Intersection of Rectangle and Ellipse

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

Created: September 20, 2015

Contents

1 Introduction

This document describes how to determine whether a rectangle and an ellipse overlap. This is a test-intersection query (do they overlap) and not a find-intersection query (where do they overlap).

The rectangle is represented by a center $\mathbf{C}$, unit-length and orthogonal axes $\mathbf{U}_0$ and $\mathbf{U}_1$, and extents $e_0 > 0$ and $e_1 > 0$. The rectangle is considered to be solid with points $\mathbf{X} = \mathbf{C} + \xi_0\mathbf{U}_0 + \xi_1\mathbf{U}_1$, where $|\xi_0| \le e_0$ and $|\xi_1| \le e_1$. In terms of vector-matrix algebra, $\mathbf{X} = \mathbf{C} + U\boldsymbol{\xi}$, where U is the 2×2 rotation matrix whose columns are the rectangle axes and $\boldsymbol{\xi}$ is the 2×1 column vector whose rows are the coordinates ξ_0 and ξ_1. Define the four vertices of the rectangle to be

$$\mathbf{P}^{(i_0,i_1)} = \mathbf{C} + (2i_0 - 1)e_0\mathbf{U}_0 + (2i_1 - 1)e_1\mathbf{U}_1$$

for i_0 and i_1 in $\{0, 1\}$.

The ellipse is represented by a center $\mathbf{K}$, unit-length and orthogonal axes $\mathbf{V}_0$ and $\mathbf{V}_1$, and axis half-lengths $a_0 > 0$ and $a_1 > 0$. The ellipse is considered to be solid with points $\mathbf{Y} = \mathbf{K} + \eta_0\mathbf{V}_0 + \eta_1\mathbf{V}_1$, where $(\eta_0/a_0)^2 + (\eta_1/a_1)^2 = 1$. In terms of vector-matrix algebra, $\mathbf{Y} = \mathbf{K} + V\boldsymbol{\eta}$, where V is the 2×2 rotation matrix whose columns are the rectangle axes and $\boldsymbol{\eta}$ is the 2×1 column vector whose rows are the coordinates η_0 and η_1. The ellipse is represented implicitly by

$$0 = Q(\mathbf{Y}) = (\mathbf{Y} - \mathbf{K})^{\mathsf{T}} M (\mathbf{Y} - \mathbf{K}) - 1 = \left|DV^{\mathsf{T}}(\mathbf{Y} - \mathbf{K})\right|^2 - 1$$

where $M = VD^2V^{\mathsf{T}}$ with $D = \text{Diagonal}(1/a_0, 1/a_1)$. The solid ellipse (ellipse and region it contains) is $Q(\mathbf{Y}) \le 0$.

The data structures listed next are used in the pseudocode.

```
struct Rectangle { Point2 center; Vector2 axis[2]; Real extent[2]; };
struct Ellipse { Point2 center; Vector2 axis[2]; Real halfLength[2]; Matrix2x2 M; };
```

The matrix M can be computed from the axis[] and halfLength[] members.

2 All-Features Testing

This is a simple approach to test the rectangle features for containment in the ellipse. If any rectangle vertex is in the ellipse, then the rectangle and ellipse overlap. If no vertex is in the ellipse, it is possible that a rectangle edge intersects the ellipse. If no vertex is in the ellipse and no edge intersects the ellipse, it is possible that the ellipse is strictly inside the rectangle, which can be determined by testing whether the ellipse center is inside the rectangle. The following discussion assumes that the tests are performed in the order mentioned, which simplifies the intersection and overlap tests.

If $\mathbf{P}$ is one of the vertices of the rectangle, containment in the ellipse is characterized by $Q(\mathbf{P}) \le 0$.

If $\mathbf{P}_0$ and $\mathbf{P}_1$ are the endpoints of a rectangle edge, the edge is represented by $\mathbf{E}(t) = \mathbf{P}_0 + (\mathbf{P}_1 - \mathbf{P}_0)t$ for $t \in [0, 1]$. When the endpoints are outside the ellipse, the segment-ellipse test-intersection query is the same as the line-ellipse intersection. The line and ellipse intersect when the quadratic function

$$\begin{aligned} q(t) &= Q(\mathbf{E}(t)) \\ &= (\mathbf{P}_1 - \mathbf{P}_0)^{\mathsf{T}} M (\mathbf{P}_1 - \mathbf{P}_0) t^2 + 2 (\mathbf{P}_1 - \mathbf{P}_0)^{\mathsf{T}} M (\mathbf{P}_0 - \mathbf{K}) t + (\mathbf{P}_0 - \mathbf{K})^{\mathsf{T}} M (\mathbf{P}_0 - \mathbf{K}) - 1 \\ &= q_2 t^2 + 2q_1 t + q_0 \end{aligned}$$

