Robust Computation of Distance Between Line Segments

David Eberly
Geometric Tools, LLC
http://www.geometrictools.com/
Copyright © 1998-2015. All Rights Reserved.

Created: January 31, 2015

Contents

1 Introduction

This document is a major rewrite of the previous version, *Distance Between Two Line Segments in 3D*, motivated by technical support questions regarding problems with the implementation when two segments are nearly parallel. The mathematics of the algorithm in the previous version are correct, but care must be taken with an implementation when computing with floating-point arithmetic. It is desirable to have a robust implementation that performs well–this document provides that. Moreover, the algorithm has been implemented on a GPU using double-precision arithmetic.

2 Mathematical Formulation

The algorithm applies to segments in any dimension. Let the endpoints of the first segment be $\mathbf{P}_0$ and $\mathbf{P}_1$ and the endpoints of the second segment be $\mathbf{Q}_0$ and $\mathbf{Q}_1$. The segments can be parameterized by $\mathbf{P}(s) = (1-s)\mathbf{P}_0 + s\mathbf{P}_1$ for $s \in [0,1]$ and $\mathbf{Q}(t) = (1-t)\mathbf{Q}_0 + t\mathbf{Q}_1$ for $t \in [0,1]$.

The squared distance between two points on the segments is the quadratic function

$$R(s,t) = |\mathbf{P}(s) - \mathbf{Q}(t)|^2 = as^2 - 2bst + ct^2 + 2ds - 2et + f \tag{1}$$

where

$$\begin{aligned} a &= (\mathbf{P}_1 - \mathbf{P}_0) \cdot (\mathbf{P}_1 - \mathbf{P}_0), \quad b = (\mathbf{P}_1 - \mathbf{P}_0) \cdot (\mathbf{Q}_1 - \mathbf{Q}_0), \quad c = (\mathbf{Q}_1 - \mathbf{Q}_0) \cdot (\mathbf{Q}_1 - \mathbf{Q}_0), \\ d &= (\mathbf{P}_1 - \mathbf{P}_0) \cdot (\mathbf{P}_0 - \mathbf{Q}_0), \quad e = (\mathbf{Q}_1 - \mathbf{Q}_0) \cdot (\mathbf{P}_0 - \mathbf{Q}_0), \quad f = (\mathbf{P}_0 - \mathbf{Q}_0) \cdot (\mathbf{P}_0 - \mathbf{Q}_0) \end{aligned} \tag{2}$$

For nondegenerate segments, $a > 0$ and $c > 0$, but the implementation will allow degenerate segments and handle them correctly. In the degenerate cases, we have either a point-segment pair or a point-point pair. Observe that

$$ac - b^2 = |(\mathbf{P}_1 - \mathbf{P}_0) \times (\mathbf{Q}_1 - \mathbf{Q}_0)|^2 \geq 0 \tag{3}$$

The segments are not parallel when their direction vectors are linearly independent, in which case the cross product of directions is not zero and $ac - b^2 > 0$. In this case, the graph of $R(s,t)$ is a paraboloid and the level sets $R(s,t) = L$ for constants L are ellipses. The segments are parallel when $ac - b^2 = 0$, in which case the graph of $R(s,t)$ is a parabolic cylinder and the level sets are lines. Define $\Delta = ac - b^2$.

In calculus terms, the goal is to minimize $R(s,t)$ over the unit square $[0,1]^2$. Because R is a continuously differentiable function, the minimum occurs either at an interior point of the square where the gradient $\nabla R = 2(as-bt+d, -bs+ct-e) = (0,0)$ or at a point on the boundary of the square. Define $F(s,t) = as-bt+d$ and $G(s,t) = -bs + ct - e$. The candidates for the minimum are the four corners $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$; the four edge points $(\hat{s}_0, 0)$, $(\hat{s}_1, 1)$, $(0, \hat{t}_0)$, and $(1, \hat{t}_1)$, where $F(\hat{s}_0, 0) = 0$, $F(\hat{s}_1, 1) = 0$, $G(0, \hat{t}_0) = 0$, and $G(1, \hat{t}_1) = 0$; and $(\bar{s}, \bar{t})$ when $\Delta > 0$, where $F(\bar{s}, \bar{t}) = 0$ and $G(\bar{s}, \bar{t}) = 0$. Some computations will show that $\hat{s}_0 = -d/a$, $\hat{s}_1 = (b-d)/a$, $\hat{t}_0 = e/c$, $\hat{t}_1 = (b+e)/c$, $\bar{s} = (be - cd)/\Delta$, and $\bar{t} = (ae - bd)/\Delta$.

A simple algorithm is to compute all 9 critical points, the last one only when $\Delta > 0$, evaluate R at those points, and select the one that produces the minimum squared distance. However, this is a slow algorithm. A smarter search for the correct critical point is called for.

