Bi-Conic Subdivision of Surfaces of Revolution and its Applications in Intersection Problems

Jinyuan Jia\textsuperscript{a}  Kai Tang\textsuperscript{b,*}  Ajay Joneja\textsuperscript{c}

\textsuperscript{a}Department of Computer Science, Hong Kong University of Science & Technology, Kowloon, Hong Kong
\textsuperscript{b}Department of Mechanical Engineering, Hong Kong University of Science & Technology, Kowloon, Hong Kong
\textsuperscript{c}Department of IEEM, Hong Kong University of Science & Technology, Kowloon, Hong Kong

Abstract
This paper presents a novel method for subdivision of surfaces of revolutions. Such surfaces occur in a wide variety of applications. Our method approximates the surface by a series of revolute quadrics. To do this transformation, we develop a new technique for approximating the generatrix by a series of pairs of conic sections. By the use of an error estimate based on convex combinations, an efficient least-squares approach is proposed that yields near-optimum fitting. Due to the versatility of the approximating curves, the resulting surface approximation is shown to be significantly more efficient than other tessellation methods in terms of the number of segments required at a given precision level. This in turn allows us to implement efficient and robust algorithms for some of the fundamental intersection problems related to such surfaces. In particular, novel intersection techniques based on the proposed subdivision method are introduced in this paper for the two most fundamental types of intersection in geometric modeling and computer graphics – line/surface and surface/surface intersections. Both use a tight bounding volume called a cylindrical bounding shell, and employ efficient, binary tree based data structures. Several examples are provided for each type of the intersections in the paper. The experimental results show that our method outperforms conventional methods significantly in both computing time and memory cost.

Keywords: Quadric decomposition, bi-conic arc fitting, surfaces of revolution, revolute quadrics

* Corresponding author: Email: mektang@ust.hk; Phone: (852) 2358-8656; Fax: (852) 2358-1543
1. Introduction

Surface rendering and point location on a surface can be accomplished more easily in an implicit than a parametric form. Therefore much research has focused on constructing piecewise algebraic splines. Quadrics are the lowest order curved surfaces; they have good parametric and implicit forms. Constructing a piecewise quadric spline to approximate an arbitrary surface is an important problem in CAGD. Since Powell and Sabin [39] proposed a piecewise $GC^k$ quadric interpolant over a triangular mesh with prescribed normal vectors, many extensions have been proposed for constructing a general Powell-Sabin interpolant. Dahmen [11] and Guo [20] constructed a tangent-plane continuous piecewise triangular quadric net by using the implicit Bezier form in [40] and the split idea of Powell-Sabin for bivariate $GC^k$-piecewise quadric interpolation respectively. Froumentin and Chaillou [19] proposed a simple piecewise quadric interpolant by using simplex B-splines. While their approaches are purely algebraic, Bangert and Prautzsch [2] devised a geometric $GC^k$ piecewise quadric interpolant, by providing a geometric interpretation to certain parameters, it treats Powell-Sabin quadric interpolant as a special case. Several works have discussed how to construct a convex Powell-Sabin interpolant efficiently [3, 9, 17, 49].

However, while suitable for general surfaces, the aforementioned schemes become inefficient or too complicated when they are directly applied to special surfaces. It is natural to explore development of specific quadric subdivision schemes for special surfaces, such as surfaces of revolution, Steiner surfaces, ruled surfaces, canal surfaces, etc. These surfaces have wide applications in CAD/CAM and geometric modeling, and all of them have some useful geometric properties that can be exploited to find more efficient quadric subdivision schemes.

In this paper we develop a new revolute quadric decomposition for surfaces of revolution. A surface of revolution is subdivided into a collection of coaxial trimmed revolute quadrics by correspondingly subdividing the generatrix into a collection of coaxial bi-conic arcs in 2D space. In Fig. 1, we show several schemes of approximation of a surface of revolution. Such approximation is important and actually necessary in a number of applications. In particular, programs that require intersection of a surface with other entities must resort to some form of approximation. An example is ray tracing, where the original surface (of high degree) has to be approximated by simpler geometric elements so that its intersection with a line can be determined within a given tolerance. Since these intersection calculations are usually kernel operations in almost every geometric computation/query, they must be robust, efficient in memory utilization, and fast. Our primary motivation of developing the proposed revolute quadric decomposition is thus to verify that intersection computations based on this new approximation scheme, as compared to the existing ones, can be greatly enhanced in terms of these criteria. Based on the proposed approximation scheme we have also implemented novel algorithms for two fundamental intersections involving surfaces of revolution – (1) intersection of a straight line with a surface of revolution, and (2) intersection between two surfaces of revolution. Our main work and contributions are:

(1) a new conic fitting method - the coaxial bi-conic arc spline - is proposed for fitting the generatrix of a surface of revolution;

(2) efficient intersection algorithms based on the proposed revolute quadric decomposition are designed and implemented for two fundamental geometric operations, namely the intersection between a line and a surface of revolution, and the intersection between two surfaces of revolution themselves;
(3) A selected set of representative existing approximation/intersection schemes for the two intersection operations are implemented; they include the Kajiya's and the truncated cones methods for line/surface intersection, and the Kim's and the truncated cones methods for surface/surface intersection; and finally
(4) the implemented algorithms of both (2) and (3) are then run against a set of test examples and their performance data (e.g., running time and memory requirement) are tabulated for comparison and verification.

Fig. 1 Four decomposition schemes of surfaces of revolution: (a) quadrilateral decomposition, (b) circle decomposition, (c) truncated cone decomposition, and (d) revolute quadric decomposition

The rest of this paper is organized as follows. Section 2 presents the details of the proposed revolute quadric decomposition method. In section 3, the basics of our line/surface and surface/surface intersection algorithms, based on the proposed revolute quadric decomposition method, are described. The experimental results and performance data are then shown in section 4, with the conclusion given in section 5.

2. Revolute Quadric Subdivision

A conic section can be represented as a quadratic polynomial equation

\[ Ax^2 + By^2 + Cxy + Dx + Ey + F = 0. \]

When the revolute axis is the x-axis, the above is reduced to:

\[ Ax^2 + By^2 + Dx + F = 0. \]

Furthermore, we exclude the degenerate case of vertical lines; thus \( B \neq 0 \). Dividing both sides by \( B \), we get:

\[ Ax^2 + y^2 + Dx + F = 0. \]

Here for convenience we still use \( A, D \) and \( F \) to denote \( A/B, D/B \) and \( F/B \), respectively. As a result, there are only three free parameters \( A, D \) and \( F \). They correspond geometrically to two axial radii \( r_a, r_b \), and the x-coordinate \( C_x \) of the conic centre, which together determine the shape of a conic section. On the other hand, by letting the axis of surface of revolution be the x-axis, we reduce the problem of subdividing a surface of revolution into a group of \( GC^i \) coaxial revolute quadrics to that of subdividing the generatrix into a group of \( GC^i \) coaxial conic arcs \( C_i \):

\[ Ax^2 + y^2 + Dx + F_i = 0, \quad i = 1, 2, \ldots, n. \]
There have been some conic fitting algorithms for general curves, for example, see [12, 15, 16, 18, 33, 34, 38, 40]. Among them, [33] presented a classical construction of piecewise conics with tangent continuity; others seek to approximate a planar curve with a group of general conic arcs with curvature continuity. The problem of computing the offsets of piecewise conic arcs was addressed in [16]. In [38] a locally convex $GC^2$ or $GC^3$ conic spline of Hermite-like curvature continuous curves was constructed. A practical construction of $GC^1$ or $GC^2$ conic interpolants for arbitrary type of planar curves was given in [40]. For the purpose of optimal conic approximation to a given planar curve, [18] derived a theoretical upper bound on the Hausdorff distance between a planar curve and the fitting conics in terms of the maximum norm error function between the two. Furthermore, [15] characterized the necessary and sufficient condition for a conic section to be the optimal approximation of a given planar curve and exemplified the characterization on a rational cubic Bezier curve, numerically. However, all of them concern with either general conic fitting or optimized single conic approximation, other than our special case of coaxial conic arcs; they cannot be directly applied in our case. It is thus necessary to devise a specific coaxial conic arc fitting scheme for our revolute quadric subdivision.

