Moving Along a Curve with Specified Speed

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

Created: April 28, 2007
Last Modified: April 2, 2019

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 arc length*. 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, $\mathbf{X}(t)$, 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 $\mathbf{X}(t)$ at time t. The velocity $\mathbf{V}(t)$ of the particle is the rate of change of position with respect to time, a quantity measured by the derivative

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

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

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

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 arc length*. In this case, the curve parameter is typically named s, and the parameter is referred to as the *arc length parameter*. The value s is a measure of distance along the curve. The arc-length parameterization of the curve is written as $\mathbf{X}(s)$, 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 $\mathbf{X}(s) = (\cos(s), \sin(s))$ for $s \in [0, 2\pi)$. The velocity is $\mathbf{V}(s) = (-\sin(s), \cos(s))$ and the speed is $\sigma(s) = |d\mathbf{V}/ds| = |(-\sin(s), \cos(s))| = 1$, which is constant for all s. The particle travels with unit speed around the circle, so $\mathbf{X}(s)$ is an arc-length parameterization of the circle.

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

Suppose you were given $\mathbf{Y}(t)$, 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 arc-length parameter s, say by a function $t = f(s)$, so that $\mathbf{X}(s) = \mathbf{Y}(t)$. In our circle example, $t = \sqrt{s}$. The process of determining the relationship $t = f(s)$ is referred to as *reparameterization by arc length*. 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 Arc Length

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

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

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

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

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

As expected, the speed of traversal along the curve is the length of the velocity vector. This equation tells you how 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

$$s = g(t) = \int_{t_{\min}}^{t} \left|\frac{d\mathbf{Y}(\tau)}{dt}\right| d\tau \tag{5}$$

When time $t = t_{\min}$, the arc length 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 arc length 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 arc length s from the integration. However, what is needed is the inverse problem. Given an arc length s, we want to know the time t at which this arc length occurs; that is, you first decide the distance the object should move along the curve and then figure out the time t at which that occurs. The position on the curve an arc length of s is $\mathbf{Y}(t)$. 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

Define $F(t) = g(t) - s$. Given a value s, the problem is now to find a value t so that $F(t) = 0$. This is a root-finding problem that you might try solving using Newton's method. If $t_0 \in [t_{\min}, t_{\max}]$ is an initial guess for t, Newton's method produces a sequence

$$t_{i+1} = t_i - \frac{F(t_i)}{F'(t_i)}, \quad i \geq 0 \tag{7}$$

where

$$F'(t) = \frac{dF}{dt} = \frac{dg}{dt} = \left|\frac{d\mathbf{Y}}{dt}\right| \tag{8}$$

The iterate t_1 is determined by t_0, $F(t_0)$, and $F'(t_0)$. Evaluation of $F'(t_0)$ is straightforward because you already have a formula for $\mathbf{Y}(t)$ and can compute $d\mathbf{Y}/dt$ from it. Evaluation of $F(t_0)$ requires computing $g(t_0)$, an integration that can be approximated using standard numerical integrators.

A reasonable choice for the initial iterate is

$$t_0 = t_{\min} + \frac{s}{L}\ (t_{\max} - t_{\min}) \tag{9}$$

where the value s/L is the fraction of total arc length at which the particle should be located. The initial iterate uses that same fraction applied to the parameter interval $[t_{\min}, t_{\max}]$. 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 only Newton's method. The derivative $F(t)$ is a nondecreasing function because its derivative $F'(t) = |d\mathbf{Y}/dt|$ 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 should 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 time 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, so the interval length strictly decreases over time. Listing 1 contains pseudocode for the hybrid algorithm.

