Space-Efficient Outlines from Image Data via Vertex Minimization and Grid Constraints

John D. Hobby

Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

When processing shape information derived from a noisy source such as a digital scanner, it is often useful to construct polygonal or curved outlines that match the input to within a specified tolerance and maximize some intuitive notions of smoothness, simplicity, and best fit. The outline description should also be concise enough to be competitive with binary image compression schemes. Otherwise, there will be a strong temptation to lose the advantages of the outline representation by converting back to a binary image format.

This paper proposes a two-stage pipeline that provides separate control over the twin goals of smoothness and conciseness: the first stage produces a specification for a set of closed curves that minimize the number of inflections subject to a specified error bound; the second stage produces polygonal outlines that obey the specifications, have vertices on a given grid, and have nearly the minimum possible number of vertices. Both algorithms are reasonably fast in practice, and can be implemented largely with low-precision integer arithmetic.

Space-Efficient Outlines from Image Data via Vertex Minimization and Grid Constraints

John D. Hobby

Bell Laboratories
Murray Hill, NJ 07974

When processing shape information derived from a noisy source such as a digital scanner, it is often useful to construct polygonal or curved outlines that match the input to within a specified tolerance and maximize some intuitive notions of smoothness, simplicity, and best fit. The outline description should also be concise enough to be competitive with binary image compression schemes. Otherwise, there will be a strong temptation to lose the advantages of the outline representation by converting back to a binary image format.

This paper proposes a two-stage pipeline that provides separate control over the twin goals of smoothness and conciseness: the first stage produces a specification for a set of closed curves that minimize the number of inflections subject to a specified error bound; the second stage produces polygonal outlines that obey the specifications, have vertices on a given grid, and have nearly the minimum possible number of vertices. Both algorithms are reasonably fast in practice, and can be implemented largely with low-precision integer arithmetic.

1 Introduction

In fields such as image processing, font generation, and optical character recognition, it is often useful to extract outlines from a digital image. In the case of binary images, a naive approach to this problem yields jagged polygonal outlines that have large numbers of very short edges as shown in Figure 1a. Such outlines are undesirable because the jagged appearance is due to noise introduced by the scanning process. A suitable polygonal approximation such as in Figure 1b has a smoother appearance and fewer vertices. Filtering out the noise and reducing the vertex count makes the outlines more useful and speeds subsequent processing.

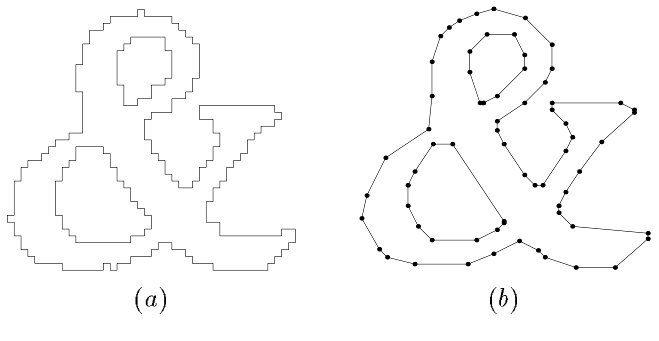

Figure 1: (a) A simulated character shape outline as might be obtained from a digital image; (b) a polygonal approximation with a smoother appearance.

There are many polygonal approximation algorithms that convert input such as Figure 1a into output reminiscent of Figure 1b. This is probably due to the ill-defined nature of the problem and the competing goals of speed, vertex minimization, and faithfulness to the input. Often neglected is the goal of smoothing out the "jaggies" caused by noise from the scanning process. A carefully-designed polygonal approximation algorithm can actually *improve* the outlines by eliminating noise-induced "jaggies." The algorithm that generated Figure 1b tries to do this by emphasizing smoothness and quality of fit with little regard to reducing the vertex count [7].

The goal of the present work is to retain the advantages of smooth, well-fitting outlines, but simplify them enough so that they can be stored compactly. What are these advantages? Consider an electronic library project involving scanned document images as described by O'Gorman [14]. When a document is first scanned into the system, the resulting page images need to be processed to remove noise and correct for the skew angle. The skew comes about because the lines of text are not likely to be perfectly aligned with the scanner's pixel grid. It is important to correct the skew because small angle rotations lead to annoying image defects, especially when viewed on a computer screen. Agazzi, Church, and Gale [1] have found that a good way to do the required rotations is to use the algorithm from [7] to generate outlines, then rotate the outlines and scan-convert them to get another binary image.

It would be much better to retain the outlines, since the page images need to be rescaled to cope with a wide range of output devices as explained in [1]. In other words, we need simpler outlines to avoid the destructive process of scan-converting and then later rescaling the resulting bitmaps. Hence we want to retain most of the smoothness and accuracy of the outlines from [7], but allow them to be stored in roughly the same space as is required by the best available compression techniques for binary images.

We therefore refer to the algorithm from [7] as *Stage 1*, and add a *Stage 2* that post-processes the Stage 1 output to reduce the vertex count and restrict the vertex coordinates to an integer grid. This creates a trade-off between the quality of the outlines produced and the space required to store them. The entire process can be viewed as an attempt to find the most concise outlines possible subject to constraints on the smoothness and quality of fit.

The Stage 1 algorithm achieves smoothness by minimizing the number of inflections in the output [7]; i.e., vertices can be classified as left turns or right turns and the number of alternations between the two is minimized subject to a bound on the approximation error. The only other approach that minimizes inflections is Montanari's idea of minimizing the perimeter subject to the error bounds [13]. (See also the work of Sklansky [24, 25, 27].) Unfortunately, this tends to maximize the error rather than minimize it, since minimizing the perimeter demands taking the extreme inside track when going around a curve. Figure 2 illustrates this by comparing the minimum perimeter curve with the result of the Stage 1 algorithm. Note that the Stage 1 output includes a sequence of trapezoids that define a class of minimum-inflection curves that stay within a tolerance of the original input.

Now consider the goal of minimizing the vertex count subject to some kind of pointwise error bound. Williams does this by keeping track of cones that intersect circular neighborhoods of successive data points [30]. Kurozumi and Davis simplify the problem by allowing gaps between segments so that a simple greedy algorithm based on convex hulls minimizes the number of output segments [11]. Dunham uses dynamic programming to select a minimal subset of the input vertices [2], and Imai and Iri use a graph-based algorithm to accomplish essentially the same thing [8]. Guibas, Hershberger, et. al. use computational geometry techniques and dynamic programming to minimize the vertex count under a variety of conditions [6]. The method of Sharaiha and Christofides recognizes digitized straight line segments and uses a visibility graph to construct a minimum-vertex approximation [23]. Pikaz and Dinstein use breadth-first search to select a minimal subset of the input vertices, and use the Manhattan metric to measure pointwise error [19].

Kahan and Snoeyink give theoretical bounds on the vertex count for a polygonal path through a simple polygon with vertices restricted to an integer grid [9], but their algorithms are not intended to give good results for polygons like those in Figure 2b and the results presented later will bear this out.

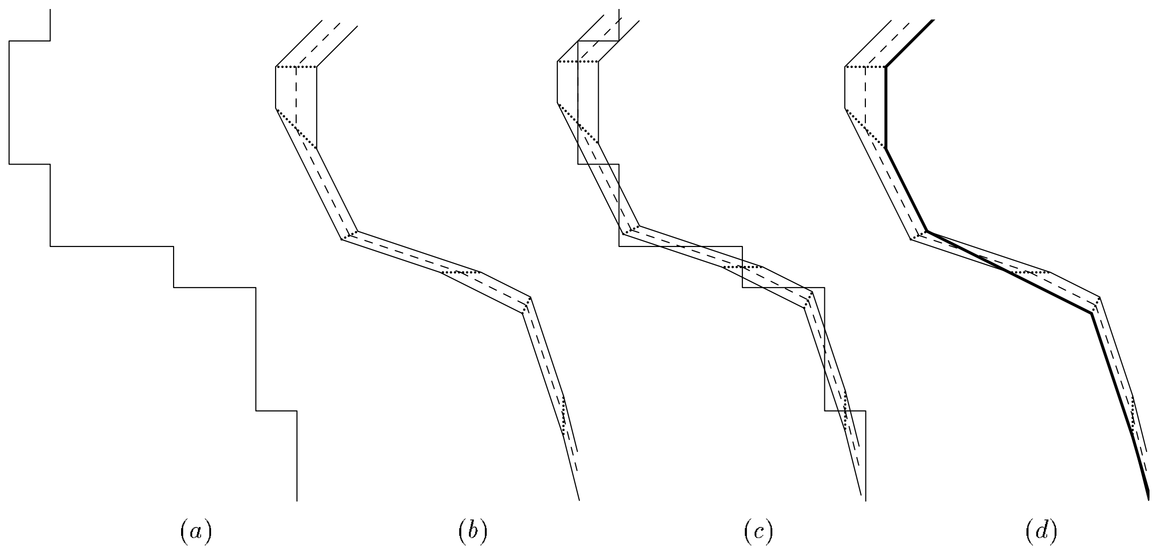

Figure 2: (a) Part of an outline extracted from a binary image; (b) corresponding output from the Stage 1 algorithm with the best polygonal approximation marked by a dashed line; (c) the superposition of (a) and (b); (d) the Stage 1 output with the minimum perimeter path shown as a heavy line.

Instead of minimizing the vertex count, Pavlidas uses Newtonian iteration to minimize the total squared error subject to a constraint on the number of output segments [15]. The segments are not required to meet exactly. Such gaps can be avoided by settling for approximately minimum error [17]. Another approach by Ray and Ray is to select each segment to minimize an objective function based on the segment length minus a penalty term for the sum of the pointwise errors [21].

If the vertex count only needs to be approximately minimum, we can use a simple greedy algorithm that runs in linear time. Tomek does this for functions of one variable by ensuring that a slab centered on the new segment contains as many data points as possible [28]. Pavlidas extends this algorithm to polygons on the plane [16]. Sklansky and Gonzalez achieve a similar result by maintaining a cone that describes the set of allowable directions for the next segment [26]. Leung and Yang do the same thing with a slightly different technique for maintaining the set of directions [12].

There are also a lot of polygonal approximation algorithms that do not try to minimize the vertex count. Ramer's recursive subdivision algorithm is a classic example [20]. Wall and Danielsson's method is fast and popular in application domains such as optical character recognition [29]. It chooses the next output vertex by keeping track of the area between the segment and corresponding part of the input polygon.

None of the algorithms cited above impose grid constraints except in a very theoretical context or by restricting the output vertices to be a subset of the input vertices. In addition, none of them try to maximize smoothness and quality of fit the way the Stage 1 algorithm does. Applications that benefit from such high-quality polygonal approximations include optical character recognition (OCR), understanding engineering drawings, robotics, and processing fingerprints. The more concise output generated by the two stage algorithm could benefit any of these applications, since they all involve processing the extracted outlines, and reducing the vertex count speeds up the processing.

A major area where vertex minimization and grid constraints are important is data compression. Virtually any image representation scheme that involves outlines could benefit from this. For instance, the present algorithm has been considered for very low bit-rate video encoding [3].

In the electronic library application, another possible benefit from storing documents in outline format is to provide input for an OCR process. Intense commercial interest in optical character recognition has lead to a wide variety of competing systems, and some of them do convert input images into outline form. Since the large number of "typographical errors" introduced by typical OCR systems are a major limitation, it is important to have the best possible outlines for those systems that use outlines.

Would any of these applications benefit from spline approximations? We do not consider this because the question is outside the scope of this paper and there is evidence that spline approximations would be of marginal utility for the application to scanned document images. The character shapes in a 400 dot-per-inch image with 10 point text are only about 40 pixels high, and they do not tend to have big sweeping curves. Figure 1b is an extreme case, but even there, it is hard to see how a single curve segment could replace more than five or ten edges.

Section 2 discusses the interface between Stage 1 and Stage 2 and explains how Stage 1 simplifies the application of grid constraints in Stage 2. The next three sections cover the Stage 2 algorithm: Section 3 explains how to scan the grid points visible from a given view point and chose the one that allows the best potential for further progress; Section 4 discusses how to handle the competing demands of maximizing forward and backward visibility at an inflection; and Section 5 presents the main algorithm for Stage 2. Next, Section 6 discusses an intermediate pipeline stage that provides better control over the approximation error in Stage 2. Although dynamic programming techniques could probably be used to minimize the vertex count, the experimental results in Section 7 suggest that it is much better to settle for near minimum since this produces a reasonably fast algorithm. Finally, Section 8 gives a few concluding remarks and the appendix gives proofs that the casual reader may want to skip.

2 The Output of Stage 1

Since the Stage 1 algorithm is described in [7], we need only to explain how to interface to it. The goal of Stage 1 is to take an input polygon and a "noise" tolerance ϵ, and guess what original outline might have produced the observed input. It does this by assuming that horizontal and vertical deviations of less than ϵ may be due to noise and that the original outline has the minimum possible number of inflections. For input like that in Figures 1a and 2a, ϵ should typically be slightly more $\frac{1}{2}$ pixel unit.

The input for Stage 1 is the tolerance ϵ and a polygon with vertices

$$(X_1, Y_1),\ (X_2, Y_2),\ (X_3, Y_3), \ldots$$

The tolerance is enforced by requiring the output to pass within ∞-norm distance ϵ of each vertex. Hence, the approximation must pass through squares of the form

$$\{\, (x, y) \mid X_i - \epsilon < x \leq X_i + \epsilon,\ Y_i - \epsilon < y \leq Y_i + \epsilon \,\} \tag{1}$$

as indicated by dashed lines in Figure 3a.

Figure 3b illustrates the corresponding output of Stage 1. It is a sequence of trapezoids of the form

$$R_1R_2L_2L_1,\ R_2R_3L_3L_2,\ R_3R_4L_4L_3, \ldots, \tag{2}$$

where the ith trapezoid has parallel edges R_iR_{i+1} and $L_{i+1}L_i$. (It is possible to have $R_i = R_{i+1}$ or $L_i = L_{i+1}$, in which case the trapezoid degenerates to a triangle.)

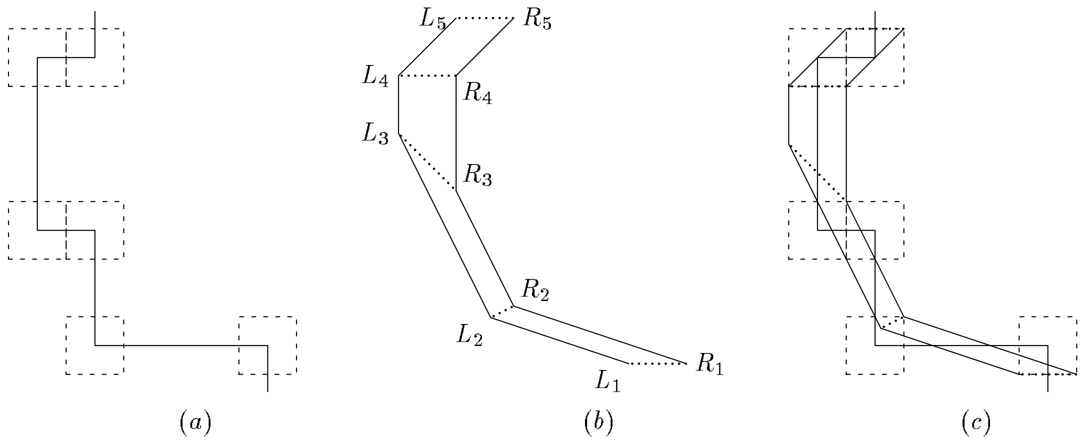

Figure 3: (a) Part of an outline extracted from a binary image with the squares defined by (1) indicated by dashed lines; (b) corresponding output from the Stage 1 algorithm; (c) the superposition of (a) and (b).

The polygonal approximation recommended in [7] is the one whose ith vertex is

$$\frac{L_i + R_i}{2}.$$

It obeys the input specifications in that it passes through the squares defined by (1) in order. In fact, [7] shows that this holds for any polygonal approximation that passes through the trapezoids defined by (2) in order and has no more inflections than the recommended one.

