{s}$.

5 Computing the Negative Root

Equation (15) may be rewritten as a quartic polynomial equation:

$$p(s) = (d_0 s - 1)^2 (d_1 s - 1)^2 - d_0 k_0^2 (d_1 s - 1)^2 - d_1 k_1^2 (d_0 s - 1)^2 = 0 \tag{16}$$

The closed-form equations for the roots of a quartic may be used to compute the real-valued roots, one of which will be negative. These equations are known to be numerically ill-conditioned, and we do not need all the roots.

An algorithm that is better suited for numerical computations is described next. Define the function

$$f(s) = \frac{d_0 k_0^2}{(d_0 s - 1)^2} + \frac{d_1 k_1^2}{(d_1 s - 1)^2} - 1 \tag{17}$$

The sets of roots of $p(s)$ and $f(s)$ are the same. The first derivative is

$$f'(s) = \frac{-2 d_0^2 k_0^2}{(d_0 s - 1)^3} + \frac{-2 d_1^2 k_1^2}{(d_1 s - 1)^3} \tag{18}$$

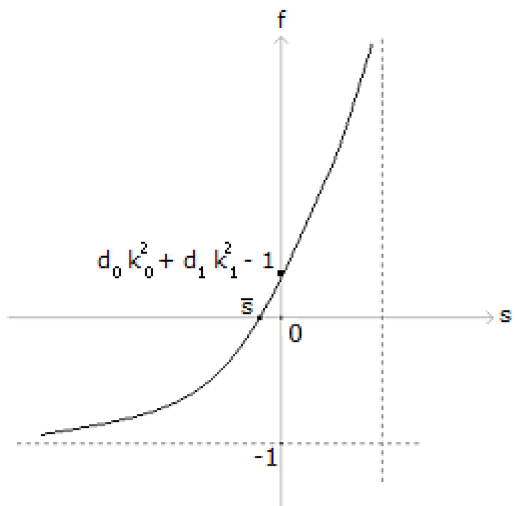
and the second derivative is

$$f''(s) = \frac{6d_0^3k_0^2}{(d_0s-1)^4} + \frac{6d_1^3k_1^2}{(d_1s-1)^4} > 0 \tag{19}$$

I have indicated that the second derivative is always positive, which makes $f(s)$ a *convex function*. Such functions are ideal for locating roots using Newton's method.

Part of the graph of $f(s)$ is illustrated by Figure 5.1.

Figure 5.1 The left-most branch of the graph of $f(s)$. The vertical asymptote is at $s = 1/d_1$ when $k_1 \neq 0$ or at $s = 1/d_0$ when $k_1 = 0$.



Because we want only the negative roots, the left-most branch of the graph is of interest. A simple calculation shows that

$$f(0) = d_0k_0^2 + d_1k_1^2 - 1 = Q(\mathbf{0}) > 0 \tag{20}$$

The positivity is guaranteed because the origin $\mathbf{0}$ is outside the ellipse. Also,

$$\lim_{s \to -\infty} f(s) = -1 \tag{21}$$

The convexity of f and Equations (20) and (21) guarantee that $f(s)$ has a unique negative root $\bar{s}$. We may use Newton's method to numerically estimate this root:

$$s_0 = 0; \quad s_{i+1} = s_i - \frac{f(s_i)}{f'(s_i)}, \; i \geq 0 \tag{22}$$

The convexity of f guarantees convergence of the sequence, and in practice the number of iterates for an accurate estimate is small.

