Fast and Accurate Parametric Curve Length Computation

Article *in* Journal of Graphics Tools · January 2001

DOI: 10.1080/10867651.2001.10487548

David R. Forsey
University of British Columbia

Fast and accurate parametric curve length computation

Stephen Vincent (Adobe Systems)
David Forsey (Radical Entertainment)

Abstract

This paper describes a simple technique for evaluating the length of a parametric curve. The technique approximates sections of the curve to the arc of a circle using only sample points on the curve: no computation of derivatives or other properties is required. This method is almost as quick to implement and compute as chord-length summation but is much more accurate : it also provides recursion criteria for adaptive sampling. At the end of paper, we discuss briefly the way the algorithm extends to estimate the area of a parametric surface.

Introduction

The simplest, quickest, and perhaps most widely used technique for determining the length of a parametric curve is chord-length summation (1). Other standard approaches include recursive summation such as the DeCastlejeu algorithm for Bézier curves (2), or integrating the arc length function either by some quadrature technique or by recasting the problem in the form of integrating a differential equation using a technique such as 4th-order Runge-Kutta.

This paper presents a new technique that has the following properties:

- easy to implement.
- cheap to compute.
- good worst-case accuracy characteristics.
- applies to any parametric curve.
- requires only evaluation of points on the curve.

The technique can be summarized as follows:

- Given points precomputed along the curve
- Approximate successive triples of points by circular arcs, the lengths of which can be computed cheaply
- For the basic algorithm, sum the length of non-overlapping arcs; alternatively, sum overlapping arcs to get a second length estimate
- Regions of high curvature can readily be detected and the algorithm made recursive as necessary.

Test results show that the accuracy of the method increases as the 4th power of the number of points evaluated: so doubling the work will result in a 16-fold reduction in the error. Typically this means that evaluating 50 points along the curve will result in an error of a few parts in 10^8, representing an improvement of 3 orders of magnitude over that obtained by chord-length summation of distances between the same sample points along the curve.

Estimation of arc length

Consider three points along a curve : **P0, P1,** and **P2**. A circle with radius **r** and center **C** can be fitted to these points. Let **D1** be the distance from **P0** to **P2**, and **D2** be the sum of the distances from **P0** to **P1** and from **P1** to **P2**.

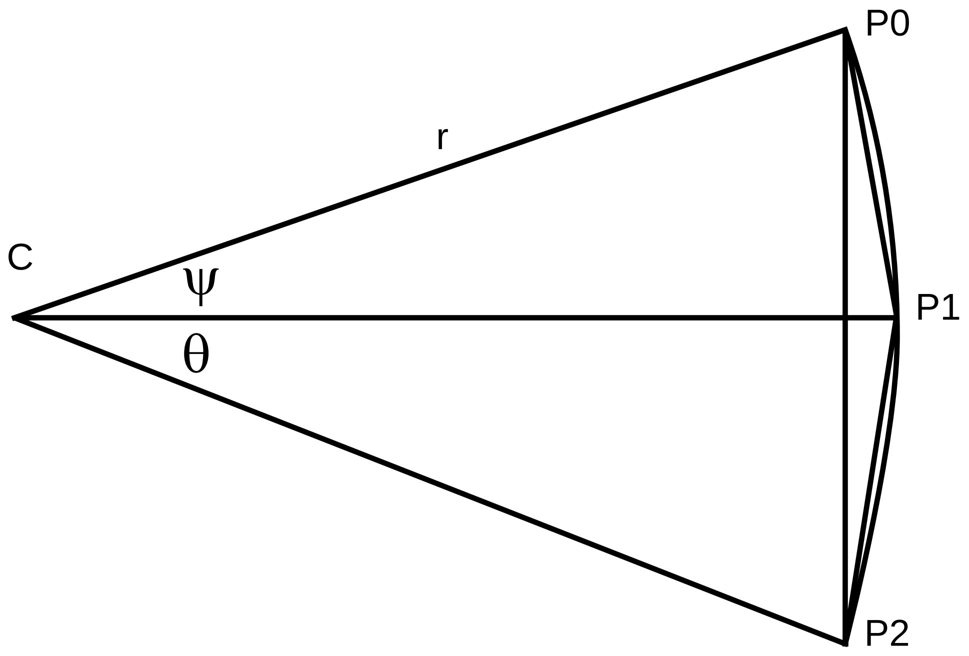


Fig 1

For a circular arc with angle x the arc length is r*x, and for an isosceles triangle with edges r and peak x the length of the base is 2*r*sin(x/2). So we have:

> Arc length = r*Ø + r*ψ
> D1 = 2*r*sin((Ø+ ψ)/2))
> D2 = 2*r*sin(Ø/2)+2*r*sin(ψ/2)

If we assume that P1 is approximately equidistant between P0 and P2, so that Ø and ψ are approximately equal, we get:

> Arc length = 2*r* Ø
> D1 = 2*r*sin(Ø)
> D2 = 4*r*sin(Ø/2)

If we also assume that the region has low curvature, then the angles are small, and we can approximate sin(x) with the first two terms of the Maclaurin series, sin(x) = x - x^3 / 3!.

