Moving Along a Curve with Specified Speed

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

Created: April 28, 2007
Last Modified: June 24, 2010

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
- performance issues

The concepts apply to curves in any dimension. The curves may be defined as a collection of contiguous curves; for example, 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 time 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 (unit speed along the curve), 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, and 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

$$\sigma(s) = |(-\sin(s), \cos(s))| = 1$$

which is a constant for all time. 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

$$\sigma(t) = \left|(-2t\sin(t^2), 2t\cos(t^2)\right| = 2t$$

The speed increases steadily with time, so the particle does not move with constant speed around the circle.

Suppose that you were given $\mathbf{Y}(t)$, the parameterization of the circle for which the particle does not travel with constant speed. And suppose 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 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 since $|d\mathbf{X}/ds| = 1$.

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 this 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 $t = t_{\min}$, $s = g(t_{\min}) = 0$. This makes sense since the particle hass not yet moved–the distance traveled is zero. When $t = t_{\max}$, $s = g(t_{\max}) = L$, 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. The idea in the original application is that you decide the distance that the object should move

first, then figure out what position $\mathbf{Y}(t)$ corresponds to it. Abstractly, this means inverting the function g to obtain

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

Mathematically, the chances of inverting g in a closed form are slim to none. You 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, then 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 since 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.

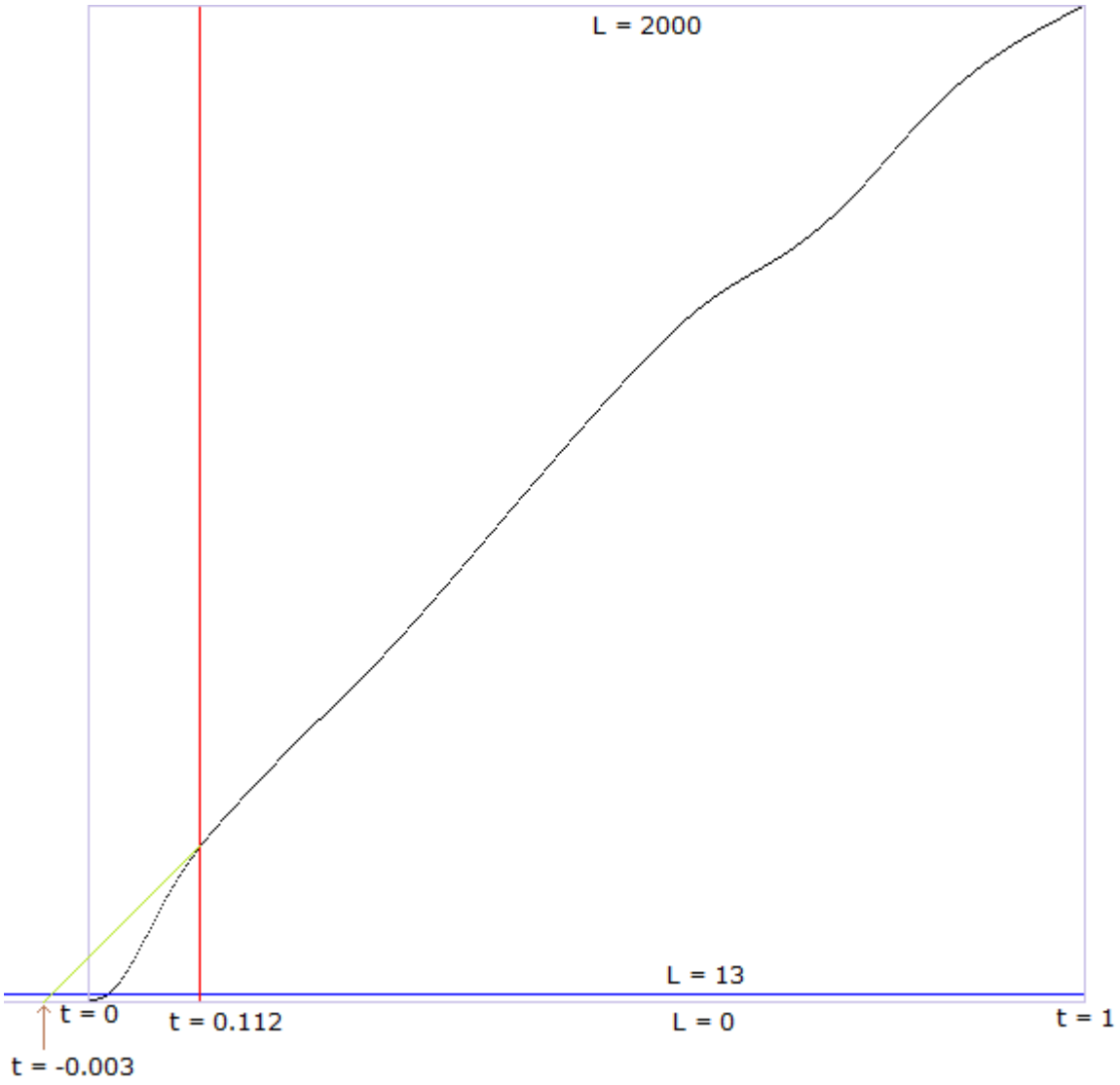
A reasonable choice for the initial iterate is