6 Computing the Contact Time

In the previous construction, the ellipses were stationary and assumed to be in tangential contact. When the ellipses are moving, we replace in the construction $\mathbf{K}_0$ by $\mathbf{K}_0 + t\mathbf{V}_0$ and $\mathbf{K}_1$ by $\mathbf{K}_1 + t\mathbf{V}_1$. The center of the axis-aligned ellipse is time varying,

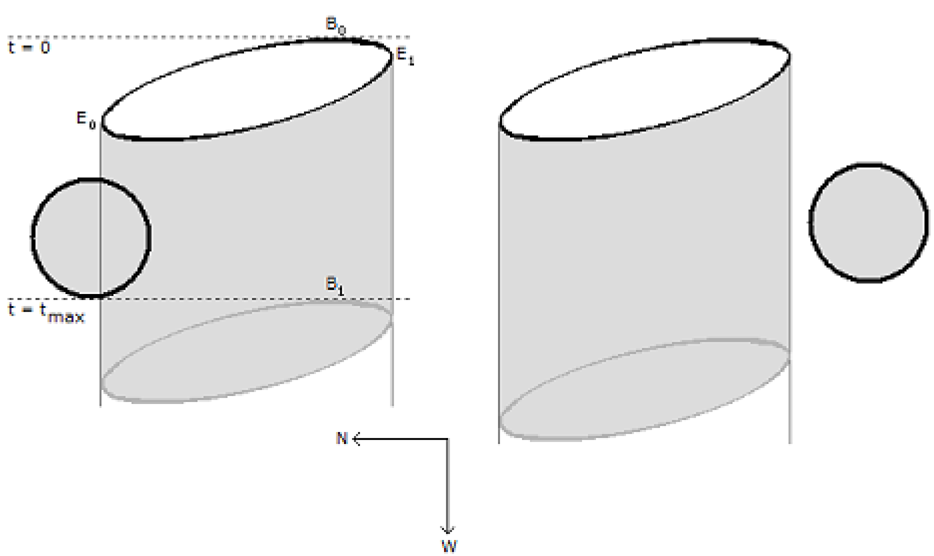
$$\mathbf{C}(t) = R^{\mathrm{T}} D_0^{1/2} R_0^{\mathrm{T}} (\mathbf{K}_1 - \mathbf{K}_0 + t\mathbf{V}) = \mathbf{K} + t\mathbf{W} \tag{23}$$

where $\mathbf{V} = \mathbf{V}_1 - \mathbf{V}_0$ is the relative velocity. The last equality defines the velocity $\mathbf{W}$, which is the relative velocity of the axis-aligned ellipse compared to the stationary circle. For notation's sake, let $\mathbf{K} = (k_0, k_1)$, $\mathbf{W} = (w_0, w_1)$, and $\mathbf{C} = (c_0, c_1) = (k_0 + tw_0, k_1 + tw_1)$.

The circle and ellipse are initially separated, as Figure 3.1 illustrates. A quick-out test is convenient here. The initially closest point $\mathbf{P}$ to the circle moves with velocity $\mathbf{W}$. If the velocity vector forms an acute angle with $\mathbf{P}$, then initially $\mathbf{P}$ moves away from the circle and must do so for all later times. Thus, when $\mathbf{W} \cdot \mathbf{P} \geq 0$, the ellipse will never intersect the circle.

When $\mathbf{W} \cdot \mathbf{P} < 0$, we would like to know whether or not the ellipse will intersect the circle. The ellipse sweeps a region $\mathcal{R}$ that is bounded by two parallel lines. An intersection between the ellipse and circle will occur if and only if the circle intersects the region $\mathcal{R}$. This is a tractable problem. The bounding parallel lines have direction $\mathbf{W}$ and pass through the two extreme points in the perpendicular direction $\mathbf{W}^{\perp} = (w_1, -w_0)$. Determining whether or not the circle intersects these lines, or is between them, is a matter of algebra. Figure 6.1 illustrates the geometric relationship between the circle and $\mathcal{R}$.

Figure 6.1 The ellipse sweeps a region bounded by two parallel lines. Left: The ellipse will intersect the circle when the circle intersects the swept region. Right: The ellipse will not intersect the circle when the circle is disjoint from the swept region.



The extreme points $\mathbf{E}_0$ and $\mathbf{E}_1$ are in the directions of $\pm\mathbf{N}$ where $\mathbf{N} = \mathbf{W}^{\perp}$. The point $\mathbf{B}_0$ is extreme in the direction of $-\mathbf{W}$. The extreme point $\mathbf{B}_1 = \mathbf{B}_0 + t_{\max}\mathbf{W}$ is chosen such that the moving ellipse has completely passed the circle. Thus, the contact time is in the interval $[0, t_{\max}]$.

