Approximate Arc Length Parametrization

Marcelo Walter and Alain Fournier
Department of Computer Science
The University of British Columbia
{marcelow|fournier}@cs.ubc.ca

August, 1996

Abstract

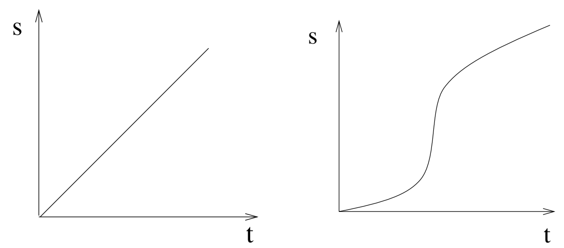
Current approaches to compute the arc length of a parametric curve rely on table lookup schemes. We present an approximate closed-form solution to the problem of computing an arc length parametrization for any given parametric curve. Our solution outputs a one or two-span Bézier curve which relates the length of the curve to the parametric variable. The main advantage of our approach is that we obtain a simple continuous function relating the length of the curve and the parametric variable. This allows the length to be easily computed given the parametric values. Tests with our algorithm on several thousand curves show that the maximum error in our approximation is 8.7% and that the average of maximum errors is 1.9%. Our algorithm is fast enough to compute the closed-form solution in a fraction of a second. After that a user can interactively get an approximation of the arc length for an arbitrary parameter value.

Keywords: arc-length parametrization, approximation, curve design, Bézier parametric curves.

1 Introduction

For a general parametric curve $C(t)$, an arc length parametrization $C(s)$ is such that the length l between two points on the curve $C(s_0)$ and $C(s_1)$ is $l = s_1 - s_0$. In practice any linear relationship between l and s will be called an arc-length parametrization since in this case the curve is easily re-parameterized. For most formulations used in curve design, the length of the curve is not linearly related to the values of the parameter. In many applications of parametric curves it is useful or essential to be able to relate easily the parametric values to the arc length, and reciprocally. A typical example is in computer assisted animation systems where the animator defines a flexible object (such as a shoe-lace, a rope, etc..) whose length is to be kept constant. Arc length parametrization is also needed to compute the speed along curves, such as motion paths used in animation. In this case as well a fast approximate solution is a very useful one.

It is important to note that in most cases we do not necessarily need an *arc-length parametrization*, but just an easy way to relate parameters and length. We can express how the length of a given curve changes with the parametric variable with a graph like the ones in Figure 1. In (a) the curve being considered is parameterized according to arc-length since the length is proportional to t, the parametric variable. In (b) the curve is not parameterized according to arc length. The graphs in Figure 1 also give us the clue on how to determine arc-length or to establish an arc-length parametrization for any given curve.



(a) Arc length parametrization.

(b) Non-arc length parametrization.

Figure 1: Types of parametrization. s is the curve length and t is the parametric variable.

If we are able to construct the curve which describes how the length varies with the parametric variable, we can determine from that curve an arc length parametrization, or from any pair of values of t deduce the length between the corresponding points. Our goal when developing this work is related to the more general problem of keeping the length of a curve constant while it is being manipulated. Having a fast way to relate the length to the parametric variable is a first step towards that goal.

In this paper we present an approach where we express the length of a given parametric curve as a function of the parametric variable using a cubic polynomial expressed as a one- or two-span Bézier curve. Arc length is a strictly monotonically increasing function of t for any *regular parametrization*, that is, when $ds/dt \neq 0$, therefore we can assume that a one or two-span Bézier curve has enough flexibility to represent a large range of possibilities for "length versus t" curves with small or null errors. Our tests approximating cubic parametric curves will indicate that this assumption is true. Besides, our tests will also show that the average and maximum errors are small enough for our method to be useful in a large range of applications.

Comparing to previous approaches ours has also the advantage of less computation since we only need to evaluate the curve length at a fixed number of points (at most 7, when the arc length is computed for cubic parametric curves). Once we have these values we compute the control vertices which define the "length versus t" Bézier curve. The approximation is interpolatory in the sense that we force it to agree with the function being approximated at particular points and from these points we derive the Bézier approximation. Ultimately, we developed a closed-form solution to approximate the arc length of any parametric curve, which is presented in Section 3 of the paper. Once we have the variation of the length against the parametric variable expressed as a continuous curve, it makes subsequent arc length calculations only a matter of computing one point of a Bézier curve, a constant time operation.

