A Knowledge-Guided Approach to Line NURBS Curve Intersection

Khairan Rajab, Najran University, Saudi Arabia, khairanr@gmail.com

Les A. Piegl, University of South Florida, USA, les@piegl.com

ABSTRACT

This paper presents a robust algorithm to solve the line-curve intersection problem used frequently in design, manufacturing, graphics, art, etc. A global solution is proposed, i.e. all the intersections are found and computed to high accuracy requirements. The emphasis is on robustness, reliability and to handle geometric as well as numerical anomalies. The main thrusts of the method lie in the use of a knowledge-guided NURBS system, a tight biarc decomposition and proper pre- and post-processing of the entities as well as the intersections. All these contribute to achieve a high level of reliability: the method is immune to such cases as tangential intersections, inflection points, or line-curve overlaps. The intersection points produce various relationships that are recorded in the knowledge-guided system so that all the results are reproducible in the receiving system, should the intersection point be recomputed with a different level of accuracy.

Keywords: intersection, robustness, tolerancing, knowledge-guided systems, NURBS.

1. INTRODUCTION

Intersection algorithms have been the subjects of intensive research since the early days of modern CAD/CAM. The majority of intersection methods published in the literature deals with parametric entities [1–3,8,11–13,15,17,18,27,28,31], subdivision techniques [5,16,30], algebraic methods [14,29], special entity types such as natural quadrics [10,19], evolutionary algorithms [7], or computational geometry techniques [4]. A few honorable attempts at robustness revolve around those of solid modeling [6] or the use of interval arithmetic [9].

The line segment has played an important role in engineering design; it is used for two dimensional design, trimming off excess parts of curves meeting with straight edges, cutting parts of 2D drawings (potentially meeting with anomalies), performing inside/outside tests using ray shooting, processing 2D objects via scan-line methods, etc. Another important application is in nuclear engineering where the flying path of a neutron needs to be intersected with curves fast and efficiently (after the accident in Fukushina, nuclear engineers are looking at non-standard geometries represented by NURBS). Although general curve-curve intersection methods could handle lines, our experience shows that special purpose methods are by far more reliable and accurate.

It is our belief that the issue of robustness requires a whole lot more than just mathematics. The lack of robustness comes from many sources:

- **Geometric uncertainties:** mostly the result of lack of knowledge about the entities, i.e. we are not certain what the intersections are due to not seeing and understanding the type of the curve and the relationships between the entities.
- **Geometric anomalies:** cases like cusps, tangent points, overlapping entities, degenerate cases can make the numerical code fail.
- **Tolerance inconsistencies:** during the course of the algorithm a myriad of tolerances, e.g. manufacturing, parallel, perpendicular, angular, algebraic, etc., are used that may have no relationship to one another, e.g. how to adjust a tolerance used in an algebraic equation to ensure the results are accurate to a model space manufacturing tolerance.
- **Numerical/imprecise computation:** round-off errors that propagate through the system and cause inaccurate results, non-convergence, singularity, etc., create a chaos in most numerical code used in CAD/CAM systems.
- **Inappropriate mathematics:** although math is nice and well in itself, the implementation of different techniques or formulae produces

spectacularly different results from the point of view of accuracy, speed and reliability.

- **Inappropriate error bounds/measures:** almost all numerical techniques require some kind of error estimates or bounds on the error. There remain a few challenging problems such as parametric vs. geometric errors or computing tight error bounds to avoid data explosion.
- **General or special purpose algorithms:** general purpose methods can be applied to many problems, they are easy to write, however, maintenance can be a nightmare as they tend to fail frequently and can be quite slow. On the other hand, special purpose algorithms work only on specific problems, e.g. cylinder-cylinder intersection, they are tedious to write, however, they are fast, robust and easy to maintain.

The algorithm presented in this paper satisfies the following robustness requirements:

- **Consistent tolerancing:** it uses only *one* type of tolerance, a model space point coincidence tolerance. No other tolerances are allowed.
- **Global solution:** it finds *all* intersections regardless of anomalies or geometric complexity.
- **Accuracy:** the intersections are found to within engineering tolerances (we use 10^{-6} throughout the algorithm).
- **Performance:** the algorithm is robust and computes all intersections in real time.
- **Bounded geometry:** curve and line segments are intersected, i.e. the intersections are properly clipped or avoided if they fall outside the arc.
- **Commercially verified:** the algorithm is implemented with KGNurbs, a derivative of a commercial NURBS kernel created by the second author.
- **Data exchange:** the intersections can be reproduced at the receiving end as KGNurbs builds a knowledge base that allows design replay.

Our decades of experience in geometric computing has taught us a lesson: the more we know about the entities we compute with, the better the robustness of our algorithms. During the days of engineering drawings where intersections were computed by hand, everything seemed more reliable: the engineer could *see* the entities, *knew* where they were and what the intersections must be, and had the *experience* to construct the intersection points or the curves. Intersection algorithms tend to be blind: they are searching for something they know nothing about. Our biologically inspired knowledge-guided system [22–25] relies on a knowledge base (basic knowledge about the entities combined with a sophisticated relationship graph) and knowing what the entities are and how they are related, e.g. seeing their relative positions. Our experience has demonstrated that special purpose intersection algorithms are far superior to general ones enhancing the robustness immensely.