The algorithm presented here for computing the contact point has the circle-region intersection test in it implicitly. There is a trade-off: The implicit test requires more computational cycles than the direct test. But the ideas here apply to higher-dimensional ellipsoids, where the direct test becomes more computationally intensive. For example, the direct test in 3D requires testing whether a sphere intersects the ellipsoid-sweep region which happens to be a right-elliptical cylinder. Computing the equation of the elliptical cylinder is tedious, but tractable. The intersection test can then be reduced to a 2D problem of testing whether or not a circle and ellipse overlap. The algorithm here has no component for reduction of dimension, which makes it easy to adapt to higher dimensions.

With the introduction of the time parameter, the function of Equation (17) is now dependent on two parameters,

$$f(s,t) = \frac{d_0 c_0^2}{(d_0 s - 1)^2} + \frac{d_1 c_1^2}{(d_1 s - 1)^2} - 1 \tag{24}$$

where c_0 and c_1 are linear functions of t. For each time t, we may compute the negative root of $f(s,t) = 0$, call it $s(t)$ to indicate that it now depends on the choice of t (it is a function of t). For notational simplicity, I have dropped the overbar on s. It is therefore the case that $f(s(t),t) \equiv 0$; as a function of t, f is zero for all t. This allows us to compute the rate of change of s with respect to t. Using the chain rule from Calculus,

$$f_s(s(t),t)s'(t) + f_t(s(t),t) \equiv 0 \tag{25}$$

where $f_s = \partial f/\partial s$ and $f_t = \partial f/\partial t$ are the first-order partial derivatives of f,

$$f_s = \frac{-2d_0^2 c_0^2}{(d_0 s - 1)^3} + \frac{-2d_1^2 c_1^2}{(d_1 s - 1)^3}, \quad f_t = \frac{d_0 (c_0^2)'}{(d_0 s - 1)^2} + \frac{d_1 (c_1^2)'}{(d_1 s - 1)^2} \tag{26}$$

The derivative of $s(t)$ is therefore

$$s'(t) = -\frac{f_t(s(t),t)}{f_s(s(t),t)} \tag{27}$$

Let $\mathbf{P}(t)$ be the ellipse point closest to the origin at time t. We are trying to compute the contact time t_{contact} for which the ellipse is just touching the circle. When this happens, the ellipse point $\mathbf{P}(t_{\text{contact}})$ closest to the origin must be on the circle; that is, $|\mathbf{P}(t_{\text{contact}})|^2 - 1 = 0$. Define the function $h(t) = |\mathbf{P}(t)|^2 - 1$. As a function of s and t, the right-hand side of the equation is

$$g(s,t) = \frac{d_0^2 c_0^2 s^2}{(d_0 s - 1)^2} + \frac{d_1^2 c_1^2 s^2}{(d_1 s - 1)^2} - 1 \tag{28}$$

in which case $h(t) = g(s(t),t)$, where s is the negative root at time t of $f(s,t) = 0$. Using the chain rule, the first derivative of $h(t)$ is

$$h'(t) = g_s(s(t),t)s'(t) + g_t(s(t),t) = \frac{2d_0(c_0^2)'s}{d_0 s - 1} + \frac{2d_1(c_1^2)'s}{d_1 s - 1} \tag{29}$$

where

$$g_s = \frac{-2d_0^2 c_0^2 s}{(d_0 s - 1)^3} + \frac{-2d_1^2 c_1^2 s}{(d_1 s - 1)^3}, \quad g_t = \frac{d_0^2 (c_0^2)' s^2}{(d_0 s - 1)^2} + \frac{d_1^2 (c_1^2)' s^2}{(d_1 s - 1)^2} \tag{30}$$

The right-most equality of Equation (29) was obtained by substituting into $g_s(s,t)s' + g_t(t,s)$ the expressions of Equations (30), (27), and (26). I used Mathematica to simplify the resulting expression to that shown in the right-most equality.