Part of the goal of Stage 2 will be to find a polygonal approximation that passes through the trapezoids and has vertices confined to some kind of grid. Hence we must be careful to choose the grid and the tolerance ϵ so that such a *grid-restricted polygonal approximation* is guaranteed to exist.

We can get an idea about how to do this by noting that most of the trapezoid vertices in Figure 3c lie at the corners of dashed squares. In particular, this holds for R_2, R_3, R_4, and R_5. Comparing Figure 3b with Figure 2d, we find that these are the vertices that occur on Montanari's minimum-perimeter path. Hence we arrange for the corners of the squares to lie at grid points, and hope that this will make enough of the trapezoid vertices lie at grid points so that the minimum perimeter path through the trapezoids will be a grid-restricted polygonal approximation.

If the initial outlines come from binary images, the X_i, Y_i coordinates are integer numbers of pixel units. Then we can choose the grid for the output vertices to be some integer factor K times finer than the pixel grid,[1] and choose ϵ to be a multiple of $1/K$ pixel units. This forces the corners of the squares to lie at grid points.

Montanari's minimum-perimeter path through (1) has vertices at the corners of the squares, hence it is a grid-restricted polygonal approximation. In fact, these vertices are *concave trapezoid vertices*, i.e., they are like R_2, R_3, and R_4 in Figure 3b, not like the *convex trapezoid vertices* L_2, L_3, L_4. The critical property is that the concave trapezoid vertices must lie at grid points.

It is clear from [7] that this minimum-perimeter path can be made to pass through the trapezoids from Stage 1. Minor modifications are needed because [7] uses strict inequalities in (1) and treats the trapezoid edges (solid lines in Figure 3b) as out of bounds. We have the following theorem:

[1]The choice of K is discussed in Section 7. It should probably be ≤ 4 in order to reduce the number of bits needed to store the output coordinates.

Theorem 2.1 *Suppose that the input for Stage 1 has integer vertex coordinates on a pixel grid and that the output grid is finer by some integer factor K, where the tolerance ϵ is a multiple of $1/K$. If the Stage 1 algorithm is modified as explained above, then there is a minimum-inflection polygonal path through the trapezoid sequence with all vertices at output grid points.*

3 The Best Visible Grid Point Subproblem

The Stage 2 algorithm takes a trapezoid sequence as described in Section 2 and tries to find a grid-restricted polygonal path that passes through the trapezoids in order and has few vertices. Begin by considering the simple case where there are no inflections, so we are given a sequence of trapezoids whose outer and inner boundary make only right turns (or only left turns). If there are only right turns, the vertices on the outer boundary are $L_1, \ldots, L_k$ and the vertices on the inner boundary are $R_1, \ldots, R_k$ as in Figure 3b. (In the left-turning case, $L_1, \ldots, L_k$ are on the inner boundary and $R_1, \ldots, R_k$ are on the outer boundary). Refer to these inner and outer boundaries as the *inner chain* and *outer chain*, respectively.

We also assume we are given a start point P_{start}, which we assume lies in trapezoid $R_1R_2L_2L_1$, and a goal point P_{goal}, which we assume lies in the trapezoid $R_{k-1}R_kL_kL_{k-1}$. Our goal is to find a polygonal path $P_0, P_1, \ldots, P_m$ with $P_0 = P_{\text{start}}$ and $P_m = P_{\text{goal}}$ that stays inside the trapezoids and has a minimum number of vertices. Moreover, the vertices of the path are confined to lie on a given grid.

Without the grid constraints, we would just start at P_0, find an "optimal" visible point P_1^{opt} as shown in Figure 4, and use the same strategy to generate P_2^{opt}, P_3^{opt}, etc. until finding a P_i^{opt} from which P_{goal} is visible. All such optimal points P_i^{opt} lie on the outer chain as does P_1^{opt} in the figure.

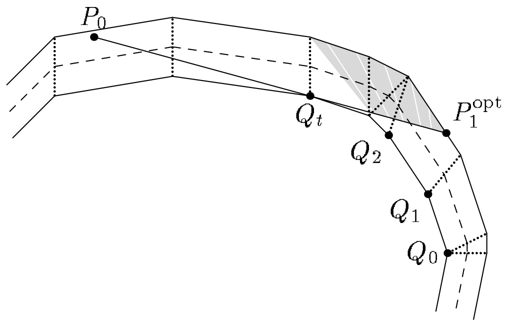

Figure 4: Input trapezoids for Stage 2 and part of the region to search when finding the best grid point visible from P_0. All points on a single white stripe are equally desirable.

The region visible from P_0 is delimited by a tangent line from P_0 to the inner chain. (There are two tangent tangent lines and we want the "forward" one). Since such tangent lines can be ordered according to their point of tangency, points such as P_0 can be ranked according to the limit of visibility from there. When two tangent lines have the same point of tangency, their direction angles can be used to break the tie.

The same ranking scheme applies when points P_i are restricted to be grid points, but it is necessary to search for the best grid point visible from the previous P_i. The region to search for the next point P_{i+1} is determined the the forward tangent line from P_i to the inner chain. Call this the *search polygon for* P_i. It is bounded by the tangent line and part of the outer chain. The shaded region in Figure 4 is part of the search polygon for P_0 and the rankings are suggested by white stripes. Points Q_0, Q_1, ..., Q_t are the possible points of tangency that determine the ranking for points in the search polygon. They are vertices on the inner chain.

The simplest method for finding the best grid point visible from a point such as P_0 is to scan each row of grid points in the search polygon and stop when the scanning process has covered all points that are as good as the current best grid point. This turns out not to be the best way to find good visible grid points, but it is an important subproblem that we will need later. Hence, Section 3.1 shows how suitable affine transformations can make this simple scanning process reasonably efficient.

Section 3.2 shows how to use continued fractions to find the best visible grid point, and Section 3.3 generalizes this to cover a subproblem that will be useful when dealing with inflections.

3.1 Scanning a Simple Polygon for Good Grid Points

Consider the problem of scanning the grid points in a closed convex polygon. Since the polygon could be very long and narrow and rotated by any angle, we first transform the coordinate system to reduce to simpler cases. The idea is to choose an affine transformation that maps grid points to grid points and causes the polygon to occupy a significant fraction of its bounding box.

Start with a polygon $\mathcal{P}$ and a direction vector $\bar{d}$ in which scanning should proceed. In other words, the definition of a "good" grid point is one where the dot product with $\bar{d}$ is as small as possible. (This is not the ranking we get from the best visible grid point subproblem—it corresponds to pretending that the white lines in Figure 4 are parallel).

We start by constructing supporting lines for $\mathcal{P}$ perpendicular to $\bar{d}$ and completing a circumscribing parallelogram as shown in Figure 5. This is done by taking the points of support A and C, and finding a pair of supporting lines parallel to the dashed line AC. Since the resulting parallelogram has twice the area of the convex quadrilateral $ABCD$, it has at most twice the area of $\mathcal{P}$. (Somewhat better ratios could be obtained via the more sophisticated techniques of Schwarz et. al. [22].)

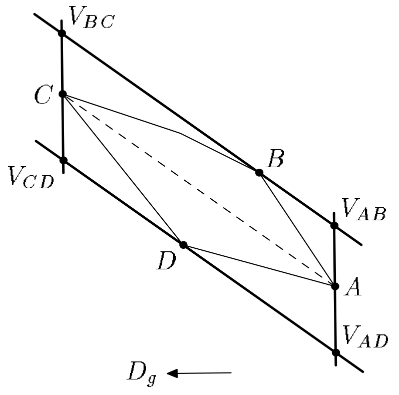

Figure 5: The construction for a parallelogram that circumscribes $\mathcal{P}$, has two edges perpendicular to $\bar{d}$, and has area no more than twice the area of $\mathcal{P}$.

If the circumscribing parallelogram is $V_{AB}V_{BC}V_{CD}V_{AD}$ as shown in Figure 5, the area of the bounding box is

$$(|x_1| + |x_2|)(|y_1| + |y_2|), \tag{3}$$

where $(x_1, y_1) = V_{AB} - V_{AD}$ and $(x_2, y_2) = V_{CD} - V_{AD}$. To find a transformation matrix T that maps grid points to grid points and minimizes (3), we can use a sequence of elementary transformations that alternate between the goals of reducing $|x_1| + |x_2|$ and $|y_1| + |y_2|$.

For example, suppose the construction in Figure 5 gives $V_{AB} - V_{AD} = (-5.3, 3.7)$ and $V_{CD} - V_{AD} = (-13.4, 9.1)$. Initially, we have $(x_1, y_1) = (-5.3, 3.7)$, $(x_2, y_2) = (-13.4, 9.1)$, and $T = \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)$. Next, choose an integer k and transform each (x, y) by replacing x with $x + ky$ so as to minimize $|x_1| + |x_2|$. The best choice is $k = 1$, giving $(x_1, y_1) = (-1.6, 3.7)$, $(x_2, y_2) = (-4.3, 9.1)$, and $T = \left(\begin{smallmatrix}1&1\\0&1\end{smallmatrix}\right)$. To minimize $|y_1| + |y_2|$ via a transformation $y \leftarrow y + lx$, choose $l = 2$ so that $(x_1, y_1) = (-1.6, 0.5)$, $(x_2, y_2) = (-4.3, 0.5)$, and $T = \left(\begin{smallmatrix}1&1\\2&3\end{smallmatrix}\right)$. To minimize $|x_1|+|x_2|$ via a transform $x \leftarrow x+ky$, one of the optimal k values is $k = 4$. This gives $(x_1, y_1) = (0.4, 0.5)$, $(x_2, y_2) = (-2.3, 0.5)$, and $T = \left(\begin{smallmatrix}9&13\\2&3\end{smallmatrix}\right)$. Subsequent transformations do not help, so the algorithm terminates. Algorithm 1 formalizes this process.

Algorithm 1 Construct a 2×2 transformation matrix T that maps the set of integer grid points $\mathbb{Z}^2$ to itself and tries to map (x_1, y_1) and (x_2, y_2) so as to reduce the bounding box area (3) as much as possible.

```
1. Set $T_{11} = T_{22} = 1$ and $T_{12} = T_{21} = 0$ so $T$ is the $2 \times 2$ identity matrix.
2. Choose $k \in \mathbb{Z}$ to minimize $|x_1{+}ky_1| + |x_2{+}ky_2|$. In case of ties, prefer $k = 0$ if possible. Then
   set $x_1 = x_1 + ky_1$, $x_2 = x_2 + ky_1$, $T_{11} = T_{11} + kT_{21}$, and $T_{12} = T_{12} + kT_{22}$.
3. Choose $l \in \mathbb{Z}$ to minimize $|y_1{+}lx_1| + |y_2{+}lx_2|$. In case of ties, prefer $l = 0$ if possible. Then
   set $y_1 = y_1 + lx_1$, $y_2 = y_2 + lx_1$, $T_{21} = T_{21} + lT_{11}$, and $T_{22} = T_{22} + lT_{12}$.
4. Stop if $k = l = 0$; otherwise go to Step 2.
```

The purpose of Algorithm 1 is to find a transformation matrix T that reduces the problem of scanning grid points the polygon $\mathcal{P}$ to the case when $\mathcal{P}$ occupies a significant fraction of its bounding box.[2] Once this is achieved, an ordinary scan-conversion algorithm can find the grid points in the transformed $\mathcal{P}$ without considering more than a small multiple of $\sqrt{\text{area}(\mathcal{P})}$ scan lines. Some additional care is required to scan in $\bar{d}$ order, but the details are unimportant in practice.

Algorithm 1 is fast in practice because the time for each iteration of Steps 2 and 3 is a small constant, and the number of iterations seldom exceeds two or three. (The running time can be proven by noting that the bounding box area (3) gets reduced geometrically.[3]). Counting the cost of scan-conversion, the time bound for finding k grid points in an n-vertex polygon $\mathcal{P}$ with the aid of the transformation computed by Algorithm 1 is

$$O(n + k) \tag{4}$$

plus some overhead that depends weakly on $\mathcal{P}$.

3.2 Using Continued Fractions to Find the Best Visible Grid Point

The polygon searching algorithm in Section 3.1 is a useful tool, but it is not ideally suited to the problem of finding the best visible grid point as illustrated in Figure 4. The figure contains clues to the nature of the trouble: the shaded polygon has an arbitrary boundary, and the white stripes are not parallel. More precisely, the search polygon for P_0 can be large and we would like to search just a portion of it, but it is not clear how big a portion to search or what direction to use for $\bar{d}$. There is no unique $\bar{d}$ that is always consistent with the notion of ranking grid points according to the tangent line they lie on.

[2] In fact, the transformed version of $\mathcal{P}$ occupies at least $\frac{1}{4}$ of its bounding box. A proof of this result is available on request.

[3] This proof is also available on request.

We want to find the best grid point visible from the given point. Without loss of generality, we assume that the given point is P_0. Recall that P_1^{opt} is the best visible point without grid constraint, and let Q_0 be the vertex on the inner chain where the forward tangent line from P_1^{opt} touches the inner chain as shown in Figure 4. Let $Q_1, Q_2, \ldots, Q_t$ be the other inner chain vertices where the forward tangent from a point in the search polygon for P_0 can touch the inner chain. We can assume that the vertices Q_i are defined so that for any $i < t$, the portion of the inner chain between Q_i and Q_{i+1} is a nontrivial straight line segment. Note that the vertices Q_i are concave trapezoid vertices, and hence they are grid points.

For the moment we only consider the visible grid points whose tangent line to the inner chain goes through Q_0. If there is no such grid point, then we repeat the process for Q_1, Q_2, Q_3, etc., until a visible grid point is found. Of all the visible grid points with a tangent line through Q_0, which is the best one? This must be the one such that the direction from Q_0 to that grid point is closest to the direction from Q_0 to P_1^{opt}. Since Q_0 is a grid point, we must approximate the direction from Q_0 to P_1^{opt} by a rational direction (from Q_0), with the constraint that the grid point defining the rational direction should lie in the search polygon of P_0. This calls for continued fractions.

The idea of using continued fractions is that they allow us to find rational directions between a direction $D_{\mathrm{st}} = P_1^{\mathrm{opt}} - Q_0$ and $D_{\mathrm{fin}} = P_0 - P_1^{\mathrm{opt}}$. Our goal is to find a visible grid point for which the direction from Q_0 is as close to D_{st} as possible. We will do this by starting from Q_0 and stepping from grid point to grid point, where each step is along a rational direction chosen to be as close to D_{st} as possible subject to the constraint that it must be possible to take a step in that direction without crossing the outer chain. The chosen directions will get further and further from D_{st}, but we need to keep them between D_{st} and D_{fin} in order to guarantee that we keep approaching the search polygon of P_0.

Continued fractions provide an efficient way to scan for short rational direction vectors between D_{st} and D_{fin}. The goal is to find a sequence of rational directions $D_0, D_1, D_2, \ldots, D_k$ from which the desired short rational direction vectors can be constructed. We assume that D_0 is closest to D_{st} and D_k is closest to D_{fin}. Figure 6a shows an example for $D_0 = (-1, 7)$ and $D_3 = (-11, 3)$. The desired short rational directions are the vectors from $(0, 0)$ to the integer grid points that are marked by dots along the D_0D_1, D_1D_2, and D_2D_3 segments. A key property of the construction is that these rational direction vectors are "as short as possible" in the sense the shaded region in the figure contains no grid points.

When using this construction, directions closest to D_0 are considered "best." We start at Q_0 in Figure 6b and take one step in the best possible rational direction, and repeat this process until reaching $\ell(P_1^{\mathrm{opt}}P_0)$, where $\ell(AB)$ is the line defined by segment AB and directed from A to B. The first step goes to $\bar{R}_1$ because $\bar{R}_1'$ is out of bounds. The next step goes to $\bar{R}_2$, and the process terminates there because $\bar{R}_2$ is on the correct side of $\ell(P_1^{\mathrm{opt}}P_0)$. The argument that $\bar{R}_2$ must be the best grid point visible from P_0 is based on the fact that there are no grid points in the interior of the shaded region.