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

where the value s/L is the fraction of 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 (sufficiency determined by the application) or until a maximum number of iterates has been computed (maximum determined by the application).

There is a potential problem when using only Newton's method. The derivative $F(t)$ is a nondecreasing function because its derivative $F'(t)$ is nonnegative (the magnitude of the speed). 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 $[t_{\min}, t_{\max}]$. The original implementation used only Newton's method. A bug report was filed for a NURBS curve whereby an assertion was triggered, indicating that a Newton iterate was outside the domain. Figure 3.1 shows the graph of $g(t)$ (the arc length function) for $t \in [0, 1]$, the domain of the NURBS curve.

Figure 3.1 The graph of the arc length function for a NURBS curve. The graph is drawn in black. Observe that $g''(t) = F''(t)$ changes sign a few times (sometimes the graph is convex, sometimes it is concave).



The horizontal axis corresponds to t and the vertical axis corresponds to $g(t)$. In the example, the total length of the curve was 2000 and the user wanted to know at what time $\bar{t}$ the length 13 occurred. The initial iterate was $t_0 = 0.006$. The first Newton's iterate generated was $t_1 = 0.112$, which is shown in the figure (red vertical bar). The tangent line at that time is drawn in green. The intersection with the t-axis is chosen to

be the next iterate, which happened to be $t_2 = -0.003$. An assertion in the code was triggered, complaining that t_2 is outside the domain $[0, 1]$.

To avoid the problem with iterates outside the domain, a hybrid of Newton's method and bisection should be used. A root-bounding interval is maintained along with the iterates. A candidate 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.

```
float tmin, tmax;  // The curve parameter interval [tmin,tmax].
Point Y (float t);   // The position Y(t), tmin <= t <= tmax.
Point DY (float t);  // The derivative dY(t)/dt, tmin <= t <= tmax.
float Speed (float t) { return Length(DY(t)); }
float ArcLength (float t) { return Integral(tmin,t,Speed()); }
float L = ArcLength(tmax);  // The total length of the curve.

float GetCurveParameter (float s)  // 0 <= s <= L, output is t
{
    // Initial guess for Newton's method.
    float t = tmin + s*(tmax - tmin)/L;

    // Initial root-bounding interval for bisection.
    float lower = tmin, upper = tmax;

    for (int i = 0; i < imax; i++)  // 'imax' is application-specified
    {
        float F = ArcLength(t) - s;
        if (Abs(F) < epsilon)  // 'epsilon' is application-specified
        {
            // |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.
        float DF = Speed(t);
        float tCandidate = t - F/DF;

        // Update the root-bounding interval and test for containment of
        // the candidate.
        if (F > 0)
        {
            upper = t;
            if (tCandidate <= lower)
            {
                // Candidate is outside the root-bounding interval.  Use
                // bisection instead.
                t = 0.5*(upper + lower);
```

```
            }
            else
            {
                // There is no need to compare to 'upper' because the tangent
                // line has positive slope, guaranteeing that the t-axis
                // intercept is smaller than 'upper'.
                t = tCandidate;
            }
        }
        else
        {
            lower = t;
            if (tCandidate >= upper)
            {
                // Candidate is outside the root-bounding interval.  Use
                // bisection instead.
                t = 0.5*(upper + lower);
            }
            else
            {
                // There is no need to compare to 'lower' because the tangent
                // line has positive slope, guaranteeing that the t-axis
                // intercept is larger than 'lower'.
                t = tCandidate;
            }
        }
    }

    // A root was not found according to the specified number of iterations
    // and tolerance.  You might want to increase iterations or tolerance or
    // integration accuracy.  However, in this application it is likely that
    // the time values are oscillating, due to the limited numerical
    // precision of 32-bit floats.  It is safe to use the last computed time.
    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 it is generally faster to compute the t-value.

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 corresponds to s (a numerical approximation). 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.

Some pseudocode for computing t for a specified s that uses a 4th-order Runge-Kutta method is shown next.