Since we assume that we can always approximate the curve "length versus t" as a one or two-span Bézier curve, we have an intrinsic associated error. We show in Section 5 that the error magnitude is small enough for many practical applications. Section 5 also presents the results we achieved when running the algorithm on 3 different sets of two-dimensional cubic parametric curves. The last section presents some comments on possible extensions of this work and on how to achieve smaller errors.

2 Previous Work

The general problem can be stated as follows. From differential geometry, we know that the arc length for a parametric curve in $\Re^3$ is given by [fari90]:

$$s(t) = \int_0^t \sqrt{\dot{x}(t)^2 + \dot{y}(t)^2 + \dot{z}(t)^2}\, dt \tag{1}$$

where the dots denote derivatives with respect to t. This formula when evaluated gives us the length as a function of t. The direct problem computes $s(t)$ for a given t. Analytic solutions for these kind of integrals only exist for very simple functions of low degree (at most 2). Numerically, this integral can be computed using Romberg-integration technique [rals65, pres92], but the cost of this numerical procedure is rather high (remember that in cases of practical interest $x(t)$, $y(t)$ and $z(t)$ are cubic polynomials in t). In [grav95] an approximation for $s(t)$ is computed by adaptive subdivision of the curve. The approximated length is an average of the polygon length (the sum of the sides of the control polygon as defined by the control vertices) and cord length of the curve. This solution is limited to Bézier curves and cannot be used to solve the inverse problem.

An interesting related but different problem is to find a parametric curve which satisfies an arc length constraint, that is, has a specific arc length. Roulier [roul93] presents a solution for this problem where a Bézier curve of specified arc length is computed, given the two end points, two corresponding unit vectors and a positive number for the desired length. The problem is reduced to solving numerically a single non-linear equation in one variable. On the same problem Jou and Han [jou92] build a minimal-energy spline subject to a desired arc length and some other end constraints. Their energy function is related to the curvature of the curve. Fiume [fium95] developed a new class of blending functions called isometric polynomial which allow the manipulation of parametric curves with an arc length constraint. To meet this goal at nearly interactive speeds, he approximates equation (1) by a quadrature-like scheme where the square root is approximated by a power series of an arbitrary low degree polynomial in s.

The inverse problem, determining t for a given length S, can be solved by finding the root of an equation of the type:

$$s(t) - S = 0 \tag{2}$$

This can be solved by Newton-Raphson technique, as for instance in [shar82].

Another class of solutions to the inverse problem uses table lookup schemes. The idea is to create a table where each entry is a pair (t_i, s_i) where s_i is the arc length at parameter value t_i. Once this table is built subsequent arc length determinations use the table to find the interval in which is the desired arc length. That means the table is searched for values $s_i = s(t_i)$ and $s_{i+1} = s(t_{i+1})$ such that $s_i \leq S \leq s_{i+1}$. The desired t lies between t_i and t_{i+1}. The approaches then differ on how these values are used.

In [gira87] for example, the desired t is computed by linearly interpolating t_i and t_{i+1}. A more refined solution is presented in [guen90] where the table entries are adaptively computed according to a desired accuracy. Newton-Raphson is then used to find the root of equation (2) and the table values are used to narrow down the root search.

Table lookup approaches heavily depend on the original number of table entries and for a given accuracy it is not clear how many entries are needed. Besides, the desired solution is achieved after evaluating a fairly large number of curve lengths, which is expensive since it means computing equation (1) whereas in our solution a fixed number of evaluations is needed. The exact number of evaluations in our algorithm depends on the degree of the parametric curve for which we are computing the approximation. For a cubic curve we only need to compute $s(t)$ for 7 specific values of t.

Our solution is presented for the direct problem, that is, given t find $s(t)$. However, an approximate solution to the inverse problem could also be computed by reversing the roles of t and $s(t)$ in the proposed algorithm.