The organization of the paper is as follows. Section 2 formulates the problem mathematically. The main part of the paper is presented in Section 3 where the details of the method are given after an overview of the algorithm. Section 4 covers knowledge management and in Section 5 some examples are presented. A Conclusion section closes the paper.

2. PROBLEM FORMULATION

To better comprehend the algorithm, some mathematical notations are presented first. The curve is defined as a NURBS curve degree p as follows [21]:

$$C(u) = \sum_{i=0}^{n} N_{i,p}(u) P_i^w$$

where P_i^w are the weighted control points, and $N_{i,p}(u)$ are the normalized B-splines defined over the non-uniform and clamped knot vector

$$U = \{\underbrace{u_0 = \cdots = u_p}_{p+1}, u_{p+1}, \ldots, u_n, \underbrace{u_{m-p} = \cdots = u_m}_{p+1}\}$$

The line is represented parametrically:

$$L(t) = (1-t)Q_s + tQ_e$$

where Q_s and Q_e denote the start and end points. Now, the intersection problem is formulated as follows: find the geometric as well as the parametric locations of all intersection points to satisfy the following condition:

$$|C(u_i) - L(t_i)| < tol \quad u_i, t_i; i = 0, \ldots, k$$

where u_i and t_i are the parametric locations of the intersection points on the curve and on the line, respectively, and *tol* is a model space point coincidence tolerance, the only tolerance that is used and allowed in the method.

3. ALGORITHM OVERVIEW

The algorithm presented herein has the following major components:

1. Find information about the entity types such as lines (defined as linear splines) or circles (one can also consider other types of conic sections). The line-line case is robust, however, the line-circle case is non-trivial due mainly to the need to recover the parameter and that circles are defined differently in different systems, e.g. degree 2 versus degree 5, producing different parametrizations.

2. Decompose the NURBS curve if it has a straight-line path. Based on the convex hull property, if degree plus one or more control points are collinear, the curve has a line path. Note that the curve is not a line on that path, it is still a high degree curve exhibiting collinearity on a segment. Extracting these segments is important because of the numerical issues arising from line-curve overlap.
3. Get a biarc approximation of the curve even if it is a line segment. Note that our biarc curves approximate both the geometry as well as the parametrization, which is necessary to recover both the geometric as well as the parametric locations of the intersection points.
4. For each biarc, get the intersection points on the line segment and the biarc (Bezier circular arc) and recover the local (Bezier arc) as well as the global parameters.
5. Use the line-biarc intersection points and their parameters as start points for a Newton-type method to home in on the true intersection points. Note that the biarc approximation tolerance can be much larger than the required intersection tolerance as we are looking for good start points (in our system we used 10^{-3} as a biarc tolerance and 10^{-6} for intersection tolerance).
6. Purge the intersection points and generate the output. Using the biarc approach multiple intersections are possible at the locations where the biarcs meet. However, these can be identified and purged out.
7. Update the knowledge base with the newly created relationships saving all the parameters necessary to reproduce the results.

The rest of this section provides the critical details of most of the steps above.

3.1. NURBS Curve Decomposition

In order for the line curve intersection to work properly, straight segments must be extracted from the curve. Based on simple B-spline theory, the curve exhibits what is called the strong convex hull property: if the parameter moves within a knot span, i.e. $u \in [u_i, u_{i+1})$, then the curve $C(u|[u_i, u_{i+1}))$ is in the convex hull of the control points $P_{i-p}, \ldots, P_i$. From this follows that if the control points $P_k, \ldots, P_l$ are collinear, then the curve $C(u|[u_{k+p}, u_{l+1}))$ exhibits a straight path. The outline of the algorithm is as follows:

- Step through the control points and find collapsing convex hulls defined by at least $p+1$ consecutive control points.
- Record the parameter intervals over which the curve is straight.
- Decompose the curve into straight as well as non-straight segments.
- Process each segment independently and collect all intersection points.

Note that the straight curve segments are not linear splines so they have to be approximated by biarcs to get the proper parameters of the intersections. However, parallel and overlapping cases can be handled, i.e. if the curve path is straight and the line is parallel or overlapping with this path, the segment is skipped and no intersection is recorded.

Figure 1 shows a simple example of a curve with a straight path defined by five collinear control points (top). The result of the decomposition is three curve segments (bottom, the curves are pulled apart for better visualization) that are processed independently.

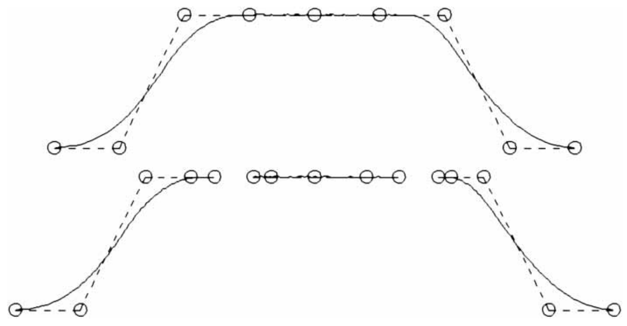
Fig. 1: Curve with a straight path (top) and extracted segments (bottom).