```
float tmin, tmax;  // The curve parameter interval [tmin,tmax].
Point Y (float t);   // The position Y(t), tmin <= t <= tmax.
Point DY (float t);  // The derivative dY(t)/dt, tmin <= t <= tmax.
float Speed (float t) { return Length(DY(t)); }
float ArcLength (float t) { return Integral(tmin,t,Speed()); }
float L = ArcLength(tmax);  // The total length of the curve.

float GetCurveParameter (float s)  // 0 <= s <= L, output is t
{
    float t = tmin;  // initial condition
    float h = s/n;  // step size, 'n' is application-specified
    for (int i = 1; i <= n; i++)
    {
        // The divisions here might be a problem if the divisors are
        // nearly zero.

        float k1 = h/Speed(t);
        float k2 = h/Speed(t + k1/2);
        float k3 = h/Speed(t + k2/2);
        float 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 reparameterization by arc length and reparameterization 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. Since $\mathbf{X}(t) = \mathbf{Y}(u)$, 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).

Some pseudocode for computing u given t is the following.

```
float umin, umax;  // The curve parameter interval [umin,umax].
Point Y (float u);   // The position Y(u), umin <= u <= umax.
Point DY (float u);  // The derivative dY(u)/du, umin <= u <= umax.
float LengthDY (float u) { return Length(DY(u)); }
float ArcLength (float t) { return Integral(umin,u,LengthDY()); }
float L = ArcLength(umax);  // The total length of the curve.
float tmin, tmax;  // The user-specified time interval [tmin,tmax]
float Sigma (float t);  // The user-specified speed at time t.

float GetU (float t)  // tmin <= t <= tmax
{
    float ell = Integral(tmin,t,Sigma());  // 0 <= ell <= L
    float u = GetCurveParameter(ell);  // umin <= u <= umax
    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 (root finding) or a Runge-Kutta Method (differential equation solving).

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.

Some pseudocode for computing t for a specified s that uses a 4th-order Runge-Kutta method is shown next.

```
float umin, umax;  // The curve parameter interval [umin,umax].
Point Y (float u);   // The position Y(u), umin <= u <= umax.
Point DY (float u);  // The derivative dY(u)/du, umin <= u <= umax.
float LengthDY (float u) { return Length(DY(u)); }
float ArcLength (float t) { return Integral(umin,u,LengthDY()); }
float L = ArcLength(umax);  // The total length of the curve.
float tmin, tmax;  // The user-specified time interval [tmin,tmax]
float Sigma (float t);  // The user-specified speed at time t.

float GetU (float t)  // tmin <= t <= tmax
{
    float h = (t - tmin)/n;  // step size, 'n' is application-specified
    float u = umin;  // initial condition
    t = tmin;  // initial condition
    for (int i = 1; i <= n; i++)
    {
        // The divisions here might be a problem if the divisors are
        // nearly zero.

        float k1 = h*Sigma(t)/LengthDY(u);
        float k2 = h*Sigma(t + h/2)/LengthDY(u + k1/2);
        float k3 = h*Sigma(t + h/2)/LengthDY(u + k2/2);
        float 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 is about reparameterization by arc length and has to do with implementing the function `GetCurveParameter(s)` for multiple curve segments. Such an implementation may be used automatically in the application for specifying the speed of traversal.