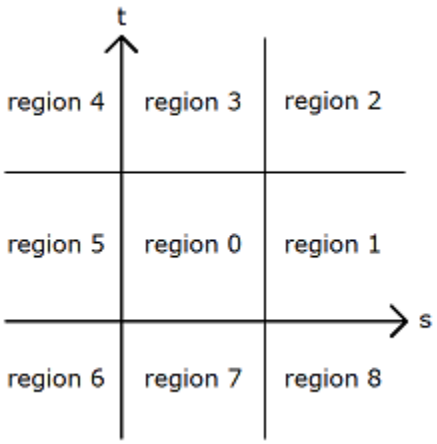
3 The Previous Version of the Critical Point Search

The implementation for segment-segment distance in the Wild Magic code is based on the following observations.

3.1 Nonparallel Segments

When $\Delta > 0$ the line segments are not parallel. The gradient of R is zero only when $\bar{s} = (be - cd)/\Delta$ and $\bar{t} = (bd - ae)/\Delta$. If $(\bar{s}, \bar{t}) \in [0,1]^2$, then we have found the minimum of R; otherwise, the minimum must occur on the boundary of the square. To find the correct boundary, consider Figure 1,

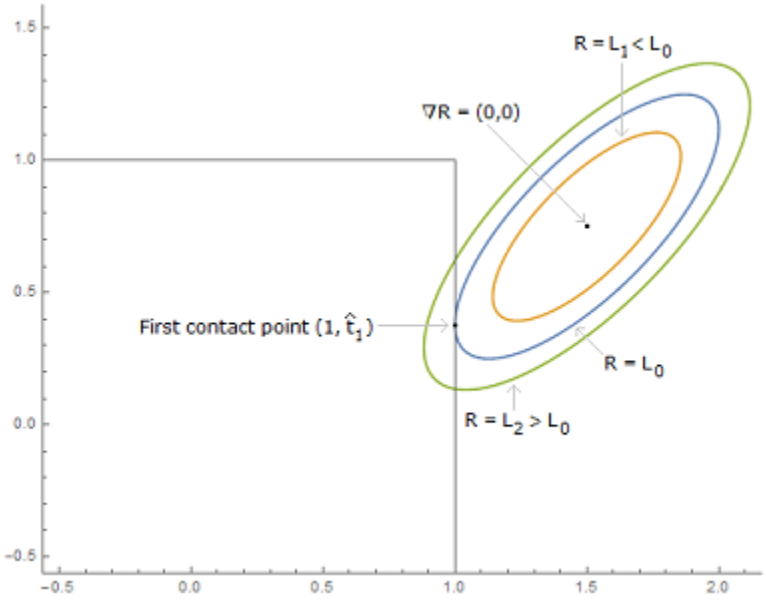
Figure 1 Partitioning of the st-plane by the unit square $[0,1]^2$.



The central square labeled region 0 is the domain of R, $(s,t) \in [0,1]^2$. If $(\bar{s}, \bar{t})$ is in region 0, then the two closest points on the line segments are interior points of those segments.

Suppose $(\bar{s}, \bar{t})$ is in region 1. The level curves of R are ellipses. At the point where $\nabla R = (0,0)$, the level curve degenerates to a single point $(\bar{s}, \bar{t})$. The global minimum of R occurs there, call it $L_{\min}$. As the level values L increase from $L_{\min}$, the corresponding ellipses are increasingly farther away from $(\bar{s}, \bar{t})$. There is a smallest level value L_0 for which the corresponding ellipse just touches the domain edge $s = 1$ at a value $\hat{t}_1 \in [0,1]$. For level values $L < L_0$, the corresponding ellipses do not intersect the domain. For level values $L > L_0$, portions of the domain lie inside the corresponding ellipses. In particular any points of intersection of such an ellipse with the edge must have a level value $L > L_0$. Therefore, $R(1,t) > R(1,\hat{t}_1)$ for $t \in [0,1]$ and $t \neq \hat{t}_1$. The point $(1,\hat{t}_1)$ provides the minimum squared-distance between two points on the line segments. The point on the first line segment is an endpoint and the point on the second line segment is interior to that segment. Figure 2 illustrates the idea by showing various level curves.

Figure 2 Various level curves $R(s,t) = L$. The plot is a modified output from `ContourPlot` using Mathematica 10.0; Wolfram Research, Inc.; Champaign, Illinois, 2014.



An alternate way of visualizing where the minimum distance point occurs on the boundary is to intersect the graph of R with the plane $s = 1$. The curve of intersection is a parabola and is the graph of $\phi(t) = R(1,t)$ for $t \in [0,1]$. Now the problem has been reduced by one dimension to minimizing a function $\phi(t)$ for $t \in [0,1]$. The minimum of $\phi(t)$ occurs either at an interior point of $[0,1]$, in which case $\phi'(t) = 0$ at that point, or at an endpoint $t = 0$ or $t = 1$. Figure 2 shows the case when the minimum occurs at an interior point. At that point the ellipse is tangent to the line $s = 1$. In the endpoint cases, the ellipse may just touch one of the corners of the domain but not necessarily tangentially.

