Distance from a Point to an Ellipse, an Ellipsoid, or a Hyperellipsoid

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

Created: June 18, 2011

Contents

1 Introduction

This document is a merged version of two older documents on computing the distance from a point to an ellipse in 2D and the distance from a point to an ellipsoid in 3D. The algorithms are based on a common theme that the older point-ellipse document explained but the older point-ellipsoid document did not. The common theme is presented in more detail. Moreover, the theme applies to a hyperellipsoid in n dimensions.

2 Distance from a Point to an Ellipse

A general ellipse in 2D is represented by a center point $\mathbf{C}$, an orthonormal set of axis-direction vectors $\{\mathbf{U}_0, \mathbf{U}_1\}$, and associated extents e_i with $e_0 \geq e_1 > 0$. The ellipse points are

$$\mathbf{P} = \mathbf{C} + x_0\mathbf{U}_0 + x_1\mathbf{U}_1 \tag{1}$$

where

$$\left(\frac{x_0}{e_0}\right)^2 + \left(\frac{x_1}{e_1}\right)^2 = 1 \tag{2}$$

If $e_0 = e_1$, then the ellipse is a circle with center $\mathbf{C}$ and radius e_0. The orthonormality of the axis directions and Equation (1) imply $x_i = \mathbf{U}_i \cdot (\mathbf{P} - \mathbf{C})$. Substituting this into Equation (2), we obtain

$$(\mathbf{P} - \mathbf{C})^{\mathrm{T}} M (\mathbf{P} - \mathbf{C}) = 1 \tag{3}$$

where $M = RDR^{\mathrm{T}}$ with R an orthogonal matrix whose columns are $\mathbf{U}_0$ and $\mathbf{U}_1$ and with D a diagonal matrix whose diagonal entries are $1/e_0^2$ and $1/e_1^2$.

The problem is to compute the distance from a point $\mathbf{Q}$ to the ellipse. It is sufficient to solve this problem in the coordinate system of the ellipse; that is, represent $\mathbf{Q} = \mathbf{C} + y_0\mathbf{U}_0 + y_1\mathbf{U}_1$. The distance from $\mathbf{Q}$ to the closest point $\mathbf{P}$ on the ellipse as defined by Equation (3) is the same as the distance from $\mathbf{Y} = (y_0, y_1)$ to the closest point $\mathbf{X} = (x_0, x_1)$ on the standard ellipse of Equation (2).

We may additionally use symmetry to simplify the construction. It is sufficient to consider the case when (y_0, y_1) is in the first quadrant: $y_0 \geq 0$ and $y_1 \geq 0$. For example, if (y_0, y_1) is in the second quadrant, where $y_0 < 0$, and if (x_0, x_1) is the closest point (which must be in the second quadrant), then $(-y_0, y_1)$ is in the first quadrant and $(-x_0, x_1)$ is the closest ellipse point (which must be in the first quadrant). If we reflect the query point into the first quadrant, we can construct the closest point in the first quadrant, and then unreflect the result back to the original quadrant.

2.1 The Closest Point's Normal is Directed Toward the Query Point

A parameterization of the standard ellipse is $\mathbf{X}(\theta) = (e_0 \cos\theta, e_1 \sin\theta)$ for $\theta \in [0, 2\pi)$. The squared distance from $\mathbf{Y}$ to any point on the ellipse is

$$F(\theta) = |\mathbf{X}(\theta) - \mathbf{Y}|^2 \tag{4}$$

This is a nonnegative, periodic, and differentiable function; it must have a global minimum occurring at an angle for which the first-order derivative is zero,

$$F'(\theta) = 2(\mathbf{X}(\theta) - \mathbf{Y}) \cdot \mathbf{X}'(\theta) = 0 \tag{5}$$

For the derivative to be zero, the vectors $(\mathbf{X}(\theta) - \mathbf{Y})$ and $\mathbf{X}'(\theta)$ must be perpendicular. The vector $\mathbf{X}'(\theta)$ is tangent to the ellipse at $\mathbf{X}(\theta)$. This implies the vector from $\mathbf{Y}$ to the closest ellipse point $\mathbf{X}$ must be normal to the curve at $\mathbf{X}$. Using the implicit form of the ellipse, namely, $G(x_0, x_1) = (x_0/e_0)^2 + (x_1/e_1)^2 - 1 = 0$, the gradient of $G(x_0, x_1)$ is a normal vector to the ellipse at (x_0, x_1). Half the gradient is also normal, so we have $(y_0, y_1) - (x_0, x_1) = t\nabla G(x_0, x_1)/2 = t(x_0/e_0^2, x_1/e_1^2)$ for some scalar t, or

$$y_0 = x_0\left(1 + \frac{t}{e_0^2}\right), \quad y_1 = x_1\left(1 + \frac{t}{e_1^2}\right) \tag{6}$$

If (y_0, y_1) is outside the ellipse, it is necessary that $t > 0$. If (y_0, y_1) is inside the ellipse, it is necessary that $t < 0$. If (y_0, y_1) is already on the ellipse, then $t = 0$ and the distance is zero.

2.2 The Case of a Circle

If $e_0 = e_1$, then the ellipse is a circle. The origin $(0, 0)$ has infinitely many closest circle points (all of them), but clearly the distance from the origin to the circle is e_0. The closest circle point to a point $(y_0, y_1) \neq (0, 0)$ is $(x_0, x_1) = e_0(y_0, y_1)/|(y_0, y_1)|$. Equation (6) is consistent with this, because it implies $(x_0, x_1) = (e_0^2/(t + e_0^2))(y_0, y_1)$ for some t; that is, (x_0, x_1) is parallel to (y_0, y_1) and must have length e_0. It is easily shown that $t = -e_0^2 + e_0\sqrt{y_0^2 + y_1^2}$.

For the remainder of Section 2, we assume $e_0 > e_1$ and that all analysis is restricted to the first quadrant.

2.3 The Query Point is the Origin

Let $y_0 = 0$ and $y_1 = 0$. Equation (6) becomes $0 = x_0(1 + t/e_0^2)$ and $0 = x_1(1 + t/e_1^2)$. We have four cases to consider.

- If $x_0 = 0$ and $x_1 = 0$, the point is not on the ellipse; this case may be discarded.
- If $x_0 = 0$ and $x_1 \neq 0$, then $t = -e_1^2$ and the only constraint on x_1 is that (x_0, x_1) be a point on the ellipse, which means $x_1 = e_1$. The candidate closest point is $(0, e_1)$.
- If $x_0 \neq 0$ and $x_1 = 0$, then $t = -e_0^2$ and the only constraint on x_0 is that (x_0, x_1) be a point on the ellipse, which means $x_0 = e_0$. The candidate closest point is $(e_0, 0)$.
- If $x_0 \neq 0$ and $x_1 \neq 0$, then $t = -e_0^2$ and $t = -e_1^2$, which is a contradiction because $e_0 \neq e_1$; this case may be discarded.

The only candidate ellipse points in the first quadrant closest to $(0, 0)$ are $(e_0, 0)$ and $(0, e_1)$. Of these two, $(0, e_1)$ is closer. In summary: *The closest ellipse point in the first quadrant to* $(0, 0)$ *is the point* $(0, e_1)$ *with distance* $d = e_1$.

2.4 The Query Point is on the Vertical Axis

Let $y_0 = 0$ and $y_1 > 0$. Equation (6) becomes $0 = x_0(1 + t/e_0^2)$ and $y_1 = x_1(1 + t/e_1^2)$. We have two cases to consider.

- If $x_0 = 0$, then for (x_0, x_1) to be on the ellipse, we need $x_1 = e_1$. The candidate closest point is $(0, e_1)$.
- If $x_0 \neq 0$, then $t = -e_0^2$ and $y_1 = x_1(1 - e_0^2/e_1^2) \leq 0$, where the last inequality is a consequence of $x_1 \geq 0$ and $e_0 > e_1$. This contradicts the assumption that $y_1 > 0$.

The only candidate ellipse point is $(0, e_1)$. In summary: *The closest in the first quadrant to $(0, y_1)$ for $y_1 > 0$ is $(0, e_1)$ with distance $d = |y_1 - e_1|$.*

2.5 The Query Point is on the Horizontal Axis

Let $y_0 > 0$ and $y_1 = 0$. Equation (6) becomes $y_0 = x_0(1 + t/e_0^2)$ and $0 = x_1(1 + t/e_1^2)$. We have two subcases to consider.

