Moving Along a Curve with Specified Speed

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

Created: April 28, 2007
Last Modified: August 20, 2023

Contents

1 Introduction

A frequently asked question in computer graphics is how to move an object (or camera eyepoint) along a parameterized curve with constant speed. The method to do this requires the curve to be *reparameterized by arclength*. A more general problem is to specify the speed of the object at every point of the curve. This also requires the concept of reparameterization of the curve. I cover both topics in this document, including discussion of the theory, implementations of the numerical algorithms and performance issues. The concepts apply to curves in any dimension. The curves may be defined as a collection of contiguous curves such as piecewise Bézier curves.

2 Basic Theory of Curves

Consider a parametric curve, $\boldsymbol{X}(t)$, for $t \in [t_{\min}, t_{\max}]$. The variable t is referred to as the curve parameter. The curve may be thought of as the path of a particle whose position is $\boldsymbol{X}(t)$ at time t. The velocity $\boldsymbol{V}(t)$ of the particle is the rate of change of position with respect to time, a quantity measured by the derivative

$$\boldsymbol{V}(t) = \frac{d\boldsymbol{X}}{dt} \tag{1}$$

The velocity vector $\boldsymbol{V}(t)$ is tangent to the curve at the position $\boldsymbol{X}(t)$. The speed $\sigma(t)$ of the particle is the length of the velocity vector

$$\sigma(t) = |\boldsymbol{V}(t)| = \left|\frac{d\boldsymbol{X}}{dt}\right| \tag{2}$$

The requirement that the speed of the particle be constant for all time is stated mathematically as $\sigma(t) = c$ for all t, where c is a specified positive constant. The particle is said to travel with *constant speed* along the curve. If $c = 1$, the particle is said to travel with *unit speed* along the curve. When the velocity vector has unit length for all time, the curve is said to be *parameterized by arclength*. In this case, the curve parameter is typically named s, and the parameter is referred to as the *arclength parameter*. The value s is a measure of distance along the curve. The arclength parameterization of the curve is written as $\boldsymbol{X}(s)$, where $s \in [0, L]$ and L is the total length of the curve.

For example, a circular path in the plane that has center $(0, 0)$ and radius 1 is parameterized by $\boldsymbol{X}(s) = (\cos(s), \sin(s))$ for $s \in [0, 2\pi)$. The velocity is $\boldsymbol{V}(s) = (-\sin(s), \cos(s))$ and the speed is $\sigma(s) = |d\boldsymbol{V}/ds| = |(-\sin(s), \cos(s))| = 1$, which is constant for all s. The particle travels with unit speed around the circle, so $\boldsymbol{X}(s)$ is an arclength parameterization of the circle.

A different parameterization of the circle is $\boldsymbol{Y}(t) = (\cos(t^2), \sin(t^2))$ for $t \in [0, \sqrt{2\pi})$. The velocity is $\boldsymbol{V}(t) = (-2t\sin(t^2), 2t\cos(t^2))$ and the speed is $\sigma(t) = |d\boldsymbol{V}/dt| = \left|(-2t\sin(t^2), 2t\cos(t^2)\right| = 2t$. The speed increases with t, so the particle does not move with constant speed around the circle.

Suppose you were given $\boldsymbol{Y}(t)$, the parameterization of the circle for which the particle does not travel with constant speed, and you want to change the parameterization so that the particle does travel with constant speed. That is, you want to relate the parameter t to the arclength parameter s, say by a function $t = f(s)$, so that $\boldsymbol{X}(s) = \boldsymbol{Y}(t)$. In the circle example, $t = \sqrt{s}$. The process of determining the relationship $t = f(s)$ is referred to as *reparameterization by arclength*. How do you actually find this relationship between t and s? Generally, there is no closed-form expression for the function f. Numerical methods must be used to compute t for each specified s.