To distinguish between the interior point and endpoint cases, the same partitioning idea applies in the one-dimensional case. The interval $[0,1]$ partitions the real line into three intervals, $t < 0$, $t \in [0,1]$, and $t > 1$. Let $\phi'(\tilde{t}) = 0$. If $\tilde{t} < 0$, then $\phi(t)$ is an increasing function for $t \in [0,1]$. The minimum restricted to $[0,1]$ must occur at $t = 0$, in which case R attains its minimum at $(s,t) = (1,0)$. If $\tilde{t} > 1$, then $\phi(t)$ is a decreasing function for $t \in [0,1]$. The minimum for ϕ occurs at $t = 1$ and the minimum for R occurs at $(s,t) = (1,1)$. Otherwise, $\tilde{t} = \hat{t}_1 \in [0,1]$, ϕ attains its minimum at $\hat{t}_1$ and R attains its minimum at $(s,t) = (1,\hat{t}_1)$.

The occurrence of $(\bar{s},\bar{t})$ in region 3, 5, or 7 is handled in the same way as when the global minimum is in region 0. If $(\bar{s},\bar{t})$ is in region 3, then the minimum occurs at $(\hat{s}_1, 1)$ for some $\hat{s}_1 \in [0,1]$. If $(\bar{s},\bar{t})$ is in region 5, then the minimum occurs at $(0,\hat{t}_0)$ for some $t \in [0,1]$. Finally, if $(\bar{s},\bar{t})$ is in region 7, then the minimum occurs at $(\hat{s}_0, 0)$ for some $\hat{s}_0 \in [0,1]$. Determining whether the first contact point is at an interior or endpoint of the appropriate interval is handled in the same way discussed earlier.

If $(\bar{s},\bar{t})$ is in region 2, it is possible the level curve of R that provides first contact with the domain touches either edge $s = 1$ or edge $t = 1$. Let $\nabla R = (R_s, R_t)$ where R_s and R_t are the first-order partial derivatives

of R. Because the global minimum occurs in region 2, it must be that the partial derivatives cannot both be positive. The choice is made for edge $s = 1$ or $t = 1$ based on the signs of $R_s(1,1)$ and $R_t(1,1)$. If $R_s(1,1) > 0$, then $R_t(1,1) \leq 0$. As you walk along the edge $s = 1$ towards $(1,1)$, $R(1,t)$ must decrease to $R(1,1)$. The directional derivative $(-1,0)\cdot(R_s(1,1), R_t(1,1)) = -R_s(1,1) < 0$, which means R must decrease from $R(1,1)$ as you walk along the edge $t = 1$ towards $(0,1)$. Therefore, the minimum must occur on the edge $t = 1$. Similarly, if $R_t(1,1) > 0$, then $R_s(1,1) \leq 0$. As you walk along the edge $t = 1$ towards $(1,1)$, $R(s,1)$ must decrease to $R(1,1)$. The directional derivative $(0,-1) \cdot (R_s(1,1), R_t(1,1)) = -R_t(1,1) < 0$, which means R must decrease from $R(1,1)$ as you walk along the edge $s = 1$ towards $(1,0)$. Therefore, the minimum must occur on the edge $s = 1$. In the final case that $R_s(1,1) \leq 0$ and $R_t(1,1) \leq 0$, the minimum occurs at $(1,1)$. Determining whether the minimum is interior to the edge or an endpoint is handled as in the case of region 1. The occurence of $(\bar{s}, \bar{t})$ in regions 4, 6, or 8 is handled similarly

3.2 Parallel Segments

When $\Delta = 0$, the segments are parallel and the gradient of R is zero on an entire st-line, $as - bt + d = 0$ for all real-valued t. If this line intersects the domain in a segment, then all points on that segment attain the minimum squared distance. Geometrically, this means that the projection of one segment onto the line containing the other segment is an entire interval of points. If the line $as - bt + 0$ does not intersect the domain, then only a pair of endpoints attains minimum squared distance, one endpoint per segment. In parameter space, the minimum is attained at a corner of the domain. In either case, it is sufficient to search the boundary of the domain for the critical point that leads to a minimum.

The quadratic factors to $R(s,t) = a(s - bt/a)^2 + 2d(s - bt/a) + f$, where $ac = b^2$, $e = bd/a$, and $b \neq 0$. R is constant along lines of the form $s - bt/a = k$ for constants k, and the minimum of R occurs on the line $as - bt + d = 0$. This line must intersect both the s-axis and the t-axis because a and b are not zero. Because of parallelism, the line is also represented by $-bs + ct - e = 0$.