The continued fraction algorithm can be thought of in terms of rational directions by writing (q, p) in place of each fraction p/q. Expressed in this form, the algorithm generates a sequence of approximations $\bar{D}_0, \bar{D}_1, \bar{D}_2, \ldots, \bar{D}_k$ to a rational direction $\bar{D}$, where $\bar{D}_0 = (1, 0)$, $\bar{D}_1 = (0, 1)$, and $\bar{D}_k = \bar{D}$. The algorithm also generates integer coefficients c_i, where $\bar{D}_i = \bar{D}_{i-2} + c_i\bar{D}_{i-1}$. The approximation sequences

$$\bar{D}_0, \bar{D}_2, \bar{D}_4, \ldots, \bar{D}_{2\lfloor (k-1)/2 \rfloor}, \ \bar{D}_k \tag{5}$$

and

$$\bar{D}_1, \bar{D}_3, \bar{D}_5, \ldots, \bar{D}_{2\lfloor k/2 \rfloor - 1}, \ \bar{D}_k \tag{6}$$

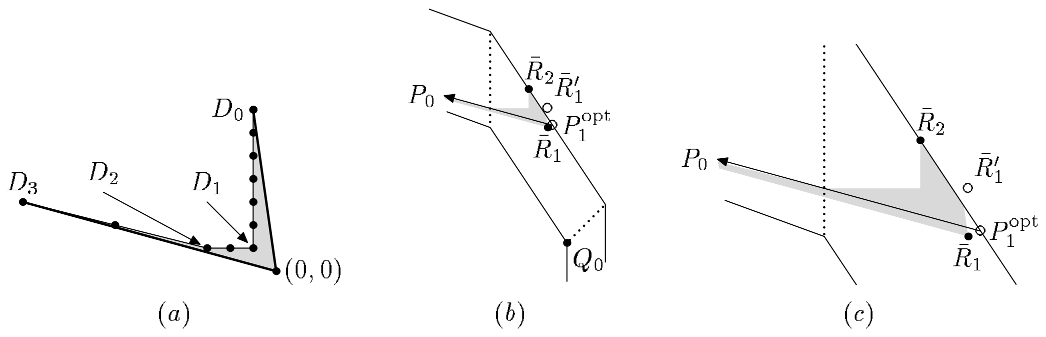

Figure 6: (a) Rational directions between $D_0 = (-1, 7)$ and $D_3 = (-11, 3)$ with a grid-point-free region shaded. Dots mark the ends of the direction vectors. (b) How to use the rational directions to find the best grid point $\bar R_2$ that is visible from P_0. (Point P_0 should be some distance away in the direction of the arrow.) (c) A close-up of the region near $\bar R_1$ and $\bar R_2$.

approach $\bar D$ from opposite sides. Adding intermediate directions

$$\bar D_{i-2} + \bar D_{i-1},\ \bar D_{i-2} + 2\bar D_{i-1},\ \bar D_{i-2} + 3\bar D_{i-1}, \ldots, \bar D_{i-2} + (c_i - 1)\bar D_{i-1}$$

between each $\bar D_{i-2}$ and $\bar D_i$ where $2 \le i \le k$ refines (5) and (6) so that consecutive approximates $\bar D'$ and $\bar D''$ are *CF-neighbors.* Any rational rational direction between $\bar D'$ and $\bar D''$ can be written as

$$m\bar D' + n\bar D'',$$

where m and n are positive integers. We say that rational directions $D_i, D_{i+1} \in \mathbb{Z}^2$ are *connected by CF-neighbors* if there exists an integer c_i such that

$$D_{i+1} + \frac{j(D_i - D_{i+1})}{c_i}, \quad \text{for} \quad 0 \le j \le c_i \tag{7}$$

are rational directions in $\mathbb{Z}^2$ and the directions for consecutive j values are CF-neighbors.

For instance, $D_2 = (1, 3)$ and $D_3 = (3, 11)$ are connected by CF-neighbors because letting $c_2 = 2$ in (7) generates a sequence

$$(1, 3),\ (2, 7),\ (3, 11),$$

where $(1, 3)$ and $(2, 7)$ are CF-neighbors and so are $(2, 7)$ and $(3, 11)$. (The easiest way to see this is to note that matrices $\left(\begin{smallmatrix}1 & 3\\ 2 & 7\end{smallmatrix}\right)$ and $\left(\begin{smallmatrix}2 & 7\\ 3 & 11\end{smallmatrix}\right)$ have unit determinants.)

We need an algorithm for finding a sequence of directions connected by CF-neighbors, given starting and ending directions $D_{\rm st}$ and $D_{\rm fin}$. The idea is to start with the continued fraction expansion for $D_{\rm st}$ and then use directions from (5) or (6). If $D_{\rm fin}$ lies in the first quadrant, there will be some j for which $D_{\rm fin}$ is between D_{j-2} and D_j. A suitable 2 by 2 transformation matrix allows the remaining directions to be found by taking the continued fraction expansion for a transformed version of $D_{\rm fin}$ and then applying the inverse transform to directions from (5) or (6).

Suppose $D_{\rm st} = (7, 1)$ and $D_{\rm fin} = (3, 11)$. Taking the continued fraction expansion for $\frac{1}{7}$ gives

$$(1, 0),\ (1, 0),\ (7, 1) \quad \text{and} \quad (0, 1),\ (7, 1)$$

for (5) and (6). In this case it is (6) that gives directions between $D_{\rm st}$ and $D_{\rm fin}$. Using (7) with $D_1 = (0, 1)$ and $D_3 = (7, 1)$, we find that the CF-neighbors around $D_{\rm fin}$ are $(0, 1)$ and $(1, 1)$. Next we output (6) down to $D_{\rm fin}$, giving just $(7, 1)$.

The transformation $T(x,y) = (x, y-x)$ maps directions between $(0,1)$ and $(1,1)$ into the first quadrant. Since $T(D_{\text{fin}}) = (3,8)$, we take the continued fraction expansion for $\frac{8}{3}$, obtaining

$$(1,0),\ (1,2),\ (3,8) \quad \text{and} \quad (0,1),\ (1,3),\ (3,8)$$

for (5) and (6). Applying T^{-1} to (5) gives $(1,1)$, $(1,3)$, $(3,11)$. Hence we output these directions and terminate.

Algorithm 2 formalizes this process. It assumes that D_{st} belongs to the first quadrant. If not, it suffices to rotate by a multiple of 90°. (Such a rotation reduces the situation shown in Figure 6a to the example given above.)

Algorithm 2 Given initial and final directions D_{st} and D_{fin} with D_{st} in the first quadrant, output a sequence of rational directions such that the first one agrees with D_{st}, the last one agrees with D_{fin}, and consecutive outputs are connected by CF-neighbors.

1. Use the continued fraction algorithm to approximate D_{st}.
2. If D_{fin} is clockwise from D_{st}, output the entries of (5) in reverse order, stopping when some $\bar{D}_i$ is not clockwise from D_{fin}. This may require extending (5) by prepending $(0,-1)$ or $(-1,0)$, $(0,-1)$.
3. If D_{fin} is counter-clockwise from D_{st}, output the entries of (6) in reverse order, stopping when some $\bar{D}_i$ is not counter-clockwise from D_{fin}. This may require extending (6) by prepending $(-1,0)$ or $(0,-1)$, $(-1,0)$.
4. Let $\bar{D}_j$ be the last direction in the output so far, and find the CF-neighbors $\bar{D}'$, $\bar{D}''$ between $\bar{D}_j$ and $\bar{D}_{j+1}$ such that D_{fin} is between $\bar{D}'$ and $\bar{D}''$. Then find a, b such that $D_{\text{fin}} = a\bar{D}' + b\bar{D}''$. and use the continued fraction algorithm to approximate (a,b). Output the entries of (5) transformed by the matrix that maps $(1,0)$ to $\bar{D}'$ and $(0,1)$ to $\bar{D}''$.

The running time for Algorithm 2 is dominated by the continued fraction expansions of D_{st} and D_{fin}. This is known to be

$$O\big(\log(\|D_{\text{st}}\|) + \log(\|D_{\text{fin}}\|)\big),$$

where $\|\cdot\|$ denotes any standard vector norm [10].

The output of Algorithm 2 can be used as described above to make successive steps from a starting point Q_0 toward an initial goal P_1^{opt}, turning as necessary to avoid crossing the outer chain. Call the directed line from P_1^{opt} toward P_0 is the *near line*, and refer to the outer chain as the *far path*. The near line defines a limit of visibility from P_0 that the stepping process must reach.

If we want the best grid point that can see Q_0, the region to be searched is the dark-shaded region in Figure 7. Initialize a boundary line ℓ_{lim} to $\ell(Q_0Q_1)$ and use Algorithm 2 to find directions $D_0, D_1, \ldots, D_n$, where D_0 agrees with $P_1^{\text{opt}} - Q_0$ and D_n agrees with $P_0 - P_1^{\text{opt}}$. The main loop maintains a current point P_c (initially Q_0) and tries to add some positive combination of D_i and D_{i+1}, where i is a loop variable that starts at zero and is incremented after each iteration.

The body of the main loop starts by trying to find a pair of CF-neighbors D, D' between D_i and D_{i+1}, where $P_c + D'$ is on the wrong side of the far path and $P_c + D$ is not. If this cannot be done, we go on to the next iteration. Otherwise, a positive integer multiple of D is added to P_c so as to approach the near line without crossing the far path. It may also be necessary to look for grid points of the form $P_c + jD + kD'$ with $j, k \geq 1$. If there are any such points on the correct sides of the near line and the far path, we adjust ℓ_{lim} to restrict future attention to the region where better grid points could be found.

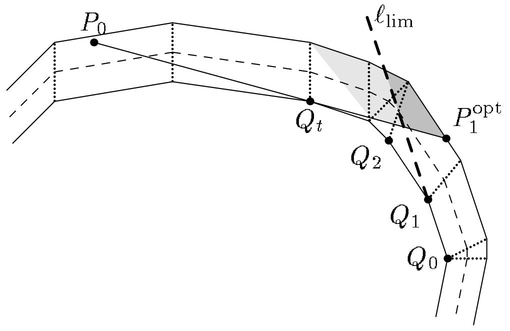

Figure 7: Input trapezoids for Stage 2 and the regions searched by Algorithm 3 when it is started at Q_0 (dark region) and Q_1 (lighter region).

Algorithm 7 formalizes the above procedure. It requires a function *test_pts*() that takes a uniformly-spaced sequence of integer vectors and finds the last vector V for which $P_c + V$ is not outside of the far path, where P_c is the current grid point. In order to do this with sequences of the form (7) from the output of Algorithm 2, that algorithm should output the parameters c_0, c_1, c_2, $\ldots$, c_{n-1} that appear in (7). It can easily do so because the continued fraction algorithm computes these parameters.

Algorithm 3 Find the grid point that is visible from a given grid point P_0 and has the best visibility in the direction of trapezoid vertex Q_0 as shown in Figure 7.

1. Use Algorithm 2 with $D_{\text{st}} = P_1^{\text{opt}} - Q_0$ and $D_{\text{fin}} = P_0 - P_1^{\text{opt}}$ to produce directions D_j for $0 \le j \le n$ and integer parameters c_j for $0 \le j < n$. Then set $i = 0$, $P_c = Q_0$, $\ell_{\text{lim}} = \ell(Q_0Q_1)$, and give P_b a null value.
2. Apply the *test_pts*() function to (7). If it fails, increment i and repeat this step; otherwise, let D be the function result and let $D' = D + (D_i - D_{i+1})/c_i$.
3. If $P_c + D$ and P_1^{opt} are on opposite sides of ℓ_{lim}, stop. The best visible grid point is P_b.
4. If necessary, search for points $P_c + jD + kD'$ that lie on or between the near line and the far path and not on the wrong side of ℓ_{lim}. If successful, let P_b be the best point found and let $\ell_{\text{lim}} = \ell(Q_0P_b)$.
5. Find the smallest integer l for which $P_c + lD$ is across the near line. Then apply the *test_pts*() function to the sequence D, $2D$, $3D$, $\ldots$, lD. Let the result be D'' and set $P_c = P_c + D''$.
6. If P_c and P_1^{opt} are on opposite sides of ℓ_{lim}, stop. The best visible grid point is P_b.
7. If $D'' = lD$, halt—the best visible grid point is P_c. Otherwise, go back to Step 2

Since Algorithm 3 starts with $\ell_{\text{lim}} = \ell(Q_0Q_1)$, the search is restricted to the region where Q_0 is the point of tangency for the tangent lines that determine which grid point is best. (This is the dark-shaded region in Figure 7). If there are no grid points in this region, Algorithm 3 will return a null value and it will have to be restarted with Q_1 playing the role of Q_0 so as to search the lightly-shaded region in Figure 7. The algorithm could be generalized to avoid starting over from scratch in such cases, but this turns out to be relatively unimportant in practice.

Note that Step 4 says to search a certain polygon "if necessary." Figure 8a shows an example where such a search is necessary. In this case, a D' step from P_c takes one across the far path but a $D + D'$ step goes to P' which is on the far path and hence "in bounds". If the far path is not directed into the D, D' sector when it crosses the D' ray from P_c, points such as P' in Figure 8b must be across the far path. There are two reasons for this: Step 2 of the algorithm guarantees that $P_c + D'$ is across the far path; and the far path direction at the D' ray from P_c and the lack of inflections prevent the far path from crossing the D-directed ray from $P_c + D'$. Hence the "if necessary" test in Step 4 should be a test of the far path direction at the as it crosses the D' ray from P_c.

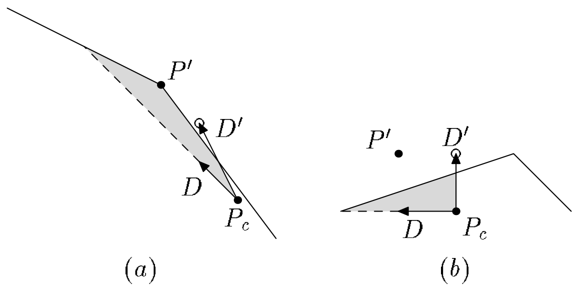

Figure 8: (a) A case where the far path direction crosses the D' ray from P_c with a direction in the D, D' sector, allowing the shaded region to contain a grid point P'. (b) a case where the far path direction avoids the D, D' sector and grid points such as P' need not be considered.

Since experimental evidence shows it is very seldom necessary to search the convex polygon described in Step 4, it is not worth developing a special algorithm for this purpose. It suffices to use Algorithm 1 in combination with an ordinary polygon scan-conversion algorithm as explained in Section 3.1.

Run time bounds for Algorithm 3 would not be particularly informative because of unlikely but theoretically expensive operations such as the search in Step 4. Refer to Theorem A.1 in Appendix A for a proof that Algorithm 3 finds the correct grid point.

3.3 The Trade-off between Forward and Backward Visibility

The previous discussion has centered on finding a grid point P_1 that is visible from P_0 and allows the next point P_2 to be placed as far from P_0 as possible. We now generalize this situation to make the above algorithms work in the presence of inflections. The problem is to search for grid points in a region defined by a tangent line such as the line $\ell(Q_t P_1^{\mathrm{opt}})$ that is tangent at Q_t in Figure 9. Since Algorithm 3 does not allow inflections in the far path, it may be necessary to trim the search region by extending the inflection edge as indicated by the heavy dashed line in Figure 9. (It is seldom necessary to extend the edge very far, hence only a tiny portion of the dashed line is above $\ell(Q_t P_1^{\mathrm{opt}})$ in the figure).