has real-valued roots. The last equality defines the coefficients q_i. The quadratic has real-valued roots when its discriminant is nonnegative: $q_1^2 - q_0 q_2 \geq 0$. It is possible to skip the vertex containment tests and use instead a modification of the edge-ellipse tests. If $q(t)$ has two real-valued roots (possibly equal), say, $t_0 \leq t_1$, then the edge intersects the solid ellipse when $[0,1] \cap [t_0, t_1] \neq \emptyset$ (the intersection of t-intervals is nonempty).

If none of the rectangle edges overlap the solid ellipse, the test for the ellipse center contained in the rectangle requires converting the ellipse center to box coordinates: $\boldsymbol{\xi} = U^{\mathsf{T}}(\mathbf{K} - \mathbf{C})$. The test for containment is $|\xi_0| \leq e_0$ and $|\xi_1| \leq e_1$.

```
bool EllipseContainsVertex (Ellipse E, Vector2 PmK)
{
    return Dot(PmK, E.M*PmK) <= 1;
}

bool EllipseOverlapsEdge (Ellipse E, Vector2 P0mK, Vector3 P1mK)
{
    Vector2 D = P1mK - P0mK, MP0mK = E.M*P0mK;
    Real q0 = Dot(P0mK, MP0mK) - 1, q1 = Dot(D, MP0mK), q2 = Dot(D, E.M*D);
    if (discr >= 0)
    {
        // The line containing P0 and P1 intersects the ellipse.  Determine
        // whether the segment connecting P0 and P1 intersects the ellipse.
        Real rootDiscr = sqrt(discr);
        Real t0 = (-q1 - rootDiscr) / q2, t1 = (-q1 + rootDiscr) / q2;
        return Overlaps([0,1], [t0,t1]);
    }
    else
    {
        return false;
    }
}

bool RectangleContainsPoint (Rectangle R, Vector2 V)
{
    return fabs(Dot(R.axis[0], V)) <= R.extent[0]
        && fabs(Dot(R.axis[1], V)) <= R.extent[1];
}

bool TestIntersectionAllFeatures (Rectangle R, Ellipse E)
{
    // Translate so ellipse center is at origin.
    Vector2 CmK = R.center - E.center;

    Vector2 PmK[4];
    for (int i1 = 0, s1 = -1; i1 < 2; ++i1, s1 += 2)
    {
        for (int i0 = 0, s0 = -1; i0 < 2; ++i0, s0 += 2, ++j)
        {
            PmK[j] = CmK + s0*R.extent[i0]*R.axis[i0] + s1*R.extent[i1]*R.axis[i1];
            if (EllipseContainsVertex(E, PmK[j])) { return true; }
        }
    }

    int2 edge[4] = { (0,1), (1,3), (3,2), (2,0) };
    for (int j = 0; j < 4; ++j)
    {
        if (EllipseOverlapsEdge(E, PmK[edge[j][0]], PmK[edge[j][1]]) { return true; }
    }

    return RectangleContainsPoint(R, -CmK);
}
```