3.3 A Nonrobust Implementation

The implementation of the algorithm was designed so that at most one floating point division occurs when computing the minimum distance and corresponding closest points. Moreover, the division was deferred until needed–in some cases no division was needed.

The logic for computing parameters for closest points on segments is shown in the next listing. The design goal was to minimize division operations for speed. The cost of divisions is not as big an issue as it was many years ago, unless you have an application that calls the distance query at real-time rates for a large number of segment pairs.

```
Compute a, b, c, d, e with dot products;
det = a*c - b*b;
if (det > 0) // nonparallel segments
{
    bte = b*e, ctd = c*d;
    if (bte <= ctd)  // s <= 0
    {
        if (e <= zero)  // t <= 0 (region 6)
        {
            s = (-d >= a ? 1 : (-d > 0 ? -d/a : 0));  t = 0;
        }
        else if (e < c) // 0 < t < 1 (region 5)
        {
            s = 0;  t = e/c;
        }
        else  // t >= 1 (region 4)
        {
            s = (b-d >= a ? 1 : (b-d > 0 ? (b-d)/a : 0));  t = 1;
        }
    }
    else  // s > 0
    {
        s = bte - ctd;
        if (s >= det)  // s >= 1
        {
            if (b+e <= 0)  // t <= 0 (region 8)
            {
                s = (-d <= 0 ? 0 : (-d < s ? -d/a : 1));  t = 0;
            }
            else if (b+e < c)  // 0 < t < 1 (region 1)
            {
                s = 1;  t = (b+e)/c;
            }
            else  // t >= 1 (region 2)
            {
                s = (b-d <= 0 ? 0 : (b-d < a ? (b-d)/a : 1));  t = 1;
            }
        }
        else  // 0 < s < 1
        {
            ate = a*e, btd = b*d;
            if (ate <= btd)  // t <= 0 (region 7)
            {
                s = (-d <= 0 ? 0 : (-d >= a ? 1 : -d/a));  t = 0;
            }
            else  // t > 0
            {
                t = ate - btd;
                if (t >= det)  // t >= 1 (region 3)
                {
                    s = (b-d <= 0 ? 0 : (b-d >= a ? 1 : (b-d)/a));  t = 1;
                }
                else  // 0 < t < 1 (region 0)
                {
                    s /= det;
                    t /= det;
                }
            }
        }
    }
}
else // parallel segments
{
    if (e <= 0)
    {
        s = (-d <= 0 ? 0 : (-d >= a ? 1 : -d/a));  t = 0;
    }
    else if (e >= c)
    {
        s = (b-d <= 0 ? 0 : (b-d >= a ? 1 : (b-d)/a));  t = 1;
    }
    else
```

```
    {
        s = 0;  t = e/c;
    }
}
```

The code block for region 0 is shown in the listing. To arrive at this block, the presumption is that the minimum of $R(s,t)$ occurs at an interior point of the domain. The divisions by Δ were deferred until actually needed. The code conforms to the mathematics of the problem, but unfortunately when computing with floating-point arithmetic, the (s,t) value might not be accurate due to rounding errors in the division. Two problems can occur in the division. Firstly, the numerators and denominator are both small, which can lead to inaccurate floating-point results. Example 1 illustrates this using double-precision numbers.

Example 1 An example where the minimum point is interior but some numerical rounding errors lead to an inaccurate result.

```
// The input segments.
P0 = (-1.0264718499965966,   9.6163341007195407e-007, 0.0)
P1 = ( 0.91950808032415809, -1.0094441192690283e-006, 0.0)
Q0 = (-1.0629447383806110,   9.2709540082141753e-007, 0.0)
Q1 = ( 1.0811583868227901,  -1.0670017179567367e-006, 0.0)

// Computations that lead to region 0.
a = 3.7868378892150543
b = 4.1723816501877566
c = 4.5971782115109665
d = 0.070975508796052952
e = 0.078201633969291237
det = 1.2079226507921703e-013
s = 5.0737192225369654e-014  // numerator before division by det
t = 4.8072656966269278e-014  // numerator before division by det
s = 0.42003676470588236      // after division by det
t = 0.39797794117647056      // after division by det

// The distance |[(1-s)*P0 + s * P1] - [(1-t)*Q0 + t*Q1]|.
distance = 0.00055025506003679664

// The parameters using exact rational arithmetic with conversion back to
// double precision at the end (the conversion can lose some precision).
sTrue = 0.42457281934252261
tTrue = 0.40235148377129676

// The true distance is zero (no precision loss on conversion to double).
distanceTrue = 0
```

⋈

Secondly, subtractive cancellation occurs when computing the numerators in the first place, leading to a misclassification; that is, the numerical computations lead you to region 0 but theoretically the minimum occurs on the domain boundary. Example 2 illustrates this.