- If $x_1 = 0$, then for (x_0, x_1) to be on the ellipse, we need $x_0 = e_0$. The candidate closest point is $(e_0, 0)$. The squared distance to this candidate is

$$d_0^2 = (y_0 - e_0)^2 \tag{7}$$

- If $x_1 \neq 0$, then $t = -e_1^2$ and $y_0 = x_0(1 - e_1^2/e_0^2)$. It follows that $x_0 = e_0^2 y_0/(e_0^2 - e_1^2) \leq e_0$. The last inequality is a consequence of (x_0, x_1) being on the ellipse. The implication is that $y_0 \leq (e_0^2 - e_1^2)/e_0 < e_0$. Notice that this limits the construction to points $(y_0, 0)$ inside the ellipse. For points $(y_0, 0)$ with $y_0 > (e_0^2 - e_1^2)/e_0$, which includes points inside and outside the ellipse, the closest point is necessarily $(e_0, 0)$. When $y_0 \leq (e_0^2 - e_1^2)/e_0$, for (x_0, x_1) to be on the ellipse, we may solve for $x_1 = e_1\sqrt{1 - (x_0/e_0)^2}$. The squared distance from the candidate (x_0, x_1) to $(y_0, 0)$ is

$$d_1^2 = (x_0 - y_0)^2 + x_1^2 = e_1^2 \left(1 - \frac{y_0^2}{e_0^2 - e_1^2}\right) \tag{8}$$

The two candidates are $(e_0, 0)$ and (x_0, x_1). We need to determine which of these is closer to $(y_0, 0)$. It may be shown that

$$d_0^2 - d_1^2 = \frac{(e_0 y_0 + e_1^2 - e_0^2)^2}{e_0^2 - e_1^2} \geq 0 \tag{9}$$

This implies that (x_0, x_1) is the closer point. In summary: *The closest point in the first quadrant to $(y_0, 0)$ for $0 < y_0 < e_0 - e_1^2/e_0$ is (x_0, x_1) with $x_0 = e_0^2 y_0/(e_0^2 - e_1^2)$, $x_1 = e_1\sqrt{1 - (x_0/e_0)^2}$, and distance $d = \sqrt{(x_0 - y_0)^2 + x_1^2} = e_1\sqrt{1 - y_0^2/(e_0^2 - e_1^2)}$. The closest point to $(y_0, 0)$ for $y_0 \geq e_0 - e_1^2/e_0$ is $(e_0, 0)$ with distance $d = |y_0 - e_0|$.*

2.6 The Query Point is Strictly in the First Quadrant

Let $y_0 > 0$ and $y_1 > 0$. Equation (6) is solved for

$$x_0 = \frac{e_0^2 y_0}{t + e_0^2}, \quad x_1 = \frac{e_1^2 y_1}{t + e_1^2} \tag{10}$$

for some scalar t. We know that the closest point in the first quadrant requires $x_0 \geq 0$ and $x_1 \geq 0$, which implies $t > -e_0^2$ and $t > -e_1^2$. Because $e_0 > e_1$, it is enough to analyze only those t-values for which $t > -e_1^2$.

Substitute Equation (10) in Equation (2) to obtain

$$F(t) = \left(\frac{e_0 y_0}{t + e_0^2}\right)^2 + \left(\frac{e_1 y_1}{t + e_1^2}\right)^2 - 1 = 0 \tag{11}$$

The first equality defines the function $F(t)$ with domain $(-e_1^2, \infty)$. The candidates for the ellipse point (x_0, x_1) closest to (y_0, y_1) are generated by the roots t to $F(t) = 0$.

The first-order and second-order derivatives of F are

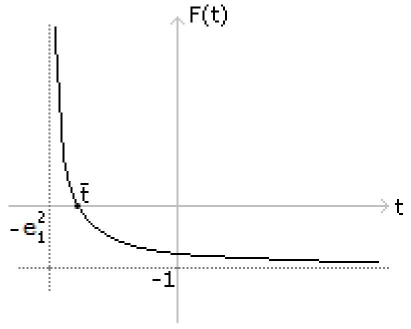
$$F'(t) = \frac{-2e_0^2 y_0^2}{(t + e_0^2)^3} + \frac{-2e_1^2 y_1^2}{(t + e_1^2)^3}, \quad F''(t) = \frac{6e_0^2 y_0^2}{(t + e_0^2)^4} + \frac{6e_1^2 y_1^2}{(t + e_1^2)^4} \tag{12}$$

We know that $y_0 > 0$, $y_1 > 0$, and $t > -e_1^2$. These conditions imply $F'(t) < 0$ and $F''(t) > 0$ for $t > -e_1^2$. Observe that

$$\lim_{t \to -e_1^{2+}} F(t) = +\infty, \quad \lim_{t \to \infty} F(t) = -1 \tag{13}$$

The first expression is a one-sided limit where t approaches e_1^2 through values larger than e_1^2. We have shown that $F(t)$ is a strictly decreasing function for $t \in (-e_1^2, +\infty)$ that is initially positive, then becomes negative. Consequently it has a unique root on the specified domain. Figure 2.1 shows a typical graph of $F(t)$.

Figure 2.1 A typical graph of $F(t)$ for $t > -e_1^2$. The unique root $\bar{t}$ is shown.



Observe that the domain $(-e_1^2, \infty)$ contains 0 and that $F(0) = (y_0/e_0)^2 + (y_1/e_1)^2 - 1$. When (y_0, y_1) is inside the ellipse, $F(0) < 0$, in which case $\bar{t} < 0$. When (y_0, y_1) is outside the ellipse, $F(0) > 0$, which does not explicitly constrain the sign of $\bar{t}$, but we do know geometrically that $\bar{t} > 0$. When (y_0, y_1) is on the ellipse, $F(0) = 0$ and so $\bar{t} = 0$.

2.7 A Summary of the Mathematical Algorithm

Pseudocode is listed next that summarizes the algorithm for computing the ellipse point (x_0, x_1) closest to the query point (y_0, y_1) in the first quadrant. The function also computes the distance.

```
Real Distance (Real e0, Real e1, Real y0, Real y1, Real& x0, Real& x1)
{
    Real distance;
    if (y1 > 0)
    {
        if (y0 > 0)
        {
            Compute the unique root tbar of F(t) on (-e1*e1,+infinity);
            x0 = e0*e0*y0/(tbar + e0*e0);
            x1 = e1*e1*y1/(tbar + e1*e1);
            distance = sqrt((x0 - y0)*(x0 - y0) + (x1 - y1)*(x1 - y1));
        }
        else  // y0 == 0
        {
            x0 = 0;
            x1 = e1;
            distance = abs(y1 - e1);
        }
    }
    else  // y1 == 0
    {
        if (y0 < e0 - e1*e1/e0)
        {
            x0 = e0*e0*y0/(e0*e0 - e1*e1);
            x1 = e1*sqrt(1 - (x0/e0)*(x0/e0));
            distance = sqrt((x0 - y0)*(x0 - y0) + x1*x1);
        }
        else
        {
            x0 = e0;
            x1 = 0;
            distance = abs(y0 - e0);
        }
    }
    return distance;
}
```

2.8 Robust Root Finders

The critical part of the algorithm is computing the unique root of $F(t)$ on $(-e_1^2, \infty)$. The simplest way to compute the root robustly is to use bisection on an interval that bounds the root. The bisection is applied until the endpoints of the bounding interval are consecutive floating-point numbers. For 32-bit `float`, the maximum number of bisections is 128. For 64-bit `double`, the maximum number of bisections is 1024. In practice for this example, the actual number of bisections is typically smaller than the maximum.

It is also possible to use Newton's Method. The theoretical problem is naturally suited for this root finder, but the use of floating-point arithmetic requires you to be careful in the implementation. You should probably use a hybrid between Newton's Method and bisection.

2.8.1 Bisection Method

You may locate the unique root $\bar{t}$ of $F(t)$ in the interval $(-e_1^2, \infty)$ using bisection. A bounding interval is needed. Observe that if

$$t_0 = e_1 y_1 - e_1^2 \tag{14}$$

then

$$F(t_0) = \left(\frac{e_0 y_0}{t_0 + e_0^2} \right)^2 > 0 \tag{15}$$

Also observe that $t + e_0^2 > t + e_1^2$, in which case

$$F(t) = \left(\frac{e_0 y_0}{t + e_0^2}\right)^2 + \left(\frac{e_1 y_1}{t + e_1^2}\right)^2 - 1 < \frac{e_0^2 y_0^2 + e_1^2 y_1^2}{(t + e_1^2)^2} - 1 \tag{16}$$

The right-hand side of this equation is negative for $t > t_1$ where

$$t_1 = -e_1^2 + \sqrt{e_0^2 y_0^2 + e_1^2 y_1^2} \tag{17}$$

Thus, a bounding domain for the root is $[t_0, t_1]$, where t_0 is defined by Equation (14) and t_1 is defined by Equation (17).

You may bisect until you reach your own stopping criterion, say, $|t_1 - t_0| < \varepsilon$ for a specified small number $\varepsilon > 0$. The source code shown later bisects until t_0 and t_1 are consecutive floating-point numbers, effectively the best that you can do for the specified precision you are using for your floating-point numbers.

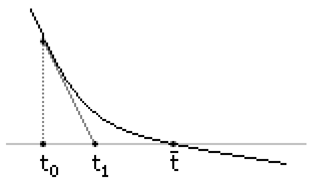
2.8.2 Newton's Method

The condition $F''(t) > 0$ says that F is a *convex function*. Such functions are ideally suited for the application of Newton's Method for root finding. Given an initial guess t_0, the Newton iterates are

$$t_{n+1} = t_n - \frac{F(t_n)}{F'(t_n)}, \quad n \geq 0$$

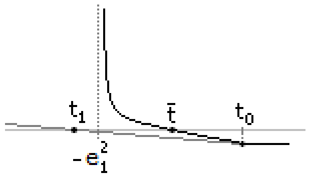
It is important to choose an initial guess for which the method converges. Newton's method has an intuitive geometric appeal to it. The next value t_{n+1} is computed by determining where the tangent line to the graph at $(t_n, F(t_n))$ intersects the t-axis. The intersection point is $(t_{n+1}, 0)$. If we choose an initial guess $t_0 < \bar{t}$, the tangent line to $(t_0, F(t_0))$ intersects the t-axis at $(t_1, 0)$ where $t_0 < t_1 < \bar{t}$. Figure 2.2 illustrates this.

Figure 2.2 An initial guess t_0 to the left of $\bar{t}$ guarantees $t_0 < t_1 < \bar{t}$.



If we were instead to choose an initial guess $t_0 > \bar{t}$, the new iterate satisfies $t_1 < \bar{t}$, but potentially $t_1 < -e_1^2$ which puts it outside the domain of interest, namely $t \in (-e_1^2, \infty)$. Figure 2.3 illustrates this.

Figure 2.3 An initial guess t_0 to the right of $\bar{t}$ guarantees $t_1 < \bar{t}$ but does not guarantee $t_1 > -e_1^2$.