3 Closest-Features Testing

In worst case, the all-features approach requires 4 vertex containment tests, 4 edge overlap tests, and 1 point-in-rectangle test. This does not take into account closest-feature information, but a small modification does. First, test whether the ellipse center is in the rectangle. If it is, the objects intersect. If it is not, then you will know the 1 or 2 edges of the rectangle that are visible to the ellipse center. In worst case, this limits you to 1 point-in-rectangle test, 3 vertex containment tests, and 2 edge overlap tests.

```
bool TestOverlap1Edge (Rectangle R, Ellipse E, Vector2 CmK, int vertex[])
{
    Vector2 V[2];
    for (int j = 0; j < 2; ++j)
    {
        int s0 = 2*(vertex[j] & 1) - 1, s1 = 2*(vertex[j] & 2) - 1;
        V[j] = CmK + s0*R.extent[0]*R.axis[0] + s1*R.extent[1]*R.axis[1];
        if (EllipseContainsVertex(E, V[j])) { return true; }
    }
    return EllipseOverlapsEdge(E, V[0], V[1]);
}

bool TestOverlap2Edges (Rectangle R, Ellipse E, Vector2 CmK, int vertex[])
{
    Vector2 V[3];
    for (int j = 0; j < 3; ++j)
    {
        int s0 = 2*(vertex[j] & 1) - 1, s1 = 2*(vertex[j] & 2) - 1;
        V[j] = CmK + s0*R.extent[0]*R.axis[0] + s1*R.extent[1]*R.axis[1];
        if (EllipseContainsVertex(E, V[j])) { return true; }
    }
    for (int j0 = 2, j1 = 0; j1 < 3; j0 = j1++)
    {
        if (EllipseOverlapsEdge(E, V[j0], V[j1])) { return true; }
    }
    return false;
}

bool ContainsPoint (Rectangle, Ellipse, Vector2, int[])
{
    return true;
}

TestOverlapFunction TestOverlap[3] = { ContainsPoint, TestOverlap1Edge, TestOverlap2Edges };
struct Query { int function; int vertex[3]; };
Query query[3][3] =
{
    {
        {2, {2,0,1}},  // -e0 < x0 and -e1 < x1
        {1, {2,0  }},  // -e0 < x0 and x1 <= e1
        {2, {3,2,0}}   // -e0 < x0 and e1 < x1
    },
    {
        {1, {0,1  }},  // x0 <= e0 and -e1 < x1
        {0, {     }},  // x0 <= e0 and x1 <= e1
        {1, {3,2  }}   // x0 <= e0 and e1 < x1
    },
    {
        {2, {0,1,3}},  // e0 < x0 and -e1 < x1
        {1, {1,3  }},  // e0 < x0 and x1 <= e1
        {2, {1,3,2}}   // e0 < x0 and e1 < x1
    }
};

bool TestIntersectionClosestFeatures (Rectangle R, Ellipse E)
{
    // Transform the ellipse center to rectangle coordinate system.
    Vector2 KmC = E.center - R.center;
    Real xi[2] = { Dot(R.axis[0], KmC), Dot(R.axis[1], KmC) };
```

```
    int select[2] =
    {
        (-R.extent[0] < xi[0] ? 0 (xi[0] <= R.extent[0] ? 1 : 2)),
        (-R.extent[1] < xi[1] ? 0 (xi[1] <= R.extent[1] ? 1 : 2))
    };

    Query q = query[select[0]][select[1]];
    return TestOverlap[q.function](R, E, -KmC, q.vertex);
}
```

4 Conversion to Point-Parallelogram Distance Query

The objects can be transformed so that the ellipse becomes a circle and the rectangle becomes a parallelogram. The rectangle and ellipse overlap if and only if the parallelogram and circle overlap. The latter query reduces to comparing the distance from circle center to parallelogram with the circle radius.

To compute the distance efficiently, we need to determine the 1 or 2 edges of the parallelogram that are closest to the circle center. The analysis is similar to that in the pseudocode TestIntersectionSomeFeatures. In fact, the only difference between the two algorithms is that the current one involves distance, which uses the standard Cartesian metric. The previous algorithm is essentially the same algorithm but with the Euclidean metric imposed by the ellipse. It turns out that we may develop this algorithm without converting the ellipse to a circle.

As in the previous section, we may determine which of 9 cases apply to the ellipse center relative to the rectangle. In a case where the ellipse center is outside the rectangle, we want to find the closest edge point relative to the metric of the ellipse. But what does this mean? The function $Q(\mathbf{Y})$ defines a family of ellipses, each ellipse corresponding to a scalar c for which $Q(\mathbf{Y}) = c$. Generally, implicit curves of this type are referred to as *level curves* of the function. The original ellipse has level value $c = 0$. The ellipses nested inside the original have level values $c < 0$, where the ellipse center is a degenerate curve (a single point) when $c = -1$. Ellipses outside the original have level values $c > 0$. We want to find the point $\mathbf{P}$ on the visible rectangle edges that minimize Q, call this value $c_{\min}$. The point $\mathbf{P}$ is considered to be the point closest to the original ellipse relative to the imposed metric, which is matrix M. If $c_{\min} \le 0$, the rectangle and ellipse must overlap and $\mathbf{P}$ is in the overlap set. In the Cartesian case, the quadratic has the matrix $M = I$ (the identity). Minimizing distance is equivalent to minimizing the quadratic when $M = I$.

