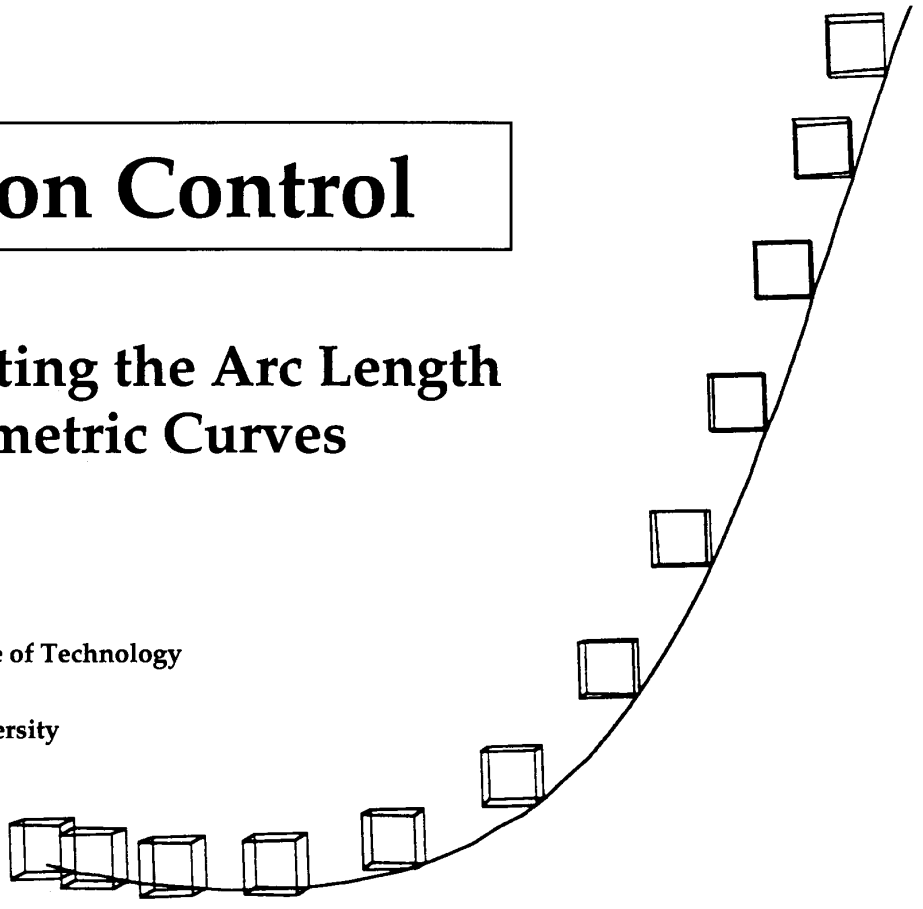
Motion Control

Computing the Arc Length of Parametric Curves

Brian Guenter
Georgia Institute of Technology
Richard Parent
Ohio State University



Parametric space curves are frequently used in computer animation to define motion paths. Specifying constraints on motion is simpler if the curve is parameterized by arc length. Since parametric curves of practical interest cannot be parameterized by arc length, we developed a new numerical technique to compute arc length. The new algorithm is shown to be significantly more efficient than a previous algorithm[1] in the case when many arc length evaluations are made on a single curve, and the algorithm is usually faster even if only a single arc length evaluation is made. Since curves used for path description in computer animation are typically evaluated at hundreds or thousands of points, computational savings using the new algorithm can be quite large.

Parametric space curves are frequently used to define object or virtual camera motion in computer animation. Objects are moved along the curve by choosing a value of the parameterizing variable and then calculating the coordinates of the corresponding point on the curve. It is difficult to specify speed directly because the relationship between the parameterizing variable and curve length is usually very non-linear. In the special case when a unit change in the parameterizing variable results in a unit change in curve length, the curve is said to be parameterized by arc length. Several types of motion specification become simpler if the motion control curve can be parameterized by arc length. Other applications of arc length parameterization include geometrically uniform, as opposed to parametrically uniform, subdivision of space curves or surfaces and warping of objects while maintaining arc length. Unfortunately many types of parametric space curves are difficult or impossible to parameterize by arc length.