In many applications, the curve along which a particle or camera travels is specified in p pieces,

$$\mathbf{Y}(t) = \begin{cases} \mathbf{Y}_1(t), & t \in [a_0, a_1] \\ \mathbf{Y}_2(t), & t \in [a_1, a_2] \\ \vdots, & \vdots \\ \mathbf{Y}_{p-1}(t), & t \in [a_{p-2}, a_{p-1}] \\ \mathbf{Y}_p(t), & t \in [a_{p-1}, a_p] \end{cases} \tag{21}$$

where the a_i are monotonic increasing and specified by the user. It must be that $t_{\min} = a_0$ and $t_{\max} = a_p$. Also required is that the curve segments match at their endpoints: $\mathbf{Y}_i(a_i) = \mathbf{Y}_{i+1}(a_i)$ 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 $\mathbf{Y}(0) = \mathbf{Y}_1(a_0)$. The discussion in the previous sections still applies. We may implement

```
Point Y (float t);   // The position Y(t), tmin <= t <= tmax.
Point DY (float t);  // The derivative dY(t)/dt, tmin <= t <= tmax.
float Speed (float t) { return Length(DY(t)); }
float ArcLength (float t) { return Integral(tmin,t,Length(DY()); }
```

so that the function `Y(t)` determines the appropriate curve segment of index i in order to evaluate the position. The same index i is used to evaluate the derivative function `DY(t)` and the speed function `Speed(t)`. The Newton's Method applied to the problem involves multiple calls to the function `ArcLength(t)`. Notice that this function calls a numerical integrator, which in turn calls the function `DY()` multiple times. Each such call of `DY(t)` requires a lookup for i that corresponds to the interval $[a_{i-1}, a_i]$ that contains t. If $i > 1$, we need to compute and sum the total arc lengths of all the segments before the i-th segment, and then add to that the partial arc length of the curve on $[a_{i-1}, t]$.

To avoid repetitive calculations of the total arc lengths of the curve seegments, an application may precompute them. Let L_i denote the total length of the curve $\mathbf{Y}_i(t)$; that is

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

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

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

If $s = 0$ or $s = L$, it is clear how to choose the t-values. For $s \in (0, L)$, define $L_0 = 0$ and search for the index $i \ge 1$ so that $L_{i-1} \le s < L_i$.

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 endpoint $\mathbf{Y}_i(a_{i-1})$. That is, we must solve for t in the integral equation

$$s - L_{i-1} = \int_{a_{i-1}}^{t} \left| \frac{d\mathbf{Y}_i(\tau)}{dt} \right| d\tau \tag{24}$$

Some pseudocode for this is shown next.

```
int p = <user-specified number of curve segments>;
float A[p+1] = <monotonic increasing sequence of interval endpoints>;
float tmin = A[0], tmax = A[p];

// These functions require A[i-1] <= t <= A[i].
Point Y (int i, float t);   // The position Y[i](t).
Point DY (int i, float t);  // The derivative dY[i](t)/dt.
float Speed (int i, float t) { return Length(DY[i](t)); }
float ArcLength (int i, float t) { return Integral(A[i-1],t,Speed(i,); }

// Precompute the lengths of the curve segments.
float LSegment[p+1], L = 0;
LSegment[0] = 0;
for (i = 1; i <= p; i++)
{
    LSegment[i] = ArcLength(i,A[i]);
    L += LSegment[i];
}

float GetCurveParameter (float s)  // 0 <= s <= L, output is t
{
    if (s <= 0) { return tmin; }
    if (s >= L) { return tmax; }

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

    // Initial guess for Newton's method is dt0.
    float length0 = s - A[i-1];
    float length1 = A[i] - A[i-1];
    float dt1 = time[i+1] - time[i];
    float dt0 = dt1*length0/length1;

    // Initial root-bounding interval for bisection.
    float lower = 0, upper = dt1;

    for (int j = 0; j < jmax; ++j)  // 'jmax' is application-specified
    {
        float F = ArcLength(i, dt0) - length0;
```

```
    if (Abs(F) < epsilon)  // ‘epsilon’ is application-specified
    {
        // |ArcLength(i, dt0) - length0| is close enough to zero, report
        // time[i]+dt0 as the time at which ’length’ is attained.
        return time[i] + dt0;
    }

    // Generate a candidate for Newton’s method.
    float DF = GetSpeed(i, dt0);
    float dt0Candidate = dt0 - F/DF;

    // Update the root-bounding interval and test for containment of the
    // candidate.
    if (F > 0)
    {
        upper = dt0;
        if (dt0Candidate <= lower)
        {
            // Candidate is outside the root-bounding interval.  Use
            // bisection instead.
            dt0 = 0.5*(upper + lower);
        }
        else
        {
            // There is no need to compare to ’upper’ because the tangent
            // line has positive slope, guaranteeing that the t-axis
            // intercept is smaller than ’upper’.
            dt0 = dt0Candidate;
        }
    }
    else
    {
        lower = dt0;
        if (dt0Candidate >= upper)
        {
            // Candidate is outside the root-bounding interval.  Use
            // bisection instead.
            dt0 = 0.5*(upper + lower);
        }
        else
        {
            // There is no need to compare to ’lower’ because the tangent
            // line has positive slope, guaranteeing that the t-axis
            // intercept is larger than ’lower’.
            dt0 = dt0Candidate;
        }
    }
}
```

```
    // A root was not found according to the specified number of iterations
    // and tolerance.  You might want to increase iterations or tolerance or
    // integration accuracy.  However, in this application it is likely that
    // the time values are oscillating, due to the limited numerical
    // precision of 32-bit floats.  It is safe to use the last computed time.
    return time[i] + dt0;
}
```

