Chapter 7
Intersection in 2D

This chapter contains information on computing the intersection of geometric primitives in 2D. The simplest object combinations to analyze are those for which one of the objects is a linear component (line, ray, segment). These combinations are covered in the first four sections. Section 7.5 covers the intersection of a pair of quadratic curves; Section 7.6 covers the problem of intersection of a pair of polynomial curves. The last section is about the method of separating axes, a very powerful technique for dealing with intersections of convex objects.

7.1 Linear Components

Recall from Chapter 5 the definitions for lines, rays, and segments. A *line* in 2D is parameterized as $P + t\vec{d}$, where $\vec{d}$ is a nonzero vector and where $t \in \mathbb{R}$. A *ray* is parameterized the same way except that $t \in [0, \infty)$. The point P is the origin of the ray. A *segment* is also parameterized the same way except that $t \in [0, 1]$. The points P and $P + \vec{d}$ are the end points of the segment. A *linear component* is the general term for a line, a ray, or a segment.

Given two lines $P_0 + s\vec{d}_0$ and $P_1 + t\vec{d}_1$ for $s, t \in \mathbb{R}$, they are either intersecting, nonintersecting and parallel, or the same line. To help determine which of these cases occurs, define for two input 2D vectors the scalar-valued operation $\text{Kross}((x_0, y_0), (x_1, y_1)) = x_0y_1 - x_1y_0$. The operation is related to the cross product in 3D given by $(x_0, y_0, 0) \times (x_1, y_1, 0) = (0, 0, \text{Kross}((x_0, y_0), (x_1, y_1)))$. The operation has the property that $\text{Kross}(\vec{u}, \vec{v}) = -\text{Kross}(\vec{v}, \vec{u})$.

A point of intersection, if any, can be found by solving the two equations in two unknowns implied by setting $P_0 + s\vec{d}_0 = P_1 + t\vec{d}_1$. Rearranging terms yields $s\vec{d}_0 - t\vec{d}_1 = P_1 - P_0$. Setting $\vec{\Delta} = P_1 - P_0$ and applying the Kross operation yields

$\text{Kross}(\vec{d}_0, \vec{d}_1)\, s = \text{Kross}(\vec{\Delta}, \vec{d}_1)$ and $\text{Kross}(\vec{d}_0, \vec{d}_1)\, t = \text{Kross}(\vec{\Delta}, \vec{d}_0)$. If $\text{Kross}(\vec{d}_0, \vec{d}_1) \neq 0$, then the lines intersect in a single point determined by $s = \text{Kross}(\vec{\Delta}, \vec{d}_1)/\text{Kross}(\vec{d}_0, \vec{d}_1)$ or $t = \text{Kross}(\vec{\Delta}, \vec{d}_0)/\text{Kross}(\vec{d}_0, \vec{d}_1)$. If $\text{Kross}(\vec{d}_0, \vec{d}_1) = 0$, then the lines are either nonintersecting and parallel or the same line. If the Kross operation of the direction vectors is zero, then the previous equations in s and t reduce to a single equation $\text{Kross}(\vec{\Delta}, \vec{d}_0) = 0$ since $\vec{d}_1$ is a scalar multiple of $\vec{d}_0$. The lines are the same if this equation is true; otherwise, the lines are nonintersecting and parallel.

If using floating-point arithmetic, distinguishing the nonparallel from the parallel case can be tricky when $\text{Kross}(\vec{d}_0, \vec{d}_1)$ is nearly zero. Using the relationship of Kross to the 3D cross product, a standard identity for the cross product in terms of Kross is $\|\text{Kross}(\vec{d}_0, \vec{d}_1)\| = \|\vec{d}_0\|\|\vec{d}_1\||\sin\theta|$, where θ is the angle between $\vec{d}_0$ and $\vec{d}_1$. For the right-hand side of the last equation to be nearly zero, one or more of its three terms must be nearly zero. A test for parallelism using an absolute error comparison $\|\text{Kross}(\vec{d}_0, \vec{d}_1)\| \leq \varepsilon$ for some small tolerance $\varepsilon > 0$ may not be suitable for some applications. For example, two perpendicular direction vectors that have very small length can cause the test to report that the lines are parallel when in fact they are perpendicular. If possible, the application should require that the line directions be unit-length vectors. The absolute error test then becomes a test on the sine of the angle between the directions: $\|\text{Kross}(\vec{d}_0, \vec{d}_1)\| = |\sin\theta| \leq \varepsilon$. For small enough angles, the test is effectively a threshold on the angle itself since $\sin\theta \doteq \theta$ for small angles. If the application cannot require that the line directions be unit length, then the test for parallelism should be based on relative error:

$$\frac{\|\text{Kross}(\vec{d}_0, \vec{d}_1)\|}{\|\vec{d}_0\|\|\vec{d}_1\|} = |\sin\theta| \leq \varepsilon$$