This time, two sets of tangent lines are used for ranking the grid points. The white stripes roughly parallel to $\ell(Q_t P_1^{\mathrm{opt}})$ are tangent lines with points of tangency at Q_t or at its predecessor Q_{t+1}. Such tangent lines measure visibility back towards Q_{t+1}, and they are ranked according to the direction of the tangent line. The other white stripes in the figure are tangent lines with points of tangency at successors of Q_t. The successor point marked Q_0 is the point of tangency for P_1^{opt}. Tangent lines through points such as Q_0 correspond to the best visibility in the forward direction. They are also ranked according to their direction. As the tangent line direction moves away from that of $\ell(P_1^{\mathrm{opt}} Q_0)$, the point of tangency moves back to Q_0's predecessors Q_1, Q_2, ... and the forward visibility gets worse.

We can find the trade-off between forward and backward visibility by simply running Algorithm 3 more than once. Each subsequent call to Algorithm 3 uses a near line based on the previous result

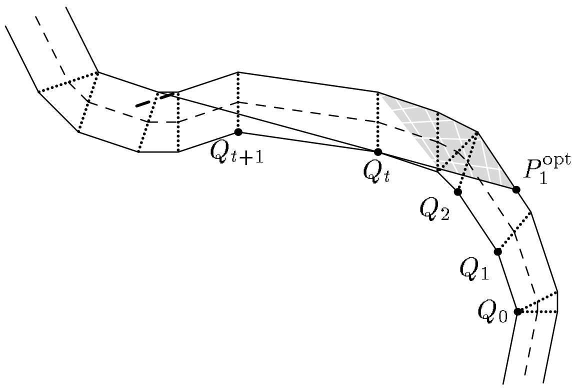

Figure 9: Input trapezoids for Stage 2 and the region to search when finding the trade-off between forward and backward visibility. All points on a single white stripe are equally desirable for the one direction or the other.

as shown in Figure 10. First, Algorithm 3 finds a grid point P_1 that has best forward vision in the region delimited by the line $\ell(Q_t P_1^{\mathrm{opt}})$. Next we restrict the region by using $\ell(Q_t P_1)$ as the near line, and use Algorithm 3 to find a grid point P_2 that has the best forward vision in the restricted region.[4] We could then find another grid point by using $\ell(Q_{t+1}P_2)$ as the near line. In order for Algorithm 3 to choose P_1, the dark shaded region in Figure 10 must be free of grid points since any grid points there would have better forward vision than P_1. Similarly, the light shaded region must be free of grid points in order for P_2 to be chosen.

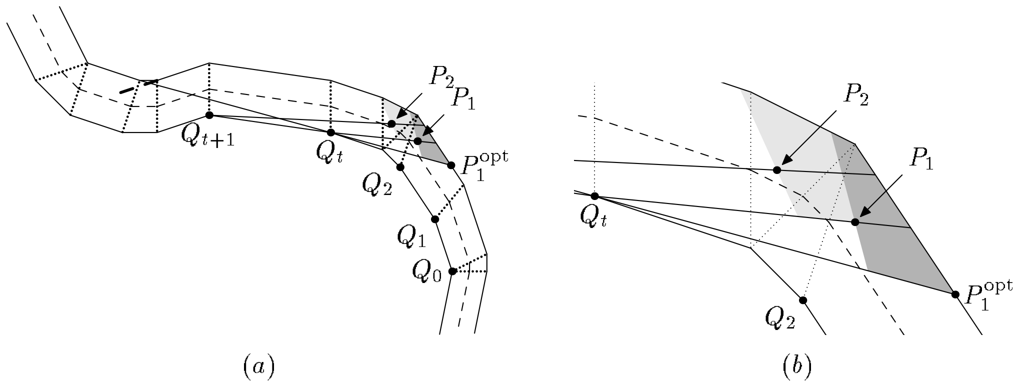

Figure 10: (a) Input trapezoids for Stage 2 and the near lines to use when finding the first two points P_1 and P_2 on trade-off between forward and backward visibility. (b) A close-up of the area near P_1 and P_2. The interiors of the shaded regions are free of grid points.

Each P_i has the best possible forward vision subject to the constraint that it be outside of the tangent line through P_{i-1}. This is exactly the same as requiring P_i to have better backward vision than P_{i-1}. This generates the trade-off between forward and backward vision if we are careful to require P_i to be strictly outside of the tangent line through P_{i-1}. The following theorem summarizes the above argument:

Theorem 3.1 *Suppose Algorithm 3 has found the grid point P_1 that has best forward visibility and lies in the region defined by a tangent line $\ell(Q_t P_1^{\mathrm{opt}})$ as shown in Figure 10. Then running*

[4] In practice, it would be best to modify Algorithm 3 to take advantage of the grid point P_1 on the near line.

Algorithm 3 $j-1$ more times as explained above generates a sequence of grid points P_2, P_3, P_4, $\ldots$, P_j, where each P_i has the best possible forward vision subject to the condition that it have better backward vision than P_{i-1}.

4 Finding Mutually-Visible Grid Points at an Inflection

When there is inflection in the sequence of trapezoid directions, the first step is to find an inner common tangent line such as the line $\ell(Q'_tQ_t)$ in Figure 11. There is a unique inner common tangent line for each inflection. It has one point of tangency Q_t at a trapezoid vertex following the inflection trapezoid and another point of tangency Q'_t at a preceding trapezoid vertex. The common tangent line divides the region covered by the trapezoids near inflection into a forward region $\mathcal{R}$ and a backward region $\mathcal{R}'$. The object is to find a pair of mutually-visible grid points $P \in \mathcal{R}$ and $P' \in \mathcal{R}'$.

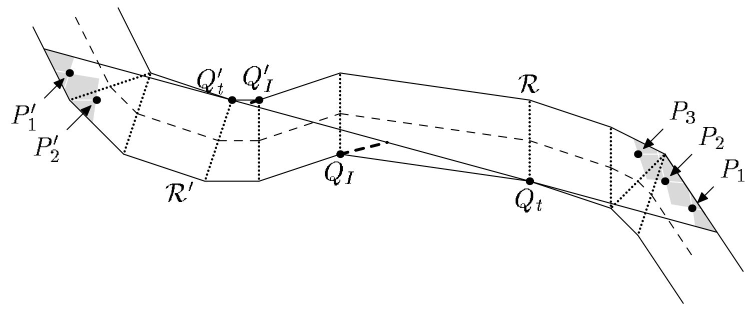

Figure 11: How to use the techniques of Section 3.3 to find a good pair of mutually visible points at an inflection. The $\ell(Q'_tQ_t)$ line is the inner common tangent, and the shaded regions are free of grid points.

Points in $P_i \in \mathcal{R}$ are ranked according to their forward visibility by finding for each P_i, a tangent line through that point and some successor of Q_t and rating P_i according to the direction of the tangent line as explained in Section 3. A similar ranking scheme rates points in $\mathcal{R}'$ according to their backward vision by finding tangent lines through some predecessor of Q'_t. In Figure 11, P_1 is the best grid point in $\mathcal{R}$ and P'_1 is the best grid point in $\mathcal{R}'$.

If the best grid point $P_1 \in \mathcal{R}$ can see the best grid point $P'_1 \in \mathcal{R}'$, then it is natural to consider segment P'_1P_1 to be the best way for a grid-restricted polygonal path to get past the inflection. Unfortunately, P_1 and P'_1 might not be able to see each other, in which case more work is required to find an optimal pair of mutually-visible grid points $P \in \mathcal{R}$ and $P' \in \mathcal{R}'$. There can then be a trade-off between the best forward visibility from P and the best backward visibility from P'. The purpose of this section is to see how to find at least one point on this trade-off.

In order to search for pairs of mutually-visible points, we need to know what can prevent a point $P \in \mathcal{R}$ from seeing a point $P' \in \mathcal{R}'$: segment $P'P$ could cross a trapezoid edge near Q'_t; or it could cross an edge near Q_t. The former case involves $E(Q'_t, Q'_I)$, where Q'_I is the last concave trapezoid vertex among the immediate successors of Q'_t, and function E gives the trapezoid edges between two specified vertices together with the edge into the first vertex and the edge out of the last vertex. The other way P can fail to see P' is for $P'P$ to cross $E(Q_I, Q_t)$, where Q_I is the first of the concave trapezoid vertices immediately preceding Q_t. It is not possible for $P'P$ to cross both the $E(Q'_t, Q'_I)$ obstacle and the $E(Q_I, Q_t)$ obstacle.

Figure 11 suggests using the techniques of Section 3.3 to find the best few grid points in $\mathcal{R}$, and then find a similar list of points P'_1, P'_2, $\ldots$ in $\mathcal{R}'$. In a situation similar to Figure 11, we might proceed as follows.[5]

First, find a grid point $P_1 \in \mathcal{R}$ with best forward visibility and a grid point $P_1' \in \mathcal{R}'$ with best backward visibility. Suppose $P_1'P_1$ crosses the $E(Q_t', Q_I')$ obstacle and $P_2'P_1$ crosses the $E(Q_I, Q_t)$ obstacle, where P_2' is chosen to have the best backward visibility among all grid points whose visibility around the $E(Q_t', Q_I')$ obstacle is sufficient to allow any hope of seeing P_1. Thus no grid point in $\Psi_{\mathcal{R}'}(P_2')$ can see P_1, where $\Psi_{\mathcal{R}'}()$ is the function that finds the region of $\mathcal{R}'$ where the backward visibility is at least as good as a given point. Since P_1 is the only grid point in the region $\Psi_{\mathcal{R}}(P_1)$ of points in $\mathcal{R}$ with forward visibility at least as good as P_1, no grid point in $\Psi_{\mathcal{R}'}(P_2')$ can see any grid point in $\Psi_{\mathcal{R}}(P_1)$.

The logical next step is to examine P_2, P_3, ... , looking for a grid point with enough visibility around the $E(Q_I, Q_t)$ obstacle to make it possible to see P_2'. Suppose P_2 does not satisfy this condition, but P_3 does. At this point P_2' and P_3 might be mutually visible, or there might be some other pair of mutually-visible points $P' \in \Psi_{\mathcal{R}'}(P_2')$ and $P \in \Psi_{\mathcal{R}}(P_3)$.

If the above procedures fail to find any mutually-visible grid points, we have a situation very similar that before the examination of P_2 and P_3, except that $\mathcal{R}$ and $\mathcal{R}'$ play opposite roles. The two situations are as follows, where k and k' are integer parameters to be determined later:

1. Either $k' = 0$ or the following hold: no grid point in $\Psi_{\mathcal{R}}(P_k)$ can see any grid point in $\Psi_{\mathcal{R}'}(P_{k'}')$; point P_k cannot see $P_{k'}'$ because of the $E(Q_t', Q_I')$ obstacle; and other grid points in $\Psi_{\mathcal{R}}(P_k)$ cannot see $P_{k'}'$ because of the $E(Q_I, Q_t)$ obstacle.
2. Either $k = 0$ or the following hold: no grid point in $\Psi_{\mathcal{R}}(P_k)$ can see any grid point in $\Psi_{\mathcal{R}'}(P_{k'}')$; point $P_{k'}'$ cannot see P_k because of the $E(Q_I, Q_t)$ obstacle; and other grid points in $\Psi_{\mathcal{R}'}(P_{k'}')$ cannot see P_k because of the $E(Q_t', Q_I')$ obstacle.

Before examining P_2 and P_3, we have Situation 2 with $k = 1$ and $k' = 2$; after searching $\Psi_{\mathcal{R}'}(P_2')$ and $\Psi_{\mathcal{R}}(P_3)$, we have Situation 1 with $k = 3$ and $k' = 2$. This suggests an algorithm where k and k' start at 0 or 1 and we alternate between Situations 1 and 2 while k and k' increase as we use Algorithm 3 to find more and more grid points in $\mathcal{R}$ and $\mathcal{R}'$. Algorithm 4 gives the details.

The algorithm works by letting regions $\Psi_{\mathcal{R}'}(P_{k'}')$ and $\Psi_{\mathcal{R}}(P_k)$ expand until reaching a pair of mutually visible grid points. It assumes that $\Psi_{\mathcal{R}'}(P_0')$ and $\Psi_{\mathcal{R}}(P_0)$ are defined to be the empty set. The following theorem is easily proved by using the invariants that Situation 1 holds at the start of Step 2 and Situation 2 holds at the start of Step 5.

Theorem 4.1 *Let P_k and $P_{k'}'$ be as explained above. Algorithm 4 finds mutually-visible points $\bar{P}' \in \Psi_{\mathcal{R}'}(P_{k'}')$ and $\bar{P} \in \Psi_{\mathcal{R}}(P_k)$ such that no other mutually-visible points $P' \in \Psi_{\mathcal{R}'}(P_{k'}')$ and $P \in \Psi_{\mathcal{R}}(P_k)$ can have P' better than $\bar{P}'$ and P better than $\bar{P}$.*

Before going on, we need to clarify Steps 3 and 6 of Algorithm 4. In Step 3, requiring that the $E(Q_I, Q_t)$ obstacle not block vision to P_{k-1} is equivalent to restricting to a half plane defined by the tangent line from P_{k-1} to the $E(Q_I, Q_t)$ obstacle. The hard part is searching for a grid point $\bar{P} \in \Psi_{\mathcal{R}}(P_k)$ that can see $\bar{P}'$.

The tangent lines from $\bar{P}'$ to the $E(Q_t', Q_I')$ and $E(Q_I, Q_t)$ obstacles define a cone in which $\bar{P}$ must lie in order to see $\bar{P}'$. Call this the *visibility cone* for $\bar{P}'$. Either the visibility cone contains some P_i, P_{i+1}, P_{i+2}, ... , P_{j-1} for $1 \le i < j \le k$ as shown in Figure 12, or it falls between P_{j-1} and P_j for some $j \le k$ as shown in Figure 13. In the former case, the best grid point visible from $\bar{P}'$ must be P_i because Theorem 3.1 guarantees that P_i is the best grid point in $\mathcal{R}$ whose visibility around the $E(Q_I, Q_t)$ obstacle exceeds that of P_{i-1}.

[5]Figure 11 has Points P_1, P_2, P_3 and P_1' repositioned to make the grid-point-free regions more visible. They would have to be much closer to the $\ell(Q_t'Q_t)$ line in order for the $E(Q_t', Q_I')$ and $E(Q_I, Q_t)$ obstacles to interfere with mutual visibility as supposed in the following discussion.