Example 2 An example where the minimum point is classified as interior but the theoretically correct point is on the domain boundary.

```
// The input segments.
P0 = (-1.0896217473782599,   9.7236145595088601e-007, 0.0)
P1 = ( 0.91220578597858548, -9.4369829432107506e-007, 0.0)
Q0 = (-0.90010447502136237,  9.0671446351334441e-007, 0.0)
Q1 = ( 1.0730877178721130,  -9.8185787633992740e-007, 0.0)

// Computations that lead to region 0.
a = 4.0073134733092228
b = 3.9499904603425491
```

```
c = 3.8934874300993294
d = -0.37938089385085144
e = -0.37395400223322062
det = 1.7763568394002505e-015
s = 2.2204460492503131e-016  // numerator before division by det
t = 4.4408920985006262e-016  // numerator before division by det
s = 0.125    // after division by det
t = 0.25     // after division by det

// The distance |[(1-s)*P0 + s * P1] - [(1-t)*Q0 + t*Q1]|.
distance = 0.43258687891076358
```

The divisions are accurate but the numerators are not because of subtractive cancellation. The algorithm using exact arithmetic produces

```
sTrue = 0.094672127942504153
tTrue = 0
distanceTrue = 1.1575046138574105e-007
```

The exact value for the s-numerator `b*e - c*d` is of the form $1.u*2^{-54}$, where u has 197 bits. The conversion to double produces `-3.6091745045569584e-017`, so in fact s is negative and the algorithm never gets into the region-0 block. The error due to the misclassification is enormous! $\bowtie$

4 Sunday's Implementation

Many years ago, Dan Sunday contacted me about my implementation and said he had a more efficient one; I do not recall whether we communicated by email or Usenet newsgroup. I never investigated, keeping what I had because no one was reporting bugs and the code appeared to be sufficiently fast. Recently, someone contacted me and said he was using Dan's online code for a physical simulation and was having robustness problems. That code can be found at Distance between 3D Lines and Segments. The function in question is `dist3D_Segment_to_Segment`.

Indeed the code appears to be shorter in length. As with other 2D problems that can be factored into two 1D problems, the algorithm he uses has that flavor. He uses a threshold on the determinant in order to classify whether segments are parallel or not, something I had tried in my implementation initially. After enough bug reports of failed segment-segment distance queries, I removed the threshold. For nonparallel segments, Dan's idea is to find the location of the root of the s-derivative first. Then he locates the root of the t-derivative but in some cases has to recompute the s-derivative root.

The threshold `SMALL_NUM` is `1e-08` and is used to decide whether or not the segments are nonparallel. If parallel, the following comment occurs in the code where s is set to zero "force using point P0 on segment S1". It is not clear why one should start here, and in fact I will present a couple of examples where this leads to inaccurate results. The threshold is also used on the numerators for `sc` and `tc`. No comment is made why the test is applied to numerators. Typically, a threshold in this context is applied to a denominator to avoid a division by zero. Perhaps in his experiments with his implementation, he ran into numerical problems when the numerators were small as well as the denominators.

In comparing my implementation to Dan's, I kept track of the input that led to the biggest difference in our results.

Example 3 An example where forcing the parallel case to start the search at $s = 0$ fails.

```
// The input segments.
```

```
P0 = ( 0.77998990099877119, 0.61192502360790968,  -0.22703111823648214)
P1 = ( 0.53215344529598951, 0.85724585503339767,  -0.10102437809109688)
Q0 = (-0.21277333982288837, 0.35091548087075353,  -0.49557160679250956)
Q1 = ( 0.11881479667499661, 0.022494725417345762, -0.66426620958372951)

// Computations that lead to region 0.
a = 0.13748291766867621
b = -0.18400473826578512
c = 0.24626875388961456
d = -0.14817393336298745
e = 0.19816623075105058
D = 3.1111132767214222e-009  // the 'det', go to parallel case
sN = 0
sD = 1
tN = 0.19816623075105058
tD = 0.24626875388961456

// tN >= 0 and tN <= tD, so jump to computing sc (s) and tc 9t)
s = 0
t = 0.80467468008497312
distance = 0.98303584847514447

// Using exact arithmetic with the (nonrobust) algorithm.  The algorithm
// is, of course, robust when using exact arithmetic.
sTrue = 1
tTrue = 0.057504219522762176
distanceTrue = 0.98292397116488739
```

The difference between estimated distance and true distance is approximately $1e-04$. Although that might not seem significant, it was for the person running the physical simulation. Notice that the true closest parameters occur for $s = 1$, not for $s = 0$. Whether or not this makes a difference depends on the actual closest points. $\bowtie$

Another example is shown next.