Since arc length parameterization is impossible for most curves, several approximate numerical reparameterization techniques have been developed. B-spline curves can be approximately reparameterized by arc length by adjusting the knots of the B-spline curve,[2] but this process can cause the knots to clump and create discontinuities. Our algorithm, although developed independently, improves on a previous algorithm[1] by using a different numerical integration procedure that recursively subdivides

0272-17-16/90/0500-0072$01.00 ©1990 IEEE

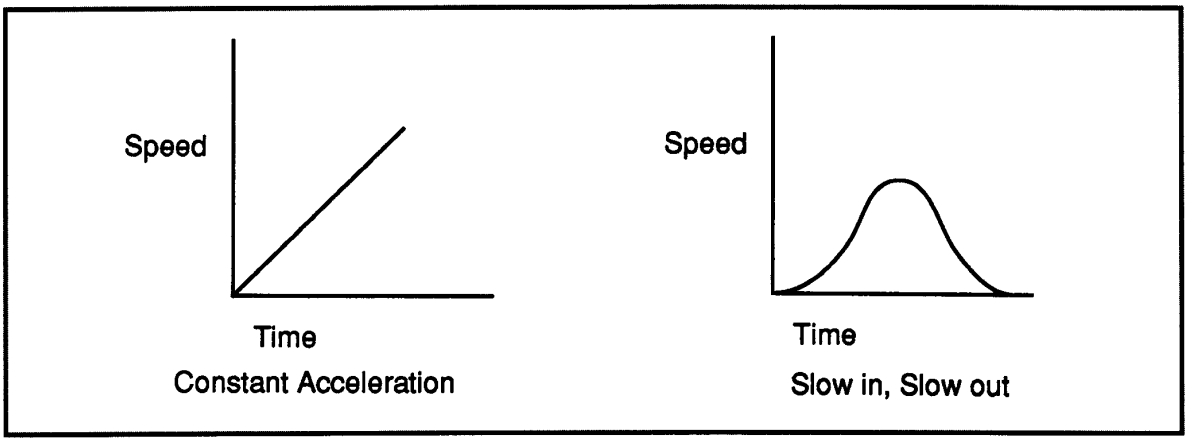


Figure 1. Time-speed curves.

the curve and creates a table of the subdivision points. We use this table to reduce greatly the computation required for subsequent arc length calculations. After table construction, the new algorithm takes nearly constant time for each arc length calculation. The time complexity of the old algorithm is difficult to analyze because it is dependent on the shape, arc length, and number of control points in the curve. A linear increase in the number of control points can result in a more than linear increase in computation. Examples of this type of behavior are shown in the "Results" section of this article.

Time-speed curve control

Time-speed curves relate speed to time through a function $v(t)$. In Figure 1, a constant acceleration and a sinusoidal speed function are shown. More complex time-speed profiles can be created from a sequence of time-speed curves spliced together to maintain continuity of various orders for the function $v(t)$.[3] (The space curve which defines the path along which the object moves is independent of the time-speed curves which define the relative velocity along the path as a function of time. A single space curve could have several time-speed curves defined along it.)

Motion control frequently requires specifying position, speed, and time simultaneously. An example might be specifying the motion of a hand as it reaches out to grasp an object: the hand accelerates toward the object, slows down as it comes close and finally picks the object up. In this example the position and speed are simultaneously specified as a function of time.

Stating the problem more formally, each point to be constrained is an n-tuple, s_i, v_i, a_i, ... , t_i where s_i is distance along the curve, v_i is speed, a_i is acceleration (the ellipsis, ... , indicates that higher order derivatives may also be constrained), and t_i is the time at which all the constraints must be satisfied.

Define the zero-order-constraint problem to be that of satisfying sets of two-tuples s_i, t_i, while variables such as speed and acceleration are allowed to take on any values. Zero-order-constrained motion is illustrated on the left in Figure 2. Notice that there is continuity of position but not speed. The first-order-constraint problem requires satisfying sets of three-tuples s_i, v_i, t_i as shown on the right in Figure 2.

It is sometimes convenient to specify only the first k of the n constraints and force higher order terms to be continuous from time-speed curve c_i to c_{i+1} . This is useful in the case of first-order constraints to allow speed and position to be explicitly specified while forcing acceleration to be continuous from one time-speed curve to the next.