Given two endpoints and two associated tangential vectors, if a single coaxial conic is used for interpolation, it requires four constraints, i.e., two positional and two tangential. However, a coaxial conic arc has only three free parameters $A$, $D$, and $F$. This results in an over-constrained problem. In order to deal with this problem, here we propose a coaxial bi-conic spline. For example, given two endpoints $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$, and their associated tangent vectors $T_1(T_{x1}, T_{y1})$ and $T_2(T_{x2}, T_{y2})$, we employ two coaxial conic arcs, which preserve $GC^1$ continuity at their common endpoint $P_m(x_m, y_m)$, rather than a single coaxial conic arc. Our idea is similar to the bi-arc spline method [32, 36, 50, 51]. Clearly, two coaxial conic arcs provide six degrees of freedom, giving rise to six constraints at $P_1$, $P_2$ and $P_m$ for $GC^1$ interpolation. Thus, it is plausible to use coaxial bi-conic splines to resolve the over-constrained issue. The coaxial bi-conic arc fitting method consists of the following four major steps:

- **Partitioning of the generatrix**
  To partition the generatrix into a set of smaller curve segments such that each segment is $x$-monotonic. The conic arc fitting is applied to each segment individually, maintaining $GC^1$ continuity at its two end points. Hereafter, it is assumed that the given generatrix is already in the $x$-monotonic form.

- **Polygonalization**
  To sample the generatrix into a group of points $P_i$ and tangent vectors $T_i$ under the specified error tolerance $e$ for deriving the coefficients $A_i, D_i$ and $F_i$ of the fitting bi-conic arcs, for $i = 1, 2, ..., n$.

- **Local bi-conic fitting**
  To construct an optimal $GC^1$ coaxial bi-conic interpolator to fit the generatrix locally between consecutive sampling points.

- **Global bi-conic fitting**
  To approximate the generatrix with local coaxial bi-conic interpolation within a specified tolerance, $e$; the objective is to minimize the number of local interpolators.

It is worth noting that since our polygonization is independent of the form of the generatrix, it is applicable to both parametric and algebraic representations, and even the discrete form.
2.1. Single coaxial conic fitting

It is possible to use a $GC^2$ conic fitting method to approximate an arbitrary generatrix curve based on single conic arc fitting in [26]. As shown in Figure 2, given $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$, and the tangential vector $T_1(T_{x_1}, T_{y_1})$ or the derivative $d_1=\frac{T_{y_1}}{T_{x_1}}$ at $P_1(x_1, y_1)$, a coaxial conic arc $C_1: Ax^2 + y^2 + Dx + F = 0$ on $[x_1, x_2]$ can be constructed by solving the system of linear equations (1), which interpolates $P_1, P_2$ and $T_1$.

$$
\begin{align*}
Ax_1^2 + y_1^2 + Dx_1 + F &= 0 \\
Ax_2^2 + y_2^2 + Dx_2 + F &= 0 \\
(2Ax_1 + D) &= -2y_1
\end{align*}
$$

$$
\Rightarrow \quad \begin{pmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ 2x_1 & 1 & 0 \end{pmatrix} \begin{pmatrix} A \\ D \\ F \end{pmatrix} = \begin{pmatrix} -y_1^2 \\ -y_2^2 \\ -2y_1\frac{T_{y_1}}{T_{x_1}} \end{pmatrix}
$$

By solving (1), we can obtain $A, D$ and $F$. We then can compute the derivative of $C_1$ at the other end point $x_2, d_2 = \frac{2Ax_2 + D}{y_2}$. To maintain $C^1$ continuity between $C_1$ (on $[x_1, x_2]$) and the next conic arc $C_2$ (on $[x_2, x_3]$) at the common joint point $P_3(x_3, y_3)$, the derivative of $C_2$ at $x_3$ must be equal to $d_2$. Therefore, the derivative at the starting point of each conic arc depends on that of the ending point of the previous fitting conic arc. Only the derivative of the first fitting conic arc at its starting point is required which agrees with that of the generatrix. We call this dependence the “global derivative dependence”. Though extremely simple, due to its ignorance of the tangents at the interior sample points, the single-conic-fitting scheme tends to produce oscillations along the generatrix (see Figure 3).

![Fig. 2 Fitting a generatrix based on single conic arc fitting scheme](image)

![Fig. 3 Oscillations due to single conic fitting](image)

2.2. Coaxial bi-conic splines

Given $P_1(x_1, y_1)$ and $P_3(x_3, y_3)$, and their associated tangential vectors $T_1(T_{x_1}, T_{y_1})$ and $T_3(T_{x_3}, T_{y_3})$, two coaxial conic arcs $C_i: Ax_i^2 + y_i^2 + Dx_i + F_i = 0$ on $[x_1, x_3]$, and $C_2: Ax_2^2 + y_2^2 + Dx_2 + F_2 = 0$ on $[x_1, x_2]$ are joined together to fit the generatrix on $[x_1, x_3]$, where $x_m \in [x_1, x_3]$. By $P_m(x_m, y_m)$ we denote the common endpoint of two coaxial conic arcs $C_1$ and $C_2$, which should preserve $GC^2$ continuity between $C_1$ and $C_2$ at $P_m$. Here, $P_m$ may or may not lie on the generatrix, so as to provide the required flexibility; it must not however violate the prescribed error tolerance. In total, there are now six coefficients (degrees of freedom), $A_1, D_1, F_1, A_2, D_2$, and $F_2$ , subject to the six constraints:
• $C_1$ passing through $P_1$;
• $C_2$ passing through $P_3$;
• $C_1$ being tangent to $T_1$;
• $C_2$ being tangent to $T_3$;
• $C_1$ joining $C_2$ at $P_m$ ($GC^0$ continuity);
• The derivatives of $C_1$ and $C_2$ agree at $P_m$ ($GC^1$ continuity).

Accordingly, the following system of equations is derived:

\[
\begin{align*}
A_1 x_1^2 + y_1^2 + D_1 x_1 + F_1 &= 0 \\
(2 A_1 x_1 + D_1) T_{x_1} &= 2 y_1 T_{y_1} \\
A_2 x_2^2 + y_2^2 + D_2 x_2 + F_2 &= 0 \\
(2 A_2 x_2 + D_2) T_{x_2} &= 2 y_2 T_{y_2} \\
A_1 x_m^2 + y_m^2 + D_2 x_m + F_1 &= A_2 x_m^2 + y_m^2 + D_2 x_m + F_2 \\
\frac{2 A_1 x_m + D_1}{y_m} &= \frac{2 A_2 x_m + D_2}{y_m}.
\end{align*}
\] (2)

By eliminating $y_m$ in the fifth and sixth equations in (2), we obtain the system (3) of linear equations about $x_m$ as

\[
\begin{align*}
A_1 x_1^2 + y_1^2 + D_1 x_1 + F_1 &= 0 \\
(2 A_1 x_1 + D_1) T_{x_1} &= 2 y_1 T_{y_1} \\
A_2 x_2^2 + y_2^2 + D_2 x_2 + F_2 &= 0 \\
(2 A_2 x_2 + D_2) T_{x_2} &= 2 y_2 T_{y_2} \\
A_1 x_m^2 + D_2 x_m + F_1 &= A_2 x_m^2 + D_2 x_m + F_2 \\
\frac{2 A_1 x_m + D_1}{y_m} &= \frac{2 A_2 x_m + D_2}{y_m}.
\end{align*}
\] (3)

The coefficients $A_1$, $D_1$, $F_1$, $A_2$, $D_2$, and $F_2$ are expressed below explicitly as functions of the two end points and their associated tangents, in addition to $x_m$ which is the only unknown variable that is left as a freedom for the optimal bi-conic fitting.