3 Reparameterization by Arclength

Let $\boldsymbol{X}(s)$ be an arclength parameterization of a curve. Let $\boldsymbol{Y}(t)$ be another parameterization of the same curve, in which case $\boldsymbol{Y}(t) = \boldsymbol{X}(s)$ implies a relationship between t and s. We may apply the chain rule from Calculus to obtain

$$\frac{d\boldsymbol{Y}}{dt} = \frac{d\boldsymbol{X}}{ds}\frac{ds}{dt} \tag{3}$$

Computing lengths and using the convention that speeds are nonnegative, we have

$$\left|\frac{d\boldsymbol{Y}}{dt}\right| = \left|\frac{d\boldsymbol{X}}{ds}\frac{ds}{dt}\right| = \left|\frac{d\boldsymbol{X}}{ds}\right|\left|\frac{ds}{dt}\right| = \frac{ds}{dt} \tag{4}$$

where the right-most equality is true because $|d\boldsymbol{X}/ds| = 1$.

As expected, the speed of traversal along the curve is the length of the velocity vector. This equation tells you how speed ds/dt depends on time t. To obtain a relationship between s and t, you have to integrate the speed to obtain s as a function $g(t)$ of time

$$s = g(t) = \int_{t_{\min}}^{t} \left|\boldsymbol{Y}'(\tau)\right| \, d\tau \tag{5}$$

where $\boldsymbol{Y}'(t) = d\boldsymbol{Y}(t)/dt$. It is common practice not to use the same variable name for both the variable of integration and a limit of integration, which is why I chose τ as the variable of integration. The derivative function has two names, so to speak. The first name is $d\boldsymbol{Y}/dt$ and the second name is $\boldsymbol{Y}'$. The integrand of equation (5) uses the second name. If instead the first name is used, there appears to be ambiguity in that the upper limit of integration is t, the first name of the derivative has a t in the denominator, but the variable of integration is τ. To avoid the ambiguity having t in the first name of the derivative function, I chose to use the second name.

When time $t = t_{\min}$, the arclength is $s = g(t_{\min}) = 0$. This makes sense because the particle has not yet moved—the distance traveled is zero. When time $t = t_{\max}$, the arclength is $s = g(t_{\max}) = L$, which is the total distance L traveled along the curve. Given the time t, we can determine the corresponding arclength s from the integration. However, what is needed is the inverse problem. Given an arclength s, we want to know the time t at which this arclength occurs; that is, you first decide the distance the object should move along the curve and then determine the time t at which that distance occurs. The position on the curve at an arclength s is $\boldsymbol{Y}(t)$. To compute t, we must invert the function g to obtain

$$t = g^{-1}(s) \tag{6}$$

Inverting g in terms of elementary functions is usually not possible, so we have to rely on numerical methods to compute $t \in [t_{\min}, t_{\max}]$ given a value of $s \in [0, L]$.

3.1 Numerical Solution: Inverting an Integral Using Bisection

The simplest but conservative approach to inverting the integral is to use bisection for $F(t) = g(t) - s$ where $g(t)$ is the function of equation (5). Listing 1 contains code for the algorithm.