Listing 1. The pseudocode listed here inverts the integral of equation (5) to obtain the time t for a specified arc length s. The algorithm is a hybrid of Newton's method and bisection.

```
// The t-domain for the curve is [t_min, t_max].
Real tmin, tmax;

// The position is Y(t).
Point Y(Real t);

// The velocity is dY(t)/dt.
Point DYDT(Real t);

// The speed is |dY(t)/dt|.
Real Speed(Real t)
{
    return Length(DYDT(t));
}

// The arclength is s = integral_{t_min}^{t} |dY(tau)/dt| dtau.
Real ArcLength(Real t)
{
    // The Integral function is a numerical integrator for the speed function on the interval [t_min, t].
    return Integral(tmin, t, Speed());
}

// The total length of the curve is L.
Real L = ArcLength(tmax);

// The input is s in [0, L] and the output is t in [t_min, t_max].
Real GetCurveParameter(Real s)
{
    // Choose the initial guess for Newton's method.
    Real t = tmin + (tmax - tmin) * (s / L);

    // Initialize the root-bounding interval for bisection.
    Real lower = tmin, upper = tmax;

    // The positive parameter numIterations is application-specified.
    for (int iterate = 0; iterate < numIterations; ++iterate)
    {
```

```
        Real F = ArcLength(t) - s;

        // The nonnegative parameter epsilon is application-specified.
        if (Abs(F) <= epsilon)
        {
            // |F(t)| is close enough to zero. Report t as the time at which length s is attained.
            return t;
        }

        // Generate a candidate for Newton's method.
        Real DFDT = Speed(t);   // F'(t) > 0 is guaranteed.
        Real tCandidate = t - F / DFDT;

        // Update the root-bounding interval and test for containment of the candidate.
        if (F > 0)
        {
            // F > 0 implies tCandidate < t <= upper.
            upper = t;
            if (tCandidate <= lower)
            {
                // The candidate is outside the root-bounding interval, so use bisection instead.
                t = (upper + lower) / 2;
            }
            else
            {
                // The candidate is in [lower,upper].
                t = tCandidate;
            }
        }
        else  // F < 0
        {
            // F < 0 implies lower <= t < tCandidate.
            lower = t;
            if (tCandidate >= upper)
            {
                // The candidate is outside the root-bounding interval, so use bisection instead.
                t = (upper + lower) / 2;
            }
            else
            {
                // The candidate is in [lower,upper].
                t = tCandidate;
            }
        }
    }

    // A root was not found according to the specified number of iterations and tolerance. You might
    // want to increase imax or epsilon or the integration accuracy. However, in this application it is
    // likely that the time values are oscillating because of the limited numerical precision of floating-point
    // numbers. It is safe to report the last computed time as the t-value corresponding to the s-value.
    return t;
}
```

The function Integral(tmin, t, f()) is any numerical integrator that computes the integral of $f(\tau)$ over the interval $\tau \in [t_{\min}, t]$. A good integrator will choose an optimal set of τ-samples in the interval to obtain an accurate estimate. This gives the Newton's-method approach a global flavor. In comparison, setting up the problem to use a numerical solver for differential equations has a local flavor—you have to apply the solver for many consecutive τ samples before reaching a final estimate for t given s. I prefer the Newton's-method approach because generally t can be computed faster than with differential equation solvers.

3.2 Numerical Solution: Solving a Differential Equation

Equation (4) may be rewritten as the differential equation

$$\frac{dt}{ds} = \frac{1}{|d\mathbf{Y}(t)/dt|} = F(t) \tag{10}$$

where the last equality defines the function $F(t)$. The independent variable is s and the dependent variable is t. The differential equation is referred to as *autonomous*, because the independent variable does not appear in the right-hand function.

Equation (10) may be solved numerically with any standard differential equation solver; I prefer Runge–Kutta methods. For the sake of illustration only, Euler's method may be used. If s is the user-specified arc length and if n steps of the solver are desired, then the step size is $h = s/n$. Setting $t_0 = t_{\min}$, the iterates for Euler's method are

$$t_{i+1} = t_i + hF(t_i) = t_i + \frac{h}{|d\mathbf{Y}(t_i)/dt|}, \quad i \ge 0 \tag{11}$$

The final iteration gives you $t = t_n$, the t-value that approximates s. In Newton's method, you iterate until convergence, a process that you hope will happen quickly. In Euler's method, you iterate n times to compute a sequence of ordered t-values. For a reasonable numerical approximation to t, n might have to be quite large.