Example 4 An example of two segments that intersect, so their closest points are the same and the distance is zero. The first segment has endpoints $\mathbf{P}_0 = (0, 0, 0)$ and $\mathbf{P}_1 = (1, 0, 0)$. The second segment has endpoints $\mathbf{Q}_0 = (-\varepsilon, \phi + \delta, 0)$ and $\mathbf{Q}_1 = (\varepsilon, \phi - \delta, 0)$. The idea is to choose offset $\phi > 0$ close to zero, $\delta > 0$ close to zero so that the segments are nearly parallel, and $\varepsilon > 0$ that leads to generating the incorrect (s, t) for the closest points.

```
// The input segments.
double delta = 0.25 * 1e-04;
double epsilon = sqrt(delta);
double phi = 1e-05;
P0 = (0, 0, 0)
P1 = (1, 0, 0)
Q0 = (-epsilon, phi + delta, 0)
Q1 = (+epsilon, phi - delta, 0)

// Computations that lead to region 0.
a = 1.0
b = 0.01
c = 0.00010000250000000000
d = 0.0050000000000000001
e = 5.0001750000000003e-005
D = 2.4999999999987055e-009  // the 'det', go to parallel case
sN = 0
sD = 1
tN = 5.0001750000000003e-005
tD = 0.00010000250000000000

// tN >= 0 and tN <= tD, so jump to computing sc (s) and tc 9t)
s = 0
t = 0.50000499987500313
```

```
distance = 9.9998750023437051e-006

// Using exact arithmetic with the (nonrobust) algorithm.  The algorithm
// is, of course, robust when using exact arithmetic.  This call went to
// the region-0 case because the segments intersect.
sTrue = 0.0020000000000000000
tTrue = 0.70000000000000007
distanceTrue = 0.0
```

⋈

The performance difference between the nonrobust code and Sunday's code was negligible.

5 A Robust Implementation

The primary failure of the nonrobust implementation is when the computation of numerator $be - cd$ for s or numerator $ae - bd$ for t has subtractive cancellation. The division by a nearly zero Δ amplifies the error so that the computed (s, t) is a really bad estimate of the location of the minimum. It is not tractable to try refining the estimate because you have no idea in which direction to search for a better estimate.

5.1 Conjugate Gradient Method

Instead, we may use a *constrained conjugate gradient* approach, one that is robust numerically. To motivate this, note that the quadratic function to minimize is

$$R(s,t) = \begin{bmatrix} s & t \end{bmatrix} \begin{bmatrix} a & -b \\ -b & c \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} + 2 \begin{bmatrix} d & -e \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} + f = \mathbf{p}^{\mathsf{T}} M \mathbf{p} + 2\mathbf{K}^t \mathbf{p} + f$$

where $\mathbf{p}$ is the 2-tuple of s and t, and M and $\mathbf{K}$ are clear from context. The minimum of R occurs when its gradient is the zero vector,

$$\nabla R = 2M\mathbf{p} + 2\mathbf{K} = \mathbf{0}$$

The solution is

$$\mathbf{p} = -M^{-1}\mathbf{K} = \frac{-1}{ac - b^2} \begin{bmatrix} c & b \\ b & a \end{bmatrix} \begin{bmatrix} d \\ -e \end{bmatrix} = \frac{1}{ac - b^2} \begin{bmatrix} be - cd \\ ae - bd \end{bmatrix}$$

This solution is what we are trying to compute numerically.

With no constraints on $\mathbf{p}$, the conjugate gradient method for minimization may be used to compute the minimum of R in two steps. The first step starts at any point $\mathbf{p}_0$ and computes the minimum of R along the line $\mathbf{p}_0 + u(1, 0)$. Let that point be named $\mathbf{p}_1$. The second step starts at the point and computes the minimum of R along the line $\mathbf{p}_1 + v\mathbf{D}$, where $\mathbf{D} = (D_0, D_1)$ is a *conjugate direction* corresponding to $(1, 0)$ relative to the matrix M. Such a direction has the property

$$0 = (1, 0) \cdot M\mathbf{D} = aD_0 - bD_1$$

One such choice is $\mathbf{D} = (b, a)$. With constraints, the second step requires slightly more attention.

5.2 Constrained Conjugate Gradient Method

First, compute the parameter point $(\hat{s}_0, 0)$ that minimizes R along the line $t = 0$. The solution is $\hat{s}_0 = -d/a$. Second, compute the parameter point $(\hat{s}_1, 1)$ that minimizes R along the line $t = 1$. The solution is $\hat{s}_1 = (b - d)/a$. The difference of these points is

$$(\hat{s}_1, 1) - (\hat{s}_0, 0) = (b/a, 1)$$

which happens to be a conjugate direction for $(1, 0)$. We may compute the intersection of the parameter domain and the line through these two points. If there is no intersection or the intersection is a single point, the minimum of R occurs at a corner of the domain. If there is a segment of intersection, the minimum of R must occur on that segment.