3.2. Biarc Approximation

The curve segments obtained in the previous step are handed over to the biarc decomposition preprocessor [22,26]. This unit obtains a piecewise circular approximation of the curve to within a tolerance that generates a reasonable number of arcs as well as a good start point for a numerical method to compute the true intersections (the number of arcs and the accuracy of the approximation can be tailored to the needs of the application). The advantages of biarc decomposition are many fold:

- The complex NURBS curve is decomposed into simple and well-known segments.
- The exact number of intersection points with a circular arc is known and is computed in a robust manner.
- The biarc approximation provides excellent start points and accounts for all intersections.
- The biarc decomposition can be done with a much larger tolerance and it serves as a fast pre-processor (in all in our tests we used 10^{-3} for biarc approximation and 10^{-6} for final accuracy).
- Once the biarc functions are well tested, they require almost no maintenance.

The essence of the biarc decomposition is to obtain a piecewise circular curve that approximates both the shape of the curve as well as its parametrization. The outline of the algorithm is presented below.

```
initialize a stack with the start and end knots
while stack is not empty do
    $(u_l, u_r) \leftarrow$ pop the stack
    extract segment $C(u|[u_l, u_r])$
    $P_s = C(u_l); T_s = C'(u_l); P_e = C(u_r);$
    $T_e = C'(u_r);$
    get the biarc $B(u) = B(u|\langle P_s, T_s, P_e, T_e\rangle)$
    get the error curve $E(u) = C(u|[u_l, u_r]) - B(u)$
    $u_m \leftarrow$ compute the maximum error
    if the error is not within tolerance
        $(u_m, u_r) \rightarrow$ push the stack
        $(u_l, u_m) \rightarrow$ push the stack
    else
        save $B(u)$
    end
end
stitch biarcs together to form a composite curve
```

The critical element of the above algorithm is the computation of the parametric error between the curve and a biarc segment. The error curve is computed symbolically [20]:

$$C(u) = \frac{N_C(u)}{d_C(u)} \quad B(u) = \frac{N_B(u)}{d_B(u)}$$

$$\Rightarrow \quad E(u) = \frac{N_C(u)d_B(u) - N_B(u)d_C(u)}{d_c(u)d_B(u)}$$

To compute the error curve $E(u)$, the following utilities are required:

- Decompose the curve into numerator curve and denominator function.
- Compute the product of a B-spline curve and a B-spline function.
- Compute the product of two B-spline functions.
- Compute the sum/difference of two B-spline curves.
- Assemble a NURBS curve from numerator curve and denominator function.

These utilities are part of the suit of symbolic operators that perform algebraic operations on functions, curves, surfaces and volumes. Since the error curve is a NURBS curve, the maximum error and its parametric location can be computed using the well-known knot refinement utility. Refining the error curve a few times, the control polygon forms a very good approximation of the curve giving an excellent error bound:

$$\varepsilon_{\max} = \max(|P_i|), i = 0, \ldots, n$$

where $P_i, i = 0, \ldots, n$ denote the Euclidean control points of the error curve. The parameter value at which the maximum error is recorded is the node t_j that belongs to the longest position vector p_j of the error curve and is computed as follows [21]:

$$t_j = \frac{1}{p}\sum_{k=1}^{p} u_{j+k}$$

Figure 2 shows an example of biarc decomposition. A degree four curve is decomposed into three biarcs, one C-shaped and two S-shaped, resulting in six circular arc segments that join with tangent vector continuity. The middle part of the figure shows the error curve with three loops corresponding to the three biarcs.

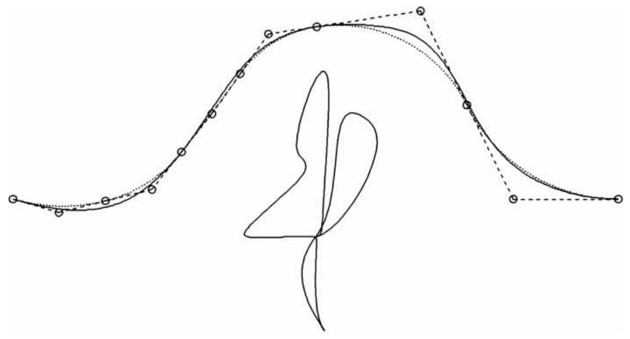
Fig. 2: Biarc decomposition (top) and error curve (middle).