Pseudocode for computing t for a specified s that uses a 4th-order Runge–Kutta method is shown in Listing 2.

Listing 2. Pseudocode for a 4th-order Runge–Kutta method to solve numerically the equation (10).

```
// The input is s ∈ [0, L] and the output is t ∈ [tmin, tmax].
Real GetCurveParameter(Real s)
{
    Real t = tmin, h = s / imax;
    for (int i = 1; i <= imax; ++i)
    {
        Real k1 = h / Speed(t);
        Real k2 = h / Speed(t + k1 / 2);
        Real k3 = h / Speed(t + k2 / 2);
        Real k4 = h / Speed(t + k3);
        t += (k1 + 2 * (k2 + k3) + k4) / 6;
    }
    return t;
}
```

---

4 Reparameterization for Specified Speed

In the previous section, we had a parameterized curve, $\mathbf{Y}(t)$, and asked how to relate the time t to the arc length s in order to obtain a parameterization by arc length, $\mathbf{X}(s) = \mathbf{Y}(t)$. 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, $\mathbf{Y}(u)$ for $u \in [u_{\min}, u_{\max}]$, and determine a parameterization by time t, say, $\mathbf{X}(t) = \mathbf{Y}(u)$ 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 arc length of the curve; that is,

$$L = \int_{u_{\min}}^{u_{\max}} \left| \frac{d\mathbf{Y}}{du} \right| du \tag{12}$$

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.

The speed, as a function of time, is also user-specified. Let the function be named $\sigma(t)$. 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

$$L = \int_{t_{\min}}^{t_{\max}} \sigma(t)\, dt \tag{13}$$

In practice, it may be quite tedious to choose $\sigma(t)$ to exactly meet the constraint of Equation (13). 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 (13). Specifically, let $\bar{\sigma}(t)$ be the function you have chosen based on obtaining a desired shape for its graph. The actual speed function you use is

$$\sigma(t) = \frac{L\bar{\sigma}(t)}{\int_{t_{\min}}^{t_{\max}} \bar{\sigma}(t)\, dt} \tag{14}$$

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

4.1 Numerical Solution: Inverting an Integral

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,

$$\ell = \int_{t_{\min}}^{t} \sigma(\tau)\, d\tau \in [0, L] \tag{15}$$

Now use the method described earlier for choosing the curve parameter u that gets you to this arc length. That is, you must numerically invert the integral equation

$$\ell = \int_{u_{\min}}^{u} \left| \frac{d\mathbf{Y}(\mu)}{du} \right| d\mu \tag{16}$$

Here is where you need not to get confused by the notation. In the section on reparameterization by arc length, the arc length parameter was named s and the curve parameter was named t. Now we have the arc length parameter named ℓ and the curve parameter named u. The technical difficulty in discussing simultaneously reparameterizing by arc length 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.

An equivalent formulation that leads to the same numerical solution is the following. Because $\mathbf{X}(t) = \mathbf{Y}(u)$, we may apply the chain rule from calculus to obtain

$$\frac{d\mathbf{X}}{dt} = \frac{d\mathbf{Y}}{du}\frac{du}{dt} \tag{17}$$

Now compute the lengths to obtain

$$\sigma(t) = \left|\frac{d\mathbf{X}}{dt}\right| = \left|\frac{d\mathbf{Y}}{du}\right|\frac{du}{dt} \tag{18}$$

Once again we assume that u and t increase jointly—the particle can never retrace portions of the curve, in which case $du/dt \geq 0$ and the absolute value signs on that term are not necessary. We may separate the variables and rewrite this equation as

$$\int_{u_{\min}}^{u} \left|\frac{d\mathbf{Y}(\mu)}{du}\right| d\mu = \int_{t_{\min}}^{t} \sigma(\tau)\, d\tau = \ell \tag{19}$$

The right-most equality is just our definition of the length traveled by the particle along the curve through time t. This last equation is the same as Equation (16). Pseudocode for computing u given t is contained in Listing 3.