When there is a segment of intersection, let its endpoints be $\mathbf{E}_0$ and $\mathbf{E}_1$. Define $\phi(z) = R((1 - z)\mathbf{E}_0 + z\mathbf{E}_1)$ for $z \in [0, 1]$. The minimization is now a 1D problem. The minimum must occur when $H(z) := \phi'(z) = 0$ or at an endpoint of the interval ($z = 0$ or $z = 1$). The function $H(z)$ is linear, so we have a simple equation to solve when $H(0)$ and $H(1)$ have opposite signs. If either of $H(0)$ and $H(1)$ is zero, then the corresponding (s, t) point is the location of where the gradient of R is $(0, 0)$. If $H(0)$ and $H(1)$ are both positive or both negative, the minimum of H occurs at than endpoint corresponding to the minimum absolute value of those derivatives.

The beauty of the approach is that when $H(z) = 0$ has a solution for $z \in (0, 1)$, then that solution is $\bar{z} = H(0)/(H(0) - H(1))$. If $H(0)$ is nearly zero and suffers from subtractive cancellation, then $H(0) - H(1)$ has the same sign and is nearly zero. If the numerical errors cause the computed $\bar{z}$ to be outside $[0, 1]$, simply clamp the result to the interval. $H(z)$ is nearly zero on the entire segment, so $\phi(z)$ (R on the segment) is nearly constant. The clamping will not significantly affect the accuracy of the result. Because we are in a 1D situation, the subtractive cancellation cannot steer us away from the line segment that contains the minimum. This is much better than in the 2D situation where the subtractive cancellation moves you away from the true minimum in an unknown direction.

5.3 An Implementation

The implementation is provided in the file `GteDistSegmentSegment`. In addition to the robustness for two nondegenerate segments, code was added to handle degenerate segments. Those cases are easy to understand, so let us focus on the nondegenerate segments. Consider the code shown in the next listing.

```
Compute a, b, c, d, e;
f00 = d, f10 = d + a, f01 = d - b, f11 = d + a - b;  // dR/ds at corners
g00 = -e, g10 = -e - b, g01 = -e + c, g11 = -e -b + c;  // dR/dt at corners
hatS[0] = GetClampedRoot(a, f00, f10);
hatS[1] = GetClampedRoot(a, f01, f11);
classify[0] = (hatS[0] <= 0 ? -1 : (hatS[0] >= 1 ? 1 : 0));
classify[1] = (hatS[1] <= 0 ? -1 : (hatS[1] >= 1 ? 1 : 0));
if (classify[0] == -1 && classify[1] == -1)  // minimum occurs on s = 0 for 0 <= t <= 1
{
    parameter[0] = 0;
    parameter[1] = GetClampedRoot(c, g00, g01);
}
else if (classify[0] == +1 && classify[1] == +1)  // minimum occurs on s = 1 for 0 <= t <= 1
{
    parameter[0] = 1;
    parameter[1] = GetClampedRoot(c, g10, g11);
}
else
{
```

```
    // The line dR/ds = 0 intersects domain [0,1]^2 in a nondegenerate segment.
    // The edge[i] flags stores which domain edge contains the endpoint end[i].
    int edge[2];
    Real end[2][2];
    ComputeIntersection(hatS, classify, end);

    // Analyze the function H(z) = dR/dt((1-z)*end[0] + z*end[1]) for z in [0,1].
    ComputeMinimumParameters(hatS, edge, end, parameter);
}
```

The function `GetClampedRoot` returns the root of the linear function $h(z) = h_0 + \sigma z$ on the interval $[0, 1]$, or if the root is outside the interval, it is clamped to the interval. It is required that h_0 and $h_1 = h_0 + \sigma$ have opposite signs, in which case there is a root on the real line.

```
Real GetClampedRoot(Real sigma, Real h0, Real h1)
{
    if (h0 >= 0) { return 0; }
    if (h1 <= 0) { return 1; }
    Real root = -h0 / sigma;
    if (root > 1) { root = 0.5; }
    return root;
}
```

The code has a guard against the numerical division generating a number larger than one (it cannot be negative). In this case, h_0 and h_1 are both nearly zero, so choosing a root of 0.5 does not significantly affect the accuracy of downstream computations. If you want an ultimate guard, replace the assignment to 0.5 by a (slower) bisection routine applied to $h(z)$.