3 The Algorithm

Let $Q_m(t) = (x(t), y(t), z(t))$ be a parametric curve of degree m in $\Re^3$ with $0 \leq t \leq 1 \subset \Re$. We want to compute a *length curve*, that is, a 2D parametric curve which express how the length $s(t)$ of $Q_m(t)$ varies with t. We will call this curve $L(t) = (t, s(t))$, where $s(t)$ is given as [fari90]:

$$s(t) = \int_0^t ||\dot{Q}_m(t)||dt$$

and

$$||\dot{Q}_m(t)|| = \sqrt{\dot{x}(t)^2 + \dot{y}(t)^2 + \dot{z}(t)^2}$$

For simplicity we will use a normalized version $\hat{s}(t)$ of $s(t)$ such that $\hat{s}(t) = 1$ for $s(1)$. The corresponding length curve is then $\hat{L}(t) = (t, \hat{s}(t))$. For now we will assume that $\hat{L}(t)$ can be adequately approximated by a cubic polynomial represented as a one-span Bézier curve. Later on we review this assumption and expand the solution for cases where we need more than one span. The problem of finding an approximation for $\hat{L}(t)$ can now be formulated as follows:

Given a parametric curve $Q_m(t)$, find the 4 control vertices $V_i, i = 0, 1, 2, 3$ which define a Bézier curve $\hat{L}_B(t)$ of degree 3 such that this curve fits as an approximation for $\hat{L}(t)$.

Figure 2 illustrates our problem.

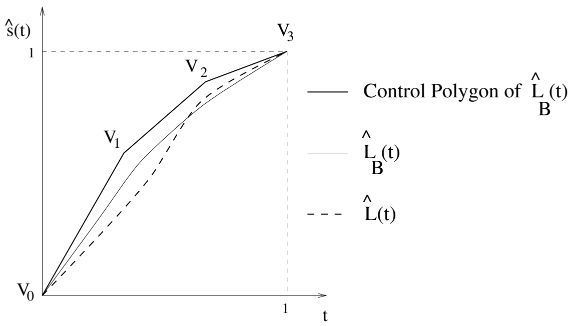


Figure 2: Length against t curve: real (dotted line) and approximation (solid thinner line).

From the way we formulated the problem, we have two control vertices already defined $V_0 = (0, 0)$ and $V_3 = (1, 1)$, since it is reasonable to impose[1] that $\hat{s}(0) = 0$ and $\hat{s}(1) = 1$. Our problem reduces therefore to the problem of computing V_1 and V_2. Expressing $\hat{L}_B(t)$ in matrix form we have:

$$\hat{L}_B(t) = TBV \tag{3}$$

where $T = [t^3\ t^2\ t\ 1]$, $B = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$ $V = \begin{bmatrix} 0 \\ V_1 \\ V_2 \\ 1 \end{bmatrix}$. $\hat{L}_B(t)$ is a two-dimensional curve. Let $x_{\hat{L}_B}(t)$ and $y_{\hat{L}_B}(t)$ be the x and y components of $\hat{L}_B(t)$. We have to compute now $V_1 = (V_{1x}, V_{1y})$ and $V_2 = (V_{2x}, V_{2y})$.

3.1 Computing V_{1x} and V_{2x}

Recall that $\hat{L}_B(t)$ is an approximation for $\hat{L}(t) = (t, \hat{s}(t))$, therefore the x-component of $\hat{L}_B(t)$ should equal t, that is, $x_{\hat{L}_B}(t) = t$. This requirement characterizes $\hat{L}_B(t)$ as a functional curve [fari90] and from the linear precision property of Bézier curves follows that $V_{1x} = \frac{1}{3}$ and $V_{2x} = \frac{2}{3}$.

[1] Because of this imposition the approximation might not be optimal in a *minimax* sense.

3.2 Computing V_{1y} and V_{2y}

Now we can compute V_{1y} and V_{2y}. In order to do that we need the 2 values of the curve length along the curve for V_{1x} and V_{2x} just computed. We will assume now that we know the values for $\hat{s}(1/3)$ and $\hat{s}(2/3)$. Later on we explain how we calculate these two values. From equation 3 follows:

$$V_{1y} = \frac{18\hat{s}(1/3) - 9\hat{s}(2/3) + 2}{6} \tag{4}$$

$$V_{2y} = \frac{-9\hat{s}(1/3) + 18\hat{s}(2/3) - 5}{6} \tag{5}$$

In summary, in order to compute the approximation to the real length versus t curve, we need to compute the actual curve length at 3 points along the curve, respectively at $t = 1/3$, $t = 2/3$ and $t = 1$, i.e., $s(1/3)$, $s(2/3)$ and $s(1)$. Having these 3 values we can compute $\hat{s}(1/3) = \frac{s(1/3)}{s(1)}$ and $\hat{s}(2/3) = \frac{s(2/3)}{s(1)}$.