3.3. Line Segment Bezier Circle Intersection

Once the original curve is decomposed into circle segments, it is intersected with the input line segment. It may sound trivial, however, to obtain robustness a few issues need to be addressed. The first is the trivial rejection. Since we have a large number of circular arcs, it is important to reject as many arcs as possible that miss the input line. As Figure 3 illustrate, the line Bezier triangle test may not be very efficient. In fact, a large piece of the bounding triangle can be intersected without touching the circular arc. Other bounding objects can be thought of, e.g. a tighter convex hull obtained by knot insertion, however, line and bounding polygon intersection, although conceptually simple, computational can be quite involved (accounting for vertex intersections and overlaps). Since the circle is the simplest object in the Universe, we implemented a circle bounding technique, as illustrated in Figure 3. The method is simple: split the arc in 2, 4, etc., pieces and draw enclosing circles. If the line segment intersects the circular arc, it must also intersect the enclosing circles. The line enclosing circle test is incredibly simple and is robust.

The second issue is the intersection of the line with a circle. Now, it is a textbook example, however, all textbooks get this wrong. The method involves representing the circle as an implicit curve, the line as a parametric entity and then solving a quadratic equation for the intersection points. We have at least

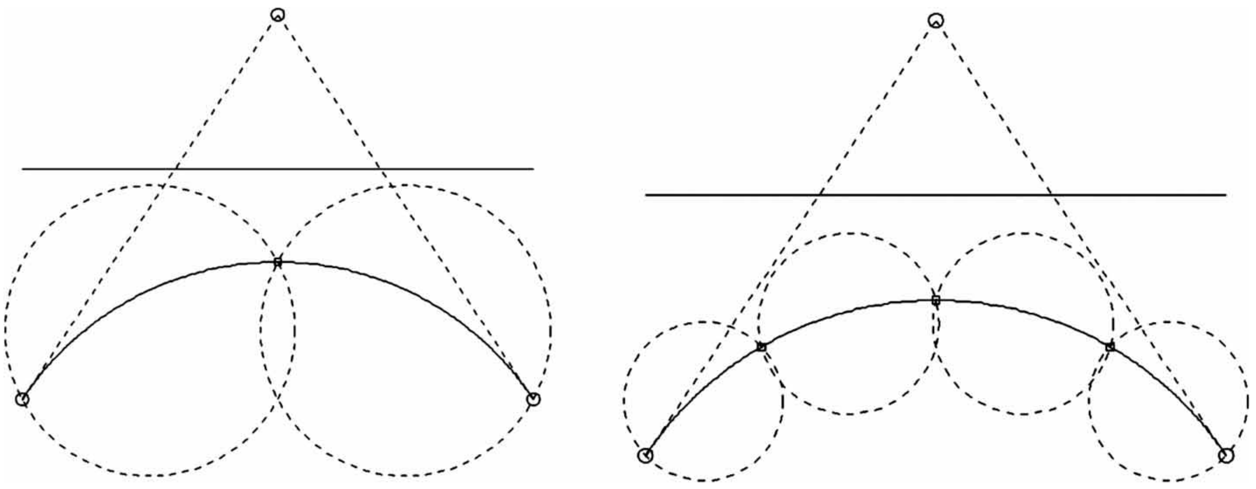

Fig. 3: Bounding the circular arc with circles for quick rejection test.

two problems with this: (1) quadratic equation solvers are not very robust, and (2) they require algebraic tolerances which we do not allow. So the line-circle intersection problem has to be solved geometrically using only the point coincidence tolerance. Fortunately, this is a trivial problem and can be coded in a very robust manner. Based on the notation in Figure 4, project the center of the circle to the line and compute the entities as shown in the figure. Then the intersections are obtained as follows:

$$Q_1 = F - dV^U \quad Q_2 = F + dV^U \quad d = \sqrt{r^2 - a^2}$$

where V^U is the unit direction vector representing the line. The beauty of this method is that it requires only distance calculations and that the decision of 0, 1 or 2 intersection(s) is based on the model space tolerance:

$$0 : a > r + tol \qquad 1 : |a - r| \le tol \qquad 2 : a < r - tol$$

The next important matter is to decide if a circle intersection point is on the Bezier circle or not, and if yes, get the corresponding parameter. Based on the notation of Figure 5, the conditions are as follows:

$$d(P_1, Q_1) \le d(P_1, F) \quad \Rightarrow \quad Q_1 \in C(u)$$
$$d(P_1, Q_2) > d(P_1, F) \quad \Rightarrow \quad Q_2 \notin C(u)$$

Again, only simple distance calculations are required to make a robust decision. Now, if the point is on the Bezier circle, the parameter corresponding to the intersection point needs to be recovered. Based on some simple observations, the parameter corresponding to Q_1 is computed as follows [21]:

$$u_B = \frac{a}{1+a} \quad a = \sqrt{\frac{|P_0 - F|}{|P_2 - F|}}$$

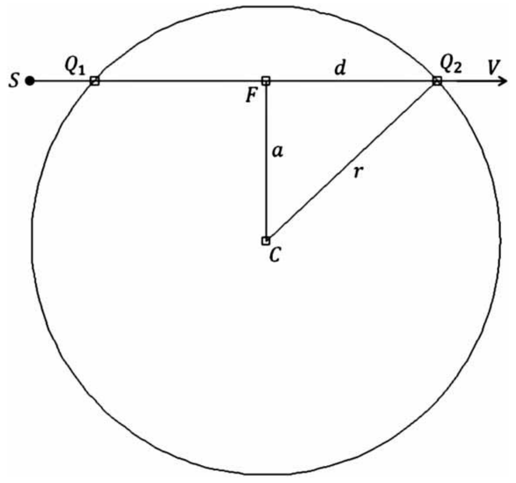

Fig. 4: Line circle intersection.