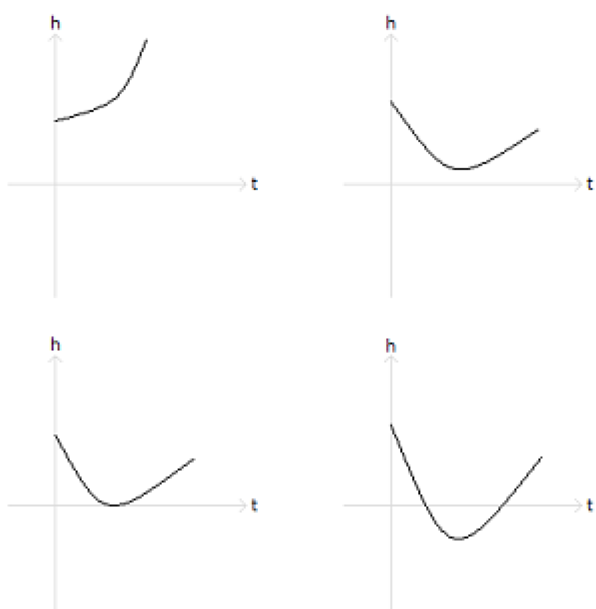
Half the second derivative of $h''(t)$ may be computed from the right-most equality of Equation (29),

$$\frac{h''(t)}{2} = \left[\frac{d_0 s(c_0^2)''}{d_0 s - 1} + \frac{d_1 s(c_1^2)''}{d_1 s - 1}\right] - \left[\frac{\left(\frac{d_0(c_0^2)'}{(d_0 s-1)^2} + \frac{d_1(c_1^2)'}{(d_1 s-1)^2}\right)^2}{\left(\frac{d_0^2 c_0^2}{(d_0 s-1)^3} + \frac{d_1^2 c_1^2}{(d_1 s-1)^3}\right)}\right] > 0 \tag{31}$$

where the positivity is guaranteed as long as s is negative, which it is by construction (it is a negative root of a function). To verify the positivity, the first square-bracketed expression is positive because $(c_0^2)'' = 2w_0^2$, $(c_1^2)'' = 2w_1^2$ and because $d_0 s/(d_0 s - 1) > 0$ and $d_1 s/(d_1 s - 1) > 0$ as long as $s < 0$. The numerator of the second square-bracketed term is positive because it is the square of an expression. The denominator is negative because $(d_0 s - 1)^3 < 0$ and $(d_1 s - 1)^3 < 0$ as long as $s < 0$. The negative sign in front of the second square-bracketed term makes the denominator positive. Thus, the combination of the expressions on the right-hand side of Equation (31) are positive as long as s is negative.

I have shown that $h(0) > 0$, because the initial closest point is outside the circle, and that $h''(t) > 0$, in which case $h(t)$ is a convex function. Figure 6.2 illustrates four different possibilities for the graph of h.

Figure 6.2 Possibilities for the graph of $h(t)$. Upper left: The ellipse is moving away from the circle, so there is no intersection at any time. Upper right: The ellipse initially moves towards the circle, but then veers away and passes it without intersection. Lower left: The ellipse moves towards the circle and just grazes it at the contact time. Lower right: The ellipse moves towards the circle, just touching the circle at contact time, and then interpenetrates for a period of time.



Because $h(t)$ is convex, we may use Newton's method to locate the smallest positive root,

$$t_0 = 0; \;\; t_{i+1} = t_i - \frac{h(t_i)}{h'(t_i)}, \; i \geq 0 \tag{32}$$

but with one constraint. In the lower-left and lower-right images of Figure 6.2, the iterates are guaranteed to be on the left of the root of h. Moreover, the derivatives at the iterates are always negative. During Newton's method, if we ever discover that $h'(t_i) \geq 0$, then we know we have one of the graphs of the upper-left and upper-right images of the figure. In this case, we terminate the iteration and conclude that the ellipse and circle do not intersect.