There are different quadrature methods to compute these lengths at specific t values. A quadrature formula gives us an approximation to the definite integral of a function $f(t)$ as a linear combination of values of $f(t)$:

$$\int_a^b f(t)dt = \sum_{j=1}^{n} H_j \, f(a_j) + E \tag{6}$$

where E is the absolute error in the approximation which includes the derivatives terms of $f(t)$ and H_j are specific weights. When equation (6) is computed with the a_j given as zeros of the Legendre polynomial of degree n we call the quadrature a *Legendre-Gauss quadrature formula* or simply *Gaussian quadrature*. These abscissas and weights are tabulated in many numerical books such as [rals65].

We know from numerical literature [rals65] that if we are not limited, by the way the problem is formulated, to equally spaced points when computing the integral, we may benefit from Gaussian quadratures which can give us higher accuracy than that of Newman-Cotes quadratures[2] if the integrand is smooth in the sense of being well approximated by a polynomial [pres92] which is the case in our problem.

At the same time since we have in mind applications where $\hat{L}_B(t)$ must be computed in real-time, it is important to compute it in the fastest and most accurate way possible. Thus, Gaussian quadratures are natural candidates in our case since they have slightly more favorable error terms than Newton-Cotes formulas for the same small number of points [rals65].

[2] A quadrature formula in which the abscissas (i.e., the points at which the integrand is evaluated) are constrained to be equally spaced is called a Newton-Cotes quadrature formula [rals65].

The number n of points used to compute the quadrature is directly related to the accuracy of the solution. For n points the highest degree polynomial for which E can be made null is $2n - 1$. For $n = 3$, for example, we have the following closed-form solution for computing $s(t)$ at a specific b value:

$$\begin{aligned} s(b) \approx \quad & \frac{b}{2}\left\{\frac{5}{9}||\dot{Q}_m\left(\frac{1.774597b}{2}\right)|| + \frac{8}{9}||\dot{Q}_m\left(\frac{b}{2}\right)|| \right. \\ & \left. + \frac{5}{9}||\dot{Q}_m\left(\frac{0.225403b}{2}\right)||\right\} \end{aligned} \tag{7}$$

and

$$||\dot{Q}_m(t)|| = \sqrt{\dot{x}(t)^2 + \dot{y}(t)^2 + \dot{z}(t)^2}$$

We show in Section 5 the effect of n on the accuracy of our approximation.

3.3 Approximating $\hat{L}_B(t)$ with more than one span

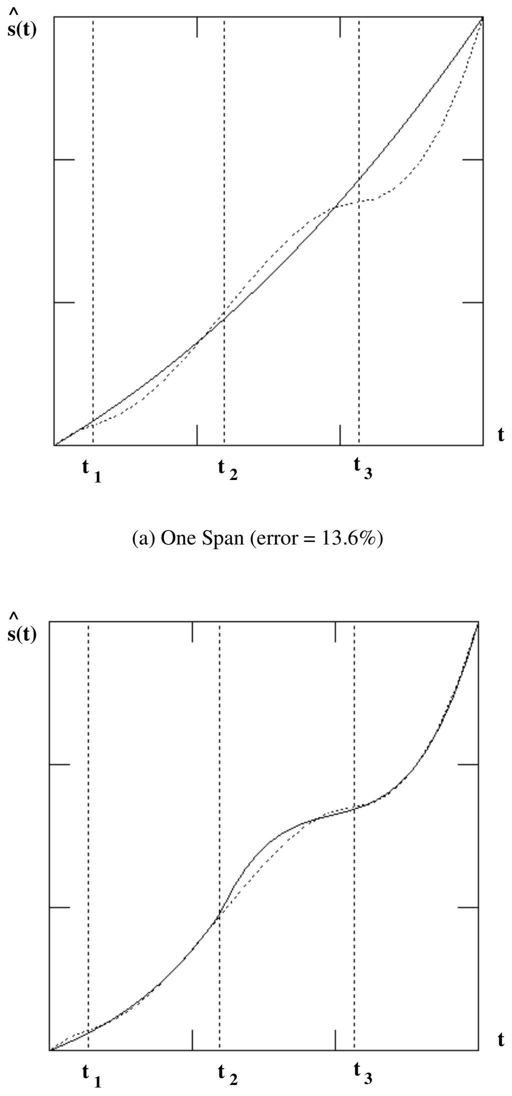
We assumed earlier that a one-span Bézier curve was enough to adequately approximate $\hat{L}(t)$. This assumption is limited by the fact that a one-span Bézier curve can have at most one inflexion point and in general $s(t)$ will have more than one inflexion point. In order to address this issue we have to restrict ourselves to some cases of practical interest, for instance, when the curves for which we need the arc length approximation are cubic. In this case we can assess qualitatively the behaviour of $s(t)$ and use an adaptive scheme to compute an approximation. The criterion used to assess $s(t)$ is the number of inflexion points it has. The number of inflexion points of a function gives us some information about the general behaviour of the function and it is therefore a good candidate to drive an adaptive solution such as ours. For a cubic parametric curve, $s(t)$ will have at most 3 inflexion points since its second derivative is of degree 3, as demonstrated below. The first and second derivatives of $s(t)$ (equation 1) with respect to t are:

$$\dot{s}(t) = (\dot{x}(t)^2 + \dot{y}(t)^2 + \dot{z}(t)^2)^{1/2}$$

$$\ddot{s}(t) = \frac{1}{2}\frac{2\dot{x}(t)\ddot{x}(t) + 2\dot{y}(t)\ddot{y}(t) + 2\dot{z}(t)\ddot{z}(t)}{(\dot{x}(t)^2 + \dot{y}(t)^2 + \dot{z}(t)^2)^{1/2}} \tag{8}$$

The numerator of equation (8) is a cubic equation in t with at most 3 real roots which are the inflexion points of $s(t)$. Our adaptive solution uses the number of inflexion points as input information to decide on how many Bézier spans we need to adequately approximate $s(t)$. The valid inflexion points in our case are the real roots in the interval $(0, 1)$. Our solution will either use one or two spans. For curves where $s(t)$ has zero or one inflexion point we compute the approximation as a one span cubic Bézier following the algorithm described in subsections 3.1 and 3.2. For curves where $s(t)$ has 2 or 3 inflexion points we compute the approximation $\hat{L}_B(t)$ as a two-span cubic Bézier curve. In order to decide where to "break" the two spans, we use the values of t which define the inflexion points of $s(t)$. When it has two inflexion points at t_1 and t_2, the first span approximates $\hat{s}(t)$ for $0 \leq t < (t_2 + t_1)/2$ and the second span for $(t_2 + t_1)/2 \leq t \leq 1$. When $s(t)$ has 3 inflexion points at t_1, t_2, and t_3 the first span approximates $\hat{s}(t)$ for $0 \leq t < t_2$ and the second span for $t_2 \leq t \leq 1$. Since $s(t)$ is a continuous function we have a positional constraint at the t value where the two spans meet. One important observation is that the approximation is only used to control the original curve and in particular a derivative discontinuity in $\hat{L}_B(t)$ does not mean a discontinuity in the original curve(s). The computational cost of using two spans is exactly double the cost of computing only one, plus the cost of computing the number of inflexion points[3] for $s(t)$.

In Figure 3 we can see an example where $s(t)$ has 3 inflexion points (t_1, t_2, and t_3). In Fig. 3(a) $\hat{L}_B(t)$ has only one span and in Fig. 3(b) it has two spans. The decrease in error is very significative, of order 3. We explain how the error is being computed in the next section.



(a) One Span (error = 13.6%)

(b) Two Spans (error = 3.9%)

Figure 3: Difference in the approximating curve when using one or two spans. Dotted line is $\hat{L}(t)$ and the solid line is $\hat{L}_B(t)$. The vertical dotted lines are the inflexion points of $s(t)$.

3.4 Summary of the algorithm

Here is a summary of our algorithm to compute $\hat{L}_B(t)$ as a one or two-span Bézier curve using 3 points Legendre-Gauss quadrature for cubic parametric curves:

1. Find the number of inflexion points for $s(t)$
2. If the number of inflexion points is 0 or 1:
   (a) Compute $s(1/3)$, $s(2/3)$ and $s(1)$ using equation (7)
   (b) Compute $\hat{s}(1/3) = \frac{s(1/3)}{s(1)}$ and $\hat{s}(2/3) = \frac{s(2/3)}{s(1)}$
   (c) Compute V_{1y} and V_{2y} using equations (4) and (5). The 4 control vertices which define $\hat{L}_B(t)$ are then:

$$V_0 = (0, 0) \; V_1 = (1/3, V_{1y})$$

$$V_2 = (2/3, V_{2y}) \; V_3 = (1, 1)$$

[3] Note that we do not constrain the approximation to have the same inflexion point(s) as s(t) itself. This can be added as a constraint but it is not clear whether it would reduce the error more than by introducing some other constraints.