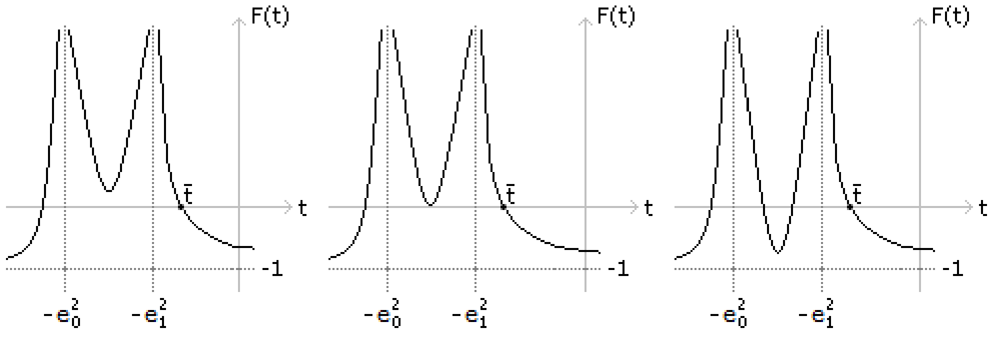


To avoid this potential problem, it is better to choose an initial guess to the left of the root. Any $t_0 > -e_1^2$ for which $F(t_0) > 0$ will work. In particular, you can choose $t_0 = -e_1^2 + e_1 y_1$, in which case $F(t_0) = [e_0 y_0/(e_1 y_1 + e_0^2 - e_1^2)]^2 > 0$.

The stopping criterion for Newton's Method typically involves testing the values $F(t_n)$ for closeness to zero. For robustness, testing is recommended to determine whether progress is made in the domain; that is, if the difference $|t_{n+1} - t_n|$ of consecutive iterates is sufficiently small, then the iteration terminates.

As mentioned previously, you need to be careful when using Newton's Method. At first it is mathematically comforting to have a convex function $F(t)$ for $t \in (-e_1^2, \infty)$, knowing that there is a unique root. However, there are potential problems when y_1 is nearly zero. When $y_0 > 0$ and $y_1 > 0$, Figure 2.4 shows typical graphs of $F(t)$.

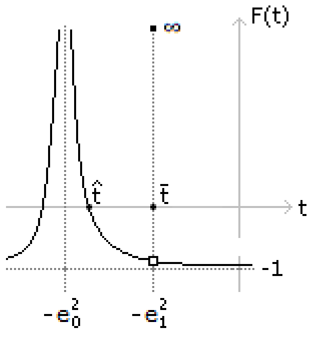
Figure 2.4 Typical graphs of $F(t)$ when $y_0 > 0$ and $y_1 > 0$.



The three possibilities are based on where (y_0, y_1) is located with respect to the *evolute* of the ellipse. Define $G(y_0, y_1) = (e_0 y_0)^{2/3} + (e_1 y_1)^{2/3} - (e_0^2 - e_1^2)^{2/3}$. The evolute is the level set defined by $G(y_0, y_1) = 0$. The left image of Figure 2.4 corresponds to (y_0, y_1) outside the evolute, so $G(y_0, y_1) > 0$. The middle image corresponds to (y_0, y_1) on the evolute itself, so $G(y_0, y_1) = 0$. The right image corresponds to (y_0, y_1) inside the evolute, so $G(y_0, y_1) < 0$. You can generate the evolute equation by eliminating t from the simultaneous equations $F(t) = 0$ and $F'(t) = 0$.

Imagine choosing positive values of y_1 closer and closer to zero. The vertical asymptotes at $t = -e_0^2$ and $t = -e_1^2$ remain so, but the shape of the graph of $F(t)$ near the asymptote $t = -e_1^2$ changes: it begins to hug the vertical asymptote. In the limiting sense as y_1 approaches zero, the graph is illustrated by Figure 2.5.

Figure 2.5 The limiting graph of $F(t)$ when $y_0 > 0$ and $y_1 \to 0^+$. Notice that there is effectively a discontinuity at $t = -e_1^2$, and the largest root jumps from $\bar{t}$ to $\hat{t}$.



Just setting $y_1 = 0$, the largest root of $F(t)$ is $\hat{t} \in (-e_0^2, -e_1^2)$. As a function of y_1, the largest root of $F(t)$ is discontinuous in the sense that $\lim_{y_1 \to 0^+} \bar{t}(y_1) \neq \hat{t} = -e_0^2 + e_0 y_0$.

Thus, the graph of $F(t)$ has a topological change at $y_1 = 0$, which can cause numerical problems with Newton's Method. When $y_1 > 0$, notice that $F'(t_0) = -2e_0^2 y_0^2/(e_1 y_1 + e_0^2 - e_1^2)^2 - 2/(e_1 y_1)$, which is a negative number of large magnitude when y_1 is nearly zero. You should guard against numerical problems when computing an iterate, accepting the iterate if it is in the current root-bounding interval and rejecting it otherwise. When rejecting the iterate, apply a bisection step on the root-bounding interval. This hybrid method is more robust than just computing iterates alone. Of course, the slower bisection method (or root finder such as Brent's Method) is robust.

2.8.3 Conversion to a Polynomial Equation

An approach for ellipsoids, but valid for ellipses as well, is described in [1]. The roots of the following quartic polynomial may be computed, leading to candidates for the closest ellipse point:

$$P(t) = (t + e_0^2)^2 (t + e_1^2)^4 F(t) = e_0^2 y_0^2 (t + e_1^2)^2 + e_1^2 y_1^2 (t + e_0^2)^2 - (t + e_0^2)^2 (t + e_1^2) \tag{18}$$

When $y_1 > 0$, the roots of $P(t)$ are the same as those for $F(t)$, and the largest root of $P(t)$ is the same as the largest root for $F(t)$. However, when $y_1 = 0$, $P(t)$ has a double root at $t = -e_1^2$ but $F(t)$ does not have a root at $t = -e_1^2$.

The Graphics Gems IV article mentions that the largest root of $P(t)$ is the one that leads to the closest point. This is not true when $y_1 = 0$. As mentioned in the previous paragraph. In this case, $P(t) = (t+e_1^2)^2[e_0^2 y_0^2 - (t+e_0^2)^2]$. The largest root is $\bar{t} = -e_1^2$ and another root is $\hat{t} = -e_0^2 + e_0 y_0$, as shown in Figure 2.5. It is $\hat{t}$ that determines the closest ellipse point, not $\bar{t}$.

The article also mentions that Newton's Method may be used, starting with an initial guess greater than the largest root. When (y_0, y_1) is inside the ellipse, then $\bar{t} < 0$ and so the initial guess is chosen to be $t_0 = 0$. When (y_0, y_1) is outside the ellipse, a geometric and algebraic argument is used to construct an initial guess

$t_0 = e_0\sqrt{y_0^2 + y_1^2}$. No analysis is provided for the graph of $P(t)$ at the largest root. In particular, there are no guarantees that $P(t)$ is convex at that root, so the convergence of the Newton iterates is not guaranteed. Even so, when $y_1 = 0$, the iterates might converge to the root $\bar{t} = -e_1^2$, which is not the correct root that determines the closest ellipse point.

Empirical evidence suggests that Newton's Method applied to $P(t)$ to find its largest root suffers from numerical problems when y_0 or y_1 is nearly zero. For example, if y_0 is positive and if y_1 is a very small positive number, then $\bar{t}$ is the largest root of $P(t)$. It is *nearly* a double root, which can cause problems with Newton's Method. Specifically, if a function $G(t)$ has a double root at $\bar{t}$, Newton's Method must be slightly modified to $t_{i+1} = t_i - 2G(t_i)/G'(t_i)$. Generally, if $G(t)$ has a root of multiplicity n, then Newton's Method should be $t_{i+1} = t_i - nG(t_i)/G'(t_i)$.

And finally, the initial guess suggested in the Graphics Gems IV article is also quite large. From the discussion of the bisection method, $t_1 = -e_1 + \sqrt{e_0^2 y_0^2 + e_1^2 y_1^2}$ is an estimate greater than the maximum root but smaller than $t_0 = e_0\sqrt{y_0^2 + y_1^2}$. To see this,

$$(t_1 + e_1)^2 = e_0^2 y_0^2 + e_1^2 y_1^2 = t_0^2 - (e_0^2 - e_1^2)y_1^2 < t_0^2 \tag{19}$$

which implies $t_1 < t_1 + e_1 < t_0$.

2.9 An Implementation

An implementation that uses the robust bisection method, where the iterates are computed until the bounding interval endpoints are consecutive floating-point numbers is listed next. The full source code is found in

```
WildMagic5/SampleMathematics/DistancePointEllipseEllipsoid
```