6 Avoiding the Numerical Solution at Runtime

This section is about reparameterization by arc length and has to do with avoiding the computation expense at runtime for computing `GetCurveParameter(s)` by approximating the relationship between s and t with a polynomial-based function.

In practice, calling `GetCurveParameter(s)` many times in a real-time application can be costly in terms of computational cycles. 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 accuracy of t.

As a simple example, suppose you are willing to use a piecewise linear polynomial to approximate the relationship between s and t. You get to select $n + 1$ samples, computing 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

$$\frac{t - t_{i-1}}{t_i - t_{i-1}} = \frac{s - s_{i-1}}{s_i - s_{i-1}} \tag{25}$$

Naturally, you can use higher-degree fits, say, with piecewise Bézier polynomial curves. Some pseudocode for single-curve scenarios is shown next.

```
float tmin, tmax;  // The curve parameter interval [tmin,tmax].
Point Y (float t);   // The position Y(t), tmin <= t <= tmax.
Point DY (float t);  // The derivative dY(t)/dt, tmin <= t <= tmax.
float Speed (float t) { return Length(DY(t)); }
float ArcLength (float t) { return Integral(tmin,t,Speed()); }
float L = ArcLength(tmax);  // The total length of the curve.
float GetCurveParameter (float s);  // Described previously.

int n = <user-specified quantity>;
float 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]);

float GetApproximateCurveParameter (float 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].

    float 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

```
float t = PolynomialFit(s);
```

where `float PolynomialFit (float s)` is the implementation of $f(s)$ and uses the samples `tSample[]` and `sSample[]` in its evaluation.

Some pseudocode for multiple curve segments is shown next.

```
int p = <user-specified number of curve segments>;
float A[p+1] = <monotonic increasing sequence of interval endpoints>;
float tmin = A[0], tmax = A[p];

// These functions require A[i-1] <= t <= A[i].
Point Y (int i, float t);   // The position Y[i](t).
Point DY (int i, float t);  // The derivative dY[i](t)/dt.
float Speed (int i, float t) { return Length(DY[i](t)); }
float ArcLength (int i, float t) { return Integral(A[i-1],t,Speed(i,); }
float GetCurveParameter (float s);  // Described previously.

// Precompute the lengths of the curve segments.  These are used by
// GetCurveParameter.
float LSegment[p+1], L = 0;
```

```
LSegment[0] = 0;
for (i = 1; i <= p; i++)
{
    LSegment[i] = ArcLength(i,A[i]);
    L += LSegment[i];
}

int n = <user-specified quantity>;
float 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]);

float GetApproximateCurveParameter (float 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].

    float t = PolynomialFit(s);
    return t;
}
```