Others[3,4] have described techniques for controlling motion using parametric curves, but several simple, common types of control are troublesome using these techniques.

- Maintaining constant speed or constant acceleration along a space curve
- Controlling position and speed simultaneously at specific times during movement along a space curve

The first problem involves satisfying zero-order constraints at the curve endpoints and specifying a

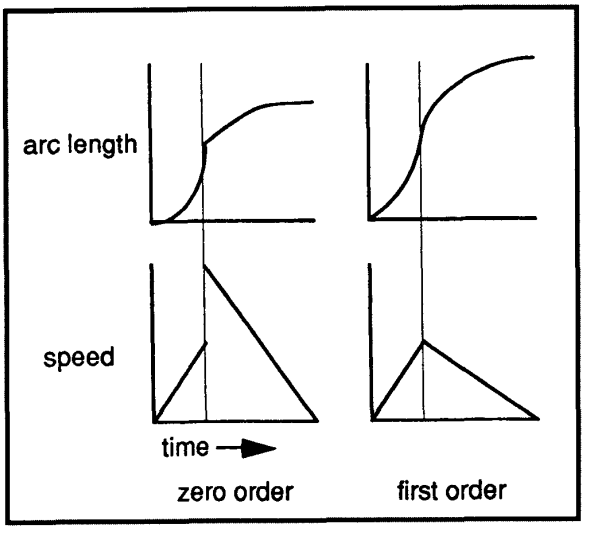


Figure 2. Computing arc length.

speed profile between the endpoints. Constant speed requires a constant time-speed curve and constant acceleration requires a linear time-speed curve. If the beginning and ending tuples are s_i, t_i and $s_i + 1$, t_{i+1} then the total distance traveled, s_{tot}, is

$$s_{tot} = s_{i+1} - si \tag{1}$$

The integral of the speed time function, which is the area under the time-speed curve, is the distance traveled and so must be equal to s_{tot}. If the integral of the time-speed function is not equal to the desired total distance, then it can be scaled by a factor k. From basic calculus scaling the integral is equivalent to scaling the function and then integrating.

The scale factor k can be found easily. Assume that the integral of the time-speed curve can be computed

$$F_s(t) = \int f_s(t)dt \tag{2}$$

then k is

$$k = \frac{S_{tot}}{[F_s(t_1) - F_s(t_0)]} \tag{3}$$

Then satisfying zero-order continuity constraints just requires scaling the time-speed curve. For constant and linear speed profiles, integrating is trivial. Because polynomial spline equations can always be analytically integrated, we can create more general speed profiles from polynomial splines of arbitrary order. For example, assume that a constant acceleration time-speed curve is given with equation $8t$ and integral $4t^2$. Also assume that the tuples to be satisfied are $s_0 = 0$, $t_0 = 0$, $s_1 = 5$, and $t_1 = 3$. The scale constant k is found by solving the equation

$$k = \frac{S_{tot}}{[4(t_1^2 - t_0^2)]} \tag{4}$$

For this particular problem k is 5/36. We can use any time-speed function as long as its integral can be evaluated analytically or numerically.

If speed and position have to be specified simultaneously at several times during the animation, then scaling of the time-speed curve will not work. Only one scaling of the time-speed curve satisfies zero-order constraints, so speed at the endpoints of the time-speed curve cannot be specified independently of position. First-order constraints must be satisfied in order to guarantee continuity of speed at the points where the time-speed curves are spliced together. Assume that the time/arc length curve is a cubic of the form $a_0t^3 + a_1t^2 + a_2t + a_3$. With an equation of this order for each time-speed curve segment, we can satisfy first-order constraints of the form s_i, v_i, t_i. This yields a set of equations

$$\begin{bmatrix} t_i^3 & t_i^2 & t_i & 1 \\ t_{i+1}^3 & t_{i+1}^2 & t_{i+1} & 1 \\ 3t_i^2 & 2t_i & 1 & 0 \\ 3t_{i+1}^2 & 2t_{i+1} & 1 & 0 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} s_0 \\ s_1 \\ v_0 \\ v_1 \end{bmatrix} \tag{5}$$

for each time-speed curve segment (this is essentially the same as performing Hermite interpolation on the time/arc length curve) that can be solved for the coefficients a_0, a_1, a_2, and a_3.