Listing 3. Pseudocode for computing t from u so that at position $\mathbf{Y})(u)$ the speed is the user-specified function $\sigma(t)$.

```
// The u-domain for the curve is [u_min, u_max].
Real umin, umax;

// The position is Y(u).
Point Y(Real u);

// The u-velocity is dY(u)/du.
Point DYDY(Real u);

// The u-speed is |dY(u)/du|.
Real LengthDYDU(Real u)
{
    return Length(DYDU(u));
}

// The arclength is ℓ = ∫_{u_min}^{u} |dY(μ)/du| dμ.
Real ArcLength(Real u)
{
    // The Integral function is a numerical integrator for the u-speed function.
    return Integral(umin, u, LengthDYDU());
}

// The total length of the curve is L.
Real L = ArcLength(umax);

// The user-specified t-domain for the curve is [t_min, t_max].
Real tmin, tmax;

// The user-specified speed at time t is σ(t).
Real Sigma(Real t);

// The input is t ∈ [t_min, t_max] and the output is u ∈ [u_min, u_max].
Real GetU(Real t)
{
    // ℓ is in [0, L].
    Real ell = Integral(tmin,t,Sigma());

    // u is in [u_min, u_max]. The function GetCurveParameter is that of Listing 1.
    Real u = GetCurveParameter(ell);
    return u;
}
```

The function Integral is a numerical integrator. The function GetCurveParameter was discussed previously in this document, which may be implemented using either Newton's method for root finding or a Runge–Kutta method for solving a differential equation.

4.2 Numerical Solution: Solving a Differential Equation

Equation (17) may be solved directly using a differential equation solver. The equation may be written as

$$\frac{du}{dt} = \frac{\sigma(t)}{|d\mathbf{Y}(u)/du|} = F(t,u) \tag{20}$$

where the right-most equality defines the function F. The independent variable is t and the dependent variable is u. This is a *nonautonomous* differential equation, since the right-hand side depends on both the independent and dependent variables.

Pseudocode for computing t for a specified u that uses a 4th-order Runge–Kutta method is shown in Listing 4.

Listing 4. Pseudocode for a 4th-order Runge–Kutta method to solve numerically the equation (20).

```
// The input is t ∈ [t_min, t_max] and the output is u ∈ [u_min, u_max].
Real GetU(Real t)
{
    Real h = (t - tmin) / imax, u = umin, t = tmin;
    for (int i = 1; i <= imax; ++i)
    {
        Real k1 = h * Sigma(t) / LengthDY(u);
        Real k2 = h * Sigma(t + h / 2) / LengthDY(u + k1 / 2);
        Real k3 = h * Sigma(t + h / 2) / LengthDY(u + k2 / 2);
        Real k4 = h * Sigma(t + h) / LengthDY(u + k3);
        t += h;
        u += (k1 + 2 * (k2 + k3) + k4) / 6;
    }
    return u;
}
```

5 Handling Multiple Contiguous Curves

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

$$\mathbf{Y}(t) = \begin{cases} \mathbf{Y}_1(t), & t \in [t_0, t_1] \\ \mathbf{Y}_2(t), & t \in [t_1, t_2] \\ \vdots & \vdots \\ \mathbf{Y}_{p-1}(t), & t \in [t_{p-2}, t_{p-1}] \\ \mathbf{Y}_p(t), & t \in [t_{p-1}, t_p] \end{cases} \tag{21}$$

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: $\mathbf{Y}_i(t_i) = \mathbf{Y}_{i+1}(t_i)$ for all relevant i. Let L be the total length of the curve $\mathbf{Y}(t)$.