The function `DistancePointEllipseSpecial` requires that $e_0 \geq e_1$, $y_0 \geq 0$, and $y_1 \geq 0$. The function `DistancePointEllipse` is the general function to call. It reflects components of (y_0, y_1) as needed to place the query point in the first quadrant, sorts the (e_0, e_1) values to be correctly ordered, and then calls `DistancePointEllipseSpecial`. The returned (x_0, x_1) is postprocessed to match the quadrant of the query point.

```
//----------------------------------------------------------------------------
// The ellipse is (x0/e0)^2 + (x1/e1)^2 = 1 with e0 >= e1.  The query point is
// (y0,y1) with y0 >= 0 and y1 >= 0.  The function returns the distance from
// the query point to the ellipse.  It also computes the ellipse point (x0,x1)
// in the first quadrant that is closest to (y0,y1).
//----------------------------------------------------------------------------
template <typename Real>
Real DistancePointEllipseSpecial (const Real e[2], const Real y[2], Real x[2])
{
    Real distance;
    if (y[1] > (Real)0)
    {
        if (y[0] > (Real)0)
        {
            // Bisect to compute the root of F(t) for t >= -e1*e1.
            Real esqr[2] = { e[0]*e[0], e[1]*e[1] };
            Real ey[2] = { e[0]*y[0], e[1]*y[1] };
            Real t0 = -esqr[1] + ey[1];
            Real t1 = -esqr[1] + sqrt(ey[0]*ey[0] + ey[1]*ey[1]);
            Real t = t0;
            const int imax = 2*std::numeric_limits<Real>::max_exponent;
            for (int i = 0; i < imax; ++i)
            {
                t = ((Real)0.5)*(t0 + t1);
                if (t == t0 || t == t1)
                {
                    break;
                }

                Real r[2] = { ey[0]/(t + esqr[0]), ey[1]/(t + esqr[1]) };
                Real f = r[0]*r[0] + r[1]*r[1] - (Real)1;
                if (f > (Real)0)
                {
                    t0 = t;
                }
                else if (f < (Real)0)
                {
                    t1 = t;
                }
                else
                {
                    break;
                }
            }

            x[0] = esqr[0]*y[0]/(t + esqr[0]);
            x[1] = esqr[1]*y[1]/(t + esqr[1]);
            Real d[2] = { x[0] - y[0], x[1] - y[1] };
            distance = sqrt(d[0]*d[0] + d[1]*d[1]);
        }
        else  // y0 == 0
        {
            x[0] = (Real)0;
            x[1] = e[1];
            distance = fabs(y[1] - e[1]);
        }
    }
    else  // y1 == 0
    {
        Real denom0 = e[0]*e[0] - e[1]*e[1];
        Real e0y0 = e[0]*y[0];
        if (e0y0 < denom0)
        {
            // y0 is inside the subinterval.
            Real x0de0 = e0y0/denom0;
            Real x0de0sqr = x0de0*x0de0;
            x[0] = e[0]*x0de0;
            x[1] = e[1]*sqrt(fabs((Real)1 - x0de0sqr));
            Real d0 = x[0] - y[0];
            distance = sqrt(d0*d0 + x[1]*x[1]);
        }
        else
        {
            // y0 is outside the subinterval.  The closest ellipse point has
            // x1 == 0 and is on the domain-boundary interval (x0/e0)^2 = 1.
            x[0] = e[0];
            x[1] = (Real)0;
            distance = fabs(y[0] - e[0]);
        }
    }
    return distance;
}
//----------------------------------------------------------------------------
// The ellipse is (x0/e0)^2 + (x1/e1)^2 = 1.  The query point is (y0,y1).
// The function returns the distance from the query point to the ellipse.
// It also computes the ellipse point (x0,x1) that is closest to (y0,y1).
//----------------------------------------------------------------------------
template <typename Real>
Real DistancePointEllipse (const Real e[2], const Real y[2], Real x[2])
{
    // Determine reflections for y to the first quadrant.
    bool reflect[2];
    int i, j;
    for (i = 0; i < 2; ++i)
    {
        reflect[i] = (y[i] < (Real)0);
    }

    // Determine the axis order for decreasing extents.
    int permute[2];
    if (e[0] < e[1])
    {
        permute[0] = 1;  permute[1] = 0;
    }
    else
    {
        permute[0] = 0;  permute[1] = 1;
    }

    int invpermute[2];
    for (i = 0; i < 2; ++i)
    {
        invpermute[permute[i]] = i;
    }

    Real locE[2], locY[2];
    for (i = 0; i < 2; ++i)
    {
        j = permute[i];
        locE[i] = e[j];
        locY[i] = y[j];
        if (reflect[j])
        {
            locY[i] = -locY[i];
        }
    }

    Real locX[2];
    Real distance = DistancePointEllipseSpecial(locE, locY, locX);

    // Restore the axis order and reflections.
    for (i = 0; i < 2; ++i)
    {
        j = invpermute[i];
        if (reflect[j])
        {
            locX[j] = -locX[j];
        }
        x[i] = locX[j];
    }

    return distance;
}
//----------------------------------------------------------------------------
```

3 Distance from a Point to an Ellipsoid

A general ellipsoid in 3D is represented by a center point $\mathbf{C}$, an orthonormal set of axis-direction vectors $\{\mathbf{U}_0, \mathbf{U}_1, \mathbf{U}_2\}$, and associated extents e_i with $e_0 \geq e_1 \geq e_2 > 0$. The ellipsoid points are

$$\mathbf{P} = \mathbf{C} + x_0\mathbf{U}_0 + x_1\mathbf{U}_1 + x_2\mathbf{U}_2 \tag{20}$$

where

$$\left(\frac{x_0}{e_0}\right)^2 + \left(\frac{x_1}{e_1}\right)^2 + \left(\frac{x_2}{e_2}\right)^2 = 1 \tag{21}$$

If $e_0 = e_1 = e_2$, then the ellipsoid is a sphere with center $\mathbf{C}$ and radius e_0. If $e_0 = e_1 > e_2$, the ellipsoid is said to be an *oblate spheroid.* If $e_0 > e_1 = e_2$, the ellipsoid is said to be a *prolate spheroid.* The orthonormality of the axis directions and Equation (20) imply $x_i = \mathbf{U}_i \cdot (\mathbf{P} - \mathbf{C})$. Substituting this into Equation (21), we obtain

$$(\mathbf{P} - \mathbf{C})^{\mathrm{T}} M (\mathbf{P} - \mathbf{C}) = 1 \tag{22}$$

where $M = RDR^{\mathrm{T}}$ with R an orthogonal matrix whose columns are $\mathbf{U}_0$, $\mathbf{U}_1$, and $\mathbf{U}_2$ and with D a diagonal matrix whose diagonal entries are $1/e_0^2$, $1/e_1^2$, and $1/e_2^2$.

The problem is to compute the distance from a point $\mathbf{Q}$ to the ellipsoid. It is sufficient to solve this problem in the coordinate system of the ellipsoid; that is, represent $\mathbf{Q} = \mathbf{C} + y_0\mathbf{U}_0 + y_1\mathbf{U}_1 + y_2\mathbf{U}_2$. The distance from $\mathbf{Q}$ to the closest point $\mathbf{P}$ on the ellipsoid is the same as the distance from $\mathbf{Y} = (y_0, y_1, y_2)$ to the closest point $\mathbf{X} = (x_0, x_1, x_2)$ on the standard ellipsoid of Equation (21).

As in the 2D ellipse problem, we may additionally use symmetry to simplify the construction. It is sufficient to consider the case when (y_0, y_1, y_2) is in the first octant: $y_0 \geq 0$, $y_1 \geq 0$, and $y_2 \geq 0$.

3.1 The Closest Point's Normal is Directed Toward the Query Point

A parameterization of the standard ellipsoid is $\mathbf{X}(\theta, \phi) = (e_0 \cos\theta \sin\phi, e_1 \sin\theta \sin\phi, e_2 \cos\phi)$ for $\theta \in [0, 2\pi)$ and $\phi \in [0, \pi]$. The squared distance from $\mathbf{Y}$ to any point on the ellipsoid is

$$F(\theta, \phi) = |\mathbf{X}(\theta, \phi) - \mathbf{Y}|^2 \tag{23}$$

This is a nonnegative, doubly periodic, and differentiable function; it must have a global minimum occurring at angles for which the first-order partial derivatives are zero,

$$\frac{\partial F}{\partial \theta} = 2(\mathbf{X}(\theta, \phi) - \mathbf{Y}) \cdot \frac{\partial \mathbf{X}}{\partial \theta} = 0, \quad \frac{\partial F}{\partial \phi} = 2(\mathbf{X}(\theta, \phi) - \mathbf{Y}) \cdot \frac{\partial \mathbf{X}}{\partial \phi} = 0 \tag{24}$$

For the derivatives to be zero, the vector $(\mathbf{X}(\theta, \phi) - \mathbf{Y})$ must be perpendicular to the tangent vectors $\partial\mathbf{X}/\partial\theta$ and $\partial\mathbf{X}/\partial\phi$. This implies the vector from $\mathbf{Y}$ to the closest ellipsoid point $\mathbf{X}$ must be normal to the surface at $\mathbf{X}$. Using the implicit form of the ellipsoid, namely, $G(x_0, x_1, x_2) = (x_0/e_0)^2 + (x_1/e_1)^2(+x_2/e_2)^2 - 1$, the gradient of $G(x_0, x_1, x_2)$ is a normal vector to the ellipsoid at (x_0, x_1, x_2). Half the gradient is also normal, so we have $(y_0, y_1, y_2) - (x_0, x_1, x_2) = t\nabla G(x_0, x_1, x_2)/2 = t(x_0/e_0^2, x_1/e_1^2, x_2/e_2^2)$ for some scalar t, or

$$y_0 = x_0\left(1 + \frac{t}{e_0^2}\right), \quad y_1 = x_1\left(1 + \frac{t}{e_1^2}\right), \quad y_2 = x_2\left(1 + \frac{t}{e_2^2}\right) \tag{25}$$

If (y_0, y_1, y_2) is outside the ellipsoid, it is necessary that $t > 0$. If (y_0, y_1, y_2) is inside the ellipsoid, it is necessary that $t < 0$. Of course, if (y_0, y_1, y_2) is already on the ellipsoid, then $t = 0$ and the distance is zero.

3.2 The Case of a Sphere

If $e_0 = e_1 = e_2$, then the ellipsoid is a sphere. The origin $(0, 0, 0)$ has infinitely many closest sphere points (all of them), but clearly the distance from the origin to the sphere is e_0. The closest sphere point to $(y_0, y_1, y_2) \neq (0, 0, 0)$ is $(x_0, x_1, x_2) = e_0(y_0, y_1, y_2)/|(y_0, y_1, y_2)|$. Equation (25) is consistent with this, because it implies $(x_0, x_1, x_2) = (e_0^2/(t+e_0^2))(y_0, y_1, y_2)$ for some t; that is, (x_0, x_1, x_2) is parallel to (y_0, y_1, y_2) and must have length e_0. It is easily shown that $t = -e_0^2 + e_0\sqrt{y_0^2 + y_1^2 + y_2^2}$.