An example of motion controlled using this form of cubic time-speed curve is shown in Figure 3. In this example there are two triples to be satisfied: $s_0 = 0$, $v_0 = 0$, $t_0 = 0$, $s_1 = 1$, $v_1 = 0$, $t_1 = 20$. These triples specify a speed of 0 at the endpoints of the curve with unconstrained speed elsewhere. The spacing between the objects is smaller at the endpoints of the curve. This indicates that speed approaches zero there.

This type of control can lead to non-intuitive types of motion because the interpolating time/arc length curve need not be positive or monotonic. We see these anomalies as retrograde motion when only forward motion is expected. We can add more control points to give finer control of motion and eliminate the retrograde motion.

Computing arc length numerically

The curves we will be dealing with are space curves parameterized by a single variable u

$$\mathbf{R}(u) = (x(u), y(u), z(u)) \tag{6}$$

Length along the curve is denoted by s. It is a standard result from vector calculus[5] that the length from $\mathbf{R}(u_1)$ to $\mathbf{R}(u_2)$ is derived from

$$s = \int_{u_1}^{u_2} \left|\left| \frac{d\mathbf{R}}{du} \right|\right| du \tag{7}$$

where $\frac{d\mathbf{R}}{du}$ is

$$\frac{d\mathbf{R}(u)}{du} = \left(\frac{dx(u)}{du} \frac{dy(u)}{du} \frac{dz(u)}{du} \right) \tag{8}$$

and

$$\left|\left| \frac{d\mathbf{R}(u)}{du} \right|\right| = \sqrt{\left(\frac{dx(u)}{du}\right)^2 + \left(\frac{dy(u)}{du}\right)^2 + \left(\frac{dz(u)}{du}\right)^2} \tag{9}$$

Let the function LENGTH(u_1, u_2) be the arc length from $\mathbf{R}(u_1)$ to $\mathbf{R}(u_2)$. Now there are two problems to be solved.

1. Given parameters u_1 and u_2 find LENGTH(u_1, u_2).
2. Given an arc length s and a parameter value u_1, find u_2 such that LENGTH(u_1, u_2) = s. Equivalently, find the zero of s – LENGTH(u_1,u_2) = 0.

In general, neither of these problems has an analytic solution, so we must use numerical techniques.

Calculating the LENGTH Function

Calculating the LENGTH function involves evaluating the arc length integral. Gaussian quadrature[6] is used because it lends itself to the table building procedure described later and it requires few function evaluations to yield high accuracy.

Gaussian quadrature is defined over an integration interval from –1 to 1. An arbitrary integration interval (a, b) can be fitted to this interval by the linear transformation

$$t = \frac{2x - a - b}{b - a} \tag{10}$$

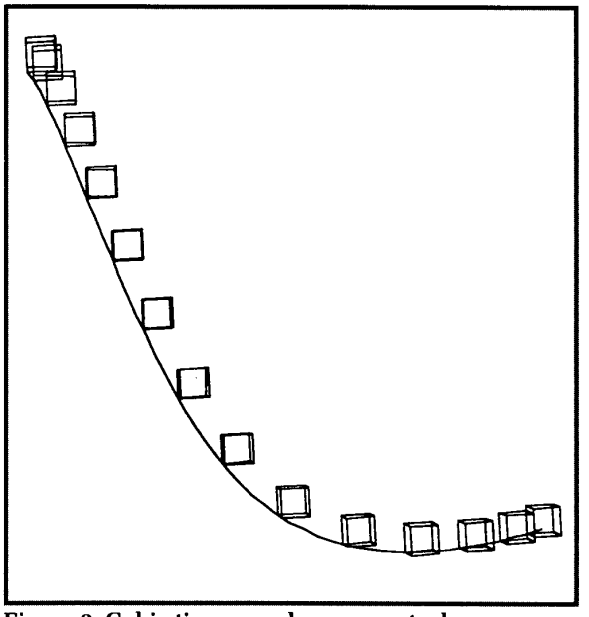


Figure 3. Cubic time-speed curve control.