Listing 1. Pseudocode for inverting the integral of equation (5) to obtain the time t for a specified arclength s. The algorithm uses bisection. The type Real is float or double. The functions Y, DYDT, Speed, and Arclength are provided by a curve class.

```
class Curve
{
public:
    // The position is Y(t).
    Vector Y(Real t) { function body; }

    // The velocity is dY(t)/dt = Y'(t).
    Vector DYDT(Real t) { function body; }

    // The speed is |Y'(t)|.
    Real Speed(Real t) { return Length(DYDT(t)); }

    // The arclength is s = ∫_{tmin}^{t} |Y'(τ)| dτ. Function Integral is a numerical integrator for the speed function over [tmin, t].
    Real Arclength(Real t0, Real t1) { return Integral(t0, t1, Speed()); }

    // The curve domain is [tMin,tMax]. The curve length is totalArclength = Arclength(tMin,tMax).
    Real tMin, tMax, totalArclength;
}

Real F(Curve curve, Real t, Real s)
{
    return curve.Arclength(curve.tMin, t) − s;
}

bool BisectionConverged(Real tMin, Real tMax, Real s, Real& tMid, Real& fMid)
{
    if (tMid == tMin || tMid == tMax)
    {
        // The precision of type Real is such that tMin and tMax are consecutive floating-point numbers. Their average cannot
        // be a floating-point number strictly between them. This is the best you can do using type Real. Return the t-endpoint
        // whose f-value has smaller magnitude.
        Real fMin = F(tMin, s), fMax = F(tMax, s);
        if (fMin <= fMax)
        {
            tMid = tMin;
            fMid = fMin;
        }
        else
        {
            tMid = tMax;
            fMid = fMax;
        }
        return true;
    }
    return false;
}

struct GCPOutput
{
    GCPOutput() : t(0), f(0), numIntersections(0) {}
    GCPOutput(Real inT, Real inF, int inNumIntersections) : t(inT), f(inF), numIntersections(numIntersections) {}
    Real t, f;
    int numIntersections;
}
```

```
GCPOutput GetTParameter(Curve curve, Real s)
{
    // Clamp the input to the valid interval of lengths [0, L].
    if (s <= 0) { return GCPOutput(curve.tMin, 0, 0); }
    if (s >= curve.totalArclength) { return GCPOutput(curve.tMax, 0, 0); }

    // Compute a t-root of F(t, s) for the specified s-value. We know that F(t_min) < 0 and F(t_max) > 0. Rather than
    // interval [t_min, t_max], choose a subinterval using an initial use the initial guess for the t-root.
    Real tMin = curve.tMin, tMax = curve.tMax;
    Real tMid = tMin + (tMax − tMin) * (s / curve.totalArclength), fMid = F(tMid, s);
    if (fMid > 0) { tMax = tMid; } else { tMin = tMid; }

    // Choose maxIterations sufficiently large for convergence. The value 4096 is sufficient. In practice, the number
    // of iterations for float is no larger than approximately 24 and for double is no larger than approximately 53.
    int maxIterations = 4096;
    int numIterations{};
    for (numIterations = 1; numIterations < maxIterations; ++numIterations)
    {
        // Compute the t-midpoint and the corresponding f-value. Exit early if the f-value is zero.
        tMid = (tMin + tMax) / 2;
        fMid = F(tMid, s);
        if (fMid == 0)
        {
            break;
        }

        // Convergence occurs when tMid is tMin or tMax.
        if (BisectionConverged(tMin, tMax, s, tMid, fMid))
        {
            break;
        }

        // Update the correct t-endpoint using the t-midpoint.
        if (fMid > 0)
        {
            tMax = tMid;
        }
        else
        {
            tMin = tMid;
        }
    }

    return GCPOutput(tMid, fMid, numInterations);
}
```

3.2 Numerical Solution: Inverting an Integral Using Newton's Method

Define $F(t) = g(t) - s$. The t-derivative is $F'(t) = g'(t) = |\boldsymbol{Y}'(t)|$. Given a value s, the problem is to find a value t so that $F(t) = 0$. This is a root-finding problem that can be solved using Newton's method. If $t_0 \in [t_{\min}, t_{\max}]$ is an initial guess for t, Newton's method produces a sequence

$$t_{i+1} = t_i - \frac{F(t_i)}{F'(t_i)} = t_i - \frac{g(t_i) - s}{|\boldsymbol{Y}'(t)|}, \quad i \ge 0 \tag{7}$$

The iterate t_1 is determined by s, t_0, $g(t_0)$, and $|\boldsymbol{Y}'(t_0)|$. Evaluating $|\boldsymbol{Y}'(t_0)|$ is straightforward because we already have a formula for $\boldsymbol{Y}(t)$ and can compute $\boldsymbol{Y}'(t)$ from it. Evaluating $g(t_0)$ requires an integration that can be approximated using standard numerical integrators. A reasonable choice for the initial iterate is

$$t_0 = t_{\min} + \frac{s}{L}\ (t_{\max} - t_{\min}) \tag{8}$$

where the value s/L is the fraction of total arclength at which the particle should be located. The subsequent iterates are computed until either $F(t_i)$ is sufficiently close to zero or until a maximum number of iterates has been computed.