3. Else $s(t)$ has 2 or 3 inflexion points:

   (a) If $s(t)$ has 2 inflexion points at t_1 and t_2. Compute $t_{mid} = (t_2 + t_1)/2$;

   (b) Else $s(t)$ has 3 inflexion points at t_1, t_2, and t_3. Do $t_{mid} = t2$;

   (c) t_{mid} is the point at which the two spans meet. Apply the algorithm described for one-span above twice, when $0 \le t < t_{mid}$ and when $t_{mid} \le t \le 1$.

If we are using more than 3 points to compute the quadrature, the only difference in the algorithm described above will be on step 2(a), where we compute the curve length at specific t values.

4 Errors

It is important to distinguish between the two kinds of errors present in our formulation. The first one is the intrinsic error associated with the numerical computation of the Gaussian quadrature. We can make this error as small as desired if we use enough points when computing the quadrature. The bigger n the smaller the error. For $n = 3$, for example, we have an associated error of:

$$E = \left(\frac{2^3(3!)^2}{6!}\right)^2 \frac{2}{7}\frac{f^{vi}(\eta)}{6!}$$

$$= \frac{f^{vi}(\eta)}{15750} \qquad \eta \in [-1, 1]$$

The second kind of error is the difference between the real curve $\hat{L}(t)$ and the one computed using our algorithm $\hat{L}_B(t)$. There are many possible metrics to measure this error and the most appropriate one is possibly connected to the specific application where the arc length parametrization is necessary. To assess the adequacy of our algorithm we are using a *local* metric defined as the absolute difference between the real length and the computed one as follows:

$$| \hat{L}(t) - \hat{L}_B(t) | \tag{9}$$

It is important to notice that this difference is already relative in the sense that since we are using the normalized version of the real length, the total length of the curve being considered is 1.

5 Results

In order to assess how well $\hat{L}_B(t)$ is fitting $\hat{L}(t)$ we tested our algorithm using 3 sets of 2D cubic Bézier curves. It is important to note that our solution is not limited to 2D curves and the tests could as easily be done for any 3D parametric curves. In this case there would be the extra cost associated with the addition of a third dimension. The first two sets are the Bézier curves which define 2 families (Cooper and Zurich) of fonts (4 fonts per family) from Bitstream, Inc. Figure 4 shows one character from the Cooper family. The third set is a set of 3000 random Bézier

Figure 4: Example of a Cooper font composed of 22 Bézier curves.

curves we created. Each curve in this set is described by its 4 control vertices in 2D. The 8 values were generated by independent calls to a uniformly distributed random generator number function described in [pres92]. The curve depicted in Figure 8 for example has control vertices defined by $(6.885279, 0.753269)$,$(9.738172, 4.203904)$, $(1.734599, 8.224792)$, and $(8.154818, 4.145680)$. These test sets provided a reasonable challenge to our algorithm since the range of curves they span is considerably large for any design for practical purposes. Before using the curves in the test sets we assessed them with respect to how far they were from being already arc length parameterized. We measured $|s(t) - t|$ for t varying from 0 to 1 in steps of 0.05 and kept the maximum value among subintervals. For the random test set for example, the range of values varied from 0.013059 to 0.438717, with an average of 0.1707. We considered these curves far enough from being arc length parameterized to be useful in our tests.

The maximum and average errors for our tests are summarized in Table 1. Since we have 4 fonts per family, we are only listing for each family the font which generated the largest error.

For each curve in each test set we first computed $\hat{L}_B(t)$ as presented in Section 3 of the paper. Then, we made t vary from 0 to 1 in steps of 0.05 and computed the error for each Δt as presented in equation (9). We kept for each curve only the maximum value among all t subintervals.

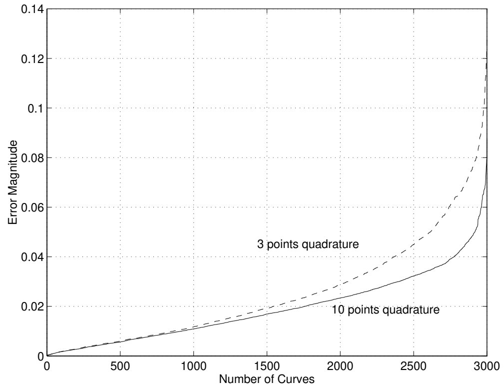


Figure 5: Distribution of errors ($|\hat{L}_B(t) - \hat{L}(t)|$) for the random test set.