Substituting into the above, expanding, and recombining to eliminate r and Ø gives:

> Arc length = D2 + (D2 - D1) / 3

The 3D case reduces to 2D by simply working in the plane defined by **P0**, **P1** and **P2** : then C will of necessity lie in the same plane.

In the case of a straight line, D1 and D2 will be equal.

There are two assumptions in the above derivation, firstly that P1 is approximately equidistant from the other two points, secondly that the angles are small. Geometrically it can be detected if either assumption is invalid and this will be discussed further under awkward cases.

Basic algorithm

An odd number of points along the curve are pre-computed. Then successive length estimation of segments along the curve using the above “circle-arc” approximation provides the total length of the curve.

Pre-computation of a suitable number of points (this is discussed later but 31 would be a reasonable value for a parametric cubic curve) along the curve is essential for proper functioning of the algorithm : a purely recursive approach cannot be employed. As an example, a curve that looked like that shown in Fig. 3 would be erroneously interpreted as a straight line by a purely recursive approach.

Sliding Window Estimation

The sliding window estimation makes two length estimates for each subsection of the curve from the same set of samples. The average length will be a more accurate estimate of the arc-length of that segment, and as a bonus detects regions where our circular approximation assumption is poor by examining the magnitude of the difference between the two estimates.

A high order error term is associated with the circle-arc ratio calculation: however, any overestimation of the length of a curve segment is cancelled out by a corresponding underestimate of the length of an adjacent segment.

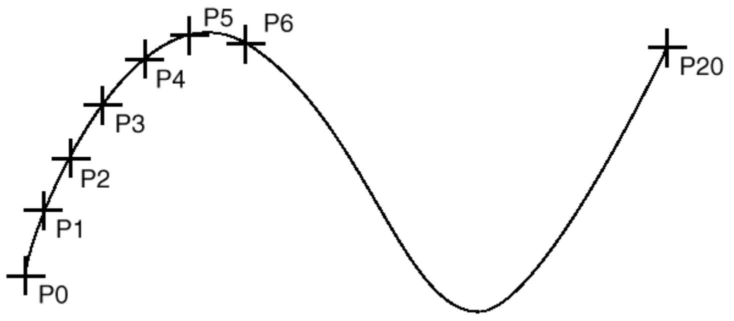


Fig 2

In the example shown in Figure 2, the positions of the 21 samples along the curve are calculated at equal parametric intervals. For all segments, except the first and last, two estimates of the length are calculated as follows:

1) Consider the segment from P1 to P2. With the arc length estimate from P0 to P2, the arc length from P1 to P2 will be approximately equal to the ratio of the straight line distance from P1 to P2 to the sum of the straight line distances from P0 to P1 and again from P1 to P2.

2) A second estimate is made by treating the points (P1, P2, P3) as the arc of a circle and applying a similar calculation to the above.

3) If the two estimates are sufficiently close, they are averaged. Estimates that differ by more than a given amount are an indication that a finer sampling of the region is required.

At the start and end segments of the curve there is only a single estimate for the length of curve in those regions. The length of these segments can be evaluated recursively, or, if the curve was part of a curvature continuous set of curves, by using a sub-section that straddles the adjacent curves.

Generally, the sliding window approach improves accuracy by a factor of two over the basic algorithm, which in turn is 200 times more accurate than chord-length summation (see below).

An alternative and simpler approach to the sliding window modification is to simply perform the basic algorithm twice and average the results. The second sampling uses the same set of points as the first but the indexing of points used for the circle-arc calculation is off by one with respect to the first sample. An appropriate adjustment as above must be made at the ends of the curve for the second sampling. In this approach, we don't check the difference between the estimates of a segment to decide whether to recurse, but instead use the same criteria as the basic algorithm. It is this method that is shown in the pseudocode.