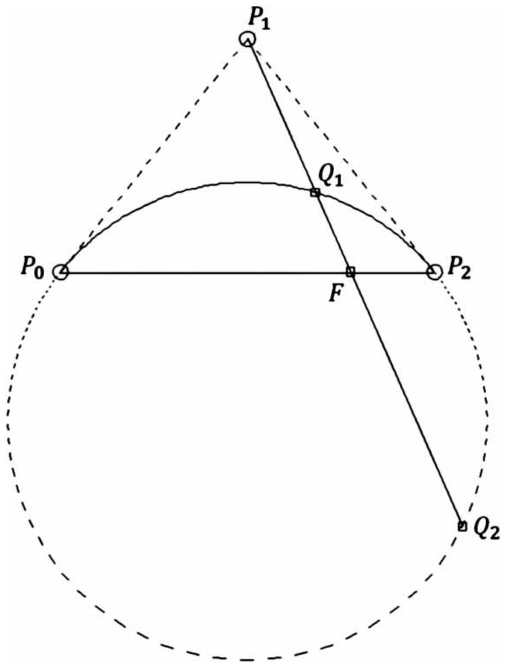

Fig. 5: Classifying Bezier circle intersections.

That is, the circular arc is a map of the straight line P_0P_2 from the middle control point P_1. The parameter for the intersecting line is recovered by simple linear interpolation. Returning to the global intersection problem, the start parameter for the numerical process is obtained via linear interpolation of the end knots used to obtain the Bezier segment:

$$u_0 = UL + u_B \cdot (UR - UL)$$

where *UL* and *UR* are the left and right knots used to extract the Bezier circle from the biarc approximation.

3.4. Intersection Computation

Given the intersection of the line segment with the piecewise circular arcs approximating the input curve, it is now time to compute accurate intersections with the initial curve. That is, given the start parameters u_0 and t_0, we are looking for a numerical process that starts at these parameters and converges to u and t so that

$$|C(u) - L(t)| < tol$$

where *tol* is the accuracy requirement for the intersection process. In order to account for special cases like tangential intersections, we use a distance formula, see Figure 6.

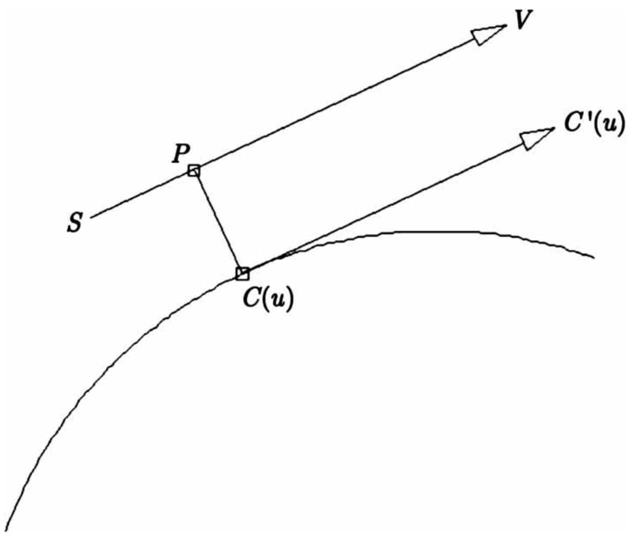

Fig. 6: Computing the distance between a curve and a line.

Any point on the line is expressed parametrically as $P = S + tV$. Forming the bivariate function

$$r(u,t) = C(u) - P = C(u) - S - tV$$

the conditions to be satisfied for distance computation are

$$f(u,t) = r(u,t) \cdot C'(u) = 0$$
$$g(u,t) = r(u,t) \cdot V = 0$$

These equations can be solved iteratively as follows. Let

$$\delta_i = \begin{bmatrix} \Delta u \\ \Delta t \end{bmatrix} = \begin{bmatrix} u_{i+1} - u_i \\ t_{i+1} - t_i \end{bmatrix}$$

$$J_i = \begin{bmatrix} f_u & f_t \\ g_u & g_t \end{bmatrix} = \begin{bmatrix} |C'(u_i)|^2 + r \cdot C''(u_i) & -V \cdot C'(u_i) \\ V \cdot C'(u_i) & -V \cdot V \end{bmatrix}$$

$$\kappa_i = -\begin{bmatrix} f(u_i,t_i) \\ g(u_i,t_i) \end{bmatrix}$$

Then at the i-th iteration the simple system of linear equations $J_i\delta_i = \kappa_i$ need to be solved in order to get a new set of parameters

$$u_{i+1} = \Delta u + u_i$$
$$t_{i+1} = \Delta t + t_i$$

There are two main convergence criteria:

- Point coincidence: $|C(u_i) - L(t_i)| < tol$
- Perpendicularity: $(C(u_i) - L(t_i)) \perp V \quad \wedge \quad (C(u_i) - L(t_i)) \perp C'(u_i)$

Point coincidence is trivial, however, perpendicularity needs some attention. It is typically checked using a zero cosine tolerance but since we allow only point coincidence tolerance, perpendicularity needs to be measured differently. Since in most engineering applications lines are finite, e.g. they form edges of objects, we introduce a condition that is applicable for line segments. Let the lines be $\ell(P_s, P_e)$ and $\ell(Q_s, Q_e)$. Then using a simple algorithm to project a point onto a line, the conditions become

$$P_s, P_e \to \ell(Q_s, Q_e) \quad \mapsto \quad P_s^\ell, P_e^\ell$$
$$Q_s, Q_e \to \ell(P_s, P_e) \quad \mapsto \quad Q_s^\ell, Q_e^\ell$$
$$d(P_s^\ell, P_e^\ell) < tol \quad \wedge \quad d(Q_s^\ell, Q_e^\ell) < tol$$

To ensure that the algorithm works correctly, a few other issues need to be addressed such as when the parameter gets stuck at the end of the curve, or it jumps to the other side of the parameter interval in case of closed curves. Because the biarc preprocessor produces very good start points and reproduces end points as well as turning points quite well, the numerical process never encountered issues with convergence.

3.5. Purging Intersection Points

Once the numerical method completed processing all intersection candidates, it is time to check for duplicate points. Duplicate intersections typically happen when the line hits the biarc right at the junction, or very near to it and the numerical method converges to the same point. These duplicate points are easily purged out as they are either identical (to within round off error) or they are less than tolerance apart.

3.6. Knowledge Management

There are two types of knowledge information that our system produces: (1) knowledge that is recorded when the objects are created, and (2) knowledge that is acquired after the entities take part in various operations such as intersections. For the first part, we group the information as follows:

- **Identity:** the name of the object, the date of its creation and whether it is rational or not.
- **Classification:** the type of the object, its origin and destination.
- **Representation:** anomalies such as parametrization factor, continuity as well as irregularities are recorded.
- **Definition:** this contains the typical numerical data of control points and knots.
- **Geometry:** important numerical quantities such as arc length and curvature extremes are stored.

The knowledge data above is typically mined at the beginning of operations to gather information such as curve type, parametrization or irregularities, so that the function can make intelligent decisions on how to best process the entity. Once the processing is complete, e.g. the intersections have been computed, important relationships are recorded in the second part of the knowledge base which is a relationship graph. For our intersection process the following knowledge base update is made:

- The *curve-line* graph is updated by adding the *intersection* relationship. The information stored are the IDs, the intersection points, the parameters, the tolerance and the name of the function that produced the intersection points.
- The *curve-point* graph is updated by the *incidence* relationship storing the IDs, the tolerance and the function that tests the incidence.
- The *line-point* graph is updated by adding the *incidence* relationship with data such as IDs, tolerance and function name.

In a typical design scenario the user may click on a point of intersection and wonder where it came from. The system queries the knowledge base and finds that this point is in incidence relationship with the curve and the line. Further inquiries reveal that the line and the curve are in an intersection relationship and that everything is available to reproduce this point. So if the accuracy is not sufficient in the receiving system, the intersection can be recomputed to any desired tolerance. The key philosophy of KGNurbs is design replay, i.e. the entire design or any part thereof can be reproduced because no valuable information (design intent) ever gets lost. Now, because building the knowledge base is an overhead, the system gives the designer the option to ignore the knowledge building and focus entirely on numerical data. This may be important during preliminary design or rapid prototyping where the prototype may get trashed at the end of the project.

4. EXAMPLES

A few examples are presented in this section. Since the method was developed for engineering applications, not for mathematics, the examples are realistic rather than tricky with all sorts of loops, although that would not make any difference. The first example is shown in Figure 7. The curve is a difficult one exhibiting lots of inflection points along a path that is nearly collinear. All inflection points as well as difficult touch point are handled quite well.

The second set of examples is from bioengineering, Figure 8. The contour curves represent a cross section of a bone (left) and a one of the lung (right). What is difficult about these examples is that the curves exhibit wild local curvature changes which is challenging to handle numerically.