There is a potential problem when using Newton's method. The derivative $F(t)$ is a nondecreasing function because its derivative $F'(t) = |d\boldsymbol{Y}/dt|$ is nonnegative. The second derivative is $F''(t)$. If $F''(t) \geq 0$ for all $t \in [t_{\min}, t_{\max}]$, then the function is said to be *convex* and the Newton iterates are guaranteed to converge to the root. However, $F''(t)$ can be negative, which might lead to Newton iterates outside the domain $[t_{\min}, t_{\max}]$. To avoid this problem, a hybrid of Newton's method and bisection can be used. A root-bounding interval is maintained along with the iterates. A candidate Newton's iterate is computed. If it is inside the current root-bounding interval, it is accepted as the next root estimate. If it is outside the interval, the midpoint of the interval is used instead. Regardless of whether a Newton iterate or bisection is used, the root-bounding interval is updated and the interval length strictly decreases over time. Listing 2 contains code for the hybrid algorithm.

Listing 2. Pseudocode for inverting the integral of equation (5) to obtain the time t for a specified arclength s. The algorithm is a hybrid of Newton's method and bisection. The type Real is float or double. Some of the functions referenced here were defined in listing 1.

```
GCPOutput GetTParameter(Curve curve, Real s) const
{
    // Clamp the input to the valid interval of lengths [0, L].
    if (s <= 0) { return GCPOutput(curve.tMin, 0, 0); }
    if (s >= curve.totalArclength) { return GCPOutput(curve.tMax, 0, 0); }

    // Compute a t-root of F(t, s) for the specified s-value. We know that F(tmin) < 0 and F(tmax) > 0. Rather than
    // interval [tmin, tmax], choose a subinterval using an initial use the initial guess for the t-root.
    Real tMin = curve.tMin, tMax = curve.tMax;
    Real tMid = tMin + (tMax - tMin) * (s / curve.totalArclength), fMid = F(tMid, s);
    if (fMid > 0) { tMax = tMid; } else { tMin = tMid; }

    // Store the iterates from Newton's method in order to determine whether a cycle has occurs. If it does, further iterates will
    // already be in the set, so the function should return when a cycle is detected.
    set tIterates{};

    // Choose numIterations sufficiently large for convergence. The value 4096 is sufficient. In practice, the number
    // of iterations for float is no larger than approximately 24 and for double is no larger than approximately 53.
    int maxIterations = 4096;
    int numIterations{};
    for (numIterations = 1; numIterations < maxIterations; ++numIterations)
    {
        // Test whether tMid is an iterate visited previously. If so, a cycle has occurred.
        if (tIterates.insert(tMid).second == false) { break; }

        // Compute the t-midpoint and the corresponding f-value. Exit early if the f-value is zero.
        fMid = F(tMid, s);
        if (fMid == 0)
        {
            break;
        }

        // Update the bisection interval knowing the sign of fMid. The current tMid becomes an endpoint of this interval.
        if (fMid > 0)
        {
            tMax = tMid;
        }
        else
        {
            tMin = tMid;
        }
```

```
        // Evaluate DFDT(tMid).  The speed is guaranteed to satisfy dfdt ≥ 0.
        Real dfdt = DFDT(tMid);
        if (dfdt == 0)
        {
            // Division by zero is not allowed.  Try the bisection step.
            if (BisectionConverged(tMin, tMax, s, tMid, fMid))
            {
                break;
            }
        }

        Real tNext = tMid − fMid / dfdt;
        if (tNext == tMid)
        {
            // The precision of type Real is insufficient to disambiguate t and tNext.  This is the best you can do using type Real.
            break;
        }

        // Determine whether to accept the Newton step or try the bisection step.
        tMid = tNext;
        if (tMid < tMin || tMid > tMax)
        {
            // Try the bisection step.
            if (BisectionConverged(tMin, tMax, s, tMid, fMid))
            {
                break;
            }
        }
    }

    return GCPOutput(tMid, fMid, numIterations);
}
```