\[
A_1 = \frac{T_{y_1} T_{x_2}(y_2^2 - y_1^2) - T_{x_1} T_{y_2}(2 x_1 y_1 - x_2 y_2) + T_{x_1} T_{x_2} y_1 + (T_{y_1} T_{x_2} y_2 - T_{y_2} T_{x_1} y_1)}{T_{y_1} T_{x_2}(x_m - x_1)(x_m - x_2)}
\]

\[
A_2 = \frac{T_{y_1} T_{x_2}(y_2^2 - y_1^2) - T_{x_1} T_{y_2}(2 x_1 y_1 - x_2 y_2) + T_{x_1} T_{x_2} y_1 + (T_{y_1} T_{x_2} y_2 - T_{y_2} T_{x_1} y_1)}{T_{y_1} T_{x_2}(x_m - x_2)(x_m - x_1)}
\]

\[
D_1 = \frac{2(T_{y_1} T_{x_2} x_2(y_2^2 - y_1^2) + T_{x_1} T_{x_2} x_1(y_2^2 - x_1 y_1) + (T_{y_1} T_{x_2} y_2 x_2 - T_{y_2} T_{x_1} y_1 x_2) x_m)}{T_{y_1} T_{x_2}(x_m - x_2)(x_m - x_1)}
\]

\[
D_2 = \frac{2(T_{y_1} T_{x_2} x_2(y_2^2 - y_1^2) + x_2(T_{y_1} T_{x_2} y_2 x_2 - T_{y_2} T_{x_1} y_1 x_2) + (T_{y_1} T_{x_2} x_2 y_2 - T_{y_2} T_{x_1} x_2 y_2) x_m)}{T_{y_1} T_{x_2}(x_m - x_2)(x_m - x_1)}
\]

\[
F_1 = \frac{T_{y_1} T_{x_2} x_1(y_2^2 - y_1^2) + x_1(y_1 T_{x_2} T_{x_2} y_2 - T_{y_2} T_{x_1} x_2 y_2) + x_1^2 T_{x_1} y_2 x_2^2 + x_1^2 T_{y_1} y_2 x_2^2 + x_1^2 - 2 x_1^2 T_{y_1} y_2 x_2}{T_{y_1} T_{x_2}(x_m - x_2)(x_m - x_1)}
\]

\[
F_2 = \frac{T_{y_1} T_{x_2} x_1(y_2^2 - y_1^2) + x_1(y_1 T_{x_2} T_{x_2} y_2 - T_{y_2} T_{x_1} x_2 y_2) + x_1^2 T_{x_1} y_2 x_2^2 + x_1^2 T_{y_1} y_2 x_2^2 + x_1^2 - 2 x_1^2 T_{y_1} y_2 x_2}{T_{y_1} T_{x_2}(x_m - x_2)(x_m - x_1)}
\]

All of them are rational linear functions about $x_m$; they can be simplified as:
\[
A_1 = \frac{h_1 + h_2 x_m}{g_1 + g_2 x_m}, \quad D_1 = \frac{h_3 + h_4 x_m}{g_1 + g_2 x_m}, \quad F_1 = \frac{h_5 + h_6 x_m}{g_1 + g_2 x_m},
\]
\[
A_2 = \frac{k_1 + k_2 x_m}{g_3 + g_4 x_m}, \quad D_2 = \frac{k_3 + k_4 x_m}{g_3 + g_4 x_m}, \quad F_2 = \frac{k_5 + k_6 x_m}{g_3 + g_4 x_m},
\]

and their derivatives with respect to \(x_m\) can be rewritten as:

\[
A'_1 = \frac{h_2 g_1 - h_1 g_2}{(g_1 + g_2 x_m)^2}, \quad D'_1 = \frac{h_4 g_1 - h_3 g_2}{(g_1 + g_2 x_m)^2}, \quad F'_1 = \frac{h_6 g_1 - h_5 g_2}{(g_1 + g_2 x_m)^2},
\]
\[
A'_2 = \frac{k_2 g_3 - k_1 g_4}{(g_3 + g_4 x_m)^2}, \quad D'_2 = \frac{k_4 g_3 - k_3 g_4}{(g_3 + g_4 x_m)^2}, \quad F'_2 = \frac{k_6 g_3 - k_5 g_4}{(g_3 + g_4 x_m)^2},
\]

where \(h_i, h_2, h_3, h_4, h_5, h_6, k_1, k_2, k_3, k_4, k_5, k_6, g_1, g_2, g_3\) and \(g_4\) are the expressions about \(x_1, y_1, x_2, y_2, T_{11}, T_{12}, T_{21}\) and \(T_{22}\) and can be regarded as constants.

![Fig. 4 C-type coaxial bi-conic arc spline](image1)

![Fig. 5 S-type coaxial bi-conic arc spline](image2)

Like circular biarc splines, a pair of coaxial bi-conics can deal with two types of shapes: C-type (see Figure 4) and S-type (see Figure 5). Thus our proposed coaxial bi-conic splines can fit an arbitrary generatrix flexibly by using either of C-type and S-type.

### 2.3. Local optimal coaxial bi-conic arc fitting

With different \(x_m\) in Eq. (4), we can construct different pairs of coaxial bi-conic arcs, all interpolating the same two points and their tangents; this thus calls for the optimal coaxial bi-conic fitting. Taking advantage of the fact that both the generatrix and our conics are \(x\)-monotone, let’s set the error measure between the generatrix and a pair of fitting bi-conic arcs to be the integral (between the two end points of the bi-conics) of the absolute difference in \(y\). The problem of finding the best \(x_m\) for the optimal bi-conic arcs fitting of a generatrix can then be formulated as

\[
E_{x_m}(\mathbf{x}(x_m)) = \left( \int_{x_1}^{x_2} ||C_1(x, x_m) - G(x)||^2 \, dx + \int_{x_m}^{x_2} ||C_2(x, x_m) - G(x)||^2 \, dx \right)
\]

where \(C_1(x, x_m)\) and \(C_2(x, x_m)\) are the explicit equations of the two coaxial bi-conic arcs, and \(G(x)\) is the explicit representation of the generatrix.

Initially, we tried the golden ratio searching method to determine the \(x_m\) for minimizing the error \(e(x_m)\) as in Eq. (6). Let \(x_m = x_1 + t(x_2 - x_1)\) \((0 \leq t \leq 1)\), we consider the errors \(e_1\) and \(e_2\) of \(e(x_m)\) corresponding to the two golden ratios \(t = t_1 = 0.382\) and \(t = t_2 = 0.618\), as shown in Figure 6. If \(e_1 > e_2\), the left subinterval [0, 0.382] is
removed from further consideration; otherwise, we delete the right subinterval [0.618, 1]. By recursively shrinking the interval, it eventually converges to a medium section point \( x_m \) as the size of final interval approaches zero (in practice, a small positive number less than the given threshold \( \varepsilon \)). However, simplicity notwithstanding, this \textit{golden ratio searching method} works well only when the error function \( e(x_m) \) has only one local minimum, which though is found by our experiments to be very rare in practice. As a result, the golden ratio searching method fails in many cases, as exemplified in Figure 7.

![Fig. 6 Finding the best \( x_m \) by golden ratio searching](image)

As a better alternative, we introduce the concept of \textit{convex combination} which, together with the least square fitting technique, finds a better \( x_m \) for minimizing the error \( e(x_m) \) in a more general situation. We call this method the \textit{LSF method} (least square fitting) and describe it in detail next.

**Convex combination**

By extending the domains of \( x \) of \( C_1: \ y = \sqrt{A_1 x^2 + D_1 x + E_1} \) and \( C_2: \ y = \sqrt{A_2 x^2 + D_2 x + E_2} \) from \([x_l, x_l] \) and \([x_m, x_m] \) to \([x_l, x_2] \) \((x_l < x_m < x_2)\), we employ convex combination of \( C_1 \) and \( C_2 \), namely \( C(\lambda, x) = (1-\lambda)C_1(x) + \lambda C_2(x) \) \((0 \leq \lambda \leq 1)\), to fit the generatrix \( G(x) \) in \([x_l, x_2] \). This convex combination removes the "hard" integral truncation form at \( x_m \) in Eq. (6) and converts Eq. (6) to a two-variable optimization problem as

\[
E_{\lambda, x_m} = \left\{ \int_{x_l}^{x_2} \left[ (1-\lambda)C_1(x, x_m) + \lambda C_2(x, x_m) - G(x) \right]^2 dx \right\}.
\]

As shown below, this approximation makes the LSF more tractable. We note here that the error between \( G(x) \) and the convex combination and differs slightly from the actual error between \( G(x) \) and the relevant quadric. To check whether this deviation is significant, we compared the number of bi-quadratics required to approximate different curves as computed by the LSF method as well as by a brute force fitting method using extremely fine sampling. In each case, this difference was less than 5%, indicating that this approximation works fairly well in practice.