When we do have a contact point, it is in the transformed coordinates. Let the computed contact point be

$$\mathbf{Z}_{\text{contact}} = \mathbf{P}(t_{\text{contact}}) \tag{33}$$

We then compute

$$\mathbf{Y}_{\text{contact}} = R\mathbf{Z}_{\text{contact}} \tag{34}$$

In the original coordinate system, the contact point is

$$\mathbf{X}_{\text{contact}} = \mathbf{K}_0 + R_0 D_0^{-1/2} \mathbf{Y}_{\text{contact}} \tag{35}$$

7 Source Code for Swept Ellipses

Here is Wild Magic code for computing the contact time and contact point, if they exist, for two moving ellipses.

```
double ComputeClosestPoint (const Matrix2d& D, const Vector2d& K,
    Vector2d& closestPoint)
{
    double d0 = D[0][0], d1 = D[1][1];
    double k0 = K[0], k1 = K[1];
    double d0k0 = d0*k0;
    double d1k1 = d1*k1;
    double d0k0k0 = d0k0*k0;
    double d1k1k1 = d1k1*k1;

    if (d0k0k0 + d1k1k1 - 1.0 < 0.0)
    {
        // The ellipse contains the origin, so the ellipse and circle are
        // overlapping.
        return FLT_MAX;
    }

    const int maxIterations = 128;
    const double epsilon = 1e-08;

    double s = 0.0;
    int i;
    for (i = 0; i < maxIterations; ++i)
    {
        // Compute f(s).
        double tmp0 = d0*s - 1.0;
        double tmp1 = d1*s - 1.0;
        double tmp0sqr = tmp0*tmp0;
        double tmp1sqr = tmp1*tmp1;
        double f = d0k0k0/tmp0sqr + d1k1k1/tmp1sqr - 1.0;

        if (fabs(f) < epsilon)
        {
            // We have found a root.
            break;
        }

        // Compute f'(s).
        double tmp0cub = tmp0*tmp0sqr;
        double tmp1cub = tmp1*tmp1sqr;
        double fder = -2.0*(d0*d0k0k0/tmp0cub + d1*d1k1k1/tmp1cub);

        // Compute the next iterate sNext = s - f(s)/f'(s).
        s -= f/fder;
    }

    // assert:  i < maxIterations

    closestPoint[0] = d0k0*s/(d0*s - 1.0);
    closestPoint[1] = d1k1*s/(d1*s - 1.0);
    return s;
}
```

```
bool ComputeContact (const Matrix2d& D, const Vector2d& K, const Vector2d& W,
    double& contactTime, Vector2d& Zcontact)
{
    double d0 = D[0][0], d1 = D[1][1];
    double k0 = K[0], k1 = K[1];
    double w0 = W[0], w1 = W[1];

    int maxIterations = 128;
    double epsilon = 1e-08;

    contactTime = 0.0;
    for (int i = 0; i < maxIterations; ++i)
    {
        // Compute h(t).
        Vector2d Kmove = K + contactTime*W;
        double s = ComputeClosestPoint(D, Kmove, Zcontact);
        double tmp0 = d0*Kmove[0]*s/(d0*s - 1.0);
        double tmp1 = d1*Kmove[1]*s/(d1*s - 1.0);
        double h = tmp0*tmp0 + tmp1*tmp1 - 1.0;

        if (fabs(h) < epsilon)
        {
            // We have found a root.
            return true;
        }

        // Compute h'(t).
        double hder = 4.0*tmp0*w0 + 4.0*tmp1*w1;
        if (hder > 0.0)
        {
            // The ellipse cannot intersect the circle.
            return false;
        }

        // Compute the next iterate tNext = t - h(t)/h'(t).
        contactTime -= h/hder;
    }

    ComputeClosestPoint(D, K + contactTime*W, Zcontact);
    return true;
}
```

```
bool FindContact (const Ellipse2d &ellipse0, const Vector2d& velocity0,
    const Ellipse2d &ellipse1, const Vector2d& velocity1, double& contactTime,
    Vector2d& contactPoint)
{
    // Get the parameters of ellipse0.
    Vector2d K0 = ellipse0.Center;
    Matrix2d R0(ellipse0.Axis, true);
    Matrix2d D0(
        1.0/(ellipse0.Extent[0]*ellipse0.Extent[0]),
        1.0/(ellipse0.Extent[1]*ellipse0.Extent[1]));

    // Get the parameters of ellipse1.
    Vector2d K1 = ellipse1.Center;
    Matrix2d R1(ellipse1.Axis, true);
    Matrix2d D1(
        1.0/(ellipse1.Extent[0]*ellipse1.Extent[0]),
        1.0/(ellipse1.Extent[1]*ellipse1.Extent[1]));

    // Compute K2.
    Matrix2d D0NegHalf(ellipse0.Extent[0], ellipse0.Extent[1]);
    Matrix2d D0Half(1.0/ellipse0.Extent[0], 1.0/ellipse0.Extent[1]);
    Vector2d K2 = D0Half*((K1 - K0)*R0);

    // Compute M2.
    Matrix2d R1TR0D0NegHalf = R1.TransposeTimes(R0*D0NegHalf);
    Matrix2d M2 = R1TR0D0NegHalf.TransposeTimes(D1)*R1TR0D0NegHalf;

    // Factor M2 = R*D*R^T.
    Matrix2d R, D;
    M2.EigenDecomposition(R, D);

    // Compute K.
    Vector2d K = K2*R;

    // Compute W.
    Vector2d W = (D0Half*((velocity1 - velocity0)*R0))*R;

    // Transformed ellipse0 is Z^T*Z = 1 and transformed ellipse1 is
    // (Z-K)^T*D*(Z-K) = 0.

    // Compute the initial closest point.
    Vector2d P0;
    if (ComputeClosestPoint(D, K, P0) >= 0.0)
    {
        // The ellipse contains the origin, so the ellipses were not
        // separated.
        return false;
    }
    double dist0 = P0.Dot(P0) - 1.0;
    if (dist0 < 0.0)
    {
        // The ellipses are not separated.
        return false;
    }

    Vector2d Zcontact;
    if (!ComputeContact(D, K, W, contactTime, Zcontact))
    {
        return false;
    }

    // Transform contactPoint back to original space.
    contactPoint = K0 + R0*D0NegHalf*R*Zcontact;
    return true;
}
```