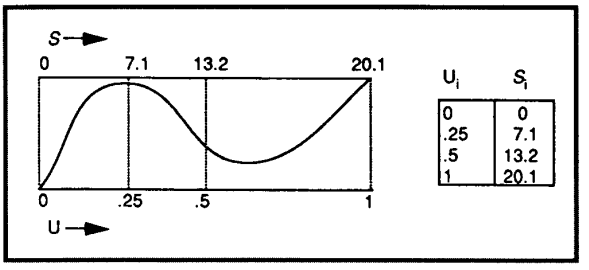


Figure 4. Adaptive subdivision points.

to give

$$\int_a^b f(x) = \int_{-1}^{1} \frac{(b-a)}{2} f\left(\frac{(b-a)t + b + a}{2}\right) \tag{11}$$

The integrand is evaluated at fixed points in the interval –1 to 1 and weighted. The weights and evaluation points for different orders of Gaussian quadrature have been tabulated and can be found in many mathematical handbooks.

The new algorithm makes the Gaussian integration adaptive. The adaptive program divides the interval of integration in half and integrates each subhalf. Then we compare the sum of the two subhalves with the value calculated for the entire interval. If the dif-

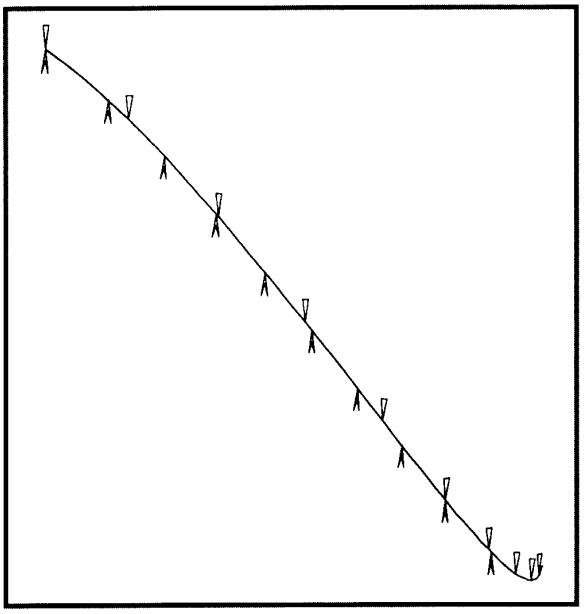


Figure 5. Space curve marked at constant parametric and arc length intervals.

ference between these two values is less than the desired accuracy, then the procedure returns the sum of the subhalves; otherwise it keeps on subdividing.

Adaptive subdivision is the key to the speed of the new algorithm. Initially we use adaptive Gaussian quadrature to calculate the length of the entire space curve. During this process, we create a table of subdivision points is created as shown in Figure 4. Each entry in the table is a pair (u_i, s_i) where s_i is the arc length at parameter value u_i. After the table is built, all subsequent arc length calculations are greatly accelerated by using the table to find the region in which arc length is to be calculated.

When calculating LENGTH(0, u) the table is searched to find the values u_i, u_{i+1} such that $u_i \le u \le u_{i+1}$. The arc length from u_i to u is then computed using non-adaptive Gaussian quadrature. This can be done because the subdivision table has been constructed so that non-adaptive Gaussian quadrature will give the required accuracy over the interval from u_i to u_{i+1}. (This is not guaranteed. Errors may be larger for pathological curves, and in these cases, any sort of adaptive Gaussian quadrature is not likely to yield accurate results. Other more time consuming numerical integration techniques must be applied. However the technique has yielded the necessary accuracy on all curves tested to date, so in practice it appears to be quite robust.) Then LENGTH(0,u) is equal to s_i + LENGTH(u_i, u) and LENGTH(u_1, u_2) = LENGTH(0, u_2) – LENGTH(0, u_1).

A pseudocode description of the adaptive integration and table building procedure follows:

```
procedure Subdivide (left, right,
full_int, total_length, epsilon)
mid = (left+right)/2
 left_value = Integral (left, mid)
 right_value = Integral (mid, right)
   if  |full_int  -  (left_value  +
right_value)|> epsilon
 then
     left_sub  =Subdivide(left,  mid,
left_value,  epsilon/2.0)
  total_length = total_length + left_sub
  Add_element_to_subdivision_table (mid,
total_length)
      return  Subdivide(mid,  right,
right_value, epsilon/2.0) + left_sub)
 end
 else return(left_value + right_value)
 endif
end procedure Subdivide
```

```
procedure Adaptive_integration (left,
right, epsilon)
 full_int = Integral (left,right)
 total_length = 0
  return  (Subdivide  (left,  right,
full_int, total_length, epsilon))
end procedure Adaptive_integration
```

The function `Integral` performs non-adaptive Gaussian quadrature, and the function `Add_element_to_subdivision_table` fills the arc length subdivision table. This function maintains an index into the current subdivision table entry. Each time the function is called, the index is incremented. Consequently the table will be filled with increasing values of u and s.