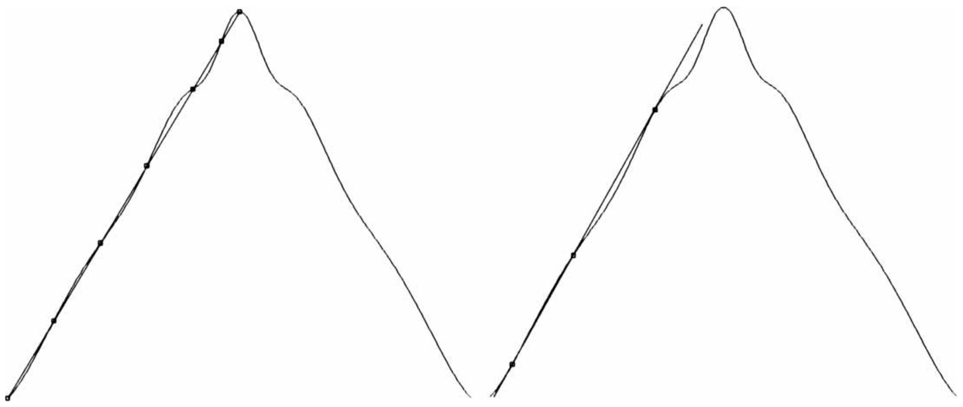

Fig. 7: Intersection example with inflection, touch and turning points.

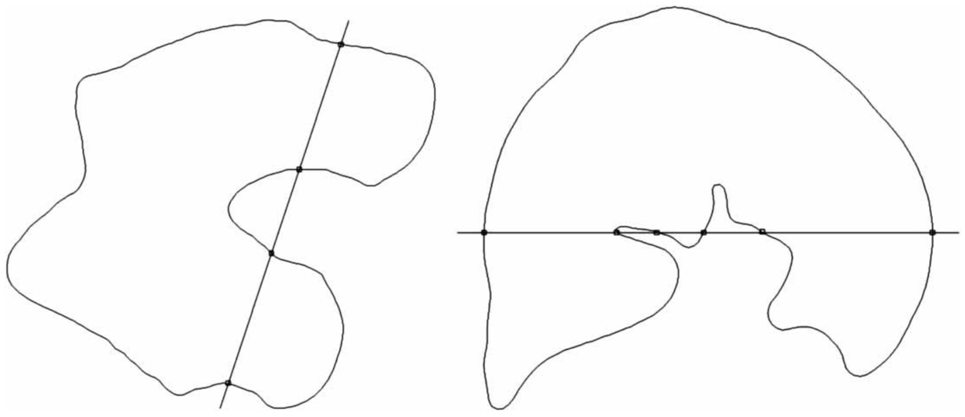

Fig. 8: Bone and lung contour curve intersections.

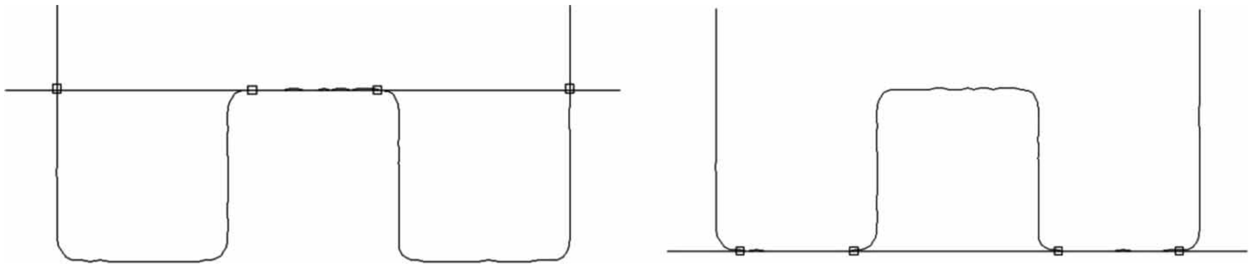

Fig. 9: Handling intersections with straight paths.

The final set of examples is shown in Figure 9. It shows a typical case scenario when engineering drawings contains lots of straight paths and the line happens to hit right along one of the straight pieces. While the line overlap with the straight curve is handled properly, the benefit of extracting all curve segments from the input curve is evident: the overlap is omitted but the non-straight pieces are properly intersected, i.e. the ends or the beginnings of the curve segments where it begins or ends the straight path are true intersections.

5. CONCLUSIONS

A robust algorithm to compute all the intersections between a line and a NURBS curve is presented. Instead of making the problem as part of a general purpose curve-curve intersection problem, we argue that general purpose intersection algorithms are inferior to special ones when the emphasis is on robustness, accuracy, speed and reliability. We believe that robustness can be achieved in CAD systems without changing the arithmetic. It requires knowledge, careful algorithm design and consistent tolerancing.

Since intersection algorithms require a lot of code and an entire numerical system (that may take at least 6 months to build), a comparison with other methods would be prohibitively time consuming. An objective comparison is left to independent developers who have both the resources as well as the need to implement a dozen or so intersection methods and compare their performance given a set of applications.

ACKNOWLEDGEMENTS

This research was supported by Najran University, Saudi Arabia, under grant No. ESCI/13/35. All opinions, findings, conclusions and recommendations expressed in this paper are those of the authors and do not necessarily reflect Najran University.

REFERENCES

[1] Asteasu, C.; Orbegozo, A.; Parametric piecewise surface intersection, Computers & Graphics, 15(1), 1991, 9–13.

[2] Aziz, N. M.; Bata, R. R.: Bezier surface/surface intersection, IEEE Computer Graphics and Applications, 10, 1990, 50–58.