Algorithm 4 Find a pair of mutually-visible grid points $\bar{P}' \in \mathcal{R}'$ and $\bar{P} \in \mathcal{R}$ so as to try to maximize the backward visibility from $\bar{P}'$ and the forward visibility from $\bar{P}$.

```
1. Use Algorithm 3 to find the best grid point $P_1 \in \mathcal{R}$. Then set $k = 1$ and $k' = 0$.
2. Use Algorithm 3 to find the next best grid points $P'_{k'+1}$, $P'_{k'+2}$, $P'_{k'+3}$ in $\mathcal{R}'$ as explained in
   Section 3.3, stopping at the first $P'_{l'}$ for which $P'_{l'}P_k$ does not cross the $E(Q'_t, Q'_I)$ obstacle.
3. If $k > 1$, use Algorithm 1 and polygon scan-conversion to find grid points in the portion of
   $\Psi_{\mathcal{R}'}(P'_{l'}) \setminus \Psi_{\mathcal{R}'}(P'_{k'})$ for which the $E(Q_I, Q_t)$ obstacle does not block vision to $P_{k-1}$. Let $\bar{P}'$
   be the best such grid point that can see grid points in $\Psi_{\mathcal{R}}(P_k)$; let $\bar{P}$ be the best grid point in
   $\Psi_{\mathcal{R}}(P_k)$ that can see $\bar{P}'$. (If $\bar{P}'$ and/or $\bar{P}$ fail to exist, go on to Step 4.)
4. If Step 3 has found a pair $(\bar{P}', \bar{P})$, halt and return $(\bar{P}', \bar{P})$. If $P'_{l'}$ can see $P_k$, halt and return
   $(P'_{l'}, P_k)$. Otherwise set $k' = l'$.
5. Use Algorithm 3 to find the next best grid points $P_{k+1}$, $P_{k+2}$, $P_{k+3}$ in $\mathcal{R}$ as explained in
   Section 3.3, stopping at the first $P_l$ for which $P'_{k'}P_l$ does not cross the $E(Q_I, Q_t)$ obstacle.
6. If $k' > 1$, use Algorithm 1 and polygon scan-conversion to find grid points in the portion of
   $\Psi_{\mathcal{R}}(P_l) \setminus \Psi_{\mathcal{R}}(P_k)$ for which the $E(Q'_t, Q'_I)$ obstacle does not block vision to $P'_{k'-1}$. Let $\bar{P}$ be
   the best such grid point that can see grid points in $\Psi_{\mathcal{R}'}(P'_{k'})$; let $\bar{P}'$ be the best grid point in
   $\Psi_{\mathcal{R}'}(P'_{k'})$ that can see $\bar{P}$. (If $\bar{P}$ and/or $\bar{P}'$ fail to exist, go on to Step 7.)
7. If Step 6 has found a pair $(\bar{P}', \bar{P})$, halt and return $(\bar{P}', \bar{P})$. If $P_l$ can see $P'_{k'}$, halt and return
   $(P'_{k'}, P_l)$. Otherwise set $k = l$ and go to Step 2.
```

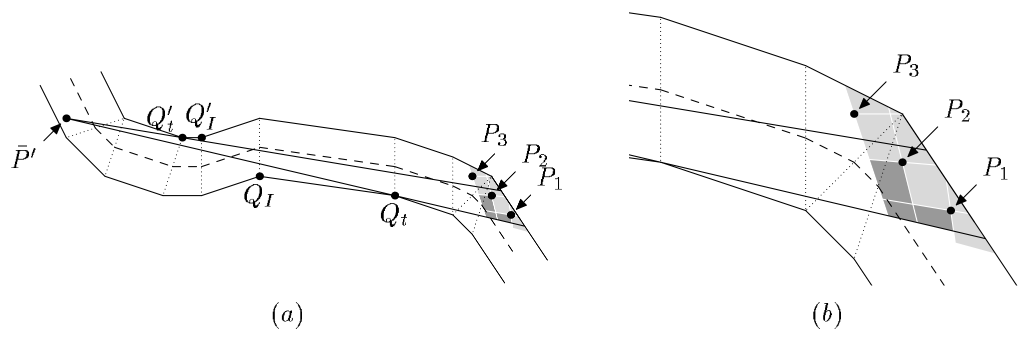

Figure 12: (a) The search area for $\bar{P}$ in Step 3 of Algorithm 4 (shaded) with grid-point-free regions shaded lightly; (b) A close-up showing that P_1 is the best grid point in the cone.

The other case is when the visibility cone for $\bar{P}'$ falls between P_{j-1} and P_j. There is then a convex quadrilateral that needs to be searched for grid points. In the example of Figure 13, $j = 2$ and the quadrilateral is the dark shaded region. In general, it is the intersection of the visibility cone with

$$\Psi_{\mathcal{R}}(P_k) \setminus \Psi_{\mathcal{R}}(P_j). \tag{8}$$

Algorithm 5 summarizes this process of finding the best grid point in $\Psi_{\mathcal{R}}(P_k)$ that can see $\bar{P}'$. An almost identical algorithm can be used to find the best grid point in $\Psi_{\mathcal{R}'}(P'_{k'})$ that can see $\bar{P}$ as required by Step 6 of Algorithm 4.

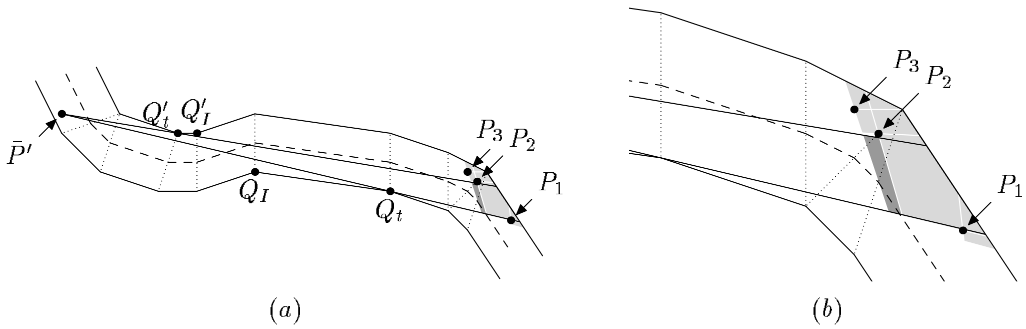

Figure 13: (a) The search area for $\bar{P}$ in Step 3 of Algorithm 4 (shaded) with grid-point-free regions shaded lightly; (b) A close-up showing how the visibility cone falls between P_1 and P_2.

Algorithm 5 Find the best grid point in $\Psi_{\mathcal{R}}(P_k)$ that can see $\bar{P}'$. This is needed to implement Step 3 of Algorithm 4.

1. Locate the visibility cone for $\bar{P}'$ relative to P_1, P_2, P_3, ... , P_k. Let i be the smallest index for which does not hit $\bar{P}'P_i$ the $E(Q_I, Q_t)$ obstacle, and let j be the smallest index for which $\bar{P}'P_j$ hits the $E(Q'_t, Q'_I)$ obstacle.
2. If $i < j$ return P_i. Otherwise, use Algorithm 1 and polygon scan-conversion to search for grid points in the intersection of (8) with the visibility cone and return the best such grid point.

The contribution of Algorithm 5 and its variant to the run time of Algorithm 4 depends on the distribution of k and l values during Steps 3 and 6 of the algorithm and on the number of grid points in the parallelograms from which apexes of the visibility cones are chosen. A theoretical analysis of these quantities would probably be uninformative since it is difficult to get tight upper bounds and Section 7.2 will show that they are small in practice.

The rest of the run time for Algorithm 4 is dominated by the calls to Algorithm 3 in Steps 1, 2 and 5. This is also difficult to bound theoretically, but the average number of calls to Algorithm 3 per invocation of Algorithm 4 typically ranges from 3 to 4.8 in practice. (See Section 7.2 for details.)

5 The Main Algorithm for Stage 2

The main algorithm for Stage 2 depends on handling inflections as explained in Section 4 and using the ideas of Section 3 to find minimum-vertex grid-restricted polygonal paths between inflections. The major complication is that inflections cannot always be considered in isolation. We say that common tangents $\ell(Q_{j-1}Q_j)$ and $\ell(Q_{j+1}Q_{j+2})$ *interfere* if they cross each other inside a trapezoid that is bounded by a segment of the path from Q_j to Q_{j+1} as shown in Figure 14.

In Figure 14, interfering tangent lines $\ell(Q_1Q_2)$ and $\ell(Q_3Q_4)$ cross each other to form the light shaded region and interfering tangent lines $\ell(Q_3Q_4)$ and $\ell(Q_5Q_6)$ delimit the dark shaded region. Considering the inflections in isolation would lead to an output path with two vertices per inflection, even though it would be better to find a polygonal path of the form $P_1P_2P_3P_4$ where P_1P_2 crosses $\ell(Q_1Q_2)$, P_2P_3 crosses $\ell(Q_3Q_4)$, and P_3P_4 crosses $\ell(Q_5Q_6)$. In other words, the goal is to find grid points $P_1 \in \mathcal{R}'$, P_2 in the light shaded region, P_3 in the dark region, and $P_4 \in \mathcal{R}$ so that P_1 has the best possible backward visibility, P_4 has the best possible forward visibility, and P_i can see P_{i+1} for $1 \leq i < 4$.

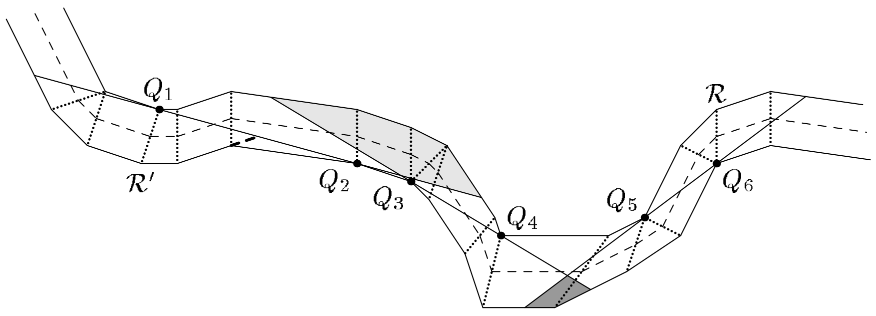

Figure 14: Regions to search when common tangents at successive inflections interfere.

A dynamic programming approach could probably be used in the case of Figure 14 to find the trade-off between backward visibility from $P_1 \in \mathcal{R}'$ and forward visibility from $P_4 \in \mathcal{R}$. This turns out not to be worthwhile because the simple greedy approach explained below performs almost as well in practice.

How should the greedy algorithm handle the situation shown in Figure 14? Start by using Algorithm 1 and polygon scan-conversion to find a grid point P_2 in the light shaded region near the intersection of $\ell(Q_1Q_2)$ and $\ell(Q_3Q_4)$. Then use the same algorithms to scan the dark shaded region for a grid point P_3 that can see P_2 and lies close to $\ell(Q_5Q_6)$. Finally, Algorithm 3 can be used to find grid points P_1 and P_4 such that P_4 can see P_3 and has the best possible forward visibility and P_1 can see P_2 and has optimal backward visibility.

In general, there could be any number of inflections where the ith common tangent line $\ell(Q_{2i-1}Q_{2i})$ interferes with its successor $\ell(Q_{2i+1}Q_{2i+2})$. This produces a whole series of regions like the shaded regions in Figure 14. The greedy algorithm handles this case by finding a grid point P_2 in the first such region, then finding P_{i+1} in the ith region for $i = 2, 3, 4, \ldots$. It is greedy in the sense that it never backtracks and it selects each point in a manner intended to maximize the chance of finding a visible grid point in the next region.

When searching for some P_{i+1}, the algorithm might find that no grid points in the appropriate region can see P_i. This can be handled by simply ignoring the fact that $\ell(Q_{2i-1}Q_{2i})$ interferes with $\ell(Q_{2i+1}Q_{2i+2})$. For instance, this happens for $i = 2$ in the case of Figure 14 if no grid point in the dark shaded region can see P_2. Ignoring the interference between $\ell(Q_3Q_4)$ and $\ell(Q_5Q_6)$ involves using Algorithm 3 to select the best grid point visible from P_2 and then using the techniques of Section 4 to find a pair of mutually-visible grid points that span the Q_5Q_6 inflection.

The overall algorithm begins by testing whether there are any inflections. If not, repeated calls to Algorithm 3 can generate a sequence of grid points, each visible from the previous one. If there are inflections, we examine all the interfering common tangents and find grid points as explained above. For inflections where there is no interference, Algorithm 4 can select grid points. The remaining task is to connect the chosen grid points with grid-restricted polygonal paths. This can be done by repeated calls to Algorithm 3 as in the inflection-free case.

The above ideas lead to Algorithm 6. It uses the term *interference region* to refer to convex regions such as the shaded regions in Figure 14 where the cone defined by the interfering tangent lines intersects the trapezoids. Function $\mathrm{inc}_k(i)$ is the function that returns $i+1$ if $i<k$ and 1 if $i=k$. The notation P_Λ refers to a dummy point that cannot occur in the input.

Algorithm 6 Find a grid-restricted polygonal path that passes through the sequence of trapezoids produced by Stage 1.

1. Let k be the number of inflections and find points $Q_1, Q_2, Q_3, \ldots, Q_{2k}$ that define the inner common tangents. Also initialize $M[i] = P_\Lambda$ for $i \le i \le k$.
2. If $k > 0$, go on to Step 3. Otherwise, let P_0 be any concave trapezoid vertex and use Algorithm 3 to find and output grid points $P_1, P_2, P_3, \ldots$, each visible from its predecessor. Halt after the first $i > 2$ for which P_i can see P_1.
3. Try to find the first l for which $\ell(Q_{2l-1}Q_{2l})$ does not interfere with $\ell(Q_{2l'-1}Q_{2l'})$, where $l' = \mathrm{inc}_k(l)$.
4. If there is such an l, set $\bar{P} = P_\Lambda$; otherwise set $l = k$, $l' = 1$, and use Algorithm 1 and polygon scan-conversion to find a grid point $\bar{P}$ in the interference region for $\ell(Q_{2k-1}Q_{2k})$ and $\ell(Q_1Q_2)$. Then set $M[k] = \bar{P}$ and $i = l'$.
5. Set $i' = \mathrm{inc}_k(i)$ and check whether $\ell(Q_{2i-1}, Q_{2i})$ and $\ell(Q_{2i'-1}, Q_{2i'})$ interfere. If so, use Algorithm 1 and polygon scan-conversion to find a grid point $\bar{P}'$ in the interference region that is as close to $\ell(Q_{2i'-1}, Q_{2i'})$ as possible and satisfies the following: either $\bar{P} = P_\Lambda$ or $\bar{P}'$ can see $\bar{P}$. If successful set $\bar{P} = \bar{P}'$ and $M[i] = \bar{P}'$; otherwise, set $\bar{P} = P_\Lambda$.
6. If $i' \ne l$, set $i = i'$ and go back to Step 5. Otherwise, set $i = 1$.
7. Let $i' = \mathrm{inc}_k(i)$. If $M[i'] = P_\Lambda$ and $M[i] \ne P_\Lambda$, use Algorithm 3 to make $M^-[i']$ the grid point visible from $M[i]$ with best forward visibility. If $M[i] = P_\Lambda$ and $M[i'] \ne P_\Lambda$, use Algorithm 3 to make $M^+[i]$ the grid point visible behind $M[i']$ with best backward visibility. If $M[i] = M[i'] = P_\Lambda$, use Algorithm 4 to set $M^+[i]$ and $M^-[i']$.
8. If $i \ne k$, set $i = i'$ and go back to Step 7. Otherwise, set $i = 1$.
9. If $M[i] \ne P_\Lambda$, output $M[i]$. Otherwise use Algorithm 3 to find and output a minimal sequence of grid points starting at $M^-[i]$ and ending at $M^+[i]$ where each point is visible from its predecessor.
10. If $i \ne k$, set $i = i + 1$ and go back to Step 9. Otherwise, halt.

Step 2 handles the case where there are no inflections; Steps 4–6 scan for interfering inflections and set up an array M whose ith entry gives a suitable grid point in the interference region that follows the ith inflection. Null values in this array mean that the final output will have at least two vertices between the inflections i and $\mathrm{inc}_k(i)$. Next, Steps 7 and 8 choose output vertices before and after each inflection and set up arrays M^- and M^+ to store output vertices not already in the M array. Finally, Steps 9 and 10 output the vertices stored in the M, M^- and M^+ arrays, adding intermediate vertices if necessary.

In Step 4, the scan for a grid point $\bar{P}$ should be designed to favor points close to where $\ell(Q_{2k-1}Q_{2k})$ and $\ell(Q_1Q_2)$ intersect. Step 5 also uses Algorithm 1 to scan a polygon for grid points. It is best to precompute the tangent lines that delimit the cone where $\bar{P}$ is visible so that the interference region can be intersected with the cone before starting Algorithm 1 and doing the scan-conversion.

Note that Algorithm 6 invokes Algorithms 1 and 4 at most once per inflection, and it invokes Algorithm 3 at most once per output vertex. Section 3.1 explains that the time for Algorithm 1 is likely to be dominated by (4) in practice. Sections 3.2 and 4 explain that worst case bounds for Algorithms 3 and 4 would not be very useful, but that they take nearly constant time in practice.

Adding up all these contributions and letting $k = 1$ in (4), it seems reasonable to expect the overall running time to be linear in the number of trapezoids produced by Stage 1. Since [7] gives a linear time bound for Stage 1 this gives linear time overall. The results in Section 7.3 will support the claim that the running time is linear in practice.