Given an arc length $s \in [0, L]$, we wish to compute $t \in [t_{\min}, t_{\max}]$ such that $\mathbf{Y}(t)$ is the position that is a distance s measured along the curve from the initial point $\mathbf{Y}(t_{\min}) = \mathbf{Y}_1(t_0)$. The pseudocode of Listing 2 is applicable here, but 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 naive 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 arc lengths for each curve segment and the partial sums of the arc lengths. 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 $\mathbf{Y}_i(t)$; that is

$$L_i = \int_{t_{i-1}}^{t_i} \left| \frac{d\mathbf{Y}_i}{dt} \right| dt \tag{22}$$

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

$$L = L_1 + \cdots L_p = \sum_{i=1}^{p} L_i \tag{23}$$

For compact summation indexing, define $L_0 = 0$. If the user-specified arc length is $s = 0$, the returned t-value is $t = t_{\min}$. If the user-specified arc length is $s = L$, the returned t-value is $t_{\max}$. For the user-specified arc length $s \in (0, L)$, 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 $\mathbf{Y}_i(t)$. 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 $\mathbf{Y}_i(t_{i-1})$. That is, we must solve for t in the integral equation

$$s - \sum_{j=0}^{i-1} L_j = \int_{t_{i-1}}^{t} \left| \frac{d\mathbf{Y}_i(\tau)}{dt} \right| d\tau \tag{24}$$

Listing 5 contains pseudocode for computing t from a user-specified s.

Listing 5. The pseudocode listed here inverts the integral of equation (5) to obtain the time t for a specified arc length s when the curve $\mathbf{Y}(t)$ is defined as a collection of curve segments.

```
// The user-specified number of curve segments, p ≥ 1.
int p;

// The user-specified increasing sequence of times t_0 through t_p for the curve segment endpoints.
Real time[p+1];

// The domain for Y(t) is [t_min, t_max].
Real tmin = time[0], tmax = time[p];

// The position is Y_i(t) for t ∈ [t_{i-1}, t_i].
Point Y(int i, Real t);
```

```
// The velocity is d\mathbf{Y}_i(t)/dt for t \in [t_{i-1}, t_i].
Point DYDT(int i, Real t);

// The speed is |d\mathbf{Y}_i(t)/dt| for t \in [t_{i-1}, t_i].
Real Speed(int i, Real t)
{
    return Length(DYDT(i, t));
}

// The arclength is s = \int_{t_{i-1}}^{t} |d\mathbf{Y}_i(\tau)/dt| d\tau for t \in [t_{i-1}, t_i].
Real ArcLength(int i, Real t)
{
    // The Integral function is a numerical integrator for the speed function on the interval [t_{i-1}, t].
    return Integral(time[i-1], t, Speed(i,));
}

// Precompute the lengths of the curve segments and the partial sums of the lengths.
Real lengthSegment[p+1], lengthSum[p+1];
lengthSegment[0] = 0;
lengthSum[0] = 0;
for (int i = 1; i <= p; ++i)
{
    lengthSegment[i] = ArcLength(i, time[i]);
    lengthSum[i] = lengthSum[i-1] + lengthSegment[i];
}
Real lengthTotal = lengthSum[p];

// The input is s \in [0, L] and the output is t \in [t_{min}, t_{max}].
// The arc lengths are manipulated relative to the curve segement i,
// but the times are manipulated in the global time scale.
Real GetCurveParameter(Real s)
{
    // Clamp s to [0, L] and specially handle the curve endpoints.
    if (s <= 0)
    {
        return tmin;
    }

    if (s >= lengthTotal)
    {
        return tmax;
    }

    // Compute the index i for which the desired t is in [t_{i-1}, t_i].
    int i;
    for (i = 1; i < p; i++)
    {
        if (s < lengthSum[i])
        {
            break;
        }
    }
    // At this point we know that \sum_{j=0}^{i-1} L_j \le s < \sum_{j=0}^{i} L_j.

    // Choose the initial guess for Newton's method using s and t relative to the segment i.
    Real segS = s - lengthSum[i-1];
    Real segL = lengthSegment[i];
    Real segTMin = time[i-1];
    Real segTMax = time[i];
    Real segT = segTMin + (segTMax - segTMin) * (segS / segL);

    // Initialize the root-bounding interval for bisection.
    Real lower = segTMin, upper = segTMax;

    // The positive parameter numIterations is application-specified.
    for (int iterate = 0; iterate < numIterations; ++iterate)
    {
        Real F = ArcLength(i, segT) - segS;

        // The nonnegative parameter epsilon is application-specified.
```

```
        if (Abs(F) <= epsilon)
        {
            // |F(t_seg)| is close enough to zero. Report t_seg as the time at which length s is attained.
            return segT;
        }

        // Generate a candidate for Newton's method.
        Real DFDT = Speed(i, segT);   // F'(t) > 0 is guaranteed.
        Real segTCandidate = segT - F / DFDT;

        // Update the root-bounding interval and test for containment of the candidate.
        if (F > 0)
        {
            // F > 0 implies segTCandidate < segT ≤ upper.
            upper = segT;
            if (segTCandidate <= lower)
            {
                // The candidate is outside the root-bounding interval, so use bisection instead.
                segT = (upper + lower) / 2;
            }
            else
            {
                // The candidate is in [lower,upper].
                segT = segTCandidate;
            }
        }
        else   // F < 0
        {
            // F < 0 implies lower ≤ segT < segTCandidate.
            lower = segT;
            if (segTCandidate >= upper)
            {
                // The candidate is outside the root-bounding interval, so use bisection instead.
                segT = (upper + lower) / 2;
            }
            else
            {
                // The candidate is in [lower,upper].
                segT = segTCandidate;
            }
        }
    }

    // A root was not found according to the specified number of iterations and tolerance. You might
    // want to increase imax or epsilon or the integration accuracy. However, in this application it is
    // likely that the time values are oscillating because of the limited numerical precision of floating-point
    // numbers. It is safe to report the last computed time as the t-value corresponding to the s-value.
    return segT;
}
```