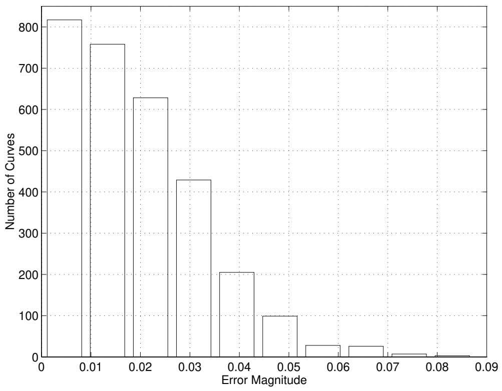


Figure 6: Histogram of errors for the random test set (10 points quadrature).

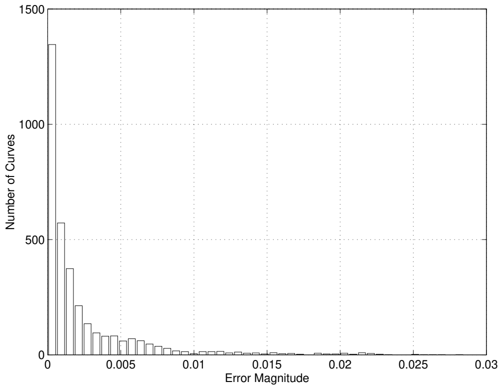


Figure 7: Histogram of errors for the Cooper family font set (10 points quadrature).

Errors			
Test Set	Cooper	Zurich	Random
number of curves	3401	1285	3000
max of set	0.060545	0.030641	0.087392
avg of max errors	0.002432	0.001708	0.019202
avg of averages	0.000946	0.000737	0.007405

Table 1: Errors for $\Delta t = 0.05$, 10 points quadrature.

The "true" value of the curve length, $\hat{L}(t)$, was computed using Simpson's rule with $1.0e-4$ of fractional accuracy. As we can see from the table, the errors for the two font families are considerably small. The maximum error of 6.1% for the Cooper family font, for instance, would mean a difference of about 6 pixels in 100 pixels and the average error would mean a difference of less than half a pixel for 100 pixels. The errors for the random test set were higher than those of the two family fonts as expected. The maximum error for this set was about 44% higher than the maximum error for the font sets but still the average error was only 1.9%. We believe that our random test set gives an upper bound for the maximum error associated with our algorithm.

In Figure 5 we have the errors for the random test set sorted and plotted as a continuous curve. We can clearly see the effect that the number of points used to compute the quadrature has on the final errors. As expected, 10 points quadrature gives us a reasonable improvement over 3 points quadrature. We can see also that for 10 points quadrature more than 2500 curves had errors smaller than 4%, that is, more than 83% of the curves in the set. Figure 6 shows the histogram of errors for the random test set and Figure 7 for the worst case set from the Cooper family font test (since that was the set with the highest error among the two families). Figure 8 shows the Bézier curve from the random test set which had the maximum error when we used 10 points for computing the quadrature. In Figure 9 we show the computed approximation. This is a typical case where our adaptive scheme for deciding how many spans to use fails to adequately approximate the real arc length. There are cases where in spite of the fact that $s(t)$ has only one inflexion point, a two-span Bézier would provide a better approximation. In that case a good alternate strategy is to keep only one span, but to constrain the curve to go through the inflexion point with the tangent determined by $\frac{ds}{dt}$ at that point. A similar approach can be used for the three inflexion points case to determine the two spans.

6 Conclusions

We presented a closed-form solution to compute a cubic 2D Bézier curve which approximates how the length of

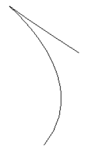


Figure 8: Curve from the random set for which the largest error in the approximation was computed.

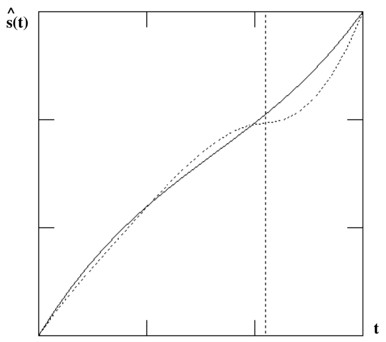


Figure 9: Length against t for curve in Figure 7. $n = 10$, $\Delta t = 0.05$. Error = 8.7%. The dotted line is $\hat{L}(t)$ and the solid line is $\hat{L}_B(t)$. The dotted vertical line indicates the inflexion point of $\hat{L}(t)$.