6 Refinements to the Trapezoid Sequence

A problem with the interface between Stage 1 and Stage 2 is that the error tolerance is playing two roles: in Stage 1, it allows for noise introduced during the printing and scanning process; while Stage 2 uses the error tolerance to decide how much the outlines can be altered in order to achieve simplicity and compactness. The purpose of this section is to provide separate control over these two types of error.

The idea behind Stage 1 error tolerance is that a relatively smooth underlying shape gives rise to jagged outlines, and then the algorithm attempts to reconstruct the original shape by assuming it has the minimum number of inflections allowed by the error tolerance. For example, there is an underlying ampersand shape that gave rise to Figure 1a, and the output of Stage 1 in Figure 1b is much closer to that underlying shape. We call this approximation *Stage 1 midline approximation* because it is formed by taking the midline through each trapezoid as shown in Figure 2.

There needs to be a secondary error tolerance ϵ_2 that limits how far the output of Stage 2 can deviate from the Stage 1 midline approximation. This tolerance applies to the ∞-norm distance d_∞ between parallel lines. Imposing it requires modifying the trapezoid sequence before running Stage 2. The new trapezoid sequence should be based on lines $\ell_1^R, \ell_2^R, \ldots,$ and $\ell_1^L, \ell_2^L, \ldots,$ chosen so that

$$\begin{aligned} d_\infty(\ell_i^R, \ell(R_iR_{i+1})) &= d_\infty(\ell_i^L, \ell(L_iL_{i+1})) \\ d_\infty(\ell_i^R, \ell_i^L) &= \min\bigl(2\epsilon_2,\, d_\infty(\ell(R_iR_{i+1}), \ell(L_iL_{i+1}))\bigr), \end{aligned} \tag{9}$$

where $\ell(AB)$ is the directed line containing segment AB. Figure 15 illustrates how this can be done. Replacing trapezoids $R_1R_2L_2L_1$, $R_2R_3L_3L_2$, and $R_3R_4L_4L_3$ with modified versions involves replacing the solid lines with the heavy dashed lines so as to eliminate the shaded parts of the of the original trapezoids.

Trapezoids like $R_1R_2L_2L_1$ in Figure 15 are little changed because $\ell(R_1R_2)$ and $\ell(L_2L_1)$ are within ∞-norm distance ϵ_2 of the midline approximation. When this happens, parts of neighboring trapezoids can be within ∞-norm distance ϵ_2 of the midline approximation even though they otherwise would not be. For instance, the lightly shaded region in the figure is close to the $R_1R_2L_2L_1$ midline but outside of the tolerance for $R_2R_3L_3L_2$.

6.1 The Simple Inflection-Free Case

Let us see how to construct a refined trapezoid sequence in a simple case. (Since the trapezoids are allowed to degenerate to triangles, we must write $R_{i+1} - R_i + L_{i+1} - L_i$ for the direction of the

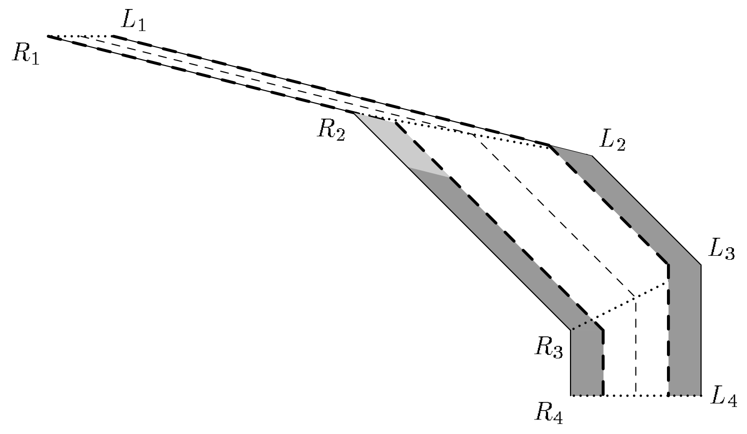

Figure 15: Part of a trapezoid sequence with regions that violate the secondary tolerance ϵ_2 shaded. The heavy dashed lines are at ∞-norm distance ϵ_2 from the midline approximation (light dashed line).

parallel edges of the ith trapezoid.) Suppose we can choose j and k so that the direction sequence

$$R_{i+1} - R_i + L_{i+1} - L_i, \text{ for } i = j-1,\ j,\ j+1, \ldots, k \tag{10}$$

has no inflections and is confined to 180° sector. Thus as i increases, the directions always turn left or always turn right. Either way L_{j-1}, L_j, L_{j+1}, ..., L_{k+1} and R_{j-1}, R_j, R_{j+1}, ..., R_{k+1} define the boundaries of two convex regions. Extending segments $L_{j-1}L_j$, $R_{j-1}R_j$, L_kL_{k+1}, and R_kR_{k+1} to infinity produces semi-infinite regions $\mathcal{L}_{jk}$ and $\mathcal{R}_{jk}$, one of which is contained in the other. Region $\mathcal{L}_{jk}$ is the intersection of half planes bounded by directed lines $\ell(L_iL_{i+1})$ for $j-1 \le i \le k$, and $\mathcal{R}_{jk}$ is the intersection of similar half planes bounded by $\ell(R_iR_{i+1})$ for $j-1 \le i \le k$.

The natural way to construct a refined trapezoid sequence is to find directed lines

$$\ell^R_{j-1},\ \ell^R_j, \ldots, \ell^R_k \tag{11}$$

and

$$\ell^L_{j-1},\ \ell^L_j, \ldots, \ell^L_k \tag{12}$$

satisfying (9), and use them to define half planes that intersect to form semi-infinite regions $\mathcal{R}'_{jk}$ and $\mathcal{L}'_{jk}$ analogous to $\mathcal{R}_{jk}$ and $\mathcal{L}_{jk}$. The only problem is that there might not be a grid-restricted path in the difference between $\mathcal{R}_{jk}$ and $\mathcal{L}_{jk}$ because crucial grid points might lie in excluded regions such as the lightly-shaded parallelogram in Figure 15. Hence we need to alter $\mathcal{R}'_{jk}$ and/or $\mathcal{L}'_{jk}$ so that their difference includes appropriate grid points.

If as in Figure 15, the directions (10) turn right as i increases, $\mathcal{R}'_{jk}$ is contained in $\mathcal{L}'_{jk}$ and the modified trapezoid sequence will have a left boundary based on $\mathcal{L}'_{jk}$ and a right boundary based on the convex hull of the grid points in $\mathcal{R}'_{jk}$. The alternative when (10) turns right is to have the right boundary based on $\mathcal{R}'_{jk}$ and the left boundary based on the convex hull of the grid points in $\mathcal{L}'_{jk}$.

It is clear how to take a sequence of directed lines (11) or (12), construct the convex hull, and output the vertices in order. This gives the boundary of $\mathcal{L}'_{jk}$, but finding the inner boundary requires a special algorithm that computes the convex hull of the grid points in $\mathcal{R}'_{jk}$ or $\mathcal{L}'_{jk}$. A routine of this type appears in [9], but it is useful to have an alternative based on the algorithms presented in previous sections.

This requires a subroutine that takes directed lines $\bar{\ell}$ and ℓ_i and a grid point P on $\bar{\ell}$, and finds the boundary of the convex hull of the set of grid points to the right of both directed lines. Call this the *grid hull subroutine.* Figure 16 gives an example where grid points are marked by dots and the desired convex hull boundary shown as a heavy line.

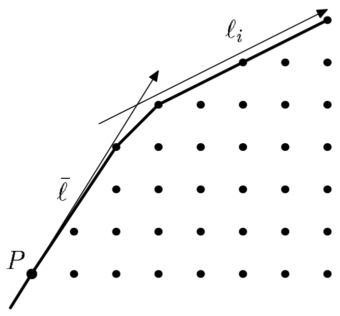

Figure 16: Directed lines $\bar{\ell}$ and ℓ_i and the convex hull of the grid points in the region to the right of both lines. The directed lines are shown as arrows and the convex hull boundary is marked by a heavy line.

Algorithm 3 can be used to implement the grid hull subroutine if it is given special input data structures and the algorithm is modified to output each new value of P_c as a convex hull vertex. Use P as Q_0; use ℓ_i as both the near line and the far path; and place Q_1 anywhere so that $\ell(Q_0Q_1)$ is parallel to $\bar{\ell}$. This makes P_1^{opt} the point where $\bar{\ell}$ and ℓ_i cross. These inputs guarantee that Step 4 will never need to do a search.

Theorem 6.1 *If the grid hull subroutine is implemented as described above, it successfully computes the convex hull of the grid points in the region bounded by $\bar{\ell}$ and ℓ_i.*

Proof. Refer to Corollary A.2 in Appendix A. □

To find the convex hull of the grid points $\mathcal{R}'_{jk}$, we just apply the grid hull subroutine to each pair of lines from (11) and intersect the resulting convex regions. (The requirement for a grid point P on one of the lines $\bar{\ell}$ is no problem since all the lines we are dealing with have rational directions so it is easy to find a grid point as close to $\bar{\ell}$ as possible.) Since we have seen that finding $\mathcal{L}'_{jk}$ is easy, the entire refinement process for the right-turning, inflection-free case can be summarized by Algorithm 7. If the directions (10) turn left as i increases, an identical algorithm works, except that ℓ_i^R and ℓ_i^L have to play opposite roles and we need left halfplanes instead of right halfplanes.

Except for the time spent in the grid hull subroutine, Algorithm 7 takes time linear in the number of input plus output trapezoids. This is because each step can consider the lines in order according to their direction angles, and no backtracking is necessary except when removing segments that turn out not to be part of the desired boundary. As for the grid hull subroutine, Section 3.2 argued that this algorithm is almost linear in practice. If it is almost linear in the size of its output, the same holds for Algorithm 7.

6.2 Finishing the Refined Trapezoid Sequence

In order to extend the techniques of the previous section to handle the entire trapezoid sequence, we have to cope with inflections and deal with the restriction that the direction sequence (10) is confined to a 180° sector. The overall approach is to break up the sequence of input trapezoids at

Algorithm 7 Given a trapezoid sequence $R_iR_{i+1}L_{i+1}L_i$ for $j-1 \le i \le k$, and a tolerance ϵ_2, construct an appropriate refined trapezoid sequence. The directions (10) are assumed to turn right as i increases and remain confined to a 180° sector.

1. Find directed lines $\ell^R_{j-1}, \ell^R_j, \ldots, \ell^R_k$ and $\ell^L_{j-1}, \ell^L_j, \ldots, \ell^L_k$ satisfying (9).
2. Find the boundary of the region $\mathcal{L}'_{jk}$ formed by intersecting $H(\ell^L_{j-1}), H(\ell^L_j), \ldots, H(\ell^L_k)$, where $H(\ell^L_i)$ is the half plane to the right of ℓ^L_j.
3. Use $1+k-j$ calls to the grid hull subroutine to find the convex hull of the grid points inside of $H(\ell^R_{i-1}) \cap H(\ell^R_i)$ for $j \le i \le k$. Then find the boundary of the region $\mathcal{R}'_{jk}$ that is formed by intersecting all these convex hulls.
4. Treat the boundaries of $\mathcal{L}'_{jk}$ and $\mathcal{R}'_{jk}$ as ordered lists of directed segments, and consider interleaving the lists according to their direction angles. Insert null segments whenever one list skips over a segment direction found in the other list. The resulting one-to-one correspondence defines the desired trapezoid sequence.

inflections, run Algorithm 7 on the resulting inflection-free subsequences, and carefully merge the results.

First, consider the 180° restriction. We have already noted that each step of Algorithm 7 can consider the segments in order and do only limited backtracking. Such an implementation can be applied to any inflection-free trapezoid sequence, even though the resulting actions may be hard to describe in terms of operations on convex regions if the 180° restriction is violated. Rather than burdening the reader with detailed descriptions of how and why this works, we will just assume that Algorithm 7 "does the right thing" in the case of an inflection-free trapezoid subsequence that violates the 180° restriction. (The Guibas-Ramshaw-Stolfi theory of polygonal tracings provides useful background information [5].)

What about inflections? In the trapezoid sequence of Figure 17a, subdividing at $R_6R_7L_7L_6$ yields inflection-free subsequences

$$R_1R_2L_2L_1, \ldots, R_6R_7L_7L_6 \quad \text{and} \quad R_6R_7L_7L_6, \ldots, R_{11}R_{12}L_{12}L_{11}.$$

Using Algorithm 7 to refine each subsequence gives Figure 17b. The remaining task is to merge separately refined subsequences such as those in Figure 17b.

Figure 18 illustrates the merging process. Figure 18a is based on the subsequence that ended at $R_iR_{i+1}L_{i+1}L_i$. As explained in Section 6.1, this trapezoid has been extended to infinity in the direction implied by the arrows. The heavy dashed lines are the boundaries of the refined trapezoids.

Figure 18b is analogous to Figure 18a, except it shows the initial segments of the result of refining the subsequence starting at $R_iR_{i+1}L_{i+1}L_i$; i.e., Figure 18a shows what happens before reaching the inflection and Figure 18b shows what happens after passing the inflection. The idea behind the merging process is to find a smooth way of connecting the heavy dashed lines in Figure 18a to the corresponding ones in Figure 18b. This can be done by finding inner common tangents: the new left-side boundary follows the heavy dashed boundaries of the lightly-shaded regions and passes between them along their common tangent; the new right side boundary passes between the darker regions along their common tangent.

First, we run Algorithm 7 on the sequence of trapezoids ending at $R_iR_{i+1}L_{i+1}L_i$ and let B^-_L and B^-_R are the left- and right-side boundaries of the refined trapezoid sequences. Then running Algorithm 7 on the trapezoid sequence that starts at $R_iR_{i+1}L_{i+1}L_i$ produces left- and right-side boundaries B^+_L and B^+_R. The original trapezoids define a one-to-one correspondences between the

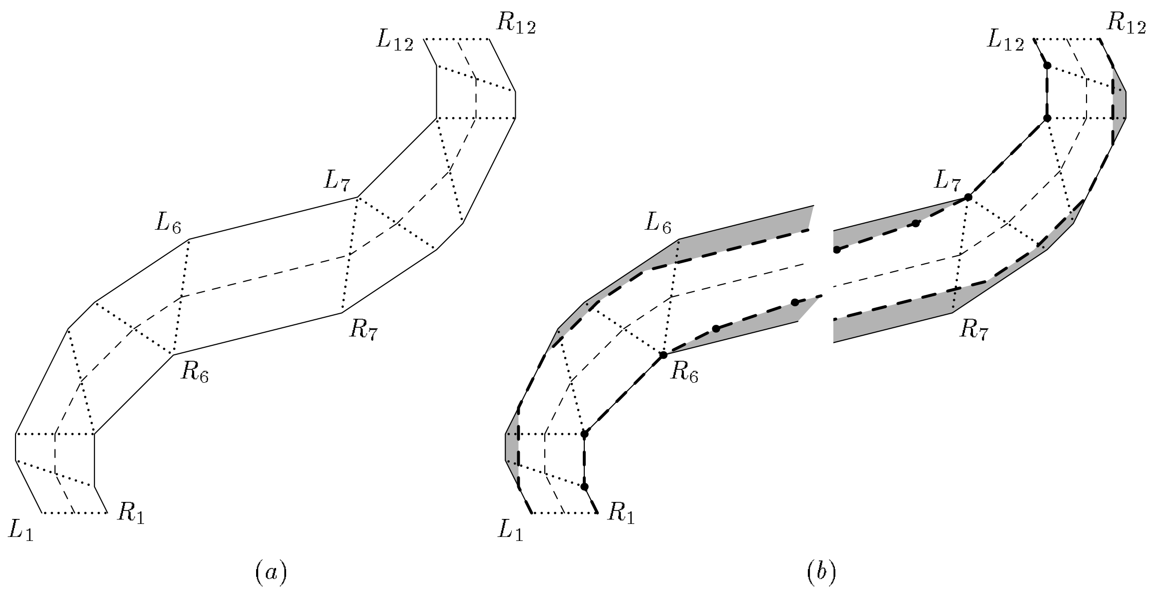

Figure 17: (a) Part of a trapezoid sequence as produced by Stage 1; (b) The result of splitting at $R_6R_7L_7L_6$ and computing separate refinements for each half. Heavy dashed lines show the boundaries of the refined trapezoids and the shaded areas show the regions removed by the refinement process.