Awkward cases

There are certain configurations of the shape of a parametric curve that can adversely affect the performance of the basic circle-arc algorithm - points of inflection and regions of high curvature, where the assumption that the arc of a circle can adequately represent a curve segment is no longer valid. Fortunately, detection of these cases is straightforward and recursive decomposition of the curve in the problem area will yield a more accurate estimate.

The notation used in these diagrams follows that used in the description of the algorithm above.

Case 1: Points of inflection

It is possible to have a point of inflection in the middle of a curve segment, which, with the y-scale exaggerated, looks like the curve in Figure 3:

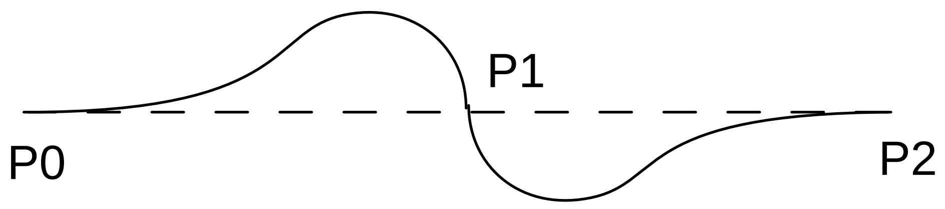


Fig 3

In such a case, the distance from P0 to P2 will equal the sum of the distances from P0 to P1 and from P1 to P2. The basic circle-arc algorithm erroneously estimates the curve as a straight line and so underestimates the length of the curve. In practice, the curvature in the region of an inflection point is mild and the resultant error small. If sampled at a reasonable density, little difference in accuracy is apparent between estimates for Bézier curves with and without inflection points. When using the sliding window approach, there will be a significant difference between the two estimates of the length of the curve in the region around the inflection point, indicating that a finer sampling is required to provide an accurate length estimate.

Case 2: Regions of high curvature

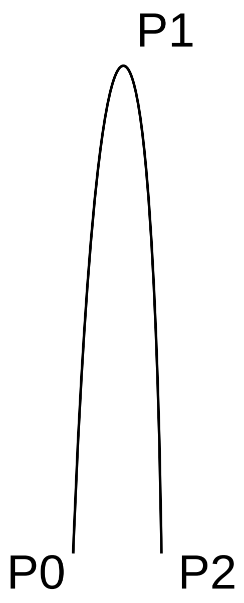


Fig 4a

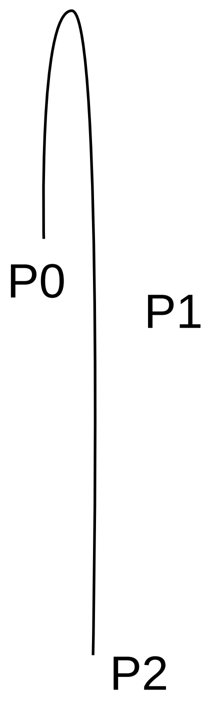


Fig 4b

In Figure 4a, the circle-arc algorithm overestimates the length of the curve. Note however, that D2/D1 >> 1 in such a case and thus regions of high curvature where a finer step size is required for a better length estimate can be detected.