The square root calculations for the two lengths and the division can be avoided by using instead the equivalent inequality

$$\|\text{Kross}(\vec{d}_0, \vec{d}_1)\|^2 \leq \varepsilon^2\|\vec{d}_0\|^2\|\vec{d}_1\|^2$$

If the two linear components are a line ($s \in \mathbb{R}$) and a ray ($t \geq 0$), the point of intersection, if it exists, is determined by solving for s and t as shown previously. However, it must be verified that $t \geq 0$. If $t < 0$, the first line intersects the line containing the ray, but not at a ray point. Computing the solution t as specified earlier involves a division. An implementation can avoid the cost of the division when testing $t \geq 0$ by observing that $t = \text{Kross}(\vec{\Delta}, \vec{d}_0)\text{Kross}(\vec{d}_0, \vec{d}_1)/(\text{Kross}(\vec{d}_0, \vec{d}_1))^2$ and using the equivalent test $\text{Kross}(\vec{\Delta}, \vec{d}_0)\text{Kross}(\vec{d}_0, \vec{d}_1) \geq 0$. If in fact the equivalent test shows that $t \geq 0$ and if the application needs to know the corresponding point of intersection, only then should t be directly computed, thus deferring a division until it is needed. Similar tests on s and t may be applied when either linear component is a ray or a segment.

Finally, if the two linear components are on the same line, the linear components intersect in a t-interval, possibly empty, bounded, semi-infinite, or infinite. Computing the interval of intersection is somewhat tedious, but not complicated. As an example, consider the case when both linear components are line segments, so $s \in [0, 1]$ and $t \in [0, 1]$. We need to compute the s-interval of the second line segment that corresponds to the t-interval $[0, 1]$. The first end point is represented as $P_1 = P_0 + s_0\vec{d}_0$; the second is represented as $P_1 + \vec{d}_1 = P_0 + s_1\vec{d}_0$. If $\vec{\Delta} = P_1 - P_0$, then $s_0 = \vec{d}_0 \cdot \vec{\Delta}/\|\vec{d}_0\|^2$ and $s_1 = s_0 + \vec{d}_0 \cdot \vec{d}_1/\|\vec{d}_0\|^2$. The s-interval is $[s_{\min}, s_{\max}] = [\min(s_0, s_1), \max(s_0, s_1)]$. The parameter interval of intersection is $[0, 1] \cap [s_{\min}, s_{\max}]$, possibly the empty set. The 2D points of intersection for the line segment of intersection can be computed from the interval of intersection by using the interval end points in the representation $P_0 + s\vec{d}_0$.

The pseudocode for the intersection of two lines is presented below. The return value of the function is 0 if there is no intersection, 1 if there is a unique intersection, and 2 if the two lines are the same line. The returned point `I` is valid only when the function returns 1.

```
int FindIntersection(Point P0, Point D0, Point P1, Point D1, Point& I)
{
    // Use a relative error test to test for parallelism.  This effectively
    // is a threshold on the angle between D0 and D1.  The threshold
    // parameter 'sqrEpsilon' can be defined in this function or be
    // available globally.

    Point E = P1 - P0;
    float kross = D0.x * D1.y - D0.y * D1.x;
    float sqrKross = kross * kross;
    float sqrLen0 = D0.x * D0.x + D0.y * D0.y;
    float sqrLen1 = D1.x * D1.x + D1.y * D1.y;
    if (sqrKross > sqrEpsilon * sqrLen0 * sqrLen1) {
        // lines are not parallel
        float s = (E.x * D1.y - E.y *D1.x) / kross;
        I = P0 + s * D0;
        return 1;
    }

    // lines are parallel
    float sqrLenE = E.x * E.x + E.y * E.y;
    kross = E.x * D0.y - E.y * D0.x;
    sqrKross = kross * kross;
    if (sqrKross > sqrEpsilon * sqrLen0 * sqrLenE) {
        // lines are different
        return 0;
    }
```

```
    // lines are the same
    return 2;
}
```