This solves the first problem posed above, that is, given u_1 and u_2 find LENGTH(u_1, u_2). To solve the second problem, we must use numerical root finding techniques.

Finding u given s

The solution of the equation s – LENGTH(u_1, u) = 0 gives the value of u that is at arc length s from $\mathbf{R}(u_1)$. Since arc length is a strictly monotonically increasing function of u, the solution is unique if $\| d\mathbf{R}/du \|$ is not identically 0 over some interval. We used Newton-

Table 1. Time to compute u given total curve length.

	Table size	Curve length	New alg.: Table creation	Romberg: One evaluation (T_r)	New alg.: One evaluation (T_g)	T_r/T_g
Curve 1	5	1	1.07 sec.	1.5 sec.	.17 sec.	8.8
Curve 2	40	3.11	5.97 sec.	15.2 sec.	.17 sec.	89
Increase in computation from Curve 1 to Curve 2			5.6 times	10 times	no increase	

Raphson iteration to find the root of the equation because it converges rapidly and requires little calculation at each iteration. Newton-Raphson iteration generates the sequence $\{p_n\}$ where

$$p_n = p_{n-1} - \frac{f(p_n - 1)}{f'(p_n - 1)} \quad (12)$$

$f(p_{n-1})$ is equal to s – LENGTH(u_1, p_{n-1}) and $f'(p_{n-1})$ is $d\mathbf{R}/du$ evaluated at p_{n-1}.

Problems may arise: Some of the p_n may lie outside the region of definition of the space curve and $d\mathbf{R}/du$ may be zero, or nearly zero. In the first case all succeeding elements p_{n+1}, p_{n+2} ... will be undefined. The latter problem will cause division by zero or by a very small number. When either problem occurs, binary subdivision can be used instead of Newton-Raphson iteration.

When finding u such that LENGTH(0, u) = s, we searched the subdivision table to find s_i and s_{i+1} such that $s_i \le s \le s_{i+1}$. The solution then lies between u_i and u_{i+1}. Newton-Raphson iteration is applied on this subinterval. We approximated the first iterate by linearly interpolating s between the endpoints of the interval.

Although Newton-Raphson iteration may diverge, previous work[1] and experience with the new algorithm indicate that divergence is rarely encountered in practice. It is worth noting that the integration and root finding procedures are completely independent of the type of space curve used. The only procedure that needs to be modified to accommodate new curve types is the derivative calculation subroutine, which is usually a short program.

Table 2. Relative times for random curves.

Curve	Romberg (T_r)	New algorithm (T_g)	Relative time T_r/T_g
1	342 sec.	13.4 sec.	25.5
2	383 sec.	14.3 sec.	26.7
3	358 sec.	14.4 sec.	24.9
4	168 sec.	12.8 sec.	13.1
5	538 sec.	15.5 sec.	34.7

Results

We compared the algorithms in Table 1. Space curve 1 has control points <–1 1 0> <0 3 0> <1 –2 0> <4 –1.5 0> <4.1 –1.3 0>. Figure 5 shows this space curve marked at constant parametric and arc length intervals. The arc length markers are upward pointing arrows and the parametric markers point down. Spacing by arc length remains constant, while parametric spacing becomes much tighter toward the lower right part of the curve. If the motion of the object was controlled by constant parametric steps, the object would slow down dramatically in this part of the curve.

Space curve 2 has control points <–1 1 0> <0 3 0> <0 –2 0> <4 –1.5 0> <4.1 –1.3 0> <–1 1 0> <0 3 0> <1 –2 0> <4 –1.5 0> <4.1 –1.3 0>. The times shown in Table 1 are for a single computation of u given s equal to the total arc length of the space curve. The new algorithm has an 89 to one speed advantage over Adaptive Romberg integration for space curve 2 after we create the arc length table. Building the arc length table and computing u corresponding to the total length of the space curve takes less time than performing Adaptive Romberg integration once, so the new technique is faster even for a single evaluation.