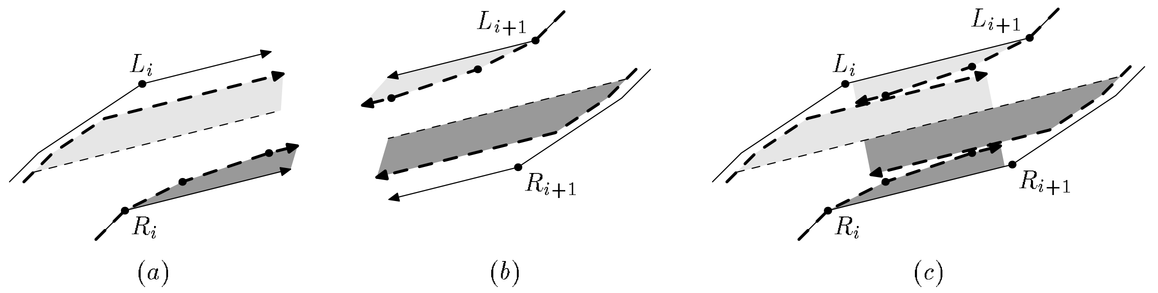

Figure 18: (a) The last trapezoids from separately refining a subsequence ending at $R_iR_{i+1}L_{i+1}L_i$; (b) the first trapezoids resulting from separate refinement starting at $R_iR_{i+1}L_{i+1}L_i$; (c) the two refined subsequences just prior to merging.

segments of B_L^- and B_R^- and between the segments of B_L^+ and B_R^+, where each segment is paired with the (possibly degenerate) parallel segment from the other side of its trapezoid. We must restore this correspondence after using the common tangents to join B_L^- to B_L^+ and B_R^- to B_R^+. This completes Algorithm 8.

Algorithm 8 Merge trapezoid sequences created by breaking at an inflection and refining separately. Polygonal paths B_L^- and B_R^- are the left- and right-side boundaries for the refined trapezoid sequence preceding the inflection, and B_L^+ and B_R^+ are the left- and right-side boundaries for the trapezoids after the inflection.

1. Scan backward from the end of B_L^- and forward from the beginning of B_L^+ until finding a common tangent V_1V_2. Then replace everything after V_1 in B_L^- and everything before V_2 in B_L^+ with the segment V_1V_2.
2. Scan backward from the end of B_R^- and forward from the beginning of B_R^+ until finding a common tangent V_3V_4. Then replace everything after V_3 in B_R^- and everything before V_4 in B_R^+ with the segment V_3V_4.
3. The following segments are now unpaired: V_1V_2, V_3V_4, and the former partners of the segments removed to make room for V_1V_2 and V_3V_4. Pair them up by inserting null segments as necessary. Make sure that the resulting sequence of directions has only one inflection.

6.3 Choosing the Secondary Tolerance

Sections 6.1 and 6.2 explain how to do refinement on inflection-free subsequences of the trapezoid sequence and then merge them together to create a refined version of the original data. The purpose of the refinement is to force the output of Stage 2 to stay within a specified tolerance of the Stage 1 midline approximation. Equation (9) limits the ∞-norm deviation to ϵ_2, but this is subsequently relaxed by up to 1 grid unit so as to ensure the existence of an appropriate grid-restricted polygonal approximation:

Theorem 6.2 *If a refined trapezoid sequence is computed by Algorithms 7 and 8, the result will have all concave trapezoid vertices at grid points. Hence there will be a grid-restricted polygonal path through the refined trapezoids.*

Proof. Section 6.1 introduces concave trapezoid vertices only via Algorithm 7, and that algorithm forces all vertices to be at grid points. Algorithm 8 does not create any new concave trapezoid vertices. Thus as explained in Section 2, the minimum-perimeter path through the trapezoids is the grid-restricted polygonal path required by the theorem. □

Even though any value of ϵ_2 results in a usable trapezoid sequence, some values may be better than others in practice. Since one grid unit can be important, we should try to choose ϵ_2 so that the region $\mathcal{R}'_{jk}$ defined in Section 6.1 is not likely to differ much from the convex hull of the grid points it contains. In other words, the lines ℓ_i^R and ℓ_i^L should pass through grid points when positioned according to (9). This can be done by running Stage 1 with a grid two times coarser than the target grid and choosing ϵ_2 to be a multiple of the target grid spacing. This makes the trapezoid edges from Stage 1 lie on lines that pass through the course grid so that their bisectors will pass through points on the fine grid. Making ϵ_2 a multiple of the fine grid spacing then forces ℓ_i^R and ℓ_i^L to pass through grid points.

7 Experimental Results

The Stage 1 and Stage 2 algorithms have been implemented in C++ and tested on binary images of pages of scanned text from a standard database of test documents [18]. The images were converted into outlines using the naive algorithm, then Stage 1 converted the outlines into trapezoid sequences and Stage 2 finished the job. Section 7.1 analyzes the storage space and image quality for the resulting grid-restricted outlines; Section 7.2 discusses which parts of Stage 2 are important in practice and which are not; and Section 7.3 gives execution times.

7.1 The Trade-Off between Space-Efficiency and Image Quality

How well does Stage 2 achieve its goal of producing space-efficient outlines without compromising too much on image quality? To answer this, we need ways of measuring space efficiency, and we need to try out various choices of the grid spacing and tolerance parameters and examine images created from the resulting outlines.

Table 1 gives statistics that show how the algorithms perform on a fairly typical sample image.[6] The table shows that Stage 2 reduced the vertex count from the Stage 1 midline approximation by a factor of 1.4 to 2.3, depending on the Stage 1 error tolerance, the secondary tolerance ϵ_2 and the grid spacing. It achieved this result while simultaneously imposing grid constraints that the Stage 1 midline approximation does not satisfy.

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
grid spacing	1	0.5	0.5	0.33	0.25	0.25	0.25	0.25
Stage 1 tolerance	1	1	1	0.67	0.75	0.75	0.75	0.5
secondary tolerance	–	–	0.5	0.33	–	0.5	0.25	0.25
input vertices	39438	40261	40261	45303	45444	45444	45444	43827
output vertices	17777	17417	19662	26265	21500	22508	28016	32401
lower bound	16961	16989	19106	25174	20929	21906	27344	31759
outline bytes	29100	33111	35661	47269	43578	44859	51965	57872
CCITT-g4 bytes	47072	47072	47072	47072	47072	47072	47072	47072

Table 1: Overall statistics for the Stage 2 algorithm on input from a 300 dpi binary image of a page of text from [18]. Columns *A*-*H* give statistics for various settings of the grid spacing, the Stage 1 error tolerance, and the Section 6 secondary tolerance.

Table 1 lists a lower bound on the vertex count. This is the result of the Stage 2 algorithm in the limiting case where the grid spacing approaches zero. In other words, it is the result of ignoring the grid constraints. This greatly simplifies the Stage 2 algorithm and produces a situation where the greedy approach of Algorithm 6 is clearly optimal. (The resulting algorithm is similar to that of Goodrich [4].) Since the vertex counts in Table 1 are all within 4.8% of the lower bound, they must be close to the best possible. This stands in sharp contrast to the theoretical algorithms from [9] where imposing grid constraints increases the vertex count by a logarithmic factor.

In order to get a realistic estimate of the space required to store the outlines, a simple binary file format was developed. In this format, each outline is encoded as a pair of starting coordinates followed by a vertex count and a list of $(\Delta x, \Delta y)$ pairs. All numbers are encoded using a scheme where small numbers require 4 to 8 bits, and roughly two additional bits are required each time the magnitude doubles. Table 1 lists the number bytes for this scheme in the "outline bytes" row.

[6]The table is based on journal page image a002 from [18]. It is a fairly clean image scanned at 300 dots per inch. It includes text and mathematical formulas, but no drawings or halftone images.

It ranges from 62% to 123% of the size of the compressed binary image file that served as input to Stage 1. This input was in TIFF format with the best compression scheme readily available for binary images: CCITT Group 4 facsimile compression. A simple bitmap would have required more than a megabyte.

The grid spacing and tolerance values in Table 1 represent a trade-off between image quality and the compactness of the outline representation. The table shows how the outline byte count grows when we reduce the Stage 1 tolerance, the secondary tolerance, or the grid spacing, and Figures 19b–d show how reducing the tolerances improves image quality. On the other hand, reducing the Stage 1 tolerance below 0.5 pixels would prevent the "jaggies" in Figure 19a from being eliminated.

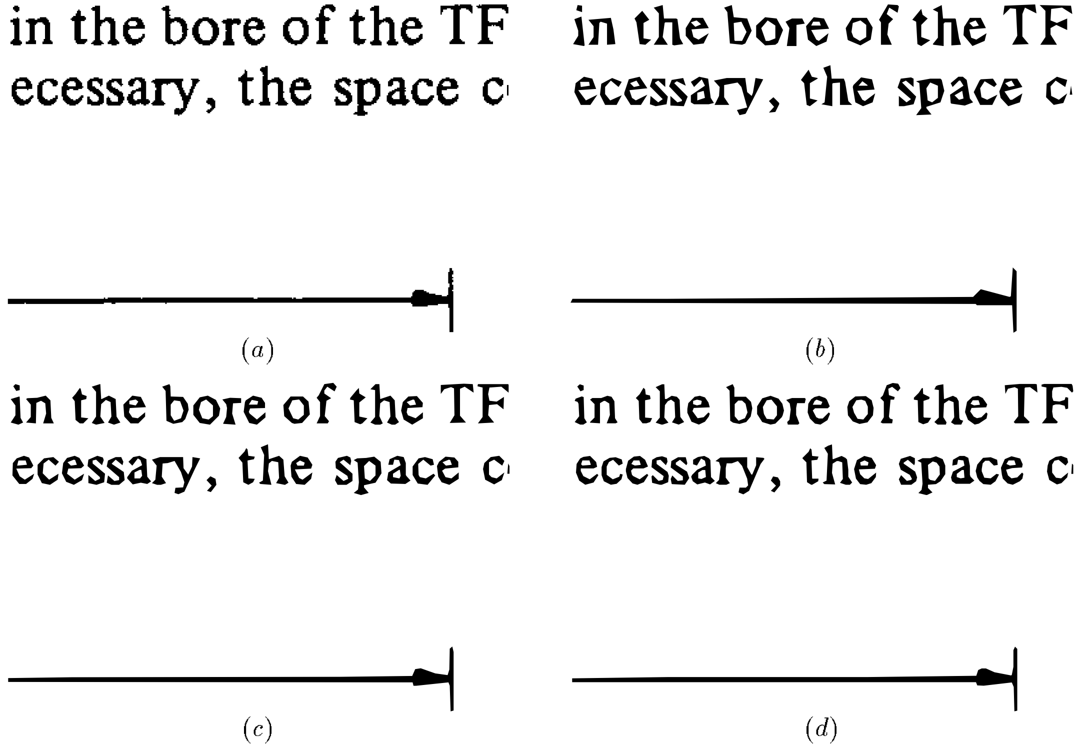

Figure 19: A magnified portion of test document `h047` reproduced by various methods. (a) pixel replication from the raw image; (b) from outlines generated with grid spacing and tolerances as in column C of Table 1; (c–d) the same for columns D and H, respectively.

Table 1 shows that the outline byte count generally compares favorably to TIFF image files with CCITT-g4 compression when the Stage 1 tolerance is 1 pixel and the grid spacing and secondary tolerance are each $\frac{1}{2}$ pixel. Smaller tolerances would increase the ratio of outline bytes to bytes in the TIFF files as suggested by Table 1. Refer to Figure 20 for a full accounting of how this compression ratio depends on the complexity of the test pages with and without halftone pictures. Since the database from [18] has separate flags for the presence of drawings and halftone images, Figure 20 makes a similar distinction. The algorithm does particularly well on simple line drawings, but drawings containing shaded areas are a difficult case.

We can conclude that Stage 2 does a good job of minimizing the vertex count, but the overall space requirements are harder to judge. For a typical page image at 300 dots per inch in CCITT-g4 format, the space required for the outlines ranges from a little less to a little more depending on the quality desired. If we want the outlines to be more useful than the original bitmap image

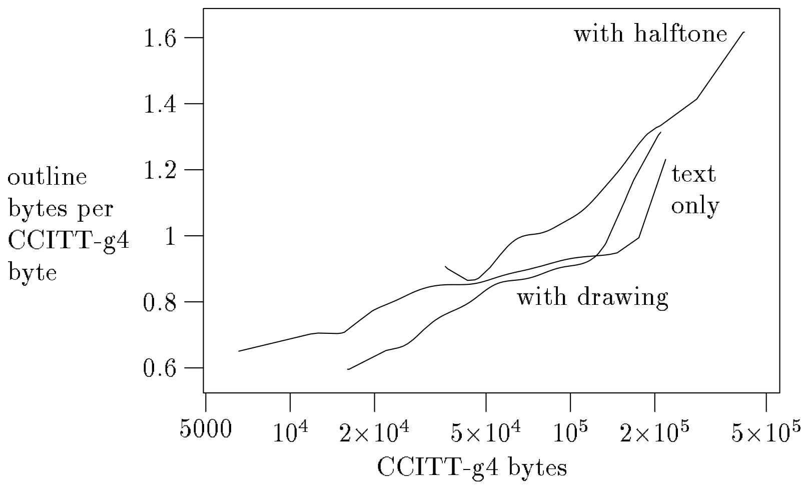

Figure 20: Average ratio of outline bytes to length of the CCITT-g4-compressed TIFF file as a function of the TIFF file size.

as suggested by the motivation in Section 1, we should probably choose the quality depicted in Figure 19d. This entails a space penalty of about 23% relative to the CCITT-g4 bitmaps.

7.2 Appropriateness of Implementation Choices

Sections 3.2, 4 and 5 made compromises designed to keep the algorithms as simple as possible by assuming that certain special cases are rare in practice. Let us see which parts of Stage 2 are most important in practice and how the simplifying assumptions are born out by the statistics in Table 2.

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
grid spacing	1	0.5	0.5	0.33	0.25	0.25	0.25	0.25
Stage 1 tolerance	1	1	1	0.67	0.75	0.75	0.75	0.5
secondary tolerance	–	–	0.5	0.33	–	0.5	0.25	0.25
input vertices	39438	40261	40261	45303	45444	45444	45444	43827
output vertices	17777	17417	19662	26265	21500	22508	28016	32401
lower bound	16961	16989	19106	25174	20929	21906	27344	31759
inflections	3461	3466	3466	5691	5233	5233	5233	6325
interfering inflections	1607	1628	1362	2306	2394	2272	1880	2260
M entries in Alg. 6	1600	1620	1360	2291	2370	2254	1867	2259
Alg. 3 invocations	17194	17167	19477	25830	23951	24063	28742	31478
number from Alg. 4	7553	7874	8199	11678	13541	12761	12319	12522
restarts	1218	532	837	1024	554	654	698	418
polygon searches	513	423	408	368	610	546	400	46
visibility cone searches	14	14	7	20	39	12	10	20

Table 2: Statistics for the detailed behavior of the Stage 2 algorithm on the input image used by Table 1. Columns *A*-*H* give statistics for various settings of the grid spacing, the Stage 1 error tolerance, and the Section 6 secondary tolerance.

Section 3.2 simplified Algorithm 3 by designing it so that it has to be started over from scratch with a new Q_0 vertex if it fails to find a grid point. This is relatively harmless in the tabulated cases because the number of such restarts was never more than 7.1% of the total invocations.