The pseudocode for the intersection of two line segments is presented below. The return value of the function is 0 if there is no intersection, 1 if there is a unique intersection, and 2 if the two segments overlap and the intersection set is a segment itself. The return value is the number of valid entries in the array `I[2]` that is passed to the function. Relative error tests are used in the same way as they were in the previous function.

```
int FindIntersection(Point P0, Point D0, Point P1, Point D1, Point2 I[2])
{
    // segments P0 + s * D0 for s in [0, 1], P1 + t * D1 for t in [0,1]

    Point E = P1 - P0;
    float kross = D0.x * D1.y - D0.y * D1.x;
    float sqrKross = kross * kross;
    float sqrLen0 = D0.x * D0.x + D0.y * D0.y;
    float sqrLen1 = D1.x * D1.x + D1.y * D1.y;
    if (sqrKross > sqrEpsilon * sqrLen0 * sqrLen1) {
        // lines of the segments are not parallel
        float s = (E.x * D1.y - E.y * D1.x) / kross;
        if (s < 0 or s > 1) {
            // intersection of lines is not a point on segment P0 + s * D0
            return 0;
        }

        float t = (E.x * D0.y - E.y * D0.x) / kross;
        if (t < 0 or t > 1) {
            // intersection of lines is not a point on segment P1 + t * D1
            return 0;
        }

        // intersection of lines is a point on each segment
        I[0] = P0 + s * D0;
        return 1;
    }

    // lines of the segments are parallel
    float sqrLenE = E.x * E.x + E.y * E.y;
    kross = E.x * D0.y - E.y * D0.x;
    sqrKross = kross * kross;
    if (sqrKross > sqrEpsilon * sqrLen0 * sqrLenE) {
```

```
        // lines of the segments are different
        return 0;
    }

    // Lines of the segments are the same.  Need to test for overlap of
    // segments.
    float s0 = Dot(D0, E) / sqrLen0, s1 = s0 + Dot(D0, D1) / sqrLen0, w[2];
    float smin = min(s0, s1), smax = max(s0, s1);
    int imax = FindIntersection(0.0, 1.0, smin, smax, w);
    for (i = 0; i < imax; i++)
        I[i] = P0 + w[i] * D0;
    return imax;
}
```

The intersection of two intervals $[u_0, u_1]$ and $[v_0, v_1]$, where $u_0 < u_1$ and $v_0 < v_1$, is computed by the function shown below. The return value is 0 if the intervals do not intersect; 1 if they intersect at a single point, in which case `w[0]` contains that point; or 2 if they intersect in an interval whose end points are stored in `w[0]` and `w[1]`.

```
int FindIntersection(float u0, float u1, float v0, float v1, float w[2])
{
    if (u1 < v0 || u0 > v1)
        return 0;

    if (u1 > v0) {
        if (u0 < v1) {
            if (u0 < v0) w[0] = v0; else w[0] = u0;
            if (u1 > v1) w[1] = v1; else w[1] = u1;
            return 2;
        }  else {
            // u0 == v1
            w[0] = u0;
            return 1;
        }
    } else {
        // u1 == v0
        w[0] = u1;
        return 1;
    }
}
```

7.2 Linear Components and Polylines

The simplest algorithm for computing the intersection of a linear component and a polyline is to iterate through the edges of the polyline and apply an intersection test for linear component against line segment. If the goal of the application is to determine if the linear component intersects the polyline without finding where intersections occur, then an early out occurs once a polyline edge is found that intersects the linear component.

If the polyline is in fact a polygon and the geometric query treats the polygon as a solid, then an iteration over the polygon edges and applying the intersection test for a line or a ray against polygon edges is sufficient to determine intersection. If the linear component is a line segment itself, the iteration is not enough. The problem is that the line segment might be fully contained in the polygon. Additional tests need to be made, specifically point-in-polygon tests applied to the end points of the line segment. If either end point is inside the polygon, the segment and polygon intersect. If both end points are outside, then an iteration over the polygon edges is made and segment-segment intersection tests are performed.