---

4 Reparameterization for Specified Speed

In the previous section, we had a parameterized curve, $\boldsymbol{Y}(t)$, and asked how to relate the time t to the arclength s in order to obtain a parameterization by arclength, $\boldsymbol{X}(s) = \boldsymbol{Y}(t)$. Equivalently, the idea may be thought of as choosing the time t so that a particle traveling along the curve arrives at a specifed distance s along the curve at the given time.

In this section, we start with a parameterized curve, $\boldsymbol{Y}(u)$ for $u \in [u_{\min}, u_{\max}]$, and determine a parameterization by time t, say, $\boldsymbol{X}(t) = \boldsymbol{Y}(u)$ for $t \in [t_{\min}, t_{\max}]$, so that the speed at time t is a specified function $\sigma(t)$. Notice that in the previous problem, the time variable t is already that of the given curve. In this problem, the curve parameter u is not about time—it is solely whatever parameter was convenient for parameterizing the curve. We need to relate the time variable t to the curve parameter u to obtain the desired speed.

Let L be the arclength of the curve; that is,

$$L = \int_{u_{\min}}^{u_{\max}} \left| \frac{d\boldsymbol{Y}}{du} \right| du \tag{9}$$

This may be computed using a numerical integrator. A particle travels along this curve over time $t \in [t_{\min}, t_{\max}]$, where the minimum and maximum times are user-specified values.

From calculus and physics, the distance traveled by the particle along a path is the integral of the speed

over that path. Thus, we have the constraint

$$L = \int_{t_{\min}}^{t_{\max}} \sigma(t)\, dt \tag{10}$$

In practice, it may be quite tedious to choose $\sigma(t)$ to exactly meet the constraint of Equation (10). Moreover, when moving a camera along a path, the choice of speed might be stated in words rather than as equations, say, "Make the camera travel slowly at the beginning of the curve, speed up in the middle of the curve, and slow down at the end of the curve." In this scenario, most likely the function $\sigma(t)$ is chosen so that its graph in the (t, σ) plane has a certain shape. Once you commit to a mathematical representation, you can apply a scaling factor to satisfy the constraint of Equation (10). Specifically, let $\widehat{\sigma}(t)$ be the function you have chosen based on obtaining a desired shape for its graph. The actual speed function you use is

$$\sigma(t) = \frac{L\widehat{\sigma}(t)}{\int_{t_{\min}}^{t_{\max}} \widehat{\sigma}(t)\, dt} \tag{11}$$

By the construction, the integral of $\sigma(t)$ over the time interval is the length L.

The direct method of solution is to choose a time $t \in [t_{\min}, t_{\max}]$ and compute the distance ℓ traveled by the particle for that time,

$$\ell = \int_{t_{\min}}^{t} \sigma(\tau)\, d\tau \in [0, L] \tag{12}$$

Now use the method described previously for choosing the curve parameter u that gets you to the arclength ℓ. That is, you must numerically invert the integral equation

$$\ell = \int_{u_{\min}}^{u} \left|\boldsymbol{Y}'(\mu)\right|\, d\mu \tag{13}$$

In the section on reparameterization by arclength, the arclength parameter was named s and the curve parameter was named t. Now we have the arclength parameter named ℓ and the curve parameter named u. The technical difficulty in discussing simultaneously reparameterizing by arclength and reparameterizing to obtain a desired speed is that time t comes into play in two different ways, so you will just have to bear with the notation that accommodates all variables involved.