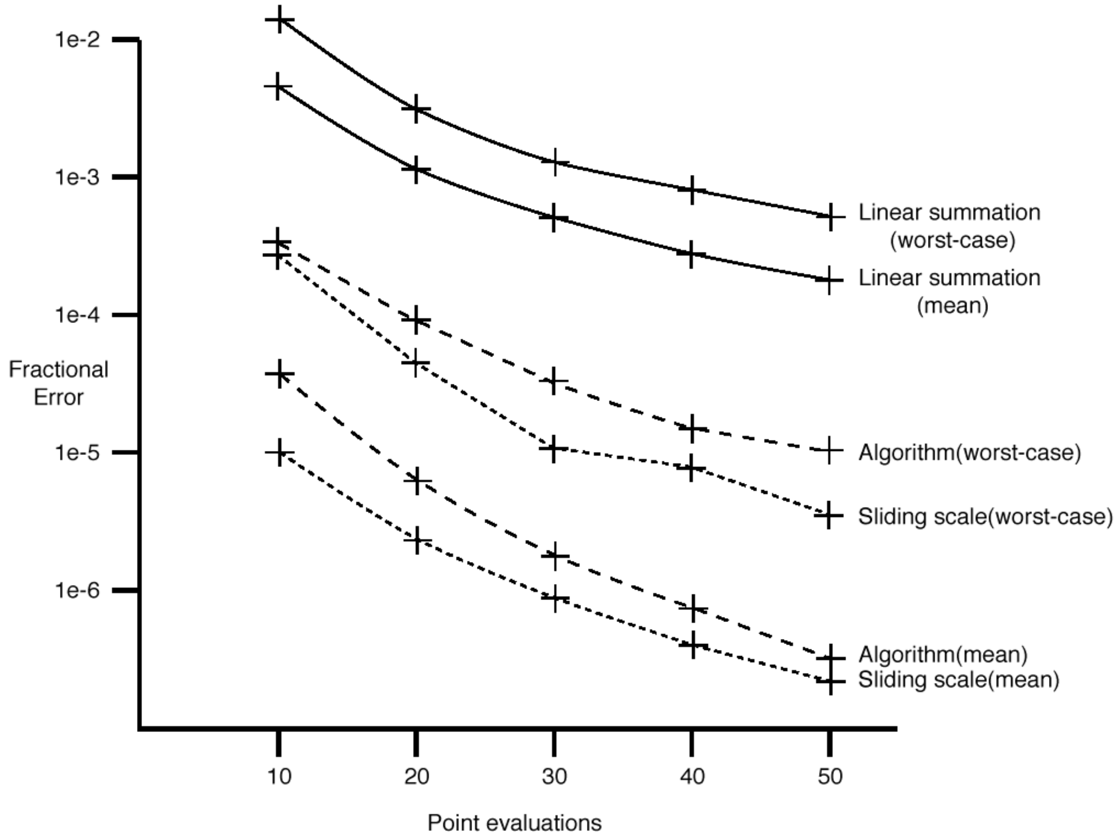
Figure 4b illustrates a variation of this situation. Here the cusp lies close to P0 and so D2 will approximately be equal to D1: i.e. the algorithm will think the curve is a straight line between P0 and P1 (the x-scale has been exaggerated somewhat for clarity). In this case, the ratio of the two straight-line segments that make up D2 will differ significantly from one.

Performance Measurements

Performance testing of the circle-arc algorithm used:

a) Simple curves - such as the ellipse and sin curve.
b) A variety of cubic Bézier curves representative of those found in typical computer graphic applications.

The principle test involved constructing 3D cubic Bézier whose control points were random values within the unit cube. With a sample size of 1000, the following figure shows the results obtained for chord-length summation, the basic circle-arc algorithm and the circle-arc algorithm with the sliding window modification. Results are compared against those obtained using linear chord summation using 100000 data points. The number of point evaluations refers to the precomputed points only : because of the recursion, the number of points may go up.



The black line indicates chord-length summation, the dashed line the basic arc-length algorithm, and the dotted line the arc-length algorithm with the sliding window modification.

Additional tests utilized the following curves.

a) 3D cubic Bézier curves whose first 3 control points are random values within the unit cube; the 4th lying at (1000,0,0)
b) 2D cubic Bézier curves with cusps.
c) 2D cubic Bézier curves with inflection points.

Results for these algorithms were comparable to those obtained for random curves. The results obtained for ellipse and sin curves were equivalent to those for the random cubic Bézier curve tests.

From the data, the error estimate of arc length is inversely proportional to the fourth power of the number of point evaluations: in contrast, chord-length summation has an error inversely proportional to the square of the number of point evaluations.

Comparison with quadrature methods

We conducted tests to compare the accuracy of this method against integration of the arc-length function using either Simpson's rule or Gauss-Legendre quadrature. For 'typical' curves, using approximately the same amount of work (in the sense of calculating derivatives as opposed to curve points), a quadrature method out-performs the circle-arc approach by at least an order of magnitude. However, Gaussian quadrature in general entails calculation of weights and abscissa at each point, which we assume are pre-computed in the above estimation of cost.

However, numerical integration methods have poor worst-case characteristics - behaving badly in the vicinity of a cusp. In such cases, which are easy to detect using a geometric approach, the error in estimating the total length of the curve is up to two orders of magnitude greater than that using the circle-arc algorithm. Guenter and Parent discuss the difficult problem of cusp detection in their paper on adaptive subdivision using Gaussian quadrature (3), but conclude that any quadrature technique will have a problem with pathological cases.

Extension to area calculations

Initial results indicate that this technique extends well to calculating the area of a parametric surface. An element of the surface can be defined by 4 points (u0,u1,v0,v1), the area of which (A1) can be approximated as the sum of two triangles. Recursively subdividing this element into 4 sub-elements, and summing the approximate area of these 4 sub-elements in the same way results in a more accurate estimate (A2). Combining these two estimates in a way similar to that used for curves yields an estimate for the surface area of A2 + (A2-A1)/3. This gives seems to give good results, however the curvature characteristics of surfaces are sufficiently complex to require separate coverage.