For an edge parameterized by $\mathbf{P}_0 + t(\mathbf{P}_1 - \mathbf{P}_0)$ with $t \in [0, 1]$, the quadratic is as defined previously: $q(t) = q_2 t^2 + 2q_1 t + q_0$. The minimum occurs either at an endpoint of the t-interval or at an interior point $\bar{t}$ of the interval where the derivative is zero, $q'(\bar{t}) = 0$. For the case when a single edge is the only candidate for closest feature, we need compare at most 3 values: $q(0) \le 0$, $q(1) \le 0$, and $q(\bar{t}) \le 0$, the latter one only when $\bar{t} \in (0, 1)$. For the case when two edges are candidates for closest feature, we need compare at most 5 values, 3 per edge but one of them shared.

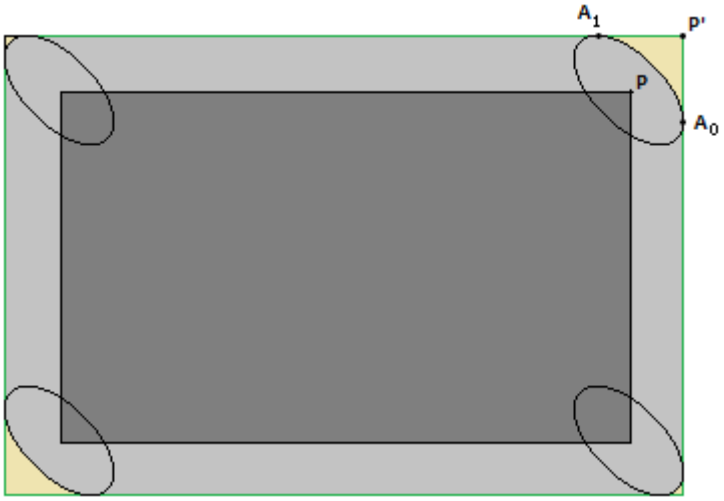
The derivative of $q(t)$ is $q'(t) = 2q_2 t + 2q_1$. Because M is positive definite, $q_2 > 0$. The derivative is zero at $\bar{t} = -q_1/q_2$. In order that $\bar{t} \in (0, 1)$, we need $0 < -q_1 < q_2$. When this is true, $q(\bar{t}) = (q_0 q_2 - q_1^2)/q_2$. The test $q(\bar{t}) \le 0$ is equivalent to the test $q_1^2 - q_0 q_2 \ge 0$, which is exactly the test used in the edge cases for TestOverlap2 and TestOverlap3. As it turns out, viewing the problem as point-parallelpiped distance does not gain us anything, because it is equivalent to the optimized TestIntersectionSomeFeatures.

5 Minkowski Sum of Rectangle and Ellipse

A different approach involves shrinking the ellipse to a point and growing the rectangle to a larger one with elliptically rounded corners. The latter object is the Minkowski sum of rectangle and ellipse. The rectangle and ellipse overlap when the ellipse center is in the Minkowski sum.

Translate the rectangle center to the origin. This is equivalent to processing the rectangle R with center $\mathbf{0}$ and the ellipse with center $\mathbf{K} - \mathbf{C}$. I will continue to refer to the ellipse center as $\mathbf{K}$. In this setting, the Minkowski sum has a bounding rectangle R' that is centered at the origin and has the same axes as the original rectangle. The extents are larger, of course. Figure 1 illustrates this.

Figure 1. The gray-shaded region is the Minkowski sum of the rectangle and ellipse. The ellipses at the original vertices are shown. The bounding box of the sum is also shown (in green). The yellow-shaded regions are the focus of the analysis here.



We need to compute the points on the corner ellipses that support R'. Let $\mathbf{P}$ be a corner of R. The extreme points in a specified unit-length direction $\mathbf{N}$ on the ellipse defined by $Q(\mathbf{X}) = (\mathbf{X} - \mathbf{K})^{\mathsf{T}} M (\mathbf{X} - \mathbf{K}) - 1 = 0$ are points on the ellipse with normals parallel to $\mathbf{N}$ (two of them). Normal vectors to the ellipse are provided by the gradient, $\nabla Q = 2M(\mathbf{X} - \mathbf{K})$. The gradient is parallel to the specified direction when $M(\mathbf{X} - \mathbf{K}) = s\mathbf{N}$ for some scalar s. Thus, $\mathbf{X} - \mathbf{K} = sM^{-1}\mathbf{N}$. Substituting this into the quadratic equation, we can solve for $s = 1/\sqrt{\mathbf{N}^{\mathsf{T}} M^{-1} \mathbf{N}}$. The two extreme points are $\mathbf{X} = \mathbf{K} \pm M^{-1}\mathbf{N}/\sqrt{\mathbf{N}^{\mathsf{T}} M^{-1} \mathbf{N}}$. The distance from $\mathbf{K}$ to the extreme points measured along the line of direction $\mathbf{N}$ is $\ell = \sqrt{\mathbf{N}^{\mathsf{T}} M^{-1} \mathbf{N}}$.