In summary, the parameter t is chosen as the value for which we need to compute u so that the speed at $\boldsymbol{Y}(u)$ is $\sigma(t)$. First, compute the distance ℓ traveled along the $\boldsymbol{Y}$-curve during the time interval $[t_{\min}, t]$. This is accomplished by a numerical integration of equation (12). Second, compute the corresponding u-parameter for the $\boldsymbol{Y}$-curve from the arclength ℓ using one of the GetTParameter implementations.

5 Handling Multiple Contiguous Curves

This section describes how to implement the function GetTParameter(s) for a continuous curve defined as a collection of p curve segments,

$$\boldsymbol{Y}(t) = \begin{cases} \boldsymbol{Y}_1(t), & t \in [t_0, t_1] \\ \boldsymbol{Y}_2(t), & t \in [t_1, t_2] \\ \vdots & \vdots \\ \boldsymbol{Y}_{p-1}(t), & t \in [t_{p-2}, t_{p-1}] \\ \boldsymbol{Y}_p(t), & t \in [t_{p-1}, t_p] \end{cases} \tag{14}$$

where the t_i are monotonic increasing and specified by the user. The curve domain is $[t_{\min}, t_{\max}]$ where $t_{\min} = t_0$ and $t_{\max} = t_p$. For continuity at the user-specified times, the curve segments match at their endpoints: $\boldsymbol{Y}_i(t_i) = \boldsymbol{Y}_{i+1}(t_i)$ for all relevant i. Let L be the total length of the curve $\boldsymbol{Y}(t)$.

Given an arclength $s \in [0, L]$, we wish to compute $t \in [t_{\min}, t_{\max}]$ such that $\boldsymbol{Y}(t)$ is the position that is a distance s measured along the curve from the initial point $\boldsymbol{Y}(t_{\min}) = \boldsymbol{Y}_1(t_0)$. The implementations of the functions Y(t), DYDT(t), Speed(t) and Arclength(t) hide the details of selecting the index i for which $t \in [t_i, t_{i+1}]$. In a naïve implementation, index i is computed by each function, which is inefficient. Moreover, the function Arclength(t) is called multiple times in the pseudocode, which amounts to computing the lookup indices multiple times. To avoid the redundant calculations, we can precompute the arclengths for each curve segment and the partial sums of the arclengths. The hybrid Newton-bisection method always computes t-iterates for the same curve segment, so the computation of index i is performed only once.

Let L_i denote the length of the curve segment $\boldsymbol{Y}_i(t)$; that is

$$L_i = \int_{t_{i-1}}^{t_i} \left| \boldsymbol{Y}_i'(\tau) \right| \, d\tau \tag{15}$$

for $1 \le i \le p$. The total length of the curve $\boldsymbol{Y}(t)$ is

$$L = L_1 + \cdots L_p = \sum_{i=1}^{p} L_i \tag{16}$$

For compact summation indexing, define $L_0 = 0$. If the user-specified arclength is $s = 0$, the returned t-value is $t = t_{\min}$. If the user-specified arclength is $s = L$, the returned t-value is $t_{\max}$. For the user-specified arclength $s \in (0, L)$, search for the index $i \in \{1, \ldots, p\}$ so that $\sum_{j=0}^{i-1} L_j \le s < \sum_{j=0}^{i} L_j$.

Now we can restrict our attention to the curve segment $\boldsymbol{Y}_i(t)$. We need to know how far along this segment to choose the the position so that it is located $s - L_{i-1}$ units of distance from the initial curve point $\boldsymbol{Y}_i(t_{i-1})$. That is, we must solve for t in the integral equation

$$s - \sum_{j=0}^{i-1} L_j = \int_{t_{i-1}}^{t} \left| \boldsymbol{Y}_i'(\tau) \right| \, d\tau \tag{17}$$

6 Implementation

A C++ implementation using Geometric Tools is ReparameterizeByArclength.h, and uses the GTE base class ParametricCurve<N,T> to support querying the curve for position, velocity, and arclength. The class also has support for multiple contiguous curves. A sample main program is shown in listing 3.