We can intersect the domain and the segments with endpoints $(\hat{s}_0, 0)$ and $(\hat{s}_1, 1)$. However, $\hat{s}_i$ might be unreliable numerically when either of a or c is nearly zero and the $\hat{s}_i$ values are large-magnitude floating-point numbers. To avoid this numerical problem, the function `ComputeIntersection` computes the intersection points directly on the domain edges. The `classify[]` values allows us to determine which domain edges are intersected. For example, if `classify[0]` is -1 and `classify[1]` is 0, then the initial segment must intersect the domain edge $s = 0$ for some $t \in [0, 1]$. The computation of t itself involves a division by b, but to be in this situation we know that $b \neq 0$. It can be close to zero, but then so is the numerator. We can use the same idea as in `GetClampedRoot`–test the numerical result of the division and, if outside $[0, 1]$, use 0.5 as the value. Again, for an ultimate guard you can use bisection applied to a linear function.

Once we have the endpoints, we must evaluate $H(z)$ at those points, the job of `GetMinimumParameters`. If $H(0)$ is zero, the function returns the (s, t) parameters at the corresponding endpoint. If $H(1)$ is zero, the function returns the (s, t) parameters at the corresponding endpoint. If $H(0)$ and $H(1)$ have opposite signs, then we solve $H(z)$ for its unique root. Once again, we test the numerical result of a division and choose 0.5 if that result is outside $[0, 1]$. The final cases are when $H(0)$ and $H(1)$ have the same sign. The endpoint with minimum H-value is not the (s, t) that minimizes the function. We need to apply `GetClampedRoot` on the edge containing that endpoint, which is why we compute the `edge[]` flags.

The examples discussed previously were also tested using the robust algorithm.

```
// Example 3.1
// nonrobust algorithm
s = 0.42003676470588236
t = 0.39797794117647056
distance = 0.00055025506003679664
// exact arithmetic
sTrue = 0.42457281934252261
tTrue = 0.40235148377129676
distanceTrue = 0.0
// robust algorithm
sRobust = 0.41853798375537504
tRobust = 0.39687428969544680
```

```
distanceRobust = 9.7307189345304538e-010

// Example 3.2
// nonrobust
s = 0.125
t = 0.25
distance = 0.43258687891076358
// exact arithmetic
sTrue = 0.094672127942504153
tTrue = 0.0
distanceTrue = 1.1575046138574105e-007
// robust algorithm
sRobust = 1.0
tRobust = 0.91846616235720013
distanceRobust = 1.1575046138574101e-007

// Example 4.1
// Sunday's implementation
s = 0.0
t = 0.80467468008497312
distance = 0.98303584847514447
// exact arithmetic
sTrue = 1.0
tTrue = 0.057504219522762176
distanceTrue = 0.98292397116488739
// robust algorithm
sRobust = 1.0
tRobust = 0.057504219522762086
distanceRobust = 0.98292397116488739

// Example 4.2
// Sunday's implementation
s = 0
t = 0.50000499987500313
distance = 9.9998750023437051e-006
// exact arithmetic
sTrue = 0.0020000000000000000
tTrue = 0.70000000000000007
distanceTrue = 0.0
// robust algorithm
sRobust = 0.0020000000000054211
tRobust = 0.70000000000054219
distanceRobust = 2.7122314947662727e-017
```

In Example 2, $\hat{s}_0 = 0.094672127942504153$, $\hat{s}_1 > 1$, so we compute the t-intersection on $s = 1$, which is $\tilde{t} = 0.91846616235720013$. The endpoints of the segment defining $H(z)$ are $\mathbf{E}_0 = (\hat{s}_0, 0)$ and $\mathbf{E}_1 = (1, \tilde{t})$. The values of H are $H(0) = -5.5511151231257827e - 017$ and $H(1) = 0$. There is numerical error in these computations which lead to selection of $\mathbf{E}_1$ rather than $\mathbf{E}_0$ as the minimum point. However, R does not vary much on the segment, so there is not much error in the distance computation. What this does say, though, is you can expect reasonable continuity in the distance computations, but you cannot expect closest points to have continuity. Even theoretically, the continuity of closest points is not guaranteed as you vary the segments from nonparallel to nonparallel, passing through a parallel configuration along the way.

6 Sample Application and an Implementation on a GPU

The sample application

```
GeometricTools/GTEngine/Samples/Geometrics/DistanceSegments3
```

has a GPU implementation of the robust algorithm and uses double-precision numbers. The port from CPU to GPU is straightforward. However, HLSL does not have a double-precision `sqrt` function, so the squared distance is computed and read back. It is possible to roll your own minimax approximation (see my GPGPU book) for HLSL.

The sample application has tests for accuracy and performance. On the performance end, a set of 16384 segments is run through an all-pairs distance computation, which is approximately 134 million queries. The CPU code runs on a single thread. On an Intel i7–3990K CPU running at 3.2GHz, the code runs in approximately 9.18 seconds. On an AMD 7970 GPU that is not overclocked, the code runs in approximately 2.85 seconds when closest points are computed and read back from the GPU. A large part of this time is for the read back, not for the computations. If the HLSL file is configured to read back only the squared distances and (s, t) values, the run time is approximately 1.04 seconds.