**Least square fitting**

Ideally, to solve the minimization problem of Eq. (7), given the definition of \( G(x) \), the error function \( e(\lambda, x_m) \) should be expressed as a function of the boundary conditions at the two end points, in addition to the two parameters \( \lambda \) and \( x_m \), and the optimal solution can then be obtained by computing the roots of the two equations
\[ \frac{\partial \varepsilon(\lambda, x_m)}{\partial \lambda} = 0 \text{ and } \frac{\partial \varepsilon(\lambda, x_m)}{\partial x_m} = 0. \] Unfortunately, getting the exact analytical solution is not possible due to the complexity of the resulting system of equations. For instance, even for a simple cubic Bezier \( G(x) \), the final \( \varepsilon(\lambda, x_m) \) consists of more than 60 terms with both \( \lambda \) and \( x_m \) on the denominators of some fractions and inside some squares of root; due to the fractional and square-root terms, the expressions for \( \frac{\partial \varepsilon(\lambda, x_m)}{\partial \lambda} = 0 \) and \( \frac{\partial \varepsilon(\lambda, x_m)}{\partial x_m} = 0 \) are even more complex.

As an alternative, we approximate Eq. (7) by its discrete form and employ the least squares fitting method to find the best \( \lambda \) and \( x_m \) based on the discrete data. Specifically, let \( P_i(x_i, y_i) \) for \( i = 0, 1, 2, ..., N \) be \( N \) samples on the generatrix \( G(x) \), where the two end points \( P_0(x_0, y_0) \) and \( P_N(x_N, y_N) \) delimit the \( x \)-interval of the interpolating bicorons. The expanded discrete form of (7) then becomes:

\[ \varepsilon_{\lambda, x_m} (\lambda, x_m) = \frac{1}{2N} \sum_{j=0}^{N} \left[ (1 - \lambda) (A_j x_j^3 + y_j^3 + D_j x_j + F_i) + \lambda (A_j x_j^3 + y_j^3 + D_j x_j + F_i) \right]^2, \tag{8} \]

where it is noted that all \( A_1, D_1, F_1, A_2, D_2, \) and \( F_2 \) are functions of \( x_m \) as decided by Eq. (4).

An iterative procedure is used to optimize \( \varepsilon(\lambda, x_m) \) in Eq. (8) with respect to \( \lambda \) and \( x_m \). The iterations take the univariate search fashion. That is, given the \( k \)-th estimate \( x_m^{(k)} \) of \( x_m \), we obtain the \( k \)-th estimate \( \lambda^{(k)} \) of \( \lambda \) by minimizing \( \varepsilon(\lambda, x_m^{(k)}) \), and then fix \( \lambda^{(k)} \) to get the \((k+1)\)-th estimate \( x_m^{(k+1)} \) of \( x_m \). By alternating switching between \( \lambda \) and \( x_m \) and performing the resulting univariate optimizations, the intermediate solution \( (\lambda^{(k)}, x_m^{(k)}) \) converges to the optimum with increasing \( k \). For the univariate optimization of \( \varepsilon(\lambda^{(k)}, x_m^{(k)}) \), we derive the partial derivative

\[ \frac{\partial \varepsilon}{\partial \lambda} = \frac{1}{N} \sum_{j=0}^{N} \left[ (1 - \lambda) (A_j x_j^3 + y_j^3 + D_j x_j + F_i) + \lambda (A_j x_j^3 + y_j^3 + D_j x_j + F_i) \right] \frac{d}{d\lambda} \left[ (A_j - A_1) x_j^2 + (D_j - D_1) x_j + (F_2 - F_i) \right], \tag{9} \]

and then set it to 0; this has a direct solution:

\[ \lambda = \frac{-2 \sum_{j=0}^{N} \left( A_j x_j^3 + y_j^3 + D_j x_j + F_i \right) \left( A_j - A_1 \right) x_j^2 + (D_j - D_1) x_j + (F_2 - F_i) \right]}{\sum_{j=0}^{N} \left( A_j - A_1 \right) x_j^2 + (D_j - D_1) x_j + (F_2 - F_i) \right)}. \tag{10} \]

Next, for the univariate optimization of \( \varepsilon(\lambda^{(k)}, x_m^{(k)}) \), we differentiate it with respect to \( x_m \) and then set it to 0:

\[ \frac{\partial \varepsilon}{\partial x_m} = 2 \sum_{j=0}^{N} \left( (1 - \lambda^{(k)}) (A_j x_j^3 + y_j^3 + D_j x_j + F_i) + \lambda^{(k)} (A_j x_j^3 + y_j^3 + D_j x_j + F_i) \right) \left[ (1 - \lambda^{(k)}) \left( \frac{d}{d x_m} x_j^3 + \frac{d}{d x_m} x_j^3 + \frac{d}{d x_m} x_j + \frac{d}{d x_m} F_i \right) + \lambda^{(k)} \left( \frac{d}{d x_m} x_j^3 + \frac{d}{d x_m} x_j^3 + \frac{d}{d x_m} x_j + \frac{d}{d x_m} F_i \right) \right] = 0. \tag{11} \]
Unlike the case of \( \lambda \), solving Eq. (11) analytically for \( x_m \) becomes too formidably arduous, even though theoretically it is doable (the entire derivation occupies more than one page). We solve Eq. (11) numerically, using the secant method [8].

**Good initial value for the iteration**

Having a good initial \( x_m^{(0)} \) is crucial for fast convergence of the prescribed iterative procedure. Rather than taking an arbitrary value, say \( x_m^{(0)} = (x_0 + x_N)/2 \), here we propose a method for setting the initial value \( x_m^{(0)} \) as follows.

Let

\[
\Delta C(x_m) = C_1(x_m) - C_2(x_m) = (A_1 - A_2) \sum_{j=0}^{N} x_j^2 + (D_1 - D_2) \sum_{j=0}^{N} x_j + (F_1 - F_2) = 0,
\]

which leads to

\[
A_1 \sum_{j=0}^{N} x_j^2 + D_1 \sum_{j=0}^{N} x_j + F_1 = A_2 \sum_{j=0}^{N} x_j^2 + D_2 \sum_{j=0}^{N} x_j + F_2.
\]

By substituting \( A_1, D_1, F_1, A_2, D_2 \) and \( F_2 \) in Eq. (4) into Eq. (13) and letting

\[
m_1 = \sum_{j=0}^{N} x_j^2 \text{ and } m_2 = \sum_{j=0}^{N} x_j,
\]

we have

\[
\frac{w_1 + w_2 x_m^{(0)}}{g_1 + g_2 x_m^{(0)}} = \frac{w_3 + w_4 x_m^{(0)}}{g_3 + g_4 x_m^{(0)}},
\]

where

\[
w_1 = h_1 m_1 + h_3 m_2 + k_5, \quad w_2 = h_2 m_1 + h_4 m_2 + k_6, \quad w_3 = k_1 m_1 + k_2 m_2 + k_3, \quad w_4 = k_2 m_1 + k_3 m_2 + k_6.
\]

Equation (14) can be transformed into a quadratic equation about \( x_m \)

\[
Q_2 x_m^{(0)} + Q_3 x_m^{(0)} + Q_0 = 0,
\]

where

\[
Q_2 = w_2 g_4 - w_4 g_2, \quad Q_1 = (w_1 g_4 + w_2 g_3 - w_4 g_1 - w_3 g_2), \quad Q_0 = w_1 g_3 - w_3 g_1.
\]

By solving Eq. (15), we can get a good initial value of \( x_m^{(0)} \).

The following algorithm outlines the complete iterative procedure. Here, \( T \) is the user-specified maximum number of iterations allowed, and \( \varepsilon \) is the threshold of the relative error for controlling the iteration. Steps 14–16 deal with the special case when \( x^{(k)} \) is out of the scope \([x_0, x_N]\); we observe that in this case the optimal section-point \( x_m \) coincides with one of the two endpoints, and the one with the smaller error is selected. To ensure
numerical stability, we compute the error for all the iterated $x_m^{(k)} (k = 1, 2, ..., T)$ in Step 18, and return the $x_m$ with the smallest error.