3.3 The Case of an Oblate Spheroid

If $e_0 = e_1 > e_2$, then the ellipsoid is an oblate spheroid. The standard ellipse equation reduces to $(x_0^2 + x_1^2)/e_0^2 + x_2^2/e_2^2 = 1$, which is the equation of an ellipse in the (r, x_2)-plane where $r = \sqrt{x_0^2 + x_1^2}$. An implementation can make the reduction by mapping the query point (y_0, y_1, y_2) to $(\sqrt{y_0^2 + y_1^2}, y_2)$ and using the point-ellipse algorithm discussed in Section 2.

3.4 The Case of a Prolate Spheroid

If $e_0 > e_1 = e_2$, then the ellipsoid is a prolate spheroid. The standard ellipse equation reduces to $x_0^2/e_0^2 + (x_1^2 + x_2^2/e_2^2 = 1$, which is the equation of an ellipse in the (x_0, r)-plane where $r = \sqrt{x_1^2 + x_2^2}$. An implementation can make the reduction by mapping the query point (y_0, y_1, y_2) to $(y_0, \sqrt{y_1^2 + y_2^2}$ and using the point-ellipse algorithm discussed in Section 2.

For the remainder of Section 3, we assume that $e_0 > e_1 > e_2$ and that all analysis is restricted to the first octant.

3.5 The Query Point is the Origin

Equation (25) becomes $0 = x_i(1 + t/e_i^2)$ for $i = 0, 1, 2$. The construction of candidate points is similar to that shown in Section 2.3. There are eight subcases to consider. The subcase $x_0 = x_1 = x_2 = 0$ is discarded because the point is not on the ellipsoid. Any two subcases that require $t = -e_i^2$ and $t = -e_j^2$ with $i \neq j$ are contradictory because $e_i \neq e_j$; there are four such subcases. The remaining three subcases lead to candidates $(e_0, 0, 0)$, $(0, e_1, 0)$, and $(0, 0, e_2)$. Of these, $(0, 0, e_2)$ is closest to the origin. In summary: *The closest ellipsoid point in the first octant to* $(0, 0, 0)$ *is* $(0, 0, e_2)$ *with distance* $d = e_2$.

3.6 The Query Point has $y_2 > 0$

Equation (25) leads to $y_2 = x_2(1 + t/e_2^2)$. Because $y_2 > 0$ and the search for closest point is in the first octant, it must be that $x_2 > 0$ and $1 + t/e_2^2 > 0$. We have four cases to consider depending on whether y_0 or y_1 are zero or positive.

- Let $y_0 = 0$ and $y_1 = 0$; then $0 = x_0(1 + t/e_0^2)$ and $0 = x_1(1 + t/e_1^2)$. If $x_0 > 0$, then $t = -e_0^2$ and $y_2 = x_2(1 - e_0^2/e_1^2) < 0$. The last inequality is due to $e_0 > e_1$, but is a contradiction because $y_2 > 0$. Similarly, if $x_1 > 0$, then $t = -e_1^2$ and $y_2 = x_2(1 - e_1^2/e_2^2) < 0$, also a contradiction. It must be that $x_0 = 0$ and $x_1 = 0$, so the closest ellipsoid point is $(x_0, x_1, x_2) = (0, 0, e_2)$.
- Let $y_0 > 0$ and $y_1 = 0$; then $y_0 = x_0(1 + t/e_0^2)$, in which case $x_0 > 0$. We saw previously that $y_1 = 0$ forces $x_1 = 0$. This reduces the 3D point-ellipsoid problem to the 2D point-ellipse problem where the ellipse is $(x_0/e_0)^2 + (x_2/e_2)^2 = 1$ with query point (y_0, y_2) whose components are both positive; see Section 2.6.
- Let $y_1 > 0$ and $y_0 = 0$; then $y_1 = x_1(1 + t/e_1^2)$, in which case $x_1 > 0$. We saw previously that $y_0 = 0$ forces $x_0 = 0$. This reduces the 3D point-ellipsoid problem to the 2D point-ellipse problem where the ellipse is $(x_1/e_1)^2 + (x_2/e_2)^2 = 1$ with query point (y_1, y_2) whose components are both positive.; see Section 2.6.

- Let $y_0 > 0$ and $y_1 > 0$; then Equation (25) is solved for $x_i = e_i^2 y_i/(t+e_i^2)$ for $0 \le i \le 2$. The algorithm for computing the closest ellipsoid point is similar to that of Section 2.6. Substituting the x_i into Equation (21), we obtain

$$F(t) = \left(\frac{e_0 y_0}{t+e_0^2}\right)^2 + \left(\frac{e_1 y_1}{t+e_1^2}\right)^2 + \left(\frac{e_2 y_2}{t+e_2^2}\right)^2 - 1 = 0 \tag{26}$$

The first equality defines the function $F(t)$ with domain $(-e_2^2, \infty)$. The candidates for the ellipsoid point (x_0, x_1, x_2) closest to (y_0, y_1, y_2) are generated by the roots t to $F(t) = 0$.

The first-order and second-order derivatives of F are

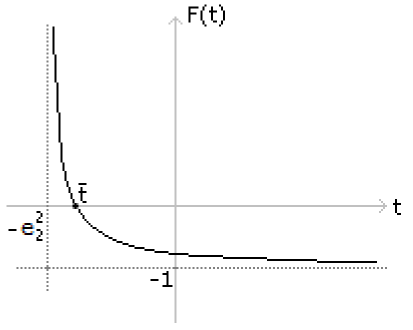
$$F'(t) = \frac{-2e_0^2 y_0^2}{(t+e_0^2)^3} + \frac{-2e_1^2 y_1^2}{(t+e_1^2)^3} + \frac{-2e_2^2 y_2^2}{(t+e_2^2)^3}, \quad F''(t) = \frac{6e_0^2 y_0^2}{(t+e_0^2)^4} + \frac{6e_1^2 y_1^2}{(t+e_1^2)^4} + \frac{6e_2^2 y_2^2}{(t+e_2^2)^4} \tag{27}$$

We know that $y_0 > 0$, $y_1 > 0$, $y_2 > 0$, and $t > -e_2^2$. These conditions imply $F'(t) < 0$ and $F''(t) > 0$ for $t > -e_2^2$. Observe that

$$\lim_{t \to -e_2^{2+}} F(t) = +\infty, \quad \lim_{t \to \infty} F(t) = -1 \tag{28}$$

The first expression is a one-sided limit where t approaches e_2^2 through values larger than e_2^2. We have shown that $F(t)$ is a strictly decreasing function for $t \in (-e_2^2, +\infty)$ that is initially positive, then becomes negative. Consequently it has a unique root on the specified domain. Figure 3.1 shows a typical graph of $F(t)$.

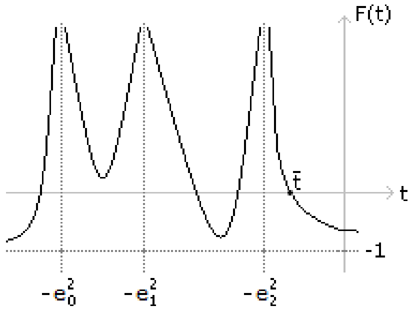
Figure 3.1 A typical graph of $F(t)$ for $t > -e_2^2$. The unique root $\bar{t}$ is shown.



Observe that the domain $(-e_2^2, \infty)$ contains 0 and that $F(0) = (y_0/e_0)^2 + (y_1/e_1)^2 + (y_2/e_2)^2 - 1$. When (y_0, y_1, y_2) is inside the ellipsoid, $F(0) < 0$, in which case $\bar{t} < 0$. When $(y_0, y_1, y_2))$ is outside the ellipsoid, $F(0) > 0$, which does not explicitly constrain the sign of $\bar{t}$, but we do know geometrically that $\bar{t} > 0$. When (y_0, y_1, y_2) is on the ellipse, $F(0) = 0$ and so $\bar{t} = 0$.

For all t, a typical graph of $F(t)$ is shown in Figure 3.2.

Figure 3.2 A typical graph of $F(t)$ when $y_0 > 0$, $y_1 > 0$, and $y_2 > 0$.



The illustration shows $F(t)$ with 4 roots. However, it is possible for $F(t)$ to have up to 6 roots.

3.7 The Query Point has $y_2 = 0$

The construction is analogous to that of Section 2.5. Equation (25) states that $y_0 = x_0(1 + t/e_0^2)$, $y_1 = x_1(1 + t/e_1^2)$, and $0 = x_2(1 + t/e_2^2)$. The last equation has two cases.

- Let $x_2 = 0$. For $(x_0, x_1, 0)$ to be on the ellipsoid, we need $(x_0/e_0)^2 + (x_1/e_1)^2 = 1$. The 3D point-ellipsoid problem is therefore reduced to the 2D point-ellipse problem with query point (y_0, y_1).
- Let $x_2 > 0$; then $t = -e_2^2$, $y_0 = x_0(1-e_2^2/e_0^2)$, and $y_1 = x_1(1-e_2^2/e_1^2)$. It follows that $x_0 = e_0^2 y_0/(e_0^2-e_2^2)$ and $x_1 = e_1^2 y_1/(e_1^2 - e_2^2)$. Because (x_0, x_1, x_2) is on the ellipsoid, we know that

$$1 \geq 1 - \left(\frac{x_2}{e_2}\right)^2 = \left(\frac{x_0}{e_0}\right)^2 + \left(\frac{x_1}{e_1}\right)^2 = \left(\frac{e_0 y_0}{e_0^2 - e_2^2}\right)^2 + \left(\frac{e_1 y_1}{e_1^2 - e_2^2}\right)^2 \tag{29}$$

  The implication is that (y_0, y_1) is contained by a subellipse of the x_0x_1-domain for the hemiellipsoid $x_2 \geq 0$. For (y_0, y_1) outside this subellipse, the closest point is necessarily on the ellipse $(x_0/e_0)^2 + (x_1/e_1)^2 = 1$. When (y_0, y_1) is inside the subellipse, for (x_0, x_1, x_2) to be on the ellipsoid, we may solve for $x_2 = e_2\sqrt{1 - (x_0/e_0)^2 - (x_1/e_1)^2}$.