8 Extension to Ellipsoids

The reduction of swept ellipses to the circle and axis-aligned ellipse is shown in Equations (3) through (12). The construction is independent of dimension, so it also applies to swept ellipsoids.

Using the same notation as in Section 4, let $\mathbf{P} = (p_0, p_1, p_2)$, $\mathbf{K} = (k_0, k_1, k_2)$, and $D = \mathrm{Diag}(d_0, d_1, d_2)$. Equation (15) becomes

$$0 = Q(\mathbf{P}) = \frac{d_0 k_0^2}{(d_0 s - 1)^2} + \frac{d_1 k_1^2}{(d_1 s - 1)^2} + \frac{d_2 k_2^2}{(d_2 s - 1)^2} - 1 \tag{36}$$

The function whose negative root determines the closest ellipse point is now as shown together with its derivatives

$$f(s) = \frac{d_0 k_0^2}{(d_0 s - 1)^2} + \frac{d_1 k_1^2}{(d_1 s - 1)^2} + \frac{d_2 k_2^2}{(d_2 s - 1)^2} - 1 \tag{37}$$

and

$$f'(s) = \frac{-2 d_0^2 k_0^2}{(d_0 s - 1)^3} + \frac{-2 d_1^2 k_1^2}{(d_1 s - 1)^3} + \frac{-2 d_2^2 k_2^2}{(d_2 s - 1)^3} \tag{38}$$

and

$$f''(s) = \frac{6 d_0^3 k_0^2}{(d_0 s - 1)^4} + \frac{6 d_1^3 k_1^2}{(d_1 s - 1)^4} + \frac{6 d_2^3 k_1^2}{(d_2 s - 1)^4} > 0 \tag{39}$$

so $f(s)$ is a convex function that has a unique negative root $\bar{s}$. Newton's method may be used to compute robustly $\bar{s}$.