Algorithm LSF_search

Input: $T$, $c_0$, $(x_j, y_j) (j = 0, 2, ..., N)$;
Output: $x_m$;

begin

$k \leftarrow -1$; $e_m \leftarrow +\infty$;

for $k$ from 0 to $T - 1$ do // iteration of $x_m^{(k)}$ and $x_n^{(k)}$

$k \leftarrow k + 1$;

if $k = 0$ then

compute the initial value $x_m^{(0)}$ according to (15);

else

compute $x_m^{(k)}$ according to (11);

end if;

compute $x_n^{(k)}$ according to (10);

if $x_m^{(k)}$ is not within $[x_0, x_N]$ then

compute the errors at $x_0 + \varepsilon$ and $x_N - \varepsilon$ respectively;

return the one with the smaller error;

end if;

compute error $e^*$ of the fitting BC arc at $x_m^{(k)}$;

if $e_m < e^*$ then

$e_m \leftarrow e^*$; $x_m = x_m^{(k)}$;

end if;

end for

while $(k > 0)$ and $|e(x_n^{(k)}), x_m^{(k)}| - e(x_n^{(k-1)}), x_m^{(k-1)}| > \varepsilon$ and $(k < T)$

return $x_m$;

end;
For comparison, we also implemented the golden ratio searching method and the brute force searching method. As illustrated by the example shown in Figure 7, the LSF method significantly outperforms the golden ratio searching method; in this particular example, the result of the LSF method is very close to the optimum (obtained by brute force). Further, even after the first iteration, the result of the LSF searching is already close to the optimum (Figure 7); thus it converges quite rapidly (only 3 iterations). Since the quality of the local fitting directly influences the final number of the bi-conic arcs used to approximate the generatrix, this indicates that at the same tolerance, the LSF method will generate less conic arcs than the golden ratio searching method. This is ratified by the data given in Table 1, where the generatrix used is shown in Figure 9.

<table>
<thead>
<tr>
<th>Tolerance</th>
<th>$10^{-2}$</th>
<th>$10^{-3}$</th>
<th>$10^{-4}$</th>
<th>$10^{-5}$</th>
<th>$10^{-6}$</th>
<th>$10^{-7}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Optimal fitting (obtained by brute force)</strong></td>
<td>22</td>
<td>34</td>
<td>74</td>
<td>159</td>
<td>343</td>
<td>738</td>
</tr>
<tr>
<td><strong>Golden ratio searching</strong></td>
<td>31</td>
<td>54</td>
<td>116</td>
<td>245</td>
<td>514</td>
<td>1130</td>
</tr>
<tr>
<td><strong>LSF searching method</strong></td>
<td>22</td>
<td>44</td>
<td>86</td>
<td>174</td>
<td>361</td>
<td>759</td>
</tr>
</tbody>
</table>

**2.4. Global coaxial bi-conic fitting**

The fitting method presented in the previous subsection provides a locally optimal $GC^1$ coaxial conic interpolator, given two arbitrary endpoints and their associated tangential vectors. Naturally, the other concern is to globally fit the generatrix by a sequence of quadric-pairs within a specified error tolerance $e$. We introduce an adaptive error-driven bi-section marching method that uses the bi-quadratic fitting method. The error metric used this time is the Hausdorff distance between the generatrix and the conic arcs (see Fig. 8). The algorithm is detailed below.

**Algorithm Global_fitting**

**Step 1** $x_1 \leftarrow x_p$, $x_2 \leftarrow x_p$, $x_3 \leftarrow x_p$.

**Step 2** fit the generatrix on the $x$-monotonic interval $[x_1, x_2]$ with a bi-conic arc interpolator using algorithm LSF;

**Step 3** calculate the error $d$ between the generatrix and the fitting bi-conic arc interpolator;

**Step 4** if $d > e$, shrink the fitting interval, set $x_3 \leftarrow (x_1 + x_2)/2$, goto **Step 2** (to fit it again on the new $[x_1, x_3]$);

**Step 5** if ($d < e$) and $(x_2 \neq x_p)$, enlarge the fitting interval by setting $x_2 \leftarrow (x_1 + x_2)/2$, and goto **Step 2**;

**Step 6** /* $d \leq e$ */

(1) output the fitting interval $[x_1, x_2]$ and its corresponding coaxial bi-conic arc interpolator;

(2) if $x_2 = x_p$,

   goto **Step 7**;

   else

   $x_1 \leftarrow x_3$;  $x_2 \leftarrow x_p$;

   goto **Step 2** for the next bi-conic arc fitting;

endif;

**Step 7** Exit.
(1) if \( d = e \), start next bi-conic fitting

(2) if \( d > e \), shrink and refit

(3) if \( d < e \), stretch and refit

Fig. 8 Global bi-conic fitting based on bi-section division and greedy stretching

To compare our proposed bi-conic revolute quadric decomposition with other quadric decomposition methods, we also implemented the truncated cone subdivision method and the single conic fitting method, and applied all the three to a number of test examples; one example is shown in Figure 9. For the generatrix in Figure 9, the number of the basic fitting elements (coaxial revolute quadrics for the conic fitting and truncated cones for the truncated cones fitting) are tabulated in Table 2, categorized based on different fitting error tolerance \( e \). The table shows that both revolute quadric decomposition schemes, namely single-conic and bi-conic, result in much less basic fitting elements than the truncated cone decomposition. Further, the bi-conic fitting method outperforms the single conic fitting method by a large margin. In general, this reduction in basic fitting elements becomes more pronounced once the error tolerance \( e \) gets tighter. For instance, in the example of Figure 9, the ratio of the number of basic fitting elements of the truncated cones method vs. the single conic fitting vs. the bi-conic fitting is 1.27:1.09:1 for a large \( e = 10^{-2} \), 5.29:1.76:1 when \( e = 10^{-5} \), and 12.14:1.51:1 when \( e = 10^{-7} \). Since downstream applications, especially intersections, often require a tighter \( e \) for more accurate results, the proposed bi-conic fitting method is thus clearly advantageous over the other two decomposition methods.

Fig. 9 Global coaxial bi-conic spline fitting of a general generatrix
Table 2 Comparison among the three decomposition methods

<table>
<thead>
<tr>
<th>Number of basis fitting elements</th>
<th>Tolerance</th>
<th>$10^2$</th>
<th>$10^3$</th>
<th>$10^4$</th>
<th>$10^5$</th>
<th>$10^6$</th>
<th>$10^7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Truncated cones (linear approximation)</td>
<td>56</td>
<td>182</td>
<td>581</td>
<td>1842</td>
<td>5829</td>
<td>18436</td>
<td></td>
</tr>
<tr>
<td>Revolute quadrics (single conic arcs fitting)</td>
<td>48</td>
<td>98</td>
<td>274</td>
<td>577</td>
<td>1277</td>
<td>2299</td>
<td></td>
</tr>
<tr>
<td>Bi-conic Revolute quadrics (bi-conic arcs fitting)</td>
<td>44</td>
<td>86</td>
<td>172</td>
<td>348</td>
<td>722</td>
<td>1518</td>
<td></td>
</tr>
</tbody>
</table>

3. Applications

The bi-conic subdivision scheme provides an excellent basis for implementing robust and efficient algorithms for a variety of intersection problems. We will use two representative intersection problems, namely the line/surface intersection and the surface/surface intersection, to demonstrate this point. Specifically, it is shown in this section that by using our proposed decomposition method we can design intersection algorithms for these problems which are far more efficient than some representative existing approximation schemes.

3.1. Line/surface intersection

Line/surface intersection, used in ray tracing, has been a focal point of extensive research in computer graphics due to its importance in graphics rendering. For a good summary of ray tracing algorithms over the last 30 years, see [21]. In our special setting of ray tracing a surface of revolution, there exist a number of algorithms [1, 4, 7, 24, 43, 48]. Among them a classical work is Kajiya's algorithm [24] in which he first reduced the ray/surface intersection to the intersection between a quadratic curve and an arbitrary curve in the cutting plane of the ray and the revolute axis of the surface, and then employed a strip tree [5] to compute the intersections of the two curves. Another simple yet popular method for ray tracing a surface of revolution is to subdivide the surface into a set of coaxial truncated cones along the revolute axis which then reduces the ray/surface intersection into the intersection between the ray and truncated cones (for example see [1, 43]). Both of them will be used as benchmarks for the comparison. In the following, we entail the key components of our ray tracing algorithm using the proposed revolute quadric decomposition scheme.