Listing 3. Sample code for using the class ReparameterizeByArcLength<N,T>.

```
#include <Mathematics/BezierCurve.h>
#include <Mathematics/PolynomialCurve.h>
#include <Mathematics/ReparameterizeByArclength.h>

int main()
{
    // The curve is Y(t) = (t^3, t^5) for t in [-1,1].
    std::array<Polynomial1<double>, 2> coeff{};
    coeff[0] = { 0.0, 0.0, 0.0, 1.0 };
    coeff[1] = { 0.0, 0.0, 0.0, 0.0, 0.0, 1.0 };
    double tMin = -1.0, tMax = 1.0;
    auto curve = std::make_shared<PolynomialCurve<2, double>>(tMin, tMax, coeff);
    double totalArclength = curve->GetTotalLength();
    // totalArclength = 2.9053418626475693
    double arclength{};

    ReparameterizeByArclength<2, double> repar2(curve);
    ReparameterizeByArclength<2, double>::Output output2{};

    // use bisection
    output2 = repar2.GetT(0.123, true);
    arclength = curve->GetLength(tMin, output2.t);
    // output2.t = -0.97809022308903937
    // output2.f = -2.9143354396410359e-16
    // output2.iterations = 50
    // arclength = 0.12299999999999971

    // use Newton-bisection hybrid
    output2 = repar2.GetT(0.123, false);
    arclength = curve->GetLength(tMin, output2.t);
    // output2.t = -0.97809022308903926
    // output2.f = 2.7755575615628914e-16
    // output2.iterations = 6
    // arclength = 0.12300000000000028

    // use bisection
    output2 = repar2.GetT(0.5 * totalArclength, true);
    arclength = curve->GetLength(tMin, output2.t);
    // output2.t = -8.3535909652709961e-05
    // output2.f = 0.0
    // output2.iterations = 25
    // arclength = 1.4526709313237847

    // use Newton-bisection hybrid
    output2 = repar2.GetT(0.5 * totalArclength, false);
    arclength = curve->GetLength(tMin, output2.t);
    // output2.t = -8.3543227023068973e-05
    // output2.f = 0.0
    // output2.iterations = 29
    // arclength = 1.4526709313237847

    // The curve is a Bezier curve of degree 8.
    std::array<Vector3<double>, 9> controls{};
    controls[0] = { 0.0, 0.0, 0.0 };
    controls[1] = { 1.0, 0.0, 0.125 };
    controls[2] = { 0.0, 1.0, 0.25 };
    controls[3] = { -1.0, 0.0, 0.375 };
    controls[4] = { 0.0, -1.0, 0.5 };
    controls[5] = { 2.0, 0.0, 0.625 };
```

```
    controls[6] = { 0.0, 2.0, 0.75 };
    controls[7] = { -2.0, 0.0, 0.875 };
    controls[8] = { 0.0, -2.0, 1.0 };
    auto bezier = std::make_shared<BezierCurve<3, double>>(8, controls.data());
    ReparameterizeByArclength<3, double> repar3(bezier);
    ReparameterizeByArclength<3, double>::Output output3{};
    totalArclength = bezier->GetTotalLength();
    // totalArclength = 4.7072995195418841

    // use bisection
    output3 = repar3.GetT(0.75 * totalArclength, true);
    arclength = bezier->GetLength(0.0, output3.t);
    // output3.t = 0.93554114969681224
    // output3.f = 0.0
    // output3.iterations = 50
    // arclength = 3.5304746396564131

    // use Newton-bisection hybrid
    output3 = repar3.GetT(0.75 * totalArclength, false);
    arclength = bezier->GetLength(0.0, output3.t);
    // output3.t = 0.93554114969681224
    // output3.f = 0.0
    // output3.iterations = 7
    // arclength = 3.5304746396564131
}
```