---

6 Avoiding the Numerical Solution at Runtime

In practice, calling GetCurveParameter(s) many times in a real-time application can be costly. An alternative is to compute a set of pairs (s, t) and fit this set with a polynomial-based function. This function is evaluated with fewer cycles but in exchange for less accurate t-values.

As a simple example, suppose you are willing to use a piecewise linear polynomial to approximate the relationship between s and t. You select $n + 1$ samples and compute t_i for $s_i = Li/n$ for $0 \le i \le n$. Compute the index $i \ge 1$ for which $s_{i-1} \le s < s_i$, and then compute t for which $(t - t_{i-1})/(t_i - t_{i-1}) = (s - s_{i-1})/(s_i - s_{i-1})$.

Naturally, you can use higher-degree fits, say, with piecewise Bézier polynomial curves. Listing 6 contains pseudocode for the fitting and t-from-s computations.

Listing 6. Pseudocode for reparameterization by arc length of a curve using fitting of (s, t) samples.

```
int n = <user-specified quantity>;
Real sSample[n+1], tSample[n+1], tsSlope[n+1];
sSample[0] = 0;
tsSlope[0] = tmin;
stSlope[0] = 0;  // unused in the code
for (i = 1; i < n; i++)
{
    sSample[i] = L * i / n;
    tSample[i] = GetCurveParameter(sSample[i]);
    tsSlope[i] = (tSample[i] - tSample[i-1]) / (sSample[i] - sSample[i-1]);
}
sSample[n] = L;
tSample[n] = tmax;
tsSlope[n] = (tSample[n] - tSample[n-1]) / (sSample[n] - sSample[n-1]);

Real GetApproximateCurveParameter(Real s)
{
    if (s <= 0) { return tmin; }
    if (s >= L) { return tmax; }

    int i;
    for (i = 1; i < n; i++)
    {
        if (s < sSample[i])
        {
            break;
        }
    }
    // We know that sSample[i-1] <= s < sSample[i].

    Real t = tSample[i-1] + tsSlope[i] * (s - sSample[i-1]);
    return t;
}
```

If instead you used a polynomial fit, say $t = f(s)$, then the line of code before the return statement would be Real t = PolynomialFit(s); where Real PolynomialFit (Real s) is the implementation of $f(s)$ and uses the samples tSample[] and sSample[] in its evaluation.