Cylindrical bounding shell

The bounding volume is an important technique for fast determination of surface intersection. We adopt a cylindrical bounding shell (CBS) for surfaces of revolution (instead of a single bounding cylinder) which uses two bounding cylinders to enclose a surface of revolution both internally and externally, as shown in Fig. 10, where $r$ and $R$ are the inner and outer radii, respectively the maximum and minimum distances of the generatrix $C$ to the revolute axis. Our CBS idea is motivated by two considerations. Firstly, a cylindrical bounding shell encloses a surface of revolution more tightly than a traditional bounding box, a bounding sphere, or a single bounding cylinder. Secondly, the intersection determination for a ray and a cylinder is done geometrically without solving quadratic equations.

Intersection determination of ray-CBS

Intersection determination of a ray and a cylindrical bounding shell can be done geometrically by comparing between $d$ (the distance between the revolute axis and the ray) and $r$ (the radius of the inner bounding cylinder) or
$R$ (the radius of the outer bounding cylinder), as illustrated in Fig. 10. Basically, the following steps are performed for this determination:

1. compute the following three distances
   
   $d \leftarrow$ distance between the ray and the revolute axis;
   
   $d_a \leftarrow$ distance between the revolute axis and the point $p_a$ (which is the intersection between the ray and the top base face of the cylinder);
   
   $d_b \leftarrow$ distance between the revolute axis and the point $p_b$ (which is the intersection between the ray and the bottom base face of the cylinder);

2. if $(\min \{d, d_a, d_b\} < r)$ or $(d > R)$ then NO intersection;

3. otherwise, subdivide the surface and check each sub-division.

![Fig. 10 Intersection between a ray and a CBS](image1)

![Fig. 11 Binary cylindrical bounding shell tree](image2)

**Binary cylindrical bounding shell tree**

Assuming a surface of revolution is subdivided into $n$ truncated revolute quadrics, a brute force intersection searching would require intersecting the ray with every one of them. A hierarchical data structure, called *binary cylindrical bounding shell tree* (BCBST), is adopted to reduce the computational effort. As shown in Fig. 11, the root of this tree corresponds to the CBS of the surface of revolution itself, and each leaf bounds a single subdivided revolute quadric. Intersection determination is done by a recursive binary search on the BCBST of the surface. If the ray does not intersect $CBS_0$ at the root, it certainly has no intersection with the surface. Otherwise, the intersection tests are continued on its two children $CBS_{11}$ and $CBS_{12}$ respectively. Note that each node in the BCBST has a corresponding BCS; these are computed only once, and contribute to the pre-processing cost.

The efficiency of intersection computation is largely decided by ray intersection searching in BCBST. To facilitate this search, we introduce the term *valid revolute axial intersection interval* (VRAII) which is an interval on the revolute axis whose corresponding revolute quadrics are deemed to be potential candidates intersected by the given ray. The idea is to reduce the number (size) of VRAIIs as small as possible before the real ray-quadric intersections are calculated. This is achieved by clipping the cylindrical bounding shell, as described next.
Ray clipping cylindrical bounding shell

Let us examine how a ray pierces through a surface of revolution and its cylindrical bounding shell CBS ($r<d<R$). As demonstrated in Fig. 12, the ray first approaches the outer bounding cylinder at $P_1$; after piercing through the surface of revolution, it reaches the inner bounding cylinder at $P_2$; this is the first round of piercing through the cylindrical bounding shell. Next, the ray approaches the inner bounding cylinder again at $P_3$, then it goes through the surface of revolution again, and finally it approaches the outer bounding cylinder at $P_4$. Assuming the revolute axis of the surface of revolution to be the $z$-axis and the $z$-coordinates of the four intersection points $P_1$, $P_2$, $P_3$ and $P_4$ to be $z_1$, $z_2$, $z_3$, $z_4$, respectively, it can be seen that only the revolute quads within $z$-intervals $[z_1, z_2]$ and $[z_3, z_4]$ can possibly have intersections with the ray. Intervals $[z_1, z_2]$ and $[z_3, z_4]$ are noted as potential intersection intervals (PII) and the corresponding reduction procedure is referred to as cylindrical bounding shell clipping. The first round CBS clipping reduces PII from $[z_{\min}, z_{\max}]$ to $[z_1, z_2]$ and $[z_3, z_4]$, as shown in Fig. 12 where the root of BCBST $T$ (light blue triangle) corresponds to interval $[z_{\min}, z_{\max}]$. The two yielding intervals $[z_1, z_2]$ and $[z_3, z_4]$ correspond to the two subtrees $T_1$ and $T_2$ (two smaller red triangles). We can then locate all the revolute quads within $[z_1, z_2]$ and $[z_3, z_4]$ by using the standard range search method. Thus, we jump from root $T$ directly to $T_1$ and $T_2$. The searching depth for the VRAIs in BCBST is decreased from $O(\log n)$ to $\max\{\text{depth of } T_1, \text{depth of } T_2\}$. The VRAIs can be obtained by refining PII of a ray and a CBS recursively with the proposed CBS clipping technique.

The reduction extent of PII by CBS clipping depends on the configuration of the ray and the CBS. The CBS clipping can only reduce the PII and shorten the searching depth of CBS tree in the case of good relative orientations and positions between a ray and a CBS. Similar to Bezier clipping technique, a 70% criterion is also employed in our CBS clipping: if the CBS clipping can reduce the size of PII by more than 70%, then it is carried out and the intersection searching goes in a 'leap' way; otherwise, we move to the two children to do the further intersection determination in stepwise way. In our method, thus, the two ways are interleaved for the efficiency of intersection searching. The algorithm SearchVRAII given below entails this combined search for intersection determination. In SearchVRAII, the variable node stands for the current node, corresponding to a hierarchical CBS; initially, it is the root in BCBST; the variable range is the current VRAI. RayClipCBS is the procedure of
CBS clipping by a given ray (as illustrated in Fig. 12); \textbf{Overlap}(range1, range2) returns the overlapped parts of range1 and range2; \textbf{distance}(ray, node) represents the distance between a ray and the CBS at node node in BCBST; finally, \textbf{RangeSearch}(range, node) is a standard function in computational geometry, which search out all the RQs within range in the subtree rooted at node.

Algorithm \textbf{SearchVRAII}(ray, node, range)

1. Begin
2. if node is a leaf of BCBST
3. 
\hspace{1em} I = \textbf{IntersectRayQuadric}(ray, node);
4. 
\hspace{1em} return I;
5. end if;
6. range\_old = range;
7. range\_clipping = \textbf{RayClipCBS}(ray, node);
8. range\_new = \textbf{Overlap}(range\_old, range\_clipping);
9. if range\_new is empty
10. 
\hspace{1em} return nil;
11. end if;
12. if range\_new < 70\% of range\_old
13. 
\hspace{1em} I = \textbf{RangeSearch}(range\_new, node);
14. 
\hspace{1em} return I;
15. else
16. 
\hspace{2em} I_{left} = \textbf{SearchVRAII}(ray, node.chil\_left, range\_new);
17. 
\hspace{2em} I_{right} = \textbf{SearchVRAII}(ray, node.chil\_right, range\_new);
18. 
\hspace{2em} if distance(ray, I_{left}) < distance(ray, I_{right})
19. 
\hspace{3em} return I_{left}.
20. else
21. 
\hspace{3em} return I_{right}.
22. end if;
23. end if;
24. end.