[3] Boender, E.: A survey of intersection algorithms for curved surfaces, Computers & Graphics, 15(1), 1991, 109–115.

[4] Boissonnat, J.-D.; Snoeyink, J.: Efficient algorithms for line and curve segment intersection using restricted predicates, Computational Geometry, 16(1), 2000, 35–52.

[5] Dokken, T.: Finding intersections of B-spline represented geometries using recursive subdivision techniques, Computer Aided Geometric Design, 2(1–3), 1985, 189–195.

[6] Fang, S.; Bruderlin, B.; Zhu, X.: Robustness in solid modeling: a tolerance-based intuitionistic approach, Computer-Aided Design, 25(9), 1993, 567–576.

[7] Hawat, R.; Piegl, L. A.: Curve-curve intersection via genetic algorithms, Mathematical Engineering in Industry, 7(2), 1998, 269–282.

[8] Hedrick, R. W.; Bedi, S.: Method for intersection of parametric surfaces, Transaction CSMI, 14(3), 1990, 79–84.

[9] Hu, C.-Y.; Maekawa, T.; Sherbrooke, E. C.; Patrikalakis, N. M.: Robust interval algorithm for curve intersections, Computer-Aided Design, 28(6–7), 1996, 495–506.

[10] Johnstone, J. K.; Shene, C. K.: Computing the intersection of plane and natural quadric, Computers & Graphics, 16, 1992, 179–186.

[11] Kim, D.-S.; Lee, S.-W.; Shin, H.: A cocktail algorithm for planar Bezier curve intersections, Computer-Aided Design, 30(13), 1998, 1047–1051.

[12] Limaiem, A.; Truchu, F.: Geometric algorithms for the intersection of curves and surfaces, Computers & Graphics, 19(3), 1995, 391–403.

[13] Manocha, D.; Demmel, J.: Algorithms for intersecting parametric and algebraic curves II: multiple intersections, Graphical Models and Image Processing, 57(2), 1995, 81–100.

[14] Manocha, D.; Krishnan, S.: Algebraic pruning: a fast technique for curve and surface intersection, Computer-Aided Design, 14(9), 1997, 823–845.

[15] Markot, R. P.; Magedson, R. L.: Solutions of tangential surface and curve intersections, Computer-Aided Design, 21(7), 1989, 421–427.

[16] Morken, K.; Reimers, M.; Schulz, C.: Computing intersections of planar spline curves using knot insertion, Computer Aided Geometric Design, 26(3), 2009, 351–356.

[17] Mullenheim, G.: On determining start point for surface/surface intersection algorithm, Computer Aided Geometric Design, 8, 1991, 401–408.

[18] Patrikalakis, N. M.; Maekawa, T.: Intersection problems, in Shape Interrogation for Computer Aided Design and Manufacturing, 2010, 109–160, Springer-Verlag, NY.

[19] Piegl, L.: Geometric method of intersecting natural quadrics represented in trimmed surface form, Computer-Aided Design, 21(4), 1989, 201–212.

[20] Piegl, L.; Tiller, W.: Symbolic operators for NURBS, Computer-Aided Design, 29(5), 1997, 361–368.

[21] Piegl, L.; Tiller, W.: The NURBS Book, Springer-Verlag, New York, 1997.

[22] Piegl, L. A.; Tiller, W.: Biarc approximation of NURBS curves, Computer-Aided Design, 34, 2002, 807–814.

[23] Piegl, L. A.: Knowledge-Guided Computation for Robust CAD, Computer-Aided Design and Applications, 2(5), 2005, 685–695.

[24] Piegl, L. A.: Knowledge-Guided NURBS: Principles and Architecture, Computer-Aided Design and Applications, 3(6), 2006, 719–729.

[25] Piegl, L. A.; Rajab, K.; Smarodzinava, V.; Valavanis, K. P.: Fault-tolerant computing in a knowledge-guided NURBS environment, Computer-Aided Design and Applications, 6(6), 2009, 809–823.

[26] Piegl, L. A.; Rajab, K.; Smarodzinava, V.; Valavanis, K. P.; Using a biarc filter to compute curvature extremes of NURBS curves, Engineering with Computers, 25(4), 2009, 379–387.

[27] Pratt, M. J.; Geisow, A. D.: Surface/surface intersection problems, in J. A. Gregory (Ed.) The Mathematics of Surfaces, Oxford, 1986.

[28] Rohmfeld, R. F.: Classification of curve-curve intersections from the CAD/CAM viewpoint, Computer Graphics International, 1996, 230–239.

[29] Sederberg, T. W.; Parry, S.: Comparison of three curve intersection algorithms, Computer-Aided Design, 18(1), 1986, 58–63.

[30] Sederberg, T. W.; Nishita, T.: Curve intersection using Bezier clipping, Computer-Aided Design, 22(9), 1990, 538–549.

[31] Yang, P.; Qian, X.: Direct Boolean intersection between acquired and designed geometry, Computer-Aided Design, 41(2), 2009, 81–94.