Table 2 shows timings for five random B-spline space curves. Each space curve had 10 control points whose coordinates were chosen randomly from the interval 0.0 to 10.0. The equation s – LENGTH(0, u) was solved for 40 points evenly spaced in arc length on each space curve. We include table construction time in the time for the new algorithm. The new algorithm is from 13 to 34 times as fast as the old algorithm. Total time for the new algorithm is almost constant while total time for the old algorithm varies by a factor of 3.2.

All programs were written in Common Lisp running on a MacII. All timings are in real (not CPU) seconds. Double precision arithmetic was used throughout. Other similar numerical programs that we worked on

run five to 10 times faster when rewritten in C. It would be reasonable to expect that a C implementation of the new algorithm could execute 30 to 60 times per second on the Mac II. On a typical $10,000 Unix workstation, five to 10 times greater speed could be expected.

The current implementation uses linear search of the arc length subdivision table. For space curves with few control points, this takes negligible time compared to the numerical computations. Space curves with more control points could use binary search to make table search time **O**(log n) where n is the number of entries in the arc length table. The Newton-Raphson root finding performs a fixed number of iterations. The implementation described here limits the number of iterations to eight, which has provided convergence for all the space curves tested to date although pathological curves could require more. Each iteration requires one application of non-adaptive Gaussian quadrature, so the root-finding step is essentially constant time. The average time complexity of the new algorithm is therefore **O**(log n) excluding table construction time. Since table search time is much less than the time for the Newton-Raphson step, the new algorithm will be very nearly **O**(1) until the arc length subdivision table becomes quite large. ■

References

1. R.J. Sharpe and R.W. Thorpe, "Numerical Method for Extracting an Arc Length Parameterization from Parametric Curves". *Computer Aided Design*, Vol. 12, No. 2, March 1982, pp. 79-81.
2. C.J. Judd, and P.J. Hartley, "Parameterization and Shape of B-spline curves for CAD," *Computer Aided Design*, Vol. 12, No. 5, Sept. 1980, pp. 235-238.
3. S. Steketee and N. Badler, "Parametric Keyframe Interpolation Incorporation Kinetic Adjustment and Phrasing Control", Computer Graphics (Proc. Siggraph), Vol. 19, No. 3, July 1985, pp. 255-262.
4. D. Kochanek, and R. Bartels, "Interpolating Splines with Local Tension, Continuity and Bias Control", *Computer Graphics* (Proc. Siggraph), Vol. 18, No. 3, July 1984, pp. 33-41.
5. J.E. Marsden and T.J. Tromba, *Vector Calculus*. New York, W. H. Freeman Company, 1976, pp. 148-150.
6. R.L. Burden, J.D. Faire, and A.C. Reynolds, *Numerical Analysis*, Prindle, Webster, Schmidt, Boston, 1981.

Brian Guenter is an assistant professor of computer science at Georgia Institute of Technology. His research interests include computer animation, dynamic simulation models, image synthesis algorithms, multidimensional signal processing, and visual psychophysics as applied to image synthesis. He received a BS in electrical engineering from the University of Colorado in 1981, an MS in computer science from the Ohio State University in 1984, and a PhD in computer science, also from Ohio State, in 1989. He is a member of ACM and IEEE.

Richard Parent is an assistant professor of computer and information science at Ohio State University. His research interests include computer animation, knowledge-based systems, parallel processing, and high-performance graphics workstations.

Parent received his BS in computer science and mathematics from the University of Dayton in 1972, and his MS and PhD in computer science from Ohio State University in 1973 and 1977. He is a member of IEEE Computer Society and ACM.

Brian Guenter can be reached at the Department of Information and Computer Science, Georgia Institute of Technology, Atlanta GA 30332-0280 or on E-mail at guenter@hydra.gatech.edu

Richard Parent can be reached at the Department of Computer and Information Science, 228 Civil And Aeronautical Engineering Building, 2036 Neil Avenue,The Ohio State University, Columbus OH 43210-1277 or on E-mail at parent@cis.ohio-state.edu.