3.2. Surface/surface intersection

Surface/surface intersection is perhaps the most basic and most important geometric computation required in geometric modelling, CAD/CAM, or any other surface-based modelling systems. However, it keeps being a very challenging task to compute the intersection curves robustly, accurately and efficiently for two arbitrary surfaces, even for two simple surfaces, e.g. quadric surfaces [37]. For the special case of intersecting two surfaces of revolution, two intersection algorithms stand out – the circle-decomposition + zero-set method and the truncated cones method – and they will be our benchmarks for comparison. The first method represents a surface of revolution by a set of circles along the revolute axis and converts the intersection between two surfaces to the intersections between two families of circles. Heo et al [22] gave an elegant formula for this conversion, where they reformulated the intersection problem of two surfaces of revolution as a simpler problem of searching a
bivariate zero-set \[\|f(u) - g(v)\| = |r(v)|\] in the 2D parameter space in [25]. However, there is no general analytical solution to this zero-set problem when the degree of the generatrix is higher than 2, thus numerical solutions have to be used. The second method approximates a surface of revolution by a set of coaxial truncated cones and reduces the surface/surface intersection problem to the intersection between two families of truncated cones [26].

Our revolute quadric decomposition reduces the intersection problem of two surfaces of revolution to the intersection problem of two families of revolute quadrics. Although two surfaces of revolution are decomposed into \(m\) and \(n\) revolute quadrics respectively, only a small subset of these have potential intersections in practical cases. The notion of **valid revolute axial intersection intervals** (VRAII) of two surfaces of revolution is introduced to exploit this and improve the efficiency of intersection problems (see Fig. 15). Basically, there are four major steps for computing the intersection curves of two surfaces of revolution (RSIC): (i) estimating VRAII of two surfaces of revolution; (ii) computing the intersection curves of two revolute quadrics (RQIC) for each pair of overlapping revolute quadrics within VRAII; (iii) trimming RQIC by its four bounding circles respectively to obtain a bounded RQIC segment; and finally (iv) merging all the individual RQIC segments into the final intersection curve, RSIC. In the following, we describe these steps in some details.

**Estimating VRAII of two surfaces of revolution**

Consider two surfaces of revolution \(R_1\) and \(R_2\) (Figure 15). Construct the inner bounding cylinder for each (\(C_{in}\) and \(D_{in}\)), and the outer bounding cylinders (\(C_{out}\) and \(D_{out}\)), as shown in Figure 16. Their potential intersection intervals can be computed as follows. \(C_{p1}\) and \(C_{p2}\) are the two intervals within \(C_{ext}\) but outside \(C_{int}\). \(D_{p1}\) and \(D_{p2}\) are two intervals within \(D_{ext}\) but outside \(D_{int}\). Thus, all the revolute quadrics of \(R_1\) and \(R_2\) can be put into three categories:

- **non-intersection domains**: the revolute quadrics of \(R_1\) outside \(C_{ext}\) cannot intersect the revolute quadrics of \(R_2\) outside \(D_{ext}\); we thus can ignore these pairs of revolute quadrics for intersection;
- **definite intersection domains**: the revolute quadrics of \(R_1\) within \(C_{int}\) definitely have intersections with some of the revolute quadrics of \(R_2\) within \(D_{int}\);
- **potential intersection domains**: the revolute quadrics of \(R_1\) within \(C_{p1}\) and \(C_{p2}\) may either intersect the revolute quadrics of \(R_2\) within \(D_{p1}\) and \(D_{p2}\) or not, in this case, we construct new cylindrical bounding shells for them, and compute the intersection intervals again for further determination. By refining this procedure recursively, we can get the final VRAII of the two surfaces of revolution.

![Fig. 15 VRAII of two surfaces of revolution](image1)

![Fig. 16 Estimating VRAII by intersecting two CBSs](image2)
Computing $RQIC$

Computing the $RQIC$ of two revolute quadrics is a fundamental computation for the final $RSIC$. Levin presented a method for computing the intersection curves of two general quadratic surfaces (QSIC) in [28, 29] based on finding a ruled quadric $R(t, t)$ in the pencil of two quadrics. Goldman [27] simplified this method for the special case of intersecting two RQs; this modified method is chosen to compute $RQIC$ in our algorithm.

Computing $RSIC$

The algorithm ComputingRSIC ($R_1$, $R_2$) given below outlines the procedure of computing the $RSIC$ between two series of revolute quadrics within a given VRAII. It involves first computing all the individual $RQIC$s and then trimming and merging them as a group of independent components that form the branches of the final $RQIC$. The merge step in the algorithm checks against all the possible topological scenarios, including coincident and degenerate intersection cases, to ensure the integrity of each branch of the $RSIC$; the resourceful existing topological analyses on $RSIC$ (see for example [42, 44, 45, 46, 47]) are utilized by us for this checking.

Algorithm ComputingRSIC ($R_1$, $R_2$)

Input: two series of revolute quadrics $RQ_i$ (for $R_1$) and $RQ_j$ (for $R_2$) within VRAII;
Output: list of branches (loops) of $RSIC$;

1. Begin
2. for each revolute quadric $RQ_i$ of $R_1$ within VRAII ($i = 1, 2, ..., m$);
3. for each revolute quadric $RQ_j$ of $R_1$ within VRAII ($j = 1, 2, ..., n$);
4. begin
5. if $RQ_i$ has intersection with $RQ_j$
6. compute the bounded $RQIC$ of $RQ_i$ and $RQ_j$ (trimming $RQIC$);
7. if $RQIC$ has no intersection with its four bounding circles $C_{1,i}, C_{2,i}, C_{2,i}, C_{2,i}$
8. forms an independent loop by itself;
9. else if both $RQIC_{i,j}$ and $RQIC_{i,j}$ have intersection with $C_{1,i}$ at the same endpoint(s)
10. merge $RQIC_{i,j}$ into the same branch with $RQIC_{i,j}$
11. else if both $RQIC_{i,j}$ and $RQIC_{i,j}$ have intersection with $C_{2,i}$ at the same endpoint(s);
12. merge $RQIC_{i,j}$ into the same branch with $RQIC_{i,j}$
13. else if both $RQIC_{i,j}$ has no intersection with both $C_{1,i}$ and $C_{2,i}$
14. create a new branch beginning with $RQIC_{i,j}$;
15. else if $RQIC_{i,j}$ has no intersection with both $C_{1,i}$ and $C_{2,j}$
16. close the current branch with $RQIC_{i,j}$
17. else
18. create a new branch begin with $RQIC_{i,j}$
19. end if;
20. end for;
21. end for;
22. check all the active branches during merging procedure and store them as a group of independent branches
23. if a branch begins and ends with a single-branch $RQIC$
24. identify it as a closed loop;
25. endif;
26. End.
4. Experimental and comparison results

We have implemented the proposed revolute quadric (RQ) decomposition method, as well as the two associated intersection algorithms for line/surface and surface/surface intersections as described in section 3. To compare our algorithms with others, we also implemented some representative existing intersection algorithms. Specifically, for line/surface intersection, the Kajiya's method [24] and the truncated cones (or TC for brevity) method [1] were implemented; and for surface/surface intersection, the Kim's method [25], which is based on circle decomposition and zero-set searching, and also the truncated cones method [26] were implemented. A number of test examples were then taken for the experiments, of which two for each intersection type are selected and given next.

4.1. Experimental results of line/surface intersection

In our experiments, a scene consists of objects made of surfaces of revolutions and is rendered on the screen by ray-tracing the surfaces. The hardware configuration consists of a 500x500 pixels screen, a Pentium IV CPU (1GHz), and 512MB RAM under Windows ME. Four factors are considered in the performance data for comparison: (1) the rendering time, (2) the rendering time measured as a multiple of the time taken by the Kajiya's method, (3) the occupied memory, and (4) the memory usage as a multiple of the space required by our implementation of the Kajiya's method.

The first example contains 32 bottles with various orientations and positions. Fig. 13 is the picture obtained by ray casting these multiple bottles without shadows and reflections. In the second example, the scene is an outdoor restaurant patio consisting of three round tables, six chairs, six cups, three bottles, three bowls on the tables, and the patio rails supported by forty round rail fixtures; totally, the scene contains one hundred surfaces of revolution. Fig. 14 displays the picture generated by ray tracing the patio with one shadow and three reflections.