If the intersection query is going to be performed often for a single polyline but with multiple linear components, then some preprocessing can help reduce the computational time that is incurred by the exhaustive edge search. One such algorithm for preprocessing involves *binary space partitioning* (BSP) *trees,* discussed in Section 13.1. In that section there is some material on intersection of a line segment with a polygon that is already represented as a BSP tree. The exhaustive search of n polygon edges is an $O(n)$ process. The search through a BSP tree is an $O(\log n)$ process. Intuitively, if the line segment being compared to the polygon is on one side of a partitioning line corresponding to an edge of the polygon, then that line segment need not be tested for intersection with any polygon edges on the opposite side of the partition. Of course, there is the preprocessing cost of $O(n \log n)$ to build the tree.

Another possibility for reducing the costs is to attempt to rapidly cull out segments of the polyline so they are not used in intersection tests with the linear component. The culling idea in Section 6.7 may be used with this goal.

7.3 Linear Components and Quadratic Curves

We discuss in this section how to test or find the intersection points between a linear component and a quadratic curve. The method for an implicitly defined quadratic curve is presented first. The special case for intersections of a linear component and a circle or arc are presented second.

7.3.1 Linear Components and General Quadratic Curves

A quadratic curve is represented implicitly by the quadratic equation $X^{\mathrm{T}}\mathbf{A}X + B^{\mathrm{T}}X + c = 0$, where $\mathbf{A}$ is a 2×2 symmetric matrix, B is a 2×1 vector, c is a scalar, and X is the 2×1 variable representing points on the curve.

The intersection of a line $X(t) = P + t\vec{d}$ for $t \in \mathbb{R}$ and a quadratic curve is computed by substituting the line equation into the quadratic equation to obtain

$$\begin{aligned} 0 &= X(t)^{\mathrm{T}}\mathbf{A}X(t) + B^{\mathrm{T}}X(t) + c \\ &= \left(\vec{d}^{\,\mathrm{T}}\mathbf{A}\vec{d}\right) t^2 + \vec{d}^{\,\mathrm{T}}\left(2\mathbf{A}P + B\right) t + \left(P^{\mathrm{T}}\mathbf{A}P + B^{\mathrm{T}}P + c\right) \\ &=: e_2 t^2 + e_1 t + e_0 \end{aligned}$$

This quadratic equation can be solved using the quadratic formula, but attention must be paid to numerical issues, for example, when e_2 is nearly zero or when the discriminant $e_1^2 - 4e_0e_2$ is nearly zero. If the equation has two distinct real roots, the line intersects the curve in two points. Each root $\bar{t}$ is used to compute the actual point of intersection $X(\bar{t}) = P + \bar{t}\vec{d}$. If the equation has a repeated real root, then the line intersects the curve in a single point and is tangent at that point. If the equation has no real-valued roots, the line does not intersect the curve.

If the linear component is a ray with $t \geq 0$, an additional test must be made to see if a root $\bar{t}$ to the quadratic equation is nonnegative. It is possible that the line containing the ray intersects the quadratic curve, but the ray itself does not. Similarly, if the linear component is a line segment with $t \in [0, 1]$, additional tests must be made to see if a root $\bar{t}$ to the quadratic equation is also in $[0, 1]$.

If the application's goal is to determine only if the linear component and quadratic curve intersect, but does not care about where the intersections occur, then the root finding for $q(t) = e_2t^2 + e_1t + e_0 = 0$ can be skipped to avoid the expensive square root and division that occur in the quadratic formula. Instead we only need to know if $q(t)$ has a real-valued root in $\mathbb{R}$ for a line, in $[0, \infty)$ for a ray, or in $[0, 1]$ for a line segment. This can be done using Sturm sequences, as described in Section A.5. This method uses only floating-point additions, subtractions, and multiplications to count the number of real-valued roots for $q(t)$ on the specified interval.

7.3.2 Linear Components and Circular Components

A *circle* in 2D is represented by $\|X - C\|^2 = r^2$, where C is the center and $r > 0$ is the radius of the circle. The circle can be parameterized by $X(\theta) = C + r\hat{u}(\theta)$, where $\hat{u}(\theta) = (\cos\theta, \sin\theta)$ and where $\theta \in [0, 2\pi)$. An *arc* is parameterized the same way except that $\theta \in [\theta_0, \theta_1]$ with $\theta_0 \in [0, 2\pi)$, $\theta_0 < \theta_1$, and $\theta_1 - \theta_0 < 2\pi$. It is also possible to represent an arc by center C, radius r, and two end points A and B that correspond

to angles θ_0 and θ_1, respectively. The term *circular component* is used to refer to a circle or an arc.