3.8 A Summary of the Mathematical Algorithm

Pseudocode is listed next that summarizes the algorithm for computing the ellipsoid point (x_0, x_1, x_2) closest to the query point (y_0, y_1, y_2) in the first octant. The function also computes the distance.

```
Real Distance (Real e0, Real e1, Real e2, Real y0, Real y1, Real y2, Real& x0, Real& x1, Real& x2)
{
    Real distance;
    if (y2 > 0)
    {
        if (y1 > 0)
        {
            if (y0 > 0)
            {
                Compute the unique root tbar of F(t) on (-e2*e2,+infinity);
                x0 = e0*e0*y0/(tbar + e0*e0);
                x1 = e1*e1*y1/(tbar + e1*e1);
                x2 = e2*e2*y2/(tbar + e2*e2);
                distance = sqrt((x0 - y0)*(x0 - y0) + (x1 - y1)*(x1 - y1) + (x2 - y2)*(x2 - y2));
            }
            else  // y0 == 0
            {
                x0 = 0;
                distance = Distance(e1, e2, y1, y2, x1, x2);
            }
        }
        else  // y1 == 0
        {
            x1 = 0;
            if (y0 > 0)
            {
                distance = DistancePointEllipseSpecial<Real>(e0, e2, y0, y2, x0, x2);
            }
            else  // y0 == 0
            {
                x0 = 0;
                x2 = e2;
                distance = abs(y2 - e2);
            }
        }
    }
    else  // y2 == 0
    {
        Real denom0 = e0*e0 - e2*e2, denom1 = e1*e1 - e2*e2, ey0 = e0*y0, ey1 = e1*y1;
        bool inSubEllipse = false;
        bool inAABBSubEllipse = (ey0 < denom0 && ey1 < denom1);
        if (inAABBSubEllipse)
        {
            Real xde0 = ey0/denom0, xde1 = ey1/denom1;
            Real xde0sqr = xde0*xde0, xde1sqr = xde1*xde1;
            Real discr = 1 - xde0sqr - xde1sqr;
            if (discr > 0)
            {
                x0 = e0*xde0;
                x1 = e1*xde1;
                x2 = e2*sqrt(discr);
                distance = sqrt((x0 - y0)*(x0 - y0) + (x1 - y1)*(x1 - y1) + x2*x2);
                inSubEllipse = true;
            }
        }
        if (!inSubEllipse)
        {
            x2 = 0;
            distance = Distance(e0, e1, y0, y1, x0, x1);
        }
    }
    return distance;
}
```

3.9 Robust Root Finding and Conversion to a Polynomial

The ideas of Section 2.8 apply equally well in the point-ellipsoid problem. The bisection method is robust. The initial bounding interval is $[t_0, t_1]$ with $t_0 = -e_2^2 + e_2 y_2$ and $t_1 = -e_2^2 + \sqrt{(e_0 y_0)^2 + (e_1 y_1)^2 + (e_2 y_2)^2}$.

Newton's Method may be used with initial guess t_0, but the same numerical issues must be addressed when y_2 is nearly zero (in the limit as y_2 goes to zero, the vertical asymptote disappears). A hybrid of Newton's Method and bisection should be used to guarantee convergence to the root.

The conversion to a polynomial of degree 6 is suggested in [1]. The roots of the polynomial may be computed, leading to candidates for the closest ellipsoid point:

$$P(t) = (t+e_0^2)^2(t+e_1^2)^2(t+e_2^2)^2F(t) = e_0^2y_0^2(t+e_1^2)^2(t+e_2^2)^2+e_1^2y_1^2(t+e_0^2)^2(t+e_2^2)^2+e_2^2y_2^2(t+e_0^2)^2(t+e_1^2)^2 \quad (30)$$

When $y_2 > 0$, the roots of $P(t)$ are the same as those for $F(t)$, and the largest root of $P(t)$ is the same as the largest root of $F(t)$. However, when $y_2 = 0$, $P(t)$ has a double root at $t = -e_2^2$ but $F(t)$ does not have a root at $t = -e_2^2$. Just as in the point-ellipse distance algorithm, this causes numerical problems when y_2 is nearly zero (and when any of the y_i are nearly zero).

The Graphics Gems IV article suggests using Newton's Method starting with an initial guess greater than the largest root of $P(t)$, namely, $t_0 = e_0\sqrt{y_0^2 + y_1^2 + y_2^2}$. As in Section 2.8.3, the convergence is not guaranteed, especially when y_2 is nearly zero. The discussion about double roots in that section applies here as well, and if you choose to use Newton's Method for $P(t)$, a better initial guess is $t_1 = -e_1 + \sqrt{e_0^2y_0^2 + e_1^2y_1^2 + e_2^2y_2^2}$. It may be shown that $t_1 < t_1 + e_2 < t_0$ (same argument as in the section for point-ellipse).

3.10 An Implementation

An implementation that uses the robust bisection method, where the iterates are computed until the bounding interval endpoints are consecutive floating-point numbers is listed next. The full source code is found in

```
WildMagic5/SampleMathematics/DistancePointEllipseEllipsoid
```

The function `DistancePointEllipsoidSpecial` requires that $e_0 \geq e_1 \geq e_2$, $y_0 \geq 0$, $y_1 \geq 0$, and $y_2 \geq 0$. The function `DistancePointEllipsoid` is the general function to call. It reflects components of (y_0, y_1, y_2) as needed to place the query point in the first octant, sorts the (e_0, e_1, e_2) values to be correctly ordered, and then calls `DistancePointEllipsoidSpecial`. The returned (x_0, x_1, x_2) is postprocessed to match the octant of the query point.