References

1) Michael E. Mortenson : Geometric Modelling. published by John Wiley and Sons 1985. p299.

2) Gravesen : Graphics Gems V p199

3) Guenter & Parent : IEEE Computer Graphics and Applications : 10 (3) : 72-78 , May 1990

4) Numerical Recipes in C , 2nd Edition , "W.H.Press , S.A. Teukolsky, W.T.Vettering, B.P.Flannery , Cambridge University Press 1992

Pseudocode

Sample C code is provided on the web site.

The basic algorithm is presented in two forms : with and without the sliding window modification. In each case, an array of precomputed data points is passed in, but additionally the lower-level GetSegmentLength routine can optionally adaptively recurse and call GetPoint() to evaluate more points on the curve in regions of high curvature.

```
double GetSegmentLength ( t0, t1, t2 ,pt0, pt1, pt2 )
{
        //      Compute the length of a small section of a parametric curve
        //       from t0 to t2, with t1 being the mid point.
        //      The 3 points are precomputed.

        d1 = Distance(pt0,pt2)
        da = Distance(pt0,pt1)
        db = Distance(pt1,pt2)
        d2=da+db

        //      if we're in a region of high curvature, recurse, otherwise return the
        //      length as ( d2 + ( d2 – d1 ) / 3 )

        if ( d2<epsilon)
          return ( d2 + ( d2 – d1 ) / 3 )
         else
        if ( d1 < epsilon || d2/d1 > kMaxArc ||
             da < epsilon2 || db/da > kLenRatio ||
             db < epsilon2 || da/db > kLenRatio )
        {
                //      Recurse

                /*
                epsilon and epsilon2 in the above expression are there to guard against division
                by zero and underflow. The value of epsilon2 should be less than half  that of
                epsilon otherwise unnecessary recursion occurs : values of 1e-5 and 1e-6 work
                satisfactorily.

                kMaxArc implicitly refers to the maximum allowable angle that can be
                subtended by a circular arc : a value of 1.05 gives good results in practice.

                kLenRatio refers to the maximum allowable ratio in the distance between
                successive data points : a value of 1.2 gives reasonable results.
                */

                GetPoint ( (t0+t1)/2, mid_pt )
                d3 = GetSegmentLength ( t0, (t0+t1)/2, t1, pt0, mid_pt, pt1)

                GetPoint ( (t1+t2)/2, mid_pt )
                d4 = GetSegmentLength ( t1, (t1+t2)/2, t2, pt1, mid_pt, pt2 )

                return d3 + d4
        }
        else
          return ( d2 + ( d2 – d1 ) / 3 )

}

//      The basic algorithm : estimates curve length from min_t to max_t
//      sampling the curve at n_pts ( which should be odd ).

CurveLength ( min_t , max_t , n_pts )
{
        P : array of n_pts
        T : corresponding array of t values
        length = 0

        for ( i=0 ; i<n_pts-1; i+=2 )
        {
                length += GetSegmentLength
```

```
                                        ( T[i],T[i+1],T[i+2] , P[i],P[i+1],P[i+2] )
            }
}

//          The above algorithm but with the sliding window modification

CurveLengthSlidingWindow ( min_t , max_t , n_pts )
{

            P : array of n_pts
            T : corresponding array of t values

            //          Get first estimate as computed above

            length_a = 0;

            for ( i=0 ; i<n_pts-1; i+=2 )
            {
                        length_a += GetSegmentLength
                                        ( T[i],T[i+1],T[i+2] , P[i],P[i+1],P[i+2] )
            }

            length_b = 0

            //          Start at the second evaluated point : the first will be at min_t

            for ( i=1 ; i<n_pts-2  ; i++ )
            {
                        length_b += GetSegmentLength
                                        ( T[i],T[i+1],T[i+2] , P[i],P[i+1],P[i+2] )

            }

            //          End point corrections : first the first half-section

            mid_t = (T[0]+T[1])/2
            GetPoint ( mid_t, mid_pt )

            length_b += GetSegmentLength
                                        ( T[0],(T[0]+T[1])/2,T[1] , P[0],mid_pt,P[1] )

            //          And the last 2 points

            mid_t = (T[n_pts-2]+T[n_pts-1])/2
            GetPoint ( mid_t, mid_pt )

            length_b += GetSegmentLength
                                        ( T[n_pts-2],mid_t,T[n_pts-1] ,
                                          P[n_pts-2],mid_pt,P[n_pts-1] )

            return ( length_a + length_b ) / 2
}
```