Consider first a parameterized line $X(t) = P + t\vec{d}$ and a circle $\|X - C\|^2 = r^2$. Substitute the line equation into the circle equation, define $\vec{\Delta} = P - C$, and obtain the quadratic equation in t:

$$\|\vec{d}\|^2 t^2 + 2\vec{d} \cdot \vec{\Delta} t + \|\vec{\Delta}\|^2 - r^2 = 0$$

The formal roots of the equation are

$$t = \frac{-\vec{d} \cdot \vec{\Delta} \pm \sqrt{(\vec{d} \cdot \vec{\Delta})^2 - \|\vec{d}\|^2(\|\vec{\Delta}\|^2 - r^2)}}{\|\vec{d}\|^2}$$

Define $\delta = (\vec{d} \cdot \vec{\Delta})^2 - \|\vec{d}\|^2(\|\vec{\Delta}\|^2 - r^2)$. If $\delta < 0$, the line does not intersect the circle. If $\delta = 0$, the line is tangent to the circle in a single point of intersection. If $\delta > 0$, the line intersects the circle in two points.

If the linear component is a ray, and if $\bar{t}$ is a real-valued root of the quadratic equation, then the corresponding point of intersection between line and circle is a point of intersection between ray and circle if $\bar{t} \geq 0$. Similarly, if the linear component is a segment, the line-circle point of intersection is also one for the segment and circle if $\bar{t} \in [0, 1]$.

If the circular component is an arc, the points of intersection between the linear component and circle must be tested to see if they are on the arc. Let the arc have end points A and B, where the arc is that portion of the circle obtained by traversing the circle counterclockwise from A to B. Notice that the line containing A and B separates the arc from the remainder of the circle. Figure 7.1 illustrates this. If P is a point on the circle, it is on the arc if and only if it is on the same side of that line as the arc. The algebraic condition for the circle point P to be on the arc is $\text{Kross}(P - A, B - A) \geq 0$, where $\text{Kross}((x_0, y_0), (x_1, y_1)) = x_0 y_1 - x_1 y_0$.

7.4 Linear Components and Polynomial Curves

Consider a line $P + t\vec{d}$ for $t \in \mathbb{R}$ and a polynomial curve $X(s) = \sum_{i=0}^{n} \vec{A}_i s^i$, where $\vec{A}_n \neq \vec{0}$. Let the parameter domain be $[s_{\min}, s_{\max}]$. This section discusses how to compute points of intersection between the line and curve from both an algebraic and geometric perspective.

7.4.1 Algebraic Method

Intersections of the line and curve, if any, can be found by equating $X(s) = P + t\vec{d}$ and solving for s by eliminating the t-term using the Kross operator:

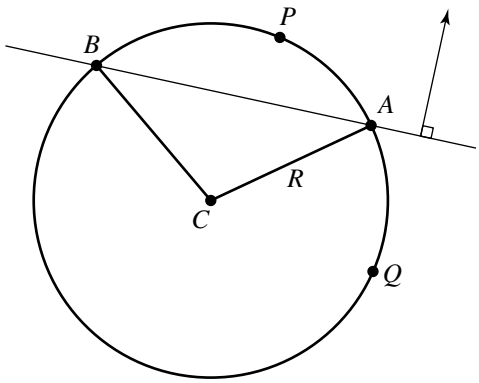

Figure 7.1 An arc of a circle spanned counterclockwise from A to B. The line containing A and B separates the circle into the arc itself and the remainder of the circle. Point P is on the arc since it is on the same side of the line as the arc. Point Q is not on the arc since it is on the opposite side of the line.

$$\sum_{i=0}^{n} \Big(\text{Kross}(\vec{d}, \vec{A}_i)\Big)\, s^i = \text{Kross}(\vec{d}, X(s)) = \text{Kross}(\vec{d}, P + t\vec{d}) = \text{Kross}(\vec{d}, P)$$

Setting $c_0 = \text{Kross}(\vec{d}, \vec{A}_0 - P)$ and $c_i = \text{Kross}(\vec{d}, \vec{A}_i)$ for $i \geq 1$, the previous equation is reformulated as the polynomial equation $q(s) = \sum_{i=0}^{n} c_i s^i = 0$. A numerical root finder can be applied to this equation, but beware of c_n being zero (or nearly zero) when $\vec{d}$ and $\vec{A}_n$ are parallel (or nearly parallel). Any $\bar{s}$ for which $q(\bar{s}) = 0$ must be tested for inclusion in the parameter domain $[s_{\min}, s_{\max}]$. If so, a point of intersection has been found.