The code also calls the point-ellipse distance function when the point-ellipsoid query reduces to a 2D problem. As we will see in the last section of this document, the recursion on dimension is not necessary. The point-hyperellipsoid query does all the necessary work to compute the distance and closest point.

```
//----------------------------------------------------------------------------
// The ellipsoid is (x0/e0)^2 + (x1/e1)^2 + (x2/e2)^2 = 1 with e0 >= e1 >= e2.
// The query point is (y0,y1,y2) with y0 >= 0, y1 >= 0, and y2 >= 0.  The
// function returns the distance from the query point to the ellipsoid.  It
// also computes the ellipsoid point (x0,x1,x2) in the first octant that is
// closest to (y0,y1,y2).
//----------------------------------------------------------------------------
template <typename Real>
Real DistancePointEllipsoidSpecial (const Real e[3], const Real y[3],
    Real x[3])
{
    Real distance;
    if (y[2] > (Real)0)
    {
        if (y[1] > (Real)0)
```

```
{
    if (y[0] > (Real)0)
    {
        // Bisect to compute the root of F(t) for t >= -e2*e2.
        Real esqr[3] = { e[0]*e[0], e[1]*e[1], e[2]*e[2] };
        Real ey[3] = { e[0]*y[0], e[1]*y[1], e[2]*y[2] };
        Real t0 = -esqr[2] + ey[2];
        Real t1 = -esqr[2] + sqrt(ey[0]*ey[0] + ey[1]*ey[1] +
            ey[2]*ey[2]);
        Real t = t0;
        const int imax = 2*std::numeric_limits<Real>::max_exponent;
        for (int i = 0; i < imax; ++i)
        {
            t = ((Real)0.5)*(t0 + t1);
            if (t == t0 || t == t1)
            {
                break;
            }

            Real r[3] = { ey[0]/(t + esqr[0]), ey[1]/(t + esqr[1]),
                ey[2]/(t + esqr[2]) };
            Real f = r[0]*r[0] + r[1]*r[1] + r[2]*r[2] - (Real)1;
            if (f > (Real)0)
            {
                t0 = t;
            }
            else if (f < (Real)0)
            {
                t1 = t;
            }
            else
            {
                break;
            }
        }

        x[0] = esqr[0]*y[0]/(t + esqr[0]);
        x[1] = esqr[1]*y[1]/(t + esqr[1]);
        x[2] = esqr[2]*y[2]/(t + esqr[2]);
        Real d[3] = { x[0] - y[0], x[1] - y[1], x[2] - y[2] };
        distance = sqrt(d[0]*d[0] + d[1]*d[1] + d[2]*d[2]);
    }
    else  // y0 == 0
    {
        x[0] = (Real)0;
        Real etmp[2] = { e[1], e[2] };
        Real ytmp[2] = { y[1], y[2] };
        Real xtmp[2];
        distance = DistancePointEllipseSpecial<Real>(etmp, ytmp, xtmp);
        x[1] = xtmp[0];
        x[2] = xtmp[1];
    }
}
else  // y1 == 0
{
    x[1] = (Real)0;
    if (y[0] > (Real)0)
    {
        Real etmp[2] = { e[0], e[2] };
        Real ytmp[2] = { y[0], y[2] };
        Real xtmp[2];
        distance = DistancePointEllipseSpecial<Real>(etmp, ytmp, xtmp);
        x[0] = xtmp[0];
        x[2] = xtmp[1];
    }
    else  // y0 == 0
    {
        x[0] = (Real)0;
        x[2] = e[2];
        distance = fabs(y[2] - e[2]);
    }
}
```

```
    }
    else  // y2 == 0
    {
        Real denom[2] = { e[0]*e[0] - e[2]*e[2], e[1]*e[1] - e[2]*e[2] };
        Real ey[2] = { e[0]*y[0], e[1]*y[1] };
        if (ey[0] < denom[0] && ey[1] < denom[1])
        {
            // (y0,y1) is inside the axis-aligned bounding rectangle of the
            // subellipse.  This intermediate test is designed to guard
            // against the division by zero when e0 == e2 or e1 == e2.
            Real xde[2] = { ey[0]/denom[0], ey[1]/denom[1] };
            Real xdesqr[2] = { xde[0]*xde[0], xde[1]*xde[1] };
            Real discr = (Real)1 - xdesqr[0] - xdesqr[1];
            if (discr > (Real)0)
            {
                // (y0,y1) is inside the subellipse.  The closest ellipsoid
                // point has x2 > 0.
                x[0] = e[0]*xde[0];
                x[1] = e[1]*xde[1];
                x[2] = e[2]*sqrt(discr);
                Real d[2] = { x[0] - y[0], x[1] - y[1] };
                distance = sqrt(d[0]*d[0] + d[1]*d[1] + x[2]*x[2]);
            }
            else
            {
                // (y0,y1) is outside the subellipse.  The closest ellipsoid
                // point has x2 == 0 and is on the domain-boundary ellipse
                // (x0/e0)^2 + (x1/e1)^2 = 1.
                x[2] = (Real)0;
                distance = DistancePointEllipseSpecial<Real>(e, y, x);
            }
        }
        else
        {
            // (y0,y1) is outside the subellipse.  The closest ellipsoid
            // point has x2 == 0 and is on the domain-boundary ellipse
            // (x0/e0)^2 + (x1/e1)^2 = 1.
            x[2] = (Real)0;
            distance = DistancePointEllipseSpecial<Real>(e, y, x);
        }
    }
    return distance;
}
//----------------------------------------------------------------------------
// The ellipsoid is (x0/e0)^2 + (x1/e1)^2 + (x2/e2)^2 = 1.  The query point is
// (y0,y1,y2).  The function returns the distance from the query point to the
// ellipsoid.   It also computes the ellipsoid point (x0,x1,x2) that is
// closest to (y0,y1,y2).
//----------------------------------------------------------------------------
template <typename Real>
Real DistancePointEllipsoid (const Real e[3], const Real y[3], Real x[3])
{
    // Determine reflections for y to the first octant.
    bool reflect[3];
    int i, j;
    for (i = 0; i < 3; ++i)
    {
        reflect[i] = (y[i] < (Real)0);
    }

    // Determine the axis order for decreasing extents.
    int permute[3];
    if (e[0] < e[1])
    {
        if (e[2] < e[0])
        {
            permute[0] = 1;  permute[1] = 0;  permute[2] = 2;
        }
        else if (e[2] < e[1])
        {
            permute[0] = 1;  permute[1] = 2;  permute[2] = 0;
        }
        else
        {
            permute[0] = 2;  permute[1] = 1;  permute[2] = 0;
        }
    }
    else
    {
        if (e[2] < e[1])
        {
            permute[0] = 0;  permute[1] = 1;  permute[2] = 2;
        }
        else if (e[2] < e[0])
        {
            permute[0] = 0;  permute[1] = 2;  permute[2] = 1;
        }
        else
        {
            permute[0] = 2;  permute[1] = 0;  permute[2] = 1;
        }
    }

    int invpermute[3];
    for (i = 0; i < 3; ++i)
    {
        invpermute[permute[i]] = i;
    }

    Real locE[3], locY[3];
    for (i = 0; i < 3; ++i)
    {
        j = permute[i];
        locE[i] = e[j];
        locY[i] = y[j];
        if (reflect[j])
        {
            locY[i] = -locY[i];
        }
    }

    Real locX[3];
    Real distance = DistancePointEllipsoidSpecial(locE, locY, locX);

    // Restore the axis order and reflections.
    for (i = 0; i < 3; ++i)
    {
        j = invpermute[i];
        if (reflect[j])
        {
            locX[j] = -locX[j];
        }
        x[i] = locX[j];
    }

    return distance;
}
//----------------------------------------------------------------------------
```

4 Distance from a Point to a Hyperellipsoid

A general hyperellipsoid in n dimensions is represented by a center point $\mathbf{C}$, an orthonormal set of axis-direction vectors $\{\mathbf{U}_i\}_{i=0}^{n-1}$, and associated extents e_i with $e_{j+1} \geq e_j > 0$ for all j. The hyperellipsoid points are

$$\mathbf{P} = \mathbf{C} + \sum_{i=0}^{n-1} x_i \mathbf{U}_i \qquad (31)$$

where

$$\sum_{i=0}^{n-1} \left(\frac{x_i}{e_i}\right)^2 = 1 \tag{32}$$

As with an ellipse or an ellipsoid, special cases occur when some of the extents are equal. The implementation handles these correctly, so these are not required to have special code to deal with them. The orthonormality of the axis directions and Equation (31) imply $x_i = \mathbf{U}_i \cdot (\mathbf{P} - \mathbf{C})$. Substituting this into Equation (32), we obtain

$$(\mathbf{P} - \mathbf{C})^{\mathrm{T}} M (\mathbf{P} - \mathbf{C}) = 1 \tag{33}$$

where $M = RDR^{\mathrm{T}}$ with R an orthogonal matrix whose columns are the $\mathbf{U}_i$ vectors and with D a diagonal matrix whose diagonal entries are the $1/e_i^2$ values.

The problem is to compute the distance from a point $\mathbf{Q}$ to the hyperellipsoid. It is sufficient to solve this problem in the coordinate system of the hyperellipsoid; that is, represent $\mathbf{Q} = \mathbf{C} + \sum_{i=0}^{n-1} y_i \mathbf{U}_i$. The distance from $\mathbf{Q}$ to the closest point $\mathbf{P}$ on the hyperellipsoid is the same as the distance from $\mathbf{Y} = (y_0, \ldots, y_{n-1})$ to the closest point $\mathbf{X} = (x_0, \ldots, x_{n-1})$ on the standard hyperellipsoid of Equation (32).

We may additionally use symmetry to simplify the construction. It is sufficient to consider the case when $y_i \geq 0$ for all i.

4.1 The Closest Point's Normal is Directed Toward the Query Point

A parameterization of the standard hyperellipsoid involves $n-1$ parameters $\mathbf{\Theta} = (\theta_0, \ldots, \theta_{n-2})$, say, $\mathbf{X}(\mathbf{\Theta})$. The actual form of the components can use a generalization of spherical coordinates. The squared distance from $\mathbf{Y}$ to any point on the hyperellipsoid is

$$F(\mathbf{\Theta}) = |\mathbf{X}(\mathbf{\Theta}) - \mathbf{Y}|^2 \tag{34}$$

This is a nonnegative differentiable function that is periodic in each of its independent variables; it must have a global minimum occurring at parameters for which the first-order partial derivatives are zero,

$$\frac{\partial F}{\partial \theta_i} = 2(\mathbf{X}(\mathbf{\Theta}) - \mathbf{Y}) \cdot \frac{\partial \mathbf{X}}{\partial \theta_i} = 0 \tag{35}$$

for all i. For the derivatives to be zero, the vector $(\mathbf{X}(\mathbf{\Theta}) - \mathbf{Y})$ must be perpendicular to the tangent vectors $\partial \mathbf{X}/\partial \theta_i$. This implies the vector from $\mathbf{Y}$ to the closest hyperellipsoid point $\mathbf{X}$ must be normal to the surface at $\mathbf{X}$. Using the implicit form of the ellipsoid, namely, $G(\mathbf{X}) = \mathbf{X}^{\mathrm{T}} D \mathbf{X} - 1$, the gradient of $G(\mathbf{X})$ is a normal vector to the hyperellipsoid at $\mathbf{X}$. Half the gradient is also normal, so we have $\mathbf{Y} - \mathbf{X} = t \nabla G(\mathbf{X})/2 = tD\mathbf{X}$ for some scalar t, or

$$y_i = x_i \left(1 + \frac{t}{e_i^2}\right) \tag{36}$$

for all i. If $\mathbf{Y}$ is outside the hyperellipsoid, it is necessary that $t > 0$. If $\mathbf{Y}$ is inside the hyperellipsoid, it is necessary that $t < 0$. Of course, if $\mathbf{Y}$ is already on the hyperellipsoid, then $t = 0$ and the distance is zero.

4.2 The Key Observations

In the point-ellipse and point-ellipsoid distance algorithms, an important observation is that the main branching depends on whether the last component of $\mathbf{Y}$ is positive or zero.

- When $y_{n-1} > 0$, notice that whenever $y_i = 0$ for any $i < n-1$, the corresponding closest point has component $x_i = 0$.
- When $y_{n-1} = 0$, notice that two cases occur. The projection of $\mathbf{Y}$ onto the coordinate hyperplane $y_{n-1} = 0$ is either inside or outside a special hyperellipsoid living in $n-1$ dimensions (the original hyperellipsoid lives in n dimensions). If inside the special hyperellipsoid, we can compute the corresponding x_i directly from Equation (36), and then compute $x_{n-1} = \sqrt{1 - \sum_{i=0}^{n-2}(x_i/e_i)^2}$. If outside the special hyperellipsoid, we have the recursion in dimension—the query reduces to point-hyperellipsoid within the hyperplane $y_{n-1} = 0$.

This suggests that we should compute a vector $\mathbf{Y}'$ from $\mathbf{Y}$ by storing all the positive components, construct a vector $\mathbf{e}'$ that stores the extents corresponding to those positive components, and then reduce the problem either to a bisection in a smaller-dimensional space or to a direct computation of the components in a smaller-dimensional space.