a given parametric curve is varying with the parametric variable. This curve is important in applications where an arc length parametrization is necessary, providing a simple way to relate the length with the parametric variable. In this way our method gives a fast way to compute an arc length parametrization, since once the length versus t curve is computed it takes only a constant time to find the parametric value associated with a given length and conversely. Our method presented a maximum error of 8.7% and an average error of 1.9% for a test set of 3000 random 2D cubic Bézier curves. We tested the algorithm also using 8 sets of 2D cubic Bézier curves designed as character fonts by Bitstream, Inc. Among all 8 sets the largest error was 6.1% and the worst average error was 0.24%. The errors were computed as the absolute difference between the real curve length and the approximated one. Although we tested only Bézier curves the method does not depend on the type of parametric curve studied. The random curves used as test presented more irregularities than curves actually designed such as for the Bitstream fonts. In particular the random set included curves with cusps, loops and stationary points.

This work has been done in connection with the problem of keeping the length of a given curve constant while the user is manipulating the curve and in this context having a fast and accurate way to compute an arc length parametrization is a first concern. The algorithm is fast enough to compute the length versus t curve in real time, i.e., while the user is manipulating a given parametric curve, our algorithm is fast enough to update the length versus t curve in real time on a Silicon Graphics Indigo 2 workstation.

A possibly serious drawback is that our constraints on $\hat{L}_B(t)$ do not guarantee its monotonicity. Even though this has not been a problem (no approximation was found to be non-monotonic for the non-random curves) it would be indicated to check for monotonicity or better to add constraints to the approximation to enforce it.

We can achieve smaller errors basically in two different ways: computing the quadrature with more points and use different ways to decide on how to select one or two-spans for the approximation. Computing the quadrature with more points is a very straightforward operation and therefore the only consideration to be made is if we are willing to pay the increase in computation time. As for different criteria on when to use two spans instead of one, there are a few issues to address. The first one is the problem of automatically detecting when two-spans are necessary, instead of one. We compute the number of inflexion points that $s(t)$ has and make a decision based on that. Our experience with the test sets have shown that sometimes this criterion is too strict in the sense that it forces a two-span approximation even when a one-span would provide an approximation with roughly the same error magnitude. A more flexible scheme using the inflexion points as the points where we force the approximation to agree with the function and its first derivatives could provide smaller errors in some cases. There is a tradeoff between a general approach like we have now and a more specialized one where the kind of approximation is derived on a case by case basis.

Acknowledgments

The help of Avi Naiman and Bitstream, Inc. in providing the font families is gratefully acknowledged. The first author gratefully acknowledges the financial support of CNPq and FUNDEPE and wishes to thank Rob Walker from UBC for valuable feedback on a first version of this paper. We gratefully acknowledge the support of the Canadian National Science and Engineering Research Council through Research Grants.

References

[fari90] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, 1990.

[fium95] Eugene Fiume. "Isometric Piecewise Polynomial Curves". *Computer Graphics Forum*, Vol. 14, pp. 47–58, Jan 1995.

[gira87] Michael Girard. "Interactive Design of 3D Computer-Animated Legged Animal Motion". *IEEE Computer Graphics and Applications*, Vol. 7, No. 6, pp. 39–51, June 1987.

[grav95] Jens Gravesen. "The Length of Bézier Curves". *Graphics Gems V*, pp. 199–205. Academic Press, Boston, 1995.

[guen90] Brian Guenter and Richard Parent. "Computing the Arc Length of Parametric Curves". *IEEE Computer Graphics and Applications*, Vol. 10, No. 3, pp. 72–78, May 1990.

[jou92] Emery Jou and Weimin Han. "Minimal Energy Splines with Various End Constraints". *Curve and Surface Design*, pp. 23–40. SIAM, 1992.

[pres92] William H. Press et al. *Numerical recipes in C : the art of scientific computing*. Cambridge University Press, 1992.

[rals65] Anthony Ralston. *A First Course in Numerical Analysis*. McGraw-Hill, 1965.

[roul93] John A. Roulier. "Specifying the Arc Length of Bézier Curves". *Computer Aided Geometric Design*, Vol. 10, No. 1, pp. 25–56, Feb 1993.

[shar82] Richard J. Sharpe and Richard W. Thorne. "Numerical Method for extracting and arc length parameterization from parametric curves". *CAD*, Vol. 14, No. 2, pp. 79–81, 1982.