EXAMPLE Let the line be $(0, 1/2) + t(2, -1)$ and the polynomial curve be $X(s) = (0, 0) + s(1, 2) + s^2(0, -3) + s^3(0, 1)$ for $s \in [0, 1]$. The curve is unimodal and has x-range $[0, 1]$ and y-range $[0, 3/8]$. The polynomial equation is $q(s) = 2s^3 - 6s^2 + 5s - 1 = 0$. The roots are $s = 1, 1 \pm \sqrt{2}/2$. Only the roots 1 and $1 - \sqrt{2}/2$ are in $[0, 1]$. Figure 7.2 shows the line, curve, and points of intersection $\vec{I}_0$ and $\vec{I}_1$. ■

The numerical root finders might have problems finding roots of even multiplicity or at a root where $q(s)$ does not have a large derivative. Geometrically these cases happen when the line and tangent line at the point of intersection form an angle that is nearly zero.

Just as in the problem of computing intersections of linear components with quadratic curves, if the application's goal is to determine only if the linear component and polynomial curve intersect, but does not care about where the intersections

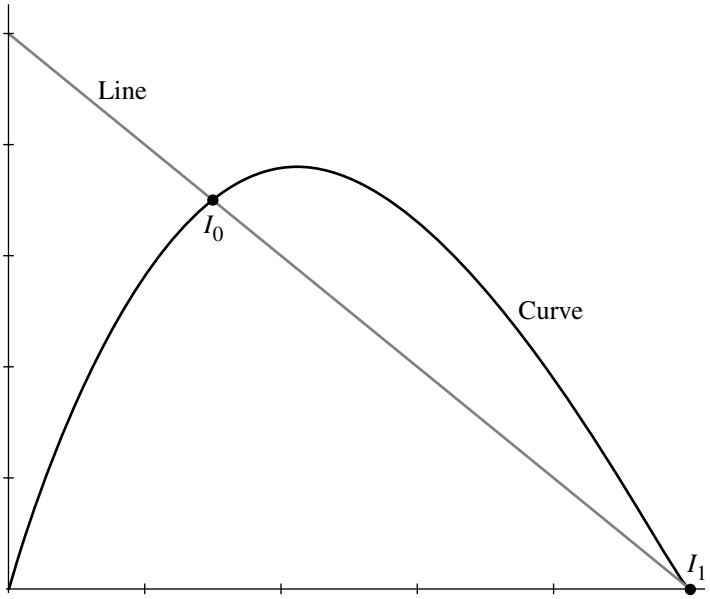

Figure 7.2 Intersection of a line and a cubic curve.

occur, then the root finding for $q(s) = 0$ can be skipped and Sturm sequences used (Section A.5) to count the number of real-valued roots in the domain $[s_{\min}, s_{\max}]$ for the curve $X(s)$. If the count is zero, then the line and polynomial curve do not intersect.

EXAMPLE Using the same example as the previous one, we only want to know the number of real-valued roots for $q(s) = 0$ in $[0, 1]$. The Sturm sequence is $q_0(s) = 2s^3 - 6s^2 + 5s - 1$, $q_1(s) = 6s^2 - 12s + 5$, $q_2(s) = 2(s - 1)/3$, and $q_3(s) = 1$. We have $q_0(0) = -1$, $q_1(0) = 5$, $q_2(0) = -2/3$, and $q_3(0) = 1$ for a total of 3 sign changes. We also have $q_0(1) = 0$, $q_1(1) = -1$, $q_2(1) = 0$, and $q_3(1) = 1$ for a total of 1 sign change. The difference in sign changes is 2, so $q(s) = 0$ has two real-valued roots on $[0, 1]$, which means the line intersects the curve. ■

7.4.2 POLYLINE APPROXIMATION

The root finding of the algebraic method can be computationally expensive. An attempt at reducing the time complexity is to approximate the curve by a polyline and find intersections of the line with the polyline. The curve polyline is obtained by subdivision (see Section A.8). The line-polyline tests that were discussed earlier in this