Another compromise in Algorithm 3 is to use a relatively crude procedure in Step 4 when searching polygons such as the one shown in Figure 8a. This is also safe because the number of polygon searches was never more than 3% of the total invocations for Algorithm 3.

The last major compromise is designing Algorithm 6 to use a greedy approach rather than the dynamic programming process alluded to at the beginning of Section 5. Since Algorithm 6 creates one M entry each time it is able to use a single grid point between a pair of interfering inflections, the close relationship between M entries and interfering inflections means that the algorithm seldom uses two vertices where one might do. Hence dynamic programming could not help very much at all.

Now consider the relative importance of Algorithm 4. It accounts for 40% to 57% of the Algorithm 3 invocations that are listed in Table 2. The number calls to Algorithm 4 is essentially the number of non-interfering inflections. More precisely, it is the number of inflections minus the number of M entries in Algorithm 6. This is $3461 - 1600 = 1861$ in Column A and it ranges up to $6325 - 2259 = 4066$ in Column H. Dividing this into the number of times Algorithm 4 invoked Algorithm 3, we find that the number of Algorithm 3 invocations per invocation of Algorithm 4 ranges from 3.1 for Column H to 4.8 for Column E.

Another potentially time-consuming step in Algorithm 4 is the visibility cone searches used in Steps 3 and 6 of Algorithm 4 and defined in Algorithm 5. Since Table 2 shows that these searches are rare, the real inner loop for Stage 2 is Algorithm 3.

7.3 Execution Speed

Testing the whole process on all 979 journal page images from [18] produced the timing and data compression statistics in Table 3. All of the test images had the same scanning resolution and similar page dimensions, but some pages were much more complicated than others. Hence, all the statistics were normalized by dividing by the number of input vertices v_I produced by the Stage 1 algorithm. This number tends to be high for images with large dark smears or halftone pictures. The table attempts to list statistics for such images separately, since they are particularly difficult cases for outline-based algorithms.

The run time was consistently proportional to v_I, except that it increased slightly when v_I was high or halftone pictures were present. Stage 2 was approximately twice as expensive as Stage 1 and combining Stage 2 with the refinement process from Section 6 increased this to a factor of 3. These ratios may be somewhat higher than necessary because no effort has been made to optimize the Stage 2 implementation.

8 Conclusion

While most of the algorithms presented here are not extremely complicated, their implementations do add up to about 5600 lines of C++, not including Stage 1. In spite of this, the results in Section 7 show that the algorithm is fast enough to be quite practical. There are numerous potential applications where it useful to extract good outlines from image data and represent the outlines compactly. The two-stage approach allows separate control over the Stage 1 noise tolerance and the output grid and auxiliary error tolerance from Section 6. By using the trapezoid sequence data from Stage 1, it avoids complications that Guibas, et. al. [6] encountered in deciding what order the output path hits the input data points.

	without halftones				with halftones		
input vertices v_I	<20k	20-60k	60-200k	>200k	<60k	60-200k	>200k
pages	24	227	608	8	16	84	12
output vertices $/v_I$	0.489	0.487	0.492	0.522	0.502	0.510	0.526
lower bound $/v_I$	0.475	0.473	0.479	0.516	0.491	0.499	0.521
Stage 1 μsec$/v_I$	68	64	63	47	62	56	45
refinement μsec$/v_I$	65	68	69	81	73	75	87
Stage 2 μsec$/v_I$	136	135	136	153	136	144	151
total μsec$/v_I$	290	286	288	307	294	300	311
outline bytes $/v_I$	0.893	0.867	0.876	0.978	0.924	0.924	0.974
CCITT-g4 bytes $/v_I$	1.398	1.051	0.959	0.717	1.054	0.882	0.672

Table 3: Statistics for test runs on journal page images from [18]. All runs used tolerance 1 pixel, grid spacing and secondary tolerance $\frac{1}{2}$ pixel. Images were classified according to the number of input vertices v_I and whether or not they include halftone figures. (This information appears in "page attribute files" that come with the test images.) Timings were made on a 150 megahertz MIPS R4400 processor and normalized by dividing by v_I. The total time includes some overhead due to data structure conversions in the interface between Stages 1 and 2.

Atts.
Appendix A
References

Appendices

A Correctness of the Best Visible Grid Point Algorithm

Theorem A.1 *Algorithm 3 finds the best grid point in the region bounded by the near line and the far path, where "best" is measured in terms of the tangent line direction as described at the beginning of Section 3.*

Proof. Note that points on the ℓ_{lim} line are equally "good" in terms of the tangent line direction ordering. By testing against this line, the algorithm explicitly checks the ordering among points found in Step 4 and between such points and the value of P_c in Step 7.

The argument that this final P_c value is the best grid point is based on excluding grid points from potentially better regions such as the shaded areas in Figures 21a and 21b. The P_c increments in successive iterations of Step 5 define a polygonal line

$$P_c^{0,m} = P_c^0 P_c^1 P_c^2 \cdots P_c^m,$$

for some m. The segments of this line are directed along the D vectors chosen in successive iterations of Step 2. Since Step 7 proceeds only if *test_pts*() rejected $D'' + D$ in Step 5, the subsequent call to *test_pts*() in Step 2 will truncate (7) before reaching the old D. Thus the segment directions along $P_c^{0,m}$ move monotonically away from the $P_1^{\text{opt}} - Q_0$ direction.

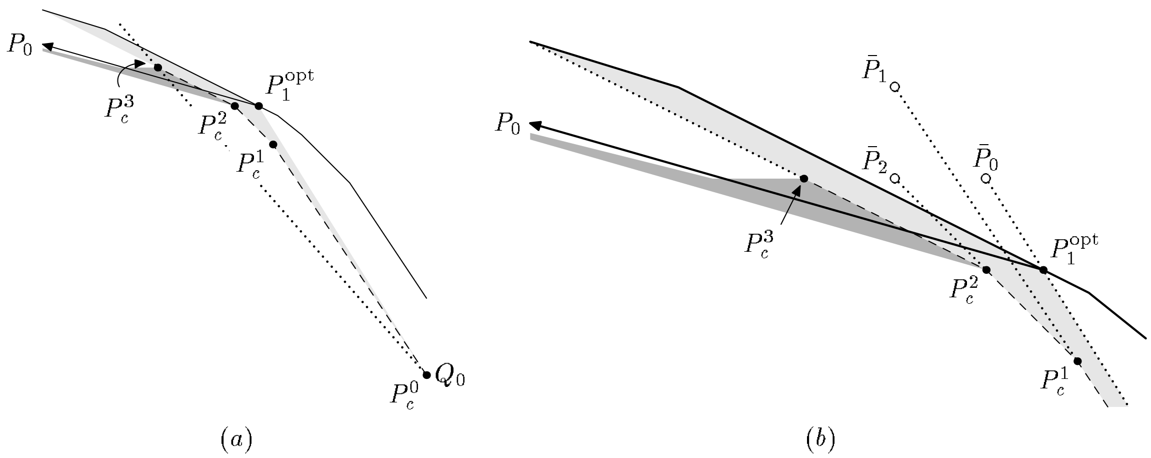

Figure 21: (a) The successive P_c values from Algorithm 3 (dashed line), with the grid-point-free region shaded. (b) A close-up of the region near P_c^3.

Let $\bar{P}_0$ be the grid point closest to P_c^0 along ray through P_1^{opt}, and let $\bar{P}_j$ be the result of adding D to the new P_c value P_c^j in Step 5. For $j < m$, all $\bar{P}_j$ points are grid points rejected by *test_pts*() as past the far path. (These points are marked by open circles in Figure 21b.) For any $j < m$, consider the cone bounded by rays through $\bar{P}_j$ and $\bar{P}_{j+1}$ from the apex P_c^j. We can use the sequence of CF-neighbor directions between $\bar{P}_j - P_c^j$ and $\bar{P}_{j+1} - P_c^{j+1}$ to divide the cone into sectors by making additional rays through the apex P_c^j along each of the new directions. If $\bar{D}_{j,k-1}$ and $\bar{D}_{j,k}$ are a two of these CF-neighbor directions, either $\bar{D}_{j,k} = \bar{P}_{j+1} - P_c^{j+1}$ or *test_pts*() has determined that $P_c^j + \bar{D}_{j,k-1}$ and $P_c^j + \bar{D}_{j,k}$ are both across the far path. In the former case, Step 4 scans the sector for grid points; in the latter case, all grid points in the sector must be across the far path. Hence we can conclude that all cones $\bar{P}_j P_c^j \bar{P}_{j+1}$ contain no interior grid points, except those considered in Step 4.

We have shown that the lightly-shaded region in Figures 21a and 21b is free of unconsidered grid points. More precisely, if $\bar{P}_c^{0,m}$ is the result of extending the last segment of $P_c^{0,m}$ until it hits the far path, then the closed region bounded by $Q_0P_1^{\text{opt}}$, the far path, and $\bar{P}_c^{0,m}$ contains no grid points except those on $\bar{P}_c^{0,m}$ which are considered in Step 4. Thus, the only grid points across the near line and better than P_c^m are those between $Q_0P_c^m$ (the dotted line in Figure 21a), $P_c^{0,m}$ (the dashed line), and the near line (the ray from P_1^{opt} directed toward P_0). Call this region R.

The darker region in Figures 21a and 21b corresponds to Figure 6a. It is a union of triangles of the form

$$P_c^{m-1} \quad P_c^{m-1}{+}D_j \quad P_c^{m-1}{+}D_{j+1}. \tag{13}$$

These contain no interior grid points, because such a grid point would have to be a weighted average of P_c^{m-1}, $P_c^{m-1}+D_{j,k-1}$, and $P_c^{m-1}+D_{j,k}$, where $D_{j,k-1}$ and $D_{j,k}$ are CF-neighbors between D_j and D_{j+1}. Since P_c^{m-1} must be short of the near line and the last triangle (13) had D_{j+1} equal to the near line direction, the triangles must cover the entire region R.

Hence if the algorithm returns P_c^m, there are no grid points between the near line and the far path better than P_c^m. The only other possibility is to return the best P_b point, but this is only done when a comparison against ℓ_{lim} shows that P_b is better than P_c^m. $\square$

The following corollary is simply a restatement of some the intermediate results in the above proof. It is needed to prove that Section 6.1 implements the grid hull subroutine correctly.

Corollary A.2 *Let $\bar{P}_c^{0,m}$ be the polygonal line determined by the successive P_c values computed by Algorithm 3 as in the proof of Theorem A.1. If no grid points are found in Step 4, then all grid points in the closed region bounded by segment $Q_0P_1^{\text{opt}}$, the far path, and $\bar{P}_c^{0,m}$ lie on $\bar{P}_c^{0,m}$. Furthermore, the vertices of $\bar{P}_c^{0,m}$ lie on grid points, and $\bar{P}_c^{0,m}$ has no inflection segments.*

References

[1] O. E. Agazzi, K. W. Church, and W. A. Gale. Using OCR and equalization to downsample documents. In *Proceedings of the 12th International Conference on Pattern Recognition*, pages 305–309, Jerusalem, Israel, October 1994.

[2] James George Dunham. Optimum uniform piecewise linear approximation of planar curves. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(1):67–75, January 1986.

[3] Touradj Ebrahimi, Homer Chen, and Barry G. Haskell. Joint motion estimation and segmentation for very low bitrate video coding. AT&T Bell Laboratories technical memorandum, 1994.

[4] Michael T. Goodrich. Efficient piecewise-linear function approximation using the uniform metric. In *Proceedings of the Tenth Annual Symposium on Computational Geometry*, pages 322–331, June 1994.

[5] L. Guibas, L. Ramshaw, and J. Stolfi. A kinetic framework for computational geometry. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, pages 100–111, 1983.

[6] Leonidas J. Guibas, John E. Hershberger, Joseph S. B. Mitchell, and Jack Scott Snoeyink. Approximating polygons and subdivisions with minimum link paths. In W. L. Hsu and R. C. T. Lee, editors, *ISA '91 Algorithms*, pages 151–162. Springer-Verlag, 1991. Lecture Notes in Computer Science 557.

[7] John D. Hobby. Polygonal approximations that minimize the number of inflections. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 93–102, January 1993.

[8] Hiroshi Imai and Masao Iri. Computational-geometric methods for polygonal approximation of a curve. *Computer Vision Graphics and Image Processing*, 36(1):31–41, October 1986.

[9] Simon Kahan and Jack Snoeyink. On the bit complexity of minimum link paths: Superquadratic algorithms for problems solvable in linear time. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry*, pages 151–158, May 1996.

[10] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, Massachusetts, 1981.

[11] Yoshisuke Kurozumi and Wayne A. Davis. Polygonal approximation by the minimax method. *Computer Graphics and Image Processing*, 19(3):248–264, July 1982.

[12] Maylor K. Leung and Yee-Hong Yang. Dynamic strip algorithm in curve fitting. *Computer Vision Graphics and Image Processing*, 51(2):145–165, August 1990.

[13] U. Montanari. A note on minimal length polygonal approximation to a digitized contour. *Communications of the ACM*, 13(1):41–47, January 1970.

[14] L. O'Gorman. Image and document processing techniques for the RightPages electronic library system. In *International Conference on Pattern Recognition (ICPR)*, pages 260–263, The Hague, September 1992.

[15] Theodosios Pavlidis. Polygonal approximations by Newton's method. *IEEE Transactions on Computers*, C-26(8):801–807, August 1977.

[16] Theodosios Pavlidis. *Structural Pattern Recognition*, pages 11–45,147–184. Springer Series in Electrophysics. Springer Verlag, 1977.

[17] Theodosios Pavlidis. *Algorithms for Graphics and Image Processing*, section 12.5. Computer Science Press, Rockville, Maryland, 1982.

[18] Ihsin T. Phillips, Su Chen, and Robert M. Haralick. CD-ROM document database standard. In *Proceedings of the International Conference on Document Analysis and Recognition*, pages 478–483, October 1993.

[19] Arie Pikaz and Its'hak Dinstein. Optimal polygonal approximation of digital curves. *Pattern Recognition*, 28(3):373–379, March 1995.

[20] Urs Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing*, 1(3):244–256, November 1972.

[21] Bimal Kumar Ray and Kumar S Ray. Determination of optimal polygon from digital curve using L_1 norm. *Pattern Recognition*, 26(4):505–509, April 1993.

[22] Christian Schwarz, Jürgen Teich, and Emo Welzl. On finding a minimal enclosing parallelogram. Technical Report TR–94–036, International Computer Science Institute, Berkeley, California, August 1994.

[23] Yazid M Sharaiha and Nicos Christofides. An optimal algorithm for the straight segment approximation of digital arcs. *CVGIP: Graphical Models and Image Processing*, 55(5):397–407, September 1993.

[24] Jack Sklansky. Recognition of convex blobs. *Pattern Recognition*, 2(1):3–10, January 1970.

[25] Jack Sklansky, Robert L. Chazin, and Bruce J. Hansen. Minimum perimeter polygons of digitized silhouetts. *IEEE Transactions on Computers*, C-21(3):260–268, March 1972.

[26] Jack Sklansky and Victor Gonzalez. Fast polygonal approximation of digitized contours. *Pattern Recognition*, 12(5):327–331, 1980.

[27] Jack Sklansky and Dennis F. Kibler. A theory of nonuniformity in digitized binary pictures. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-6(9):637–647, September 1976.

[28] Ivan Tomek. Two algorithms for piecewise-linear continuous approximation of functions of one variable. *IEEE Transactions on Computers*, 23(4):445–448, April 1974.

[29] Karin Wall and Per-Erik Danielsson. A fast sequential method for polygonal approximation of digitized curves. *Computer Vision Graphics and Image Processing*, 28(2):220–227, November 1984.

[30] Charles M. Williams. Bounded straight-line approximation of digitized planar curves and lines. *Computer Graphics and Image Processing*, 16(4):370–381, August 1981.