![Fig. 13 Ray casting 32 bottles](image1)

![Fig. 14 Ray tracing a restaurant patio](image2)

The performance data for both our algorithm and the other two are given in Table 3 and 4 for the two examples respectively. As revealed in Tables 3, the RQ method runs favorably compared to the other two methods
(1/8 ~ 1/7 of the running time by Kajiya's method and 1/4~1/3 of that of the TC method). With regards to the occupied memory, the RQ method requires 32KB, which is about only 1/58 of the required memory of the other two methods. As shown in Tables 4, the running time of the RQ method performs significantly better than other two methods (1/9 ~ 1/8 of the running time by Kajiya's method and 1/4~1/3 of that of the TC method). With regards to the occupied memory, the RQ method requires 39.14KB, which is about 1/131 of the required memory (5177KB) of the other two methods. In other words, the RQ method in this example runs approximately an order of magnitude faster than the other two methods; at the same time, the memory requirements are reduced by over two orders of magnitude.

<table>
<thead>
<tr>
<th>Method</th>
<th>Running time (Seconds)</th>
<th>Ratio of running time to Kajiya's method</th>
<th>Occupied memory (KB)</th>
<th>Ratio of memory to Kajiya's method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kajiya's</td>
<td>20.62</td>
<td>1 : 1</td>
<td>1857.75</td>
<td>1 : 1</td>
</tr>
<tr>
<td>TC</td>
<td>8.90</td>
<td>1 : 2.316</td>
<td>1860.50</td>
<td>1 : 0.904</td>
</tr>
<tr>
<td>RQ</td>
<td>3.13</td>
<td>1 : 6.587</td>
<td>32.25</td>
<td>1 : 57.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Running time (Seconds)</th>
<th>Ratio of running time to Kajiya's method</th>
<th>Occupied memory (KB)</th>
<th>Ratio of memory to Kajiya's method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kajiya's</td>
<td>126.01</td>
<td>1 : 1</td>
<td>5163.61</td>
<td>1 : 1</td>
</tr>
<tr>
<td>TC</td>
<td>48.57</td>
<td>1 : 2.592</td>
<td>5177.67</td>
<td>1 : 0.904</td>
</tr>
<tr>
<td>RQ</td>
<td>15.67</td>
<td>1 : 8.042</td>
<td>39.14</td>
<td>1 : 131.91</td>
</tr>
</tbody>
</table>

### 4.2. Experimental results of surface/surface intersection

Three criteria are used as yardsticks for comparison: (i) the computing time of the RSIC, (ii) the occupied memory, and (iii) the number of basic segments comprising the same RSIC (i.e., for the TC method a basic segment is the intersection curve between two truncated cones, or TCIC, whereas in the case of RQ method it is a RQIC). In the first example, the two surfaces to be intersected are two vases as shown in Fig. 17; the resulting RSIC is composed of three branches, one open and two closed. The second example comes from the well-known two-chalices picture which appears on the cover of Journal of CAGD, with two fairly complex shaped generatrices and an intersection curve of complicated topology and geometry. Their performance data on running time and memory requirement as well as the comparisons are shown in Table 5 and Table 6.
Fig. 17 RSIC of two vase-like surfaces with three branches

Fig. 18 RSIC of two chalices

Table 5 Comparison on performance data on computing RSIC of two vase-like surfaces

<table>
<thead>
<tr>
<th>Tolerance</th>
<th>Kim’s method</th>
<th>TC method</th>
<th>RQ method</th>
<th>Ratio of Kim : TC : RQ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time(s)</td>
<td>Mem.</td>
<td>Time(s)</td>
<td>Mem.</td>
</tr>
<tr>
<td>$10^{-1}$</td>
<td>0.555</td>
<td>243K</td>
<td>0.014</td>
<td>5.28K</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>1.790</td>
<td>262K</td>
<td>0.042</td>
<td>16.2K</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>5.260</td>
<td>543K</td>
<td>0.476</td>
<td>67K</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>7.865</td>
<td>1.2M</td>
<td>1.247</td>
<td>179K</td>
</tr>
</tbody>
</table>
Table 6 Comparison of performance data on computing RSIC of two chalices

<table>
<thead>
<tr>
<th>Tolerance</th>
<th>Kim’s method</th>
<th>TC method</th>
<th>RQ method</th>
<th>Ratio of Kim : TC : RQ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time(s)</td>
<td>Mem.</td>
<td>Time(s)</td>
<td>Mem.</td>
</tr>
<tr>
<td>$10^{-1}$</td>
<td>1.130</td>
<td>253K</td>
<td>0.019</td>
<td>17.8K</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>2.375</td>
<td>975K</td>
<td>0.251</td>
<td>58.9K</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>6.180</td>
<td>2.2M</td>
<td>2.080</td>
<td>178K</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>21.50</td>
<td>4.8M</td>
<td>14.19</td>
<td>528K</td>
</tr>
</tbody>
</table>

From the data for the two examples (Tables 5-6), we see that the RQ method yields a computing time advantage ranging from 10.8 to over 800 times faster than the other two methods. At the same time, the memory taken up by the RQ method is significantly less than the Kim’s method, and also markedly less than the TC method. When using the Kim’s method, the fundamental zero-set searching procedure demands finding all the points densely distributed along the RSIC for correct topologic analysis and also for rendering, with each point requiring a numerical solution at high precision. As a result, the Kim’s method computes the RSIC much more slowly than the other two. Judging by the third criterion, the number of basic segments comprising the intersection curve, we see from Table 7 that the number of RQIC is much smaller than the number of TCIC in both examples. As also revealed in table 7, the ratio of TCIC vs. RQIC tends to be larger as the tolerance becomes tighter. Moreover, the basic segments RQICs are GC½ connected everywhere except at some finite critical points, while the TCICs are only piecewise GC².

Table 7 Comparison of number of TCIC vs. RQIC for RSIC of two vases and two chalices

<table>
<thead>
<tr>
<th>Tolerance</th>
<th>Two Vases</th>
<th>Two Chalices</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of TCIC</td>
<td>Number of RQIC</td>
</tr>
<tr>
<td>$10^{-1}$</td>
<td>13</td>
<td>9</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>40</td>
<td>28</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>165</td>
<td>58</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>441</td>
<td>125</td>
</tr>
</tbody>
</table>

5. Conclusion

A new algorithm for subdivision of surfaces of revolution is presented in this paper. The method provides a robust technique for approximating complex generatrix curves using a series of bi-quadratics. Our conic fitting scheme differentiates from other conic approximation methods in that it uses a coaxial bi-conic interpolator, rather than a single conic, and adopts a novel convex combination technique to seek the best coaxial bi-conic interpolator for a given portion of the generatrix. This union of bi-conic fitting and optimal convex combination helps achieve significant reduction of quadrics needed to accurately approximate a surface of revolution.

We also presented two applications for the subdivision scheme. The first is a fundamental operation in computer graphics, namely finding intersection of a surface (of revolution) with a ray. A new type of bounding volume, the coaxial cylindrical shell, is created. In conjunction with a binary tree data structure, the BCBST, and simple geometric clipping, these methods lead to a robust and efficient (in computing time and memory)
implementation. The second application is in solid modeling, that is, computing the intersection curve(s) of two surfaces of revolution. Again, due to the efficiency of the bi-quadratic subdivision scheme, as well as the use of cylindrical bounding shells, hierarchical storage, and geometric clipping, the proposed method provides significant performance improvements over existing methods. For both applications, competing methods that have been reported by other researchers were implemented to allow us to make real comparisons. For all comparisons that we have made, the bi-quadratic subdivision scheme improves significantly over competing methods in computing time, and achieves this using significantly less occupied memory.

Dupont et al [13, 14] provided a near-optimal parameterization for the intersection curves of two implicit quadrics with rational coefficients. If the generatrix is represented as a NURBS curve, all the sampling points on the generatrix should be rational numbers. Then our construction of coaxial bi-conic arc splines always produces rational numbers, and thereby, always produces piecewise revolute quadrics with rational coefficients. This can be employed to further improve the robustness and accuracy of our algorithm.

Potential applications of the proposed algorithm include some computer games that use a large number of surfaces of revolution, virtual manufacturing and assembly environments where components in the form of surface of revolution usually dominate, kernel utilities for solid modelers, etc. With the emergence of special hardware for ray-quadratic intersection (for example, see [30, 31]), our algorithm, which is based on revolute quadrics fitting, is also very appealing to real time rendering of complex scenes involving a large number of surfaces of revolution, where the computational time is critical.

6. References