Let there be p positive components of $\mathbf{Y}$ located at the indices i_0 through i_{p-1}. If $y_{n-1} > 0$, then $i_{p-1} = n-1$. Construct $\mathbf{Y}' = (y_{i_0}, \ldots, y_{i_{p-1}})$. Let $\mathbf{e}' = (e_{i_0}, \ldots, e_{i_{p-1}})$ and $\mathbf{X}' = (x_{i_0}, \ldots, x_{i_{p-1}})$. The pseudocode is

```
construct Y' and e';
set X[i] = 0 whenever Y[i] = 0;
if (y[n-1] > 0)
{
    Compute the unique root tbar of F(t) = -1 + sum_{j=0}^{p-1} (e'[j]*Y'[j]/(t + e'[j]*e'[j]))^2;
    X'[j] = e'[j]*e'[j]*Y'[j]/(tbar + e'[j]*e'[j]) for all j;
    distance = Length(X' - Y');
}
else  // y[n-1] == 0
{
    a[j] = (e'[j]*e'[j] - e[n-1]*e[n-1])/e'[j] for all j;
    Let A = diag(a[0],...);
    if Y' is in the hyperellipsoid Transpose(Y')*A*Y' = 1 then
    {
        X'[j] = e'[j]*Y'[j]/a[j] for all j;
        X'[last] = X[n-1] = e[n-1]*sqrt(1 - sum_j (X'[j]/e'[j])*(X'[j]/e'[j]));
        distance = Length(X' - Y');
    }
    else
    {
        X[n-1] = 0;
        Compute the unique root tbar of F(t) = -1 + sum_{j=0}^{p-2} (e'[j]*Y'[j]/(t + e'[j]*e'[j]))^2;
        X'[j] = e'[j]*e'[j]*Y'[j]/(tbar + e'[j]*e'[j]) for all j;
        distance = Length(X' - Y');
    }
}
copy X' components to the correct locations in X;
```

4.3 An Implementation

An implementation that uses the robust bisection method, where the iterates are computed until the bounding interval endpoints are consecutive floating-point numbers is listed next. The full source code is found in

```
WildMagic5/SampleMathematics/DistancePointEllipseEllipsoid
```

The function `DistancePointHyperellipsoidSpecial` requires that the e_i are sorted in decreasing order and that $y_i \geq 0$ for all i. The function `DistancePointHyperellipsoid` is the general function to call. It reflects components of $\mathbf{Y}$ as needed to make the components nonnegative, sorts the e_i, and then calls `DistancePointHyperellipsoidSpecial`. The returned $\mathbf{X}$ is postprocessed to match the original state/location of the query point.

The bisection is implemented separately, because it is called with different number of components in the two locations mentioned in the pseudocode.

The pseudocode test for whether $\mathbf{Y}'$ is in the special hyperellipsoid has a division by zero if an a_j is zero. The actual code handles this correctly by testing first whether $\mathbf{Y}'$ is in the axis-aligned bounding hyperbox of the hyperellipsoid. If it is not, the division by zero is avoided.

```
//----------------------------------------------------------------------------
// Bisect for the root of
//   F(t) = sum_{j=0}{m-1} (e[i[j]]*y[i[j]]/(t + e[i[j]]*e[i[j]])
// for t >= -e[i[m-1]]*e[i[m-1]].  The incoming e[] and y[] values are those
// for which the query point components are positive.
//----------------------------------------------------------------------------
template <int N, typename Real>
Real DPHSBisector (int numComponents, const Real e[N], const Real y[N],
    Real x[N])
{
    Real esqr[N], ey[N], argument = (Real)0;
    int i;
    for (i = 0; i < numComponents; ++i)
    {
        esqr[i] = e[i]*e[i];
        ey[i] = e[i]*y[i];
        argument += ey[i]*ey[i];
    }

    Real t0 = -esqr[numComponents-1] + ey[numComponents-1];
    Real t1 = -esqr[numComponents-1] + sqrt(argument);
    Real t = t0;
    const int jmax = 2*std::numeric_limits<Real>::max_exponent;
    for (int j = 0; j < jmax; ++j)
    {
        t = ((Real)0.5)*(t0 + t1);
        if (t == t0 || t == t1)
        {
            break;
        }

        Real f = (Real)-1;
        for (i = 0; i < numComponents; ++i)
        {
            Real r = ey[i]/(t + esqr[i]);
            f += r*r;
        }
        if (f > (Real)0)
        {
            t0 = t;
        }
        else if (f < (Real)0)
        {
            t1 = t;
        }
        else
        {
            break;
        }
    }

    Real distance = (Real)0;
    for (i = 0; i < numComponents; ++i)
    {
        x[i] = esqr[i]*y[i]/(t + esqr[i]);
        Real diff = x[i] - y[i];
        distance += diff*diff;
    }
    distance = sqrt(distance);
    return distance;
}
//----------------------------------------------------------------------------
// The hyperellipsoid is sum_{i=0}^{N-1}(x[i]/e[i])^2 = 1 with e[i+1] >= e[i]
// for all i.  The query point is (y[0],...,y[N-1]) with y[i] >= 0 for all i.
// The function returns the distance from the query point to the
// hyperellipsoid.  It also computes the hyperellipsoid point
// (x[0],...,x[N-1]) closest to (y[0],...,y[N-1]), with x[i] >= 0 for all i.
//----------------------------------------------------------------------------
template <int N, typename Real>
Real DistancePointHyperellipsoidSpecial (const Real e[N], const Real y[N],
    Real x[N])
{
    Real distance;

    Real ePos[N], yPos[N], xPos[N];
    int numPos = 0;
    int i;
    for (i = 0; i < N; ++i)
    {
        if (y[i] > (Real)0)
        {
            ePos[numPos] = e[i];
            yPos[numPos] = y[i];
            ++numPos;
        }
        else
        {
            x[i] = (Real)0;
        }
    }

    if (y[N-1] > (Real)0)
    {
        distance = DPHSBisector<N,Real>(numPos, ePos, yPos, xPos);
    }
    else  // y[N-1] = 0
    {
        Real eNm1Sqr = e[N-1]*e[N-1];
        Real denom[N-1], ey[N-1];
        for (i = 0; i < numPos; ++i)
        {
            denom[i] = ePos[i]*ePos[i] - eNm1Sqr;
            ey[i] = ePos[i]*yPos[i];
        }

        bool inSubHyperbox = true;
        for (i = 0; i < numPos; ++i)
        {
            if (ey[i] >= denom[i])
            {
                inSubHyperbox = false;
                break;
            }
        }

        bool inSubHyperellipsoid = false;
        if (inSubHyperbox)
        {
            // yPos[] is inside the axis-aligned bounding box of the
            // subhyperellipsoid.  This intermediate test is designed to guard
            // against the division by zero when ePos[i] == e[N-1] for some i.
            Real xde[N-1], discr = (Real)1;
            for (i = 0; i < numPos; ++i)
            {
                xde[i] = ey[i]/denom[i];
                discr -= xde[i]*xde[i];
            }
            if (discr > (Real)0)
            {
                // yPos[] is inside the subhyperellipsoid.  The closest
                // hyperellipsoid point has x[N-1] > 0.
                distance = (Real)0;
                for (i = 0; i < numPos; ++i)
                {
                    xPos[i] = ePos[i]*xde[i];
                    Real diff = xPos[i] - yPos[i];
                    distance += diff*diff;
                }
                x[N-1] = e[N-1]*sqrt(discr);
                distance += x[N-1]*x[N-1];
                distance = sqrt(distance);
                inSubHyperellipsoid = true;
            }
        }

        if (!inSubHyperellipsoid)
        {
            // yPos[] is outside the subhyperellipsoid.  The closest
            // hyperellipsoid point has x[N-1] == 0 and is on the
            // domain-boundary hyperellipsoid.
            x[N-1] = (Real)0;
            distance = DPHSBisector<N,Real>(numPos, ePos, yPos, xPos);
        }
    }

    // Fill in those x[] values that were not zeroed out initially.
    for (i = 0, numPos = 0; i < N; ++i)
    {
        if (y[i] > (Real)0)
        {
            x[i] = xPos[numPos];
            ++numPos;
        }
    }

    return distance;
}
//----------------------------------------------------------------------------
// The hyperellipsoid is sum_{i=0}^{N-1}(x[i]/e[i])^2 = 1.  The query point is
// (y[0],...,y[N-1]).  The function returns the distance from the query point
// to the hyperellipsoid.   It also computes the hyperellipsoid point
// (x[0],...,x[N-1]) that is closest to (y[0],...,y[N-1]).
//----------------------------------------------------------------------------
template <int N, typename Real>
Real DistancePointHyperellipsoid (const Real e[N], const Real y[N], Real x[N])
{
    // Determine reflections for y to the first octant.
    bool reflect[N];
    int i, j;
    for (i = 0; i < N; ++i)
    {
        reflect[i] = (y[i] < (Real)0);
    }

    // Determine the axis order for decreasing extents.
    std::vector<std::pair<Real,int> > permute(N);
    for (i = 0; i < N; ++i)
    {
        permute[i].first = -e[i];
        permute[i].second = i;
    }
    std::sort(permute.begin(), permute.end());

    int invpermute[N];
    for (i = 0; i < N; ++i)
    {
        invpermute[permute[i].second] = i;
    }

    Real locE[N], locY[N];
    for (i = 0; i < N; ++i)
    {
        j = permute[i].second;
        locE[i] = e[j];
        locY[i] = y[j];
        if (reflect[j])
        {
            locY[i] = -locY[i];
        }
    }

    Real locX[N];
    Real distance =
        DistancePointHyperellipsoidSpecial<N,Real>(locE, locY, locX);

    // Restore the axis order and reflections.
    for (i = 0; i < N; ++i)
    {
        j = invpermute[i];
        if (reflect[j])
        {
            locX[j] = -locX[j];
        }
        x[i] = locX[j];
    }

    return distance;
}
//----------------------------------------------------------------------------
```

References

[1] John Hart, *Distance to an Ellipsoid*, in Graphics Gems IV, Paul Heckbert ed., Academic Press, Inc., New York NY, pp. 113-119, 1994.