The moving center of the axis-aligned ellipsoid is $\mathbf{C}(t) = (c_0, c_1, c_2) = \mathbf{K} + t\mathbf{W}$, just as in Equation (23). The function $h(t)$ and its derivative $h'(t)$ are

$$h(t) = \frac{d_0^2 c_0^2 s^2}{(d_0 s - 1)^2} + \frac{d_1^2 c_1^2 s^2}{(d_1 s - 1)^2} + \frac{d_2^2 c_2^2 s^2}{(d_2 s - 1)^2} - 1 \tag{40}$$

and

$$h'(t) = \frac{2 d_0 (c_0^2)' s}{d_0 s - 1} + \frac{2 d_1 (c_1^2)' s}{d_1 s - 1} + \frac{2 d_2 (c_2^2)' s}{d_2 s - 1} \tag{41}$$

It may be shown that $h''(t) > 0$, so $h(t)$ is a convex function. Newton's method may be used to compute robustly t_{contact}.

Once we have the contact point, Equations (33) through (35) may be use to transform back to the original coordinates.

9 Source Code for Swept Ellipsoids

Here is Wild Magic code for computing the contact time and contact point, if they exist, for two moving ellipsoids.

```
double ComputeClosestPoint (const Matrix3d& D, const Vector3d& K,
    Vector3d& closestPoint)
{
    double d0 = D[0][0], d1 = D[1][1], d2 = D[2][2];
    double k0 = K[0], k1 = K[1], k2 = K[2];
    double d0k0 = d0*k0;
    double d1k1 = d1*k1;
    double d2k2 = d2*k2;
    double d0k0k0 = d0k0*k0;
    double d1k1k1 = d1k1*k1;
    double d2k2k2 = d2k2*k2;

    if (d0k0k0 + d1k1k1 + d2k2k2 - 1.0 < 0.0)
    {
        // The ellipsoid contains the origin, so the ellipsoid and sphere are
        // overlapping.
        return FLT_MAX;
    }

    const int maxIterations = 128;
    const double epsilon = 1e-08;

    double s = 0.0;
    int i;
    for (i = 0; i < maxIterations; ++i)
    {
        // Compute f(s).
        double tmp0 = d0*s - 1.0;
        double tmp1 = d1*s - 1.0;
        double tmp2 = d2*s - 1.0;
        double tmp0sqr = tmp0*tmp0;
        double tmp1sqr = tmp1*tmp1;
        double tmp2sqr = tmp2*tmp2;
        double f = d0k0k0/tmp0sqr + d1k1k1/tmp1sqr + d2k2k2/tmp2sqr - 1.0;

        if (fabs(f) < epsilon)
        {
            // We have found a root.
            break;
        }

        // Compute f'(s).
        double tmp0cub = tmp0*tmp0sqr;
        double tmp1cub = tmp1*tmp1sqr;
        double tmp2cub = tmp2*tmp2sqr;
        double fder = -2.0*(d0*d0k0k0/tmp0cub + d1*d1k1k1/tmp1cub
            + d2*d2k2k2/tmp2cub);

        // Compute the next iterate sNext = s - f(s)/f'(s).
        s -= f/fder;
    }

    // assert:  i < maxIterations

    closestPoint[0] = d0k0*s/(d0*s - 1.0);
    closestPoint[1] = d1k1*s/(d1*s - 1.0);
    closestPoint[2] = d2k2*s/(d2*s - 1.0);
    return s;
}
```

```
bool ComputeContact (const Matrix3d& D, const Vector3d& K, const Vector3d& W,
    double& contactTime, Vector3d& Zcontact)
{
    double d0 = D[0][0], d1 = D[1][1], d2 = D[2][2];
    double k0 = K[0], k1 = K[1], k2 = K[2];
    double w0 = W[0], w1 = W[1], w2 = W[2];

    int maxIterations = 128;
    double epsilon = 1e-08;

    contactTime = 0.0;
    for (int i = 0; i < maxIterations; ++i)
    {
        // Compute h(t).