In our application, the directions of interest are $\mathbf{U}_0$ and $\mathbf{U}_1$. The increases in the extents of the original rectangle are $\ell_i = \sqrt{\mathbf{U}_i^{\mathsf{T}} M^{-1} \mathbf{U}_i}$ for $i = 0, 1$; that is, the extents of R' are $e_i + \ell_i$ for $i = 0, 1$.

If $\mathbf{K}$ is outside R', the rectangle and ellipse do not overlap. If $\mathbf{K}$ is inside R', we need to determine whether $\mathbf{K}$ is outside the ellipses at the corners of R. If so, the rectangle and ellipse do not overlap. Sign testing will allows us to test exactly one corner for $\mathbf{K}$.

Figure 1 shows two extreme points at the upper-right corner of R. The corner is labeled $\mathbf{P}$ and the exteme points are labeled $\mathbf{A}_0$ and $\mathbf{A}_1$. Knowing that $\mathbf{K}$ is inside R', the region outside the ellipse at that corner is characterized as follows. Define $\mathbf{\Delta} = \mathbf{K} - \mathbf{P}$. Compute the representation of $\mathbf{\Delta}$ in terms of the vectors formed by the extreme points and the ellipse center, $\mathbf{\Delta} = z_0(\mathbf{A}_0 - \mathbf{P}) + z_1(\mathbf{A}_1 - \mathbf{P})$. If $z_0 > 0$ and $z_1 > 0$, then $\mathbf{K}$ is in the quadrilateral formed by $\mathbf{P}$, $\mathbf{A}_0$, $\mathbf{A}_1$, and the corner $\mathbf{P}'$ of R'. To be outside the ellipse, we additionally need $\mathbf{\Delta}^{\mathsf{T}} M \mathbf{\Delta} > 1$. The extreme points are $\mathbf{A}_i = \mathbf{P} + M^{-1}\mathbf{U}_i/\ell_i$. This leads to

$$M(\mathbf{K} - \mathbf{P}) = z_0 M(\mathbf{A}_0 - \mathbf{P}) + z_1 M(\mathbf{A}_1 - \mathbf{P}) = (z_0/\ell_0)\mathbf{U}_0 + (z_1/\ell_1)\mathbf{U}_1$$

which implies $z_0 = \ell_0 \mathbf{U}_0^{\mathsf{T}} M \mathbf{\Delta}$ and $z_1 = \ell_1 \mathbf{U}_1^{\mathsf{T}} M \mathbf{\Delta}$. Pseudocode for the algorithm is listed next. In practice, the Ellipse data structure stores an orientation matrix V rather than M. The computation of ℓ_i becomes

$$\ell_i^2 = \mathbf{U}_i^{\mathsf{T}} M^{-1} \mathbf{U}_i = \mathbf{U}_i^{\mathsf{T}} V D^{-2} V^{\mathsf{T}} \mathbf{U}_i = (a_0 \mathbf{U}_i \cdot \mathbf{V}_0)^2 + (a_1 \mathbf{U}_i \cdot \mathbf{V}_1)^2$$

so you do not need to apply a general inversion algorithm to M.

```
bool TestIntersectionMinkowski (Rectangle R, Ellipse E)
{
    // Compute the increase in extents for R'.
    Real L[2];
    for (int i = 0; i < 2; ++i)
    {
        L[i] = sqrt(Dot(R.axis[i], Inverse(E.M)*R.axis[i]));
    }

    // Transform the ellipse center to rectangle coordinate system.
    Vector2 KmC = E.center - R.center;
    Real xi[2] = { Dot(R.axis[0], KmC), Dot(R.axis[1], KmC) };

    if (fabs(xi[0]) <= R.extent[0] + L[0] && fabs(xi[1]) <= R.extent[1] + L[1])
    {
        Real s[2] = { (xi[0] >= 0 ? 1 : -1), (xi[1] >= 0 ? 1 : -1) };
        Vector2 PmC = s[0]*R.extent[0]*R.axis[0] + s[1]*R.extent[0]*R.axis[1];
        Vector2 MDelta = E.M*(KmC - PmC);
        for (int i = 0; i < 2; ++i)
        {
            if (s[i]*Dot(R.axis[i], MDelta) <= 0)
            {
                return true;
            }
        }
        return Dot(delta, E.M*delta) <= 1;
    }
    return false;
}
```