        Vector3d Kmove = K + contactTime*W;
        double s = ComputeClosestPoint(D, Kmove, Zcontact);
        double tmp0 = d0*Kmove[0]*s/(d0*s - 1.0);
        double tmp1 = d1*Kmove[1]*s/(d1*s - 1.0);
        double tmp2 = d2*Kmove[2]*s/(d2*s - 1.0);
        double h = tmp0*tmp0 + tmp1*tmp1 + tmp2*tmp2 - 1.0;

        if (fabs(h) < epsilon)
        {
            // We have found a root.
            return true;
        }

        // Compute h'(t).
        double hder = 4.0*tmp0*w0 + 4.0*tmp1*w1 + 4.0*tmp2*w2;
        if (hder > 0.0)
        {
            // The ellipsoid cannot intersect the sphere.
            return false;
        }

        // Compute the next iterate tNext = t - h(t)/h'(t).
        contactTime -= h/hder;
    }

    ComputeClosestPoint(D, K + contactTime*W, Zcontact);
    return true;
}
```

```
bool FindContact (const Ellipsoid3d &ellipsoid0, const Vector3d& velocity0,
    const Ellipsoid3d &ellipsoid1, const Vector3d& velocity1,
    double& contactTime, Vector3d& contactPoint)
{
    // Get the parameters of ellipsoid0.
    Vector3d K0 = ellipsoid0.Center;
    Matrix3d R0(ellipsoid0.Axis, true);
    Matrix3d D0(
        1.0/(ellipsoid0.Extent[0]*ellipsoid0.Extent[0]),
        1.0/(ellipsoid0.Extent[1]*ellipsoid0.Extent[1]),
        1.0/(ellipsoid0.Extent[2]*ellipsoid0.Extent[2]));

    // Get the parameters of ellipsoid1.
    Vector3d K1 = ellipsoid1.Center;
    Matrix3d R1(ellipsoid1.Axis, true);
    Matrix3d D1(
        1.0/(ellipsoid1.Extent[0]*ellipsoid1.Extent[0]),
        1.0/(ellipsoid1.Extent[1]*ellipsoid1.Extent[1]),
        1.0/(ellipsoid1.Extent[2]*ellipsoid1.Extent[2]));

    // Compute K2.
    Matrix3d D0NegHalf(ellipsoid0.Extent[0], ellipsoid0.Extent[1],
        ellipsoid0.Extent[2]);
    Matrix3d D0Half(1.0/ellipsoid0.Extent[0], 1.0/ellipsoid0.Extent[1],
        1.0/ellipsoid0.Extent[2]);
    Vector3d K2 = D0Half*((K1 - K0)*R0);

    // Compute M2.
    Matrix3d R1TR0D0NegHalf = R1.TransposeTimes(R0*D0NegHalf);
    Matrix3d M2 = R1TR0D0NegHalf.TransposeTimes(D1)*R1TR0D0NegHalf;

    // Factor M2 = R*D*R^T.
    Matrix3d R, D;
    M2.EigenDecomposition(R, D);

    // Compute K.
    Vector3d K = K2*R;

    // Compute W.
    Vector3d W = (D0Half*((velocity1 - velocity0)*R0))*R;

    // Transformed ellipsoid0 is Z^T*Z = 1 and transformed ellipsoid1 is
    // (Z-K)^T*D*(Z-K) = 0.

    // Compute the initial closest point.
    Vector3d P0;
    if (ComputeClosestPoint(D, K, P0) >= 0.0)
    {
        // The ellipsoid contains the origin, so the ellipsoids were not
        // separated.
        return false;
    }
    double dist0 = P0.Dot(P0) - 1.0;
    if (dist0 < 0.0)
    {
        // The ellipsoids are not separated.
        return false;
    }

    Vector3d Zcontact;
    if (!ComputeContact(D, K, W, contactTime, Zcontact))
    {
        return false;
    }

    // Transform contactPoint back to original space.
    contactPoint = K0 + R0*D0NegHalf*R*Zcontact;
    return true;
}
```